

RealView® Compilation Tools

バージョン 3.0

アセンブラガイド

ARM®

RealView Compilation Tools

アセンブラーガイド

Copyright © 2002-2006 ARM Limited. All rights reserved.

リリース情報

本書には以下の変更が追加されています。

変更履歴

日付	発行	機密保持ステータス	変更
2002年8月	A	非機密扱い	第1.2版
2003年1月	B	非機密扱い	第2.0版
2003年9月	C	非機密扱い	RVDS v2.0 リリース（第2.0.1版）
2004年1月	D	非機密扱い	RVDS v2.1 リリース（第2.1版）
2004年12月	E	非機密扱い	RVDS v2.2 リリース（第2.2版）
2005年5月	F	非機密扱い	RVDS v2.2 SP1 リリース（第2.2版）
2006年3月	G	非機密扱い	RVDS v3.0 リリース（第3.0版）

著作権

* または™のマークが付いた言葉およびロゴは、ARM Limited が所有する登録商標または商標です。本書に記載されている他の製品名は、各社の所有する商標です。

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

本書に記載されている製品は、今後も継続的に開発・改良の対象となります。本書に含まれる製品およびその利用方法についての情報は、ARM が利用者の利益のために提供するものです。したがって当社では、製品の市販性または利用の適切性を含め、暗示的・明示的に関係なく一切の責任を負いません。

本書は、本製品の利用者をサポートすることだけを目的としています。本書に記載されている情報の使用、情報の誤りまたは省略、あるいは本製品の誤使用によって発生したいかなる損失・損傷についても、ARM Limited は一切責任を負いません。

機密保持ステータス

本書は非機密扱いであり、本書を使用、複製、および開示する権利は、ARM および ARM が本書を提供した当事者との間で締結した契約の条項に基づいたライセンスの制限により異なります。

製品ステータス

本書の情報は最終版であり、開発済み製品に対応しています。

Web アドレス

<http://www.arm.com>

目次

RealView Compilation Tools アセンブラーガイド

序章

本書について	viii
フィードバック	xii

第 1 章

はじめに

1.1 RealView Compilation Tools アセンブラーについて	1-2
---	-----

第 2 章

ARM アセンブリ言語の記述

2.1 はじめに	2-2
2.2 ARM アーキテクチャの概要	2-3
2.3 アセンブリ言語モジュールの構造	2-11
2.4 条件実行	2-17
2.5 レジスタへの定数のロード	2-23
2.6 レジスタへのアドレスのロード	2-31
2.7 多重レジスタロード / ストア命令	2-37
2.8 マクロの使用	2-43
2.9 シンボルバージョンの追加	2-46
2.10 フレームディレクティブの使用	2-47
2.11 アセンブリ言語に関する変更	2-48

第 3 章**アセンブラーに関する参考情報**

3.1	コマンド構文	3-2
3.2	ソース行の形式	3-17
3.3	定義済みのレジスタおよびコプロセッサ名	3-18
3.4	組み込み変数および定数	3-19
3.5	シンボル	3-21
3.6	式、リテラル、演算子	3-27
3.7	診断メッセージ	3-40
3.8	C プリプロセッサを使用する	3-41

第 4 章**ARM 命令と Thumb 命令**

4.1	命令の概要	4-2
4.2	メモリアクセス命令	4-8
4.3	汎用データ処理命令	4-47
4.4	乗算命令	4-78
4.5	サチュレート命令	4-99
4.6	並列命令	4-104
4.7	パック命令と展開命令	4-112
4.8	分岐命令	4-120
4.9	コプロセッサ命令	4-126
4.10	その他の命令	4-136
4.11	擬似命令	4-155

第 5 章**NEON と VFP プログラミング**

5.1	NEON / VFP レジスタバンク	5-8
5.2	条件コード	5-10
5.3	一般的な情報	5-12
5.4	NEON と VFP に共通の命令	5-17
5.5	NEON 論理演算と比較演算	5-24
5.6	NEON 汎用データ処理命令	5-32
5.7	NEON シフト命令	5-44
5.8	NEON 汎用算術命令	5-51
5.9	NEON 乗算命令	5-64
5.10	NEON 要素と構造体のロード / ストア命令	5-69
5.11	NEON 擬似命令	5-77
5.12	ベクタ浮動小数点コプロセッサ	5-81
5.13	VFP レジスタ	5-82
5.14	VFP ベクタ演算とスカラ演算	5-85
5.15	VFP / NEON システムレジスタ	5-87
5.16	ゼロクリアモード	5-91
5.17	VFP 命令	5-93
5.18	VFP 擬似命令	5-108
5.19	VFP ディレクティブとベクタ表記	5-109

第 6 章**ワイヤレス MMX テクノロジの命令**

6.1	はじめに	6-2
6.2	ワイヤレス MMX テクノロジに対する ARM のサポート	6-3
6.3	命令の概要	6-7

第 7 章**ディレクティブリファレンス**

7.1	ディレクティブの一覧（アルファベット順）	7-2
7.2	シンボル定義ディレクティブ	7-4
7.3	データ定義ディレクティブ	7-17
7.4	アセンブリ制御ディレクティブ	7-32
7.5	フレームディレクティブ	7-41
7.6	通知ディレクティブ	7-56
7.7	命令セットと構文選択のディレクティブ	7-61
7.8	その他のディレクティブ	7-63

序章

本章では、*RealView Compilation Tools* アセンブラーガイド（本書）について概説します。
本章は以下のセクションから構成されています。

- 本書について (P. viii)
- フィードバック (P. xi)

本書について

本書では、*RealView® Compilation Tools* (RVCT) アセンブラーに関するチュートリアル情報および参考情報を提供しています。独立型アセンブラー `armasm` および C コンパイラや C++ コンパイラのインラインアセンブラーについて説明しています。また、アセンブラーに指定できるコマンドラインオプション、アセンブリ言語プログラミングに使用できるアセンブリ言語ニーモニック、擬似命令、マクロ、ディレクティブ、および ARM®、Thumb®-2、Thumb、およびベクタ浮動小数点 (VFP) 命令セットについても説明します。

対象読者

本書は、RVCT を使用してアプリケーションを作成している開発者を対象としています。本書の内容は、*RealView Compilation Tools v3.0 Essentials Guide* で説明されている ARM 開発ツールを熟知した経験豊富なソフトウェア開発者を対象に書かれています。

本書の構成

本書は以下の章から構成されています。

第 1 章 はじめに

RVCT アセンブラーおよびアセンブリ言語の概要を説明します。

第 2 章 ARM アセンブリ言語の記述

ARM アセンブラーおよびアセンブリ言語を使用する際のチュートリアル情報を提供しています。

第 3 章 アセンブラーに関する参考情報

ARM アセンブラーで使用できる言語の構文と体系について詳しく説明します。

第 4 章 ARM 命令と Thumb 命令

ARM 命令セットと Thumb 命令セット (Thumb-2 と以前の Thumb、および Thumb-2EE を含む) について詳しく説明します。

第 5 章 NEON と VFP プログラミング

ARM NEON™ Technology と VFP 命令セットについて詳しく説明します。また、VFP 固有のアセンブリ言語に関する情報も提供しています。

第 6 章 ワイヤレス MMX テクノロジの命令

ARM のワイヤレス MMX™ テクノロジのサポートについて詳しく説明します。

第7章 ディレクティブリファレンス

ARM アセンブラー `armasm` で利用できるアセンブラディレクティブについて詳しく説明します。

本書では、ARM ソフトウェアがデフォルトの場所にインストールされていることを前提としています。例えば、Windows 環境では、デフォルトの場所は `volume:\Program Files\ARM` になります。パス名を参照する際、`install_directory` の部分をこの場所に読み替えて下さい。例えば、本書では、`install_directory\Documentation\...` のようなパス名が使用されます。ARM ソフトウェアを別の場所にインストールした場合は、ファイルパスの見方を変える必要があります。

表記規則

本書では以下の表記規則を使用しています。

monospace コマンド、ファイル名、プログラム名、ソースコードなど、キーボードから入力可能なテキストを示しています。

monospace コマンドまたはオプションに使用可能な略語を示します。コマンド名またはオプション名をすべて入力する代わりに、下線部分の文字だけを入力することができます。

monospace italic

コマンドまたは関数の引数で、特定の値に置き換えることが可能なものを示しています。

monospace bold

サンプルコード以外に使用される言語キーワードを示しています。

italic 重要事項、重要用語、相互参照、引用箇所を斜体で記載しています。

bold メニュー名などのユーザインターフェース要素を太字で記載しています。また、適宜記述リスト内の重要箇所と ARM プロセッサの信号名にも太字を用いています。

参考資料

ここでは、ARM プロセッサファミリのコード開発に関する補足情報を記載した ARM Limited および各社の出版物を紹介します。

ARM は自社出版物の定期的な更新・修正を行っています。最新の正誤表、追補表、ARM に関する FAQ については、<http://www.arm.com> をご覧下さい。

ARM の出版物

本書では、RVCT 付属の開発ツールの参考情報を提供しています。このほか、本製品には以下のマニュアルが同梱されています。

- *RealView Compilation Tools v3.0 基本操作ガイド* (ARM DUI 0202J)
- *RealView Compilation Tools v3.0 コンパイラ / ライブラリガイド* (ARM DUI 0205J)
- *RealView Compilation Tools v3.0 リンカ / ユーティリティガイド* (ARM DUI 0206J)
- *RealView Compilation Tools v3.0 デベロッパガイド* (ARM DUI 0203J)
- *RealView Development Suite 用語集* (ARM DUI 0324J)

base standard、ソフトウェアインターフェース、および ARM でサポートされている標準に関する詳細については、*install_directory\Documentation\Specifications\...* を参照して下さい。

特定の ARM 製品に関する情報については、以下のマニュアルを参照して下さい。

- *ARM アーキテクチャリファレンスマニュアル* (ARM DUI 0100J-00)
- *ARM Architecture Reference Manual Thumb-2 Supplement* (ARM DDI 0308)
- *ARM Architecture Reference Manual Security Extensions Supplement* (ARM DDI 0309)
- *ARM Architecture Reference Manual Thumb-2 Execution Environment Supplement* (ARM DDI 0397)
- *ARM Architecture Reference Manual Advanced SIMD Extension and VFPv3 Supplement* (ARM DDI 0268)
- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- お使いのハードウェアデバイスの ARM データシートまたはテクニカルリファレンスマニュアル

他の出版物

ARM アーキテクチャに関する一般的な情報については、以下の出版物を参照して下さい。Steve Furber, *ARM System-on-Chip Architecture* (2nd edition, 2000), Addison Wesley, ISBN 0-201-67519-6。

インテル® ワイヤレス MMX™ テクノロジの詳細については、*Wireless MMX Technology Developer Guide* (August, 2000、Order Number: 251793-001) を参照して下さい。このマニュアルは <http://www.intel.com> から入手できます。

フィードバック

ARM Limited では、RealView Compilation Tools および本書に関するフィードバックをお待ちしております。

RealView Compilation Tools に関するフィードバック

RVCT に関して問題がある場合は、購入元にお問い合わせ下さい。このとき、迅速かつ適切な対応をさせて頂くために、以下の情報をご用意下さい。

- お名前と会社名
- 製品のシリアル番号
- 製品のリリース情報
- プラットフォームの詳細（ハードウェアプラットフォーム、オペレーティングシステムの種類とバージョンなど）
- 問題を再現するサイズの小さな独立したサンプルコード
- 操作の目的と実際の動作に関する詳しい説明
- 使用したコマンド（コマンドラインオプションを含む）
- 問題を再現できるサンプル出力
- ツールのバージョン情報（バージョン番号、ビルド番号を含む）

本書に関するフィードバック

本書に関するご意見につきましては、以下の内容を記載した電子メールを errata@arm.com までお送り下さい。

- マニュアル名
- 文書番号
- 問題のあるページ番号
- 問題点の簡潔な説明

補足すべき点や改善すべき点についてのご提案もお待ちしております。

第1章

はじめに

本章では、*RealView® Compilation Tools* (RVCT) 付属のアセンブラーについて概説します。本章は以下のセクションから構成されています。

- *RealView Compilation Tools* アセンブラーについて (P. 1-2)

1.1 RealView Compilation Tools アセンブラーについて

RVCT では、以下のアセンブラーが提供されます。

- 独立型アセンブラー `armasm` (このアセンブラーについては本書で説明します)
- C コンパイラおよび C++ コンパイラに組み込まれている最適化インラインアセンブラーと非最適化組み込みアセンブラー (これらのアセンブラーについては本書では説明していません)

これらのアセンブラーへの入力に使用する言語は基本的に同じですが、インラインアセンブラーや組み込みアセンブラーを使用している場合は、記述できるアセンブリ言語コードに対して制限があります。インラインアセンブラーと組み込みアセンブラーの詳細については、*RealView Compilation Tools v3.0 Developer Guide* の「C、C++、およびアセンブリ言語の混用」を参照して下さい。

以前のリリースから RVCT にアップグレードしている場合は、*RealView Compilation Tools v3.0 Essentials Guide* を読んで、このリリースの新機能と拡張機能について確認して下さい。

1.1.1 ARM アセンブリ言語

RVCT v2.2 以降のアセンブラーでは、アセンブリ言語が大幅に強化されましたが、従来のコードをアセンブルできるように引き続き古い構文もサポートします。

Thumb-2 は、ARM が提供している最新のプロセッサで利用できる Thumb 命令のアップグレード版です。Thumb-2 では、ARM 命令セットのほとんどの機能が利用できます。古い Thumb アセンブリ言語では、すべての新しい命令を記述することができないため、新しいアセンブリ言語に取って代わられました。

ARM または Thumb-2 用にアセンブルできるソースコードを記述することができます。

また、新しい ARM アセンブリ言語を使用して、Thumb-2 以前のプロセッサ用のThumb コードを記述することもできます。このような場合は、コードの記述対象となるプロセッサで利用できる命令のみを使用するように注意する必要があります。本書では、この作業に必要な情報を提供しています。利用できない命令を使用すると、アセンブラーからエラーが返されます。

1.1.2 ワイヤレス MMX テクノロジの命令

アセンブラーでは、インテル® ワイヤレス MMX™ テクノロジ命令をアセンブルし、PXA270 プロセッサ用のコードを開発することがサポートされます。このプロセッサは、MMX 拡張を採用した ARMv5TE アーキテクチャを実装しています。RVCT は、ワイヤレス MMX テクノロジ制御と単一命令複数データ処理 (SIMD) レジスタをサポートしており、ワイヤレス MMX テクノロジによる開発のための新しいディレクティブが導入されています。また、ロード命令とストア命令のサポートも強化されました。RVCT でのワイヤレス MMX テクノロジのサポートに関する詳細については、「第 6 章 ワイヤレス MMX テクノロジの命令」を参照して下さい。

1.1.3 NEON Technology

ARM NEON™ Technology は、ARMv7 アーキテクチャのオプションコンポーネントです。高性能のメディアアプリケーション、信号処理アプリケーション、および組み込みプロセッサを対象にした、64 ビットと 128 ビットのハイブリッド SIMD テクノロジです。ARM コアの一部として実装されますが、独自の実行パイプラインと、ARM コアのレジスタバンクとは別のレジスタバンクがあります。

NEON では、整数、固定小数点、および単精度浮動小数点の SIMD 演算がサポートされます。これらの命令は、ARM と Thumb-2 の両方で使用できます。

NEON の詳細については、「*第 5 章 NEON と VFP プログラミング*」を参照して下さい。

1.1.4 サンプルの使用方法

本書では、RealView Development Suite に収録されているサンプルを参照します。これらは、主なサンプルディレクトリの *install_directory\RVDS\Examples* にあります。収録されているサンプルの概要については、*RealView Development Suite* スタートガイドを参照して下さい。

第 2 章

ARM アセンブリ言語の記述

本章では、ARM® アセンブリ言語を記述する際の一般原則を説明します。本章は以下のセクションから構成されています。

- はじめに (P. 2-2)
- ARM アーキテクチャの概要 (P. 2-3)
- アセンブリ言語モジュールの構造 (P. 2-11)
- 条件実行 (P. 2-17)
- レジスタへの定数のロード (P. 2-23)
- レジスタへのアドレスのロード (P. 2-31)
- 多重レジスタロード/ストア命令 (P. 2-37)
- マクロの使用 (P. 2-43)
- シンボルバージョンの追加 (P. 2-46)
- フレームディレクティブの使用 (P. 2-47)
- アセンブリ言語に関する変更 (P. 2-48)

2.1 はじめに

本章では、ARM アセンブリ言語モジュールの基本的かつ実践的な記述方法を説明します。また、ARM アセンブラー (`armasm`) の機能についても説明します。

本章では、ARM、Thumb®-2、Thumb、NEON™、および VFP の各命令セットについては詳しく説明していません。これらの命令セットについては、以下を参照して下さい。

- 第4章 *ARM 命令と Thumb 命令*
- 第5章 *NEON と VFP プログラミング*

さらに詳しい情報については、*ARM アーキテクチャリファレンスマニュアル*と *ARM Architecture Reference Manual Thumb-2 Supplement* を参照して下さい。

既に ARM および Thumb アセンブリ言語について理解しているプログラマのために、本章には、ARM アセンブリ言語の最新バージョンと ARM および Thumb アセンブリ言語の以前のバージョンとの違いについて説明しているセクション（「アセンブリ言語に関する変更」(P. 2-48)）が含まれています。

2.1.1 サンプルコード

本章では、いくつかのサンプルコードを紹介しています。これらの多くは、`install_directory\RVDS\Examples\...\asm` ディレクトリに収録されています。

アセンブリ言語ファイルのビルドとリンクを行うには、以下の手順を実行します。

1. コマンドプロンプトで `armasm --debug filename.s` と入力し、ファイルをアセンブルして、デバッグテーブルを生成します。
2. `armlink filename.o -o filename` と入力し、オブジェクトファイルをリンクして、ELF 実行可能イメージを生成します。

このイメージの実行とデバッグを行うには、*RealView 命令セットシミュレータ(RVISS)*などの適切なデバッグターゲットを使用して、*RealView Debugger* や *ARM eXtended Debugger* (AXD) などの互換性のあるデバッガにこのイメージをロードします。

アセンブラーによるソースコードの変換動作を確認するには、以下のように入力します。

```
fromelf -c filename.o
```

`armlink` と `fromelf` の詳細については、*RealView Compilation Tools v3.0 Linker and Utilities Guide* を参照して下さい。

2.2 ARM アーキテクチャの概要

このセクションでは、ARM アーキテクチャについて簡単に説明します。

ARM プロセッサは、ロード / ストアアーキテクチャを実装する典型的な RISC プロセッサです。メモリにアクセスできるのは、ロード命令とストア命令だけです。データ処理命令は、レジスタの内容に対してのみ演算を行います。

このセクションでは、以下の内容について説明します。

- アーキテクチャのバージョン
- ARM、Thumb、Thumb-2、およびThumb-2EE の命令セット
- ARM、Thumb、およびThumbEE 状態 (P. 2-4)
- プロセッサモード (P. 2-5)
- レジスタ (P. 2-5)
- 命令セットの概要 (P. 2-7)
- 命令の機能 (P. 2-9)

2.2.1 アーキテクチャのバージョン

本書に掲載されている情報とサンプルは、ARMv4 以上のアーキテクチャを実装するプロセッサが使用されていることを前提としています。上記のプロセッサでは、32 ビットのアドレス範囲を使用します。

各アーキテクチャバージョンの詳細については、*ARM アーキテクチャリファレンスマニュアル*を参照して下さい。

2.2.2 ARM、Thumb、Thumb-2、およびThumb-2EE の命令セット

ARM 命令セットは、広範な演算の実行を可能にする 32 ビット命令のセットです。

ARMv4T 以上のアーキテクチャでは、Thumb 命令セットと呼ばれる 16 ビットの命令セットが定義されています。この命令セットでは、32 ビット ARM 命令セットのほとんどの機能を使用できますが、一部の処理には他の命令が必要になります。Thumb 命令セットでは、パフォーマンスと引き換えに優れたコード密度を実現しています。

ARMv6T2 には、Thumb 命令セットのメジャーアップデートである Thumb-2 が定義されています。ARM 命令セットとほぼ同一の機能が提供されます。16 ビットと 32 ビットの命令を備え、ARM に類似したパフォーマンスと Thumb に類似したコード密度を両立しています。

ARMv6T2 には、ARM 命令セットの新しい命令も定義されています。

ARMv6 以上のアーキテクチャでは、すべての ARM および Thumb 命令はリトルエンディアンです。また、ARMv6T2 以上のアーキテクチャでは、すべての Thumb-2 命令フェッチもリトルエンディアンです。

ARMv7 には、Thumb-2 Execution Environment (Thumb-2EE) が定義されています。Thumb-2EE 命令セットは、Thumb-2 をベースに、動的に生成されるコード (つまり、実行の直前か実行中にデバイスでコンパイルされるコード) に合わせた変更や追加が行われています。

詳細については、「命令セットの概要」(P. 2-7) を参照して下さい。

2.2.3 ARM、Thumb、および ThumbEE 状態

ARM 命令を実行中のプロセッサは、*ARM 状態*で動作しています。また、Thumb 命令セットを実行中のプロセッサは、*Thumb 状態*で動作しています。

一方の状態のプロセッサで他方の命令セットの命令を実行することはできません。例えば、ARM 状態のプロセッサでは Thumb 命令を実行できません。また、Thumb 状態のプロセッサでは ARM 命令を実行できません。このため、現在の状態とは異なる命令セットの命令をプロセッサが受け取らないようにする必要があります。

ARM プロセッサでは、必ず ARM 状態でコードの実行を開始します。

Thumb-2EE により、新しい命令セットの状態として ThumbEE 状態が導入されます。この状態の命令セットは、Thumb 命令セットとほぼ同一です。ただし、動作が変更された命令、使用できなくなった命令、および新しく使用できるようになった命令があります。

状態の切り替え

各命令セットには、プロセッサ状態を切り替える命令があります。

ARM 状態と Thumb 状態との間で切り替えを行うには、ARM (CODE32)、THUMB、または CODE16 ディレクティブを使用して、正しいオペコードを生成するアセンブラーモードに切り替える必要があります。Thumb-2EE コードを生成するには、THUMBX ディレクティブを使用します。

詳細については、「命令セットと構文選択のディレクティブ」(P. 7-61) を参照して下さい。

Thumb 状態と ThumbEE 状態との間で切り替えを行うには、ENTERX (Thumb 状態から ThumbEE 状態への切り替え) または LEAVEX (Thumb-2EE 状態から Thumb 状態への切り替え) を使用します。詳細については、「その他の命令」(P. 4-136) を参照して下さい。

2.2.4 プロセッサモード

ARM プロセッサは、アーキテクチャのバージョンに応じて、さまざまなプロセッサモードをサポートします（表 2-1 参照）。

表 2-1 ARM プロセッサモード

プロセッサモード	アーキテクチャ	モード番号
ユーザ	すべて	0b10000
FIQ - 高速割り込み要求	すべて	0b10001
IRQ - 割り込み要求	すべて	0b10010
スーパーバイザ	すべて	0b10011
アボート	すべて	0b10111
未定義	すべて	0b11011
システム	ARMv4 以上	0b11111
監視	Security Extensions のみ	0b10110

ユーザモード以外のモードはすべて特権モードと呼ばれます。特権モードでは、システムリソースにフルアクセスすることができ、モードを自由に変更することができます。

通常、タスク保護が必要なアプリケーションはユーザモードで実行されます。組み込みアプリケーションの中には、スーパーバイザモードまたはシステムモードだけで実行されるものもあります。

ユーザモード以外のモードは、例外を処理したり、特権付きリソースにアクセスしたりする目的で使用されます。詳細については、*RealView Compilation Tools v3.0 Developer Guide* の「プロセッサ例外処理」を参照して下さい。

2.2.5 レジスタ

ARM プロセッサには 37 本のレジスタがあります。これらのレジスタは、部分的に重複したバンクに配置されます。プロセッサモードごとに異なるレジスタバンクが使用されます。プロセッサ例外と特権命令を処理する際、バンクレジスタによって高速なコンテキストスイッチが可能になります。レジスタバンクの構成については、*ARM アーキテクチャリファレンスマニュアル*を参照して下さい。

以下のレジスタを使用できます。

- 30 本の汎用 32 ビットレジスタ (P. 2-6)
- プログラムカウンタ (PC) (P. 2-6)
- カレントプログラムステータスレジスタ (CPSR) (P. 2-6)
- セーブドプログラムステータスレジスタ (SPSR) (P. 2-7)

30 本の汎用 32 ビットレジスタ

現在のプロセッサモードに応じて、一度に 15 本 (r0、r1、...、r13、r14) までの汎用レジスタが認識されます。

ARM、Thumb、および Thumb-2 アセンブリ言語では、r13 がスタックポインタ (sp) として使用されます。C コンパイラと C++ コンパイラでは、必ず r13 がスタックポインタとして使用されます。

ユーザモードでは、r14 がリンクレジスタ (lr) として使用され、サブルーチン呼び出しが実行されたときに復帰アドレスが r14 にストアされます。復帰アドレスがスタックにストアされる場合は、r14 を汎用レジスタとして使用することができます。

例外処理モードでは、r14 に例外の復帰アドレス（例外内でサブルーチン呼び出しが実行される場合はサブルーチンの復帰アドレス）がストアされます。復帰アドレスがスタックにストアされる場合は、r14 を汎用レジスタとして使用することができます。

プログラムカウンタ (PC)

プログラムカウンタは、r15 (pc) としてアクセスされます。プログラムカウンタは、ARM 状態では 1 命令ごとに 1 ワード (4 バイト) ずつインクリメントされ、Thumb 状態では実行される命令のサイズの分だけインクリメントされます。分岐命令は、デスティネーションアドレスをプログラムカウンタにロードします。プログラムカウンタは、データ処理命令を使用して直接ロードすることもできます。例えば、サブルーチンから戻る場合には、以下のコマンドを使用して、リンクレジスタをプログラムカウンタにコピーできます。

```
BX lr
```

現在実行中の命令のアドレスは、r15 (pc) にはストアされません。通常、実行中の命令のアドレスは、ARM 命令では pc-8、Thumb 命令では pc-4 になります。

カレントプログラムステータスレジスタ (CPSR)

CPSR には以下の情報が保持されます。

- 論理演算装置 (ALU) のステータスフラグのコピー
- 現在のプロセッサモード
- 割り込みディセーブルフラグ

条件命令が実行されるかどうかは、CPSR の ALU ステータスフラグによって決まります。詳細については、「条件実行」(P. 2-17) を参照して下さい。

Thumb 対応プロセッサまたは Jazelle® 対応プロセッサでは、現在のプロセッサ状態 (ARM、Thumb、ThumbEE、Jazelle のいずれか) も CPSR に保持されます。

ARMv5TE と ARMv6 以上のアーキテクチャでは、Q フラグも CPSR に保持されます (「ALU ステータスフラグ」(P. 2-18) 参照)。

ARMv6 以上のアーキテクチャでは、GE フラグ（「並列加算と並列減算」(P. 4-105) 参照）とエンディアンビット（「SETEND」(P. 4-146) 参照）も CPSR に保持されます。

ARMv6T2 以上のアーキテクチャの Thumb-2 命令では、CPSR に新しい状態ビットが導入されました。これらは IT 命令で IT ブロックの条件付き実行を制御するために使用されます（「IT」(P. 4-74) 参照）。

セーブドプログラムステータスレジスタ (SPSR)

SPSR は、例外発生時に CPSR を保持する目的で使用されます。各例外処理モードでは、1 つの SPSR にアクセスできます。ユーザモードとシステムモードは例外処理モードではないため、SPSR を利用できません。詳細については、*RealView Compilation Tools v3.0 Developer Guide* の「プロセッサ例外処理」を参照して下さい。

2.2.6 命令セットの概要

すべての ARM 命令の長さは 32 ビットです。これらの命令はワード境界で整列されて保持されるため、ARM 状態では命令アドレスの最下位 2 ビットが常にゼロになります。

Thumb、Thumb-2、および Thumb-2EE 命令の長さは 16 ビットまたは 32 ビットのいずれかです。これらの命令はハーフワード境界で整列されて保持されるため、Thumb 状態では命令アドレスの最下位ビットが常にゼロになります。

命令の中には、最下位ビットを使用して、分岐先のコードが Thumb コードか ARM コードかを決定するものもあります。

Thumb-2 が導入されるまで、Thumb 命令セットは、ARM 命令セットの機能の一部のサブセットにしか対応していませんでした。これは、ほぼすべての Thumb 命令が 16 ビットであったためです。Thumb-2 命令セットの機能は、ARM 命令セットの機能とほぼ同等です。

ARM 命令と Thumb 命令の構文の詳細については、「第 4 章 ARM 命令と Thumb 命令」を参照して下さい。

ARM 命令と Thumb 命令は、以下の機能グループに分類できます。

- 分岐命令
- データ処理命令
- 単一レジスタロード/ストア命令
- 多重レジスタロード/ストア命令
- ステータスレジスタアクセス命令 (P. 2-8)
- コプロセッサ命令 (P. 2-8)

分岐命令

分岐命令は以下の目的に使用されます。

- 逆方向に分岐してループを形成する。
- 条件付き構造内で順方向に分岐する。
- サブルーチンに分岐する。
- プロセッサの ARM 状態と Thumb 状態を切り替える。

データ処理命令

データ処理命令は汎用レジスタに対して演算を実行します。このグループの命令は、2本のレジスタの内容に対して加算、減算、ビットごとの論理演算などを実行し、3本目のレジスタにその結果を返すことができます。これらの命令は、単一レジスタの値や、レジスタの値と命令内で渡される定数（イミディエート値）に対して演算を実行することもできます。

Long 乗算命令は、64 ビットの結果を 2 本のレジスタに分けて返します。

単一レジスタロード / ストア命令

单一レジスタロード / ストア命令は、单一レジスタとメモリとの間で値のロードまたはストアを行います。これらの命令を使用して、32 ビットワード、16 ビットハーフワード、または 8 ビット符号なしバイトのロードまたはストアを行うことができます。また、バイトとハーフワードのロードを符号拡張またはゼロ拡張にすることにより、32 ビットレジスタを充填できます。

また、いくつかの命令は、64 ビットダブルワードの値を 2 本の 32 ビットレジスタに分けてロードまたはストアできるように定義されています。

多重レジスタロード / ストア命令

多重レジスタロード / ストア命令は、汎用レジスタのサブセットをメモリからロードするか、メモリにストアします。これらの命令の詳細については、「[多重レジスタロード / ストア命令](#)」(P. 2-37) を参照して下さい。

ステータスレジスタアクセス命令

ステータスレジスタアクセス命令は、CPSR または SPSR の内容を汎用レジスタとの間で転送します。

コプロセッサ命令

コプロセッサ命令は、ARM アーキテクチャの一般的な拡張方法をサポートします。

2.2.7 命令の機能

このセクションでは、以下の内容について説明します。

- 条件実行
- レジスタへのアクセス (P. 2-9)
- インラインバレルシフタへのアクセス (P. 2-10)

条件実行

ほぼすべての ARM 命令は、CPSR 内の ALU ステータスフラグの値に基づいて条件実行できます。分岐を使用して条件命令をスキップする必要はありませんが、一連の命令が同じ条件に依存する場合はそのようにした方がよいこともあります。

Thumb-2 に対応していないプロセッサの Thumb 状態では、条件分岐が条件実行用の唯一のメカニズムです。ほとんどのデータ処理命令は、ALU フラグを更新します。通常、命令が ALU フラグを更新するかどうかを指定することはできません。

Thumb-2 では、同じ ALU フラグと IT (If-Then) 命令を使用して、条件実行に代わるメカニズムを提供します。IT は 16 ビット命令で、この命令を使用すると最大 4 つの条件実行を行うことができます。条件実行のメカニズムを提供する命令は、他にもいくつかあります。

ARM および Thumb-2 コードでは、データ処理命令が ALU フラグを更新するかどうかを指定できます。ある 2 つの命令の間に多数の命令が存在する場合でも、一方の命令で設定された ALU フラグを使用して他方の命令の実行を制御できます。

詳細については、「条件実行」(P. 2-17) を参照して下さい。

レジスタへのアクセス

ARM 状態では、すべての命令が r0 ~ r14 にアクセスできます。また、ほとんどの命令が r15 (pc) にもアクセスできます。MRS 命令と MSR 命令を使用して CPSR と SPSR の内容を汎用レジスタに移動し、通常のデータ処理命令によってその内容を操作することができます。詳細については、「MRS」(P. 4-140) および「MSR」(P. 4-141) を参照して下さい。

Thumb-2 プロセッサの Thumb 状態でも同様の機能が提供されますが、重要度が低い r15 へのアクセスは許可されない場合があります。

Thumb-2 に対応していないプロセッサの Thumb 状態では、ほとんどの命令が r0 ~ r7 以外のレジスタにアクセスできません。少数の命令のみが r8 ~ r15 にアクセスできます。レジスタ r0 ~ r7 は Lo レジスタと呼ばれ、レジスタ r8 ~ r15 は Hi レジスタと呼ばれます。

インラインバレルシフタへのアクセス

ARM 論理演算装置には、シフト演算とロテート演算を可能にする 32 ビットのバレルシフタがあります。すべての ARM および Thumb-2 データ処理命令と単一レジスタデータ転送命令の第 2 オペランドは、データ処理またはデータ転送の実行前に、それぞれの命令の中でシフトできます。これにより、以下のようなことが可能になります。

- 位取りアドレッシング
- 定数による乗算
- 定数の構成

バレルシフタを使用した定数の生成については、「レジスタへの定数のロード」(P. 2-23) を参照して下さい。

Thumb-2 命令は、ARM 命令と同様のバレルシフタへのアクセスを提供します。

Thumb-2 に対応していないプロセッサで使用できる Thumb 命令は、他の命令を使用しない限りバレルシフタにアクセスできません。

2.3 アセンブリ言語モジュールの構造

アセンブリ言語とは、オブジェクトコードを生成するために、ARM アセンブラー (`armasm`) が解析およびアセンブルする言語です。デフォルトでは、アセンブラーはソースコードが ARM アセンブリ言語で記述されていることを前提としています。

`armasm` は、以前のバージョンの ARM アセンブリ言語で記述されたソースコードも認識します。この場合、以前のバージョンで記述されていることを示す必要はありません。

また、`armasm` は、以前のバージョンの Thumb アセンブリ言語で記述されたソースコードも認識します。この場合は、`--16` コマンドラインオプションを使用するか、ソースコードで `CODE16` ディレクティブを使用して、以前のバージョンで記述されていることを `armasm` に示す必要があります。

このセクションでは、以下の内容について説明します。

- アセンブリ言語ソースファイルのレイアウト
- ARM アセンブリ言語モジュールのサンプル (P. 2-14)
- サブルーチンの呼び出し (P. 2-16)

2.3.1 アセンブリ言語ソースファイルのレイアウト

以下はアセンブリ言語で記述するソース行の汎用形式です。

```
{label} {instruction|directive|pseudo-instruction} {;comment}
```

——注——

ラベルが指定されていない場合でも、命令、擬似命令、およびディレクティブの前には、スペースやタブなどのホワイトスペースを挿入する必要があります。

このソース行の 3 つのセクションはすべてオプションです。コードを読みやすくするために空白行を使用することもできます。

大文字と小文字の規則

命令ニーモニック、ディレクティブ、およびシンボルレジスタ名は、大文字と小文字のどちらで記述することができますが、大文字と小文字を混ぜることはできません。

行の長さ

ソースファイルを読みやすくするには、行末にバックスラッシュ (\) を挿入して長いソース行を複数行に分割します。バックスラッシュの後には文字（スペースやタブも含む）を挿入しないで下さい。アセンブラーは、バックスラッシュや行末シーケンスをホワイトスペースとして処理します。

—— 注 ——

引用符で囲まれた文字列内でバックスラッシュや行末シーケンスを使用しないで下さい。

バックスラッシュを使用した拡張も含め、行の長さは 4095 文字までに制限されています。

ラベル

ラベルは、アドレスを表すシンボルです。ラベルで指定されたアドレスはアセンブリ時に計算されます。

アセンブリは、ラベルが定義されているセクションの起点を基準としてラベルのアドレスを計算します。同じセクション内のラベルへの参照には、プログラムカウンタにオフセットを加算した値、またはプログラムカウンタからオフセットを減算した値を使用できます。これをプログラム相対アドレッシングと呼びます。

他のセクションに含まれるラベルのアドレスはリンク時（リンクによって各セクションにメモリ内の特定の位置が割り当てられたとき）に計算されます。

ローカルラベル

ローカルラベルは、ラベルのサブクラスです。ローカルラベルは 0 ~ 99 の値で始まります。他のラベルとは異なり、ローカルラベルは何度でも定義できます。ローカルラベルはマクロを使用してラベルを生成するときに便利です。アセンブリはローカルラベルへの参照を検出すると、そのローカルラベルに近いインスタンスに参照をリンクします。

ローカルラベルの有効範囲は、AREA ディレクティブによって制限されます。ROUT ディレクティブを使用すると、この有効範囲をさらに厳密に制限できます。

以下に関する詳細については、「ローカルラベル」(P. 3-25) を参照して下さい。

- ローカルラベル宣言の構文
- アセンブリがローカルラベルへの参照とラベルを関連付ける方法

コメント

行内の最初のセミコロンは、文字列定数の中に出現する場合を除き、コメントの開始を意味します。その行の終わりがコメントの終わりになります。コメントのみの行も有効です。すべてのコメントはアセンブリによって無視されます。

定数

定数には、以下のいずれかを使用できます。

数値	数値定数には以下の形式を使用できます。
	<ul style="list-style-type: none">• 10 進数 (123 など)• 16 進数 (0x7B など)• $n_{-}xxx$: n 2 ~ 9 の基數 xxx その基數の数値
ブール	ブール定数 TRUE と FALSE は、{TRUE} および {FALSE} と記述する必要があります。
文字	文字定数は、引用符で囲まれた 1 つの文字または 1 つのエスケープ文字（標準 C エスケープ文字を使用）で構成されます。
文字列	文字列は、二重引用符に囲まれた文字とスペースで構成されます。文字列内で二重引用符またはドル記号をリテラルテキスト文字として使用する場合、該当する文字を 2 つ続けて記述することにより 1 つの文字を表す必要があります。例えば、文字列内で 1 つの \$ を使用する場合は、\$\$ と記述する必要があります。文字列定数内では標準 C エスケープシーケンスを使用できます。

2.3.2 ARM アセンブリ言語モジュールのサンプル

例 2-1 は、アセンブリ言語モジュールのいくつかの主要構成要素を示しています。このサンプルは ARM アセンブリ言語で記述されています。このサンプルは、主なサンプルディレクトリの *install_directory\RVDS\Examples* に *armex.s* という名前で収録されています。このサンプルをアセンブル、リンク、および実行する方法については、「サンプルコード」(P. 2-2) を参照して下さい。

以下のセクションでは、このサンプルの構成要素について詳しく説明します。

例 2-1

AREA	ARMEx, CODE, READONLY	
		; Name this block of code ARMEx
ENTRY		; Mark first instruction to execute
start		
	MOV r0, #10	; Set up parameters
	MOV r1, #3	
	ADD r0, r0, r1	; r0 = r0 + r1
stop		
	MOV r0, #0x18	; angel_SWIreason_ReportException
	LDR r1, =0x20026	; ADP_Stopped_ApplicationExit
	SVC 0x123456	; ARM semihosting (formerly SWI)
	END	; Mark end of file

ELF セクションと AREA ディレクティブ

ELF セクションは、名前の付いた、分割不可能な独立したコードシーケンスまたはデータシーケンスです。アプリケーションを作成するには少なくとも 1 つのコードセクションが必要です。

アセンブリまたはコンパイルからの出力には、以下を含めることができます。

- 1 つ以上のコードセクション。通常、これらは読み出し専用セクションです。
- 1 つ以上のデータセクション。通常、これらは読み出し - 書き込みセクションであり、ゼロで初期化 (ZI) される場合があります。

リンクはセクション配置規則に基づいて各セクションをプログラムイメージ内に配置します。ソースファイル内で隣接するセクションが、アプリケーションイメージ内でも隣接しているとは限りません。リンクによるセクションの配置方法については、*RealView Compilation Tools v3.0 Linker and Utilities Guide* で、基本的なリンクの機能に関する章を参照して下さい。

ソースファイルでは、セクションの開始位置を AREA ディレクティブによってマークします。このディレクティブでセクションに名前を付け、属性を設定します。属性は、名前の後にコンマで区切って配置します。AREA ディレクティブの構文の詳細については、「*AREA*」(P. 7-66) を参照して下さい。

セクションには任意の名前を指定できます。ただし、アルファベット以外の文字で始まる名前は縦棒で囲む必要があります。縦棒で囲まないと、*AREA name missing* エラーが生成されます。例えば、`|1_DataArea|` のように記述します。

例 2-1 (2-14 ページ) では、`READONLY` としてマークされたコードを含む `ARMex` という名前の单一セクションを定義しています。

ENTRY ディレクティブ

ENTRY ディレクティブは、最初に実行される命令をマークします。C コードを含むアプリケーションでは、エントリポイントが C ライブラリ初期化コード内にも含まれます。初期化コードと例外ハンドラにもエントリポイントが含まれます。

アプリケーションの実行

例 2-1 (2-14 ページ) のアプリケーションコードは、ラベル `start` で実行を開始し、10 進数の 10 と 3 をレジスタ `r0` と `r1` にロードします。次に、これらのレジスタを加算し、結果を `r0` に返します。

アプリケーションの終了

メインコードの実行後、このアプリケーションは制御をデバッガに戻すことによって終了します。この動作は、以下のパラメータを設定した ARM セミホスティング SVC (デフォルトでは 0x123456) を使用して行われます。

- `r0 = angel_SWIreason_ReportException (0x18)`
- `r1 = ADP_Stopped_ApplicationExit (0x20026)`

詳細については、*RealView Compilation Tools v3.0 Compiler and Libraries Guide* の「セミホスティング」を参照して下さい。

END ディレクティブ

このディレクティブは、ソースファイルの処理を停止するようアセンブリに指示します。すべてのアセンブリ言語ソースモジュールは、END ディレクティブが単独で記述された行で終了する必要があります。

2.3.3 サブルーチンの呼び出し

サブルーチンを呼び出すには、リンク付き分岐命令を使用します。構文は以下のとおりです。

BL destination

destination にはサブルーチンの最初の命令に付けたラベルを指定するのが一般的です。

また、*destination* には、プログラム相対式またはレジスタ相対式を指定することもできます。詳細については、「B, BL, BX, BLX, BXJ」(P. 4-121) を参照して下さい。

BL 命令は以下を実行します。

- リンクレジスタに復帰アドレスを配置する。
- プログラムカウンタにサブルーチンのアドレスを設定する。

サブルーチンコードの実行後、BX lr 命令を使用して復帰できます。原則として、レジスタ r0 ~ r3 を使用してパラメータをサブルーチンに渡し、r0 を使用して結果を発呼側に返します。

——注——

別々にアセンブルまたはコンパイルされたモジュール間の呼び出しあは、プロシージャコール標準で定義されている制約条件や規則に準拠する必要があります。詳細については、install_directory\Documentation\Specifications\... にある *Procedure Call Standard for the ARM Architecture* (aapcs.pdf) を参照して下さい。

例 2-2 は、2 つのパラメータ値を加算して、その結果を r0 に返すサブルーチンを示しています。このサンプルは、主なサンプルディレクトリの install_directory\RVDS\Examples に subrout.s という名前で収録されています。このサンプルをアセンブル、リンク、および実行する方法については、「サンプルコード」(P. 2-2) を参照して下さい。

例 2-2

```

        AREA    subrout, CODE, READONLY      ; Name this block of code
        ENTRY                           ; Mark first instruction to execute
start   MOV     r0, #10                ; Set up parameters
        MOV     r1, #3
        BL     doadd                 ; Call subroutine
stop    MOV     r0, #0x18              ; angel_SWIreason_ReportException
        LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
        SVC     0x123456              ; ARM semihosting (formerly SWI)

doadd  ADD     r0, r0, r1             ; Subroutine code
        BX     lr                   ; Return from subroutine
        END                           ; Mark end of file

```

2.4 条件実行

ARM 状態と、Thumb-2 をサポートするプロセッサの Thumb 状態では、ほとんどのデータ処理命令で、演算結果に従ってカレントプログラムステータスレジスタ (CPSR) 内の ALU ステータスフラグを更新するかどうかを選択できます。

以前のアーキテクチャの Thumb 状態では、ほとんどのデータ処理命令が ALU ステータスフラグを自動的に更新します。フラグを更新しないように指定するオプションはありません。その他の命令では、フラグを更新できません。

ほぼすべての ARM 命令は、CPSR 内の ALU ステータスフラグの状態に基づいて条件実行できます。命令を条件実行するために追加する接尾文字の一覧については、P. 2-18 表 2-2 を参照して下さい。

ほぼすべての Thumb-2 命令は、特殊な IT (If-Then) 命令を使用することによって条件実行できます。また、条件分岐命令もいくつかあります。

Thumb-2 以前のプロセッサの Thumb 状態では、条件分岐が条件実行の唯一のメカニズムです。

フラグの状態は、更新されるまで保持されます。実行されない条件付き命令は、フラグの状態に影響を与えません。

命令には、フラグのサブセットを更新するものがあります。それ以外のフラグは、これらの命令によって更新されません。詳細については、各命令の説明を参照して下さい。

ARM 状態と、Thumb-2 をサポートするプロセッサの Thumb 状態では、以下のいずれかの時点での別の命令によってセットされたフラグに基づいて、命令を条件実行できます。

- フラグを更新した命令の直後
- フラグを更新していない任意の数の命令の後

このセクションでは、以下の内容について説明します。

- *ALU ステータスフラグ* (P. 2-18)
- *条件実行* (P. 2-18)
- *条件実行の使用* (P. 2-19)
- *条件実行の使用例* (P. 2-20)
- *Q フラグ* (P. 2-22)

2.4.1 ALU ステータスフラグ

CPSR は以下の ALU ステータスフラグを保持します。

- N** 演算結果が負の場合にセットされます。
- Z** 演算結果がゼロの場合にセットされます。
- C** 演算の結果としてキャリーが行われた場合にセットされます。
- V** 演算の結果として overflow が生じた場合にセットされます。

キャリーは、加算結果が 2^{32} 以上の場合、減算結果が正の場合、または移動命令や論理命令によるインラインパレルシフタ演算の結果として発生します。

オーバフローは、加算、減算、または比較の結果が 2^{31} 以上または -2^{31} 未満の場合に発生します。

2.4.2 条件実行

条件付きにできる ARM 命令には、任意で条件コードを指定できます。構文の説明では、条件コードを {cond} と表記しています。この条件は命令にエンコードされます。条件コードが指定されている命令は、CPSR の条件コードフラグが、指定した条件を満たしている場合にのみ実行されます。P. 2-18 表 2-2 は、使用可能な条件コードを示しています。

前に実行された IT 命令によって条件付きになった Thumb-2 命令では、同じ条件コードを使用します。

また、表 2-2 は、条件コードの接尾文字と、N、Z、C、および V フラグとの関係も示しています。

表 2-2 条件コードの接尾文字

接尾文字	フラグ	意味
EQ	Z セット	等しい
NE	Z クリア	等しくない
CS/HS	C セット	以上 (符号なし \geq)
CC/LO	C クリア	未満 (符号なし $<$)
MI	N セット	負
PL	N クリア	正またはゼロ
VS	V セット	オーバフロー
VC	V クリア	オーバフローなし

表 2-2 条件コードの接尾文字（続き）

接尾文字	フラグ	意味
HI	C セットかつ Z クリア	より大きい（符号なし >）
LS	C クリアまたは Z セット	以下（符号なし <=）
GE	N = V	符号付き >=
LT	N ≠ V	符号付き <
GT	Z クリア、N = V	符号付き >
LE	Z セット、N ≠ V	符号付き <=
AL	すべて	無条件（通常は省略します）

例 2-3 は、条件実行の例を示しています。

例 2-3

```

ADD      r0, r1, r2      ; r0 = r1 + r2, don't update flags
ADDS     r0, r1, r2      ; r0 = r1 + r2, and update flags
ADDSCS   r0, r1, r2      ; If C flag set then r0 = r1 + r2, and update flags
CMP      r0, r1          ; update flags based on r0-r1.

```

2.4.3 条件実行の使用

ARM 命令の条件実行を使用すると、コード内の分岐命令の数を減らすことができます。これによりコード密度が向上します。Thumb-2 の IT 命令を使用した場合にも同様の効果が得られます。

分岐命令により、プロセッササイクルが増加します。通常、分岐予測ハードウェアを搭載していない ARM プロセッサは、分岐が発生するたびに、プロセッサパイプラインを再充填するのに 3 プロセッササイクルを必要とします。

ARM10™ や StrongARM® などの一部の ARM プロセッサには、分岐予測ハードウェアが搭載されています。これらのプロセッサを使用するシステムでは、予測を誤った場合でもパイプラインをフラッシュして再充填するだけで済みます。

2.4.4 条件実行の使用例

この例は、ユークリッドの最大公約数 (gcd) アルゴリズムの 2 つの実装を使用し、条件実行によってコード密度と実行速度を向上させる方法を示しています。実行速度の詳細分析は、ARM7™ プロセッサでのみ実行できます。コード密度の計算は、すべての ARM プロセッサで実行できます。

C では、このアルゴリズムを以下のように表すことができます。

```
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

以下のように、分岐の条件付き実行のみを使用して gcd 関数を実装できます。

```
gcd      CMP      r0, r1
          BEQ      end
          BLT      less
          SUB      r0, r0, r1
          B       gcd
less
          SUB      r1, r1, r0
          B       gcd
end
```

分岐を使用するため、このコードでは 7 つの命令を実行します。分岐が発生するたびに、プロセッサはパイプラインを再充填して新しい位置から実行を継続する必要があります。他の命令と未実行分岐にはそれぞれ 1 サイクルが使用されます。

ARM 命令セットの条件実行機能を使用すると、以下のようにわずか 4 つの命令で gcd 関数を実装できます。

```
gcd
      CMP      r0, r1
      SUBGT   r0, r0, r1
      SUBLT   r1, r1, r0
      BNE      gcd
```

ほとんどの場合、コードサイズが小さくなるだけでなく、コードの実行速度も向上します。表 2-3 と表 2-4 は、r0 = 1、r1 = 2 の場合に、各実装によって使用されるサイクル数を示しています。この場合、分岐を使用せずに、すべての命令を条件実行にすることにより、3 サイクルを省くことができます。

条件実行を使用するコードは、 $r0 = r1$ である限り常に同じサイクル数で実行されます。それ以外の場合は、条件実行を使用するコードの方が少ないサイクル数で実行されます。

表 2-3 条件分岐のみの場合

r0: a	r1: b	命令	サイクル (ARM7)
1	2	CMP r0, r1	1
1	2	B EQ end	1 (実行されません)
1	2	BLT less	3
1	2	SUB r1, r1, r0	1
1	2	B gcd	3
1	1	CMP r0, r1	1
1	1	B EQ end	3
			合計 13

表 2-4 すべての命令を条件実行する場合

r0: a	r1: b	命令	サイクル (ARM7)
1	2	CMP r0, r1	1
1	2	SUBGT r0,r0,r1	1 (実行されません)
1	1	SUBLT r1,r1,r0	1
1	1	BNE gcd	3
1	1	CMP r0,r1	1
1	1	SUBGT r0,r0,r1	1 (実行されません)
1	1	SUBLT r1,r1,r0	1 (実行されません)
1	1	BNE gcd	1 (実行されません)
			合計 10

Thumb バージョンの gcd

条件実行が可能な 16 ビット Thumb 命令は B のみのため、Thumb コードでは条件分岐を使用して gcd アルゴリズムを記述する必要があります。

ARM コードの条件分岐の実装と同様に、Thumb コードの場合にも 7 つの命令が必要です。Thumb 命令を使用すると、全体のコードサイズは 14 バイトとなり、小さな ARM の実装 (16 バイト) と比較しても小さくなります。

さらに、16 ビットメモリを使用しているシステムでは、Thumb の実行速度が 2 番目の ARM の実装を上回ります。これは、1 つの 32 ビット ARM 命令には 2 回のフェッチが必要であるのに対し、1 つの 16 ビット Thumb 命令には 1 回のメモリアクセスで済むためです。

分岐予測とキャッシュ

実行速度を上げるためにコードを最適化するには、命令のタイミング、分岐予測ロジック、およびターゲットシステムのキャッシュ動作をよく理解しておく必要があります。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と、各プロセッサ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。

2.4.5 Q フラグ

Q フラグは、ARMv5TE、および ARMv6 以上のアーキテクチャにおいて、サチュレート算術命令 (*/QADD*, *QSUB*, *QDADD*, *QDSUB*) (P. 4-100) 参照) でいつサチュレーションが発生したか、または特定の乗算命令 (*/SMLLxy*, *SMLAxxy*) (P. 4-83) と */SMLWLW*, *SMLAWy* (P. 4-85) 参照) でいつオーバフローが発生したかを記録するために使用されます。

Q フラグはステンシルフラグです。上記の命令で Q フラグをセットすることはできますが、クリアすることはできません。こうした一連の命令を実行した後でフラグをテストすると、それらの命令の実行中に発生したサチュレーションやオーバフローを検出できるため、1 つの命令を実行するたびにフラグをチェックしなくとも済みます。

Q フラグをクリアするには、**MSR** 命令を使用します (*/MSR* (P. 4-141) 参照)。

Q フラグの状態を条件コードで直接テストすることはできません。Q フラグの状態を読み出すには、**MRS** 命令を使用します (*/MRS* (P. 4-140) 参照)。

2.5 レジスタへの定数のロード

メモリからデータをロードせずに、1つのARM命令で任意の32ビットイミディエイト定数をレジスタにロードすることはできません。これはARM命令の長さが32ビットしかないためです。Thumb命令にも同様の制限があります。

また、多くの汎用定数は、別のロード命令を使用しなくてもデータ処理命令内にオペランドとして直接含めることができます。

データロード命令を使用して32ビット値をレジスタにロードすることはできますが、多くの汎用定数は、より直接的かつ効率的な方法でロードできます。

ARMv6T2以上のアーキテクチャでは、MOV命令を実行してからMOVT命令を実行することによって、任意の32ビット値をレジスタにロードすることもできます。また、擬似命令のMOV32を使用して、命令シーケンスを作成できます。

以下のセクションでは、いくつかのロード方法について説明します。

- MOV命令とMVN命令を使用して特定範囲のイミディエート値をロードする方法
詳細については、「*MOVとMVNを使用した直接ロード*」(P. 2-24) を参照して下さい。
- MOV32擬似命令を使用して任意の32ビット定数をロードする方法
詳細については、「*MOV32を使用したロード*」(P. 2-28) を参照して下さい。
- LDR擬似命令を使用して任意の32ビット定数をロードする方法
詳細については、「*LDR Rd, =constを使用したロード*」(P. 2-28) を参照して下さい。
- 浮動小数点定数をロードする方法
詳細については、「*浮動小数点定数のロード*」(P. 2-30) を参照して下さい。

2.5.1 MOV と MVN を使用した直接ロード

ARM と Thumb-2 では、32 ビットの MOV 命令と MVN 命令を使用して、広範囲の定数値をレジスタに直接ロードできます。

16 ビットのThumb MOV 命令を使用すると、0 ~ 255 の範囲内にある任意の定数をロードできます。16 ビットの MVN 命令を使用して定数をロードすることはできません。

「ARM 状態のイミディエート定数」は、1 つの ARM 命令でロードできる値の範囲を示しています。また、「Thumb-2 イミディエート定数」(P. 2-26) は、1 つの Thumb-2 命令でロードできる値の範囲を示しています。

MOV と MVN のどちらを使用するかを決める必要はありません。アセンブラーがどちらか適切な方を使用します。この機能は値がアセンブリ時変数である場合に便利です。

使用できない定数を使用して命令を記述した場合、アセンブラーは "Immediate *n* out of range for this operation." というエラーを通知します。

ARM 状態のイミディエート定数

ARM 状態での MOV 命令と MVN 命令を使用したロードについて以下に示します。

- MOV 命令は、0x0 ~ 0xFF (0 ~ 255) の範囲内にある任意の 8 ビット定数値をロードできます。
 - これらの値は任意の偶数ビット分ロテートすることもできます。
 - 多くのデータ処理命令では、これらの値を別の命令でロードせずに、イミディエートオペランドとして使用することもできます。
- MVN 命令は、これらの値のビット単位の補数をロードできます。その数値は -(*n*+1) です。*n* は MOV で使用できる値です。
- ARMv6T2 以上のアーキテクチャでは、MOV は 0x0 ~ 0xFFFF (0 ~ 65535) 範囲内にある任意の 16 ビットの数値をロードできます。

P. 2-25 表 2-5 は、ARM 状態で提供される (データ処理命令に使用できる) 8 ビット値の範囲を示しています。

P. 2-25 表 2-6 は、ARM 状態で提供される (MOV 命令にのみ使用できる) 16 ビット値の範囲を示しています。

表 2-5 ARM 状態のイミディエート定数 (8 ビット)

バイナリ	10 進数	ステップ	16 進数	MVN の値 ^a	注
00000000000000000000000000000000abcdefgh	0 ~ 255	1	0 ~ 0xFF	-1 ~ -256	-
00000000000000000000000000000000abcdefgh00	0 ~ 1020	4	0 ~ 0x3FC	-4 ~ -1024	-
00000000000000000000000000000000abcdefgh0000	0 ~ 4080	16	0 ~ 0xFF0	-16 ~ -4096	-
00000000000000000000000000000000abcdefgh000000	0 ~ 16320	64	0 ~ 0x3FC0	-64 ~ -16384	-
...	-
abcdefgħ00000000000000000000000000000000	0 ~ 255 x 2 ²⁴	2 ²⁴	0 ~ 0xFF000000	1 ~ 256 x -2 ²⁴	-
cdefgh00000000000000000000000000ab	(ビット パターン)	-	-	(ビット パターン)	注の b 参照
eħħġħ0000000000000000000000abcd	(ビット パターン)	-	-	(ビット パターン)	注の b 参照
gh00000000000000000000000000000000abcdef	(ビット パターン)	-	-	(ビット パターン)	注の b 参照

表 2-6 MOV 命令に使用できる ARM 状態のイミディエート定数

バイナリ	10 進数	ステップ	16 進数	MVN の値	注
0000000000000000abcdefgħijklmnop	0 ~ 65535	1	0 ~ 0xFFFF	-	注の c 参照

注

以下は、表 2-5 と表 2-6 に関する追加説明です。

- a** MVN の値は、データ処理命令以外の命令では直接オペランドとして使用できません。
- b** これらの値は、ARM 状態でのみ使用できます。この表の他の値は、特に記載のない限り、Thumb-2 でも使用できます。
- c** これらの値は、ARMv6T2 以上のアーキテクチャでのみ使用できます。これらの値は、他の命令ではオペランドとして直接使用することはできません。

Thumb-2 イミディエート定数

ARMv6T2 以上のアーキテクチャの Thumb 状態での、MOV 命令と MVN 命令を使用したロードについて以下に示します。

- 32 ビットの MOV 命令では、以下をロードできます。
 - 0x0 ~ 0xFF (0 ~ 255) の範囲内にある任意の 8 ビット定数値
 - 任意のビット数分左シフトした任意の 8 ビット定数値
 - レジスタのすべての 4 バイトで重複している任意の 8 ビットのビットパターン
 - バイト 1 と 3 がゼロに設定されているときに、バイト 0 と 2 で重複している任意の 8 ビットのビットパターン
 - バイト 0 と 2 がゼロに設定されているときに、バイト 1 と 3 で重複している任意の 8 ビットのビットパターン

多くのデータ処理命令では、これらの値を別の命令でロードせずに、イミディエートオペランドとして使用することもできます。

- 32 ビットの MVN 命令は、これらの値のビット単位の補数をロードできます。その数値は $-(n+1)$ です。n は MOV で使用できる値です。
- 32 ビットの MOV 命令は、0x0 ~ 0xFFFF (0 ~ 65535) の範囲内にある任意の 16 ビットの数値をロードできます。これらの値は、データ処理命令ではイミディエートオペランドとして使用することはできません。

P. 2-27 表 2-7 は、ARMv6T2 以上のアーキテクチャの Thumb 状態で提供される（データ処理命令に使用できる）値の範囲を示しています。

P. 2-27 表 2-8 は、ARMv6T2 以上のアーキテクチャの Thumb 状態で提供される（MOV 命令にのみ使用できる）16 ビット値の範囲を示しています。

表 2-7 Thumb 状態のイミディエート定数

バイナリ	10 進数	ステップ	16 進数	MVN の値 ^a	注
00000000000000000000000000000000abcdefgh	0 ~ 255	1	0 ~ 0xFF	-1 ~ -256	-
00000000000000000000000000000000abcdefgh0	0 ~ 510	2	0 ~ 0x1FE	-2 ~ -512	-
00000000000000000000000000000000abcdefgh00	0 ~ 1020	4	0 ~ 0x3FC	-4 ~ -1024	-
...	-
0abcdefg00000000000000000000000000000000	0 ~ 255 x 2 ²³	2 ²³	0 ~ 0x7F800000	1 ~ 256 x -2 ²³	-
abcdefg00000000000000000000000000000000	0 ~ 255 x 2 ²⁴	2 ²⁴	0 ~ 0xFF000000	1 ~ 256 x -2 ²⁴	-
abcdefghabcdfehabcdfehabcdfe	(ビット パターン)	-	0xXYXYXYXY	0xXYXYXYXY	-
00000000abcdefg00000000abcdefg	(ビット パターン)	-	0x00XY00XY	0xFFXYFFXY	-
abcdefg00000000abcdefg00000000	(ビット パターン)	-	0xXY00XY00	0XYFFXYFF	-
00000000000000000000000000000000abcdefgijkl	0 ~ 4095	1	0 ~ 0xFFFF	-	注の b 参照

表 2-8 MOV 命令に使用できるThumb 状態のイミディエート定数

バイナリ	10 進数	ステップ	16 進数	MVN の値	注
0000000000000000abcdefgijklmnop	0 ~ 65535	1	0 ~ 0xFFFF	-	注の c 参照

注

以下は、表 2-7 と表 2-8 に関する追加説明です。

- a** MVN の値は、他の命令ではオペランドとして直接使用することはできません。
- b** これらの値は、ADD、SUB、および MOV 命令では直接オペランドとして使用できますが、MVN や他のデータ処理命令では直接オペランドとして使用できません。
- c** これらの値は、MOV 命令でのみ使用できます。

2.5.2 MOV32 を使用したロード

ARMv6T2 では、ARM 命令セットと Thumb-2 命令セットの両方に、以下の命令が含まれています。

- 0x00000000 ~ 0x0000FFFF の範囲内にある任意の値をレジスタにロードできる MOV 命令
- レジスタの下位半分の内容を変更することなく、0x0000 ~ 0xFFFF の範囲内にある任意の値をレジスタの上位半分にロードできる MOVT 命令

これら 2 つの命令を使用して、レジスタに任意の 32 ビット定数を構成できます。また、MOV32 擬似命令を使用することもできます。アセンブラーからは、MOV 命令と MOVT 命令のペアが生成されます。MOV32 擬似命令の構文については、「*MOV32 擬似命令*」(P. 4-158) を参照して下さい。

2.5.3 LDR Rd,=const を使用したロード

LDR Rd,=const 擬似命令を使用すると、1 つの命令で任意の 32 ビット数値定数を構成できます。MOV 命令と MVN 命令の範囲外の定数を生成するには、この擬似命令を使用します。

LDR 擬似命令は、特定の定数に対して最も効率的な命令を生成します。

- この定数を MOV 命令または MVN 命令を使用して構成できる場合、アセンブラーによってどちらか適切な命令が生成されます。
- この定数を MOV 命令または MVN 命令を使用して構成できない場合、アセンブラーによって以下の処理が行われます。
 - リテラルプール（コードに組み込まれた、定数値を保持するためのメモリの部分）に値を配置する。
 - リテラルプールから定数を読み出す、プログラム相対アドレスを使用する LDR 命令を生成する。

以下に例を示します。

```
LDR      rn, [pc, #offset to literal pool]
          ; load register n with one word
          ; from the address [pc + offset]
```

アセンブラーが生成した LDR 命令の範囲内にリテラルプールがあることを確認する必要があります。詳細については、「リテラルプールの配置」(P. 2-29) を参照して下さい。

LDR 擬似命令の構文については、「*LDR 擬似命令*」(P. 4-160) を参照して下さい。

リテラルプールの配置

アセンブラーは、各セクションの終了位置にリテラルプールを配置します。各セクションの終了位置は、次のセクションの開始位置にある AREA ディレクティブか、アセンブリの終了位置にある END ディレクティブによって定義されます。インクルードされたファイルの最後にある END ディレクティブは、セクションの終了位置を示しているわけではありません。

大きなセクションでは、デフォルトのリテラルプールが 1 つ以上の LDR 命令の範囲内に収まらない可能性があります。プログラムカウンタから定数までのオフセットは、以下の規則に従う必要があります。

- ARM または Thumb-2 コードの場合：順方向または逆方向に 4KB 未満
- 16 ビット命令を使用する Thumb コードの場合：順方向に 1KB 未満

LDR Rd,**=const** 擬似命令が定数をリテラルプールに配置するように要求している場合、アセンブラーは以下を行います。

- その定数が既存のリテラルプール内に存在し、かつそこからアドレス指定が可能であるかどうかをチェックします。可能であれば、既存の定数でアドレス指定します。
- 定数が既存のリテラルプールに存在しない場合は、次のリテラルプールへの定数の配置を試みます。

次のリテラルプールが範囲外の場合、アセンブラーはエラーメッセージを生成します。この場合は LTORG ディレクティブを使用して、コード内に別のリテラルプールを配置する必要があります。LTORG ディレクティブは失敗した LDR 擬似命令の後、かつ -4 ~ 4KB (ARM、32 ビット Thumb-2 の場合) または 0 ~ 1KB (Thumb、16 ビット Thumb-2 の場合) の範囲内の位置に配置します。詳細については、「LTORG」(P. 7-19) を参照してください。

リテラルプールは、プロセッサによって命令として実行されない位置に配置する必要があります。つまり、無条件分岐命令の後またはサブルーチンの最後にある復帰命令の後に配置して下さい。

例 2-4 はこれを実装するコードを示しています。このサンプルは、主なサンプルディレクトリの *install_directory\RVDS\Examples* に *loadcon.s* という名前で収録されています。このサンプルをアセンブル、リンク、および実行する方法については、「サンプルコード」(P. 2-2) を参照して下さい。

コメントとしてリストされている命令は、アセンブラーによって生成される ARM 命令です。

例 2-4

	AREA	Loadcon, CODE, READONLY		
	ENTRY		; Mark first instruction to execute	
start	BL	func1	; Branch to first subroutine	
	BL	func2	; Branch to second subroutine	
stop	MOV	r0, #0x18	; angel_SWIreason_ReportException	
	LDR	r1, =0x20026	; ADP_Stopped_ApplicationExit	
	SVC	0x123456	; ARM semihosting (formerly SWI)	
func1	LDR	r0, =42	; => MOV R0, #42	
	LDR	r1, =0x55555555	; => LDR R1, [PC, #offset to ; Literal Pool 1]	
	LDR	r2, =0xFFFFFFFF	; => MVN R2, #0	
	BX	lr	; Literal Pool 1 contains ; literal 0x55555555	
func2	LDR	r3, =0x55555555	; => LDR R3, [PC, #offset to ; Literal Pool 1]	
	; LDR r4, =0x66666666			; If this is uncommented it ; fails, because Literal Pool 2 ; is out of reach
	BX	lr		
LargeTable	SPACE	4200	; Starting at the current location, ; clears a 4200 byte area of memory ; to zero	
	END		; Literal Pool 2 is empty	

2.5.4 浮動小数点定数のロード

VFP 命令セットの FLD 擬似命令を使用すると、単精度または倍精度の浮動小数点定数を 1 つの命令でロードできます。

詳細については、「*VLDR 擬似命令*」(P. 5-108) を参照して下さい。

2.6 レジスタへのアドレスのロード

レジスタにアドレスをロードしなければならない場合がよくあります。例えば、変数、文字列定数、またはジャンプテーブルの開始位置のアドレスをロードする必要のある場合があります。

一般的に、アドレスは、現在のプログラムカウンタまたは他のレジスタからのオフセットで表現します。

このセクションでは、アドレスをレジスタにロードする以下の 2 つの方法について説明します。

- レジスタへの直接ロード (*ADR* と *ADRL* を使用した直接ロード参照)
- リテラルプールからのアドレスのロード (*/LDR Rd, =label* を使用したアドレスのロード) (P. 2-34) 参照)

2.6.1 ADR と ADRL を使用した直接ロード

ADR 擬似命令と *ADRL* 擬似命令を使用すると、データロード命令を使用しなくても特定の範囲内のアドレスを生成することができます。*ADR* 命令と *ADRL* 命令では、プログラム相対式を使用できます。プログラム相対式とは、任意のオフセットに付けるラベルで、このラベルのアドレスが現在のプログラムカウンタの相対アドレスとなります。

——注——

ADR 命令または *ADRL* 命令で使用するラベルは、同じコードセクション内に存在している必要があります。同じコードセクション内にないラベルを参照すると、アセンブラーによりエラーが返されます。

Thumb 状態では、16 ビットの *ADR* 命令で生成できるのはワード境界で整列させたアドレスのみです。

Thumb-2 に対応していないプロセッサでは、Thumb 状態で *ADRL* 命令を使用することはできません。

ADR

アセンブリは、以下を生成することにより、`ADR rn, label` 擬似命令を変換します。

- アドレスが範囲内にある場合は、そのアドレスをロードする 1 つの ADD 命令または SUB 命令を生成します。
- 1 つの命令でアドレスに到達できない場合は、エラーメッセージを生成します。

利用できる範囲は、使用する命令セットによって異なります。

ARM

ワード境界またはハーフワード境界で整列されているアドレスから ± 255 バイトの範囲

ワード境界で整列されているアドレスから ± 1020 バイトの範囲

32 ビットの Thumb-2 バイト、ハーフワード、またはワード境界で整列されているアドレスから ± 4095 バイトの範囲

16 ビットの Thumb 0 ~ 1020 バイトの範囲。`label` は、ワード境界で整列させる必要があります。ALIGN ディレクティブを使用して、`label` をワード境界で整列させることができます。

詳細については、「ADR」(P. 4-28) を参照して下さい。

ADRL

アセンブリは、以下を生成することにより、`ADRL rn, label` 擬似命令を変換します。

- アドレスが範囲内にある場合は、そのアドレスをロードする 2 つのデータ処理命令を生成します。
- 2 つの命令でアドレスを構成できない場合は、エラーメッセージを生成します。

利用できる範囲は、使用する命令セットによって異なります。

ARM

バイト境界またはハーフワード境界で整列されているアドレスから $\pm 64\text{KB}$ の範囲

ワード境界で整列されているアドレスから $\pm 256\text{KB}$ の範囲

32 ビットの Thumb-2 バイト、ハーフワード、またはワード境界で整列されているアドレスから $\pm 1\text{MB}$ の範囲

16 ビットの Thumb ADRL 命令は使用できません。

ADRL 擬似命令の範囲外にあるアドレスのロードについては、「`LDR Rd, =label` を使用したアドレスのロード」(P. 2-34) を参照して下さい。

ADR を使用したジャンプテーブルの実装

例 2-5 は、ジャンプテーブルを実装する ARM コードを示しています。ADR 擬似命令では、ジャンプテーブルのアドレスをロードします。このコードは、主なサンプルディレクトリの `install_directory\RVDS\Examples` に `jump.s` という名前で収録されています。このサンプルをアセンブル、リンク、および実行する方法については、「サンプルコード」(P. 2-2) を参照して下さい。

例 2-5 ARM コードによるジャンプテーブルの実装

```

        AREA      Jump, CODE, READONLY    ; Name this block of code
        CODE32
num      EQU      2                 ; Following code is ARM code
        ENTRY
start   MOV      r0, #0             ; Number of entries in jump table
        MOV      r1, #3             ; Mark first instruction to execute
        MOV      r2, #2             ; First instruction to call
        BL       arithfunc          ; Set up the three parameters
stop    MOV      r0, #0x18          ; Call the function
        LDR      r1, =0x20026        ; angel_SWIreason_ReportException
        SVC      0x123456           ; ADP_Stopped_ApplicationExit
        SVC      0x123456           ; ARM semihosting (formerly SWI)
arithfunc CMP      r0, #num          ; Label the function
        ; Treat function code as unsigned
integer  BXHS    lr                ; If code is >= num then simply return
        ADR     r3, JumpTable        ; Load address of jump table
        LDR     pc, [r3,r0,LSL#2]     ; Jump to the appropriate routine
JumpTable DCD     DoAdd
        DCD     DoSub
DoAdd   ADD      r0, r1, r2          ; Operation 0
        BX      lr                  ; Return
DoSub   SUB      r0, r1, r2          ; Operation 1
        BX      lr                  ; Return
        END
        ; Mark the end of this file

```

例 2-5 (2-33 ページ) では、関数 `arithfunc` が 3 つの引数を取り、`r0` に結果を返します。最初の引数によって、2 番目と 3 番目の引数に対して実行される演算が決まります。

引数 1 = 0 結果 = 引数 2 + 引数 3

引数 1 = 1 結果 = 引数 2 - 引数 3

ジャンプテーブルは、以下の命令とアセンブラディレクティブを使用して実装されます。

EQU	アセンブラディレクティブです。このディレクティブを使用してシンボルに値を指定します。例 2-5 (2-33 ページ) では、 <i>num</i> に値 2 を割り当てます。コード内で <i>num</i> を使用すると、値 2 に置き換えられます。EQU のこのような使用法は、C 言語で <code>#define</code> を使用して定数を定義する方法と似ています。
DCD	ストアする 1 つ以上のワードを宣言します。例 2-5 (2-33 ページ) では、各 DCD が、ジャンプテーブルの特定の節を処理するルーチンのアドレスをストアします。
LDR	<p>LDR pc,[r3,r0,LSL#2] 命令は、ジャンプテーブルから必要な節のアドレスをプログラムカウンタにロードします。この命令は以下を実行します。</p> <ul style="list-style-type: none"> • r0 が保持する節の数を 4 で乗算してワードオフセットを求める。 • 結果をジャンプテーブルのアドレスに加算する。 • 加算したアドレスの内容をプログラムカウンタにロードする。

2.6.2 LDR Rd, =label を使用したアドレスのロード

LDR Rd,= 擬似命令を使用することにより、どのような 32 ビット定数でもレジスタにロードできます（「LDR Rd, =const を使用したロード」(P. 2-28) 参照）。また、この命令には、ラベルやオフセット付きラベルなどのプログラム相対式も使用できます。

アセンブラは、以下を行うことにより LDR r0, =label 擬似命令を変換します。

- *label* のアドレスをリテラルプール（コードに組み込まれた、定数値を保持するためのメモリの部分）に配置する。
- リテラルプールからアドレスを読み出すプログラム相対 LDR 命令を生成する。以下に例を示します。

```
LDR      rn [pc, #offset to literal pool]
          ; load register n with one word
          ; from the address [pc + offset]
```

リテラルプールが範囲内にあることを確認する必要があります。詳細については、「リテラルプールの配置」(P. 2-29) を参照して下さい。

ADR 擬似命令や ADRL 擬似命令とは異なり、LDR 命令は現在のセクションに含まれないラベルに使用できます。ラベルが現在のセクションの範囲外にある場合には、アセンブラがソースファイルのアセンブリ時に再配置ディレクティブをオブジェクトコード内に配置します。この再配置ディレクティブは、リンク時にアドレスを解決するようにリンカに指示します。このアドレスは、リンカが LDR とリテラルプールを含むセクションをどこに配置しても有効です。

例 2-6 はこれを実装するコードを示しています。このサンプルは、主なサンプルディレクトリの *install_directory\RVDS\Examples* に *ldrlabel.s* という名前で収録されています。このサンプルをアセンブル、リンク、および実行する方法については、「サンプルコード」(P. 2-2) を参照して下さい。

コメントとして記載されている命令は、アセンブラによって生成される ARM 命令です。

例 2-6

	AREA	LDRlabel, CODE,READONLY	
	ENTRY		; Mark first instruction to execute
start	BL	func1	; Branch to first subroutine
	BL	func2	; Branch to second subroutine
stop	MOV	r0, #0x18	; angel_SWIreason_ReportException
	LDR	r1, =0x20026	; ADP_Stopped_ApplicationExit
	SVC	0x123456	; ARM semihosting (formerly SWI)
func1	LDR	r0, =start	; => LDR R0,[PC, #offset into ; Literal Pool 1]
	LDR	r1, =Darea + 12	; => LDR R1,[PC, #offset into ; Literal Pool 1]
	LDR	r2, =Darea + 6000	; => LDR R2, [PC, #offset into ; Literal Pool 1]
	BX	lr	; Return
	LTORG		; Literal Pool 1
func2	LDR	r3, =Darea + 6000	; => LDR r3, [PC, #offset into ; Literal Pool 1]
	; LDR	r4, =Darea + 6004	; (sharing with previous literal) ; If uncommented produces an error ; as Literal Pool 2 is out of range
	BX	lr	; Return
Darea	SPACE	8000	; Starting at the current location, ; clears a 8000 byte area of memory ; to zero ; Literal Pool 2 is out of range of ; the LDR instructions above
	END		

An LDR Rd, =label の使用例：文字列のコピー

例 2-7 は、ある 1 つの文字列を別の文字列で上書きする ARM コードルーチンを示しています。ここでは、LDR 擬似命令を使用してデータセクションから 2 つの文字列のアドレスをロードします。以下は特に重要です。

DCB DCB ディレクティブは、ストアする 1 バイト以上の値を定義します。DCB には整数値だけでなく、引用符で囲んだ文字列も使用できます。文字列の各文字は、連続したバイトに配置されます。詳細については、「DCB」(P. 7-23) を参照して下さい。

LDR、STR LDR 命令と STR 命令は、ポストインデクスアドレッシングを使用して、アドレスレジスタを更新します。例えば、以下の命令

```
LDRB    r2,[r1],#1
```

は、r1 が指すアドレスの内容を r2 にロードし、r1 を 1 ずつインクリメントします。

例 2-7 文字列のコピー

AREA	StrCopy, CODE, READONLY			
	ENTRY		;	Mark first instruction to execute
start	LDR	r1, =srcstr	;	Pointer to first string
	LDR	r0, =dststr	;	Pointer to second string
	BL	strcpy	;	Call subroutine to do copy
stop	MOV	r0, #0x18	;	angel_SWIreason_ReportException
	LDR	r1, =0x20026	;	ADP_Stopped_ApplicationExit
	SVC	0x123456	;	ARM semihosting (formerly SWI)
strcpy	LDRB	r2, [r1],#1	;	Load byte and update address
	STRB	r2, [r0],#1	;	Store byte and update address
	CMP	r2, #0	;	Check for zero terminator
	BNE	strcpy	;	Keep going if not
	MOV	pc,lr	;	Return
	AREA	Strings, DATA, READWRITE		
srcstr	DCB	"First string - source",0		
dststr	DCB	"Second string - destination",0		
	END			

2.7 多重レジスタロード / ストア命令

ARM、Thumb-2、およびThumbの命令セットには、メモリとの間で複数レジスタのロードとストアを実行する命令があります。

多重レジスタ転送命令を使用すると、複数レジスタの内容をメモリとの間で効率的に移動できます。これらの命令は、サブルーチンのエントリと終了時におけるブロックコピーやスタッック操作によく使用されます。複数の单一データ転送命令の代わりに多重レジスタ転送命令を使用すると、以下のような利点があります。

- コードサイズが小さくなる。
- 何度も命令フェッチするのではなく1回の命令フェッチで済むため、オーバヘッドが小さい。
- キャッシュなしARMプロセッサでは、多重ロード / ストア命令によって転送されるデータの先頭ワードは必ず非シーケンシャルメモリサイクルで処理されるが、その後に転送されるすべてのワードはシーケンシャルサイクルで処理することが可能。ほとんどのシステムでは、シーケンシャルメモリサイクルの方が実行速度が速くなります。

注

最も小さな番号のレジスタがアクセス先の最下位アドレスとの間で転送され、最も大きな番号のレジスタがアクセス先の最上位アドレスとの間で転送されます。命令のレジスラリスト内でのレジスタの順序は関係ありません。

レジスラリスト内のレジスタが昇順に指定されていることを確認するには、`--diag_warning 1206` アセンブリコマンドラインオプションを使用します。

このセクションでは、以下の内容について説明します。

- ARMとThumbで利用できる多重ロード / ストア命令 (P. 2-38)
- LDMとSTMによるスタッ�の実装 (P. 2-39)
- LDMおよびSTMによるブロックコピー (P. 2-41)

2.7.1 ARM と Thumb で利用できる多重ロード / ストア命令

ARM 命令セットと Thumb 命令セットでは、以下の命令が利用できます。

LDM	複数のレジスタをロードする。
STM	複数のレジスタをストアする。
PUSH	複数のレジスタをスタックにストアして、スタックポインタを更新する。
POP	複数のレジスタをスタックからロードして、スタックポインタを更新する。

LDM 命令と STM 命令を使用して以下を行えます。

- ロードまたはストアするレジスタのリストには以下を含めることができます。
 - ARM 命令では、任意またはすべての r0 ~ r15 を含めることができます。
 - 32 ビットの Thumb-2 命令では、任意またはすべての r0 ~ r12 を含めることができます。また、制限はありますが、オプションで r14 または r15 を含めることができます。
 - 16 ビットのThumb 命令と Thumb-2 命令では、任意またはすべての r0 ~ r7 を含めることができます。
- アドレスは、以下のように操作できます。
 - 転送単位でポストインクリメント
 - 転送単位でプレインクリメント (ARM 命令のみ)
 - 転送単位でポストデクリメント (ARM 命令のみ)
 - 転送単位でプレデクリメント (16 ビットのThumb ではサポートされません)
- ベースレジスタに対しては以下のいずれかの操作を実行できます。
 - メモリ内の次のブロックを指すように更新する。
 - 命令前の状態を維持する (16 ビットのThumb ではサポートされません)

メモリ内の次のブロックを指すようにベースレジスタを更新することは、ライトバックと呼ばれます。つまり、調整されたアドレスがベースレジスタに書き込まれます。

PUSH 命令と POP 命令では、以下のようにになります。

- スタックポインタ (r13) がベースレジスタとして使用され、常に更新されます。
- POP 命令では、アドレスは転送単位でポストインクリメントされます。また、PUSH 命令では、アドレスは転送単位でプレデクリメントされます。
- ロードまたはストアするレジスタのリストには以下を含めることができます。
 - ARM 命令では、任意またはすべての r0 ~ r15 を含めることができます。
 - 32 ビットの Thumb-2 命令では、任意またはすべての r0 ~ r12 を含めることができます。また、制限はありますが、オプションで r14 または r15 を含めることができます。
 - 16 ビットのThumb-2 命令と Thumb 命令では、任意またはすべての r7 ~ r15 を含めることができます。また、オプションで r14 (PUSH 命令の場合のみ) または r15 (POP 命令の場合のみ) を含めることができます。

2.7.2 LDM と STM によるスタックの実装

多重ロード / ストア命令によって、ベースレジスタを更新できます。通常、スタック操作では、ベースレジスタにはスタックポインタ r13 が使用されます。つまり、これらの命令を使用すると、1 つの命令で任意の数のレジスタに対してプッシュ操作と ポップ操作を実装することができます。

多重ロード / ストア命令は、以下のタイプのスタックに使用できます。

下降または上昇

スタックには、上位アドレスから下位アドレスに下に向かって展開される下降スタックと、下位アドレスから上位アドレスに向かって展開される上昇スタックがあります。

フルまたは空

スタックポインタは、スタック内の最後の項目（フルスタック）またはスタックの次の空き領域（空スタック）のどちらかを指すことができます。

プレ/ポストインクリメントやプレ/ポストデクリメント接尾文字を使用する代わりに、スタック指向の接尾文字を使用することで、プログラミングがさらに容易になります。スタック指向の接尾文字については、表 2-9 を参照して下さい。

表 2-9 多重レジスタロード / ストア命令の接尾文字

スタックのタイプ	プッシュ	ポップ
フル下降	STMFD (STMDB、プレデクリメント)	LDMFD (LDM、ポストインクリメント)
フル上昇	STMFA (STMIIB、プレインクリメント)	LDMFA (LDMIB、ポストデクリメント)
空下降	STMED (STMDA、ポストデクリメント)	LDMED (LDMIB、プレインクリメント)
空上昇	STMEA (STM、ポストインクリメント)	LDMEA (LDMDB、プレデクリメント)

例 :

```
STMFD    r13!, {r0-r5} ; Push onto a Full Descending Stack
LDMFD    r13!, {r0-r5} ; Pop from a Full Descending Stack
```

注

ARM アーキテクチャ向けプロシージャコール標準 (AAPCS)、および ARM と Thumb の C/C++ コンパイラでは、常にフル下降スタックを使用します。

PUSH 命令と POP 命令では、フル下降スタックが使用されることが想定されます。これらは、ライトバックを使用する STMDB 命令と LDM に適した同義語です。

ネストされたサブルーチンのレジスタのスタック

スタック操作は、サブルーチンのエントリと終了時で行うと効果的です。サブルーチンのエントリで必要なすべての作業レジスタをスタックにストアし、終了時にスタックからポップすることができます。

また、サブルーチンのエントリでリンクレジスタをスタックにプッシュしておけば、復帰アドレスを失わずに別のサブルーチンコールを安全に実行できます。この場合は、lr をポップして、その値を pc に移動する代わりに、サブルーチンの終了時にスタックから pc をポップすることによってサブルーチンから戻ることができます。以下に例を示します。

```
subroutine PUSH    {r5-r7,lr} ; Push work registers and lr
; code
BL      somewhere_else
; code
POP    {r5-r7,pc} ; Pop work registers and pc
```

注

ARM/Thumb 混合システムでこの方法を使用する場合は注意する必要があります。ARMv4T システムでは、プログラムカウンタを直接ポップしても状態を変更することはできません。このような場合は、アドレスを一時レジスタにポップして、BX 命令を使用する必要があります。

ARMv5T 以上では、この方法で状態を変更できます。

ARM と Thumb の混合については、*RealView Compilation Tools v3.0 Developer Guide* の「ARM と Thumb のインターワーク」を参照して下さい。

2.7.3 LDM および STM によるブロックコピー

例 2-8 は、一度に 1 ワードずつコピーすることにより、ソース位置の一連のワードをデスティネーションにコピーする ARM コードルーチンを示しています。このサンプルは、主なサンプルディレクトリの `install_directory\RVDS\Examples` に `word.s` という名前で収録されています。このサンプルをアセンブル、リンク、および実行する方法については、「サンプルコード」(P. 2-2) を参照して下さい。

例 2-8 LDM および STM によるブロックコピー

	AREA	Word, CODE, READONLY	; name this block of code
num	EQU	20	; set number of words to be copied
	ENTRY		; mark the first instruction called
start	LDR	r0, =src	; r0 = pointer to source block
	LDR	r1, =dst	; r1 = pointer to destination block
	MOV	r2, #num	; r2 = number of words to copy
wordcopy	LDR	r3, [r0], #4	; load a word from the source and
	STR	r3, [r1], #4	; store it to the destination
	SUBS	r2, r2, #1	; decrement the counter
	BNE	wordcopy	; ... copy more
stop	MOV	r0, #0x18	; angel_SWIreason_ReportException
	LDR	r1, =0x20026	; ADP_Stopped_ApplicationExit
	SVC	0x123456	; ARM semihosting (formerly SWI)
	AREA	BlockData, DATA, READWRITE	
src	DCD	1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4	
dst	DCD	0,0	
	END		

このモジュールは、できるだけ多くのコピーできる部分に LDM 命令と STM 命令を使用することで、効率化できます。この ARM コードで指定されているレジスタの数から、一度に転送するワード数は 8 ワードが妥当です。コピーするブロックに 8 ワード単位のブロックがいくつあるかは、以下を使用して検出できます (`r2` = コピーするワード数の場合)。

```
MOVS r3, r2, LSR #3 ; number of eight word multiples
```

この値を使用して、一度に 8 ワードずつコピーするループ内の繰り返しの回数を制御できます。残りのワード数が 8 ワードに満たない場合は、以下を使用して残りのワード数を検出できます (`r2` が破損していないことを前提とした場合)。

```
ANDS r2, r2, #7
```

例 2-9 (2-42 ページ) は、LDM 命令と STM 命令を使用してコピーするように書き直した ブロックコピー モジュールを示しています。

例 2-9 LDM および STM によるブロックコピー

num	AREA Block, CODE, READONLY	; name this block of code
	EQU 20	; set number of words to be copied
	ENTRY	; mark the first instruction called
start	LDR r0, =src	; r0 = pointer to source block
	LDR r1, =dst	; r1 = pointer to destination block
	MOV r2, #num	; r2 = number of words to copy
	MOV sp, #0x400	; Set up stack pointer (r13)
blockcopy	MOVS r3, r2, LSR #3	; Number of eight word multiples
	BEQ copywords	; Less than eight words to move?
	PUSH {r4-r11}	; Save some working registers
octcopy	LDM r0!, {r4-r11}	; Load 8 words from the source
	STM r1!, {r4-r11}	; and put them at the destination
	SUBS r3, r3, #1	; Decrement the counter
	BNE octcopy	; ... copy more
	POP {r4-r11}	; Don't need these now - restore
		; originals
copywords	ANDS r2, r2, #7	; Number of odd words to copy
	BEQ stop	; No words left to copy?
wordcopy	LDR r3, [r0], #4	; Load a word from the source and
	STR r3, [r1], #4	; store it to the destination
	SUBS r2, r2, #1	; Decrement the counter
	BNE wordcopy	; ... copy more
stop	MOV r0, #0x18	; angel_SWIreason_ReportException
	LDR r1, =0x20026	; ADP_Stopped_ApplicationExit
	SVC 0x123456	; ARM semihosting (formerly SWI)
src	AREA BlockData, DATA, READWRITE	
	DCD 1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4	
dst	DCD 0,0	
	END	

2.8 マクロの使用

マクロ定義は、MACRO ディレクティブと MEND ディレクティブの間にあるコードブロックです。これによってコードブロック全体を繰り返す代わりに使用できる名前が定義されます。マクロを使用する主な用途は以下のとおりです。

- コードブロックを1つの分かりやすい名前に置き換えることにより、ソースコードのロジックを追いややすくする。
- コードブロックの繰り返しを防ぐ。

詳細については、「*MACRO、MEND*」(P. 7-33) を参照して下さい。

このセクションでは、以下の内容について説明します。

- *Test-and-Branch* (テスト - 分岐) マクロのサンプル
- 符号なし整数除算マクロのサンプル (P. 2-44)

2.8.1 Test-and-Branch (テスト - 分岐) マクロのサンプル

Thumb-2 に対応していないプロセッサで実行する ARM コードと Thumb コードで、Test-and-Branch (テスト - 分岐) 操作を行うには2つの ARM 命令を実装する必要があります。

以下のようなマクロを定義できます。

```
MACRO
$label TestAndBranch $dest, $reg, $cc
$label CMP      $reg, #0
B$cc    $dest
MEND
```

MACRO ディレクティブの次の行は、マクロプロトタイプステートメントです。このステートメントでは、マクロの呼び出しに使用する名前 (TestAndBranch) を定義します。また、このステートメントではパラメータ (\$label、\$dest、\$reg、および \$cc) も定義しています。マクロを呼び出す場合は、これらのパラメータに値を指定する必要があります。アセンブラーは、コード内で指定された値でパラメータを置き換えます。

このマクロは、以下のように呼び出すことができます。

```
test   TestAndBranch NonZero, r0, NE
...
...
NonZero
```

代入後には以下のようになります。

```
test   CMP      r0, #0
BNE   NonZero
...
...
NonZero
```

2.8.2 符号なし整数除算マクロのサンプル

例 2-10 は、符号なし整数を除算するマクロを示しています。このマクロは、以下の 4 つのパラメータを取ります。

- \$Bot 除数を保持するレジスタ。
- \$Top 命令の実行前に被除数を保持し、命令の実行後に余数を保持するレジスタ。
- \$Div 除算の商が返されるレジスタ。余数のみが必要な場合は、NULL ("") を指定できます。
- \$Temp 計算中に使用される一時レジスタ。

例 2-10

```

MACRO
$Lab DivMod $Div,$Top,$Bot,$Temp
    ASSERT $Top <> $Bot           ; Produce an error message if the
    ASSERT $Top <> $Temp          ; registers supplied are
    ASSERT $Bot <> $Temp          ; not all different
    IF     "$Div" <> ""
        ASSERT $Div <> $Top         ; These three only matter if $Div
        ASSERT $Div <> $Bot         ; is not null ("")
        ASSERT $Div <> $Temp         ;
    ENDIF
$Lab
    MOV    $Temp, $Bot            ; Put divisor in $Temp
    CMP    $Temp, $Top, LSR #1    ; double it until
90     MOVLs $Temp, $Temp, LSL #1 ; 2 * $Temp > $Top
    CMP    $Temp, $Top, LSR #1    ;
    BLS    %b90                  ; The b means search backwards
    IF     "$Div" <> ""          ; Omit next instruction if $Div is null
        MOV    $Div, #0             ; Initialize quotient
    ENDIF
91     CMP    $Top, $Temp          ; Can we subtract $Temp?
    SUBCS $Top, $Top,$Temp        ; If we can, do so
    IF     "$Div" <> ""          ; Omit next instruction if $Div is null
        ADC    $Div, $Div, $Div    ; Double $Div
    ENDIF
    MOV    $Temp, $Temp, LSR #1    ; Halve $Temp,
    CMP    $Temp, $Bot             ; and loop until
    BHS    %b91                  ; less than divisor
MEND

```

このマクロは、複数のパラメータが同一レジスタを使用していないかどうかをチェックします。また、除余だけが要求されている場合には、生成されるコードを最適化します。

アセンブリソースで複数の `DivMod` を使用している場合にラベルの多重定義を防ぐために、このマクロではローカルラベル（90、91）を使用します。詳細については、「ローカルラベル」（P.2-12）を参照して下さい。

例 2-11 は、上記のマクロを以下のように呼び出したときに生成されるコードを示しています。

```
ratio  DivMod  r0,r5,r4,r2
```

例 2-11

```

        ASSERT r5 <> r4          ; Produce an error if the
        ASSERT r5 <> r2          ; registers supplied are
        ASSERT r4 <> r2          ; not all different
        ASSERT r0 <> r5          ; These three only matter if $Div
        ASSERT r0 <> r4          ; is not null ("")
        ASSERT r0 <> r2          ;
ratio
        MOV    r2, r4              ; Put divisor in $Temp
        CMP    r2, r5, LSR #1     ; double it until
90      MOVLS  r2, r2, LSL #1     ; 2 * r2 > r5
        CMP    r2, r5, LSR #1     ;
        BLS    %b90               ; The b means search backwards
        MOV    r0, #0              ; Initialize quotient
91      CMP    r5, r2              ; Can we subtract r2?
        SUBCS r5, r5, r2          ; If we can, do so
        ADC    r0, r0, r0          ; Double r0
        MOV    r2, r2, LSR #1     ; Halve r2,
        CMP    r2, r4              ; and loop until
        BHS    %b91               ; less than divisor

```

2.9 シンボルバージョンの追加

ARM リンカは *Base Platform ABI for the ARM Architecture* [BPABI] に準拠し、GNU 拡張シンボルバージョン管理モデルをサポートします。

既存のシンボルにシンボルのバージョンを追加するには、同じアドレスでバージョンシンボルを定義する必要があります。シンボルのバージョンは以下のような形式になります。

- デフォルト以外のバージョンの場合 : *name@ver*
- デフォルトのバージョンの場合 : *name@@ver*

シンボルのバージョンは縦棒で囲む必要があります。

例えば、デフォルトのバージョンは以下のように定義します。

```
|my_versioned_symbol@ver2| ; Default version
my_asm_function PROC
    ...
    BX lr
    ENDP
```

デフォルト以外のバージョンは以下のように定義します。

```
|my_versioned_symbol@ver1| ; Non default version
my_old_asm_function PROC
    ...
    BX lr
    ENDP
```

RVCT のシンボルバージョン管理の詳細については、*RealView Compilation Tools v3.0 Linker and Utilities Guide* のシンボルへのアクセスに関する章を参照して下さい。

2.10 フレームディレクティブの使用

以下のいずれかを実行する場合、フレームディレクティブを使用して、コードがスタックを使用する方法を記述する必要があります。

- スタック展開を使用してアプリケーションをデバッグする。
- フラットプロファイリングまたはコールグラフプロファイリングを使用する。

これらのディレクティブの詳細については、「フレームディレクティブ」(P. 7-41) を参照して下さい。

アセンブラーは、フレームディレクティブを使用して、生成する ELF 形式のオブジェクトファイルに DWARF デバッグフレーム情報を挿入します。この情報は、デバッガがスタック展開とプロファイリングを行うときに必要です。スタックチェック修飾子の詳細については、*install_directory\Documentation\Specifications\...* にある *Procedure Call Standard for the ARM Architecture* (aapcs.pdf) を参照して下さい。

以下の点に注意して下さい。

- フレームディレクティブによって、アセンブラーが生成したコードに影響が及ぶことはありません。
- アセンブラーでは、フレームディレクティブの情報を実行された命令に対して検証しません。

2.11 アセンブリ言語に関する変更

表 2-10 は、RVCT v2.2 とそれ以降で利用できる ARM アセンブリ言語の違いを示しています。古い ARM 構文は、最新のリリースでも認識されます。

表 2-10 以前の ARM アセンブリ言語からの変更点

変更点	古い ARM 構文	適切な構文
LDM および STM のデフォルトのアドレッシングモードは、IA です。	LDMIA, STMIA	LDM, STM
ARM と Thumb では、フル下降スタックの操作に PUSH ニーモニックと POP ニーモニックを使用できます。	STMFD sp!, {reglist} LDMFD sp!, {reglist}	PUSH {reglist} POP {reglist}
ARM と Thumb では、ロテートのみを伴う（他の操作は伴わない）命令の LSL、LSR、ASR、ROR、および RRX 命令ニーモニックを使用できます。	MOV Rd, Rn, LSL shift MOV Rd, Rn, LSR shift MOV Rd, Rn, ASR shift MOV Rd, Rn, ROR shift MOV Rd, Rn, RRX	LSL Rd, Rn, shift LSR Rd, Rn, shift ASR Rd, Rn, shift ROR Rd, Rn, shift RRX Rd, Rn
プログラムカウンタ相対アドレッシングには、 <i>label</i> 形式を使用します。新しいコードでは <i>offset</i> 形式は使用しないで下さい。	LDR Rd, [pc, #offset]	LDR Rd, <i>label</i>
ダブルワードメモリアクセスに両方のレジスタを指定します。使用できるレジスタの組み合わせについては、規則に従う必要があります。	LDRD Rd, addr_mode	LDRD Rd, Rd2, addr_mode
{cond} を使用する場合は、これがすべての命令の最後の要素になります。	ADD{cond}S LDR{cond}SB	ADDS{cond} LDRSB{cond}
ARM コードと Thumb-2 コードでは、ARM の条件付き形式 {cond} と Thumb-2 の IT 命令の両方を使用することができます。アセンブラーは両者の整合性をチェックして、現在の命令セットに応じて適切なコードをアセンブルします。	ADDEQ r1, r2, r3 LDRNE r1, [r2, r3]	ITEQ E ADDEQ r1, r2, r3 LDRNE r1, [r2, r3]

また、RVCT v2.2 以降では、それ以前のアセンブラーと比較して柔軟性が向上しました（表 2-11 参照）。

表 2-11 要件の緩和

変更点	適切な構文	使用可能な構文
デスティネーションレジスタが最初のオペランドと同じ場合、2 本のレジスタ指定する形式の命令を使用できます。	ADD r1, r1, r3	ADD r1, r3

RVCT v2.2 以上では、ARM アセンブリ言語を使用して Thumb プロセッサ用のソースコードを記述できます。

Thumb-2 プロセッサ以前のプロセッサ用のThumb コードを記述している場合、そのプロセッサで使用できる命令のみを使用するように注意する必要があります。使用できない命令を使用すると、アセンブラーからエラーメッセージが返されます。

Thumb-2 プロセッサ用のThumb コードを記述している場合は、可能な限り 16 ビットの命令を使用してコードサイズを小さくすることができます。

表 2-12 は、Thumb アセンブリ言語と ARM アセンブリ言語の主な相違点を示しています。アセンブラーは、構文の前に CODE16 ディレクティブが使用されているか、またはソースファイルが --16 コマンドラインオプションでアセンブルされている場合にのみ古いThumb 構文を認識します。

表 2-12 Thumb の古い構文と新しい構文の相違点

相違点	古いThumb 構文	新しいThumb 構文
LDM および STM のデフォルトのアドレスシングモードは、IA です。	LDMIA, STMIA	LDM, STM
フラグを更新する命令では後置の S を使用する必要があります。これは、32 ビットのThumb-2 命令との競合を防止するために必要な変更でした。	ADD r1, r2, r3 SUB r4, r5, #6 MOV r0, #1 LSR r1, r2, #1	ADDS r1, r2, r3 SUBS r4, r5, #6 MOVS r0, #1 LSRS r1, r2, #1
ALU 命令では、3 本のレジスタを指定するのが適切な形式です（これには、デスティネーションレジスタが最初のオペランドと同じ場合も含まれます）。	ADD r7, r8 SUB r1, #80	ADD r7, r7, r8 SUBS r1, r1, #80
Rd と Rn がどちらも Lo レジスタの場合、MOV Rd, Rn は ADDS Rd, Rn, #0 として逆アセンブルされます。	MOV r2, r3 MOV r8, r9 CPY r0, r1 LSL r2, r3, #0	ADDS r2, r3, #0 MOV r8, r9 MOV r0, r1 MOVS r2, r3
NEG Rd, Rm は RSBS Rd, Rm, #0 として逆アセンブルされます。	NEG Rd, Rm	RSBS Rd, Rm, #0
NOP 命令は、MOV r8, r8 を置き換えます（可能な場合）。	- NOP	NOP MOV r8, r8

第3章

アセンブラーに関する参考情報

本章には、ARM® アセンブラーに関する一般的な参考情報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。

- コマンド構文 (P. 3-2)
- ソース行の形式 (P. 3-17)
- 定義済みのレジスタおよびコプロセッサ名 (P. 3-18)
- 組み込み変数および定数 (P. 3-19)
- シンボル (P. 3-21)
- 式、リテラル、演算子 (P. 3-27)
- 診断メッセージ (P. 3-40)
- C プリプロセッサを使用する (P. 3-41)

本章では、ARM アセンブリ言語の記述方法についての説明は割愛いたします。チュートリアル情報については、「*第2章 ARM アセンブリ言語の記述*」を参照して下さい。

本章では、命令、ディレクティブ、および擬似命令についての説明も割愛いたします。これらの参照情報については、該当する各章を参照して下さい。

3.1 コマンド構文

このセクションでは、`armasm`についてのみ説明します。オンラインアセンブラーは、C および C++ コンパイラに属するアセンブラーであり、独自のコマンド構文を持ちません。

`armasm` コマンドラインでは、ファイル名や指定されている場合を除き、大文字と小文字の区別があります。

ARM アセンブラーは、次のコマンドを使用して起動します。

`armasm options inputfile`

ここで、`options` は、以下のオプションを任意に組み合わせて指定します。各オプションはスペースで区切って下さい。

`--16` 古いThumb構文を使用して、命令をThumb[®]命令として解釈するようにアセンブラーに指示します。このオプションは、ソースファイルの冒頭のCODE16ディレクティブに対応します。新しい構文を使用してThumb命令またはThumb-2命令を指定するには、`--thumb`オプションを使用します。

`--32` ARM命令として命令を解釈するようにアセンブラーに指示します。これがデフォルトです。

`--apcs [qualifiers]`

ARMアーキテクチャのプロシージャコール標準(AAPCS)を使用するかどうかを指定します。コードセクションの属性を一部指定することもできます。詳細については、「AAPCS」(P. 3-7)を参照して下さい。

`--arm` `--32`の同義語です。

`--bigend` ビッグエンディアンARMプロセッサに適したコードをアセンブルするようにアセンブラーに指示します。デフォルトは`--littleend`です。

`--brief_diagnostics`

「診断メッセージの出力を制御する」(P. 3-13)を参照して下さい。

`--littleend` リトルエンディアンARMプロセッサに適したコードをアセンブルするようにアセンブラーに指示します。

`--checkreglist`

RLIST、LDM、およびSTMレジスタリストをチェックして、すべてのレジスタが昇順のレジスタ番号順に指定されていることを確認するようにアセンブラーに指示します。レジスタが昇順で指定されていない場合は警告が表示されます。

このオプションは現在使用が制限されていて、今後のリリースで廃止される予定です。代わりに`--diag_warning 1206`を使用して下さい（「診断メッセージの出力を制御する」(P. 3-13)を参照）。

- cpu *name* ターゲットの CPU を設定します。「CPU 名」(P. 3-9) を参照して下さい。
- debug DWARF デバッグテーブルを生成するようにアセンブラーに指示します。
--debug は -g の同義語です。
デフォルトは DWARF 3 です。

——注——

ローカルシンボルは、--debug では保持されなくなりました。デバッグのためにローカルシンボルを保持する場合は、--keep を指定する必要があります。

--depend *dependfile*

ソースファイルの依存関係リストを *dependfile* に保存するようにアセンブラーに指示します。これらは make ユーティリティとともに使用すると便利です。

--diag_[error | remark | warning | suppress | style]

「診断メッセージの出力を制御する」(P. 3-13) を参照して下さい。

--dllexport_all

特に指定のない限り、すべてのグローバルシンボルを動的に可視化するようにアセンブラーに指示します。DLL を構築する場合にこのオプションを使用します。

--dwarf2 --debug とともに使用して DWARF 2 デバッグテーブルを生成するようにアセンブラーに指示します。

--dwarf3 --debug とともに使用して DWARF 3 デバッグテーブルを生成するようにアセンブラーに指示します。--debug が指定されている場合、これがデフォルトになります。

-g --debug の同義語です。

-m ソースファイルの依存関係リストを *stdout* に書き込むようにアセンブラーに指示します。

--md ソースファイルの依存関係リストを *inputfile.d* に書き込むようにアセンブラーに指示します。

--errors *errorfile*

エラーメッセージを *errorfile* に出力するようにアセンブラーに指示します。

--exceptions 「例外テーブル生成を制御する」(P. 3-15) を参照して下さい。

--exceptions_unwind

「例外テーブル生成を制御する」(P. 3-15) を参照して下さい。

--fpemode mode

浮動小数点適合性を指定し、ライブラリの属性と浮動小数点の最適化を設定します。「浮動小数点モデル」(P. 3-8) を参照して下さい。

--fpu name ターゲットの浮動小数点ユニット (FPU) アーキテクチャを選択します。*[FPU名]* (P. 3-10) を参照して下さい。**-i dir [,dir]...**

ソースファイルにディレクトリを追加して完全修飾名を付けます (*/GET, INCLUDE*) (P. 7-76) を参照。

--keep デバッガで使用できるように、オブジェクトファイルのシンボルテーブル内にローカルラベルを保持するようにアセンブラーに指示します (*/KEEP*) (P. 7-80) を参照。**--list file** アセンブラーによって生成されたアセンブリ言語による詳細なリストを *file* に出力するようにアセンブラーに指示します。詳細については、「リストをファイルに出力する」(P. 3-12) を参照して下さい。**--maxcache n** 最大ソースキャッシュサイズを *n* バイトに設定します。デフォルトは 8MB です。armasm では、サイズが 8MB を下回ると警告が表示されます。**--memaccess attributes**

ターゲットメモリシステムのメモリアクセス属性を指定します。「メモリアクセス属性」(P. 3-11) を参照して下さい。

注

--memaccess オプションは現在使用が制限されており、今後のリリースで廃止される予定です。

--no_esc \n や \t など、C 形式のエスケープ処理した特殊文字を無視するようにアセンブラーに指示します。**--no_exceptions**

「例外テーブル生成を制御する」(P. 3-15) を参照して下さい。

--no_exceptions_unwind

「例外テーブル生成を制御する」(P. 3-15) を参照して下さい。

--no_hide_all

SVr4 共有オブジェクトの作成時にシンボルの可視化を制御できます。エクスポートされたすべての定義および参照には動的な可視性が与えられます (*/EXPORT, GLOBAL*) (P. 7-71) を参照)。

--no_regs	レジスタ名を事前定義しないようにアセンブラーが設定されます。定義済みのレジスタ名のリストについては、「定義済みのレジスタおよびコプロセッサ名」(P. 3-18) を参照して下さい。 このオプションは現在使用が制限されていて、今後のリリースで廃止される予定です。代わりに <code>--regnames=none</code> を使用して下さい。
--no_warn	警告メッセージをオフにします。
-o <i>filename</i>	出力オブジェクトファイルに名前を付けます。このオプションが指定されていない場合、アセンブラーは、 <i>inputfilename.o</i> という形式でオブジェクトのファイル名を作成します。
<u>--predefine "directive"</u>	いざれかの SET ディレクティブを事前実行するようにアセンブラーに指示します。詳細については、「SET ディレクティブを事前に実行する」(P. 3-11) を参照して下さい。
--regnames=none	レジスタ名を事前定義しないようにアセンブラーが設定されます。定義済みのレジスタ名のリストについては、「定義済みのレジスタおよびコプロセッサ名」(P. 3-18) を参照して下さい。
--regnames=callstd	--apcs オプションで指定された、使用中の AAPCS バリアントに基づいて追加のレジスタ名を定義します（詳細については、「AAPCS」(P. 3-7) を参照）。
--regnames=all	--apcs の値に関係なくすべての AAPCS レジスタを定義します（詳細については、「AAPCS」(P. 3-7) を参照）。
--show_cmdline	アセンブラーがコマンドラインをどのように処理したのかを表示します。コマンドは正規化されて表示されます。また、via ファイルの内容は展開されます。
--split_ldm	長い LDM および STM 命令をエラーにするようにアセンブラーに指示します。詳細については、「長い LDM と STM を分割する」(P. 3-12) を参照して下さい。
--thumb	ARM 構文を使用して、Thumb 命令として命令を解釈するようにアセンブラーに指示します。このオプションは、ソースファイルの冒頭の THUMB ディレクティブに対応します。

--unaligned_access

オブジェクトファイルにおいて、非境界整列アクセスの使用を指示する属性を設定するようにアセンブラーに指示します。

--unsafe さまざまなアーキテクチャからの命令をエラーなしでアセンブルします（「診断メッセージの出力を制御する」（P. 3-13）を参照）。

--untyped_local_labels

Thumb コード内のラベルへの参照時に Thumb ビットを設定しないようアセンブラーに指示します。詳細については、「*LDR 擬似命令*」（P. 4-160）を参照して下さい。

--via file fileを開き、コマンドライン引数を読み込むようにアセンブラーに指示します。詳細については、*RealView Compilation Tools v3.0 Compiler and Libraries Guide* の付録「*via* ファイルの構文」を参照して下さい。

inputfile アセンブラーの入力ファイルを指定します。入力ファイルは、ARM アセンブリ言語ソースファイル、または Thumb アセンブリ言語ソースファイルを指定する必要があります。

3.1.1 利用可能なオプションのリストを取得する

利用可能なアセンブラーコマンドラインオプションのリストを取得するには、次のコマンドを入力します。

```
armasm --help
```

3.1.2 環境変数を使用してコマンドラインオプションを指定する

コマンドラインオプションは、RVCT30_ASMOPT 環境変数の値を設定することによって指定できます。この環境変数の構文は、コマンドライン構文と同じです。アセンブラーは、RVCT30_ASMOPT の値を読み込み、コマンド文字列の前に挿入します。これにより、RVCT30_ASMOPT で指定されたオプションは、コマンドラインの引数によってオーバーライド可能になります。

3.1.3 AAPCS

ARM アーキテクチャのプロシージャコール標準 (AAPCS) は、*ARM* アーキテクチャのアプリケーションバイナリインターフェース（基本規格）[BSABI] 仕様の一部です。AAPCS に準拠したコードを記述すると、別々にコンパイルおよびアセンブルしたモジュールを連動させることができます。

--apcs オプションで、AAPCS を使用するかどうかを指定します。コードセクションの属性を一部指定することもできます。

詳細については、*install_directory\Documentation\Specifications\...* にある *Procedure Call Standard for the ARM Architecture (aapcs.pdf)* を参照して下さい。

――注――

アセンブラーにより生成されたコードは、AAPCS 修飾子の影響を受けません。AAPCS 修飾子は、*inputfile* 内のコードが AAPCS の特定のバリアントに準拠していることをプログラマが確認するためのアサートです。AAPCS 修飾子により、アセンブラーによって生成されたオブジェクトファイル内で属性が設定されます。リンクはこれらの属性を使用してファイルの互換性をチェックし、適切なライブラリバリアントを選択します。

qualifier の値は以下のとおりです。

none	<i>inputfile</i> で AAPCS が使用されないように指定します。AAPCS レジスタは設定されません。 none を使用する場合、他の修飾子は使用できません。
/interwork	<i>inputfile</i> のコードを ARM/Thumb インターワークに適したコードにアセンブルするように指定します。インターワークの詳細については、 <i>RealView Compilation Tools v3.0 Developer Guide</i> を参照して下さい。
/nointerwork	<i>inputfile</i> のコードを ARM/Thumb インターワークに適したコードにアセンブルするように指定します。これがデフォルトです。
/ropi	<i>inputfile</i> の内容を、読み出し専用の位置非依存とするように指定します。
/noropi	<i>inputfile</i> の内容を、読み出し専用の位置非依存としないように指定します。これがデフォルトです。
/pic	/ropi の同義語です。
/nopic	/noropi の同義語です。
/rwpi	<i>inputfile</i> の内容を、読み出し - 書き込み可能な位置非依存とするように指定します。

/norwpi	<i>inputfile</i> の内容を、読み出し・書き込み可能な位置非依存としないように指定します。これがデフォルトです。
/pid	/rwp <i>i</i> の同義語です。
/nopid	/norwpi の同義語です。
/fpic	<i>inputfile</i> の内容を、FPIC アドレス指定を必要とする読み出し専用の位置非依存コードとするように指定します。
/adsabi	<i>inputfile</i> のコードが古い ADS アプリケーションバイナリインターフェース (ABI) との互換性があることを指定します。

——注——

--apcs /adsabi オプションは現在使用が制限されており、今後のリリースで廃止される予定です。

3.1.4 浮動小数点モデル

ARM アセンブラーには、浮動小数点モデルを指定するオプションがあります。

--femode *model*

ターゲットの浮動小数点モデルを選択し、リンク時に最適なライブラリを選択するように属性を設定します。

——注——

このオプションによって、記述したコードを変更する必要はありません。

model には、以下のいずれかを指定できます。

ieee_full IEEE 標準で保証されているすべての機能、演算、および表現を、単精度および倍精度で使用できます。演算モードは、実行時に動的に選択できます。

ieee_fixed 最近接値への丸め、不正確な例外なしの IEEE 標準。

ieee_no_fenv 最近接値への丸め、例外なしの IEEE 標準。このモードは、Java 浮動小数点算術モデルと互換性があります。

std	0 にフラッシュされる非正規化数があり、最近接値への丸め、例外なし IEEE 有限値。このモードは C および C++ と互換性があります。これはデフォルトオプションです。 有限値は、IEEE 標準によって予測されます。NaN (非数) および無限値は、IEEE モデルによって定義されている一部の状況で生成されない場合があります。また、生成されたとしても、記号が同じになるとは限りません。さらに、ゼロの記号が、IEEE モデルによって予測された記号でない場合もあります。
fast	ある数値が最適化を変える。この場合、高速実行の代わりに精度が犠牲になります。このモードは IEEE 互換でも、標準 C 形式でもありません。

3.1.5 CPU 名

CPU 名を指定するオプションがあります。

--cpu name ターゲットの CPU を設定します。命令によっては、間違ったターゲット CPU にアセンブルされると、エラーまたは警告が表示される場合があります（「診断メッセージの出力を制御する」(P. 3-13) も参照）。

name には、4T、5TE、6T2 などのアーキテクチャ名や、ARM7TDMI などの部品番号を指定できます。アーキテクチャの詳細については、ARM アーキテクチャリファレンスマニュアルを参照して下さい。デフォルトは ARM7TDMI® です。

リンク時のソフトウェアライブラリの選択に対する影響については、*RealView Compilation Tools v3.0 Compiler and Libraries Guide* を参照して下さい。

有効な CPU 名のリストを取得する

次のコマンドを使用してアセンブラーを起動すると、有効な CPU 名およびアーキテクチャ名のリストを取得できます。

```
armasm --cpu list
```

3.1.6 FPU 名

ARM アセンブラーには、FPU 名を指定するためのオプションがあります。

--fpu name ターゲットの浮動小数点ユニット (FPU) アーキテクチャを選択します。このオプションを指定すると、**--cpu** オプションによって設定された任意の暗示的 FPU がオーバライドされます。浮動小数点命令が間違ったターゲット FPU にアセンブルされると、エラーまたは警告が表示される場合があります。

アセンブラーによって、オブジェクトファイル内の *name* に対応するビルド属性が設定されます。オブジェクトファイル間の互換性と、選択されるライブラリはリンクによって決定されます。

name の有効値は以下のとおりです。

none 浮動小数点アーキテクチャを選択しません。これにより、アセンブルされたオブジェクトファイルとその他のすべてのオブジェクトファイルとの互換性を確保できます。

vfpv3 アーキテクチャ VFPv3 に適合する、ハードウェアのベクタ浮動小数点ユニットを選択します。

vfpv2 アーキテクチャ VFPv2 に適合する、ハードウェアのベクタ浮動小数点ユニットを選択します。

softvfp ソフトウェア浮動小数点リンクエージを選択します。**--fpu** オプションを指定せず、選択された **--cpu** オプションで特定の FPU を示唆していない場合は、これがデフォルトになります。

softvfp+vfpv2 ベクタ浮動小数点命令を使用するソフトウェア浮動小数点リンクエージが含まれた浮動小数点ライブラリを選択します。
--fpu vfpv2 を使用することと同等の機能があります。

softvfp+vfpv3 ベクタ浮動小数点命令を使用するソフトウェア浮動小数点リンクエージがある浮動小数点ライブラリを選択します。
これは、**--fpu vfpv3** を使用することと同等の機能があります。

リンク時におけるソフトウェアライブラリの選択に対する、これらの値の影響の詳細については、*RealView Compilation Tools v3.0 Compiler and Libraries Guide* の「C と C++ のライブラリ」の章を参照して下さい。

有効な FPU 名のリストを取得する

次のコマンドを使用してアセンブラーを起動すると、有効な FPU 名のリストを取得できます。

```
armasm --fpu list
```

3.1.7 メモリアクセス属性

ターゲットメモリシステムのメモリアクセス属性を指定するには、次のコマンドを使用します。

`--memaccess attributes`

デフォルトは、バイト、ハーフワード、ワードの境界整列ロードおよびストアのイネーブル化です。このデフォルトを変更するには、*attributes* を使用します。以下のいずれかの値を設定できます。

- | | |
|----------|-----------------------------|
| +L41 | 非境界整列 LDR をイネーブルにします。 |
| -L22 | ハーフワードロードをイネーブルにしません。 |
| -S22 | ハーフワードストアをイネーブルにしません。 |
| -L22-S22 | ハーフワードロードおよびストアをイネーブルにしません。 |

——注——

`--memaccess` オプションは現在使用が制限されており、今後のリリースで廃止される予定です。

3.1.8 SET ディレクティブを事前に実行する

次のオプションを使用すると、いずれかの SET ディレクティブを事前実行するようにアセンブラーに指示できます。

`--predefine "directive"`

directive は二重引用符で囲む必要があります。詳細については、「SETA、SETL、SETS」(P. 7-8) を参照して下さい。また、アセンブラーは、変数の値を設定する前に、対応する GBL、GBLS、または GBLA ディレクティブを実行して変数を定義します。

変数名では、大文字と小文字が区別されます。

——注——

使用しているシステムのコマンドラインインタフェースによっては、\" のように特殊文字を組み合わせたものを入力して、*directive* に文字列を含める必要があります。あるいは、`--via` ファイルを使用して `--predefine` 引数を含めることもできます。コマンドラインインタフェースでは `--via` ファイルからの引数を変更することはできません。

3.1.9 長い LDM と STM を分割する

多数のレジスタが含まれている LDM および STM 命令をエラーにするようにアセンブラーに指示するには、次のオプションを使用します。

`--split_ldm`

このオプションを指定すると、転送されたレジスタの最大数が次の数を超えた場合、LDM および STM 命令はエラーになります。

- 5つ（すべての STM の場合、および PC をロードしない LDM の場合）
- 4つ（PC をロードする LDM の場合）

多数レジスタの転送を回避すると、次のような ARM システムに対する割り込み遅延を減らすことができます。

- キャッシュや書き込みバッファのない ARM システム（キャッシュレス ARM7TDMI など）
- ゼロウェイト状態モード、32 ビットメモリを使用する ARM システム

また、多数レジスタの転送を回避すると、次の影響があります。

- コードサイズが増えます。
- キャッシュ付きシステムや、書き込みバッファ付きプロセッサには大きなメリットはありません。
- ゼロウェイト状態メモリを使用していないシステムや、低速なペリフェラルデバイスのあるシステムについてメリットはありません。このようなシステムの割り込み遅延は、最も速度の遅いメモリまたはペリフェラルデバイスへのアクセスについて必要となるサイクル数によって決まります。この割り込み遅延は通常、複数レジスタの転送によって誘発される遅延よりも大きくなります。

3.1.10 リストをファイルに出力する

リストをファイルに出力するには、次のオプションを使用します。

`--list file`

このオプションは、アセンブラーによって生成されたアセンブリ言語の詳細なリストを `file` に出力するようにアセンブラーに指示します。

`file` として - を指定すると、リストが `stdout` に送信されます。

`file` が指定されていない場合は、`--list=` を使用して、出力を `inputfile.lst` に送信します。

注

--list を使用すると、出力を *inputfile.lst* に送信できます。ただし、この構文は現在使用が制限されているため、この構文を使用するとアセンブラーによって警告メッセージが表示されます。この構文は、今後のリリースでは廃止されるので、代わりに --list= を使用して下さい。

--list の動作を制御するには、次のコマンドラインオプションを使用します。

--no_terse	<i>terse</i> フラグをオフにします。このオプションをオンにすると、条件アセンブリによりスキップされた行はリストには表示されません。 <i>terse</i> オプションをオフにすると、これらの行がリストに表示されます。デフォルトはオンです。
--width	リストするページの幅を設定します。デフォルトは 79 文字です。
--length	リストするページの長さを設定します。ゼロを指定した場合、ページ番号なしでリストされます。デフォルトは 66 行です。
--xref	マクロの内部と外部でシンボルが定義されている場所と使用されている場所などの相互参照情報をリストするようにアセンブラーが設定されます。デフォルトはオフです。

3.1.11 診断メッセージの出力を制御する

診断メッセージの出力を制御するためのオプションがいくつかあります。

--brief_diagnostics

簡単な診断出力形式を使用するモードをディセーブルまたはイネーブルします。イネーブルにすると、元のソース行は表示されず、1 行に收まらないエラーメッセージテキストはラップされずに表示されます。デフォルトは --no_brief_diagnostics です。

--diag_style {arm|ide|gnu}

診断メッセージを表示するための形式を指定します。

arm ARM アセンブラーの形式を使用してメッセージを表示します。
--diag_style が指定されていない場合は、これがデフォルトになります。

ide エラーのある行の行番号と文字数を表示します。これらの値は括弧に囲まれて表示されます。

gnu GNU 形式でメッセージを表示します。

--diag_error tag[, tag, ...]

指定されたタグを持つ診断メッセージにエラーの重大度を設定します (P. 3-15 表 3-1 を参照)。

--diag_remark *tag[, tag, ...]*

指定されたタグを持つ診断メッセージに注釈の重大度を設定します
(表 3-1 を参照)。

--diag_warning *tag[, tag, ...]*

指定されたタグを持つ診断メッセージに警告の重大度を設定します
(表 3-1 を参照)。

--diag_suppress *tag[, tag, ...]*

指定されたタグを持つすべての診断メッセージをディセーブルします。

--unsafe さまざまなアーキテクチャからの命令をエラーなしでアセンブルします。対応するエラーメッセージは警告メッセージに変更されます。演算子の優先順位に関する警告も非表示にされます (「2 項演算子」(P. 3-35) を参照)。

以上の --diag_ オプションのうち、4 つに、非表示にするメッセージの番号を表す *tag* を指定する必要があります。この場合、複数のタグを指定できます。例えば、1293 と 187 という番号を持つ警告メッセージを非表示にするには、次のコマンドを使用します。

```
armasm --diag_suppress 1293,187 ...
```

--diag_error、--diag_remark、および --diag_warning を指定したり、メッセージを非表示にしたりする場合は、次のようにしてアセンブラー接頭文字 A を使用できます。

```
armasm --diag_suppress A1293,A187 ...
```

診断メッセージは切り取り、コマンドラインに直接貼り付けることができます。接頭文字の使用は省略できます。ただし、接頭文字を含める場合は、armasm 識別文字と一致する必要があります。別の接頭文字が見つかると、アセンブラーはそのメッセージ番号を無視します。

表 3-1 で、オプションの説明で使用される重大度という用語のそれぞれの意味を説明します。

表 3-1 診断メッセージの重大度

重大度	説明
致命的エラー	アセンブリを停止しなければならないような問題を示します。これらのエラーには、コマンドラインエラー、内部エラー、インクリードファイルの欠落などがあります。複数のソースファイルがアセンブルされている場合は、ソースファイルのアセンブルが中止されます。
エラー	アセンブリ言語の構文ルールまたはセマンティックルールの違反を示します。アセンブリは引き続き実行されますが、オブジェクトコードは生成されません。
警告	コードに問題を引き起こす可能性のある例外的な状況を示す警告です。重大度のエラーが検出されない限り、アセンブラは引き続き実行され、オブジェクトコードが生成されます。
注釈	一般的であるが、推奨されないアセンブリ言語の慣用を示します。これらの診断情報はデフォルトでは表示されません。重大度のエラーが検出されない限り、アセンブラは引き続き実行され、オブジェクトコードが生成されます。

3.1.12 例外テーブル生成を制御する

例外テーブル生成を制御するオプションには以下の 4 つがあります。

--exceptions 遭遇したすべての関数について、例外テーブル生成をオンにするようにアセンブラに指示します。

--no_exceptions

例外テーブル生成をオフにするようにアセンブラに指示します。テーブルは生成されません。これがデフォルトです。

--exceptions_unwind

可能な場合は、関数について *unwind* テーブルを生成するようにアセンブラに指示します。これがデフォルトです。

--no_exceptions_unwind

すべての関数について *nounwind* テーブルを生成するようにアセンブラに指示します。

さらに細かく制御するには、FRAME UNWIND ON および FRAME UNWIND OFF ディレクティブを使用します。詳細については、「/FRAME UNWIND ON」(P. 7-53) と「/FRAME UNWIND OFF」(P. 7-53) を参照して下さい。

Unwind テーブル

関数は、PROC/ENDP または FUNC/ENDFUNC ディレクティブで囲まれたコードです。

例外は、*unwind* テーブルを使用して関数に伝播できます。アセンブラーは、デバッグフレーム情報からアンワインド情報を生成します。

例外は、*nounwind* テーブルを使用して関数に伝播できません。例外処理時に *nounwind* テーブルに遭遇した場合、例外処理ラインタイム環境によってプログラムが終了します。

アセンブラーでは、すべての関数および非関数について *nounwind* テーブルエントリを生成できます。関数内部でスタックの使用を記述する十分な FRAME ディレクティブが関数に含まれている場合にのみ、その関数の *unwind* テーブルをアセンブラーで生成できます。

関数は、*Exception Handling ABI for the ARM Architecture [EHABI]*、セクション 9.1 「*Constraints on Use*」で規定されている条件に準拠する必要があります。アセンブラーで *unwind* テーブルを生成できない場合は、*nounwind* テーブルを生成します。

3.2 ソース行の形式

以下は、ARMアセンブリ言語モジュールで記述するソース行の汎用形式です。

```
{symbol} {instruction|directive|pseudo-instruction} {;comment}
```

このソース行の3つのセクションはすべてオプションです。

冒頭からいきなり命令を開始することはできません。命令の前にシンボルがない場合でも、命令の先頭に空白を空ける必要があります。

本書で既に述べたとおり、ディレクティブはすべて大文字で記述したり、すべて小文字で記述したりできます。大文字と小文字を混在させてディレクティブを記述することはできません。

コードを読みやすくするために空白行を挿入してもかまいません。

*symbol*には通常、ラベルを指定します（「ラベル」(P. 3-24) および「ローカルラベル」(P. 3-25) を参照）。命令および擬似命令内ではこれが必ずラベルです。一部のディレクティブでは、これが変数または定数のシンボルです。詳細については、ディレクティブの説明を参照してください。

*symbol*は必ず最初の列から開始する必要があります。スペースやタブなどの空白文字を含めることはできません（「シンボルの命名ルール」(P. 3-21) を参照）。

3.3 定義済みのレジスタおよびコプロセッサ名

すべてのレジスタおよびコプロセッサ名では、大文字と小文字が区別されます。

3.3.1 事前宣言されているレジスタ名

以下のレジスタ名は事前宣言されています。

- r0-r15 および R0-R15
- a1-a4 (引数、結果、スクラッチレジスタ、r0 ~ r3 の同義語)
- v1-v8 (変数レジスタ、r4 ~ r11)
- sb および SB (スタティックベース、r9)
- ip および IP (内部プロシージャコードスクラッチレジスタ、r12)
- sp および SP (スタックポインタ、r13)
- lr および LR (リンクレジスタ、r14)
- pc および PC (プログラムカウンタ、r15)

3.3.2 事前宣言されているプログラムステータスレジスタ名

以下のプログラムステータスレジスタ名は事前宣言されています。

- cpsr および CPSR (現在のプログラムステータスレジスタ)
- spsr および SPSR (保存されているプログラムステータスレジスタ)。

3.3.3 事前宣言されている浮動小数点レジスタ名

以下の浮動小数点レジスタ名は事前宣言されています。

- s0-s31 および S0-S31 (VFP 単精度レジスタ)
- d0-d15 および D0-D15 (VFP 倍精度レジスタ)
- d16-d31 および D16-D31 (VFPv3 追加倍精度レジスタ)

3.3.4 事前宣言されている NEON レジスタ名

以下の NEON™ レジスタ名は事前宣言されています。

- d0-d31 および D0-D31 (VFP 倍精度レジスタ)
- q0-q15 および Q0-Q15 (クワッドワードレジスタ)

3.3.5 事前宣言されているコプロセッサ名

以下のコプロセッサ名およびコプロセッサレジスタ名は事前宣言されています。

- p0-p15 (コプロセッサ 0 ~ 15)
- c0-c15 (コプロセッサレジスタ 0 ~ 15)

3.4 組み込み変数および定数

表 3-2 に、ARM アセンブラーによって定義されている組み込み変数を記載します。

表 3-2 組み込み変数

{ARCHITECTURE}	選択された ARM アーキテクチャの名前を保持します。
{AREANAME}	現在の AREA の名前を保持します。
{ARMASM_VERSION}	armasm のバージョンごとに増加する整数を保持します。
ads\$version	{ARMASM_VERSION} と同じ値を保持します。
{CODESIZE}	{CONFIG} の同義語です。
{COMMANDLINE}	コマンドラインの内容を保持します。
{CONFIG}	アセンブラーが ARM コードをアセンブルしている場合は、値 32 を保持し、Thumb コードをアセンブルしている場合は値 16 を保持します。
{CPU}	選択されている cpu の名前を保持します。デフォルトは ARM7TDMI です。アーキテクチャがコマンドラインの --cpu オプションで指定されている場合、{CPU} は、値 "Generic ARM (汎用 ARM)" を保持します。
{ENDIAN}	アセンブラーがビッグエンディアンモードの場合は値 big を保持し、リトルエンディアンモードの場合は little を保持します。
{FPIC}	/fpic が設定されている場合は値 True を保持します。デフォルトは False です。
{FPU}	選択されている fpu の名前を保持します。デフォルトは SoftVFP です。
{INPUTFILE}	現在のソースファイルの名前を保持します。
{INTER}	/inter が設定されている場合は値 True を保持します。デフォルトは False です。
{LINENUM}	現在のソースファイル内の行番号を示す整数を保持します。
{OPT}	現在設定されているリストオプションの値。OPT ディレクティブを使用すると、現在のリストオプションを保存したり、リストオプションを変更したり、元の値を復元したりできます。
{PC} または	現在の命令のアドレス。
{PCSTOREOFFSET}	STR pc,[...] 命令または STM Rb,{..., pc} 命令のアドレスとストアアウトされた pc の値の間のオフセットです。この変数に保持される値は、指定された CPU またはアーキテクチャによって変わります。
{ROPI}	/ropi が設定されている場合は値 True を保持します。デフォルトは False です。
{RWPI}	/rwpi が設定されている場合は、値 True を保持します。デフォルトは False です。
{VAR} または @	ストレージ領域の位置カウンタの現在の値。

SETA、SETL、またはSETS ディレクティブを使用して組み込み変数を設定することはできません。組み込み変数は、次のように数式や条件式で使用できます。

```
IF {ARCHITECTURE} = "4T"
```

|ads\$version| はすべて小文字にする必要があります。その他の組み込み変数には大文字のみ、小文字のみ、または大文字と小文字を使用できます。

組み込み変数 {ARMASM_VERSION} を使用すると、armasm のバージョンを区別できます。

ADS 以前、ARM アセンブラーには、組み込み変数 {ARMASM_VERSION} が用意されていませんでした。古い開発ツールを使用してコードのバージョンを設定する場合は、組み込み変数 |ads\$version| を使用してテストを実施できます。次のコードを使用します。

```
IF :DEF: |ads$version|
    ; code for RVCT or ADS
ELSE
    ; code for SDT
ENDIF
```

表 3-3 に、ARM アセンブラーで定義されている組み込みブール定数を記載します。

表 3-3 組み込みブール定数

{FALSE}	論理定数 False
{TRUE}	論理定数 True

3.5 シンボル

シンボルを使用して、変数、アドレス、数値定数を表現できます。アドレスを表すシンボルはラベルとも呼ばれます。以下のセクションを参照して下さい。

- シンボルの命名ルール
- 変数 (P. 3-22)
- 数値定数 (P. 3-22)
- アセンブリ時の変数代入 (P. 3-22)
- ラベル (P. 3-24)
- ローカルラベル (P. 3-25)

3.5.1 シンボルの命名ルール

シンボル名には、以下の一般規則が適用されます。

- シンボル名には大文字、小文字、数値、下線付き文字を使用できます。
- ローカルラベルを除き、シンボル名の最初の文字に数値を使用しないで下さい (「ローカルラベル」 (P. 3-25) を参照)。
- シンボル名では大文字と小文字が区別されます。
- シンボル名に使用する文字はすべて有意です。
- シンボル名は、その有効範囲内で唯一である必要があります。
- 組み込み変数名または定義済みのシンボル名と同じシンボルを使用することはできません (「定義済みのレジスタおよびコプロセッサ名」 (P. 3-18) および 「組み込み変数および定数」 (P. 3-19) を参照)。
- シンボルには、命令ニーモニックまたはディレクティブと同じ名前を付けることはできません。命令ニーモニックまたはディレクティブと同じ名前を使用する場合は、次のように、複縦線を使用してシンボル名を区切って下さい。

`||ASSERT||`

縦線はシンボル名として解釈されることはありません。

シンボルに多様な文字を使用する場合は、コンパイラを操作するときなど、次のように、単縦線を使用してシンボル名を区切れます。

`|.text|`

縦線はシンボル名として解釈されることはありません。縦線内部でさらに縦線、セミコロン、改行を使用することはできません。

3.5.2 変数

変数の値は、アセンブリの実行中に変更できます。変数には以下の3つの型があります。

- 数値
- 論理
- 文字列

変数の型は変更できません。

数値変数の値の範囲は、数値定数または数値式の値の範囲と同じです（数値定数および「数値式」（P. 3-28）を参照）。

論理変数の値は、{TRUE} または {FALSE} です（「論理式」（P. 3-31）を参照）。

文字列変数の値の範囲は、文字列式の値の範囲と同じです（「文字列式」（P. 3-27）を参照）。

変数を表すシンボルを宣言するには、GBLA、G BLL、GBLS、LCLA、L CLL、および LCLS ディレクティブを使用します。これらの変数に値を割り当てるには、SETA、SETL、および SETS ディレクティブを使用します。以下のセクションを参照して下さい。

- GBLA、G BLL、GBLS (P. 7-5)
- LCLA、L CLL、LCLS (P. 7-7)
- SETA、SETL、SETS (P. 7-8)

3.5.3 数値定数

数値定数は 32 ビットの整数です。数値定数は、 $0 \sim 2^{32}-1$ の範囲の符号なし数値を使用して、または $-2^{31} \sim 2^3-1$ の範囲の符号あり数値を使用して設定できます。ただし、アセンブリでは、 $-n$ と $2^{32}-n$ は区別されません。 \geq などの関係演算子では、符号なしと解釈されます。その結果、 $0 > -1$ は {FALSE} として処理されます。

定数を定義するには、EQU ディレクティブを使用します（/EQU（P. 7-70）を参照）。定義後に数値定数の値を変更することはできません。

「数値式」（P. 3-28）および「数値リテラル」（P. 3-29）も参照して下さい。

3.5.4 アセンブリ時の変数代入

アセンブリ言語で記述されたコードの全行、またはコード行の一部に、文字列変数を代入することができます。変数の値が代入される場所に、\$接頭文字を付けた変数を挿入します。ドル記号を使用すると、行構文をチェックする前に、文字列をソースコード行に代入するようにアセンブリに指示します。

数値変数や論理変数も代入可能です。変数の現在の値は、代入前に 16 進数文字列に変換されます（論理変数の場合は T または F に変換されます）。

次の文字がシンボル名に使用できる場合は、変数名の末尾にドットを付けます（「シンボルの命名ルール」（P. 3-21）を参照）。変数を使用するには、変数の内容を設定する必要があります。

\$ という文字に値を代入しないようにするには、\$\$と指定します。これにより、1つの\$に変換されます。

文字列には、変数と\$接頭文字と一緒に含めることができます。代入は、他の場所と同じように行われます。

代入が二重引用符内部の縦線の影響を受けない場合を除き、縦線内で代入は行われません。

例

```
; straightforward substitution
    GBLS    add4ff
;
add4ff SETS    "ADD r4,r4,#0xFF"      ; set up add4ff
        $add4ff.00          ; invoke add4ff
;
; this produces
ADD r4,r4,#0xFF00

; elaborate substitution
    GBLS    s1
    GBLS    s2
    GBLS    fixup
    GBLA    count
;
count   SETA    14
s1      SETS    "a$$b$count" ; s1 now has value a$b0000000E
s2      SETS    "abc"
fixup   SETS    "|xy$s2.z|" ; fixup now has value |xyabcz|
|C$$code| MOV     r4,#16       ; but the label here is C$$code
```

3.5.5 ラベル

ラベルは、命令またはデータのメモリ内のアドレスを表すシンボルです。ラベルには、プログラム相対ラベル、レジスタ相対ラベル、または絶対アドレスがあります。

プログラム相対ラベル

プログラム相対ラベルは、数値定数あり、または数値定数なしの PC を表します。プログラム相対ラベルは、分岐命令のターゲットとして使用するか、またはコードセクションに組み込まれているデータの細目にアクセスするために使用します。プログラム相対ラベルは、命令でラベルを使用するか、またはいずれかのデータ定義ディレクティブでラベルを使用することによって定義できます。以下のセクションを参照して下さい。

- *DCB* (P. 7-23)
- *DCD*、*DCDU* (P. 7-24)
- *DCFD*、*DCF DU* (P. 7-26)
- *DCFS*、*DCFSU* (P. 7-27)
- *DCI* (P. 7-28)
- *DCQ*、*DCQU* (P. 7-29)
- *DCW*、*DCWU* (P. 7-30)

レジスタ相対ラベル

レジスタ相対ラベルは、数値定数が付記された名前付きレジスタを表します。レジスタ相対ラベルは、データセクション内のデータにアクセスするために頻繁に使用されます。レジスタ相対ラベルは、記憶域マップを使用して定義できます。EQU ディレクティブを使用すると、記憶域マップで定義されているラベルに基づいて追加のレジスタ相対ラベルを定義できます。以下のセクションを参照して下さい。

- *MAP* (P. 7-20)
- *SPACE* (P. 7-22)
- *DCDO* (P. 7-25)
- *EQU* (P. 7-70)

絶対アドレス

絶対アドレスは数値定数です。値の範囲は $0 \sim 2^{32}-1$ の整数です。絶対アドレスは、メモリを直接アドレス指定します。

3.5.6 ローカルラベル

ローカルラベルは範囲 0 ~ 99 の数値で、その後に名前を付けることもできます。ELF セクション内の複数のローカルラベルに対して同じ数値を使用できます。

ローカルラベルは、以下の方法で、アセンブリ言語モジュールのソース行内で *symbol* の代わりに使用できます（「ソース行の形式」(P. 3-17) を参照）。

- 命令やディレクティブと一緒に使用せずに単独で使用する。
- 命令を含む行の中で使用する。
- コード生成ディレクティブまたはデータ生成ディレクティブを含む行の中で使用する。

プログラム相対ラベルを使用する場所では通常、ローカルラベルが使用されます（「ラベル」(P. 3-24) を参照）。

ローカルラベルは一般に、ルーチン内部のループおよび条件コードや、ローカルのみで使用される小さいサブルーチンに対して使用されます。ローカルラベルは、特にマクロで使用すると便利です（「*MACRO*, *MEND*」(P. 7-33) を参照）。

ローカルラベルの有効範囲を制限するには、ROUT ディレクティブを使用します（「*ROUT*」(P. 7-84) を参照）。ローカルラベルに対する参照では、同じ有効範囲内で一致するラベルが参照されます。有効範囲内で一致するラベルがいずれの方向にも存在しない場合、アセンブリは、エラーメッセージを生成し、アセンブリは失敗します。

同じ有効範囲内の複数のローカルラベルについて同じ数字を使用できます。アセンブリはデフォルトで、以下のようにローカルラベル参照をリンクします。

- 有効範囲内にラベルが 1 つある場合は、同じ番号の最新のローカルラベルにリンクします。
- 有効範囲内に先行ラベルが存在しない場合は、同じ番号の次のローカルラベルにリンクします。

この検索パターンを必要に応じて変更するには、オプションパラメータを使用します。

構文

ローカルラベルの構文は次のとおりです。

n{ routname }

ローカルラベルの参照の構文は次のとおりです。

%{F|B}{A|T}n{ routname }

各パラメータには以下の意味があります。

n ローカルラベルの番号です。

routname 現在の有効範囲の名前です。

% 参照を導入します。

F 順方向検索だけを実行するようにアセンブラーに指示します。

B 逆方向検索だけを実行するようにアセンブラーに指示します。

A すべてのマクロレベルを検索するようにアセンブラーに指示します。

T このマクロレベルのみを検索するようにアセンブラーに指示します。

F も *B* も指定されていない場合、アセンブラーはまず逆方向に検索した後、順方向に検索します。

A も *T* も指定されていない場合、アセンブラーは、現在のレベルから最上位レベルまでのすべてのマクロを検索します。現在のレベルよりも下位のマクロは検索しません。

routname がラベルまたはラベルへの参照で指定されている場合、アセンブラーは、これを、直前の ROUT ディレクティブの名前と照合します。一致しない場合、アセンブラーは、エラーメッセージを生成し、アセンブリは失敗します。

3.6 式、リテラル、演算子

このセクションは以下のサブセクションから構成されています。

- 文字列式
- 文字列リテラル (P. 3-28)
- 数値式 (P. 3-28)
- 数値リテラル (P. 3-29)
- 浮動小数点リテラル (P. 3-30)
- レジスタ相対式とプログラム相対式 (P. 3-31)
- 論理式 (P. 3-31)
- 論理リテラル (P. 3-31)
- 演算子の優先順位 (P. 3-31)
- 単項演算子 (P. 3-33)
- 2 項演算子 (P. 3-35)

3.6.1 文字列式

文字列式は、文字列リテラル、文字列変数、文字列操作演算子、および括弧で構成されます。以下のセクションを参照して下さい。

- 変数 (P. 3-22)
- 文字列リテラル (P. 3-28)
- 単項演算子 (P. 3-33)
- 文字列操作演算子 (P. 3-36)
- SETA、SETL、SETS (P. 7-8)

文字列リテラル内に配置できない文字は、:CHR: 単項演算子を使用することによって文字列式内に配置できます。0 ~ 255 の ASCII 文字を使用できます。

文字列式の値の長さは 512 文字を超えないようにして下さい。長さは 0 でもかまいません。

例

```
improb SETS    "literal":CC:(strvar2:LEFT:4)
; sets the variable improb to the value "literal"
; with the left-most four characters of the
; contents of string variable strvar2 appended
```

3.6.2 文字列リテラル

文字列リテラルは、一連の文字列を二重引用符で囲んで構成します。文字列リテラルの長さは入力行の長さによって制限されます（「ソース行の形式」(P. 3-17) を参照）。

文字列内に二重引用符またはドル記号を含めるには、二重引用符またはドル記号を2文字使用します。

C言語形式の文字列エスケープシーケンスは、`--no_esc` が指定されない限りイネーブルになります（「コマンド構文」(P. 3-2) を参照）。

例

```
abc      SETS      "this string contains only one "" double quote"
def      SETS      "this string contains only one $$ dollar symbol"
```

3.6.3 数値式

数値式は、数値定数、数値変数、通常の数値リテラル、2項演算子、および括弧の組み合わせで構成されます。以下のセクションを参照して下さい。

- 数値定数 (P. 3-22)
- 変数 (P. 3-22)
- 数値リテラル (P. 3-29)
- 2項演算子 (P. 3-35)
- *SETA*、*SETL*、*SETS* (P. 7-8)

式全体が、レジスタまたはPCを含まない値に評価される場合、数値式にはレジスタ相対式またはプログラム相対式を含めることができます。

数値式は32ビット整数に評価されます。これらは、 $0 \sim 2^{32}-1$ の範囲の符号なし数値として、または $-2^{31} \sim 2^{31}-1$ の範囲の符号あり数値として解釈できます。ただし、アセンブラーは、 $-n$ と $2^{32}-n$ を区別しません。 \geq などの関係演算子では、符号なしとして解釈されます。その結果、 $0 > -1$ は {FALSE} として処理されます。

例

```
a      SETA      256*256          ; 256*256 is a numeric expression
      MOV       r1,#(a*22)        ; (a*22) is a numeric expression
```

3.6.4 数値リテラル

数値リテラルには、以下のいずれかの形式を使用できます。

decimal-digits
0xhexadecimal-digits
&hexadecimal-digits
n_base-n-digits
'character'

各パラメータには以下の意味があります。

<i>decimal-digits</i>	0～9 の数値だけを使用した文字列です。
<i>hexadecimal-digits</i>	0～9 の数字と A～F または a～f だけを使用した文字列です。
<i>n_</i>	2～9 のいずれかの数字に下線付き文字を続けた文字列です。
<i>base-n-digits</i>	0～(n-1) の数字のみを使用した文字列です。
<i>character</i>	单一引用符を除く任意の 1 文字です。单一引用符を表示する場合は、' を使用します。この場合、数値リテラルの値は文字の数値コードです。

上記以外の文字は使用できません。文字列は、0～ $2^{32}-1$ の整数に評価される必要があります（範囲が 0～ $2^{64}-1$ である DCQ および DCQU ディレクティブを除く）。

例

```
a      SETA    34906
addr   DCD     0xA10E
        LDR     r4,=&1000000F
        DCD     2_11001010
c3     SETA    8_74007
        DCQ     0x0123456789abcdef
        LDR     r1,'A'           ; pseudo-instruction loading 65 into r1
        ADD     r3,r2,#'\''       ; add 39 to contents of r2, result to r3
```

3.6.5 浮動小数点リテラル

浮動小数点リテラルには、以下のいずれかの形式を使用できます。

```
{-}digitsE{-}digits
{-}{digits}.digits{E{-}digits}
0xhexdigits
&hexdigits
```

各パラメータには以下の意味があります。

digits 0～9の数字のみを使用した文字列です。Eは大文字または小文字で記述できます。これらの形式は通常の浮動小数点表記に対応します。

hexdigits 0～9の数字とA～Fまたはa～fの文字のみを使用した文字列です。これらの形式は、コンピュータにおける数値の内部表現に対応します。無限値およびNaNを入力する場合、または使用する正しいビットパターンが不明な場合は、これらの形式を使用します。

単精度浮動小数点数値の範囲は以下のとおりです。

- 最大 : 3.40282347e+38
- 最小 : 1.17549435e-38

倍精度浮動小数点数値の範囲は以下のとおりです。

- 最大 : 1.79769313486231571e+308
- 最小 : 2.22507385850720138e-308

例

DCFD	1E308,-4E-100	
DCFS	1.0	
DCFD	3.725e15	
DCFS	0x7FC00000	; Quiet NaN
DCFD	&FFF000000000000	; Minus infinity

3.6.6 レジスタ相対式とプログラム相対式

レジスタ相対式は、数値定数あり、または数値定数なしの名前付きレジスタに評価されます（「MAP」（P. 7-20）を参照）。

プログラム相対式は、数値定数あり、または数値定数なしのプログラムカウンタ（PC）に評価されます。通常、数値式と組み合わせたラベルとなります。

例

```

LDR      r4,=data+4*n    ; n is an assembly-time variable
; code
MOV      pc,lr
data   DCD    value0
; n-1 DCD directives
DCD    valueN      ; data+4*n points here
; more DCD directives

```

3.6.7 論理式

論理式は、論理リテラル（{TRUE} または {FALSE}）、論理変数、ブール演算子、関係、および括弧を組み合わせたもので構成されます（「ブール演算子」（P. 3-39）を参照）。

関係は、変数、リテラル、定数、または適切な関係演算子が含まれる式を組み合わせたもので構成されます（「関係演算子」（P. 3-38）を参照）。

3.6.8 論理リテラル

論理リテラルは次のとおりです。

- {TRUE}
- {FALSE}

3.6.9 演算子の優先順位

ARM アセンブラーには、式で使用できる広範な演算子セットが含まれています。ARM アセンブラーに用意されている演算子の多くには、C 言語などの高級言語で使用される演算子と同様の機能があります（「単項演算子」（P. 3-33）および「2 項演算子」（P. 3-35）を参照）。

評価する時厳密な優先順位があります。

1. 最初に、括弧内の式が評価されます。
2. 演算子は優先順位に従って適用されます。
3. 隣接する単項演算子は、右から左に評価されます。
4. 優先順位が同じ 2 項演算子は、左から右に評価されます。

armasm と C 言語における演算子の優先順位

ARM アセンブラーにおける優先順位は、C 言語とは異なります。

例えば、 $(1 + 2 :SHR: 3)$ とある場合、armasm では $(1 + (2 :SHR: 3)) = 1$ と評価されます。C では、 $((1 + 2) >> 3) = 0$ と評価されます。

括弧を使うことによって、優先順位を明示的に示す方がいいです。

もし C 言語でアセンブラーと違った解析をされる式がコードに含まれている場合、通常、armasm によって次のような警告が表示されます。

A1466W:Operator precedence means that expression would evaluate differently in C (A1466W:式の演算子は、C 言語とは異なる優先順位で評価される場合があります)

ただし、--unsafe コマンドラインオプションを使用した場合、この警告は表示されません。

表 3-4 に、armasm での演算子の優先順位と、C における優先順位との比較を記載します (P. 3-33 表 3-5 を参照)。

次の表から以下の点を確認できます。

- 最も優先順位の高い演算子はリストの最上位にあります。
- 最も優先順位の高い演算子は最初に評価されます。
- 優先順位が同じ演算子は、左から右に評価されます。

表 3-4 armasm における演算子の優先順位

armasm の優先順位	C における同等の演算子
単項演算子	単項演算子
<code>* / :MOD:</code>	<code>* / %</code>
文字列操作	n/a
<code>:SHL: :SHR: :ROR: :ROL:</code>	<code><< >></code>
<code>+ - :AND: :OR: :EOR:</code>	<code>+ - & </code>
<code>= > >= < <= /= <></code>	<code>== > >= < <= !=</code>
<code>:LAND::LOR::LEOR:</code>	<code>&& </code>

表 3-5 C における演算子の優先順位

C での優先順位

単項演算子

* / %

+ - (2 項演算子として)

<< >>

< <= > >=

== !=

&

^

|

&&

||

3.6.10 単項演算子

単項演算子は最も優先順位が高く、最初に評価されます。単項演算子はそのオペランドよりも優先順位が高くなります。隣接する演算子は右から左に評価されます。

表 3-6 に、文字列を返す単項演算子を示します。

表 3-6 文字列を返す単項演算子

演算子	使用法	説明
:CHR:	:CHR:A	ASCII コード A が含まれている文字を返します。
:LOWERCASE:	:LOWERCASE:string	すべての大文字を小文字に変換して所定の文字列を返します。
:REVERSE_CC:	:REVERSE_CC:cond_code	cond_code の条件コードの逆数を返します。
:STR:	:STR:A	数値式に対応する 8 桁の 16 進数文字列、または、論理式で使用されている場合は文字列 "T" または "F" を返します。
:UPPERCASE:	:UPPERCASE:string	すべての小文字を大文字に変換して所定の文字列を返します。

表 3-7 に、数値を返す単項演算子を示します。

表 3-7 数値または論理値を返す単項演算子

演算子	使用法	説明
?	?A	シンボル A を定義する行によって生成された実行可能コードのバイト数。
+ と -	+A -A	単項プラス。単項マイナス。+ 記号と - 記号で、数値相対式とプログラム相対式に対して影響を与えることができます。
:BASE:	:BASE:A	A が PC 相対式またはレジスタ相対式の場合、:BASE: は、そのレジスタコンポーネントの番号を返します。:BASE: はマクロで使用すると便利です。
:CC_ENCODING:	:CC_ENCODING:cond_code	cond_code での条件コードの数値を返します。
:DEF:	:DEF:A	A が定義されている場合は {TRUE}、そうでない場合は {FALSE} です。
:INDEX:	:INDEX:A	A がレジスタ相対式の場合、:INDEX: はそのベースレジスタからのオフセットを返します。:INDEX: はマクロで使用すると便利です。
:LEN:	:LEN:A	文字列 A の長さ
:LNOT:	:LNOT:A	A の論理コンポーネント。
:NOT:	:NOT:A	A のビット単位コンポーネント。 ^a
:RCONST:	:RCONST:Rn	r0 ~ r15 に対応するレジスタの番号 0 ~ 15。

a. ~ は、:NOT: のエイリアス、例えば、~A です。

3.6.11 2 項演算子

2 項演算子は、これらの演算子が処理される副次式のペアの間に記述します。

2 項演算子は単項演算子よりも優先順位は低くなります。このセクションでは、この優先順位に従って 2 項演算子を取り上げています。

——注——

優先順位は C と異なります。*armasm と C 言語における演算子の優先順位* (P. 3-32) を参照して下さい。

乗算演算子

乗算演算子は、すべての 2 項演算子の中で最も優先順位が高くなります。乗算演算子は数値式にのみ作用します。

表 3-8 に乗算演算子を示します。

表 3-8 乗算演算子

演算子	エイリアス	使用法	説明
*		A*B	乗算
/		A/B	除算
:MOD:	%	A:MOD:B	A と B のモジュロ演算 (A を B で割って余りを求める)

文字列操作演算子

表 3-9 に文字列操作演算子を示します。

スライス演算子 LEFT および RIGHT の場合

- A は文字列でなければなりません。
- B は数値式でなければなりません。

CC では、A および B はいずれも文字列でなければなりません。

表 3-9 文字列操作演算子

演算子	使用法	説明
:CC:	A:CC:B	B は A の末尾に連結されます。
:LEFT:	A:LEFT:B	A の左から B 文字
:RIGHT:	A:RIGHT:B	A の右から B 文字

シフト演算子

シフト演算子は数値式に作用し、最初のオペランドを、2 番目のオペランドで指定されている量だけシフトまたは回転させます。

表 3-10 にシフト演算子を示します。

表 3-10 シフト演算子

演算子	エイリアス	使用法	説明
:ROL:		A:ROL:B	A を B ビット分左に回転させます。
:ROR:		A:ROR:B	A を B ビット分右に回転させます。
:SHL:	<<	A:SHL:B	A を B ビット分左にシフトさせます。
:SHR:	>>	A:SHR:B	A を B ビット分右にシフトさせます。

——注——

SHR は論理シフトであり、符号ビットまで波及しません。

加算、減算、および論理演算子

加算および減算演算子は数値式に作用します。

論理演算子は数値式に作用します。処理は、ビット単位で実行されます。つまり、オペランドの各ビットに関係なく結果が生成されます。

表 3-11 に加算、減算、および論理演算子を示します。

表 3-11 加算、減算、および論理演算子

演算子	エイリアス	使用法	説明
+		A+B	A を B に加算
-		A-B	A から B を減算
:AND:	&&	A:AND:B	A と B のビット単位論理積
:EOR:	^	A:EOR:B	A と B のビット単位排他的論理和 (XOR)
:OR:		A:OR:B	A と B のビット単位論理和

関係演算子

表 3-12 に関係演算子を示します。関係演算子は、同じタイプの 2 つのオペランドに作用して論理値を生成します。

オペランドには以下のタイプを指定できます。

- 数値
- プログラム相対
- レジスタ相対
- 文字列

文字列は ASCII 順序を使用してソートされます。文字列 A は、文字列 B の先頭に配置されたサブ文字列である場合、文字列 B よりも小さくなります。また、2 つの文字列で異なる左端の文字が、文字列 B よりも文字列 A で小さくなる場合も、文字列 A は文字列 B よりも小さくなります。

数値は符号なしなので、 $0>-1$ の値は {FALSE} となります。

表 3-12 関係演算子

演算子	エイリアス	使用法	説明
=	==	$A=B$	A は B と同等です。
>		$A>B$	A は B よりも大きくなります。
\geq		$A\geq B$	A は B 以上です。
<		$A<B$	A は B より小さくなります。
\leq		$A\leq B$	A は B 以下です。
/=	$\Leftrightarrow \neq$	$A\neq B$	A は B と同等ではありません。

ブール演算子

ブール演算子は、最も優先順位の低い演算子です。ブール演算子は、そのオペランドに対して標準論理演算を実行します。

3つのケースでも、A と B の両方の式は、{TRUE} または {FALSE} のいずれかに評価されなければなりません。

表 3-13 にブール演算子を示します。

表 3-13 ブール演算子

演算子	使用法	説明
:LAND:	A:LAND:B	A と B の論理積
:LEOR:	A:LEOR:B	A と B の排他的論理和 (XOR)
:LOR:	A:LOR:B	A と B の論理和

3.7 診断メッセージ

ARM アセンブラーでは、診断メッセージの範囲を指定できます。デフォルトでは、診断メッセージは表示されません。ただし、コマンドラインオプションを使用して、どのメッセージをアセンブラーで表示するかを制御できます。詳細については、「診断メッセージの出力を制御する」(P. 3-13) を参照して下さい。

3.7.1 インターロック

--cpu オプションによって選択されたプロセッサのパイプラインが原因でコード内で発生可能なインターロックに関する警告メッセージを表示できます。これには、アセンブラーの起動時に次のコマンドラインオプションを使用します。

```
armasm --diag_warning 1563
```

—————注—————

ここで、--cpu オプションは、Cortex-A8 などのマルチイッシュプロセッサを指定します。アセンブラー警告の信頼性は低くなります。

3.8 C プリプロセッサを使用する

C プリプロセッサコマンド `#include` を、アセンブリ言語のソースファイルに含めることができます。これを行うには、`armasm` を使用してファイルをアセンブルする前に、C プリプロセッサを使用してファイルを前処理する必要があります。詳細については、*RealView Compilation Tools v3.0 Compiler and Libraries Guide* を参照して下さい。

生成ファイル内の `#line` コマンドは、`armasm` によって正しく解釈されます。`#line` コマンドの情報を使用してエラーメッセージおよび `debug_line` テーブルが生成されます。

例 3-1 に、ファイル `source.s` を前処理してアセンブルするためのコマンドを示します。この例では、プリプロセッサによって、`preprocessed.s` という名前のファイルを出力します。`armasm` によって、`preprocessed.s` がアセンブルされます。

例 3-1 アセンブリ言語のソースファイルを前処理する

```
armcc -E source.s > preprocessed.s
armasm preprocessed.s
```

第 4 章

ARM 命令と Thumb 命令

本章では、ARM アセンブラーでサポートされている ARM®、Thumb®-2、Thumb-2EE、および Thumb 命令 (32 ビットと 16 ビット) 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。

- [命令の概要 \(P. 4-2\)](#)
- [メモリアクセス命令 \(P. 4-8\)](#)
- [汎用データ処理命令 \(P. 4-47\)](#)
- [乗算命令 \(P. 4-78\)](#)
- [サチュレート命令 \(P. 4-99\)](#)
- [並列命令 \(P. 4-104\)](#)
- [パック命令と展開命令 \(P. 4-112\)](#)
- [分岐命令 \(P. 4-120\)](#)
- [コプロセッサ命令 \(P. 4-126\)](#)
- [その他の命令 \(P. 4-136\)](#)
- [擬似命令 \(P. 4-155\)](#)

4.1 命令の概要

表 4-1 は、ARM、Thumb、Thumb-2、および Thumb-2EE 命令セットで使用できる命令の概要を示しています。この表を使用して、本章の残りの部分に記載されている各命令および擬似命令の説明に移動できます。

表 4-1 の [§] 列は、各命令が初めて導入された ARM アーキテクチャを示しています。6T2 は、命令が 6T2 で初めて導入されたが、Thumb-2 命令セットだけでなく ARM 命令セットでも使用できることを示しています。T2 は、命令が Thumb-2 命令セットのみで使用できることを示しています。T2-EE は、命令が Thumb-2EE のみで使用できることを示しています。XScale は、命令が XScale プロセッサのみで使用できることを示しています。7M は、命令が 7M のみで使用できることを示しています。

—— 注 ——

特に明示されない限り、Thumb-2EE 命令は Thumb-2 命令と同一です。

表 4-1 命令の参照ページ

ニーモニック	意味	ページ	§
ADC、ADD	キャリー付き加算、加算	P. 4-51	すべて
ADR	プログラム相対アドレスまたはレジスタ相対アドレス のロード（狭範囲）	P. 4-28	すべて
ADRL 擬似命令	プログラム相対アドレスまたはレジスタ相対アドレス のロード（中範囲）	P. 4-156	すべて
AND	論理積	P. 4-56	すべて
ASR	算術右シフト	P. 4-72	すべて
B	分岐命令	P. 4-121	すべて
BFC、BFI	ビットフィールドのクリア命令と挿入命令	P. 4-113	6T2
BIC	ビットクリア命令	P. 4-56	すべて
BKPT	ブレークポイント命令	P. 4-138	5
BL	リンク付き分岐命令	P. 4-121	すべて
CBZ、CBNZ	0 と比較し、0 の（または 0 でない）場合に分岐する命令	P. 4-124	T2

表 4-1 命令の参照ページ（続き）

ニーモニック	意味	ページ	§
CDP、CDP2	コプロセッサデータ処理命令	P. 4-127	すべて、5
CHKA	配列をチェックする命令	P. 4-153	7T2-EE
CLREX	排他のクリア命令	P. 4-45	6T2
CLZ	先行ゼロカウント命令	P. 4-59	5
CMN、CMP	比較命令、比較否定命令	P. 4-60	すべて
CPS	プロセッサ状態の変更命令	P. 4-143	6
CPY	コピー命令	P. 4-62	6
DBG、DMB、DSB	デバッグ、データメモリバリア、データ同期バリア	P. 4-149	7
ENTERX、LEAVEX	ThumbEEとの間の状態切り替え命令	P. 4-152	7T2-EE
EOR	排他的論理和 (XOR)	P. 4-56	すべて
HB、HBL、HBLP、HBP	ハンドラの分岐、指定されたハンドラへの分岐	P. 4-154	7T2-EE
ISB	命令同期バリア	P. 4-149	7
IT	If-Then	P. 4-74	T2
LDC、LDC2	コプロセッサロード	P. 4-132	すべて、5
LDM	多重レジスタロード	P. 4-32	すべて
LDR	レジスタロード命令	P. 4-8	すべて
LDR 擬似命令	レジスタロード擬似命令	P. 4-160	すべて
LDREX	排他的レジスタロード	P. 4-42	6
LSL、LSR	論理左シフト、論理右シフト	P. 4-72	すべて
MAR	レジスタから 40 ビット累算器への移動命令	P. 4-151	XScale
MCR、MCR2、MCRR、 MCRR2	レジスタからコプロセッサへの移動命令	P. 4-128	すべて、5、5E、6

表 4-1 命令の参照ページ (続き)

ニーモニック	意味	ページ	§
MIA、MIAPH、MIAxy	乗算および内部 40 ビット加算	P. 4-97	XScale
MLA、MLS	積和、積差	P. 4-79	すべて、6T2
MOV	移動命令	P. 4-62	すべて
MOVT	上位ハーフワードにデータを代入する命令	P. 4-65	6T2
MOV32 擬似命令	レジスタへの 32 ビット定数の移動命令	P. 4-158	6T2
MRA	40 ビット累算器からレジスタへの移動	P. 4-151	XScale
MRC、MRC2	コプロセッサからレジスタへの移動命令	P. 4-130	すべて、5
MRS	PSR からレジスタへの移動命令	P. 4-140	すべて
MSR	レジスタから PSR への移動命令	P. 4-141	すべて
MUL	乗算	P. 4-79	すべて
MVN	データの各ビットを反転させてから代入する命令	P. 4-62	すべて
NOP	操作なし	P. 4-147	すべて
ORN	論理和否定	P. 4-56	6T2
ORR	論理和	P. 4-56	すべて
PKHBT、PKHTB	ハーフワードのパック命令	P. 4-118	6
PUSH、POP	レジスタのプッシュ命令、ポップ命令	P. 4-35	すべて
QADD、QDADD、 QDSUB、QSUB	サチュレート算術演算	P. 4-100	5ExP
QADD8、QADD16、 QASX、QSUB8、 QSUB16、QSAX	並列符号付きサチュレート算術演算	P. 4-105	6
REV、REV16、REVSH、 RBIT	バイト順序の反転命令、ビットの反転命令	P. 4-70	6、6T2

表 4-1 命令の参照ページ（続き）

ニーモニック	意味	ページ	§
RFE	例外からの復帰	P. 4-38	6T2
ROR	レジスタの右ロテート	P. 4-72	すべて
RSB、RSC、SBC	逆減算、キャリー付き逆減算、キャリー付き減算	P. 4-51	すべて
SADD8、SADD16、 SASX	並列符号付き算術演算	P. 4-105	6
SBFX、UBFX	符号付き / 符号なしビットフィールドの抽出命令	P. 4-114	6T2
SDIV	符号付き除算	P. 4-77	7M
SEL	CPSR の GE フラグに基づくバイトの選択	P. 4-68	6
SEV	イベントを設定する命令	P. 4-147	K
SETEND	メモリアクセス時のエンディアン形式を設定する命令	P. 4-146	6
SHADD8、SHADD16、 SHASX、SHSUB8、 SHSUB16、SHSAX	符号付きでバイト、ハーフワード並列演算	P. 4-105	6
SMC	セキュアモニターコール	P. 4-145	Z
SMLAD	デュアル符号付き積和 ($32 \leq 32 + 16 \times 16 + 16 \times 16$)	P. 4-92	6
SMLAL	符号付き積和 ($64 \leq 64 + 32 \times 32$)	P. 4-81	M
SMLALxy	符号付き積和 ($64 \leq 64 + 16 \times 16$)	P. 4-86	5ExP
SMLALD	デュアル符号付き積和 long ($64 \leq 64 + 16 \times 16 + 16 \times 16$)	P. 4-94	6
SMLSD	デュアル符号付き乗減累算 ($32 \leq 32 + 16 \times 16 - 16 \times 16$)	P. 4-92	6
SMLS LD	デュアル符号付き乗減累算 long ($64 \leq 64 + 16 \times 16 - 16 \times 16$)	P. 4-94	6

表 4-1 命令の参照ページ (続き)

ニーモニック	意味	ページ	§
SMMUL	符号付き上位ワード乗算 ($32 \leq \text{TopWord}$ (32×32))	P. 4-90	6
SMUAD、SMUSD	デュアル符号付き乗算、および積の加算または減算	P. 4-88	6
SMULL	符号付き乗算 ($64 \leq 32 \times 32$)	P. 4-81	M
SMULxy	符号付き乗算 ($32 \leq 16 \times 16$)	P. 4-83	5ExP
SMULWy	符号付き乗算 ($32 \leq 32 \times 16$)	P. 4-85	5ExP
SRS	復帰状態のストア命令	P. 4-40	6T2
SSAT	符号付きサチュレート演算	P. 4-102	6
SSAT16	符号付き並列ハーフワードサチュレート演算	P. 4-110	6
SSUB8、SSUB16、 SSAX	並列符号付き算術演算	P. 4-105	6
STC	コプロセッサストア命令	P. 4-132	すべて
STC2	コプロセッサストア命令 (代替命令)	P. 4-134	5ExP
STM	多重レジスタストア	P. 4-32	すべて
STR	レジスタストア	P. 4-8	すべて
STREX	排他的レジスタストア	P. 4-42	6
SUB	減算	P. 4-51	すべて
SUBS PC, LR	スタックを行わない例外からの復帰命令	P. 4-55	T2
SVC (以前は SWI)	スーパーバイザコール	P. 4-139	すべて
SWP、SWPB	レジスタとメモリ間のスワップ (ARMのみ)	P. 4-46	すべて
SXT、SXTA	加算オプション付きの符号拡張	P. 4-115	6
TBB、TBH	テーブル分岐バイト、ハーフワード	P. 4-125	T2
TEQ、TST	等価テスト、テスト	P. 4-66	すべて
UADD8、UADD16、 UASX	並列符号なし算術演算	P. 4-105	6
UDIV	符号なし除算	P. 4-77	7M

表 4-1 命令の参照ページ（続き）

ニーモニック	意味	ページ	§
UHADD8、UHADD16、 UHASX、UHSUB8、 UHSUB16、UHSAX	並列符号なし半演算	P. 4-105	6
UMAAL	符号なし積和累算 long ($64 \leq 32 + 32 + 32 \times 32$)	P. 4-96	6
UMLAL、UMULL	符号なし積和、乗算 ($64 \leq 32 \times 32 + 64$)、($64 \leq 32 \times 32$)	P. 4-81	M
UQADD8、UQADD16、 UQASX、UQSUB8、 UQSUB16、UQSAX	並列符号なしサチュレート算術演算	P. 4-105	6
USAD8	符号なし絶対差の和	P. 4-108	6
USADA8	符号なし絶対差の和の累積	P. 4-108	6
USAT	符号なしサチュレート演算	P. 4-102	6
USAT16	符号なし並列ハーフワードサチュレート演算	P. 4-110	6
USUB8、USUB16、 USAX	並列符号なし算術演算	P. 4-105	6
UXT、UXTA	任意で加算を伴う符号なし拡張	P. 4-115	6
WFE、WFI、YIELD	イベント待機、割り込み待機、明け渡しを行うヒント命令	P. 4-147	6T2

4.2 メモリアクセス命令

このセクションは以下のサブセクションから構成されています。

- **アドレス境界調整 (P. 4-9)**
すべてのメモリアクセス命令に該当する、境界調整の考慮事項です。
- **LDR, STR (ゼロオフセット、イミディエートオフセット、またはプレインデクスイミディエートオフセット) (P. 4-10)**
バイト、ハーフワード、ワード、およびダブルワードのロード/ストアを実行する命令です。
- **LDR, STR (ポストインデクスイミディエートオフセット) (P. 4-15)**
バイト、ハーフワード、ワード、およびダブルワードのロード/ストアを実行する命令です。
- **LDR および STR (レジスタオフセットまたはプレインデクスレジスタオフセット) (P. 4-18)**
バイト、ハーフワード、ワード、およびダブルワードのロード/ストアを実行する命令です。
- **LDR, STR (ポストインデクスレジスタオフセット) (P. 4-22)**
バイト、ハーフワード、ワード、およびダブルワードのロード/ストアを実行する命令です。
- **LDR (プログラムカウンタ相対) (P. 4-25)**
バイト、ハーフワード、ワード、およびダブルワードのロードを実行する命令です。
- **ADR (P. 4-28)**
プログラム相対アドレスまたはレジスタ相対アドレスのロード（狭範囲、位置非依存）を実行する命令です。
- **PLD, PLI (P. 4-30)**
アドレスのプリロードを実行する命令です。
- **LDM, STM (P. 4-32)**
多重レジスタロード/ストア命令です。
- **PUSH および POP (P. 4-35)**
ローレジスタと LR (オプション) をスタックにプッシュする命令です。
ローレジスタと PC (オプション) をスタックからポップする命令です。
- **RFE (P. 4-38)**
例外から復帰する命令です。

- *SRS* (P. 4-40)
復帰状態をストアする命令です。
- *LDREX*, *STREX* (P. 4-42)
排他的レジスタロード / ストア命令です。
- *CLREX* (P. 4-45)
排他をクリアする命令です。
- *SWP*, *SWPB* (P. 4-46)
レジスタとメモリ間のデータスワップを実行する命令です。

注

この他に、*LDR* 擬似命令もあります（「*LDR* 擬似命令」(P. 4-160) 参照）。この擬似命令をアセンブルすることによって、*LDR* 命令、*MOV* 命令または*MVN* 命令が生成されます。

4.2.1 アドレス境界調整

ほとんどの場合、4 バイト転送のアドレスは 4 バイト境界で、2 バイト転送のアドレスは 2 バイト境界で整列されている必要があります。ARMv6T2 以降では、非境界整列アクセスが許可されます。ARMv7 以降では、非境界整列アクセスが義務付けられています（デフォルト）。

ARMv6 以前では、システムにシステムコプロセッサ (cp15) が存在する場合に境界調整チェックをイネーブルできます。境界調整チェックがイネーブルされている場合、ワード境界で整列されていない 32 ビット転送が行われると、境界調整例外が発生します。

システムにシステムコプロセッサ (cp15) が存在しないか、境界調整チェックがディセーブルされている場合、以下のようになります。

- *STR* では、指定されたアドレスが 4 の倍数に切り捨てられます。
- *LDR* では、以下のようにになります。
 1. 指定されたアドレスが 4 の倍数に切り捨てられます。
 2. 切り捨て後のアドレスから 4 バイトのデータがロードされます。
 3. ロードされたデータが、アドレスのビット [1:0] に応じて 1 バイト、2 バイト、または 3 バイト分右にロテートされます。

この場合、リトルエンディアンメモリシステムでは、指定されたバイトがレジスタの最下位バイトに入れられます。

ビッグエンディアンメモリシステムでは、指定されたバイトが以下のようにレジスタに入れられます。

- アドレスのビット 0 が 0 の場合、レジスタのビット [31:24] に入れます。
- アドレスのビット 0 が 1 の場合、レジスタのビット [15:8] に入れます。

4.2.2 LDR、STR（ゼロオフセット、イミディエートオフセット、またはプレインデクスイミディエートオフセット）

ロード / ストア命令です。バイトとハーフワードのロードは、32 ビットにゼロ拡張または符号拡張されます。

――注――

「擬似命令」(P. 4-155) も参照して下さい。

構文

```
op{type}{T}{cond} Rd, {Rd2,} [Rn {, #offset}]
op{type}{T}{cond} Rd, {Rd2,} [Rn , #offset]!
```

パラメータの説明：

<i>op</i>	次のいずれかを指定できます。 LDR レジスタロード STR レジスタストア
<i>type</i>	次のいずれかを指定できます。 B 符号なしバイト SB 符号付きバイト (LDR のみ) H 符号なしハーフワード SH 符号付きハーフワード (LDR のみ) - 省略 (ワード) D ダブルワード
<i>T</i>	任意に指定できる接尾文字です。T を指定すると、プロセッサが特権モードであっても、メモリシステムはプロセッサがユーザモードである場合と同様にアクセスを処理します（「プロセッサモード」(P. 2-5) 参照）。ユーザモードの場合、T による影響はありません。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。
<i>Rd</i>	ロードまたはストアする ARM レジスタを指定します。

<i>Rd2</i>	2番目にロードまたはストアする ARM レジスタを指定します (<i>type == D</i> の場合のみ)。
<i>Rn</i>	メモリアドレスのベースとなるレジスタを指定します。
<i>offset</i>	イミディエートオフセットを指定します。オフセットを指定しなかった場合、その命令はゼロオフセット命令になります。
!	任意に指定できる接尾文字です。! を指定すると、その命令はプレインデクス命令になります。この場合、 <i>Rn</i> に <i>Rd</i> または <i>Rd2</i> と同じレジスタは指定できません。

ゼロオフセット

Rn の値は、転送用のアドレスとして使用されます。

接尾文字 T は、ダブルワード命令では使用できません。

イミディエートオフセット

このオフセットは、データ転送が発生する前に *Rn* の値に適用されます。この結果は、転送用のメモリアドレスとして使用されます。使用できるオフセットの範囲を以下に示します。

- ARM ワードまたはバイト命令 : -4095 ~ +4095
- ARM 符号付きバイト、ハーフワード、符号付きハーフワード、およびダブルワード命令 : -255 ~ +255
- 接尾文字 T が指定されていないすべての Thumb-2 命令（ダブルワード命令を除く）: -255 ~ +4095
- Thumb-2 ダブルワード命令 : -1020 ~ +1020 （4 の倍数で指定して下さい）
- 接尾文字 T が指定されている Thumb-2 命令 : 0 ~ +255

接尾文字 T は、ARM 命令または Thumb-2 ダブルワード命令には使用できません。他の Thumb-2 命令にはこの接尾文字を使用できます。

プレインデクスイミディエートオフセット

このオフセットは、データ転送が発生する前に Rn の値に適用されます。この結果は、転送用のメモリアドレスとして使用されます。この結果は Rn にライトバックされます。

使用できるオフセットの範囲を以下に示します。

- ARM ワードまたはバイト命令 : -4095 ~ +4095
- ARM 符号付きバイト、ハーフワード、符号付きハーフワード、およびダブルワード命令 : -255 ~ +255
- ダブルワード命令を除くすべての Thumb-2 命令 : -255 ~ +255
- Thumb-2 ダブルワード命令 : -1020 ~ +1020 (4 の倍数で指定して下さい)

接尾文字 T は、ARM 命令または Thumb-2 命令には使用できません。

ダブルワードレジスタの制約条件

Thumb-2 命令の場合、 Rd または $Rd2$ に $r15$ は指定できません。

ARM 命令には、以下の制約条件が適用されます。

- Rd には偶数番号のレジスタを指定する必要があります。
- Rd に $r14$ は指定できません。
- $Rd2$ には $R(d + 1)$ を指定する必要があります。

16 ビット命令

Thumb-2 コード、Thumb を利用できる他のプロセッサの Thumb コード、および Thumb-2EE コードでは、これらの命令のサブセットの 16 ビットバージョンを使用できます。

16 ビット命令には、以下の制約条件が適用されます。

- ゼロオフセット命令とイミディエートオフセット命令のみを使用できます。プレインデクスイミディエートオフセット命令は使用できません。
- 接尾文字 T は使用できません。
- Rn に $r13$ 、 $r9$ 、または $r10$ 以外のレジスタを指定した場合、以下の制約条件が適用されます。
 - Rd および Rn は共に Lo レジスタである必要があります。
 - ワード命令の場合、オフセットは 0 ~ 124 の範囲内にある 4 の倍数である必要があります。
- Thumb-2EE では、LDR オフセットは -28 ~ +124 の範囲内にある 4 の倍数である必要があります。

- ハーフワード命令の場合、オフセットは 0 ~ +62 の範囲内にある 2 の倍数である必要があります。
- バイト命令の場合、オフセットは 0 ~ +31 である必要があります。
- 符号付きバイト、符号付きハーフワード、およびダブルワード命令は使用できません。
- *Rn* に r13 を指定した場合、以下の制約条件が適用されます。
 - *Rd* は Lo レジスタである必要があります。
 - 命令はワード命令である必要があります。
 - オフセットは 0 ~ +1020 の範囲内にある 4 の倍数である必要があります。
- *Rn* に r9 (Thumb-2EE のみで使用できます) を指定した場合、以下の制約条件が適用されます。
 - *Rd* は Lo レジスタである必要があります。
 - 命令はワード命令である必要があります。
 - オフセットは 0 ~ +252 の 4 の倍数である必要があります。
- *Rn* に r10 (Thumb-2EE のみで使用できます) を指定した場合、以下の制約条件が適用されます。
 - *Rd* は Lo レジスタである必要があります。
 - 命令はワードロード命令である必要があります。
 - オフセットは 0 ~ +124 の範囲内にある 4 の倍数である必要があります。

Thumb-2EE 命令

ベースレジスタの値が 0 の場合、HandlerBase - 4 にある NullCheck ハンドラへの分岐が実行されます。

r15 へのロード

ARM コードと Thumb-2 コードでは、どちらも *Rd* にプログラムカウンタを指定できます。この場合、*type* を省略する必要があります。

r15 (プログラムカウンタ) へのロードを実行すると、ロードされたアドレスにある命令への分岐が発生します。

ARMv4 では、ロードされる値のビット [1:0] は 0 である必要があります。

ARMv5T 以降には以下の特徴があります。

- r15 にロードされる値のビット [1:0] の値は 0b10 にできません。
- r15 にロードされる値のビット 0 が設定されている場合、プロセッサは Thumb 状態に切り替わります。

r15 へのロードを実行するときに、接尾文字 T は使用できません。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARM アーキテクチャのすべての T2 バリアントで使用できます。

これらの 16 ビット Thumb 命令は、ARM アーキテクチャのすべての T バリアントで使用できます。16 ビット命令には、Thumb-2EE 状態でしか使用できないものがあります（「16 ビット命令」（P. 4-12）参照）。

ARMv5 以降の T および T2 バリアントには、以下のような特徴があります。

- ARM 状態で r15 へのロードを実行すると、ロードされる値のビット 0 が 1 の場合、Thumb 状態に切り替わります。
- Thumb 状態で r15 へのロードを実行すると、ロードされる値のビット 0 が 0 の場合、ARM 状態に切り替わります。

例

```

LDR      r8,[r10]           ; loads r8 from the address in r10.

LDRNE   r2,[r5,#960]!; (conditionally) loads r2 from a word
                  ; 960 bytes above the address in r5, and
                  ; increments r5 by 960.

STR      r2,[r9,#consta-struc] ; consta-struc is an expression evaluating
                  ; to a constant in the range 0-4095.

LDR      r0,localdata        ; loads a word located at label localdata

```

4.2.3 LDR、STR (ポストインデクスイミディエートオフセット)

レジスタロード / ストア命令です。バイトとハーフワードのロードは、32 ビットにゼロ拡張または符号拡張されます。

構文

op{type}{T}{cond} Rd, {Rd2,} [Rn], #offset

パラメータの説明 :

<i>op</i>	次のいずれかを指定できます。 LDR レジスタロード STR レジスタストア
<i>type</i>	次のいずれかを指定できます。 B 符号なしバイト SB 符号付きバイト (LDR のみ) H 符号なしハーフワード SH 符号付きハーフワード (LDR のみ) - 省略 (ワード) D ダブルワード
<i>T</i>	任意に指定できる接尾文字です。T を指定すると、プロセッサが特権モードであっても、メモリシステムはプロセッサがユーザモードである場合と同様にアクセスを処理します（「プロセッサモード」(P. 2-5) 参照）。ユーザモードの場合、T による影響はありません。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。
<i>Rd</i>	ロードまたはストアする ARM レジスタを指定します。
<i>Rd2</i>	2 番目にロードまたはストアする ARM レジスタを指定します (<i>type</i> == D の場合のみ)。
<i>Rn</i>	メモリアドレスのベースとなるレジスタを指定します。Rn に Rd または Rd2 と同じレジスタは指定できません。
<i>offset</i>	イミディエートオフセットを指定します。オフセットを指定しなかった場合、その命令はゼロオフセット命令になります。

動作と制約条件

Rn の値は、転送用のメモリアドレスとして使用されます。このオフセットは、データ転送が発生した後に *Rn* の値に適用されます。この結果は *Rn* にライトバックされます。

使用できるオフセットの範囲を以下に示します。

- ARM ワードまたはバイト命令 : -4095 ~ +4095
- ARM 符号付きバイト、ハーフワード、符号付きハーフワード、およびダブルワード命令 : -255 ~ +255
- ダブルワード命令を除くすべての Thumb-2 命令 : -255 ~ +255
- Thumb-2 ダブルワード命令 : -1020 ~ +1020 (4 の倍数で指定して下さい)

接尾文字 T は、Thumb-2 命令または ARM ダブルワード命令には使用できません。その他すべての ARM 命令にはこの接尾文字を使用できます。

ダブルワードレジスタの制約条件

Thumb-2 命令の場合、*Rd* または *Rd2* に *r15* は指定できません。

ARM 命令には、以下の制約条件が適用されます。

- *Rd* には偶数番号のレジスタを指定する必要があります。
- *Rd* に *r14* は指定できません。
- *Rd2* には *R(d + 1)* を指定する必要があります。

Thumb-2EE 命令

ベースレジスタの値が 0 の場合、HandlerBase - 4 にある NullCheck ハンドラへの分岐が実行されます。

r15 へのロード

ARM コードと Thumb-2 コードでは、どちらも *Rd* にプログラムカウンタを指定できます。この場合、*type* を省略する必要があります。

r15 (プログラムカウンタ) へのロードを実行すると、ロードされたアドレスにある命令への分岐が発生します。

ARMv4 では、ロードされる値のビット [1:0] は 0 である必要があります。

ARMv5T 以降には以下ののような特徴があります。

- *r15* にロードされる値のビット [1:0] の値は 0b10 にできません。
- *r15* にロードされる値のビット 0 が設定されている場合、プロセッサは Thumb 状態に切り替わります。

r15 へのロードを実行するときに、接尾文字 T は使用できません。

r15 からのストア

Thumb コードでは、r15 からのストアは実行できません。

ARM コードでは、できる限り r15 からのストアを実行しないで下さい。

r15 からのストアを実行した場合、ストアされる値は「現在の命令のアドレス + 実装定義済み定数」となります。この定数は、特定のプロセッサに対しては常に一定です。

アセンブルするコードを他のプロセッサで使用する可能性がある場合は、以下のようなコードを使用すると、定数を実行時に確認できます。

```
SUB R1, PC, #4 ; R1 = address of following STR instruction
STR PC, [R0] ; Store address of STR instruction + offset,
LDR R0, [R0] ; then reload it
SUB R0, R0, R1 ; Calculate the offset as the difference
```

特定のプロセッサ用のコードをアセンブルする場合は、定数の値を `armasm -PCSTOREOFFSET` として使用できます。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARM アーキテクチャのすべての T2 バリアントで使用できます。

これらの命令の 16 ビットバージョンはありません。

`LDRSHT`、`LDRHT`、`STRHT`、および `LDRSBT` 命令は、ARMv6 の T2 バリアントのみで使用できます。

ARMv5 以降の T および T2 バリアントには、以下のような特徴があります。

- ARM 状態で r15 へのロードを実行すると、ロードされる値のビット 0 が 1 の場合、Thumb 状態に切り替わります。
- Thumb 状態で r15 へのロードを実行すると、ロードされる値のビット 0 が 0 の場合、ARM 状態に切り替わります。

例

```
STR      r5,[r7],#-8          ; stores a word from r5 to the address
                                ; in r7, and then decrements r7 by 8.
```

4.2.4 LDR および STR (レジスタオフセットまたはプレインデクスレジスタオフセット)

レジスタロード / ストア命令です。バイトとハーフワードのロードは、32 ビットにゼロ拡張または符号拡張されます。

構文

op{type}{cond} {Rd, {Rd2,}} [Rn, +/-Rm {, shift}]{!}

パラメータの説明 :

<i>op</i>	次のいずれかを指定できます。 LDR レジスタロード STR レジスタストア
<i>type</i>	次のいずれかを指定できます。 B 符号なしバイト SB 符号付きバイト (LDR のみ) H 符号なしハーフワード SH 符号付きハーフワード (LDR のみ) - 省略 (ワード) D ダブルワード
<i>cond</i>	任意に指定できる条件コードです (「条件実行」(P. 2-17) 参照)。
<i>Rd</i>	ロードまたはストアする ARM レジスタを指定します。
<i>Rd2</i>	2 番目にロードまたはストアする ARM レジスタを指定します (<i>type == D</i> の場合のみ)。
<i>Rn</i>	メモリアドレスのベースとなるレジスタを指定します。
<i>Rm</i>	オフセットとして使用される値を保持するレジスタを指定します。 <i>Rm</i> に r15 は指定できません。
<i>shift</i>	任意に指定できるシフトです。詳細については、「動作と制約条件」を参照して下さい。
!	任意に指定できる接尾文字です。! を指定すると、その命令はプレインデクス命令になります。この場合、 <i>Rn</i> に <i>Rd</i> または <i>Rd2</i> と同じレジスタは指定できません。

動作と制約条件

ARM では、 Rm の値は Rn の値に対して加算または減算されます。Thumb と Thumb-2 では、減算は実行できません。この結果は、転送用のメモリアドレスとして使用されます。命令がプレインデクス命令である場合、その結果は Rn にライトバックされます。

可能なシフトの範囲を以下に示します。

- ダブルワード命令を除くすべての Thumb-2 命令 : LSL 0 ~ 3
- ARM ワード命令と符号なしバイト命令には、以下のいずれかの範囲
 - LSL 0 ~ 31
 - LSR 1 ~ 32
 - ASR 1 ~ 32
 - ROR 1 ~ 31
 - RRX
- Thumb-2 ダブルワード命令、ARM 符号付きバイト命令、ARM 符号なしハーフワード命令、ARM 符号付きハーフワード命令、ARM ダブルワード命令 : シフト不可能

ダブルワードレジスタの制約条件

Thumb-2 命令の場合、 Rd または $Rd2$ に r15 は指定できません。

ARM 命令には、以下の制約条件が適用されます。

- Rd には偶数番号のレジスタを指定する必要があります。
- Rd に r14 は指定できません。
- $Rd2$ には $R(d + 1)$ を指定する必要があります。

16 ビット命令

Thumb-2 コードと、Thumb を利用できる他のプロセッサのThumb コードでは、これらの命令のサブセットの 16 ビットバージョンを使用できます。

16 ビット命令には、以下の制約条件が適用されます。

- レジスタオフセット命令のみを使用できます。プレインデクスレジスタオフセット命令は使用できません。
- Rd 、 Rn 、および Rm はすべて Lo レジスタである必要があります。
- ワード、符号なしハーフワード、符号付きハーフワード、符号なしバイト、および符号付きバイト命令を使用できます。ダブルワード命令は使用できません。

Thumb-2EE では、 Rm は以下のように左シフトする必要があります。

- LDRH、LDRSH、STRH : 1 ビット
- LDR、STR : 2 ビット

Thumb-2EE 命令

ベースレジスタの値が 0 の場合、HandlerBase - 4 にある NullCheck ハンドラへの分岐が実行されます。

r15 へのロード

ARM コードと Thumb-2 コードでは、どちらも *Rd* にプログラムカウンタを指定できます。この場合、*type* を省略する必要があります。

r15 (プログラムカウンタ) へのロードを実行すると、ロードされたアドレスにある命令への分岐が発生します。

ARMv4 では、ロードされる値のビット [1:0] は 0 である必要があります。

ARMv5T 以降には以下のような特徴があります。

- r15 にロードされる値のビット [1:0] の値は 0b10 にできません。
- r15 にロードされる値のビット 0 が設定されている場合、プロセッサは Thumb 状態に切り替わります。

r15 へのロードを実行するときに、接尾文字 T は使用できません。

r15 からのストア

Thumb コードでは、r15 からのストアは実行できません。

ARM コードでは、できる限り r15 からのストアを実行しないで下さい。

r15 からのストアを実行した場合、ストアされる値は「現在の命令のアドレス + 実装定義済み定数」となります。この定数は、特定のプロセッサに対しては常に一定です。

アセンブルするコードを他のプロセッサで使用する可能性がある場合は、以下のようないい處を確認できます。

```
SUB R1, PC, #4 ; R1 = address of following STR instruction
STR PC, [R0]    ; Store address of STR instruction + offset,
LDR R0, [R0]    ; then reload it
SUB R0, R0, R1 ; Calculate the offset as the difference
```

特定のプロセッサ用のコードをアセンブルする場合は、定数の値を `armasm` で `{PCSTOREOFFSET}` として使用できます。

アーキテクチャ

ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

32 ビット Thumb-2 命令は、ARM アーキテクチャのすべての T2 バリアントで使用できます。

16 ビット Thumb 命令は、ARM アーキテクチャのすべての T バリアントで使用できます。16 ビット命令には、Thumb-2EE 状態でしか使用できないものがあります（「16 ビット命令」（P. 4-19）参照）。

ARMv5 以降の T および T2 バリアントには、以下のような特徴があります。

- ARM 状態で r15 へのロードを実行すると、ロードされる値のビット 0 が 1 の場合、Thumb 状態に切り替わります。
- Thumb 状態で r15 へのロードを実行すると、ロードされる値のビット 0 が 0 の場合、ARM 状態に切り替わります。

4.2.5 LDR、STR（ポストインデクスレジスタオフセット）

ロード / ストア命令です。バイトとハーフワードのロードは、32 ビットにゼロ拡張または符号拡張されます。

構文

op{type}{T}{cond} Rd, {Rd2,} [Rn], +/-Rm {, shift}

パラメータの説明：

<i>op</i>	次のいずれかを指定できます。 LDR レジスタロード STR レジスタストア
<i>type</i>	次のいずれかを指定できます。 B 符号なしバイト SB 符号付きバイト (LDR のみ) H 符号なしハーフワード SH 符号付きハーフワード (LDR のみ) - 省略 (ワード) D ダブルワード
<i>cond</i>	任意に指定できる条件コードです (「条件実行」(P. 2-17) 参照)。
T	任意に指定できる接尾文字です。T を指定すると、プロセッサが特権モードであっても、メモリシステムはプロセッサがユーザモードである場合と同様にアクセスを処理します (「プロセッサモード」(P. 2-5) 参照)。ユーザモードの場合、T による影響はありません。
<i>Rd</i>	ロードまたはストアする ARM レジスタを指定します。
<i>Rd2</i>	2 番目にロードまたはストアする ARM レジスタを指定します (<i>type == D</i> の場合のみ)。
<i>Rn</i>	メモリアドレスのベースとなるレジスタを指定します。Rn に <i>Rd</i> または <i>Rd2</i> と同じレジスタは指定できません。
<i>Rm</i>	オフセットとして使用される値を保持するレジスタを指定します。Rm に r15 は指定できません。
<i>shift</i>	任意に指定できるシフトです。詳細については、「動作と制約条件」を参照して下さい。

動作と制約条件

Rn の値は、転送用のメモリアドレスとして使用されます。このオフセットは、データ転送が発生した後に *Rn* の値に適用されます。ARM では、このオフセットを *Rn* に対して加算または減算できます。Thumb-2 では、このオフセットを *Rn* に加算することはできますが、減算することはできません。この結果は *Rn* にライトバックされます。*Rn* に *r15* は指定できません。

可能なシフトの範囲を以下に示します。

- ARM ワード命令と符号なしバイト命令：以下の範囲
 - LSL 0 ~ 31
 - LSR 1 ~ 32
 - ASR 1 ~ 32
 - ROR 1 ~ 31
 - RRX
- Thumb-2 命令、ARM 符号付きバイト命令、ARM 符号なしハーフワード命令、ARM 符号付きハーフワード命令、ARM ダブルワード命令：シフト不可能

接尾文字 T は、Thumb-2 命令または ARM ダブルワード命令には使用できません。他の ARM 命令にはこの接尾文字を使用できます。

ダブルワードレジスタの制約条件

Thumb-2 命令の場合、*Rd* または *Rd2* に *r15* は指定できません。

ARM 命令には、以下の制約条件が適用されます。

- *Rd* には偶数番号のレジスタを指定する必要があります。
- *Rd* に *r14* は指定できません。
- *Rd2* には *R(d + 1)* を指定する必要があります。

Thumb-2EE 命令

ベースレジスタの値が 0 の場合、HandlerBase - 4 にある NullCheck ハンドラへの分岐が実行されます。

r15 へのロード

ARM コードと Thumb-2 コードでは、どちらも *Rd* にプログラムカウンタを指定できます。この場合、*type* を省略する必要があります。

r15（プログラムカウンタ）へのロードを実行すると、ロードされたアドレスにある命令への分岐が発生します。

ARMv4 では、ロードされる値のビット [1:0] は 0 である必要があります。

ARMv5T 以降には以下のような特徴があります。

- r15 にロードされる値のビット [1:0] の値は 0b10 にできません。
- r15 にロードされる値のビット 0 が設定されている場合、プロセッサは Thumb 状態に切り替わります。

r15 へのロードを実行するときに、接尾文字 T は使用できません。

r15 からのストア

Thumb コードでは、r15 からのストアは実行できません。

ARM コードでは、できる限り r15 からのストアを実行しないで下さい。

r15 からのストアを実行した場合、ストアされる値は「現在の命令のアドレス + 実装定義済み定数」となります。この定数は、特定のプロセッサに対しては常に一定です。

アセンブルするコードを他のプロセッサで使用する可能性がある場合は、以下のようないコードを使用すると、定数を実行時に確認できます。

```
SUB R1, PC, #4 ; R1 = address of following STR instruction
STR PC, [R0]    ; Store address of STR instruction + offset,
LDR R0, [R0]    ; then reload it
SUB R0, R0, R1 ; Calculate the offset as the difference
```

特定のプロセッサ用のコードをアセンブルする場合は、定数の値を `armasm` で `{PCSTOREOFFSET}` として使用できます。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARM アーキテクチャのすべての T2 バリアントで使用できます。

これらの命令の 16 ビットバージョンはありません。

ARMv5 以降の T および T2 バリアントには、以下のような特徴があります。

- ARM 状態で r15 へのロードを実行すると、ロードされる値のビット 0 が 1 の場合、Thumb 状態に切り替わります。
- Thumb 状態で r15 へのロードを実行すると、ロードされる値のビット 0 が 0 の場合、ARM 状態に切り替わります。

4.2.6 LDR (プログラムカウンタ相対)

レジスタロード命令です。アドレスは、プログラムカウンタからのオフセットです。バイトとハーフワードのロードは、32ビットにゼロ拡張または符号拡張されます。

構文

```
LDR{type}{cond}{.W} Rd, {Rd2,} labe1
```

パラメータの説明：

<i>type</i>	次のいずれかを指定できます。
B	符号なしバイト
SB	符号付きバイト (LDRのみ)
H	符号なしハーフワード
SH	符号付きハーフワード (LDRのみ)
-	省略 (ワード)
D	ダブルワード
<i>cond</i>	任意に指定できる条件コードです (「条件実行」(P. 2-17) 参照)。
.W	任意に指定できる幅指定子を指定です。詳細については、「Thumb-2 の LDR (プログラムカウンタ相対)」を参照して下さい。
<i>Rd</i>	ロードまたはストアする ARM レジスタを指定します。
<i>Rd2</i>	2番目にロードまたはストアする ARM レジスタを指定します (<i>type</i> == D の場合のみ)。
<i>labe1</i>	プログラム相対式を指定します。詳細については、「レジスタ相対式とプログラム相対式」(P. 3-31) を参照して下さい。 <i>labe1</i> は現在の命令から ±4KB 以内に配置する必要があります。

使用法

アセンブラーは、プログラムカウンタからオフセットを算出します。*label* が範囲外である場合、アセンブラーはエラーを生成します。

Thumb-2 の LDR（プログラムカウンタ相対）

.W 幅指定子を使用して、LDR で Thumb-2 コードの 32 ビット命令を生成させることができます。

LDR.W は、16 ビット LDR 命令を使用してターゲットに到達できる場合でも、必ず 32 ビット命令を生成します。

前方参照の場合、Thumb コードで .W を指定せずに LDR を実行すると常に 16 ビット命令が生成されますが、生成された 16 ビット命令では 32 ビット Thumb-2 LDR 命令を使用して到達できるターゲットに到達できない場合があります。

ダブルワードレジスタの制約条件

Thumb-2 命令の場合、Rd または Rd2 に r15 は指定できません。

ARM 命令には、以下の制約条件が適用されます。

- Rd には偶数番号のレジスタを指定する必要があります。
- Rd に r14 は指定できません。
- Rd2 には R(d + 1) を指定する必要があります。

16 ビット命令

Thumb-2 コードと、ARMv6 以前に準拠していて Thumb を利用できるプロセッサの Thumb コードでは、この命令の 16 ビットバージョンを使用できます。

16 ビット命令には、以下の制約条件が適用されます。

- Rd は Lo レジスタである必要があります。
- type は省略する必要があります。つまり、実行できるのはワードのロードのみです。
- オフセットは 0 ~ +1020 の範囲内にある 4 の倍数である必要があります。

Thumb-2EE 命令

ベースレジスタの値が 0 の場合、HandlerBase - 4 にある NullCheck ハンドラへの分岐が実行されます。

r15 へのロード

ARM コードと Thumb-2 コードでは、どちらも *Rd* にプログラムカウンタを指定できます。この場合、*type* を省略する必要があります。

r15（プログラムカウンタ）へのロードを実行すると、ロードされたアドレスにある命令への分岐が発生します。

ARMv4 では、ロードされる値のビット [1:0] は 0 である必要があります。

ARMv5T 以降には以下のような特徴があります。

- r15 にロードされる値のビット [1:0] の値は 0b10 にできません。
- r15 にロードされる値のビット 0 が設定されている場合、プロセッサは Thumb 状態に切り替わります。

r15 へのロードを実行するときに、接尾文字 T は使用できません。

アーキテクチャ

この ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

この 32 ビット Thumb-2 命令は、ARM アーキテクチャのすべての T2 バリアントで使用できます。

この 16 ビット Thumb 命令は、ARM アーキテクチャのすべての T バリアントで使用できます。

ARMv5 以降の T および T2 バリアントには、以下のような特徴があります。

- ARM 状態で r15 へのロードを実行すると、ロードされる値のビット 0 が 1 の場合、Thumb 状態に切り替わります。
- Thumb 状態で r15 へのロードを実行すると、ロードされる値のビット 0 が 0 の場合、ARM 状態に切り替わります。

4.2.7 ADR

プログラム相対アドレスまたはレジスタ相対アドレスをレジスタにロードする命令です。

構文

`ADR{cond}{.W} register, label`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」（P. 2-17）参照）。

.W 任意に指定できる幅指定子を指定です。詳細については、「*ADR*」（P. 4-29）を参照して下さい。

register ロードするレジスタを指定します。

label レジスタ相対式またはプログラム相対式を指定します。詳細については、「レジスタ相対式とプログラム相対式」（P. 3-31）を参照して下さい。

使用法

ADR をアセンブルすると、常に 1 つの命令が生成されます。アセンブラーは、1 つの ADD または SUB 命令を生成してアドレスのロードを試行します。1 つの命令でアドレスを構成できない場合は、エラーが生成され、アセンブルは失敗します。

ADR は、アドレスがプログラム相対またはレジスタ相対であるため、位置非依存コードを生成します。

より広範囲の有効なアドレスをアセンブルするには、ADRL 擬似命令を使用します（「*ADRL 擬似命令*」（P. 4-156）参照）。

label がプログラム相対である場合は、このパラメータを *ADR* 命令と同じアセンブラー領域のアドレスに対して評価する必要があります（「*AREA*」（P. 7-66）参照）。

範囲

利用できる範囲は、使用する命令セットによって異なります。

ARM

ワード境界またはハーフワード境界で整列されているアドレスから ± 255 バイトの範囲

ワード境界で整列されているアドレスから ± 1020 バイトの範囲

32 ビットの Thumb-2

バイト、ハーフワード、またはワード境界で整列されているアドレスから ± 4095 バイトの範囲

16 ビットの Thumb

0 ~ 1020 バイトの範囲。*label* は、ワード境界で整列させる必要があります。ALIGN ディレクティブを使用して、*label* をワード境界で整列させることができます。

上記の範囲は、現在の命令のアドレスの 2 ワード後の位置に対応しています。ARM と Thumb-2 では、境界調整がこの位置から 16 バイト以上の相対位置にある場合、より広範囲のアドレスを利用できます。

Thumb-2 の ADR

.W 幅指定子を使用して、ADR で Thumb-2 コードの 32 ビット命令を生成させることができます。

.W を指定して ADR を実行すると、16 ビットの ADD または SUB 命令でアドレスを生成できる場合でも、常に 32 ビット命令が生成されます。

前方参照の場合、Thumb コードで .W を指定せずに ADR を実行すると、常に 16 ビット命令が生成されます。ただし、生成された 16 ビット命令では、32 ビット Thumb-2 ADD 命令で生成できるアドレスを生成できない場合があります。

4.2.8 PLD、PLI

データと命令をプリロードする命令です。プロセッサは、アドレスからのデータまたは命令のロードが実行されることをメモリシステムに事前に通知することができます。

アドレスには、プログラムカウンタからのイミディエートオフセットか、あるいはレジスタからのイミディエートオフセット、レジスタオフセット、またはシフトレジスタオフセットを指定できます。

PLD と PLI の実装は、任意で行います。実装しなかった場合、これらの命令は NOP として実行されます。

構文

PLD および PLI 命令には、以下に示す 3 つの形式があります。

- イミディエートオフセットまたはゼロオフセット
- レジスタオフセットまたはシフトレジスタオフセット
- プログラムカウンタ相対

この 3 つの形式の構文を、上記と同じ順番で以下に示します。

`PLtype{cond} [Rn {, #offset}]`

`PLtype{cond} [Rn, +/-Rm {, shift}]`

`PLtype{cond} label`

パラメータの説明 :

type D (データ) または I (命令) を指定します。

cond 任意に指定できる条件コードです (「条件実行」(P. 2-17) 参照)。

注

この命令は、ARM では無条件命令です。**cond** を指定できるのは、Thumb-2 コードで、前に IT 命令を使用した場合のみです。

Rn メモリアドレスのベースとなるレジスタを指定します。

offset イミディエートオフセットを指定します。オフセットを指定しなかった場合、その命令はゼロオフセット命令になります。

Rm オフセットとして使用される値を保持するレジスタを指定します。Rm に r15 は指定できません。

shift 任意に指定できるシフトです。

label プログラム相対式を指定します。詳細については、「レジスタ相対式とプログラム相対式」(P. 3-31) を参照して下さい。

イミディエートオフセット

このオフセットは、プリロードが発生する前に Rn の値に適用されます。この結果は、プリロード用のメモリアドレスとして使用されます。使用できるオフセットの範囲を以下に示します。

- ARM 命令 : -4095 ~ +4095
- Thumb-2 命令 : -255 ~ +4095

ゼロオフセット

Rn の値は、プリロード用のアドレスとして使用されます。

レジスタオフセットまたはシフトレジスタオフセット

ARM では、 Rm の値は Rn の値に対して加算または減算されます。Thumb-2 では、 Rm の値は Rn の値に加算することはできますが、減算することはできません。この結果は、プリロード用のメモリアドレスとして使用されます。

可能なシフトの範囲を以下に示します。

- Thumb-2 命令 : LSL 0 ~ 3
- ARM 命令 : 以下のいずれかの範囲
 - LSL 0 ~ 31
 - LSR 1 ~ 32
 - ASR 1 ~ 32
 - ROR 1 ~ 31
 - RRX

プログラムカウンタ相対

アセンブラーは、プログラムカウンタからオフセットを算出します。`label` が範囲外である場合、アセンブラーはエラーを生成します。

プリロード用のアドレス境界調整

プリロード命令では、境界調整チェックは実行されません。

アーキテクチャ

PLD は、ARMv5TE 以降で使用できます。PLI は、ARMv7 以降で使用できます。

これらの 32 ビット Thumb-2 命令は、ARM アーキテクチャのすべての T2 バリアントで使用できます。16 ビットの Thumb PLD または PLI 命令はありません。

4.2.9 LDM、STM

多重レジスタロード / ストア命令です。ARM 状態ではレジスタ r0 ~ r15 の任意の組み合わせを転送できますが、Thumb 状態ではいくつかの制限があります。

/PUSH およびPOP/ (P. 4-35) も参照して下さい。

構文

op{addr_mode}{cond} Rn{!}, reglist{^}

パラメータの説明：

- | | |
|------------------|--|
| <i>op</i> | 次のいずれかを指定できます。 |
| LDM | 複数のレジスタをロードします。 |
| STM | 複数のレジスタをストアします。 |
| <i>addr_mode</i> | 以下のいずれかを指定します。 |
| IA | 転送単位でアドレスをポストインクリメントします。これはデフォルト値なので、省略できます。 |
| IB | 転送単位でアドレスをプレインクリメントします(ARMのみ)。 |
| DA | 転送単位でアドレスをポストデクリメントします(ARMのみ)。 |
| DB | 転送単位でアドレスをプレデクリメントします。 |
| <i>cond</i> | 任意に指定できる条件コードです (「条件実行」(P. 2-17) 参照)。 |
| <i>Rn</i> | ベースレジスタです。つまり、転送に使用する初期アドレスが保持される ARM レジスタです。 <i>Rn</i> に r15 は指定できません。 |
| ! | 任意に指定できる接尾文字です。! を指定すると、最終アドレスが <i>Rn</i> にライトバックされます。 |
| <i>reglist</i> | ロードまたはストアするレジスタのリストを中括弧で囲んで指定します。レジスタ範囲も指定できます。複数のレジスタまたはレジスタ範囲を指定する場合は、コンマで区切る必要があります (「例」(P. 4-34) 参照)。
詳細については、「32 ビット Thumb-2 命令の reglist に関する制約条件」(P. 4-33) を参照して下さい。 |
| ^ | 任意に指定できる接尾文字です。ARM 状態のみで使用できます。ユーザモードやシステムモードでは使用できません。この接尾文字には以下の目的があります。 <ul style="list-style-type: none"> • 命令に LDM (または LDMIA) を指定し、<i>reglist</i> にプログラムカウンタ (r15) が含まれている場合、通常の多重レジスタ転送が行われるだけでなく、SPSR が CPSR にコピーされます。これは、例外ハンドラからの復帰に必要です。したがって、この接尾文字は必ず例外モードから使用して下さい。 |

- 例外モードで使用されない場合には、現在のモードのレジスタではなく、ユーザモードのレジスタとの間でデータ転送が実行されます。

32 ビット Thumb-2 命令の reglist に関する制約条件

32 ビット Thumb-2 命令では、以下の制約条件が適用されます。

- SP をリストに含めることはできません。
- STM 命令では、プログラムカウンタをリストに含めることはできません。
- LDM 命令では、プログラムカウンタと LR を両方ともリストに含めることはできません。

16 ビット命令

Thumb-2 コードと、Thumb を利用できる他のプロセッサのThumb コードでは、これらの命令のサブセットの 16 ビットバージョンを使用できます。

16 ビット命令には、以下の制約条件が適用されます。

- reglist* に指定するレジスタはすべて Lo レジスタである必要があります。
- Rn は Lo レジスタである必要があります。
- addr_mode は省略する（または IA を指定する）必要があります。つまり、転送単位でアドレスをポストインクリメントする必要があります。
- ライトバックを指定する必要があります。

また、PUSH および POP 命令をこの形式で記述できます。PUSH と POP の一部の形式は、16 ビット命令でもあります。詳細については、「PUSH および POP」(P. 4-35) を参照して下さい。

注

これらの 16 ビット命令は、Thumb-2EE では使用できません。

Thumb-2EE 命令

ベースレジスタの値が 0 の場合、HandlerBase - 4 にある NullCheck ハンドラへの分岐が実行されます。

r15 へのロード

r15（プログラムカウンタ）へのロードを実行すると、ロードされたアドレスにある命令への分岐が発生します。

ライトバックを使用したベースレジスタのロードまたはストア

reglist に *Rn* が含まれ、接尾文字 ! でライトバックが指定されている場合、以下のようになります。

- 命令に STM または STMIA が指定され、*Rn* が *reglist* 内で最も番号の小さいレジスタである場合には、*Rn* の初期値がストアされます。
- 上記以外の場合、ロードまたはストアされる *Rn* の値は予測不可能であるため、信頼できません。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以降の T2 バリアントで使用できます。

これらの 16 ビット Thumb 命令は、ARM アーキテクチャのすべての T バリアントで使用できます。

ARMv5 以降の T バリアントでは、r15 へのロードを実行すると、以下のようになります。

- ARM 状態で、ロードされる値のビット 0 が 1 の場合、Thumb 状態に切り替わります。
- Thumb 状態で、ロードされる値のビット 0 が 0 の場合、ARM 状態に切り替わります。

この状態切り替え動作は、cp15 の L4 ビット（ビット 15）を設定することによってディセーブルできます。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を参照して下さい。

例

```
LDM      r8,{r0,r2,r9}      ; LDMIA is a synonym for LDM
STMDB   r1!,{r3-r6,r11,r12}
```

誤用例

```
STM      r5!,{r5,r4,r9} ; value stored for r5 unpredictable
LDMDA   r2, {}           ; must be at least one register in list
```

4.2.10 PUSH および POP

レジスタをスタックにプッシュする命令です。

レジスタをスタックからポップする命令です。

構文

`PUSH{cond} reglist`

`POP{cond} reglist`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

reglist レジスタまたはレジスタ範囲のリストを中括弧で囲んで指定します。レジスタ範囲も指定できます。複数のレジスタまたはレジスタ範囲を指定する場合は、コンマで区切る必要があります。

使用法

PUSH と POP は、ベースレジスタが r13 (sp) で、調整されたアドレスがベースレジスタにライトバックされる STMDA と LDM (または LDMIA) の同義語です。このような場合は、PUSH および POP ニーモニックを使用するのが適切です。

レジスタは、番号順にスタックにストアされます。最も小さな番号のレジスタが最下位アドレスにストアされます。

reglist にプログラムカウンタを含む POP

この命令は、スタックからプログラムカウンタにポップされたアドレスへの分岐を発生させます。一般的には、サブルーチンからの復帰に使用します。サブルーチンでは lr がサブルーチン開始位置でスタックにプッシュされます。

ARMv5T 以降には、以下のようない特徴があります。

- プログラムカウンタにロードされる値のビット 1:0 が b00 の場合、プロセッサは ARM 状態に切り替わります。
- ビット 1:0 の値は b10 にできません。

ARMv4T 以前では、プログラムカウンタにロードされる値のビット 1:0 が無視されるため、POP を使用して状態を変更することはできません。

16 ビット命令

Thumb-2 コードと、Thumb を利用できる他のプロセッサの Thumb コードでは、これらの命令のサブセットの 16 ビットバージョンを使用できます。

16 ビット命令には、以下の制約条件が適用されます。

- *reglist* に指定するレジスタはすべて Lo レジスタである必要があります。ただし、PUSH は LR、POP はプログラムカウンタをそれぞれ含むことができます。

Thumb-2EE 命令

sp の値が 0 の場合、HandlerBase - 4 にある NullCheck ハンドラへの分岐が実行されます。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以降の T2 バリアントで使用できます。

これらの 16 ビット Thumb 命令は、ARM アーキテクチャのすべての T バリアントで使用できます。

ARMv5 以降の T バリアントでは、r15 へのロードを実行すると、以下のようになります。

- ARM 状態で、ロードされる値のビット 0 が 1 の場合、Thumb 状態に切り替わります。
- Thumb 状態で、ロードされる値のビット 0 が 0 の場合、ARM 状態に切り替わります。

この状態切り替え動作は、cp15 の L4 ビット（ビット 15）を設定することによってディセーブルできます。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*を参照して下さい。

16 ビットの例

```
PUSH    {r0,r3,r5}
PUSH    {r1,r4-r7} ; pushes r1, r4, r5, r6, and r7
PUSH    {r0,LR}
POP     {r2,r5}
POP     {r0-r7,pc} ; pop and return from subroutine
```

ARM と 32 ビット Thumb-2 の例

```
PUSH    {r0,r3,r5}
PUSH    {r1,r4-r11}
PUSH    {r0,LR}
POP     {r8,r12}
POP     {r0-r10,pc}
```

誤用例

```
PUSH    {} ; must be at least one register in list
```

4.2.11 RFE

例外から復帰する命令です。

構文

`RFE{addr_mode}{cond} Rn{!}`

パラメータの説明：

addr_mode 以下のいずれかを指定します。

- IA 転送単位でアドレスをポストインクリメントします（フル下降スタック）。
- IB 転送単位でアドレスをプレインクリメントします（ARMのみ）。
- DA 転送単位でアドレスをポストデクリメントします（ARMのみ）。
- DB 転送単位でアドレスをプレデクリメントします。

cond 任意に指定できる条件コードです（「条件実行」（P. 2-17）参照）。

――注――

この命令は、ARM では無条件命令です。*cond* を指定できるのは、Thumb-2 コードで、前に IT 命令を使用した場合のみです。

Rn ベースレジスタを指定します。*Rn* に r15 は使用できません。

! 任意に指定できる接尾文字です。! を指定すると、最終アドレスが *Rn* にライトバックされます。

使用法

前に SRS 命令を使用して復帰状態をストアした場合は、RFE を使用して例外から復帰できます（/SRS/（P. 4-40）参照）。

Thumb-2EE では、ベースレジスタの値が 0 の場合、HandlerBase - 4 にある NullCheck ハンドラへの分岐が実行されます。

演算

Rn に保持されているアドレスとその次のアドレスから、プログラムカウンタと CPSR をロードします。また、オプションを指定して *Rn* を更新できます。

例外

データアボートが発生します。

注

RFE によって、プログラムカウンタにアドレスが書き込まれます。例外復帰後に使用される命令セットに合わせて、このアドレスの境界調整を調整する必要があります。

- ARM に復帰するには、プログラムカウンタに書き込むアドレスをワード境界で整列する必要があります。
- Thumb-2 に復帰するには、プログラムカウンタに書き込むアドレスをハーフワード境界で整列する必要があります。
- Jazelle® に復帰する場合、プログラムカウンタに書き込むアドレスの境界調整について、制限はありません。

上記の規則に従わない結果は予測不可能であり、その結果は信頼できません。ただし、適切な例外開始メカニズムの後で復帰するために命令を使用する場合、ソフトウェア側に特別な予防策は必要ありません。

アドレスがワード境界で整列されていない場合、RFE は *Rn* の最下位 2 ビットを無視します。

アーキテクチャ上では、RFE によって生成される、メモリの各ワードへのアクセスの時間的順序が定義されていません。アクセス順序の影響を受けるメモリマップされた I/O 位置では、この命令を使用しないで下さい。

mode でユーザモードを指定した場合、CPSR への書き込みについては通常の規則が適用されます。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を参照して下さい。

mode で監視モードを指定した場合、結果は予測不可能であり、信頼できません（/SMC（P. 4-145）参照）。

アーキテクチャ

この ARM 命令は、ARMv6 以降で使用できます。

この 32 ビット Thumb-2 命令は、ARMv6 以降の T2 バリアントで使用できます。

この命令の 16 ビットバージョンはありません。

例

```
RFE r13!
```

誤用例

```
RFEFD    r15      ; do not use r15
```

4.2.12 SRS

復帰状態をストアする命令です。

構文

`SRS{addr_mode}{cond} {r13{!}}, #modenum`

パラメータの説明：

addr_mode 以下のいずれかを指定します。

- IA 転送単位でアドレスをポストインクリメントします。
- IB 転送単位でアドレスをプレインクリメントします(ARMのみ)。
- DA 転送単位でアドレスをポストデクリメントします(ARMのみ)。
- DB 転送単位でアドレスをプレデクリメントします。

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

——注——

この命令は、ARM では無条件命令です。*cond* を指定できるのは、Thumb-2 コードで、前に IT 命令を使用した場合のみです。

! 任意に指定できる接尾文字です。! を指定すると、*modenum* によって指定されたモードの r13 に最終アドレスがライトバックされます。

modenum バンク付き r13 がベースレジスタとして使用されるモードの番号を指定します。詳細については、「プロセッサモード」(P. 2-5) を参照して下さい。

演算

SRS は、*modenum* によって指定されたモードの r13 が保持するアドレスとその次のワードに、現在のモードの r14 と SPSR をそれぞれストアします。また、オプションを指定して *modenum* によって指定されたモードの r13 を更新できます。この命令は、一般的に スタックへのアクセスに使用される STM 命令と互換性があります（「LDM, STM」(P. 4-32) 参照）。

使用法

SRS 命令を使用して、自動的に選択されたスタックとは別のスタックに、例外ハンドラーの復帰状態をストアできます。

Thumb-2EE では、ベースレジスタの値が 0 の場合、HandlerBase - 4 にある NullCheck ハンドラーへの分岐が実行されます。

例外

データアポートが発生します。

注

アドレスがワード境界で整列されていない場合、SRS は、指定されたアドレスの最下位 2 ビットを無視します。

アーキテクチャ上では、SRS によって生成される、メモリの各ワードへのアクセスの時間的順序が定義されていません。アクセス順序の影響を受けるメモリマップされた I/O 位置では、この命令を使用しないで下さい。

ユーザモードとシステムモードには SPSR がないため、SRS の結果は予測不可能であり、信頼できません。

アーキテクチャ

この ARM 命令は、ARMv6 以降で使用できます。

この 32 ビット Thumb-2 命令は、ARMv6 以降の T2 バリアントで使用できます。

この命令の 16 ビットバージョンはありません。

例

```
R13_usr    EQU      16
          SRSFD    #R13_usr
```

誤用例

```
SRSFD    #32          ; there is no mode 32
          SRSFDEQ #R13_usr   ; SRS is always unconditional (this is legal in Thumb-2)
```

4.2.13 LDREX、STREX

排他的レジスタロード / ストア命令です。

構文

`LDREX{size}{cond} Rd, {Rd2,} [Rn {, #offset}]`

`STREX{size}{cond} Rd, Rm, {Rm2,} [Rn {, #offset}]`

パラメータの説明 :

size	ロードまたはストアするデータのサイズとして以下のいずれかを指定します。
B	符号なしバイト
H	符号なしハーフワード
-	省略 (ワード)
D	ダブルワード
cond	任意に指定できる条件コードです (「条件実行」(P. 2-17) 参照)。
Rd	デスティネーションレジスタを指定します。命令の実行後、Rd には以下のデータがストアされます。 <ul style="list-style-type: none"> • LDREX の場合は、メモリからロードされたデータがストアされます。 • STREX の場合は、以下のデータがストアされます。 <ul style="list-style-type: none"> 0 命令が正しく実行されたとき 1 命令がロックアウトされたとき
Rd2	ダブルワードのロードに使用する 2 番目のデスティネーションレジスタを指定します。
Rm	メモリにストアするデータが保持されているソースレジスタを指定します。
Rm2	ダブルワードのストアに使用する 2 番目のソースレジスタを指定します。
Rn	メモリアドレスを保持するレジスタを指定します。
offset	Rn の値に適用されるオフセットを指定します。offset は、Thumb-2 命令で size を省略した場合のみ使用できます。offset には、0 ~ 1020 の範囲内にある 4 の倍数を指定できます。offset を省略した場合、オフセットは 0 として処理されます。

LDREX

LDREX は、メモリからデータをロードします。

- 物理アドレスに共有 TLB 属性が設定されている場合、LDREX は、その物理アドレスに現在のプロセッサの排他的アクセスを示すタグを付け、他の物理アドレスに対するこのプロセッサの排他的アクセスタグをクリアします。
- 共有 TLB 属性が設定されていない場合、LDREX は、実行中のプロセッサがまだアクセスしていないタグ付きの物理アドレスがあることを示すタグを付けます。

STREX

STREX は、メモリへの条件付きストアを実行します。条件を以下に示します。

- 物理アドレスに共有 TLB 属性が設定されておらず、実行中のプロセッサによってまだアクセスされていないタグ付きの物理アドレスが存在する場合には、この命令によるストアが実行され、タグがクリアされます。
- 物理アドレスに共有 TLB 属性が設定されておらず、実行中のプロセッサによってアクセスされていないタグ付きの物理アドレスが存在しない場合には、ストアは発生しません。
- 物理アドレスに共有 TLB 属性が設定されており、その物理アドレスに実行中のプロセッサによる排他的アクセスのタグが付けられている場合には、ストアが発生してタグがクリアされます。
- 物理アドレスに共有 TLB 属性が設定されており、その物理アドレスに実行中のプロセッサによる排他的アクセスのタグが付けられていない場合には、ストアは発生しません。

制約条件

Rd、*Rn*、*Rm*、*Rd2*、*Rm2* のいずれのパラメータにも *r15* は指定できません。

Rd に、*Rm* または *Rm2* と同じレジスタは指定できません。STREX の場合は、*Rn* と同じレジスタを指定できません。

Rd と *Rd2* に同じレジスタは指定できません。

ARM LDREXD 命令には、以下の制約条件が適用されます。

- Rd* には偶数番号のレジスタ (*r14* 以外) を指定する必要があります。
- Rd2* には *R(d+1)* を指定する必要があります。

ARM STREXD 命令に適用される制約条件を以下に示します。

- Rm* には偶数番号のレジスタ (*r14* 以外) を指定する必要があります。
- Rm2* は *R(m+1)* である必要があります。

使用法

LDREX 命令と STREX 命令を使用して、マルチプロセッサの共有メモリシステムでプロセス間通信を実装できます。

パフォーマンス上の理由から、LDREX 命令と STREX 命令対の間に記述する命令の数は最小限に抑えて下さい。

——注——

STREX 命令で使用されるアドレスは、直前に実行された LDREX 命令で使用されたアドレスと同一である必要があります。異なるアドレスに STREX 命令を実行した結果は予測不可能（つまり信頼できない）です。

ARM 命令と Thumb-2 命令の相違点

Thumb-2 では、ワード LDREX 命令と STREX 命令に、ベースレジスタからの任意のオフセットを指定できます。ARM 排他的レジスタロード / ストア命令にはオフセットを指定できません。

Thumb-2 のダブルワード LDREXD および STREXD 命令は、データ用に任意の汎用レジスタを 2 本使用できます。ARM 命令で使用できるレジスタは、連続しているレジスタに制限されます。

Thumb-2EE 命令

ベースレジスタの値が 0 の場合、HandlerBase - 4 にある NullCheck ハンドラへの分岐が実行されます。

例外

データアボートが発生します。

アーキテクチャ

size を指定しない場合、これらの ARM 命令は ARMv6 以降で使用できます。

size を指定する場合、これらの ARM 命令は ARMv6T2 以降で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以降の T2 バリアントで使用できます。

例

```

MOV r1, #0x1           ; load the 'lock taken' value
try
    LDREX r0, [LockAddr] ; load the lock value
    CMP r0, #0            ; is the lock free?
    STREXEQ r0, r1, [LockAddr] ; try and claim the lock
    CMPEQ r0, #0          ; did this succeed?
    BNE try               ; no - try again
    ....                  ; yes - we have the lock

```

4.2.14 CLREX

排他をクリアする命令です。排他アクセスを要求しているアドレスが存在することを示す、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。

構文

`CLREX{cond}`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」（P. 2-17）参照）。

—————注—————

この命令は、ARM では無条件命令です。*cond* を指定できるのは、Thumb-2 コードで、前に IT 命令を使用した場合のみです。

使用法

CLREX 命令を使用して、密に結合されている排他アクセスモニタをオープンアクセス状態に戻すことができます。これにより、メモリへのダミーストアを行う必要がなくなります。同期のプリティプサポートの詳細については、ARM アーキテクチャリファレンスマニュアルを参照してください。

CLREX が、アドレスから排他アクセスの要求がある実行中のプロセッサのグローバルレコードもクリアするかどうかは、実装によって定義されます。

アーキテクチャ

この ARM 命令は、ARMv6T2 以降で使用できます。

この 32 ビット Thumb-2 命令は、ARMv6 以降の T2 バリアントで使用できます。

16 ビットの Thumb CLREX 命令はありません。

4.2.15 SWP、SWPB

レジスタとメモリ間のデータスワップ命令です。

SWP と SWPB を使用して、セマフォを実装できます。ただし、ARMv6 以降では、SWP と SWPB の使用が廃止されています。ARMv6 以降における複雑なセマフォの実装に使用できる命令については、「LDREX、STREX」(P. 4-42) を参照して下さい。

構文

`SWP{B}{cond} Rd, Rm, [Rn]`

パラメータの説明：

<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。
<i>B</i>	任意に指定できる接尾文字です。B を指定した場合は、バイトがスワップされます。指定しなかった場合は、32 ビットワードがスワップされます。
<i>Rd</i>	ARM レジスタを指定します。メモリからのデータが <i>Rd</i> にロードされます。
<i>Rm</i>	ARM レジスタを指定します。 <i>Rm</i> の内容がメモリに保存されます。 <i>Rm</i> に <i>Rd</i> と同じレジスタは指定できます。同じレジスタを指定した場合は、そのレジスタの内容がメモリ位置の内容とスワップされます。
<i>Rn</i>	ARM レジスタを指定します。 <i>Rn</i> の内容により、スワップされるデータがあるメモリのアドレスが指定されます <i>Rn</i> には、 <i>Rd</i> および <i>Rm</i> と異なるレジスタを指定する必要があります。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

Thumb または Thumb-2 には SWP または SWPB 命令はありません。

4.3 汎用データ処理命令

このセクションは以下のサブセクションから構成されています。

- フレキシブル第2オペランド (P. 4-48)
- *ADD, SUB, RSB, ADC, SBC, RSC* (P. 4-51)
キャリー付きまたはキャリーなしの加算、減算、および逆減算です。
- *SUBS PC, LR (Thumb-2 のみ)* (P. 4-55)
スタックをポップしない例外からの復帰です。
- *AND, ORR, EOR, BIC, ORN* (P. 4-56)
論理積、論理和、排他的論理和 (XOR)、ビットクリア、および論理否定です。
- *CLZ* (P. 4-59)
先行ゼロカウント命令です。
- *CMP, CMN* (P. 4-60)
比較命令と比較否定命令です。
- *MOV, MVN* (P. 4-62)
移動命令と移動 NOT 命令です。
- *MOVT* (P. 4-65)
上位にデータを代入する命令です。
- *TST, TEQ* (P. 4-66)
テスト命令と等価テスト命令です。
- *SEL* (P. 4-68)
CPSR の GE フラグの状態に基づいて、各オペランドからバイトを選択する命令です。
- *REV, REVI6, REVSH, RBIT* (P. 4-70)
バイトまたはビットを反転する命令です。
- *ASR, LSL, LSR, ROR, RRX* (P. 4-72)
算術右シフト命令です。
- *IT* (P. 4-74)
If-Then 命令です。
- *SDIV, UDIV* (P. 4-77)
符号付き除算と符号なし除算です。

4.3.1 フレキシブル第 2 オペランド

ARM および Thumb-2 の汎用データ処理命令のほとんどに、フレキシブル第 2 オペランドを使用できます。このオペランドは、各命令の構文の記述において *Operand2* と表記されています。ARM 命令と Thumb-2 命令では、*Operand2* で使用できるオプションが異なります。

構文

Operand2 には以下の 2 つの形式を使用できます。

#constant

Rm{, shift}

パラメータの説明 :

constant 数値定数を求める式を指定します。使用できる定数の範囲は、ARM と Thumb-2 で若干異なります。詳細については、「*Operand2 の定数*」(P. 4-49) を参照して下さい。

Rm 第 2 オペランドのデータを保持する ARM レジスタです。このレジスタのビットパターンは、さまざまな方法でシフトまたはロテートできます。

shift 任意に指定できる、Rm に適用されるシフト量です。以下の値を指定できます。

ASR #n n ビット算術右シフト。 $1 \leq n \leq 32$

LSL #n n ビット論理左シフト。 $0 \leq n \leq 31$

LSR #n n ビット論理右シフト。 $1 \leq n \leq 32$

ROR #n n ビット右ロテート。 $1 \leq n \leq 31$

RRX 拡張付き 1 ビット右ロテート。

type Rs ARM のみで使用可能です。各パラメータには以下の意味があります。

type ASR、LSL、LSR、ROR のいずれかを指定します。

Rs シフト量を渡す ARM レジスタを指定します。最下位バイトのみが使用されます。

注

シフト演算の結果は命令の *Operand2* として使用されますが、Rm 自体は変更されません。

Operand2 の定数

ARM 命令において、*constant* には、32 ビットのワード内で 8 ビットの値を右に任意の偶数ビット分ロテートして得られる任意の値を指定できます。

32 ビットの Thumb-2 命令において、*constant* には以下の値を指定できます。

- 32 ビットのワード内で 8 ビットの値を左に任意のビット数シフトして得られる任意の定数
- 0x00XY00XY 形式の任意の定数
- 0xXY00XY00 形式の任意の定数
- 0xXYXXYYXY 形式の任意の定数

また、少数の命令では、*constant* により広い範囲の値を指定することができます。詳細については、各命令の説明を参照して下さい。

8 ビットの値を右に 2 ビット、4 ビット、または 6 ビット分ロテートして得られる定数は、ARM データ処理命令で使用することができますが、Thumb-2 では使用できません。その他すべての ARM 定数は、Thumb-2 でも使用できます。

ASR

Rm の内容が 2 の補数の符号付き整数とみなされる場合、*n* ビットの算術右シフトはその値が 2^n で除算されると同じです。元のビット [31] はレジスタの左 *n* ビットにコピーされます。

LSR、LSL

Rm の内容が符号なし整数とみなされる場合、*n* ビット分の論理右シフトはその値が 2^n で除算されると同じです。レジスタの左 *n* ビットは、0 に設定されます。

Rm の内容が符号なし整数とみなされる場合、*n* ビット分の論理左シフトはその値が 2^n で乗算されると同じです。このとき警告なしでオーバフローが発生する場合があります。レジスタの右 *n* ビットは、0 に設定されます。

ROR

n ビットの右ロテートにより、レジスタの右 *n* ビットが、結果の左 *n* ビットに移動します。同時に、他のすべてのビットが *n* ビット分右に移動します (P. 4-50 図 4-1 参照)。

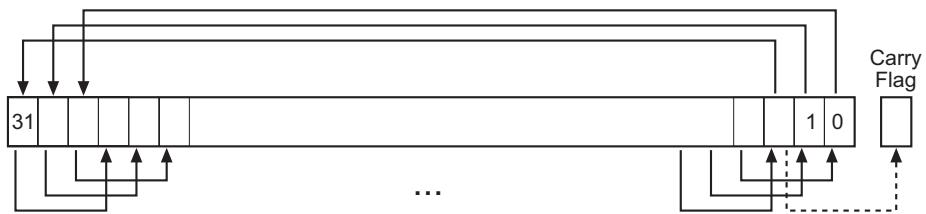


図 4-1 ROR

RRX

拡張付き右ロテートでは、 Rm の内容が 1 ビットずつ右にシフトされます。キャリーフラグは Rm のビット [31] にコピーされます（図 4-2 参照）。

接尾文字 S が指定されている場合は、 Rm のビット [0] の古い値がキャリーフラグにシフトアウトされます（「キャリーフラグ」参照）。

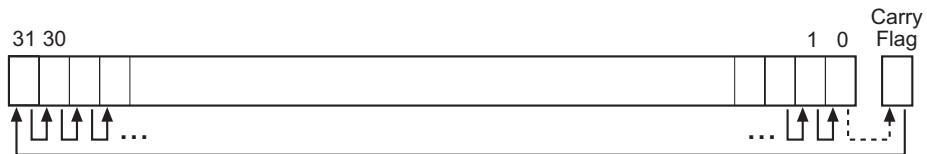


図 4-2 RRX

キャリーフラグ

以下の命令の場合、キャリーフラグが、 Rm からシフトアウトされた最後のビットに更新されます。

- 接尾文字 S を使用する場合は、MOV、MVN、AND、ORR、ORN、EOR、または BIC
- 接尾文字 S を使用する必要がない場合は、TEQ または TST

命令置換

constant を否定または論理反転する場合を除き、特定の命令対 (ADD と SUB、ADC と SBC、AND と BIC、MOV と MVN、および CMP と CMN) は等価です。

constant の値を使用できなくても、論理反転または否定にできる可能性がある場合は、アセンブラーは命令対のもう一方の命令に置き換え、*constant* を反転または否定します。

逆アセンブルリストとソースコードを比較する場合は、この点に注意して下さい。

アセンブラーの `--diag_warning 1645` コマンドラインオプションを使用して、命令置換の発生を確認できます。

4.3.2 ADD、SUB、RSB、ADC、SBC、RSC

キャリー付きまたはキャリーなしの加算、減算、および逆減算です。

「並列加算と並列減算」(P. 4-105) も参照して下さい。

構文

op{S}{cond} Rd, Rn, Operand2

パラメータの説明：

op 次のいずれかを指定します。

ADD 加算

ADC キャリー付き加算

SUB 減算

RSB 逆減算

SBC キャリー付き減算

RSC キャリー付き逆減算 (ARM のみ)

S 任意に指定できる接尾文字です。S が指定されている場合は、演算結果に基づいて条件コードフラグが更新されます（「条件実行」(P. 2-17) 参照）。

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

Rd デスティネーションレジスタを指定します。

Rn 第1オペランドを保持するレジスタを指定します。

Operand2 フレキシブル第2オペランドを指定します。このオプションの詳細については、「フレキシブル第2オペランド」(P. 4-48) を参照して下さい。「広域定数」(P. 4-52) も参照して下さい。

使用法

ADD 命令は、*Rn* と *Operand2* の値を加算します。

SUB 命令は、*Rn* の値から *Operand2* の値を減算します。

RSB (逆減算) 命令は、*Operand2* の値から *Rn* の値を減算します。*Operand2* にはさまざまなオプションがあるので、この命令は便利です。

ADC、SBC、およびRSC を使用して、マルチワード算術演算を合成できます（「マルチワード演算の例」(P. 4-54) 参照）。

ADC (キャリー付き加算) 命令は、*Rn* の値と *Operand2* の値、そしてキャリーフラグも含めて加算します。

SBC (キャリー付き減算) 命令は、*Rn* の値から *Operand2* の値を減算します。キャリー フラグがクリアされている場合は、結果から 1 が引かれます。

RSC (キャリー付き逆減算) 命令は、*Operand2* の値から *Rn* の値を減算します。キャリー フラグがクリアされている場合は、結果から 1 が引かれます。

状況によっては、ある命令をアセンブラーによって別の命令に置換できる場合があります。逆アセンブルリストを参照する場合は、この点に注意して下さい。詳細については、「命令置換」(P. 4-50) を参照して下さい。

広域定数

Thumb-2 の ADD 命令および SUB 命令では、*Operand2* は通常の値の範囲に加え、0 ~ 4095 の任意の値を取ることができます。RSB 命令、ADC 命令、SBC 命令、または RSC 命令では広域定数を使用できません。

接尾文字 S と広域定数は併用できません。

Thumb-2 命令での r15 の使用

ほとんどの命令で、*Rd*、または任意のオペランドに r15 は使用できません。

ただし、ADD 命令または SUB 命令で、定数 *Operand2* の値が 0 ~ 4095 の範囲にあり、接尾文字 S が指定されていない場合は、*Rn* に r15 を使用できます。これらの命令は PC 相対アドレスを生成するのに役立ちます。このとき、プログラムカウンタのビット [1] の値が 0 として読み出されるので、計算に使用するベースアドレスが常にワード境界で整列します。

「SUBS PC, LR (Thumb-2 のみ)」(P. 4-55) も参照して下さい。

ARM 命令での r15 の使用

Rn に r15 を指定している場合、使用される値は「命令のアドレス + 8」となります。

Rd に r15 を指定した場合、以下のようになります。

- 演算結果に対応するアドレスへの分岐が実行されます。
- 接尾文字 S を指定している場合は、現在のモードの SPSR が CPSR にコピーされます。この動作を利用して、例外から復帰することができます (*RealView Compilation Tools v3.0 Developer Guide* の「プロセッサ例外処理」を参照して下さい)。

———— 注意 ————

ユーザモードまたはシステムモードで *Rd* に r15 を使用する場合は、接尾文字 S を使用しないで下さい。このような命令による影響は予測不能（信頼できない）なうえ、アセンブリ時にアセンブラーによる警告は生成されません。

レジスタ制御シフトを行うデータ処理命令の場合には、*Rd* やオペランドに *r15* は使用できません（「フレキシブル第2オペランド」（P. 4-48）参照）。

条件フラグ

S が指定されている場合、これらの命令は演算結果に基づいて N、Z、C、および V の各フラグを更新します。

16 ビット命令

これらの命令は、Thumb コード内では次の形式で使用できます。また、Thumb-2 コード内で使用するときは 16 ビット命令になります。

ADDS *Rd*, *Rn*, #*imm* *imm* の範囲は 0 ~ 7 です。*Rd* および *Rn* は共に Lo レジスタである必要があります。

ADDS *Rd*, *Rn*, *Rm* *Rd*、*Rn*、および *Rm* はすべて Lo レジスタである必要があります。

ADD *Rd*, *Rd*, *Rm* ARMv6 以前：*Rd* または *Rm* の少なくとも一方が Hi レジスタである必要があります。ARMv6T2 以降：この制限は適用されません。

ADDS *Rd*, *Rd*, #*imm* *imm* の範囲は 0 ~ 255 です。*Rd* は Lo レジスタである必要があります。

ADCS *Rd*, *Rd*, *Rm* *Rd*、*Rn*、および *Rm* はすべて Lo レジスタである必要があります。

ADD SP, SP, #*imm* *imm* の範囲は 0 ~ 508 です（ワード境界で整列します）。

ADD *Rd*, SP, #*imm* *imm* の範囲は 0 ~ 1020 です（ワード境界で整列します）。*Rd* は Lo レジスタである必要があります。

ADD *Rd*, PC, #*imm* *imm* の範囲は 0 ~ 1020 です（ワード境界で整列します）。*Rd* は Lo レジスタである必要があります。この命令では、プログラムカウンタのビット [1:0] を 0 として読み出します。

SUBS *Rd*, *Rn*, *Rm* *Rd*、*Rn*、および *Rm* はすべて Lo レジスタである必要があります。

SUBS *Rd*, *Rn*, #*imm* *imm* の範囲は 0 ~ 7 です。*Rd* および *Rn* は共に Lo レジスタである必要があります。

SUBS *Rd*, *Rd*, #*imm* *imm* の範囲は 0 ~ 255 です。*Rd* は Lo レジスタである必要があります。

SBCS *Rd*, *Rd*, *Rm* *Rd*、*Rn*、および *Rm* はすべて Lo レジスタである必要があります。

SUB SP, SP, #*imm* *imm* の範囲は 0 ~ 508 です（ワード境界で整列します）。

RSBS *Rd*, *Rn*, #0 *Rd* および *Rn* は共に Lo レジスタである必要があります。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの 16 ビット Thumb 命令は、ARM アーキテクチャのすべての T バリアントで使用できます。

例

```

ADD      r2, r1, r3
SUBS    r8, r6, #240      ; sets the flags on the result
RSB     r4, r4, #1280     ; subtracts contents of r4 from 1280
ADCHI   r11, r0, r3       ; only executed if C flag set and Z
                           ; flag clear
RSCSLE  r0,r5,r0,LSL r4  ; conditional, flags set

```

誤用例

```

RSCSLE  r0,r15,r0,LSL r4  ; r15 not permitted with register
                           ; controlled shift

```

マルチワード演算の例

以下の 2 つの命令は、r2 と r3 に保持される 1 つの 64 ビット整数を、r0 と r1 に保持される別の 64 ビット整数に加算し、その結果を r4 と r5 に返します。

```

ADDS    r4, r0, r2      ; adding the least significant words
ADC     r5, r1, r3      ; adding the most significant words

```

以下の命令は、一方の 96 ビット整数を他方の値から減算します。

```

SUBS    r3, r6, r9
SBCS    r4, r7, r10
SBC     r5, r8, r11

```

上記の例では分かりやすくするために、マルチワードの値に使用するレジスタを連続させていますが、必ずしもそうする必要はありません。例えば、以下のように記述することもできます。

```

SUBS    r6, r6, r9
SBCS    r9, r2, r1
SBC     r2, r8, r11

```

4.3.3 SUBS PC, LR (Thumb-2 のみ)

スタックをポップしない例外からの復帰です。

構文

SUBS{cond} PC, LR, #immed_8

パラメータの説明 :

immed_8 8 ビットのイミディエート定数を指定します。

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

使用法

スタックに復帰状態がない場合、SUBS PC, LR を使用して例外から復帰できます。

SUBS PC, LR では、リンクレジスタから値を減算し、結果をプログラムカウンタにロードして、SPSR をCPSR にコピーします。

注

SUBS PC, LR によって、プログラムカウンタにアドレスが書き込まれます。例外復帰後に使用される命令セットに合わせて、このアドレスの境界調整を調整する必要があります。

- ARM 命令セットに復帰するには、プログラムカウンタに書き込むアドレスをワード境界で整列する必要があります。
- Thumb-2 命令セットに復帰するには、プログラムカウンタに書き込むアドレスをハーフワード境界で整列する必要があります。
- Jazelle 命令セットに復帰する場合、プログラムカウンタに書き込むアドレスの境界調整について、制限はありません。

上記の規則に従わない結果は予測不可能であり、その結果は信頼できません。ただし、適切な例外開始メカニズムの後で復帰するために命令を使用する場合、ソフトウェア側に特別な予防策は必要ありません。

MOVS pc, lr は、Thumb-2 の SUBS pc, lr, #0 の同義語です。

アーキテクチャ

この 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

4.3.4 AND、ORR、EOR、BIC、ORN

論理積、論理和、排他的論理和 (XOR)、ビットクリア、および論理和否定です。

構文

op{S}{cond} Rd, Rn, Operand2

パラメータの説明：

<i>op</i>	次のいずれかを指定します。
AND	論理積
ORR	論理和
EOR	排他的論理和 (XOR)
BIC	論理積否定
ORN	論理和否定 (Thumb-2 のみ)
<i>S</i>	任意に指定できる接尾文字です。S が指定されている場合は、演算結果に基づいて条件コードフラグが更新されます（「条件実行」(P. 2-17) 参照）。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。
<i>Rd</i>	デスティネーションレジスタを指定します。
<i>Rn</i>	第 1 オペランドを保持するレジスタを指定します。
<i>Operand2</i>	フレキシブル第 2 オペランドを指定します。オプションの詳細については、「フレキシブル第 2 オペランド」(P. 4-48) を参照して下さい。

使用法

AND、EOR、および ORR の各命令は、*Rn* と *Operand2* の値に対し、それぞれビットごとの論理積、排他的論理和 (XOR)、および論理和をとります。

BIC (ビットクリア) 命令は、*Rn* 内のビットと、*Operand2* の値に含まれる、対応する各ビットの補数との論理積をとります。

ORN Thumb-2 命令は、*Rn* 内のビットと、*Operand2* の値に含まれる、対応する各ビットの補数との論理和をとります。

状況によっては、アセンブラーが AND を BIC に、BIC を AND に、ORR を ORN に、または ORN を ORR に置き換える場合があります。逆アセンブルリストとソースコードを参照する場合は、この点に注意して下さい。詳細については、「命令置換」(P. 4-50) を参照して下さい。

Thumb-2 命令での r15 の使用

すべての命令で、*Rd*、または任意のオペランドに r15 は使用できません。

ARM 命令での r15 の使用

Rn に r15 を指定している場合、使用される値は「命令のアドレス + 8」となります。

Rd に r15 を指定した場合、以下のようにになります。

- 演算結果に対応するアドレスへの分岐が実行されます。
- 接尾文字 S を指定している場合は、現在のモードの SPSR が CPSR にコピーされます。この動作を利用して、例外から復帰することができます (*RealView Compilation Tools v3.0 Developer Guide* の「プロセッサ例外処理」を参照して下さい)。

———— 注意 —————

ユーザモードまたはシステムモードで *Rd* に r15 を使用する場合は、接尾文字 S を使用しないで下さい。このような命令による影響は予測不能（信頼できない）なうえ、アセンブリ時にアセンブラーによる警告は生成されません。

レジスタ制御シフトを行うデータ処理命令の場合には、*Rd* やオペランドに r15 は使用できません（「フレキシブル第2 オペランド」(P. 4-48) 参照)。

条件フラグ

S が指定されている場合、これらの命令は、以下のようになります。

- 結果に応じて N フラグおよび Z フラグを更新します。
- *Operand2* の計算中に C フラグを更新することができます（「フレキシブル第2 オペランド」(P. 4-48) 参照)。
- V フラグには影響しません。

16 ビット命令

これらの命令は、Thumb コード内では次の形式で使用できます。また、Thumb-2 コード内で使用するときは 16 ビット命令になります。

ANDS Rd, Rd, Rm *Rd* および *Rm* は共に Lo レジスタである必要があります。

EORS Rd, Rd, Rm *Rd* および *Rm* は共に Lo レジスタである必要があります。

ORRS Rd, Rd, Rm *Rd* および *Rm* は共に Lo レジスタである必要があります。

BICS Rd, Rd, Rm *Rd* および *Rm* は共に Lo レジスタである必要があります。

最初の 3 つの例は、*OPS Rd, Rm, Rd* を指定しても問題ありません。命令は変わりません。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの 16 ビット Thumb 命令は、ARM アーキテクチャのすべての T バリアントで使用できます。

ARM/Thumb-2 の例

```

AND      r9,r2,#0xFF00
ORREQ   r2,r0,r5
EORS    r0,r0,r3,ROR r6
ANDS    r9, r8, #0x19
EORS    r7, r11, #0x18181818
BIC     r0, r1, #0xab
ORN     r7, r11, r14, ROR #4
ORNS    r7, r11, r14, ASR #32

```

誤用例

```

EORS    r0,r15,r3,ROR r6      ; r15 not permitted with register
                                ; controlled shift

```

4.3.5 CLZ

先行ゼロカウント命令です。

構文

`CLZ{cond} Rd, Rm`

パラメータの説明 :

- | | |
|-------------|--|
| <i>cond</i> | 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。 |
| <i>Rd</i> | デスティネーションレジスタを指定します。 <i>Rd</i> に r15 は指定できません。 |
| <i>Rm</i> | オペランドレジスタを指定します。 <i>Rm</i> に r15 は指定できません。 |

使用法

CLZ 命令は *Rm* の値に含まれる先行ゼロの数をカウントし、結果を *Rd* に返します。ソースレジスタに設定されているビットがない場合の結果は 32 となり、ビット 31 が設定されている場合の結果はゼロになります。

条件フラグ

この命令によるフラグへの影響はありません。

アーキテクチャ

この ARM 命令は、ARMv5 以上で使用できます。

この 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

この命令の 16 ビット Thumb バージョンはありません。

例

```
CLZ      r4, r9
CLZNE   r2, r3
```

レジスタ *Rm* の値を正規化するには、CLZ Thumb-2 命令を使用し、結果として返された *Rd* 値の分 *Rm* を左シフトします。*Rm* が 0 である場合にフラグを設定するには、MOV ではなく MOVS を使用します。

```
CLZ r5, r9
MOVS r9, r9, LSL r5
```

4.3.6 CMP、CMN

比較命令と比較結果をビット反転させる命令です。

構文

`CMP{cond} Rn, Operand2`

`CMN{cond} Rn, Operand2`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

Rn 第1オペランドを保持する ARM レジスタを指定します。

Operand2 フレキシブル第2オペランドを指定します。オプションの詳細については、「フレキシブル第2オペランド」(P. 4-48) を参照して下さい。

使用法

これらの命令は、レジスタ内の値と *Operand2* を比較します。結果に基づいて条件フラグを更新しますが、結果はどのレジスタにも返しません。

CMP 命令は、*Rn* の値から *Operand2* の値を減算します。結果が破棄されることを除けば、SUBS 命令と同じです。

CMN 命令は、*Operand2* の値を *Rn* の値に加算します。結果が破棄されることを除けば、ADDS 命令と同じです。

状況によっては、アセンブラーによって CMP を CMN に、または CMN を CMP に置換できる場合があります。逆アセンブルリストを参照するときは、この点に注意して下さい。詳細については、「命令置換」(P. 4-50) を参照して下さい。

ARM 命令での r15 の使用

Rn に r15 を指定している場合、使用される値は「命令のアドレス + 8」となります。

レジスタ制御シフトを行うデータ処理命令の場合には、オペランドに r15 は使用できません（「フレキシブル第2オペランド」(P. 4-48) 参照）。

Thumb-2 命令での r15 の使用

これらの命令では、すべてのオペランドに r15 は使用できません。

条件フラグ

これらの命令は、演算結果に基づいて N、Z、C、および V の各フラグを更新します。

16 ビット命令

これらの命令は、Thumb コード内では次の形式で使用できます。また、Thumb-2 コード内で使用するときは 16 ビット命令になります。

CMP Rn, Rm *Rn* および *Rm* は、共に Lo レジスタであるか、共に Hi レジスタである必要があります。

CMN Rn, Rm *Rn* および *Rm* は共に Lo レジスタである必要があります。

CMP Rn, #imm *Rn* は Lo レジスタである必要があります。*imm* の範囲は 0～255 です。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの 16 ビット Thumb 命令は、ARM アーキテクチャのすべての T バリアントで使用できます。

例

```
CMP    r2, r9
CMN    r0, #6400
CMPGT  r13, r7, LSL #2
```

誤用例

```
CMP    r2, r15, ASR r0 ; r15 not permitted with register controlled shift
```

4.3.7 MOV、MVN

データ代入命令とデータを代入してビット反転させる命令です。

構文

`MOV{S}{cond} Rd, Operand2`

`MVN{S}{cond} Rd, Operand2`

パラメータの説明：

S 任意に指定できる接尾文字です。S が指定されている場合は、演算結果に基づいて条件コードフラグが更新されます（「条件実行」(P. 2-17) 参照）。

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

Rd デスティネーションレジスタを指定します。

Operand2 フレキシブル第 2 オペランドを指定します。オプションの詳細については、「フレキシブル第 2 オペランド」(P. 4-48) を参照して下さい。広域定数も参照して下さい。

使用法

MOV 命令は、*Operand2* の値を *Rd* にコピーします。

MVN 命令は、*Operand2* の値を取得し、その値にビットごとの論理 NOT 演算を実行して、結果を *Rd* に返します。

状況によっては、アセンブラーによって MOV を MVN に、または MVN を MOV に置換できる場合があります。逆アセンブルリストを参照するときは、この点に注意して下さい。詳細については、「命令置換」(P. 4-50) を参照して下さい。

広域定数

MOV 命令では、*Operand2* は通常の値の範囲に加え、0 ~ 65535 の任意の値を取ることができます。これらの広域定数と MVN 命令は併用できません。

接尾文字 S と広域定数は併用できません。

Rd として使用する r15 と広域定数は併用できません。

r15 の使用

Thumb-2 の MOV 命令または MVN 命令で、*Rd* または *Operand2* に r15 を使用することはできません。このセクションのこれ以降の部分は、ARM 命令に適用されます。

Rd に r15 を指定している場合、使用される値は「命令のアドレス + 8」となります。

Rd に r15 を指定した場合、以下のようにになります。

- 演算結果に対応するアドレスへの分岐が実行されます。
- 接尾文字 S を指定している場合は、現在のモードの SPSR が CPSR にコピーされます。この動作を利用して、例外から復帰することができます (*RealView Compilation Tools v3.0 Developer Guide* の「プロセッサ例外処理」を参照して下さい)。

注意

ユーザモードまたはシステムモードで *Rd* に r15 を使用する場合は、接尾文字 S を使用しないで下さい。このような命令による影響は予測不能（信頼できない）なうえ、アセンブリ時にアセンブラーによる警告は生成されません。

レジスタ制御シフトを行うデータ処理命令の場合には、*Rd* やオペランドに r15 は使用できません（「フレキシブル第2 オペランド」(P. 4-48) 参照)。

条件フラグ

S が指定されている場合、これらの命令は、以下のようになります。

- 結果に応じて N フラグおよび Z フラグを更新します。
- *Operand2* の計算中に C フラグを更新することができます（「フレキシブル第2 オペランド」(P. 4-48) 参照)。
- V フラグは更新しません。

16 ビット命令

これらの命令は、Thumb コード内では次の形式で使用できます。また、Thumb-2 コード内で使用するときは 16 ビット命令になります。

MOVS Rd, imm *Rd* は Lo レジスタである必要があります。*imm* の範囲は 0～255 です。

MOVS Rd, Rm *Rd* および *Rm* は共に Lo レジスタである必要があります。

MOV Rd, Rm ARMv5 以前のバージョンでは、*Rd* と *Rm* の少なくとも一方が Hi レジスタである必要があります。ARMv6 以上のバージョンに、この制限は適用されません。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの 16 ビット Thumb 命令は、ARM アーキテクチャのすべての T バリアントで使用できます。

例

```
MVNNE    r11, #0xF000000B ; ARM only. This constant is not available in T2.
```

誤用例

```
MVN      r15,r3,ASR r0 ; r15 not permitted with register controlled shift
```

4.3.8 MOVT

上位にデータを代入する命令です。レジスタの下位ハーフワードに影響を及ぼさずに、上位ハーフワードに 16 ビットのイミディエート値を書き込みます。

構文

`MOVT{cond} Rd, #immed_16`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」（P. 2-17）参照）。

Rd デスティネーションレジスタを指定します。*Rd* をプログラムカウンタにすることはできません。

immed_16 16 ビットのイミディエート定数を指定します。

使用法

MOVT は、*immed_16* を *Rd* [31:16] に書き込みます。この書き込みが *Rd* [15:0] に影響することはありません。

MOV と MOVT の命令対で、任意の 32 ビット定数を生成できます。

「MOV32 擬似命令」（P. 4-158）も参照して下さい。

条件フラグ

この命令によるフラグへの影響はありません。

アーキテクチャ

この ARM 命令は、ARMv6T2 以上で使用できます。

この 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

この命令の 16 ビット Thumb バージョンはありません。

4.3.9 TST、TEQ

ビットテスト命令と等価テスト命令です。

構文

`TST{cond} Rn, Operand2`

`TEQ{cond} Rn, Operand2`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

Rn 第1オペランドを保持する ARM レジスタを指定します。

Operand2 フレキシブル第2オペランドを指定します。オプションの詳細については、「フレキシブル第2オペランド」(P. 4-48) を参照して下さい。

使用法

これらの命令は、レジスタ内の値を *Operand2* に対してテストします。結果に基づいて条件フラグを更新しますが、結果はどのレジスタにも返しません。

TST 命令は、*Rn* の値と *Operand2* の値を使用してビットごとの論理積をとります。結果が破棄されることを除けば、ANDS 命令と同じです。

TEQ 命令は、*Rn* の値と *Operand2* の値を使用してビットごとの排他的論理和 (XOR) をとります。結果が破棄されることを除けば、EORS 命令と同じです。

2つの値が等しいかどうかをテストするには TEQ 命令を使用します。その際、CMP とは異なり、V フラグまたは C フラグに影響することはありません。

TEQ は値の符号をテストするのにも役立ちます。比較後の N フラグは、2つのオペランドの符号ビットの排他的論理和 (XOR) になります。

r15 の使用

Thumb-2 の TST 命令または TEQ 命令で、*Rn* または *Operand2* に r15 を使用することはできません。このセクションのこれ以降の部分は、ARM 命令に適用されます。

Rn に r15 を指定している場合、使用される値は「命令のアドレス + 8」となります。

レジスタ制御シフトを行うデータ処理命令の場合には、オペランドに r15 は使用できません（「フレキシブル第2オペランド」(P. 4-48) 参照）。

条件フラグ

これらの命令は、以下のようになります。

- 結果に応じて N フラグおよび Z フラグを更新します。
- Operand2* の計算中に C フラグを更新することが可能です（「フレキシブル第2オペランド」(P. 4-48) 参照）。
- V フラグは更新しません。

16 ビット命令

TST 命令は、Thumb コード内では次の形式で使用できます。また、Thumb-2 コード内で使用するときは 16 ビット命令になります。

TST Rn, Rm *Rn* および *Rm* は共に Lo レジスタである必要があります。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

16 ビット Thumb の TST 命令は、ARM アーキテクチャのすべての T バリアントで使用できます。

例

```
TST      r0, #0x3F8
TEQEQ   r10, r9
TSTNE   r1, r5, ASR r1
```

誤用例

```
TEQ      r15, r1, ROR r0      ; r15 not permitted with register
                                ; controlled shift
```

4.3.10 SEL

CPSR の GE フラグの状態に基づいて、各オペランドからバイトを選択する命令です。

構文

`SEL{cond} Rd, Rn, Rm`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

Rd デスティネーションレジスタを指定します。

Rn 第1オペランドを保持するレジスタを指定します。

Rm 第2オペランドを保持するレジスタを指定します。

演算

SEL 命令は、CPSR の GE フラグに基づいて、*Rn* または *Rm* からバイトを選択します。

- GE[0] が設定されている場合は、*Rn[7:0]* から *Rd[7:0]* が取得され、それ以外の場合は *Rm[7:0]* から取得されます。
- GE[1] が設定されている場合は、*Rn[15:8]* から *Rd[15:8]* が取得され、それ以外の場合は *Rm[15:8]* から取得されます。
- GE[2] が設定されている場合は、*Rn[23:16]* から *Rd[23:16]* が取得され、それ以外の場合は *Rm[23:16]* から取得されます。
- GE[3] が設定されている場合は、*Rn[31:24]* から *Rd[31:24]* が取得され、それ以外の場合は *Rm[31:24]* から取得されます。

使用法

Rd、*Rn*、または *Rm* に r15 を使用しないで下さい。

SEL 命令は、符号付き並列命令の後に使用します（「並列加算と並列減算」(P. 4-105) 参照）。この命令を使用して、複数のバイトデータまたはハーフワードデータの最大値や最小値を選択できます。

条件フラグ

この命令によるフラグへの影響はありません。

アーキテクチャ

この ARM 命令は、ARMv6 以上で使用できます。

この 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

この命令の 16 ビット Thumb バージョンはありません。

例

```
SEL      r0, r4, r5  
SELLT   r4, r0, r4
```

以下の命令シーケンスは、r1 と r2 の各バイトの符号なし最小値と等しくなるように r4 の対応するバイトを設定します。

```
USUB8   r4, r1, r2  
SEL      r4, r2, r1
```

4.3.11 REV、REV16、REVSH、RBIT

ワード内またはハーフワード内で、バイトまたはビットを反転する命令です。

構文

op{cond} Rd, Rn

パラメータの説明：

<i>op</i>	以下のいずれかを指定します。
REV	ワード内のバイト順序を反転させます。
REV16	各ハーフワード内のバイト順序を独自に反転させます。
REVSH	下位ハーフワード内のバイト順序を反転させ、それを 32 ビットに符号拡張します。
RBIT	32 ビットワード内のビット順序を反転させます。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。
<i>Rd</i>	デスティネーションレジスタを指定します。 <i>Rd</i> に r15 は指定できません。
<i>Rn</i>	オペランドを保持するレジスタを指定します。 <i>Rn</i> に r15 を使用しないで下さい。

使用法

以下の命令を使用して、エンディアン方式を変更できます。

REV	32 ビットのビッグエンディアンデータをリトルエンディアンデータに変換、または 32 ビットのリトルエンディアンデータをビッグエンディアンデータに変換します。
REV16	16 ビットのビッグエンディアンデータをリトルエンディアンデータに変換、または 16 ビットのリトルエンディアンデータをビッグエンディアンデータに変換します。
REVSH	次のいずれかの変換を行います。 <ul style="list-style-type: none"> • 16 ビットの符号付きビッグエンディアンデータを 32 ビットの符号付きリトルエンディアンデータに変換します。 • 16 ビットの符号付きビッグエンディアンデータを 32 ビットの符号付きビッグエンディアンデータに変換します。

条件フラグ

これらの命令によるフラグへの影響はありません。

16 ビット命令

これらの命令は、Thumb コード内では次の形式で使用できます。また、Thumb-2 コード内で使用するときは 16 ビット命令になります。

REV Rd, Rm *Rd* および *Rm* は共に Lo レジスタである必要があります。

REV16 Rd, Rm *Rd* および *Rm* は共に Lo レジスタである必要があります。

REVSH Rd, Rm *Rd* および *Rm* は共に Lo レジスタである必要があります。

アーキテクチャ

RBIT を除く上記の ARM 命令は、ARMv6 以上で使用できます。

ARM RBIT 命令は、ARMv6T2 以上で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの 16 ビット Thumb 命令は、ARMv6 以上のすべての T バリアントで使用できます。

例

```
REV      r3, r7
REV16    r0, r0
REVSH    r0, r5      ; Reverse Signed Halfword
REVHS    r3, r7      ; Reverse with Higher or Same condition
RBIT     r7, r8
```

4.3.12 ASR、LSL、LSR、ROR、RRX

算術右シフト命令、論理左シフト命令、論理右シフト命令、右ロテート命令、および拡張付き右ロテート命令です。

これらの命令は、第 2 オペランドレジスタがシフトされる MOV 命令の同義語です。

構文

op{S}{cond} Rd, Rm, Rs

op{S}{cond} Rd, Rm, #sh

RRX{S}{cond} Rd, Rm

パラメータの説明：

op ASR、LSL、LSR、ROR のいずれかを指定します。

S 任意に指定できる接尾文字です。S が指定されている場合は、演算結果に基づいて条件コードフラグが更新されます（「条件実行」(P. 2-17) 参照）。

Rd デスティネーションレジスタを指定します。

Rm 第 1 オペランドを保持するレジスタを指定します。このオペランドは右にシフトされます。

Rs *Rm* の値に適用するシフト値を保持するレジスタを指定します。最下位バイトのみが使用されます。

sh 定数シフトを指定します。指定可能な値の範囲は、命令によって異なります。

ASR 1 ~ 32 のシフトが可能です。

LSL 0 ~ 31 のシフトが可能です。

LSR 1 ~ 32 のシフトが可能です。

ROR 1 ~ 31 のシフトが可能です。

使用法

ASR は、レジスタの内容を 2 のべき乗で除算した、符号付きの値を求めます。空の左のビット位置には、符号ビットがコピーされます。

LSL は、レジスタを 2 のべき乗で乗算した値を求めます。LSR は、レジスタを 2 の可変乗で除算した、符号なしの値を求めます。いずれの命令でも、空のビット位置には 0 が挿入されます。

ROR は、レジスタの内容を任意の値でロテートした値を求めます。ロテートの結果右端から溢れたビットは、空の左のビット位置に挿入されます。

RRX は、レジスタの内容を右に 1 ビットシフトした値を求めます。古いキャリーフラグはビット [31] にシフトされます。接尾文字 S を指定した場合、古いビット [0] がキャリーフラグに配置されます。

条件フラグ

S が指定されている場合、これらの命令は演算結果に基づいて N と Z の各フラグを更新します。

シフト値が 0 の場合、C フラグに影響はありません。それ以外の場合、C フラグはシフトアウトされた最後のビットに更新されます。

16 ビット命令

これらの命令は、Thumb コード内では次の形式で使用できます。また、Thumb-2 コード内で使用するときは 16 ビット命令になります。

ASRS Rd, Rm, #sh Rd および Rm は共に Lo レジスタである必要があります。1 ~ 32 のシフトが可能です。

ASRS Rd, Rm, Rs Rd、Rm、および Rs はすべて Lo レジスタである必要があります。

LSLS Rd, Rm, #sh Rd および Rm は共に Lo レジスタである必要があります。0 ~ 31 のシフトが可能です。

LSLS Rd, Rm, Rs Rd、Rm、および Rs はすべて Lo レジスタである必要があります。

LSRS Rd, Rm, #sh Rd および Rm は共に Lo レジスタである必要があります。1 ~ 32 のシフトが可能です。

LSRS Rd, Rm, Rs Rd、Rm、および Rs はすべて Lo レジスタである必要があります。

RORS Rd, Rm, Rs Rd、Rm、および Rs はすべて Lo レジスタである必要があります。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの 16 ビット Thumb 命令は、ARM アーキテクチャのすべての T バリアントで使用できます。

例

ASR	r7, r8, r9
LSLS	r1, r2, r3
LSR	r4, r5, r6
ROR	r4, r5, r6

4.3.13 IT

IT (If-Then) 命令は、後続する最大 4 個の命令 (IT ブロック) を条件付き命令にします。条件をすべて同じにすることも、論理的に逆の条件を混在させることも可能です。

——注——

これは 16 ビット Thumb-2 命令です。

構文

`IT{x{y{z}}}{cond}`

パラメータの説明 :

<code>x</code>	IT ブロックで 2 番目の命令の条件を指定します。
<code>y</code>	IT ブロックで 3 番目の命令の条件を指定します。
<code>z</code>	IT ブロックで 4 番目の命令の条件を指定します。
<code>cond</code>	IT ブロックで最初の命令の条件を指定します。

IT ブロックで 2 ~ 4 番目の命令の条件は、以下のいずれかにすることができます。

`T` Then。命令に適用される条件は `cond` です。

`E` Else。命令に適用される条件は `cond` の逆です。

使用法

IT ブロック内の、`CMP`、`CMN`、および `TST` を除く 16 ビット命令は、条件コードフラグへの影響はありません。コードフラグを変更する場合、IT ブロックの命令を条件命令にしなくとも、IT を AL 条件と共に使用できます。

制約条件

IT ブロック内では、以下の命令を使用できません。

- IT 命令
- 条件分岐
- `CBZ` および `CBNZ` 命令
- `TBB` および `TBH` 命令
- `CPS`、`CPSID`、および `CPSIE` 命令
- `SETEND` 命令

無条件分岐を IT ブロック内で使用するには、ブロックの最後の命令として使用する必要があります。

条件フラグ

この命令によるフラグへの影響はありません。

例外

IT 命令とそれに対応する IT ブロックの間、または IT ブロックの内部で例外が発生する場合があります。例外が発生すると、適切な例外ハンドラが開始し、適切な復帰情報が r14 と SPSR に格納されます。

例外からの復帰に使用する目的で設計された命令は、例外から復帰するために通常どおり使用可能であり、IT ブロックの実行は正常に再開されます。これは、プログラム カウンタを変更する命令が、IT ブロック内の命令に分岐するための唯一の方法です。

アーキテクチャ

この 16 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

例

```
ITTE NE      ; assemblers can allow IT to be omitted
ANDNE r0,r0,r1 ; 16-bit AND, not ANDS
ADDSNE r2,r2,#1 ; 32-bit ADDS (16-bit ADDS does not set flags in IT block)
MOVEQ r2,r3    ; 16-bit MOV(CPY)
ITT AL       ; emit 2 non-flag setting 16-bit instructions
ADDAL r0,r0,r1 ; 16-bit ADD, not ADDS
SUBAL r2,r2,#1 ; 16-bit SUB, not SUB
ADD r0,r0,r1  ; expands into 32-bit ADD
IT NE
ADD r0,r0,r1  ; syntax error: no condition code used in IT block
```

注

IT ブロックへの分岐

「例外」の記述を除き、分岐命令、またはプログラム カウンタを変更する他の命令から、IT ブロック内の命令を分岐のターゲットにすることはできません。そのような分岐の結果は予測不可能であり、信頼できません。

IT ブロックから外部への分岐

プログラムカウンタを変更する命令を IT ブロック内で使用するには、ブロックの最後の命令として使用する必要があります。そのような命令の結果は予測不可能であり、信頼できません。

IT ブロック内の分岐

IT ブロック内の分岐は、無条件分岐としてエンコードする必要があります (IT 条件によって条件分岐に変更されます)。したがって、IT ブロックの有効範囲を拡張するため、ブロックへの条件分岐を配置することが有益な場合があります。

```
ITT      EQ  
MOVEQ   r0, r1  
BEQ     dloop
```

IT ブロック内の予測不可能な命令

B、BL、CPS など、IT ブロック内で予測不可能な命令を使用すると、これらの命令の結果が信頼できないため、警告が表示されます。また、BX、CZB、RFE など、プログラムカウンタを変更する命令についても警告が表示されます。

IT ブロック内のディレクティブは考慮されません。

BKPT IT ブロック内の BKPT 命令は、条件が偽の場合でも必ず実行されます。

4.3.14 SDIV、UDIV

符号付き除算と符号なし除算です。

構文

`SDIV{cond} Rd, Rn, Rm`

`UDIV{cond} Rd, Rn, Rm`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」（P. 2-17）参照）。

Rd デスティネーションレジスタを指定します。

Rn 被除数を保持するレジスタを指定します。

Rm 除数を保持するレジスタを指定します。

アーキテクチャ

16 ビット Thumb-2 命令 SDIV および UDIV は、ARMv7-M のみで使用できます。

32 ビットの SDIV 命令および UDIV 命令はありません。

4.4 乗算命令

このセクションは以下のサブセクションから構成されています。

- *MUL, MLA, MLS* (P. 4-79)
乗算命令、積和命令、および積差命令です (32 ビット×32 ビットを計算し、結果の下位 32 ビットを返します)。
- *UMULL, UMLAL, SMULL, SMLAL* (P. 4-81)
符号なし / 符号付き long 乗算命令と long 積和命令です (32 ビット×32 ビットを実行し、64 ビットの結果または 64 ビットの累積値を返します)。
- *SMULxy, SMLAx_y* (P. 4-83)
符号付き乗算命令と符号付き積和命令です (16 ビット×16 ビットを実行し、32 ビットの結果を返します)。
- *SMULW_y, SMLAW_y* (P. 4-85)
符号付き乗算命令と符号付き積和命令です (32 ビット×16 ビットを実行し、結果の上位 32 ビットを返します)。
- *SMLALxy* (P. 4-86)
符号付き積和命令です (16 ビット×16 ビットを実行し、64 ビットの累算値を返します)。
- *SMUAD{X}, SMUSD{X}* (P. 4-88)
積の加算または減算を伴うデュアル 16 ビット符号付き乗算命令です。
- *SMMUL, SMMLA, SMMLS* (P. 4-90)
乗算命令、積和命令、および積差命令です (32 ビット×32 ビットを実行し、結果の上位 32 ビットを返します)。
- *SMLAD, SMLSD* (P. 4-92)
デュアル 16 ビット符号付き乗算を実行し、積の加算または減算と、32 ビットの累算を行う命令です。
- *SMLALD, SMLS LD* (P. 4-94)
デュアル 16 ビット符号付き乗算を実行し、積の加算または減算と、64 ビットの累算を行う命令です。
- *UMAAL* (P. 4-96)
符号なし積和累算 long 命令です。
- *MIA, MIAPH, MIAx_y* (P. 4-97)
XScale コプロセッサ 0 命令 (内部累算を伴う乗算) です。

4.4.1 MUL、MLA、MLS

符号付きまたは符号なしの 32 ビットオペランドを使用して乗算、積和、および積差を実行し、結果の下位 32 ビットを返します。

構文

`MUL{S}{cond} Rd, Rm, Rs`

`MLA{S}{cond} Rd, Rm, Rs, Rn`

`MLS{cond} Rd, Rm, Rs, Rn`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

S 任意に指定できる接尾文字です。S が指定されている場合は、演算結果に基づいて条件コードフラグが更新されます（「条件実行」(P. 2-17) 参照）。

Thumb-2 の MLA 命令では S を使用できません。

Rd と *Rm* が同一のレジスタであり、かつすべてのオペランドが Lo レジスターである場合に限り、Thumb-2 の MUL 命令で S を使用できます。これは 16 ビット命令です。

Rd デスティネーションレジスタを指定します。

Rm, *Rs* 乗算する値を保持するレジスタを指定します。

Rn 加算する値または引かれる値を保持するレジスタを指定します。

使用法

MUL 命令は *Rm* と *Rs* の値を乗算し、演算結果の下位 32 ビットを *Rd* に返します。

MLA 命令は *Rm* と *Rs* の値を乗算し、*Rn* の値を加算して、演算結果の下位 32 ビットを *Rd* に返します。

MLS 命令は *Rm* と *Rs* の値を乗算し、その結果を *Rn* の値から減算して、最終的な演算結果の下位 32 ビットを *Rd* に返します。

Rd、*Rm*、*Rs*、または *Rn* に r15 を使用しないで下さい。

条件フラグ

S が指定されている場合、**MUL** 命令および**MLA** 命令は、以下のようになります。

- 結果に応じて N フラグおよび Z フラグを更新します。
- ARMv4 以前のプロセッサでは、C フラグおよび V フラグを破棄します。
- ARMv5 以上のプロセッサでは、C フラグまたは V フラグへの影響はありません。

16 ビット命令

MUL 命令は、Thumb コード内では次の形式で使用できます。また、Thumb-2 コード内で使用するときは 16 ビット命令になります。

MULS Rd, Rs, Rd Rd および Rs は共に Lo レジスタである必要があります。

アーキテクチャ

ARM 命令 **MUL** および **MLA** は、ARM アーキテクチャのすべてのバージョンで使用できます。

ARM 命令 **MLS** は、ARMv6T2 および ARMv7 で使用できます。

これらの Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

Thumb 命令の **MULS** は、ARM アーキテクチャのすべての T バリアントで使用できます。

例

```
MUL      r10, r2, r5
MLA      r10, r2, r1, r5
MULS    r0, r2, r2
MULLT   r2, r3, r2
MLS     r4, r5, r6, r7
```

誤用例

```
MUL      r15, r0, r3 ; use of r15 not permitted
```

4.4.2 UMULL、UMLAL、SMULL、SMLAL

符号付き／符号なし long 乗算命令と任意に指定できる累算命令です(32 ビット×32 ビットを実行し、64 ビットの累算値と結果を返します)。

構文

Op{S}{cond} RdLo, RdHi, Rm, Rs

パラメータの説明：

Op UMULL、UMLAL、SMULL、SMLAL のいずれかを指定します。

S 任意に指定できる接尾文字です。ARM 状態でのみ使用できます。*S* が指定されている場合は、演算結果に基づいて条件コードフラグが更新されます(「条件実行」(P. 2-17) 参照)。

cond 任意に指定できる条件コードです(「条件実行」(P. 2-17) 参照)。

RdLo, RdHi デスティネーションレジスタを指定します。UMLAL および SMLAL では、これらのレジスタに累積値も保存されます。

Rm, Rs オペランドを保持する ARM レジスタを指定します。

使用法

RdHi、*RdLo*、*Rm*、または*Rs* に r15 を使用しないで下さい。

RdLo と *RdHi* には、それぞれ異なるレジスタを指定する必要があります。

UMULL 命令は、*Rm* と *Rs* の値を符号なし整数と解釈します。これらの整数を乗算して、演算結果の下位 32 ビットを *RdLo* に返し、上位 32 ビットを *RdHi* に返します。

UMLAL 命令は、*Rm* と *Rs* の値を符号なし整数と解釈します。これらの整数を乗算して得られた 64 ビットの演算結果を、*RdHi* と *RdLo* が保持している 64 ビットの符号なし整数に加算します。

UMLAL 命令は、*Rm* と *Rs* の値を符号なし整数と解釈します。これらの整数を乗算して得られた 64 ビットの演算結果を、*RdHi* と *RdLo* が保持している 64 ビットの符号なし整数に加算します。

SMLAL 命令は、*Rm* と *Rs* の値を、2 の補数となる符号付き整数と解釈します。これらの整数を乗算して得られた 64 ビットの演算結果を、*RdHi* と *RdLo* が保持している 64 ビットの符号付き整数に加算します。

条件フラグ

S が指定されている場合、これらの命令は、以下のようになります。

- 結果に応じて N フラグおよび Z フラグを更新します。
- C フラグまたは V フラグは更新しません。

アーキテクチャ

これらの ARM 命令は、サポートされている ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

```
UMULL      r0, r4, r5, r6  
UMLALS    r4, r5, r3, r8
```

誤用例

```
UMULL      r1, r15, r10, r2      ; use of r15 not permitted
```

4.4.3 SMULxy、SMLAxy

符号付き乗算命令と積和命令です（16 ビット×16 ビットを実行し、32 ビットの累算値と結果を返します）。

構文

`SMUL<x><y>{cond} Rd, Rm, Rs`

`SMLA<x><y>{cond} Rd, Rm, Rs, Rn`

パラメータの説明：

<code><x></code>	B または T を指定します。Rm の下位ビット（ビット [15:0]）を使用する場合は B を、Rm の上位ビット（ビット [31:16]）を使用する場合は T を指定します。
<code><y></code>	B または T を指定します。Rs の下位ビット（ビット [15:0]）を使用する場合は B を、Rs の上位ビット（ビット [31:16]）を使用する場合は T を指定します。
<code>cond</code>	任意に指定できる条件コードです（「条件実行」（P. 2-17）参照）。
<code>Rd</code>	デスティネーションレジスタを指定します。
<code>Rm, Rs</code>	乗算する値を保持するレジスタを指定します。
<code>Rn</code>	加算する値を保持するレジスタを指定します。

使用法

`Rd, Rm, Rs`、または `Rn` に r15 を使用しないで下さい。

`SMULxy` 命令は、`Rm` と `Rs` の指定された上位半分または下位半分の 16 ビット符号付き整数を乗算し、32 ビットの演算結果を `Rd` に返します。

`SMLAxy` 命令は、`Rm` と `Rs` の指定された上位半分または下位半分の 16 ビット符号付き整数を乗算し、32 ビットの演算結果を `Rn` の 32 ビット値に加算し、その結果を `Rd` に返します。

条件フラグ

これらの命令による N、Z、C、または V フラグへの影響はありません。

累算中にオーバフローが発生した場合、SMLAx_y によって Q フラグが設定されます。Q フラグの状態を読み出すには、MRS 命令を使用します（「MRS」（P. 4-140）参照）。

—————注—————

SMLAx_y によって Q フラグがクリアされることはありません。Q フラグをクリアするには、MSR 命令を使用します（「MSR」（P. 4-141）参照）。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上、および ARMv5 の E バリアントで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

```

SMULTBEQ    r8, r7, r9
SMLABBNE   r0, r2, r1, r10
SMLABT     r0, r0, r3, r5

```

誤用例

```

SMLATB      r0,r7,r8,r15    ; use of r15 not permitted
SMLATTS     r0,r6,r2        ; use of S suffix not permitted

```

4.4.4 SMULW_y、SMLAW_y

符号付きワイド乗算と符号付きワイド積和を実行します（32 ビット×16 ビットを実行し、演算結果の上位 32 ビットを返します）。

構文

`SMULW<y>{cond} Rd, Rm, Rs`
`SMLAW<y>{cond} Rd, Rm, Rs, Rn`

パラメータの説明：

<code><y></code>	B または T を指定します。Rs の下位ビット（ビット [15:0]）を使用する場合は B を、Rs の上位ビット（ビット [31:16]）を使用する場合は T を指定します。
<code>cond</code>	任意に指定できる条件コードです（「条件実行」（P. 2-17）参照）。
<code>Rd</code>	デスティネーションレジスタを指定します。
<code>Rm, Rs</code>	乗算する値を保持するレジスタを指定します。
<code>Rn</code>	加算する値を保持するレジスタを指定します。

使用法

`Rd, Rm, Rs`、または `Rn` に r15 を使用しないで下さい。

`SMULWy` 命令は、`Rs` の指定された上位半分または下位半分の符号付き整数と、`Rm` の符号付き整数を乗算し、48 ビットの演算結果の上位 32 ビットを `Rd` に返します。

`SMLAWy` 命令は、`Rs` の選択された上位半分または下位半分の符号付き整数と `Rm` の符号付き整数を乗算し、32 ビットの演算結果を `Rn` の 32 ビット値に加算し、その結果を `Rd` に返します。

条件フラグ

これらの命令によるフラグへの影響はありません。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上、および ARMv5 の E バリアントで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

<code>SMULWB</code>	<code>r2, r4, r7</code>
<code>SMULWTVS</code>	<code>r0, r0, r9</code>

4.4.5 SMLALxy

16 ビット演算子と 64 ビットの累算値を使った符号付き積和命令

構文

`SMLAL<x><y>{cond} RdLo, RdHi, Rm, Rs`

パラメータの説明：

- <x> B または T を指定します。Rm の下位ビット（ビット [15:0]）を使用する場合は B を、Rm の上位ビット（ビット [31:16]）を使用する場合は T を指定します。
- <y> B または T を指定します。Rs の下位ビット（ビット [15:0]）を使用する場合は B を、Rs の上位ビット（ビット [31:16]）を使用する場合は T を指定します。
- cond 任意に指定できる条件コードです（「条件実行」（P. 2-17）参照）。
- RdHi, RdLo デスティネーションレジスタを指定します。これらのレジスタは累算値も保持します。
- Rm, Rs 乗算する値を保持するレジスタを指定します。

使用法

RdHi, RdLo, Rm、または Rs に r15 を使用しないで下さい。

RdHi と RdLo には、それぞれ異なるレジスタを指定する必要があります。

SMLALxy 命令は Rs の指定された上位半分または下位半分の符号付き整数を、Rm の指定された上位半分または下位半分の符号付き整数と乗算し、32 ビットの演算結果を RdHi と RdLo が保持する 64 ビット値に加算します。

条件フラグ

この命令によるフラグへの影響はありません。

—— 注 ———

SMLALxy 命令によって例外が発生することはありません。この命令でオーバフローが発生すると、警告がないまま演算結果が溢れた桁によって上書きされます。

アーキテクチャ

この ARM 命令は、ARMv6 以上、および ARMv5 の E バリアントで使用できます。

この 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

この命令の 16 ビット Thumb バージョンはありません。

例

```
SMLALTB    r2, r3, r7, r1  
SMLALBTVS  r0, r1, r9, r2
```

誤用例

```
SMLALTT    r8, r9, r3, r15 ; use of r15 not permitted
```

4.4.6 SMUAD{X}、SMUSD{X}

任意でオペランドの半分を交換できる、デュアル 16 ビット符号付き積和命令と積差命令です。

構文

op{X}{cond} Rd, Rm, Rs

パラメータの説明：

<i>op</i>	次のいずれかを指定します。
SMUAD	デュアル乗算後に積を加算します。
SMUSD	デュアル乗算後に積を減算します。
<i>X</i>	任意に指定できるパラメータです。X が指定されている場合は、乗算が行われる前に、第 2 オペランドの上位ハーフワードと下位ハーフワードが入れ替わります。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。
<i>Rd</i>	デスティネーションレジスタを指定します。
<i>Rm, Rs</i>	オペランドを保持するレジスタを指定します。

演算

Rd、*Rm*、*Rs* のいずれにも r15 を使用しないで下さい。

SMUAD は、*Rm* の下位ハーフワードと *Rs* の下位ハーフワードを乗算し、*Rm* の上位ハーフワードと *Rs* の上位ハーフワードを乗算します。次に、得られた 2 つの積を加算し、その合計を *Rd* にストアします。

SMUSD は、*Rm* の下位ハーフワードと *Rs* の下位ハーフワードを乗算し、*Rm* の上位ハーフワードと *Rs* の上位ハーフワードを乗算します。次に、最初の積から 2 番目の積を減算し、その差を *Rd* にストアします。

条件フラグ

これらの命令によるフラグへの影響はありません。

加算がオーバフローした場合、SMUAD 命令によって Q フラグが設定されます。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上、および ARMv5 の E バリアントで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

SMUAD	r2, r3, r2
SMUSDXNE	r0, r1, r2

4.4.7 SMMUL、SMMLA、SMMLS

32 ビット × 32 ビットを実行し、演算結果の上位 32 ビットのみを返す、符号付き上位ワード乗算命令です。任意で加算または減算を伴います。

構文

op{R}{cond} Rd, Rm, Rs, Rn

パラメータの説明：

<i>op</i>	次のいずれかを指定します。
SMMUL	乗算を行い、結果を切り捨てるか丸めます。
SMMLA	乗算と累算を行い、結果を切り捨てるか丸めます。
SMMLS	乗算を行い、 <i>Rn</i> から減算し、結果を切り捨てるか丸めます。
<i>R</i>	任意に指定できるパラメータです。 <i>R</i> が指定されている場合は演算結果の丸めが行われ、指定されていない場合は切り捨てられます。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。
<i>Rd</i>	デスティネーションレジスタを指定します。
<i>Rm, Rs</i>	オペランドを保持するレジスタを指定します。
<i>Rn</i>	加算する値または引かれる値を保持するレジスタを指定します。

演算

SMMUL 命令は、*Rm* と *Rs* の値を乗算し、得られた 64 ビットの結果の上位 32 ビットを *Rd* にストアします。

SMMLA は、*Rm* と *Rs* の値を乗算し、得られた積の上位 32 ビットに *Rn* の値を加算し、その結果を *Rd* にストアします。

SMMLS は、*Rm* と *Rs* の値を乗算して得られた積を、32 ビット左シフトした *Rn* の値から減算し、結果の上位 32 ビットを *Rd* にストアします。

オプションの *R* パラメータが指定されている場合は、上位 32 ビットを取り出す前に 0x80000000 が加算されます。この処理によって結果の丸めが行われます。

使用法

Rd、*Rm*、*Rs*、*Rn* のいずれにも r15 は使用しないで下さい。

条件フラグ

これらの命令によるフラグへの影響はありません。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上、および ARMv5 の E バリアントで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

```
SMMULGE    r6, r4, r3  
SMMULR     r2, r2, r2
```

4.4.8 SMLAD、SMLSD

デュアル 16 ビット符号付き乗算を実行し、積の加算または減算と、32 ビットの累算を行う命令です。

構文

op{X}{cond} Rd, Rm, Rs, Rn

パラメータの説明：

<i>op</i>	次のいずれかを指定します。 SMLAD デュアル乗算を行い、積の和を累算します。 SMLSD デュアル乗算を行い、積の差を累算します。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。
<i>X</i>	任意に指定できるパラメータです。X が指定されている場合は、乗算が行われる前に、第 2 オペランドの上位ハーフワードと下位ハーフワードが入れ替わります。
<i>Rd</i>	デスティネーションレジスタを指定します。
<i>Rm, Rs</i>	オペランドを保持するレジスタを指定します。
<i>Rn</i>	累算オペランドを保持するレジスタを指定します。

演算

SMLAD は、*Rm* の下位ハーフワードと *Rs* の下位ハーフワードを乗算し、*Rm* の上位ハーフワードと *Rs* の上位ハーフワードを乗算します。次に、得られた 2 つの積を *Rn* の値に加算し、その結果を *Rd* にストアします。

SMLSD は、*Rm* の下位ハーフワードと *Rs* の下位ハーフワードを乗算し、*Rm* の上位ハーフワードと *Rs* の上位ハーフワードを乗算します。次に、最初の積から 2 番目の積を減算し、その差を *Rn* の値に加算して、結果を *Rd* にストアします。

使用法

Rd、*Rm*、*Rs*、*Rn* のいずれにも r15 は使用しないで下さい。

条件フラグ

これらの命令によるフラグへの影響はありません。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上、および ARMv5 の E バリアントで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

SMLSD	r1, r2, r0, r7
SMLSDX	r11, r10, r2, r3
SMLADLT	r1, r2, r4, r1

4.4.9 SMLALD、SMLSLO

デュアル 16 ビット符号付き乗算を実行し、積の加算または減算と、64 ビットの累算を行う命令です。

構文

op{X}{cond} RdLo, RdHi, Rm, Rs

パラメータの説明：

<i>op</i>	次のいずれかを指定します。
SMLALD	デュアル乗算を行い、積の和を累算します。
SMLSLO	デュアル乗算を行い、積の差を累算します。
<i>X</i>	任意に指定できるパラメータです。X が指定されている場合は、乗算が行われる前に、第 2 オペランドの上位ハーフワードと下位ハーフワードが入れ替わります。
<i>cond</i>	任意の条件コードを指定します（「条件実行」(P. 2-17) 参照）。
<i>RdLo, RdHi</i>	64 ビットの結果を保持するデスティネーションレジスタを指定します。これらのレジスタには、64 ビットの累算オペランドも保持されます。
<i>Rm, Rs</i>	オペランドを保持するレジスタを指定します。

使用法

RdLo, RdHi, Rm, Rs のいずれにも r15 は使用しないで下さい。

演算

SMLALD は、*Rm* の下位ハーフワードと *Rs* の下位ハーフワードを乗算し、*Rm* の上位ハーフワードと *Rs* の上位ハーフワードを乗算します。次に、得られた 2 つの積を *RdLo* と *RdHi* の値に加算し、その結果を *RdLo* と *RdHi* にストアします。

SMLSLO は、*Rm* の下位ハーフワードと *Rs* の下位ハーフワードを乗算し、*Rm* の上位ハーフワードと *Rs* の上位ハーフワードを乗算します。次に、最初の積から 2 番目の積を減算し、その差を *RdLo* と *RdHi* の値に加算して、結果を *RdLo* と *RdHi* にストアします。

条件フラグ

これらの命令によるフラグへの影響はありません。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上、および ARMv5 の E バリアントで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

SMLALD	r10, r11, r5, r1
SMLS LD	r3, r0, r5, r1

4.4.10 UMAAL

符号なし積和累算 long 命令です。

構文

`UMAAL{cond} RdLo, RdHi, Rm, Rs`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

RdLo, RdHi 64 ビットの結果を保持するデスティネーションレジスタを指定します。これらのレジスタには、32 ビットの 2 つの累算オペランドも保持されます。

Rm, Rs 乗算オペランドを保持するレジスタを指定します。

演算

UMAAL 命令は、*Rm* と *Rs* の 32 ビット値を乗算し、*RdHi* と *RdLo* の 2 つの 32 ビット値を加算し、その 64 ビットの結果を *RdLo* と *RdHi* にストアします。

使用法

RdLo, RdHi, Rm, Rs のいずれにも r15 は使用しないで下さい。

RdLo と *RdHi* は別のレジスタである必要があります。これらが同じレジスタの場合、結果は予測不可能であり、信頼できません。

条件フラグ

この命令によるフラグへの影響はありません。

アーキテクチャ

この ARM 命令は、ARMv6 以上、および ARMv5 の E バリアントで使用できます。

この 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

この命令の 16 ビット Thumb バージョンはありません。

例

<code>UMAAL</code>	<code>r8, r9, r2, r3</code>
<code>UMAALGE</code>	<code>r2, r0, r5, r3</code>

4.4.11 MIA、MIAPH、MIAxy

XScale コプロセッサ 0 命令です。

内部積和命令です (32 ビット × 32 ビットを実行し、40 ビットの累算値を返します)。

パックハーフワードの内部積和命令です (16 ビット × 16 ビット × 2 を実行し、40 ビットの累算値を返します)。

内部積和命令です (16 ビット × 16 ビットを実行し、40 ビットの累算値を返します)。

構文

MIA{cond} Acc, Rm, Rs

MIAPH{cond} Acc, Rm, Rs

MIA<x><y>{cond} Acc, Rm, Rs

パラメータの説明 :

cond 任意に指定できる条件コードです (「条件実行」(P. 2-17) 参照)。

Acc 内部アキュムレータを指定します。標準名は accx で、*x* は 0 ~ n の整数です。*n* の値はプロセッサによって異なります。最近のプロセッサでは 0 が使用されています。

Rm, Rs 乗算する値を保持する ARM レジスタを指定します。

Rm または *Rs* に r15 を使用しないで下さい。

<x> B または T を指定します。Rm の下位ビット (ビット [15:0]) を使用する場合は B を、Rm の上位ビット (ビット [31:16]) を使用する場合は T を指定します。

<y> B または T を指定します。Rs の下位ビット (ビット [15:0]) を使用する場合は B を、Rs の上位ビット (ビット [31:16]) を使用する場合は T を指定します。

使用法

MIA 命令は *Rs* と *Rm* の符号付き整数を乗算し、演算結果を *Acc* の 40 ビット値に加算します。

MIAPH 命令は、*Rs* と *Rm* の下位半分の符号付き整数の乗算、および *Rs* と *Rm* の上位半分の符号付き整数の乗算を行い、32 ビットの 2 つの結果を *Acc* の 40 ビット値に加算します。

MIAxy 命令は、*Rs* の指定された上位半分または下位半分の符号付き整数と、*Rm* の指定された上位半分または下位半分の符号付き整数を乗算し、32 ビットの結果を *Acc* の 40 ビット値に加算します。

条件フラグ

これらの命令によるフラグへの影響はありません。

——注——

これらの命令によって例外が発生することはありません。これらの命令でオーバフローが発生すると、警告がないまま演算結果が溢れた桁によって上書きされます。

アーキテクチャ

これらの ARM 命令は、XScale プロセッサでのみ使用できます。

これらの命令の Thumb バージョンおよび Thumb-2 バージョンはありません。

例

```
MIA      acc0,r5,r0
MIALE    acc0,r1,r9
MIAPH    acc0,r0,r7
MIAPHNE  acc0,r11,r10
MIABB   acc0,r8,r9
MIABT   acc0,r8,r8
MIATB   acc0,r5,r3
MIATT   acc0,r0,r6
MIABTGT acc0,r2,r5
```

4.5 サチュレート命令

このセクションは以下のサブセクションから構成されています。

- サチュレート命令について
- *QADD*, *QSUB*, *QDADD*, *QDSUB* (P. 4-100)
- *SSAT*, *USAT* (P. 4-102)

一部の並列命令もサチュレート命令です。詳細については、「*並列命令*」(P. 4-104) を参照して下さい。

4.5.1 サチュレート命令について

以下の演算がサチュレート (SAT) 演算です。つまり、サチュレート命令に応じて、 2^n の値は以下のようになります。

- 符号付きサチュレート演算では、結果が -2^n 未満の場合には、返される結果は -2^n になります。
- 符号なしサチュレート演算では、結果が負となる場合には、返される結果はゼロとなります。
- 結果が $2^n - 1$ よりも大きい場合には、返される結果は $2^n - 1$ となります。

上記のいずれかに当たる演算をサチュレーションと呼びます。一部の命令では、サチュレーションが発生すると Q フラグを設定します。

注

サチュレーションが発生しない場合に、サチュレート命令によって Q フラグがクリアされることはありません。Q フラグをクリアするには、MSR 命令を使用します (*/MSR* (P. 4-141) 参照)。

Q フラグは、他の 2 つの命令によってもセットできますが (*/SMULxy*, *SMLAxz* (P. 4-83) および */SMULWy*, *SMLAWy* (P. 4-85) 参照)、それらの命令ではサチュレートは発生しません。

4.5.2 QADD、QSUB、QDADD、QDSUB

$-2^{31} \leq x \leq 2^{31}-1$ の符号付き範囲で演算結果をサチュレートさせる、符号付き加算、減算、倍演算と加算、および倍演算と減算です。

「並列加算と並列減算」(P. 4-105) も参照して下さい。

構文

op{cond} Rd, Rm, Rn

パラメータの説明：

op QADD、QSUB、QDADD、QDSUB のいずれかを指定します。

cond 任意の条件コードを指定します（「条件実行」(P. 2-17) 参照）。

Rd デスティネーションレジスタを指定します。

Rm, Rn オペランドを保持するレジスタを指定します。

Rd、*Rm*、または*Rn* に r15 を使用しないで下さい。

使用法

QADD 命令は、*Rm* と *Rn* の値を加算します。

QSUB 命令は、*Rm* の値から *Rn* の値を減算します。

QDADD 命令は、 $SAT(Rm + SAT(Rn * 2))$ を計算します。倍演算または加算、あるいはその両方で、サチュレーションが発生する可能性があります。倍演算でサチュレーションが発生し、かつ加算では発生しない場合、Q フラグは設定されますが、最終結果はサチュレートしません。

QDSUB 命令は、 $SAT(Rm - SAT(Rn * 2))$ を計算します。倍演算または減算、あるいはその両方で、サチュレーションが発生する可能性があります。倍演算でサチュレーションが発生し、かつ減算では発生しない場合、Q フラグは設定されますが、最終結果はサチュレートしません。

――注――

これらの命令では、すべての値が 2 の補数の符号付き整数として処理されます。

同様の、ARMv6 以上でのみ使用可能な並列命令については、「並列加算と並列減算」(P. 4-105) を参照して下さい。

条件フラグ

サチュレーションが発生すると、これらの命令によって Q フラグが設定されます。Q フラグの状態を読み出すには、MRS 命令を使用します（*/MRS/* (P. 4-140) 参照）。

他のフラグへの影響はありません。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上、および ARMv5 の E バリアントで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

```
QADD    r0, r1, r9  
QDSUBLT r9, r0, r1
```

誤用例

```
QSUBS   r3, r4, r2 ; use of S suffix not permitted  
QDADD   r11, r15, r0 ; use of r15 not permitted
```

4.5.3 SSAT、USAT

任意のビット位置に対する符号付きサチュレート命令と符号なしサチュレート命令です。オプションとして、サチュレート前にシフトを実行できます。

/SSAT16、USAT16/ (P. 4-110) も参照して下さい。

構文

op{cond} Rd, #sat, Rm{, shift}

パラメータの説明：

<i>op</i>	SSAT または USAT を指定します。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。
<i>Rd</i>	デスティネーションレジスタを指定します。 <i>Rd</i> に r15 は指定できません。
<i>sat</i>	0 ~ 31 の範囲内で、サチュレートさせるビット位置を指定します。
<i>Rm</i>	オペランドを保持するレジスタを指定します。 <i>Rm</i> に r15 は指定できません。
<i>shift</i>	任意に指定できるシフトです。以下のいずれかを指定する必要があります。 ASR # <i>n</i> <i>n</i> には、1 ~ 32 の範囲内にある値を指定します。 LSL # <i>n</i> <i>n</i> には、0 ~ 31 の範囲内にある値を指定します。

演算

SSAT 命令は指定されたシフトを行い、その後に符号付き範囲 $-2^{\text{sat}-1} \leq x \leq 2^{\text{sat}-1} - 1$ にサチュレートさせます。

USAT 命令は指定されたシフトを行い、その後に符号なし範囲 $0 \leq x \leq 2^{\text{sat}} - 1$ にサチュレートさせます。

条件フラグ

サチュレーションが発生すると、これらの命令によって Q フラグが設定されます。Q フラグの状態を読み出すには、MRS 命令を使用します (*/MRS/* (P. 4-140) 参照)。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上、および ARMv5 の E バリアントで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

```
SSAT      r7, #16, r7, LSL #4  
USATNE   r0, #7, r5
```

誤用例

```
USATGT   r0, #7, r15, ASR #16           ; use of r15 not permitted
```

4.6 並列命令

このセクションは以下のサブセクションから構成されています。

- **並列加算と並列減算** (P. 4-105)
バイト単位およびハーフワード単位のさまざまな加算と減算です。
 - **USAD8、USADA8** (P. 4-108)
符号なし絶対差の和と符号なし絶対差の和の累算です。
 - **SSATI6、USATI6** (P. 4-110)
並列ハーフワードサチュレート命令です。
- この他に、並列展開命令もあります。詳細については、*/SXT、SXTA、UXT、UXTA/* (P. 4-115) を参照して下さい。

4.6.1 並列加算と並列減算

バイト単位およびハーフワード単位のさまざまな加算と減算です。

構文

<prefix>op{cond} Rd, Rn, Rm

パラメータの説明：

<i><prefix></i>	次のいずれかを指定します。
S	符号付き余り算術演算 2^8 または 2^{16} です。CPSR の GE フラグが設定されます。
Q	符号付きサチュレート算術演算です。
SH	符号付き算術演算を行い、結果を半分にします。
U	符号なし余り算術演算 2^8 または 2^{16} です。CPSR の GE フラグが設定されます。
UQ	符号なしサチュレート算術演算です。
UH	符号なし算術演算を行い、結果を半分にします。
<i>op</i>	次のいずれかを指定します。
ADD8	バイト単位の加算を行います。
ADD16	ハーフワード単位の加算を行います。
SUB8	バイト単位の減算を行います。
SUB16	ハーフワード単位の減算を行います。
ASX	Rm のハーフワードを交換し、上位ハーフワードの加算と下位ハーフワードの減算を行います。
SAX	Rm のハーフワードを交換し、上位ハーフワードの減算と下位ハーフワードの加算を行います。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」（P. 2-17）参照）。
<i>Rd</i>	デスティネーションレジスタを指定します。Rd に r15 を使用しないで下さい。
<i>Rm, Rn</i>	オペランドを保持する ARM レジスタを指定します。Rm または Rn に r15 を使用しないで下さい。

演算

これらの命令は、オペランドのバイトまたはハーフワードに対し、個別に算術演算を実行します。2つか4つの加算または減算、あるいは1つの加算と1つの減算を実行します。

以下のようなさまざまな種類の演算を選択できます。

- 2^8 または 2^{16} の符号付き / 符号なし余り算術演算。これにより CPSR の GE フラグが設定されます（「条件フラグ」参照）。
- 符号付き範囲 $-2^{15} \leq x \leq 2^{15}-1$ または $-2^7 \leq x \leq 2^7-1$ のいずれかに対する符号付きサチュレート算術演算。これらの演算がサチュレートする場合でも Q フラグへの影響はありません。
- 符号なし範囲 $0 \leq x \leq 2^{16}-1$ または $0 \leq x \leq 2^8-1$ のいずれかに対する符号なしサチュレート算術演算。これらの演算がサチュレートする場合でも Q フラグへの影響はありません。
- 符号付き / 符号なし算術演算。結果は二分されます。これによってオーバフローが発生することはありません。

条件フラグ

これらの命令による N、Z、C、V、または Q フラグへの影響はありません。

これらの命令に接頭文字 Q、SH、UQ、および UH を使用した場合でも、フラグへの影響はありません。

これらの命令に接頭文字 S および U を使用した場合は、CPSR の GE フラグが以下のように設定されます。

- バイト単位の演算では、GE フラグは 32 ビットの SUB 命令および ADD 命令における C (キャリー) フラグと同様の方法で使用されます。

GE[0]	結果のビット [7:0] に対応します。
GE[1]	結果のビット [15:8] に対応します。
GE[2]	結果のビット [23:16] に対応します。
GE[3]	結果のビット [31:24] に対応します。
- ハーフワード単位の演算では、GE フラグは通常のワード単位の SUB 命令および ADD 命令における C (キャリー) フラグと同様の方法で使用されます。

GE[1:0]	結果のビット [15:0] に対応します。
GE[3:2]	結果のビット [31:16] に対応します。

これらのフラグを使用して、次に続く SEL 命令を制御できます（「SEL」（P. 4-68）参照）。

注

ハーフワード単位の演算では、GE[1:0] が同時にセットまたはクリアされ、GE[3:2] が同時にセットまたはクリアされます。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

SHADD8	r4, r3, r9
USAXNE	r0, r0, r2

誤用例

QHADD	r2, r9, r3	; No such instruction, should be QHADD8 or QHADD16
UQSUB16NE	r1, r15, r0	; Use of r15 not permitted
SAX	r10, r8, r5	; Must have a prefix.

4.6.2 USAD8、USADA8

符号なし絶対差の和と符号なし絶対差の和の累算です。

構文

USAD8{cond} Rd, Rm, Rs

USADA8{cond} Rd, Rm, Rs, Rn

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

Rd デスティネーションレジスタを指定します。

Rm 第1オペランドを保持するレジスタを指定します。

Rs 第2オペランドを保持するレジスタを指定します。

Rn 累算オペランドを保持するレジスタを指定します。

Rd、*Rm*、*Rs*、または*Rn*にr15を使用しないで下さい。

演算

USAD8 命令は、*Rm* と *Operand2* のそれぞれに対応するバイトの符号なし値に基づいて、4つの差分をとります。この命令は、4つの差分の絶対値を加算し、その結果を *Rd* にストアします。

USADA8 命令は、4つの差分の絶対値を *Rn* の値に加算し、その結果を *Rd* に保存します。

条件フラグ

これらの命令によるフラグへの影響はありません。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

USAD8	r2, r4, r6
USADA8	r0, r3, r5, r2
USADA8VS	r0, r4, r0, r1

誤用例

USADA8	r2, r4, r6	; USADA8 requires four registers
USAD8CC	r0, r3, r15	; use of r15 not permitted
USADA16	r0, r4, r0, r1	; no such instruction

4.6.3 SSAT16、USAT16

並列ハーフワードサチュレート命令です。

構文

op{cond} Rd, #sat, Rm

パラメータの説明：

<i>op</i>	次のいずれかを指定します。
SSAT16	符号付きサチュレーションを実行します。
USAT16	符号なしサチュレーションを実行します。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」（P. 2-17）参照）。
<i>Rd</i>	デスティネーションレジスタを指定します。
<i>sat</i>	サチュレートさせるビット位置を指定します。SSAT16 の場合は 1 ~ 16、USAT16 の場合は 0 ~ 15 の値となります。
<i>Rm</i>	オペランドを保持するレジスタを指定します。

Rd または *Rm* に r15 を使用しないで下さい。

演算

任意のビット位置に対してハーフワード単位の符号付き/符号なしサチュレーションを実行します。

SSAT16 命令は、符号付き範囲 $-2^{\text{sat}-1} \leq x \leq 2^{\text{sat}-1} - 1$ で、各ハーフワードをサチュレートします。

USAT16 命令は、符号なし範囲 $0 \leq x \leq 2^{\text{sat}} - 1$ で、各ハーフワードをサチュレートします。

条件フラグ

いずれかのハーフワードに対するサチュレーションが発生すると、これらの命令によって Q フラグが設定されます。Q フラグの状態を読み出すには、MRS 命令を使用します（/MRS/（P. 4-140）参照）。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

```
SSAT16 r7, #12, r7  
USAT16 r0, #7, r5
```

誤用例

```
SSAT16 r1, #16, r2, LSL #4 ; shifts not permitted with halfword saturations  
USAT16 r0, #11, r15          ; use of r15 not permitted
```

4.7 パック命令と展開命令

このセクションは以下のサブセクションから構成されています。

- *BFC, BFI* (P. 4-113)
ビットフィールドのクリア命令と挿入命令です。
- *SBFX, UBFX* (P. 4-114)
符号付き / 符号なしビットフィールドの抽出命令です。
- *SXT, SXTA, UXT, UXTA* (P. 4-115)
必要に応じて加算命令を伴う、符号拡張命令またはゼロ拡張命令です。
- *PKHBT, PKHTB* (P. 4-118)
ハーフワードパック命令です。

4.7.1 BFC、BFI

ビットフィールドのクリア命令と挿入命令です。レジスタ内の隣接するビットをクリアするか、またはあるレジスタから別のレジスタに隣接するビットを挿入します。

構文

```
BFC{cond} Rd, #1sb, #width
BFI{cond} Rd, Rn, #1sb, #width
```

パラメータの説明：

<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。
<i>Rd</i>	デスティネーションレジスタを指定します。 <i>Rd</i> に r15 は指定できません。
<i>Rn</i>	ソースレジスタを指定します。 <i>Rn</i> に r15 は指定できません。
<i>1sb</i>	クリアまたはコピーする最下位ビットを指定します。
<i>width</i>	クリアまたはコピーするビット数を指定します。 <i>width</i> に 0 は指定できません。また、(<i>width</i> +1sb) は 32 未満になる必要があります。

BFC

Rd の 1sb から *width* ビット分のビットがクリアされます。*Rd* 内の他のビットは変更されません。

BFI

Rd の 1sb から *width* ビット分のビットが、*Rn* のビット [0] から *width* ビット分のビットで置き換えられます。*Rd* 内の他のビットは変更されません。

条件フラグ

これらの命令によるフラグへの影響はありません。

アーキテクチャ

これらの ARM 命令は、ARMv6T2 以上で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

4.7.2 SBFX、UBFX

符号付き / 符号なしビットフィールドの抽出命令です。隣接するビットをあるレジスタから別のレジスタの最下位ビットにコピーし、32 ビットに符号拡張またはゼロ拡張します。

構文

op{cond} Rd, Rm, #lsb, #width

パラメータの説明 :

op SBFX または UBFX を指定します。

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

Rd デスティネーションレジスタを指定します。

Rm ソースレジスタを指定します。

lsb ビットフィールドの最下位ビットのビット番号を 0 ~ 31 の範囲内で指定します。

width ビットフィールドの幅を 1 ~ (1+*lsb*) の範囲内で指定します。

Rd または *Rm* に r15 を使用しないで下さい。

条件フラグ

これらの命令によるフラグへの影響はありません。

アーキテクチャ

これらの ARM 命令は、ARMv6T2 以上で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

4.7.3 SXT、SXTA、UXT、UXTA

必要に応じて加算命令を伴う、符号拡張命令またはゼロ拡張命令です。

これらの命令は以下のいずれかの処理を行います。

- 8 ビット値を 32 ビットに符号拡張またはゼロ拡張し、必要に応じて 32 ビット値を加算します。
- 16 ビット値を 32 ビットに符号拡張またはゼロ拡張し、必要に応じて 32 ビット値を加算します。
- 2 つの 8 ビット値を 2 つの 16 ビット値に符号拡張またはゼロ拡張し、必要に応じて 2 つの 16 ビット値を加算します。

構文

op<extend>{cond} Rd, {Rn,} Rm[, rotation]

パラメータの説明：

<i>op</i>	次のいずれかを指定します。
SXT	符号拡張します。
UXT	ゼロ拡張します。
SXTA	符号拡張と加算を行います。
UXTA	ゼロ拡張と加算を行います。
<i><extend></i>	次のいずれかを指定します。
B16	2 つの 8 ビット値を 2 つの 16 ビット値に拡張します。
B	8 ビット値を 32 ビット値に拡張します。
H	16 ビット値を 32 ビット値に拡張します。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。
<i>Rd</i>	デスティネーションレジスタを指定します。
<i>Rn</i>	加算する数を保持するレジスタを指定します(SXTA と UXTA の場合のみ)。
<i>Rm</i>	拡張する値を保持するレジスタを指定します。
<i>rotation</i>	次のいずれかを指定します。
ROR #8	<i>Rm</i> の値が右に 8 ビットロテートされます。
ROR #16	<i>Rm</i> の値が右に 16 ビットロテートされます。
ROR #24	<i>Rm</i> の値が右に 24 ビットロテートされます。
<i>rotation</i>	<i>rotation</i> を省略した場合、ロテートは実行されません。

Rd、*Rn*、または *Rm* に r15 を使用しないで下さい。

演算

これらの命令は以下の処理を行います。

1. Rm の値を右に 0 ビット、8 ビット、16 ビット、または 24 ビットロテートします (ARM および Thumb-2 のみ)。
2. 取得した値に対して以下のいずれかの処理を行います。
 - ビット [7:0] を抽出し、32 ビットに符号拡張またはゼロ拡張します。命令で拡張と加算を行う場合は、 Rn の値を加算します。
 - ビット [15:0] を抽出し、32 ビットに符号拡張またはゼロ拡張します。命令で拡張と加算を行う場合は、 Rn の値を加算します。
 - ビット [23:16] とビット [7:0] を抽出し、これらのビットを 16 ビットに符号拡張またはゼロ拡張します。命令で拡張と加算を行う場合は、これらのビットをそれぞれ Rn のビット [31:16] とビット [15:0] に加算し、結果のビット [31:16] とビット [15:0] を形成します。

条件フラグ

これらの命令によるフラグへの影響はありません。

16 ビット命令

これらの命令は、Thumb コード内では次の形式で使用できます。また、Thumb-2 コード内で使用するときは 16 ビット命令になります。

SXTB Rd, Rm Rd および Rm は共に Lo レジスタである必要があります。

SXTH Rd, Rm Rd および Rm は共に Lo レジスタである必要があります。

UXTB Rd, Rm Rd および Rm は共に Lo レジスタである必要があります。

UXTH Rd, Rm Rd および Rm は共に Lo レジスタである必要があります。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの 16 ビット Thumb 命令は、ARMv6 以上で使用できます。

例

```
SXTH      r3, r9, r4  
UXTAB16EQ r0, r0, r4, ROR #16
```

誤用例

```
UXTAB    r0, r2, r15      ; use of r15 not permitted  
SXTH     r9, r3, r2, ROR #12 ; rotation must be by 0, 8, 16, or 24.
```

4.7.4 PKHBT、PKHTB

ハーフワードパック命令です。

あるレジスタに保持されているハーフワードと別のレジスタに保持されているハーフワードを結合します。オペランドの 1 つは、ハーフワードを抽出する前にシフトできます。

構文

op{cond} Rd, Rn, Rm{, shift}

パラメータの説明：

<i>op</i>	次のいずれかを指定します。
PKHBT	<i>Rn</i> のビット [15:0] とシフトされた <i>Rm</i> の値のビット [31:16] を結合します。
PKHTB	<i>Rn</i> のビット [31:16] とシフトされた <i>Rm</i> の値のビット [15:0] を結合します。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。
<i>Rd</i>	デスティネーションレジスタを指定します。
<i>Rn</i>	第 1 オペランドを保持するレジスタを指定します。
<i>Rm</i>	第 2 オペランドを保持するレジスタを指定します。
<i>shift</i>	次のいずれかを指定します。
LSL # <i>n</i>	論理左シフト。 <i>n</i> には、0 ~ 31 の範囲内にある値を指定します。PKHBT でのみ使用できます。
ASR # <i>n</i>	算術右シフト。 <i>n</i> には、1 ~ 32 の範囲内にある値を指定します。PKHTB でのみ使用できます。

Rd、*Rn*、または *Rm* に r15 を使用しないで下さい。

条件フラグ

これらの命令によるフラグへの影響はありません。

アーキテクチャ

これらの ARM 命令は、ARMv6 以上で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

例

```
PKHBT r0, r3, r5      ; combine the bottom halfword of r3 with the top halfword of r5  
PKHBT r0, r3, r5, LSL #16 ; combine the bottom halfword of r3 with the bottom halfword of r5  
PKHTB r0, r3, r5, ASR #16 ; combine the top halfword of r3 with the top halfword of r5
```

また、異なるシフト値を使用して、第 2 オペランドを位取りすることもできます。

誤用例

```
PKHBT r4, r15, r1      ; use of r15 not permitted  
PKHBTEQ r4, r5, r1, ASR #8 ; ASR not permitted with PKHBT
```

4.8 分岐命令

このセクションは以下のサブセクションから構成されています。

- *B, BL, BX, BLX, BXJ* (P. 4-121)
分岐、リンク付き分岐、分岐と命令セットの切り替え、リンク付き分岐と命令セットの切り替え、および分岐と命令セットの Jazelle への変更を行う命令です。
- *CBZ, CBNZ* (P. 4-124)
ゼロとの比較と分岐を行う命令です。
- *TBB, TBH* (P. 4-125)
テーブル分岐バイトとテーブル分岐ハーフワードです。

4.8.1 B、BL、BX、BLX、BXJ

分岐、リンク付き分岐、分岐と命令セットの切り替え、リンク付き分岐と命令セットの切り替え、および分岐と Jazelle 状態への変更を行う命令です。

構文

op{cond}{.W} label

op{cond} Rm

パラメータの説明：

<i>op</i>	次のいずれかを指定します。
B	分岐命令です。
BL	リンク付き分岐命令です。
BX	分岐と命令セットの切り替えを行う命令です。
BLX	リンク付き分岐と命令セットの切り替えを行う命令です。
BXJ	分岐と Jazelle 実行状態への変更を行う命令です。
<i>cond</i>	任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。 <i>cond</i> は、これらの命令のすべての形式で使用できるわけではありません（「命令の使用可否と分岐の範囲」(P. 4-122) 参照）。
.W	Thumb-2 の 32 ビット B 命令を使用させるために命令の幅の指定子を指定します（省略可）。詳細については、「Thumb-2 の B」(P. 4-123) を参照して下さい。
<i>label</i>	プログラム相対式を指定します。詳細については、「レジスタ相対式とプログラム相対式」(P. 3-31) を参照して下さい。
<i>Rm</i>	分岐先アドレスを保持するレジスタを指定します。

演算

これらのすべての命令は、*label* への分岐または *Rm* に保持されているアドレスへの分岐を発生させます。さらに、以下の処理を行います。

- BL 命令と BLX 命令は、次の命令のアドレスを r14 (lr : リンクレジスタ) にコピーします。
- BX 命令と BLX 命令は、プロセッサ状態を ARM から Thumb に変更したり、Thumb から ARM に変更したりできます。

BLX *label* を使用すると、状態が必ず変更されます。

BX *Rm* と BLX *Rm* を使用すると、*Rm* のビット [0] からターゲットの状態を得ることができます。

— *Rm* のビット [0] が 0 の場合、プロセッサは ARM 状態に切り替わるか、ARM 状態が維持されます。

- Rm のビット [0] が 1 の場合、プロセッサは Thumb 状態に切り替わるか、Thumb 状態が維持されます。
- BXJ 命令はプロセッサの状態を Jazelle に変更します。

命令の使用可否と分岐の範囲

表 4-2 は、ARM 状態と Thumb 状態で使用できる命令を示しています。この表に記載されていない命令は使用できません。括弧書きの記載は、命令を使用できるアーキテクチャの最初のバージョンを示しています。

表 4-2 分岐命令の使用可否と分岐の範囲

命令	ARM	16 ビットの Thumb	32 ビットの Thumb-2
B label	$\pm 32MB$	(すべて)	$\pm 2KB$ (すべての T) $\pm 16MB^a$
B{cond} label	$\pm 32MB$	(すべて)	$-252 \sim +258$ (すべての T) $\pm 1MB^a$
B Rm	BX Rm を使用	BX Rm を使用	16 ビットの BX Rm を使用
B{cond} Rm	BX{cond} Rm を使用	-	-
BL label	$\pm 32MB$	(すべて)	$\pm 4MB^b$ (すべての T) $\pm 16MB$
BL{cond} label	$\pm 32MB$	(すべて)	-
BL Rm	BLX Rm を使用	BLX Rm を使用	16 ビットの BLX Rm を使用
BL{cond} Rm	BLX{cond} Rm を使用	-	-
BX Rm	使用可能 (4T、5)	使用可能 (すべての T)	16 ビットを使用
BX{cond} Rm	使用可能 (4T、5)	-	-
BLX label	$\pm 32MB$	(5)	$\pm 4MB^c$ (5T) $\pm 16MB$
BLX Rm	使用可能 (5)	使用可能 (5T)	16 ビットを使用
BLX{cond} Rm	使用可能 (5)	-	-
BXJ Rm	使用可能 (5J、6)	-	使用可能
BXJ{cond} Rm	使用可能 (5J、6)	-	-

a. この 32 ビット命令を使用するようアセンブラーに指定する場合は .W を使用します。

b. これは命令対です。

c. これは命令対です。

分岐の範囲の拡張

マシンレベルの B 命令と BL 命令では、現在の命令のアドレスからの範囲が制限されています。ただし、*label* が範囲外の場合でもこれらの命令を使用できます。ほとんどの場合、リンクが *label* を配置する場所は分かりません。必要な場合には、リンクはコードを追加してより長い分岐を可能にします (*RealView Compilation Tools v3.0 Linker and Utilities Guide* の基本的なリンクの機能に関する章を参照して下さい)。追加されたコードは *veener* (ベニア) と呼ばれます。

Thumb-2 の B

.W 幅指定子を使用して、B で Thumb-2 コードの 32 ビット命令を生成することができます。

B.W は、16 ビット命令を使用してターゲットに到達できる場合でも、常に 32 ビット命令を生成します。

前方参照の場合、Thumb コードで .W を指定せずに B を実行すると常に 16 ビット命令が生成されますが、生成された 16 ビット命令では 32 ビット Thumb-2 命令を使用して到達できるターゲットに到達できない場合があります。

Thumb-2EE の BX、BLX、BXJ

これらの命令は、Thumb-2EE コード内で分岐として使用できますが、状態の変更には使用できません。つまり、Thumb-2EE では、これらの命令を *op{cond} label* という形式で使用することができない、そして *Rm* のビット [0] には 1 を指定する必要があります。*Rm* のビット [0] が 0 の場合、命令は予測不可能であり、信頼できません。

条件フラグ

これらの命令によるフラグへの影響はありません。

アーキテクチャ

各アーキテクチャでのこれらの命令の使用可否の詳細については、「[命令の使用可否と分岐の範囲](#)」(P. 4-122) を参照して下さい。

例

B	loopA
BLE	ng+8
BL	subC
BLLT	rTX
BEQ	{pc}+4 ; #0x8004

4.8.2 CBZ、CBNZ

ゼロとの比較と分岐を行う命令です（ゼロまたはゼロでない場合に分岐します）。

——注——

これらはThumb-2 16 ビット命令です。ARM 状態では使用できません。

構文

CBZ Rn, Label

CBNZ Rn, Label

パラメータの説明：

Rn オペランドを保持するレジスタを指定します。

Label 分岐先を指定します。

使用法

CBZ 命令または CBNZ 命令を使用すると、ゼロとの比較で 1 つの命令を保存し、コードシーケンスを分岐できます。

条件コードフラグが変更されない点を除き、**CBZ Rn, Label** は以下のコマンドと同じ意味です。

CMP	<i>Rn, #0</i>
BEQ	<i>Label</i>

条件コードフラグが変更されない点を除き、**CBNZ Rn, Label** は以下のコマンドと同じ意味です。

CMP	<i>Rn, #0</i>
BNE	<i>Label</i>

制約条件

分岐先は、命令の後に 4 ~ 130 バイト以内に指定する必要があります。

これらの命令は IT ブロック内では使用できません。

条件フラグ

これらの命令によるフラグへの影響はありません。

アーキテクチャ

これらの 16 ビット Thumb-2 命令は、ARMv6 以上の T2 バリエントで使用できます。

4.8.3 TBB、TBH

テーブル分岐バイトとテーブル分岐ハーフワードです。

構文

TBB [*Rn*, *Rm*]

TBH [*Rn*, *Rm*, LSL #1]

パラメータの説明：

Rn ベースレジスタを指定します。このレジスタでは、分岐の長さのテーブルのアドレスが保持されます。

Rm インデクスレジスタを指定します。このレジスタでは、テーブル内のシングルバイトを指し示す整数が保持されます。テーブル内のオフセットは、使用する命令によって異なります。

TBB インデックスの値

TBH インデックスの値の 2 倍値

演算

これらの命令により、シングルバイトオフセット (TBB) またはハーフワードオフセット (TBH) のテーブルを使用した PC 相対の順方向の分岐が発生します。*Rn* はテーブルへのポインタを提供し、*Rm* はテーブルのインデクスを提供します。分岐の長さは、テーブルから返されたバイト (TBB) またはハーフワード (TBH) の値の 2 倍になります。

注

Rn に r15 を指定している場合、使用される値は「命令のアドレス + 4」となります。

Rm に r15 は指定できません。

Thumb-2EE では、ベースレジスタの値が 0 の場合、HandlerBase - 4 にある NullCheck ハンドラへの分岐が実行されます。

例外

データアポートが発生します。

アーキテクチャ

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の ARM バージョンおよび 16 ビット Thumb バージョンはありません。

4.9 コプロセッサ命令

このセクションでは、ベクタ浮動小数点（「第5章 NEONとVFPプログラミング」参照）およびワイヤレス MMX™ テクノロジの命令（「第6章 ワイヤレス MMX テクノロジの命令」参照）について説明していません。XScale 固有の命令についてはこの章の後半で説明しています（「その他の命令」（P. 4-136）参照）。

このセクションは以下のサブセクションから構成されています。

- *CDP, CDP2* (P. 4-127)
コプロセッサデータ演算です。
- *MCR, MCR2, MCRR, MCRR2* (P. 4-128)
コプロセッサ命令を使用した ARM レジスタからコプロセッサへの移動命令です。
- *MRC, MRC2, MRRC, MRRC2* (P. 4-130)
コプロセッサ命令を使用したコプロセッサから ARM レジスタへの移動命令です。
- *LDC, STC* (P. 4-132)
メモリとコプロセッサ間のデータ転送命令です。
- *LDC2, STC2* (P. 4-134)
メモリとコプロセッサ間のデータ転送命令（代替の命令）です。

4.9.1 CDP、CDP2

コプロセッサデータ演算です。

構文

op {cond} coproc, opcode1, CRd, CRn, CRm{, opcode2}

パラメータの説明：

op CDP または CDP2 を指定します。

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

coproc 命令が実行されるコプロセッサの名前を指定します。標準名は *pn* で、*n* は 0 ~ 15 の整数です。

opcode1 コプロセッサ固有のオペコードを指定します。

CRd, CRn, CRm コプロセッサレジスタを指定します。

opcode2 オプションとしてのコプロセッサ固有のオペコードを指定します。

使用法

これらの命令の使用方法はコプロセッサによって異なります。詳細については、コプロセッサのマニュアルを参照して下さい。

注

CDP2 は、ARM 状態では、常に条件なしで実行されます。Thumb-2 命令は IT ブロック内で条件付きにすることができます。

例外

未定義命令が発生します。

アーキテクチャ

ARM 命令 CDP は、ARM アーキテクチャのすべてのバージョンで使用できます。

ARM 命令 CDP2 は、ARMv5 以上で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

4.9.2 MCR、MCR2、MCRR、MCRR2

ARM レジスタからコプロセッサへの移動命令です。コプロセッサによっては、さまざまな演算を追加で指定できる場合があります。

構文

op {cond} coproc, opcode1, Rd, CRn, CRm{, opcode2}

op {cond} coproc, opcode1, Rd, Rn, CRm

パラメータの説明：

op MCR、MCR2、MCRR、または MCRR2 を指定します。

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

coproc 命令が実行されるコプロセッサの名前を指定します。標準名は *pn* で、*n* は 0 ~ 15 の整数です。

opcode1 コプロセッサ固有のオペコードを指定します。

Rd, Rn ARM ソースレジスタを指定します。*Rn* は、MCRR および MCRR2 でのみ使用できます。MCRR または MCRR2 では、*Rd* または *Rn* に r15 を使用しないで下さい。

CRn, CRm コプロセッサレジスタを指定します。*CRn* は、MCR および MCR2 でのみ使用できます。

opcode2 MCR および MCR2 で使用できるオプションとしてのコプロセッサ固有のオペコードを指定します。

使用法

これらの命令の使用方法はコプロセッサによって異なります。詳細については、コプロセッサのマニュアルを参照して下さい。

―― 注――

MCR2 および MCRR2 は、ARM 状態では、常に条件なしで実行されます。Thumb-2 命令は IT ブロック内で条件付きにすることができます。

例外

未定義命令が発生します。

アーキテクチャ

ARM 命令 MCR は、ARM アーキテクチャのすべてのバージョンで使用できます。

ARM 命令 MCR2 は、ARMv5 以上で使用できます。

ARM 命令 MCRR は、ARMv6 以上、および xP バリアントを除く ARMv5 の E バリアントで使用できます。

ARM 命令 MCRR2 は、ARMv6 以上で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

4.9.3 MRC、MRC2、MRRC、MRRC2

コプロセッサから ARM レジスタへの移動命令です。

コプロセッサによっては、さまざまな演算を追加で指定できる場合があります。

構文

op {cond} coproc, opcode1, Rd, CRn, CRm{, opcode2}

op {cond} coproc, opcode1, Rd, Rn, CRm

パラメータの説明：

op MRC、MRC2、MRRC、または MRRC2 を指定します。

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

coproc 命令が実行されるコプロセッサの名前を指定します。標準名は *pn* で、*n* は 0 ~ 15 の整数です。

opcode1 コプロセッサ固有のオペコードを指定します。

Rd, Rn ARM ソースレジスタを指定します。*Rn* は、MRRC および MRRC2 でのみ使用できます。MRRC または MRRC2 では、*Rd* または *Rn* に r15 を使用しないで下さい。

MRC および MRC2 では、*Rd* に r15 を使用している場合、フラグフィールドにのみ影響があります。

CRn, CRm コプロセッサレジスタを指定します。*CRn* は、MRC および MRC2 でのみ使用できます。

opcode2 MCR および MCR2 で使用できるコプロセッサ固有のオペコードを指定します（省略可）。

使用法

これらの命令の使用方法はコプロセッサによって異なります。詳細については、コプロセッサのマニュアルを参照して下さい。

————注————

MRC2 および MRRC2 は、ARM 状態では、常に条件なしで実行されます。Thumb-2 命令は IT ブロック内で条件付きにすることができます。

例外

未定義命令が発生します。

アーキテクチャ

ARM 命令 MRC は、ARM アーキテクチャのすべてのバージョンで使用できます。

ARM 命令 MRC2 は、ARMv5 以上で使用できます。

ARM 命令 MRRC は、ARMv6 以上、および xP バリアントを除く ARMv5 の E バリアントで使用できます。

ARM 命令 MRRC2 は、ARMv6 以上で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

4.9.4 LDC、STC

メモリとコプロセッサ間のデータ転送命令です。

構文

これらの命令には、以下の 3 つの形式を使用できます。

- ゼロオフセット
- プレインデクスオフセット
- ポストインデクスオフセット

この 3 つの形式の構文を、上記と同じ順番で以下に示します。

op{L}{cond} coproc, CRd, [Rn]

op{L}{cond} coproc, CRd, [Rn, #{-}offset]{!}

op{L}{cond} coproc, CRd, [Rn], #{-}offset

パラメータの説明：

op LDC または STC を指定します。

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

L 長い転送を指定するオプションの接尾文字を指定します。

coproc 命令が実行されるコプロセッサの名前を指定します。標準名は *p_n* で、*n* は 0 ~ 15 の整数です。

CRd ロードまたはストアを実行するコプロセッサレジスタを指定します。

Rn メモリアドレスのベースとなるレジスタを指定します。*r15* が指定されている場合、使用される値は「現在の命令のアドレス + 8」となります。

- 任意に指定できるマイナス符号です。- が指定されている場合、オフセットが *Rn* から減算されます。指定されていない場合は、オフセットが *Rn* に加算されます。

offset 0 ~ 1020 の範囲で 4 の倍数を求める式を指定します。

! 任意に指定できる接尾文字です。! を指定すると、オフセットを含むアドレスが *Rn* にライトバックされます。

使用法

これらの命令の使用方法はコプロセッサによって異なります。詳細については、コプロセッサのマニュアルを参照して下さい。

Thumb-2EE では、ベースレジスタの値が 0 の場合、HandlerBase - 4 にある NullCheck ハンドラへの分岐が実行されます。

例外

未定義命令が発生します。データアボートが発生します。

アーキテクチャ

これらの ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

注

STC 命令での PC 相対アドレッシングの使用は、ARMv6T2 で廃止されます。

4.9.5 LDC2、STC2

メモリとコプロセッサ間のデータ転送を実行する代替の命令です。

構文

これらの命令には、以下の 3 つの形式を使用できます。

- ゼロオフセット
- プレインデクスオフセット
- ポストインデクスオフセット

この 3 つの形式の構文を、上記と同じ順番で以下に示します。

```
op{L} coproc, CRd, [Rn]  
op{L} coproc, CRd, [Rn, #{-}offset]{!}
```

```
op{L} coproc, CRd, [Rn], #{-}offset
```

パラメータの説明：

<i>op</i>	LDC2 または STC2 を指定します。
<i>L</i>	長い転送を指定するオプションの接尾文字を指定します。
<i>coproc</i>	命令が実行されるコプロセッサの名前を指定します。標準名は <i>p<i>n</i></i> で、 <i>n</i> は 0 ~ 15 の整数です。
<i>CRd</i>	ロードまたはストアを実行するコプロセッサレジスタを指定します。
<i>Rn</i>	メモリアドレスのベースとなるレジスタを指定します。 <i>r15</i> が指定されている場合、使用される値は「現在の命令のアドレス + 8」となります。
-	任意に指定できるマイナス符号です。- が指定されている場合、オフセットが <i>Rn</i> から減算されます。指定されていない場合は、オフセットが <i>Rn</i> に加算されます。
<i>offset</i>	0 ~ 1020 の範囲で 4 の倍数を求める式を指定します。
!	任意に指定できる接尾文字です。! を指定すると、オフセットを含むアドレスが <i>Rn</i> にライトバックされます。

使用法

これらの命令の使用方法はコプロセッサによって異なります。詳細については、コプロセッサのマニュアルを参照して下さい。

注

LDC2 と STC2 は常に条件なしで実行されます。

例外

未定義命令が発生します。データアボートが発生します。

アーキテクチャ

これらの ARM 命令は、ARMv5 以上で使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

4.10 その他の命令

このセクションは以下のサブセクションから構成されています。

- *BKPT* (P. 4-138)
ブレークポイント命令です。
- *SVC* (P. 4-139)
スーパーバイザコール (以前の SWI) です。
- *MRS* (P. 4-140)
CPSR または SPSR の内容の汎用レジスタへの移動命令です。
- *MSR* (P. 4-141)
CPSR または SPSR の指定フィールドへの、イミディエート定数または汎用レジスタの内容のロード命令です。
- *CPS* (P. 4-143)
プロセッサ状態を変更する命令です。
- *SMC* (P. 4-145)
セキュアモニターコール (以前の SMI) です。
- *SETEND* (P. 4-146)
CPSR 内のエンディアンビットを設定します。
- *NOP*, *SEV*, *WFE*, *WFI*, *YIELD* (P. 4-147)
操作なし、イベントの設定、イベント待機、割り込み待機、および明け渡しを行うヒント命令です。
- *DBG*, *DMB*, *DSB*, *ISB* (P. 4-149)
デバッグ、データメモリバリア、データ同期バリア、および命令同期バリアヒント命令です。
- *MAR*, *MRA* (P. 4-151)
XScale コプロセッサ 0 命令です。
2 本の汎用レジスタと 40 ビット内部アキュムレータの間で転送を実行します。

- *MAR*, *MRA* (P. 4-151)
XScale コプロセッサ 0 命令です。
2 本の汎用レジスタと 40 ビット内部アキュムレータの間で転送を実行します。
- *ENTERX*, *LEAVEX* (P. 4-152)
ThumbEE 状態との間で状態の切り替えを行う命令です。
- *CHKA* (P. 4-153)
配列をチェックする命令です。
- *HB*, *HBL*, *HBLP*, *HBP* (P. 4-154)
ハンドラの分岐命令です。指定されたハンドラに分岐します。

4.10.1 BKPT

ブレークポイント命令です。

構文

BKPT *immed*

パラメータの説明：

immed 以下の範囲の整数を求める式を指定します。

- ARM 命令の場合は 0 ~ 65536 (16 ビット値)
- 16 ビット Thumb 命令の場合は 0 ~ 255 (8 ビット値)

使用法

BKPT 命令によって、プロセッサはデバッグ状態に入ります。デバッグツールは、この動作を使用して、特定のアドレスにある命令に到達した時点でシステム状態を調査することができます。

ARM 状態と Thumb 状態のどちらの場合も、*immed* は ARM ハードウェアによって無視されます。ただし、デバッガはこの値を使用して、ブレークポイントに関する情報を保持できます。

アーキテクチャ

この ARM 命令は、ARMv5 以上で使用できます。

この 16 ビット Thumb 命令は、ARMv5 以上の T バリアントで使用できます。

この命令の 32 ビット Thumb-2 バージョンはありません。

4.10.2 SVC

スーパーバイザコールです。

構文

SVC{*cond*} *immed*

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

immed 以下の範囲の整数を求める式を指定します。

- ARM 命令の場合は 0 ~ $2^{24}-1$ (24 ビット値)
- 16 ビット Thumb 命令の場合は 0 ~ 255 (8 ビット値)

使用法

SVC 命令によって例外が発生します。つまり、プロセッサモードがスーパーバイザモードに変更され、CPSR がスーパーバイザモードの SPSR に保存され、SVC ベクタへの分岐が実行されます (*RealView Compilation Tools v3.0 Developer Guide* の「プロセッサ例外処理」参照)。

immed はプロセッサによって無視されます。ただし、この値は、例外ハンドラで取得して、要求されているサービスを特定することができます。

——注——

ARM アセンブリ言語を開発する一環として、SWI 命令の名前が SVC に変更されました。RVCT の本リリースでは、SWI 命令は、以前は SWI だった (formerly SWI) というコメント付きで SVC に逆アセンブルされます。

条件フラグ

この命令によるフラグへの影響はありません。

アーキテクチャ

この ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

この 16 ビット Thumb 命令は、ARM アーキテクチャのすべての T バリアントおよび ARMv6 以上の T2 バリアントで使用できます。

4.10.3 MRS

CPSR または SPSR の内容の汎用レジスタへの移動命令です。

構文

MRS{cond} Rd, psr

パラメータの説明 :

- | | |
|-------------|--|
| <i>cond</i> | 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。 |
| <i>Rd</i> | デスティネーションレジスタを指定します。 <i>Rd</i> に r15 は指定できません。 |
| <i>psr</i> | CPSR または SPSR を指定します。 |

使用法

MRS と MSR を組み合わせ、PSR を更新するための読み出し - 修正 - 書き込みシーケンスで使用することにより、プロセッサモードの変更や Q フラグのクリアなどを行うことができます。

プロセススワップコードでは、関連する PSR の内容を含め、スワップアウトされているプロセスのプログラマのモデルの状態を保存する必要があります。同様に、スワッピングされているプロセスの状態も復元する必要があります。これらの操作には、**MRS/store** および **load/MSR** 命令シーケンスを使用します。

注意

プロセッサがユーザモードやシステムモードの場合は、SPSR へアクセスしないで下さい。これはユーザ自身が注意する必要があります。アセンブラーは、コードがどのプロセッサモードで実行されるかを検知しないため、警告メッセージを生成できません。

SPSR にアクセスした場合、結果は予測不可能であり、信頼できません。

プロセッサがデバッグ状態でデバッグモードを完全停止しているときにのみ、CPSR 実行状態ビットを読み出すことができます。それ以外の場合、CPSR の実行状態ビットはゼロとして読み出されます。

条件フラグ

この命令によるフラグへの影響はありません。

アーキテクチャ

この ARM 命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

この 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

この命令の 16 ビット Thumb バージョンはありません。

4.10.4 MSR

CPSR または SPSR の指定フィールドへの、イミディエート定数または汎用レジスタの内容のロード命令です。

構文

`MSR{cond} <psr>_<fields>, #constant`

`MSR{cond} <psr>_<fields>, Rm`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

<psr> CPSR または SPSR を指定します。

<fields> 移動するプログラム状態レジスタ (PSR) フィールドを指定します（複数可）。*<fields>* には以下の 1 つ以上を指定できます。

c 制御フィールドマスクバイト、PSR[7:0]

x 拡張フィールドマスクバイト、PSR[15:8]

s 状態フィールドマスクバイト、PSR[23:16]

f フラグフィールドマスクバイト、PSR[31:24]

constant 数値定数を求める式を指定します。この定数は、32 ビットのワード内で偶数ビット数分ロテートして得られる 8 ビットパターンに対応している必要があります。

——注——

この命令の `#constant` 形式は Thumb-2 では使用できません。

Rm ソースレジスタを指定します。

使用法

詳細については、*MRS* (P. 4-140) を参照して下さい。

ユーザモードでは、以下のようになります。

- CPSR の未割り当てるビット、特権付きビット、または実行状態のビットへの書き込みが無視されます。そのため、ユーザモードのプログラムが特権モードに変更されることはありません。
- ARMv6T2 以上では、(StateMask で定義された) CPSR の状態ビットへの書き込みが無視されます。この動作は、以前は、予測不可能であり、信頼できないものでした。

ユーザモードまたはシステムモードの場合に SPSR にアクセスすると、その結果は予測不可能であり、信頼できません。

条件フラグ

この命令は、f フィールドが指定されている場合にフラグを明示的に更新します。

アーキテクチャ

この ARM 命令は、ARMv3 以上で使用できます。

この 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

この命令の 16 ビット Thumb バージョンはありません。

4.10.5 CPS

CPS（プロセッサ状態の変更命令）は、他の CPSR ビットを変更することなく、CPSR のモード、A、I、およびF のビットの 1 つ以上を変更します。

CPS は、特権モードのみで使用でき、ユーザモードでは作用しません。

CPS は、条件付きにすることはできないため、IT ブロックでは使用できません。

構文

`CPSEffect iflags{, #mode}`

`CPS #mode`

パラメータの説明：

`effect` 次のいずれかを指定します。

`IE` 割り込みまたはアボートをイネーブルします。

`ID` 割り込みまたはアボートをディセーブルします。

`iflags` 以下の 1 つ以上のシーケンスを指定します。

`a` 不正確なアボートをイネーブルまたはディセーブルします。

`i` IRQ 割り込みをイネーブルまたはディセーブルします。

`f` FIQ 割り込みをイネーブルまたはディセーブルします。

`mode` 変更先のモードの番号を指定します。

条件フラグ

この命令による条件フラグへの影響はありません。

16 ビット命令

これらの命令は、Thumb コード内では次の形式で使用できます。また、Thumb-2 コード内で使用するときは 16 ビット命令になります。

`CPSIE iflags` 16 ビット Thumb 命令ではモードの変更を指定できません。

`CPSID iflags` 16 ビット Thumb 命令ではモードの変更を指定できません。

アーキテクチャ

この ARM 命令は、ARMv6 以上で使用できます。

この 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントで使用できます。

この 16 ビット Thumb 命令は、ARMv6 以上の T バリアントで使用できます。

例

```
CPSIE if      ; enable interrupts and fast interrupts  
CPSID A      ; disable imprecise aborts  
CPSID ai, #17 ; disable imprecise aborts and interrupts, and enter FIQ mode  
CPS #16       ; enter User mode
```

誤用例

```
CPSEQ #19      ; CPS cannot be conditional
```

4.10.6 SMC

セキュアモニターコールです。

詳細については、*ARM Architecture Reference Manual Security Extensions Supplement* を参照して下さい。

構文

`SMC{cond} #immed_16`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

immed_16 16 ビットのイミディエート値を指定します。この値は、ARM プロセッサでは無視されますが、SMC 例外ハンドラで使用して、要求されているサービスを特定することができます。

使用法

ARM アセンブリ言語を開発する一環として、SMI 命令の名前が SMC に変更されました。RVCT の本リリースでは、SMI 命令は、以前は SMI だったというコメント付きで SMC に逆アセンブルされます。

アーキテクチャ

この ARM 命令は、ARMv6 以上の Z バリアントで使用できます。

この 32 ビット Thumb-2 命令は、ARMv6T2 以上の Z バリアントで使用できます。

この命令の 16 ビット Thumb バージョンはありません。

4.10.7 SETEND

CPSR 内のエンディアンビットを、他のビットに影響を与えることなく設定します。

SETEND は、条件付きにすることはできないため、IT ブロックでは使用できません。

構文

`SETEND specifier`

パラメータの説明：

specifier 次のいずれかを指定します。

BE ビッグエンディアン形式

LE リトルエンディアン形式

使用法

SETEND を使用すると、異なるエンディアン形式のデータにアクセスできます。例えば、リトルエンディアン形式のアプリケーションから、ビッグエンディアン形式で DMA フォーマットされたデータフィールドにアクセスできます。

アーキテクチャ

この ARM 命令は、ARMv6 以上で使用できます。

この 16 ビット Thumb 命令は、ARMv6 以上の T バリアントで使用できます。

この命令の 32 ビット Thumb-2 バージョンはありません。

例

```
SETEND BE          ; Set the CPSR E bit for big-endian accesses
LDR    r0, [r2, #header]
LDR    r1, [r2, #CRC32]
SETEND le          ; Set the CPSR E bit for little-endian accesses for the
                   ; rest of the application
```

4.10.8 NOP、SEV、WFE、WFI、YIELD

操作なし、イベントの設定、イベント待機、割り込み待機、および明け渡しを行うヒント命令です。

構文

`NOP{cond}`

`SEV{cond}`

`WFE{cond}`

`WFI{cond}`

`YIELD{cond}`

パラメータの説明：

`cond` 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

使用法

これらはヒント命令です。これらの命令は任意に実装することができます。いずれの命令も実装しない場合は、`NOP` として動作します。

NOP

`NOP` は何も行いません。`NOP` がターゲットアーキテクチャで特定の命令として実装されない場合、アセンブラーは、`MOV r0, r0` (ARM) や `MOV r8, r8` (Thumb) など、何も行わない別の命令を生成します。

`NOP` は、必ずしも時間のかかる命令ではありません。プロセッサにより、この命令は、実行ステージに到達する前にパイプラインから削除される場合があります。

例えば、パディングに `NOP` を使用することで次に続く命令を 64 ビット境界に配置することができます。

SEV

`SEV` により、マルチプロセッサシステム内のすべてのコアに対してイベントを発生させます。`SEV` を実装する場合は、`WFE` も実装する必要があります。

WFE

イベントレジスタが設定されていない場合、**WFE** は、以下のいずれかのイベントが発生するまで実行を中断します。

- IRQ 割り込み (CPSR の I ビットでマスクされている場合を除く)
- FIQ 割り込み (CPSR の F ビットでマスクされている場合を除く)
- 不正確なデータアポート (CPSR の A ビットでマスクされている場合を除く)
- デバッグエントリ要求 (デバッグがイネーブルの場合)
- 別のプロセッサが **SEV** 命令を使用して発生したイベント

イベントレジスタが設定されている場合、**WFE** は、そのレジスタをクリアしてすぐに戻ります。

WFE を実装する場合は、**SEV** も実装する必要があります。

WFI

WFI は、以下のいずれかのイベントが発生するまで実行を中断します。

- IRQ 割り込み (CPSR の I ビットの設定とは無関係)
- FIQ 割り込み (CPSR の F ビットの設定とは無関係)
- 不正確なデータアポート (CPSR の A ビットでマスクされている場合を除く)
- デバッグエントリ要求 (デバッグがイネーブルされているかどうかは無関係)

YIELD

YIELD は、現在のスレッドが、スワップアウトできるタスク（スピンドルなど）を実行していることをハードウェアに示します。ハードウェアは、このヒントを使用して、マルチスレッドシステムでスレッドを中断および再開できます。

アーキテクチャ

これらの ARM 命令は、ARMv6T2 以上、および ARMv6 の K バリアントで使用できます。

これらの 32 ビット Thumb-2 命令は、ARMv6 以上の T2 バリアントおよび ARMv6 の K バリアントで使用できます。

これらの 16 ビット Thumb 命令は、ARMv6T2 以上、および ARMv6 の K バリアントで使用できます。

4.10.9 DBG、DMB、DSB、ISB

デバッグ、データメモリバリア、データ同期バリア、および命令同期バリアです。

構文

`DBG{cond} {#option}`

`DMB{cond} {#option}`

`DSB{cond} {#option}`

`ISB{cond} {#option}`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P.2-17) 参照）。

option ヒントの演算に任意に指定できる制限です。

使用法

これらはヒント命令です。これらの命令は任意に実装することができます。いずれの命令も実装しない場合は、`NOP` として動作します。

DBG

デバッグヒントは、デバッグシステムおよび関連するシステムにヒントを提供します。この命令がシステムでどのように使用されているかについては、システムのマニュアルを参照して下さい。

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。これにより、`DMB` 命令より前にプログラム順で発生する明示的なすべてのメモリアクセスは、`DMB` 命令より後にプログラム順で出てくる明示的なデータアクセスよりも先に観察されます。これは、プロセッサで実行している他の命令の順序に影響することはありません。

option に使用できる値は以下のとおりです。

`SY` システム全体の `DMB` 演算。これはデフォルト値なので、省略できます。

DSB

データ同期バリアは、特殊なメモリバリアとして機能します。この命令が完了するまで、この命令より後にあるプログラム順の命令は実行されません。この命令は以下の場合に完了します。

- この命令が完了する前のすべての明示的なメモリアクセス
- この命令が完了する前のキャッシュ、分岐予測子、および TLB メンテナンスのすべての処理

使用できる値は以下のとおりです。

SY	システム全体の DSB 演算。これはデフォルト値なので、省略できます。
UN	統合ポイントのみを対象とした DSB 演算。
ST	ストアの完了のみを待機する DSB 演算。
UNST	ストアの完了のみを待機し、統合ポイントのみを対象とした DSB 演算。

ISB

命令同期バリアはプロセッサのパイプラインをフラッシュするため、ISB に続くすべての命令は、ISB 命令が完了した後、キャッシュまたはメモリからフェッチされます。これにより、ISB 命令より前に実行されたコンテキスト変更処理 (ASID の変更など)、完了した TLB メンテナンス処理、分岐予測子メンテナンス処理、および CP15 レジスタへのすべての変更は、ISB より後にフェッチされた命令で認識されます。

また、ISB 命令により、この命令より後にプログラム順で出てくるすべての分岐は、必ず ISB 命令より後で認識されるコンテキストと共に分岐予測論理に書き込まれます。これは、命令ストリームを正しく実行するために必要なことです。

option に使用できる値は以下のとおりです。

SY	システム全体の DMB 演算。これはデフォルト値なので、省略できます。
----	-------------------------------------

アーキテクチャ

これらの ARM 命令および 32 ビット Thumb-2 命令は、ARMv7 以上で使用できます。

これらの命令の 16 ビット Thumb バージョンはありません。

4.10.10 MAR、MRA

XScale プロセッサ 0 命令です。

2 本の汎用レジスタと 40 ビット内部アキュムレータの間で転送を実行します。

構文

MAR{*cond*} *Acc*, *RdLo*, *RdHi*

MRA{*cond*} *RdLo*, *RdHi*, *Acc*

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

Acc 内部アキュムレータを指定します。標準名は *accx* で、*x* は 0 ~ *n* の整数です。*n* の値はプロセッサによって異なります。最近のプロセッサでは 0 が使用されています。

RdLo, *RdHi* 汎用レジスタを指定します。

使用法

MAR 命令は、*RdLo* の内容を *Acc* のビット [31:0] にコピーし、*RdHi* の最下位バイトを *Acc* のビット [39:32] にコピーします。

MRA 命令は以下を実行します。

- *Acc* のビット [31:0] を *RdLo* にコピーします。
- *Acc* のビット [39:32] を *RdHi* のビット [7:0] にコピーします。
- *Acc* のビット [39] を *RdHi* のビット [31:8] をコピーして値を符号拡張します。

アーキテクチャ

これらの ARM 命令は、XScale プロセッサでのみ使用できます。

これらの命令の Thumb バージョンおよび Thumb-2 バージョンはありません。

例

```

MAR      acc0, r0, r1
MRA      r4, r5, acc0
MARNE   acc0, r9, r2
MRAGT   r4, r8, acc0

```

4.10.11 ENTERX、LEAVEX

ENTERX を使用すると、Thumb 状態から ThumbEE 状態に切り替わるか、または ThumbEE 状態が維持されます。

LEAVEX を使用すると、ThumbEE 状態から Thumb 状態に切り替わるか、または Thumb 状態が維持されます。

構文

ENTERX

LEAVEX

使用法

IT ブロック内では、ENTERX または LEAVEX を使用しないで下さい。

アーキテクチャ

これらの命令は、ARM 命令セットでは使用できません。

これらの 32 ビット Thumb 命令および Thumb-2EE 命令は、Thumb-2EE をサポートする ARMv7 で使用できます。

これらの命令の 16 ビット Thumb-2 バージョンはありません。

詳細については、*ARM Architecture Reference Manual Thumb-2 Execution Environment Supplement* を参照して下さい。

4.10.12 CHKA

CHKA (配列のチェック命令) は、2本のレジスタにある符号なしの値を比較します。

最初のレジスタの値が2番目のレジスタの値以下、或いは等しい場合、この命令により、プログラムカウンタがlrにコピーされ、IndexCheck ハンドラへの分岐が発生します。どちらのオペランドレジスタにもr0～r15のいずれかを指定できます。

構文

CHKA Rn, Rm

パラメータの説明：

Rn 配列のサイズを指定します。

Rm 配列のインデクスを指定します。

アーキテクチャ

ARM 状態では使用できません。

この16ビット命令は、Thumb-2EEをサポートするARMv7のみで使用できます。

4.10.13 HB、HBL、HBLP、HBP

ハンドラの分岐命令です。指定されたハンドラに分岐します。

この命令は、必要に応じて、復帰アドレスを lr に保存したり、パラメータをハンドラに渡すことができます。また、その両方を行うこともできます。

構文

`HB{L} #HandlerID`

`HB{L}{P} {#immed,} #HandlerID`

パラメータの説明：

`L` 任意に指定できる接尾文字です。L を指定すると、復帰アドレスが lr に保存されます。

`P` 任意に指定できる接尾文字です。P を指定すると、`immed` の値が r8 のハンドラに渡されます。

`immed` イミディエート値を指定します。L を指定した場合、`immed` に 0 ~ 31 の範囲内の値を指定する必要があります。この接尾文字を指定しない場合、`immed` には 0 ~ 7 の範囲内の値を指定する必要があります。

`HandlerID` 呼び出すハンドラのインデクス番号を指定します。P を指定した場合、`HandlerID` に 0 ~ 31 の範囲内の値を指定する必要があります。この接尾文字を指定しない場合、`HandlerID` には 0 ~ 255 の範囲内の値を指定することができます。

アーキテクチャ

これらの命令は ARM 状態では使用できません。

これらの 16 ビット命令は、Thumb-2EE をサポートする ARMv7 の ThumbEE 状態でのみ使用できます。

4.11 擬似命令

ARM アセンブラーは、多くの擬似命令をサポートしています。これらの擬似命令は、アセンブリ時に適切な ARM 命令、Thumb 命令、または Thumb-2 命令の組み合わせに変換されます。

擬似命令については、以下のサブセクションを参照して下さい。

- *ADRL 擬似命令 (P. 4-156)*
プログラム相対アドレスまたはレジスタ相対アドレス（中範囲、位置非依存）をレジスタにロードします。
- *MOV32 擬似命令 (P. 4-158)*
32 ビット定数値またはアドレス（範囲無制限、位置依存）をレジスタにロードします。ARMv6T2 以上でのみ使用できます。
- *LDR 擬似命令 (P. 4-160)*
32 ビット定数値またはアドレス（範囲無制限、位置依存）をレジスタにロードします。すべての ARM アーキテクチャで使用できます。

4.11.1 ADRL 擬似命令

プログラム相対アドレスまたはレジスタ相対アドレスをレジスタにロードします。この命令は ADR 擬似命令と似ています。ADRL では、2 つのデータ処理命令が生成されるため、ADR より広範囲のアドレスをロードできます。

——注——

16 ビットThumb 命令をアセンブルする場合には、ADRL は使用できません。ARM コードまたはThumb-2 コードでのみ使用して下さい。

構文

`ADRL{cond} register, label`

パラメータの説明：

`cond` 任意に指定できる条件コードです（「条件実行」（P. 2-17）参照）。

`register` ロードするレジスタを指定します。

`label` レジスタ相対式またはプログラム相対式を指定します。詳細については、「レジスタ相対式とプログラム相対式」（P. 3-31）を参照して下さい。

使用法

ADRL は常に 2 つの 32 ビット命令にアセンブルされます。1 つの命令でアドレスに到達できる場合でも、2 番目の冗余命令が生成されます。

アセンブラーが 2 つの命令でアドレスを作成できない場合は、エラーメッセージが生成され、アセンブルに失敗します。より広い範囲のアドレスをロードする方法については、「*LDR 擬似命令*」（P. 4-160）を参照して下さい。「レジスタへの定数のロード」（P. 2-23）も参照して下さい。

ADRL は、アドレスがプログラム相対またはレジスタ相対であるため、位置非依存コードを生成します。

`label` にプログラム相対式を指定する場合は、ADRL 擬似命令と同じアセンブラー領域内のアドレスを指定する必要があります（「*AREA*」（P. 7-66）参照）。

範囲

利用できる範囲は、使用する命令セットによって異なります。

ARM

バイト境界またはハーフワード境界で整列されているアドレス
から $\pm 64KB$ の範囲

ワード境界で整列されているアドレスから $\pm 256KB$ バイトの範囲

32 ビットの Thumb-2 バイト、ハーフワード、またはワード境界で整列されているア
ドレスから $\pm 1MB$ バイトの範囲

16 ビットの Thumb ADRL 命令は使用できません。

上記の範囲は、現在の命令のアドレスの 2 ワード後の位置の相対範囲です。ARM と
Thumb-2 では、境界調整がこの位置から 16 バイト以上の相対位置にある場合、より広
範囲のアドレスを利用できます。

4.11.2 MOV32 擬似命令

以下のいずれかの値をレジスタにロードします。

- 32 ビット定数値
- 任意のアドレス

MOV32 は常に 2 つの 32 ビット命令を生成します。

MOV32 を使用してアドレスをロードした場合、生成されるコードは位置依存コードになります。

構文

`MOV32{cond} register, expr`

パラメータの説明 :

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

register ロード先のレジスタを指定します。

expr 以下のいずれかを指定できます。

symbol このプログラム領域または別のプログラム領域内のラベルです。

constant 任意の 32 ビット定数です。

symbol + constant ラベルと 32 ビット定数を組み合わせたものです。

アセンブラーからは、MOV と MOVT 命令対が生成されます。この命令対を使用して、32 ビット定数をロードしたり、アドレス空間全体にアクセスしたりすることができます。

symbol に外部を指定すると、アセンブラーがリンク再配置ディレクティブをオブジェクトファイル内に配置します。リンクはリンク時にアドレスを生成します。

使用法

MOV32 擬似命令の主な目的は以下のとおりです。

- 1 つの命令でイミディエート値を生成できない場合にリテラル定数を生成すること。
- プログラム相対アドレスまたは外部アドレスをレジスタにロードすること。このアドレスは、リンクが MOV32 を保持する ELF セクションをどこに配置しても有効です。

—— 注 ——

この方法でロードされたアドレスはリンク時に固定されるため、このコードは位置依存コードになります。

アーキテクチャ

この擬似命令は、ARM と Thumb-2 の両方の ARMv6 以上の T2 バリアントで使用できます。

4.11.3 LDR 擬似命令

以下のいずれかの値をレジスタにロードします。

- 32 ビット定数値
- アドレス

——注——

ここでは、LDR 擬似命令についてのみ説明します。LDR 命令の詳細については、「メモリアクセス命令」(P. 4-8) を参照して下さい。

また、LDR 擬似命令を使用した定数のロードについては、「*LDR Rd, =const を使用したロード*」(P. 2-28) を参照して下さい。

構文

`LDR{cond}{.w} register,[expr | label Expr]`

パラメータの説明：

cond 任意に指定できる条件コードです（「条件実行」(P. 2-17) 参照）。

.w 任意に指定できる幅指定子を指定です。詳細については、「*Thumb-2 の LDR*」(P. 4-162) を参照して下さい。

register ロード先のレジスタを指定します。

expr 数値定数を求める式を指定します。

- *expr* の値が範囲内の場合、アセンブラーは MOV 命令または MVN 命令を生成します。
- *expr* の値が MOV 命令または MVN 命令の範囲外の場合、アセンブラーはその定数をリテラルプールに配置し、リテラルプールから定数を読み出すプログラム相対 LDR 命令を生成します。

label Expr プログラム相対式または外部式を指定します。アセンブラーは、*label Expr* の値をリテラルプールに配置し、リテラルプールからこの値をロードするプログラム相対 LDR 命令を生成します。

label Expr が外部式であるか、または現在のセクションに含まれていない場合、アセンブラーはリンク再配置ディレクティブをオブジェクトファイル内に配置します。リンクはリンク時にアドレスを生成します。

label Expr がローカルラベル（「ローカルラベル」(P. 3-25) 参照）の場合、アセンブラーはリンク再配置ディレクティブをオブジェクトファイル内に配置し、そのローカルラベルのシンボルを生成します。アドレスはリンク時に生成されます。ローカルラベルが Thumb コードを参照する場合、アドレスの Thumb ビット（ビット 0）が設定されます。

——注——

RVCT2.2 では、アドレスの Thumb ビットが設定されませんでした。この動作に依存するコードでは、コマンドラインオプション `--untyped_local_labels` を使用して、アセンブラーが Thumb コード内のラベルを参照する際に Thumb ビットを設定しないようにします。

使用法

`LDR` 擬似命令の主な目的は以下のとおりです。

- イミディエート値が `MOV` 命令および `MVN` 命令の範囲外にあるため、レジスタに移動できない場合に、リテラル定数を生成すること。
- プログラム相対アドレスまたは外部アドレスをレジスタにロードすること。このアドレスは、リンクが `LDR` を保持する ELF セクションをどこに配置しても有効です。

——注——

この方法でロードされたアドレスはリンク時に固定されるため、このコードは位置依存コードになります。

プログラムカウンタからリテラルプールの値へのオフセットは、 $\pm 4KB$ 未満 (ARM および 32 ビット Thumb-2 の場合) または $0 \sim +1KB$ の範囲内 (Thumb および 16 ビット Thumb-2 の場合) にする必要があります。リテラルプールが範囲内にあることを必ず確認して下さい。詳細については、「`LTORG`」(P. 7-19) を参照して下さい。

`LDR` の使用方法、および `MOV` と `MVN` の詳細については、「レジスタへの定数のロード」(P. 2-23) を参照して下さい。

アーキテクチャ

この ARM 擬似命令は、ARM アーキテクチャのすべてのバージョンで使用できます。

32 ビット Thumb-2 については、「`Thumb-2 の LDR`」(P. 4-162) を参照して下さい。

この 16 ビット Thumb 擬似命令は、ARM アーキテクチャのすべての T バリアントで使用できます。

Thumb-2 の LDR

.W 幅指定子を使用して、LDR で Thumb-2 コードの 32 ビット命令を生成することができます。

LDR.W は、定数が 16 ビット MOV でロードされる場合やリテラルプールが 16 ビット PC 相対ロードの範囲内にある場合でも、常に 32 ビット命令を生成します。

.W を指定せずに LDR を実行すると、Thumb コードでは常に 16 ビット命令が生成されます。これには、演算結果として、32 ビット MOV 命令または MVN 命令で生成できる定数の 16 ビット PC 相対ロードが発生する場合も含まれます。

.W を指定せずに LDR を実行しても、16 ビットのフラグを設定する MOV 命令が生成されることはありません。アセンブラーの `--diag_warning 1727` コマンドラインオプションを使用して、16 ビット命令の使用を確認できます。

また、MOV32 も使用できます（「MOV32 擬似命令」（P. 4-158）参照）。この命令では、2 つの命令を使用して、ロード命令を使用せずに 32 ビット値をレジスタに配置できます。

例

```

LDR    r3,=0xff0      ; loads 0xff0 into r3
      ; => MOV.W r3,#0xff0
LDR    r1,=0xffff     ; loads 0xffff into r1
      ; => LDR r1,[pc,offset_to_litpool]
      ;
      ...
      ;      litpool DCD 0xffff
LDR    r2,=place       ; loads the address of
      ; place into r2
      ; => LDR r2,[pc,offset_to_litpool]
      ;
      ...
      ;      litpool DCD place

```

第 5 章

NEON と VFP プログラミング

本章では、NEON™ とベクタ浮動小数点 (VFP) コプロセッサをアセンブリ言語でプログラミングする方法について参照情報を示します。本章は以下のセクションから構成されています。

- NEON / VFP レジスタバンク (P. 5-8)
- 条件コード (P. 5-10)
- 一般的な情報 (P. 5-12)
- NEON と VFP に共通の命令 (P. 5-17)
- NEON 論理演算と比較演算 (P. 5-24)
- NEON 汎用データ処理命令 (P. 5-32)
- NEON シフト命令 (P. 5-44)
- NEON 汎用算術命令 (P. 5-51)
- NEON 乗算命令 (P. 5-64)
- NEON 要素と構造体のロード / ストア命令 (P. 5-69)
- NEON 擬似命令 (P. 5-77)
- ベクタ浮動小数点コプロセッサ (P. 5-81)
- VFP レジスタ (P. 5-82)
- VFP ベクタ演算とスカラ演算 (P. 5-85)
- VFP / NEON システムレジスタ (P. 5-87)

- ゼロクリアモード (P. 5-91)
- *VFP 命令* (P. 5-93)
- *VFP 擬似命令* (P. 5-108)
- *VFP ディレクティブとベクタ表記* (P. 5-109)

各命令の詳細については、表 5-1、P. 5-6 表 5-2、および P. 5-6 表 5-3 を参照して下さい。

表 5-1 NEON 命令の参照ページ

ニーモニック	意味	ページ
VABA、VABD	絶対差、絶対差と累積	P. 5-52
VABS	絶対値	P. 5-53
VACGE	以上 (絶対値比較)	P. 5-29
VACGT	超 (絶対値比較)	P. 5-29
VACLE	以下 (絶対値比較、擬似命令)	P. 5-79
VACLT	未満 (絶対値比較、擬似命令)	P. 5-79
VADD	加算	P. 5-54
VADDHN	加算、上位半分の選択	P. 5-55
VAND	ビット単位論理積	P. 5-25
VAND	ビット単位論理積 (擬似命令)	P. 5-78
VBIC	ビット単位ビットクリア (レジスタ)	P. 5-25
VBIC	ビット単位ビットクリア (イミディエート)	P. 5-26
VBIF	False の場合はビット単位を挿入	P. 5-27
VBIT	True の場合はビット単位を挿入	P. 5-27
VBSL	ビット単位の選択	P. 5-27
VCEQ	等しい (比較)	P. 5-30
VCLE、VCLT	以下 (比較)、未満 (比較)	P. 5-30
VCLE、VCLT	以下 (比較)、未満 (比較、擬似命令)	P. 5-80
VCLS、VCLZ、VCNT	先行符号ビットカウント、先行ゼロカウント、およびセットビットカウント	P. 5-60

表 5-1 NEON 命令の参照ページ（続き）

ニーモニック	意味	ページ
VCVT	固定小数点または整数から浮動小数点へ、浮動小数点から整数または固定小数点への変換	P. 5-33
VDUP	ベクタの全レーンへのスカラの複製	P. 5-34
VEXT	抽出	P. 5-35
VCGE	以上（比較）	P. 5-30
VCGT	超（比較）	P. 5-30
VEOR	ビット単位排他的論理和（XOR）	P. 5-25
VHADD	二分加算	P. 5-56
VHSUB	二分減算	P. 5-56
VMAX、VMIN	最大値、最小値	P. 5-59
VLD	ベクタロード	P. 5-69
VMLA	積和（ベクタ）	P. 5-65
VMLA	積和（スカラによる）	P. 5-66
VMLS	積差（ベクタ）	P. 5-65
VMLS	積差（スカラによる）	P. 5-66
VMOV	移動（イミディエート）	P. 5-36
VMOV	移動（レジスタ）	P. 5-37
VMOVL、VMOV{U}N	Long 移動、Narrow 移動（レジスタ）	P. 5-38
VMUL	乗算（ベクタ）	P. 5-65
VMUL	乗算（スカラによる）	P. 5-66
VMVN	負の移動（イミディエート）	P. 5-36
VNEG	否定	P. 5-53
VNOT	ビット単位否定	P. 5-28
VORN	ビット単位否定論理和	P. 5-25
VORN	ビット単位否定論理和（擬似命令）	P. 5-78

表 5-1 NEON 命令の参照ページ（続き）

ニーモニック	意味	ページ
VORR	ビット単位論理和（レジスタ）	P. 5-25
VROR	ビット単位論理和（イミディエート）	P. 5-26
VPADD、VPADAL	ペアワイズ加算、ペアワイズ加算累積	P. 5-57
VPMAX、VPMIN	ペアワイズ最大値、ペアワイズ最小値	P. 5-59
VQABS	絶対値、サチュレート	P. 5-53
VQADD	加算、サチュレート	P. 5-54
VQDMLAL、VQDMLSL	サチュレートダブル積和、積差	P. 5-67
VQMOV{U}N	サチュレート移動（レジスタ）	P. 5-38
VQDMUL	サチュレートダブル乗算	P. 5-67
VQDMULH	上位半分を返すサチュレートダブル乗算	P. 5-68
VQNEG	否定、サチュレート	P. 5-53
VQRDMULH	上位半分を返すサチュレートダブル乗算	P. 5-68
VQRSHL	左シフト、丸め、サチュレート（符号付き変数による）	P. 5-47
VQRSHR	右シフト、丸め、サチュレート（イミディエートによる）	P. 5-49
VQSHL	左シフト、サチュレート（イミディエートによる）	P. 5-45
VQSHL	右シフト、サチュレート（符号付き変数による）	P. 5-47
VQSHR	右シフト、サチュレート（イミディエートによる）	P. 5-49
VQSUB	減算、サチュレート	P. 5-54
VRADDH	加算、上位半分の選択、丸め	P. 5-55
VRECPE	逆数の推定	P. 5-61
VRECPS	逆数のステップ	P. 5-61
VREV	要素の順番の反転	P. 5-39
VRHADD	二分加算、丸め	P. 5-56
VRSHR	右シフト、丸め、（イミディエートによる）	P. 5-48

表 5-1 NEON 命令の参照ページ（続き）

ニーモニック	意味	ページ
VRSRA	右シフト、丸め、累積（イミディエートによる）	P. 5-48
VRSUBH	減算、上位半分の選択、丸め	P. 5-55
VRSQRTE	逆平方根の推定	P. 5-62
VRSQRTS	逆平方根のステップ	P. 5-62
VSHL	左シフト（イミディエートによる）	P. 5-45
VSHR	右シフト（イミディエートによる）	P. 5-48
VSLI	左シフトして挿入	P. 5-50
VSRA	右シフト、累積（イミディエートによる）	P. 5-48
VSRI	右シフトして挿入	P. 5-50
VST	ベクタストア	P. 5-69
VSUB	減算	P. 5-54
VSUBH	減算、上位半分の選択	P. 5-55
VSWP	ベクタのスワップ	P. 5-40
VTBL、VTBX	ベクタテーブルの検索	P. 5-41
VTST	テストビット	P. 5-31
VTRN	ベクタ置換	P. 5-42
VUZP、VZIP	ベクタのインターリープとインターリープの解除	P. 5-43

表 5-2 NEON と VFP に共通の命令の参照ページ

ニーモニック	意味	ページ	演算	アーキテクチャ
MRS	NEON / VFP システムレジスタから ARM レジスターへの転送	P. 5-23	-	すべて
MSR	ARM レジスタから NEON / VFP システムレジスターへの転送	P. 5-23	-	すべて
VLDR	ロード (『VLDR 擬似命令』(P. 5-108) も参照)	P. 5-18	スカラ	すべて
VLDM	多重ロード	P. 5-19	-	すべて
VMOV	1 本の ARM® レジスタから半分の倍精度への転送	P. 5-21	スカラ	すべて
VMOV	2 本の ARM レジスタから倍精度への転送	P. 5-20	スカラ	VFPv2
VMOV	半分の倍精度から ARM レジスタへの転送	P. 5-21	スカラ	すべて
VMOV	倍精度から 2 本の ARM レジスタへの転送	P. 5-20	スカラ	VFPv2
VMOV	単精度から ARM レジスタへの転送	P. 5-22	スカラ	すべて
VMOV	ARM レジスタから単精度への転送	P. 5-22	スカラ	すべて
VSTR	ストア	P. 5-18	スカラ	すべて
VSTM	多重ストア	P. 5-19	-	すべて

表 5-3 VFP 命令の参照ページ

ニーモニック	意味	ページ	演算	アーキテクチャ
FABS	絶対値	P. 5-94	ベクタ	すべて
FADD	加算	P. 5-95	ベクタ	すべて
FCMP	比較	P. 5-96	スカラ	すべて
FCONSTS、 FCONSTD	単精度または倍精度レジスタへの浮動小数点定数の配置	P. 5-105	スカラ	VFPv3
FCPY	コピー	P. 5-94	ベクタ	すべて
FCVTDS	単精度から倍精度への変換	P. 5-97	スカラ	すべて
FCVTSID	倍精度から単精度への変換	P. 5-98	スカラ	すべて

表 5-3 VFP 命令の参照ページ (続き)

ニーモニック	意味	ページ	演算	アーキテクチャ
FDIV	除算	P. 5-99	ベクタ	すべて
FMAC	乗累積	P. 5-100	スカラ	すべて
FMSC	乗減算	P. 5-100	ベクタ	すべて
FMUL	乗算	P. 5-101	ベクタ	すべて
FNEG	否定	P. 5-94	ベクタ	すべて
FNMAC	否定乗累積	P. 5-100	ベクタ	すべて
FNMSC	否定乗減算	P. 5-100	ベクタ	すべて
FNmul	否定乗算	P. 5-101	ベクタ	すべて
FSHTOS、FSHTOD、 FSLTOS、FSLTOD、 FUHTOS、FUHTOD、 FULTOS、FULTOD	固定小数点から単精度または倍精度浮動小数点 への変換	P. 5-106	スカラ	VFPv3
FSITO	符号付き整数から浮動小数点への変換	P. 5-102	スカラ	すべて
FSQRT	平方根	P. 5-103	ベクタ	すべて
FSUB	減算	P. 5-95	ベクタ	すべて
FTOSHS、FTOSHD、 FTOSLS、FTOSLD、 FTOUHS、FTOUHD、 FTOULS、FTOULD	単精度または倍精度浮動小数点から固定小数点 への変換	P. 5-107	スカラ	VFPv3
FTOSI、FTOUI	浮動小数点から符号付きまたは符号なし整数へ の変換	P. 5-104	スカラ	すべて
FUITO	符号なし整数から浮動小数点への変換	P. 5-102	スカラ	すべて

5.1 NEON / VFP レジスタバンク

NEON と VFPv3 は同じレジスタバンクを使用しますが、ARM レジスタバンクとは異なります。VFPv2 レジスタバンクのスーパーセットです。

以下のセクションで説明するように、NEON / VFPv3 レジスタバンクは、明示的にエイリアスされた 3 つのビューを使用して参照できます。

P. 5-9 図 5-1 は、レジスタバンクの 3 つのビューであるワード、ダブルワード、およびクワッドワードの各レジスタを示し、それらが重複する様子を示しています。

5.1.1 レジスタバンクの NEON ビュー

レジスタバンクが以下の場合の NEON ビュー

- 16 本の 128 ビットクワッドワードレジスタ、Q0 ~ Q15
- 32 本の 64 ビットダブルワードレジスタ、D0 ~ D31。このビューは、VFPv3 でも使用できます。
- 上記のビューを使用したレジスタの組み合わせ

NEON ビューには、それぞれのレジスタにサイズとタイプがすべて同じ 1、2、4、8、または 16 要素のベクタが含まれています。各要素はスカラとしてもアクセスできます。

5.1.2 レジスタバンクの VFPv3 ビュー

VFPv3 で表示されるレジスタバンクは以下のとおりです。

- 32 本の 64 ビットダブルワードレジスタ、D0 ~ D31。このビューは、NEON でも使用できます。
- 32 本の 32 ビットシングルワードレジスタ、S0-S31。このビューではレジスタバンクの半分のみにアクセスできます。
- 上記のビューのレジスタを組み合わせたもの。

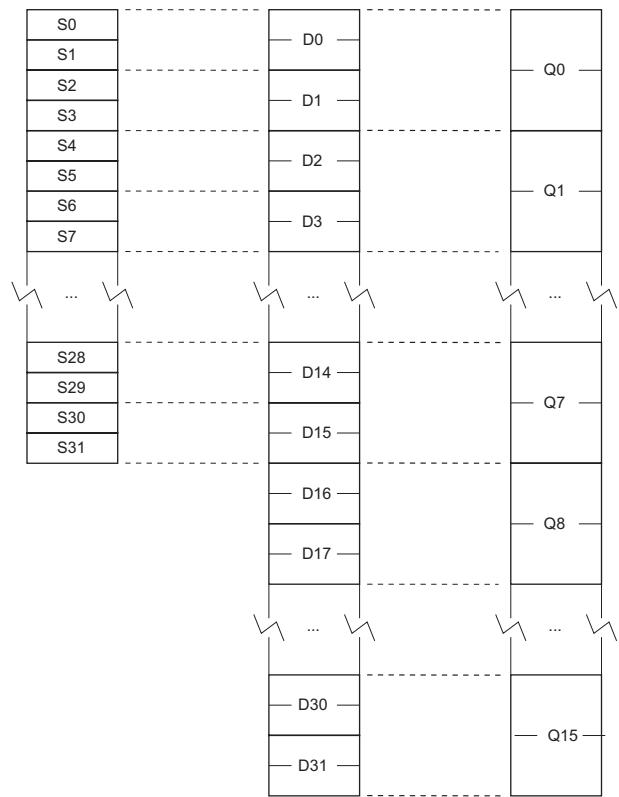


図 5-1 NEON / VFPv3 レジスタバンク

レジスタ間のマッピングは以下のとおりです。

- S_{2n} は D_{n} の最下位ハーフにマップされます。
- S_{2n+1} は D_{n} の最上位ハーフにマップされます。
- D_{2n} は Q_n の最下位ハーフにマップされます。
- D_{2n+1} は Q_n の最上位ハーフにマップされます。

例えば、 Q_6 のベクタの要素の最下位ハーフにアクセスするには D_{12} を参照し、要素の最上位ハーフにアクセスするには D_{13} を参照します。

5.2 条件コード

ARM 状態では、条件コードを使用して VFP 命令の実行を制御できます。VFP 命令は、他のほぼすべての ARM 命令と同じ方法で、CPSR 内のステータスフラグに基づいて条件実行されます。

ARM 状態では、VFP と NEON の両方に共通の命令を除き、条件コードを使用して NEON 命令の実行を制御することはできません。

Thumb-2 プロセッサの Thumb[®] 状態では、IT 命令を使用して次の NEON または VFP 命令の最大 4 つまで条件コードを設定できます。詳細については、「IT」(P. 4-74) を参照して下さい。

ステータスフラグを更新するときに使用できる VFP 命令は FCMP のみです。CPSR のフラグを直接更新するのではなく、FPSCR の別のフラグセットを更新します（「FPSCR：浮動小数点ステータス / 制御レジスタ」(P. 5-87) を参照）。

—— 注 ——

これらのフラグを使用して、条件付き VFP 命令などの条件命令を制御するには、まず FMSTAT 命令を使用して CPSR にこれらのフラグをコピーする必要があります（「MRS および MSR」(P. 5-23) を参照）。

FCMP 命令実行後のフラグは、ARM データ処理命令実行後のフラグと意味が厳密には異なります。その理由は以下のとおりです。

- 浮動小数点値が符号なしになることはないので、符号なし条件は不要です。
- *Not-a-Number* (NaN) 値には数値間または相互の順位関係がないので、不規則な順位の結果を説明するための条件を追加する必要があります。

条件コードニーモニックの意味を P. 5-11 表 5-4 に示します。

表 5-4 条件コード

ニーモニック	ARM データ処理命令後の意味	VFP FCMP 命令後の意味
EQ	等しい	等しい
NE	等しくない	等しくない、または順不同
CS / HS	キャリーセット / 符号なし以上	以上、または順不同
CC / LO	キャリークリア / 符号なし未満	未満
MI	負	未満
PL	正またはゼロ	以上、または順不同
VS	オーバフロー	順不同 (NaN オペランドが含まれている)
VC	オーバフローなし	順番を指定
HI	符号なし超	超、または順不同
LS	符号なし以下	以下
GE	符号付き以上	以上
LT	符号付き未満	未満、または順不同
GT	符号付き超	超
LE	符号付き以下	以下、または順不同
AL	常時 (通常省略)	常時 (通常省略)

注

CPSR のフラグを最後に更新した命令のタイプによって条件コードの意味が決まります。

5.3 一般的な情報

このセクションでは、説明が重複するのを避けるため、多くの命令に共通する情報を示します。このセクションは以下のサブセクションから構成されています。

- 例外
- アーキテクチャのバージョン
- NEON データ型 (P. 5-13)
- NEON の Normal, Long, Wide, Narrow、およびサチュレート命令 (P. 5-14)
- NEON スカラ (P. 5-16)
- {0,1} を超える多項式算術演算 (P. 5-16)

5.3.1 例外

例外が発生する命令の説明には、例外というサブセクションがあります。命令の説明に「例外」サブセクションがない場合は、命令によって例外が発生しません。

5.3.2 アーキテクチャのバージョン

VFP および共通の命令の説明には、命令をサポートするアーキテクチャを示すサブセクションがあります。

NEON 命令は、NEON をサポートするシステムで使用できます。NEON と VFP で共通の命令を除き、NEON 命令は、NEON をサポートするシステム以外のシステムでは使用できません。

5.3.3 NEON データ型

NEON 命令のデータ型指定子は、データ型を示す文字の後に、一般に幅を示す数値が続きます。指定子と命令ニーモニックはポイントで区切られます。

データ型は以下のいずれかになります。

U	符号なし整数
S	符号付き整数
I	型が指定されていない整数
F	浮動小数点数
P	{0,1} を超える多項式

NEON の浮動小数点数は常に 32 ビット幅です。32 はデータ型指定子から省略できます。他のデータ型は 8、16、32、または 64 ビット幅です。

{0,1} を超える多項式演算に関する詳細については、「{0,1} を超える多項式算術演算」(P. 5-16) を参照して下さい。

2 番目（または唯一）のオペランドのデータ型は命令で指定されます。

注

ほとんどの命令では使用可能なデータ型の範囲が制限されています。詳細については、命令のページを参照して下さい。

5.3.4 NEON の Normal、Long、Wide、Narrow、およびサチュレート命令

多くの NEON データ処理命令は、Normal、Long、Wide、Narrow、およびサチュレート バリアントで使用できます。

NEON 命令を実行できる対象は以下のとおりです。

- ダブルワードベクタの構成要素
 - 8 つの 8 ビット要素
 - 4 つの 16 ビット要素
 - 2 つの 32 ビット要素
 - 1 つの 64 ビット要素
- クワッドワードベクタの構成要素
 - 16 の 8 ビット要素
 - 8 つの 16 ビット要素
 - 4 つの 32 ビット要素
 - 2 つの 64 ビット要素

Normal 命令

Normal 命令はこれらのベクタ型に対して演算を実行し、オペランドベクタと同じサイズで、通常は型も同じ結果ベクタを生成します。

Long 命令

Long 命令は、ダブルワードベクタオペランドに対して演算を実行し、クワッドワードベクタ結果を生成します。結果の要素の幅は、通常、オペランドの要素の幅の 2 倍になります、型は同じです。

Long 命令は、命令ニーモニックに L を追加して指定します。

Wide 命令

Wide 命令は、それぞれ 1 つのダブルワードベクタオペランドとクワッドワードベクタオペランドに対して演算を実行します。クワッドワードベクタ結果が生成されます。結果の要素と第 1 オペランドの幅は第 2 オペランドの要素の幅の 2 倍です。

Wide 命令は、命令ニーモニックに W を追加して指定します。

Narrow 命令

Narrow 命令は、クワッドワードベクタオペランドに対して演算を実行し、ダブルワードベクタ結果を生成します。結果の要素の幅は、通常、オペランドの要素の幅の半分です。

Narrow 命令は、命令ニーモニックに N を追加して指定します。

サチュレート命令

サチュレート命令の機能の概要については、「サチュレート命令」(P. 4-99) を参照して下さい。NEON サチュレート命令がサチュレートする範囲については、表 5-5 を参照して下さい。

サチュレート命令は、V と命令ニーモニックの間に接頭文字 Q を使用して指定します。

表 5-5 NEON サチュレーションの範囲

データ型	x のサチュレーション範囲
S8	$-2^7 \leq x < 2^7$
S16	$-2^{15} \leq x < 2^{15}$
S32	$-2^{31} \leq x < 2^{31}$
S64	$-2^{63} \leq x < 2^{63}$
U8	$0 \leq x < 2^8$
U16	$0 \leq x < 2^{16}$
U32	$0 \leq x < 2^{32}$
U64	$0 \leq x < 2^{64}$

5.3.5 NEON スカラ

一部の NEON 命令はベクタと組み合わせてスカラに対して実行されます。NEON スカラは 8 ビット、16 ビット、32 ビット、または 64 ビットです。乗算命令とは違って、スカラにアクセスする命令はレジスタバンク内の要素にアクセスできます。命令構文は、ダブルワードベクタにインデクスを使用してスカラを参照し、 $Dm[x]$ は Dm の x 番目の要素になります。

乗算命令では 16 ビットまたは 32 ビットスカラのみを使用でき、レジスタバンクでアクセスできるのは最初の 32 ビットスカラだけです。つまり、乗算命令では以下のようになります。

- 16 ビットスカラはレジスタ D0 ~ D7 に制限されます (x の範囲は 0 ~ 3)
- 32 ビットスカラはレジスタ D0 ~ D15 に制限されます (x は 0 または 1)。

5.3.6 {0,1} を超える多項式算術演算

係数 0 と 1 は以下のブール算術演算規則を使用して操作します。

- $0 + 0 = 1 + 1 = 0$
- $0 + 1 = 1 + 0 = 1$
- $0 * 0 = 0 * 1 = 1 * 0 = 0$
- $1 * 1 = 1$

つまり、{0,1} を超える 2 つの多項式を加算するとビット単位排他的論理和 (XOR) と同じになり、{0,1} を超える 2 つの多項式を乗算すると、整数の乗算と同じになります。ただし一部の積は加算ではなく排他的論理和の結果です。

5.4 NEON と VFP に共通の命令

このセクションは以下のサブセクションから構成されています。

- *VLDR および VSTR* (P. 5-18)
NEON / VFP レジスタロードとストア
- *VLDM および VSTM* (P. 5-19)
NEON / VFP レジスタ多重ロードとストア
- *VMOV (2 本の ARM レジスタと NEON / VFP の間)* (P. 5-20)
2 本の ARM レジスタと 64 ビット NEON / VFP レジスタとの間で内容を転送します。
- *VMOV (1 本の ARM レジスタと NEON / VFP の間)* (P. 5-21)
ARM レジスタと半分の 4 ビット NEON / VFP レジスタとの間で内容を転送します。
- *VMOV (1 本の ARM レジスタと 単精度 VFP の間)* (P. 5-22)
32 ビット NEON / VFP レジスタと ARM レジスタとの間で内容を転送します。
- *MRS および MSR* (P. 5-23)
ARM レジスタと NEON / VFP システムレジスタとの間で内容を転送します。

5.4.1 VLDR および VSTR

NEON / VFP レジスタロードとストア

構文

`VLDR{cond} Fd, [Rn{, #offset}]`

`VSTR{cond} Fd, [Rn{, #offset}]`

`VLDR{cond} Fd, label`

`VSTR{cond} Fd, label`

各パラメータには以下の意味があります。

cond 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

Fd ロードまたは保存する NEON / VFP レジスタを指定します。NEON 命令では、**Dd** です。VFP 命令では、**Dd** または **Sd** のいずれかです。

Rn 転送用のベースアドレスを保持する ARM レジスタを指定します。

offset 任意の数値式を指定します。アセンブリ時に数値定数が求められる必要があります。値は 4 の倍数で、範囲は -1020 ~ +1020 です。値はベースアドレスに加算され、転送に使用するアドレスを形成します。

label プログラム相対式を指定します。詳細については、「レジスタ相対式とプログラム相対式」（P. 3-31）を参照して下さい。

label は現在の命令から ±1KB 以内に配置する必要があります。

使用法

VLDR 命令は、メモリから NEON / VFP レジスタをロードします。VSTR 命令は、NEON / VFP レジスタの内容をメモリに保存します。

Fd が単精度レジスタの場合は、1 ワードが転送されます（VFP のみ）。それ以外の場合には 2 ワードが転送されます。

VLDR 擬似命令もあります（「VLDR 擬似命令」（P. 5-108）を参照）。

5.4.2 VLDM および VSTM

NEON / VFP レジスタ多重ロードとストア。

構文

`VLDMmode{cond} [Rn],{!}Registers`

`VSTMmode{cond} [Rn],{!}Registers`

各パラメータには以下の意味があります。

<i>mode</i>	以下のいずれかを指定できます。
IA	各転送後にアドレスをインクリメントします。IA がデフォルトで、省略できます。
DB	各転送前にアドレスをデクリメントします。
EA	空上昇スタック演算を意味します。ロードの場合には DB、保存の場合には IA と同じです。
FD	フル下降スタック演算を意味します。ロードの場合には IA、保存の場合には DB と同じです。
<i>cond</i>	任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。
<i>Rn</i>	転送用のベースアドレスを保持する ARM レジスタを指定します。
!	これはオプションです。! は、更新されたベースアドレスを <i>Rn</i> にライトバックする必要があることを示します。

——注——

! が指定されていない場合、*mode* は IA になります。

<i>Registers</i>	連続する NEON / VFP レジスタのリストを中括弧 { および } で囲んで指定します。リストはカンマで区切って指定することも、範囲を指定することもできます。リストには少なくとも 1 本のレジスタを指定する必要があります。
	S、D、または Q レジスタを指定できますが、混在させないで下さい。レジスタの数は D レジスタは 16 本、Q レジスタは 8 本を超えないようにして下さい。Q レジスタが指定されている場合は、逆アセンブリで D レジスタとして表示されます。

——注——

`VLDM sp!,{...}` は `VPOP {...}` として逆アセンブルされます。

`VSTMDB sp!, {...}` は `VPUSH {...}` として逆アセンブルされます。

どちらの形式も使用できます。

5.4.3 VMOV (2 本の ARM レジスタと NEON / VFP の間)

2 本の ARM レジスタと 64 ビット NEON / VFP レジスタ間、または 2 本の連続する 32 ビット VFP レジスタ間で内容を転送します。

構文

`VMOV{cond} Dm, Rd, Rn`

`VMOV{cond} Rd, Rn, Dm`

`VMOV{cond} {Sm, Sm'}, Rd, Rn`

`VMOV{cond} Rd, Rn, {Sm, Sm'}`

各パラメータには以下の意味があります。

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

Dm NEON / VFP 64 ビットレジスタを指定します。

Sm VFP 32 ビットレジスタを指定します。

Sm' *Sm* の後続の VFP 32 ビットレジスタを指定します。

Rd, Rn ARM レジスタです。r15 は使用しないで下さい。

使用法

`VMOV Dm, Rd, Rn` は *Rd* の内容を *Dm* の下位半分に転送し、*Rn* の内容を *Dm* の上位半分に転送します。

`VMOV Rd, Rn, Dm` は *Dm* の下位半分の内容を *Rd* に転送し、*Dm* の上位半分の内容を *Rn* に転送します。

`VMOV Rd, Rn, {Sm, Sm'}` は *Sm* の内容を *Rd* に転送し、*Sm'* の内容を *Rn* に転送します。

`VMOV {Sm, Sm'}, Rd, Rn` は *Rd* の内容を *Sm* に転送し、*Rn* の内容を *Sm'* に転送します。

例外

これらの命令は、例外を生成しません。

アーキテクチャ

64 ビット命令は、NEON、VFPv2、および上記で使用できます。

2 x 32 ビット命令は、VFPv2 および上記で使用できます。

5.4.4 VMOV (1 本の ARM レジスタと NEON / VFP の間)

ARM レジスタと半分の倍精度浮動小数点レジスタ間で内容を転送します。

構文

`VMOV{cond} Dn[x], Rd`

`VMOV{cond} Rd, Dn[x]`

各パラメータには以下の意味があります。

cond 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

Dn[x] VFP 倍精度レジスタの必要な半分を指定します。x は 0 または 1 です。

Rd ARM レジスタを指定します。Rd を r15 にしないで下さい。

使用法

これらの命令は通常、対で使用されます。

例外

これらの命令は例外を生成しません。

例

```
VMOV    d5[1], r3
VMOV    d5[0], r12
```

```
VMOV    r5, d3[0]
VMOV    r9, d3[1]
```

```
VMOVEQ   d2[0], r1
VMOVEQ   d2[1], r0
```

5.4.5 VMOV (1 本の ARM レジスタと単精度 VFP の間)

単精度浮動小数点レジスタと ARM レジスタ間で内容を転送します。

構文

`VMOV{cond} Rd, Sn`

`VMOV{cond} Sn, Rd`

各パラメータには以下の意味があります。

cond 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

Sn VFP 単精度レジスタを指定します。

Rd ARM レジスタを指定します。*Rd* を r15 にしないで下さい。

使用法

`VMOV Rd, Sn` は *Sn* の内容を *Rd* に転送します。

`VMOV Sn, Rd` は *Rd* の内容を *Sn* に転送します。

例外

これらの命令は例外を生成しません。

5.4.6 MRS および MSR

ARM レジスタと NEON / VFP ステータスレジスタ間で内容を転送します。

構文

`MRS{cond} Rd, VFPsysreg`

`MSR{cond} VFPsysreg, Rd`

各パラメータには以下の意味があります。

cond 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

VFPsysreg VFP システムレジスタを指定します。通常 FPSCR、FPSID、または FPEXC です（「VFP レジスタ」（P. 5-82）を参照）。

Rd ARM レジスタを指定します。*Rd* を r15 にしないで下さい。

VFPsysreg が FPSCR の場合は、APSR_nzcv を指定できます。この場合、浮動小数点ステータスフラグが ARM CPSR の対応するフラグに転送されます。

使用法

MRS 命令は *VFPsysreg* の内容を *Rd* に転送します。

MSR 命令は *Rd* の内容を *VFPsysreg* に転送します。

注

これらの命令は、現在の NEON / VFP 演算がすべて完了するまで ARM を停止させます。

例外

これらの命令は例外を生成しません。

5.5 NEON 論理演算と比較演算

このセクションは以下のサブセクションから構成されています。

- *VAND*、*VBIC*、*VEOR*、*VORN*、および*VORR*（レジスタ）(P. 5-25)
ビット単位論理積、ビットクリア、排他的論理和（XOR）、否定論理和、および論理和（レジスタ）
- *VBIC* および*VORR*（イミディエート）(P. 5-26)
ビット単位ビットクリアと論理和（イミディエート）
- *VBIF*、*VBIT*、*VBSL* (P. 5-27)
False の場合はビット単位挿入、True の場合はビット単位を挿入、および選択
- *VNOT* (P. 5-28)
ビット単位否定
- *VACGE* および *VACGT* (P. 5-29)
絶対値の比較
- *VCEQ*、*VCGE*、*VCGT*、*VCLE*、および *VCLT* (P. 5-30)
比較
- *VTST* (P. 5-31)
テストビット

5.5.1 VAND、VBIC、VEOR、VORN、および VORR（レジスタ）

VAND（ビット単位論理積）、VBIC（ビットクリア）、VEOR（ビット単位排他的論理和）、VORN（ビット単位否定論理和）、および VORR（ビット単位論理和）の各命令は、2 本のレジスタ間でビット単位論理演算を実行して、デスティネーションレジスタに結果を返します。

構文

$Vop\{cond\} Qd, Qn, Qm$

$Vop\{cond\} Dd, Dn, Dm$

各パラメータには以下の意味があります。

<i>op</i>	以下のいずれかを指定できます。
AND	論理積
ORR	論理和
EOR	排他的論理和 (XOR)
BIC	論理積補数
ORN	論理和補数
<i>cond</i>	任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。
<i>Qd, Qn, Qm</i>	クワッドワード演算で使用するデスティネーションレジスタ、第 1 オペランドレジスタ、および第 2 オペランドレジスタを指定します。
<i>Dd, Dn, Dm</i>	ダブルワード演算で使用するデスティネーションレジスタ、第 1 オペランドレジスタ、および第 2 オペランドレジスタを指定します。

— 注 —

両方のオペランドに同じレジスタを使用する VORR は VMOV 命令です。VORR はこのように使用できますが、結果として生じたコードを逆アセンブルすると、VMOV 構文が生成されます。詳細については、「VMOV（レジスタ）」(P. 5-37) を参照して下さい。

5.5.2 VBIC および VORR (イミディエート)

VBIC (ビットクリアイミディエート) はデスティネーションベクタの各要素を取得し、イミディエート定数を使用してビット単位論理積補数を求め、デスティネーションベクタに結果を返します。

VORR (ビット単位論理和イミディエート) は、デスティネーションベクタの各要素を取得し、イミディエート定数を使用してビット単位論理和を実行し、デスティネーションベクタに結果を返します。

擬似命令 */VAND および VORN (イミディエート)* (P. 5-78) も参照して下さい。

構文

Vop{cond}.datatype Qd, #imm

Vop{cond}.datatype Dd, #imm

各パラメータには以下の意味があります。

op BIC または ORR を指定します。

cond 任意の条件コードを指定します (*「条件コード」* (P. 5-10) を参照)。

datatype I16 または I32 を指定します。

Qd または *Dd* ソースと結果の NEON レジスタを指定します。

imm イミディエート定数を指定します。

イミディエート定数

datatype が I16 の場合、イミディエート定数は以下のいずれかの形式に対応する必要があります。

- 0x00XY
- 0xXY00

datatype が I32 の場合、イミディエート定数は以下のいずれかの形式に対応する必要があります。

- 0x000000XY
- 0x0000XY00
- 0x00XY0000
- 0xXY000000

5.5.3 VBIF、VBIT、VBSL

VBIT (True の場合はビット単位を挿入) は、第 2 オペランドの対応するビットが 1 の場合は第 1 オペランドからデスティネーションに各ビットを挿入します。それ以外の場合は、デスティネーションビットを変更しません。

VBIF (False の場合はビット単位を挿入) は、第 2 オペランドの対応するビットが 0 の場合は、第 1 オペランドからデスティネーションに各ビットを挿入します。それ以外の場合は、デスティネーションビットを変更しません。

VBSL (ビット単位を選択) は、デスティネーションの対応するビットが 1 の場合は第 1 オペランドから、0 の場合は第 2 オペランドからデスティネーションの各ビットを選択します。

構文

Vop{cond} Qd, Qn, Qm

Vop{cond} Dd, Dn, Dm

各パラメータには以下の意味があります。

op BIT、BIF、またはBSL のいずれかを指定します。

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

Qd, Qn, Qm クワッドワード演算で使用するデスティネーションレジスタ、第 1 オペランドレジスタ、および第 2 オペランドレジスタを指定します。

Dd, Dn, Dm ダブルワード演算で使用するデスティネーションレジスタ、第 1 オペランドレジスタ、第 2 オペランドレジスタを指定します。

5.5.4 VNOT

VNOT（ベクタビット単位反転）はレジスタから値を取得し、各ビットの値を反転して、結果をデスティネーションレジスタに返します。

構文

`VNOT{cond} Qd, Qm`

`VNOT{cond} Dd, Dm`

各パラメータには以下の意味があります。

cond 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

Qd, Qm クワッドワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。

Dd, Dm ダブルワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。

5.5.5 VACGE および VACGT

ベクタ絶対値比較はベクタの各要素の絶対値を取得し、2番目のベクタの対応する要素の絶対値と比較します。条件が True の場合は、デスティネーションベクタの対応する要素はすべて 1 に設定されます。それ以外の場合は、すべて 0 に設定されます。

擬似命令 */VACLE および VACLT* (P. 5-79) も参照して下さい。

構文

VACOp{cond}.datatype Qd, Qn, Qm

VACOp{cond}.datatype Dd, Dn, Dm

各パラメータには以下の意味があります。

op 以下のいずれかを指定します。

GE 以上（絶対値）

GT 超（絶対値比較）

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

datatype F32 です。結果のデータ型は I32 です。

Qd, Qn, Qm クワッドワード演算で使用するデスティネーションレジスタ、第 1 オペランドレジスタ、および第 2 オペランドレジスタを指定します。

Dd, Dn, Dm ダブルワード演算で使用するデスティネーションレジスタ、第 1 オペランドレジスタ、第 2 オペランドレジスタを指定します。

5.5.6 VCEQ、VCGE、VCGT、VCLE、および VCLT

ベクタ比較はベクタの各要素の値を取得し、2番目のベクタの対応する要素の値または 0 と比較します。条件が True の場合は、デスティネーションベクタの対応する要素はすべて 1 に設定されます。それ以外の場合は、すべて 0 に設定されます。

擬似命令 */VCLE および VCLT/* (P. 5-80) も参照して下さい。

構文

VCop{cond}.datatype Qd, Qn, Qm

VCop{cond}.datatype Dd, Dn, Dm

VCop{cond}.datatype Qd, Qn, #0

VCop{cond}.datatype Dd, Dn, #0

各パラメータには以下の意味があります。

<i>op</i>	以下のいずれかを指定します。
EQ	等しい
GE	以上
GT	超
LE	以下 (第 2 オペランドが #0 の場合のみ)
LT	未満 (第 2 オペランドが #0 の場合のみ)
<i>cond</i>	任意の条件コードを指定します ('条件コード' (P. 5-10) を参照)。
<i>datatype</i>	以下のいずれかを指定します。 <ul style="list-style-type: none"> • EQ には I8、I16、I32、または F32 • GE、GT、LE、または LT には S8、S16、S32、U8、U16、U32、F32 (#0 形式を除く) • GE、GT、LE、または LT には S8、S16、S32、または F32 (#0 形式) 結果のデータ型は以下のとおりです。 <ul style="list-style-type: none"> • オペランドデータ型 I32、S32、U32、または F32 には I32 • オペランドデータ型 I16、S16、または U16 には I16 • オペランドデータ型 I8、S8、または U8 には I8
<i>Qd, Qn, Qm</i>	クワッドワード演算で使用するデスティネーションレジスタ、第 1 オペランドレジスタ、第 2 オペランドレジスタを指定します。
<i>Dd, Dn, Dm</i>	ダブルワード演算で使用するデスティネーションレジスタ、第 1 オペランドレジスタ、第 2 オペランドレジスタを指定します。
#0	比較のために <i>Qm</i> または <i>Dm</i> をゼロと置き換えます。

5.5.7 VTST

VTST（ベクタテストビット）は、ベクタの各要素を取得し、2番目のベクタの対応する要素を使用してビット単位論理積を求めます。結果がゼロでない場合は、デスティネーションベクタの対応する要素はすべて 1 に設定されます。それ以外の場合は、すべて 0 に設定されます。

構文

`VTST{cond}.size Qd, Qn, Qm`

`VTST{cond}.size Dd, Dn, Dm`

各パラメータには以下の意味があります。

cond 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

size 8、16、または 32 のいずれかです。

Qd, Qn, Qm クワッドワード演算で使用するデスティネーションレジスタ、第 1 オペランドレジスタ、第 2 オペランドレジスタを指定します。

Dd, Dn, Dm ダブルワード演算で使用するデスティネーションレジスタ、第 1 オペランドレジスタ、第 2 オペランドレジスタを指定します。

5.6 NEON 汎用データ処理命令

このセクションは以下のサブセクションから構成されています。

- *VCVT* (P. 5-33)
固定小数点または整数と浮動小数点間のベクタ変換
- *VDUP* (P. 5-34)
ベクタの全レーンへのスカラの複製
- *VEXT* (P. 5-35)
抽出
- *VMOV*, *VMVN* (イミディエート) (P. 5-36)
移動と負の移動 (イミディエート)
- *VMOVL*, *V{Q}MOVN*, *VQMOVUN* (P. 5-38)
移動 (レジスタ)
- *VREV* (P. 5-39)
ベクタ内の要素の反転
- *VSWP* (P. 5-40)
ベクタのスワップ
- *VTBL*, *VTBX* (P. 5-41)
ベクタテーブルの検索
- *VTRN* (P. 5-42)
ベクタ置換
- *VUZP*, *VZIP* (P. 5-43)
ベクタのインターリープとインターリープの解除

5.6.1 VCVT

VCVT (ベクタ変換) はベクタの各要素を以下のいずれかの方法で変換し、結果を 2 番目のベクタに返します。

- 浮動小数点から整数への変換
- 整数から浮動小数点への変換
- 浮動小数点から固定小数点への変換
- 固定小数点から浮動小数点への変換

構文

`VCVT{cond}.type Qd, Qm {, #fbits}`

`VCVT{cond}.type Dd, Dm {, #fbits}`

各パラメータには以下の意味があります。

<code>cond</code>	任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。
<code>type</code>	ベクタの要素のデータ型を指定します。以下のいずれかになります。
	S32.F32 浮動小数点から符号付き整数または固定小数点への変換
	U32.F32 浮動小数点から符号なし整数または固定小数点への変換
	F32.S32 符号付き整数または固定小数点から浮動小数点への変換
	F32.U32 符号なし整数または固定小数点から浮動小数点への変換
<code>Qd, Qm</code>	クワッドワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。
<code>Dd, Dm</code>	ダブルワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。
<code>fbits</code>	このパラメータが指定されている場合は、固定小数点数の小数部ビットを指定します。それ以外の場合は、浮動小数点と整数間の変換になります。 <code>fbits</code> は 0 ~ 32 の範囲です。

——注——

`fbits==0` は `fbits` を省略することと等しくなります。`fbits` は値が 0 になると、逆アセンブル時に省略されます。

5.6.2 VDUP

ベクタ複製は、デスティネーションベクタのすべての要素にスカラを複製します。ソースは NEON スカラまたは ARM レジスタです。

構文

`VDUP{cond}.size Qd, Dm[x]`

`VDUP{cond}.size Dd, Dm[x]`

`VDUP{cond}.size Qd, Rm`

`VDUP{cond}.size Dd, Rm`

各パラメータには以下の意味があります。

`cond` 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

`size` 8、16、または 32 にします。

`Qd` クワッドワード演算で使用するデスティネーションレジスタを指定します。

`Dd` ダブルワード演算で使用するデスティネーションレジスタを指定します。

`Dm[x]` NEON スカラを指定します。

`Rm` ARM レジスタを指定します。

5.6.3 VEXT

ベクタ抽出は、第2オペランドベクタの下位と第1オペランドベクタの上位から8ビット要素を抽出し、連結して、デスティネーションベクタに結果を返します。この例について、図 5-2 を参照して下さい。

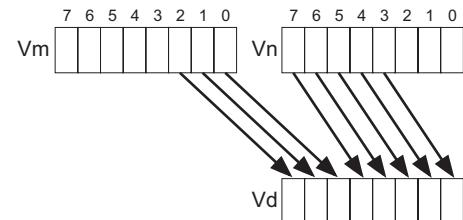


図 5-2 imm = 3 の場合のダブルワード VEXT 演算

構文

`VEXT{cond}.8 Qd, Qn, Qm, #imm`

`VEXT{cond}.8 Dd, Dn, Dm, #imm`

各パラメータには以下の意味があります。

cond 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

Qd, Qn, Qm クワッドワード演算で使用するデスティネーションレジスタ、第1オペランドレジスタ、第2オペランドレジスタを指定します。

Dd, Dn, Dm ダブルワード演算で使用するデスティネーションレジスタ、第1オペランドレジスタ、第2オペランドレジスタを指定します。

imm 第2オペランドベクタの下位から抽出する8ビット要素の数を指定します。ダブルワード演算では0～7の範囲、クワッドワード演算では0～15の範囲です。

VEXT 擬似命令

8の代わりに16、32、または64のデータ型を指定できます。この場合、#*imm*にバイトの代わりにハーフワード、ワード、またはダブルワードを使用すると、それに応じて許容範囲が小さくなります。

5.6.4 VMOV、VMVN (イミディエート)

ベクタ移動 (イミディエート) はイミディエート定数をデスティネーションレジスタに配置します。

ベクタの負の移動 (イミディエート) でもイミディエート定数がデスティネーションレジスタに配置されます。

構文

`Vop{cond}.datatype Qd, #imm`

`Vop{cond}.datatype Dd, #imm`

各パラメータには以下の意味があります。

op MOV または MVN を指定します。

cond 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

datatype I8、I16、I32、I64、または F32 を指定します。

Qd または *Dd* 結果の NEON レジスタを指定します。

imm *datatype* によって指定される型の定数です。複製されてデスティネーションレジスタに配置されます。

表 5-6 使用可能な定数

データ型	VMOV	VMVN
I8	0xXY	-
I16	0x00XY、0xXY00	0xFFXY、0xXYFF
I32	0x000000XY、0x0000XY00、0x00XY0000、 0xXY000000	0xFFFFFFFXY、0xFFFFXYFF、0xFFXYFFFF、 0xXYFFFFFF
	0x0000XYFF、0x00XYFFFF	0xFFFFXY00、0xFFXY0000
I64	バイトマスク、0xGGHHJJKKLLMMNNPP ^a	-
F32	浮動小数点数 ^b	-

a. 0xGG、0xHH、0xJJ、0xKK、0xLL、0xMM、0xNN、0xPP を 0x00 または 0xFF にします。

b. $+/-n * 2^{-r}$ と表すことができる任意の数値。ここで、*n* と *r* は整数、 $16 \leq n \leq 31$ 、 $0 \leq r \leq 7$ です。

5.6.5 VMOV (レジスタ)

ベクタ移動 (レジスタ) は、ソース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。

構文

`VMOV{cond} Qd, Qm`

`VMOV{cond} Dd, Dm`

各パラメータには以下の意味があります。

cond 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

Qd, Qm クワッドワード演算で使用するデスティネーションベクタとソースベクタを指定します。

Dd, Dm ダブルワード演算で使用するデスティネーションベクタとソースベクタを指定します。

5.6.6 VMOVL, V{Q}MOVN, VQMOVUN

VMOVL (ベクタ Long 移動) はダブルワードベクタの各要素を取得し、符号拡張またはゼロ拡張を行って元の長さの 2 倍にして、クワッドワードベクタに結果を返します。

VMOVN (ベクタ移動および Narrow) は、クワッドワードベクタの各要素の下位半分をダブルワードベクタの対応する要素にコピーします。

VQMOVN (ベクタサチュレート移動および Narrow) は、オペランドベクタの各要素をデスティネーションベクタの対応する要素にコピーします。結果の要素の幅はオペランド要素の半分で、値が結果の幅にサチュレートされます。

VQMOVUN (ベクタサチュレート移動および Narrow、符号付きオペランドと符号のない結果) は、オペランドベクタの各要素をデスティネーションベクタの対応する要素にコピーします。結果の要素の幅はオペランド要素の半分で、値は結果の幅にサチュレートされます。

構文

`VMOVL{cond}.datatype Qd, Dm`

`V{Q}MOVN{cond}.datatype Dd, Qm`

`VQMOVUN{cond}.datatype Dd, Qm`

各パラメータには以下の意味があります。

`Q` このパラメータが指定されている場合は、結果がサチュレートされます。

`cond` 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

`datatype` 以下のいずれかを指定します。

S8、S16、S32 VMOVL の場合

U8、U16、U62 VMOVL の場合

I16、I32、I64 VMOVN の場合

S16、S32、S64 VQMOVN または VQMOVUN の場合

U16、U32、U64 VQMOVN の場合。

`Qd, Dm` VMOVL のデスティネーションベクタとオペランドベクタを指定します。

`Dd, Qm` V{Q}MOV{U}N のデスティネーションベクタとオペランドベクタを指定します。

5.6.7 VREV

VREV16 (ハーフワード内のベクタ反転) は、ベクタの各ハーフワード内の 8 ビット要素の順序を逆にして、対応するデスティネーションベクタに結果を返します。

VORR (ワード内のベクタ反転) は、ベクタの各ワード内の 8 ビットまたは 16 ビット要素の順序を逆にして、対応するデスティネーションベクタに結果を返します。

VREV64 (ダブルワード内のベクタ反転) は、ベクタの各ダブルワード内の 8 ビット、16 ビット、または 32 ビット要素の順序を逆にして、対応するデスティネーションベクタに結果を返します。

構文

VREV*n{cond}.size Qd, Qm*

VREV*n{cond}.size Dd, Dm*

各パラメータには以下の意味があります。

n 16、32、または 64 のいずれかにします。

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

size 8、16、または 32 のいずれかを指定し、*n* 未満にします。

Qd, Qm クワッドワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。

Dd, Dm ダブルワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。

5.6.8 VSWP

VZIP (ベクタスワップ) は 2 つのベクタの内容を交換します。ベクタはダブルワードまたはクワッドワードのいずれかです。データ型は区別されません。

構文

`VSWP{cond} Qd, Qm`

`VSWP{cond} Dd, Dm`

各パラメータには以下の意味があります。

cond 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

Qd, Qm クワッドワード演算で使用するベクタを指定します。

Dd, Dm ダブルワード演算で使用するベクタを指定します。

5.6.9 VTBL, VTBX

VTBL (ベクターテーブル検索) は、コントロールベクタのバイトインデックスを使用し、テーブル内のバイト値を検索して新しいベクタを生成します。範囲外のインデックスは 0 を返します。

VTBX (ベクターテーブル拡張) は、同様に動作しますが、範囲外のインデックスがデスティネーション要素を変更することはありません。

構文

Vop{cond} Dd, list, Dm

各パラメータには以下の意味があります。

op TBL または TBX にします。

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

Dd デスティネーションベクタを指定します。

list テーブルを含むベクタを指定します。以下のいずれかになります。

- {*Dn*}
- {*Dn, D(n+1)*}
- {*Dn, D(n+1), D(n+2)*}
- {*Dn, D(n+1), D(n+2), D(n+3)*}.

list のすべてのレジスタは D0 ~ D31 の範囲です。

Dm インデクスベクタを指定します。

5.6.10 VTRN

VTRN (ベクタ置換) は、オペランドベクタの要素を 2×2 マトリクスの要素として処理し、マトリクスを置換します。図 5-3 と図 5-4 は VTRN の演算の例を示します。

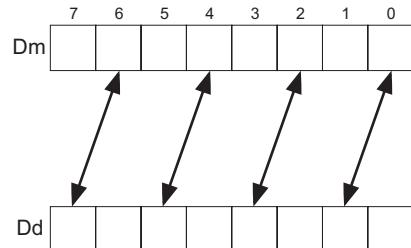


図 5-3 ダブルワード VTRN.8 演算

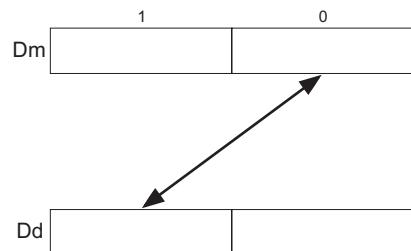


図 5-4 ダブルワード VTRN.32 演算

構文

`VTRN{cond}.size Qd, Qm`

`VTRN{cond}.size Dd, Dm`

各パラメータには以下の意味があります。

`cond` 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

`size` 8、16、または 32 のいずれかです。

`Qd, Qm` クワッドワード演算で使用するベクタを指定します。

`Dd, Dm` ダブルワード演算で使用するベクタを指定します。

5.6.11 VUZP、VZIP

VZIP (ベクタ圧縮) は 2 つのベクタの要素をインターリープします。

VUZP (ベクタ解凍) は 2 つのベクタの要素のインターリープを解除します。

インターリープ解除の例については、「*3 要素構造体から成る配列のインターリープの解除*」(P. 5-69) を参照して下さい。インターリープはこの逆のプロセスです。

構文

Vop{cond}.size Qd, Qm

Vop{cond}.size Dd, Dm

各パラメータには以下の意味があります。

op UZP または ZIP にします。

cond 任意の条件コードを指定します ('条件コード' (P. 5-10) を参照)。

size 8、16、または 32 のいずれかです。

Qd, Qm クワッドワード演算で使用するベクタを指定します。

Dd, Dm ダブルワード演算で使用するベクタを指定します。

注

以下はすべて同じ命令です。

- VZIP.32 *Dd, Dm*
- VUZP.32 *Dd, Dm*
- VTRN.32 *Dd, Dm*

命令は VTRN.32 *Dd, Dm* として逆アセンブルされます。

5.7 NEON シフト命令

このセクションは以下のサブセクションから構成されています。

- $VSHL$, $VQSHL$, $VQSHLU$ 、および $VSHLL$ (イミディエートによる) (P. 5-45)
イミディエート値による左シフト
- $V\{Q\}\{R\}SHL$ (符号付き変数による) (P. 5-47)
符号付き変数による左シフト
- $V\{R\}SHR\{N\}$, $V\{R\}SRA$ (イミディエートによる) (P. 5-48)
イミディエート値による右シフト
- $VQ\{R\}SHR\{U\}N$ (イミディエートによる) (P. 5-49)
イミディエート値による右シフトとサチュレート
- $VSLI$ および $VSRI$ (P. 5-50)
左シフトと挿入および右シフトと挿入

5.7.1 VSHL、VQSHL、VQSHLU、および VSHLL (イミディエートによる)

ベクタ左シフト、ベクタサチュレート左シフト、およびベクタ左シフト Long

これらの命令は、整数ベクタの各要素を取得し、イミディエート値で左にシフトして、デスティネーションベクタに結果を返します。

VSHL では、各要素の左の範囲外にシフトされたビットは失われます。

VQSHL と VQSHLU では、サチュレーションが発生するとステイッキー QC フラグ (FPSCR bit[27]) が設定されます。

VSHLL では、値は符号拡張またはゼロ拡張されます。

構文

`V{Q}SHL{U}{cond}.datatype Qd, Qm, #imm`

`V{Q}SHL{U}{cond}.datatype Dd, Dm, #imm`

`VSHLL{cond}.datatype Qd, Dm, #imm`

各パラメータには以下の意味があります。

`Q` このパラメータが指定されている場合、オーバフローした結果はサチュレートされます。

`U` `Q` も指定されている場合にのみ使用できます。オペランドに符号が付いても結果には符号が付きません。

`cond` 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

`datatype` 以下のいずれかを指定します。

`I8、I16、I32、I64` VSHL の場合

`S8、S16、S32` VSHLL、VQSHL、または VQSHLU の場合

`U8、U16、U32` VSHLL または VQSHL の場合

`S64` VQSHL または VQSHLU の場合

`U64` VQSHL の場合。

<i>Qd, Qm</i>	クワッドワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。
<i>Dd, Dm</i>	ダブルワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。
<i>Qd, Dm</i>	long 演算で使用するデスティネーションベクタとオペランドベクタを指定します。
<i>imm</i>	以下の範囲でシフトのサイズを指定するイミディエート定数です。 <ul style="list-style-type: none">• VSHLL では 1 ~ size(<i>datatype</i>)• VSHL、VQSHL、または VQSHLU では 1 ~ (size(<i>datatype</i>) - 1) 0 を使用できますが、生成されるコードは VMOV または VMOVL に逆アセンブリされます。

5.7.2 V{Q}{R}SHL (符号付き変数による)

VSHL (符号付き変数によるベクタ左シフト) は、ベクタの各要素を取得し、2番目のベクタの対応する要素の最下位バイトの値でシフトし、デスティネーションベクタに結果を返します。シフト値が正の場合は、左シフトになります。それ以外の場合は、右シフトになります。

結果はオプションでサチュレート、丸め、またはその両方を実行できます。サチュレーションが発生した場合はステイッキー QC フラグ (FPSCR bit[27]) が設定されます。

構文

`V{Q}{R}SHL{cond}.datatype Qd, Qn, Qm`

`V{Q}{R}SHL{cond}.datatype Dd, Dn, Dm`

各パラメータには以下の意味があります。

`Q` このパラメータが指定されている場合、オーバフローした結果はサチュレートされます。

`R` このパラメータが指定されている場合は、結果が丸められます。それ以外の場合、結果は切り捨てられます。

`cond` 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

`datatype` S8、S16、S32、S64、U8、U16、U32、または U64 のいずれかにします。

`Qd, Qn, Qm` クワッドワード演算で使用するデスティネーションベクタ、第1オペランドベクタ、および第2オペランドベクタを指定します。

`Dd, Dn, Dm` ダブルワード演算で使用するデスティネーションベクタ、第1オペランドベクタ、第2オペランドベクタを指定します。

5.7.3 V{R}SHR{N}, V{R}SRA (イミディエートによる)

V{R}SHR{N} (イミディエート値によるベクタ右シフト) は、ベクタの各要素を取得し、イミディエート値で右シフトして、デスティネーションベクタに結果を返します。結果はオプションで丸め、切り捨て、またはその両方を実行できます。

V{R}SRA (イミディエート値によるベクタ右シフトと累積) は、ベクタの各要素を取得し、イミディエート値で右シフトして、デスティネーションベクタに結果を累積します。結果はオプションで丸めることができます。

構文

```
V{R}SHR{cond}.datatype Qd, Qm, #imm
V{R}SHR{cond}.datatype Dd, Dm, #imm
V{R}SRA{cond}.datatype Qd, Qm, #imm
V{R}SRA{cond}.datatype Dd, Dm, #imm
V{R}SHRN{cond}.datatype Dd, Qm, #imm
```

各パラメータには以下の意味があります。

<i>R</i>	このパラメータが指定されている場合は、結果が丸められます。それ以外の場合、結果は切り捨てられます。						
<i>cond</i>	任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。						
<i>datatype</i>	以下のいずれかを指定します。 <table border="0"> <tr> <td>S8, S16, S32, S64</td> <td>V{R}SHR または V{R}SRA の場合</td> </tr> <tr> <td>U8, U16, U32, U64</td> <td>V{R}SHR または V{R}SRA の場合</td> </tr> <tr> <td>I16, I32, I64</td> <td>V{R}SHRN の場合</td> </tr> </table>	S8, S16, S32, S64	V{R}SHR または V{R}SRA の場合	U8, U16, U32, U64	V{R}SHR または V{R}SRA の場合	I16, I32, I64	V{R}SHRN の場合
S8, S16, S32, S64	V{R}SHR または V{R}SRA の場合						
U8, U16, U32, U64	V{R}SHR または V{R}SRA の場合						
I16, I32, I64	V{R}SHRN の場合						
<i>Qd, Qm</i>	クワッドワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。						
<i>Dd, Dm</i>	ダブルワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。						
<i>Dd, Qm</i>	Narrow 演算で使用するデスティネーションベクタとオペランドベクタを指定します。						
<i>imm</i>	0 ~ (size(<i>datatype</i>) - 1) の範囲でシフトのサイズを指定するイミディエート定数です。						

5.7.4 VQ{R}SHR{U}N (イミディエートによる)

VQ{R}SHR{U}N (ベクタサチュレート右シフト、Narrow、イミディエート値による、オプションの丸め) は、整数のクワッドワードベクタの各要素を取得し、イミディエート値で右シフトし、ダブルワードベクタに結果を返します。

サチュレーションが発生した場合はスティッキー QC フラグ (FPSCR bit[27]) が設定されます。

構文

`VQ{R}SHR{U}N{cond}.datatype Dd, Qm, #imm`

各パラメータには以下の意味があります。

R このパラメータが指定されている場合は、結果が丸められます。それ以外の場合、結果は切り捨てられます。

U このパラメータが指定されている場合は、オペランドに符号が付いていても結果には符号が付きません。それ以外の場合は、結果はオペランドと同じ型になります。

cond 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

datatype 以下のいずれかを指定します。

S16、S32、S64 VQ{R}SHRN または VQ{R}SHRUN の場合

U16、U32、U64 VQ{R}SHRN のみの場合

Dd, Qm デスティネーションベクタとオペランドベクタを指定します。

imm 0 ~ (size(datatype) – 1) の範囲でシフトのサイズを指定するイミディエート定数です。

5.7.5 VSLI および VSRI

VSLI (ベクタ左シフトと挿入) は、ベクタの各要素を取得し、イミディエート値で左シフトして、デスティネーションベクタに結果を挿入します。各要素の左の範囲外にシフトされたビットは失われます。

VSRI (ベクタ右シフトと挿入) は、ベクタの各要素を取得し、イミディエート値で右シフトして、デスティネーションベクタに結果を挿入します。各要素の右の範囲外にシフトされたビットは失われます。

構文

`\op{cond}.size Qd, Qm, #imm`

`\op{cond}.size Dd, Dm, #imm`

各パラメータには以下の意味があります。

`op` SLI または SRI にします。

`cond` 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

`size` 8、16、32、または 64 のいずれかにします。

`Qd, Qm` クワッドワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。

`Dd, Dm` ダブルワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。

`imm` 以下の範囲でシフトのサイズを指定するイミディエート定数です。

- VSLI では $0 \sim (\text{size}-1)$
- VSRI では $1 \sim \text{size}$

5.8 NEON 汎用算術命令

このセクションは以下のサブセクションから構成されています。

- $VABA\{L\}$ および $VABD\{L\}$ (P. 5-52)
ベクタ絶対差と累積、絶対差
- $V\{Q\}ABS$ および $V\{Q\}NEG$ (P. 5-53)
ベクタ絶対値および否定
- $V\{Q\}ADD$ 、 $VADDL$ 、 $VADDW$ 、 $V\{Q\}SUB$ 、 $VSUBL$ 、 $VSUBW$ (P. 5-54)
ベクタ加算および減算
- $V\{R\}ADDHN$ および $V\{R\}SUBHN$ (P. 5-55)
上位半分を選択するベクタ加算および減算
- $V\{R\}HADD$ および $VHSUB$ (P. 5-56)
ベクタ二分加算および減算
- $VPADD\{L\}$ 、 $VPADAL$ (P. 5-57)
ベクタペアワイズ加算および加算累積
- $VMAX$ 、 $VMIN$ 、 $VPMAX$ 、 $VPMIN$ (P. 5-59)
ベクタ最大値、最小値、ペアワイズ最大値、およびペアワイズ最小値
- $VCLS$ 、 $VCLZ$ 、 $VCNT$ (P. 5-60)
ベクタ先行符号ビットカウント、先行ゼロカウント、およびセットビットカウント
- $VRECPE$ および $VRSQRTE$ (P. 5-61)
ベクタ逆数の推定および逆平方根の推定
- $VRECPS$ および $VRSQRTS$ (P. 5-62)
ベクタ逆数のステップおよび逆平方根のステップ

5.8.1 VABA{L} および VABD{L}

VABA (ベクタ絶対差と累積) は、あるベクタの要素を対応する別のベクタの要素から減算し、その結果の絶対値をデスティネーションベクタの要素に累積します。

VABD (ベクタ絶対差) は、あるベクタの要素を対応する別のベクタの要素から減算し、その結果の絶対値をデスティネーションベクタの要素に返します。

いずれの命令にも Long バージョンがあります。

構文

`Vop{cond}.datatype Qd, Qn, Qm`

`Vop{cond}.datatype Dd, Dn, Dm`

`VopL{cond}.datatype Qd, Dn, Dm`

各パラメータには以下の意味があります。

op ABA または ABD にします。

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

datatype 以下のいずれかを指定します。

- VABA、VABAL、または VABDL の場合、S8、S16、S32、U8、U16、または U32
- VABD の場合、S8、S16、S32、U8、U16、U32、または F32

Qd, Qn, Qm クワッドワード演算で使用するデスティネーションベクタ、第 1 オペランドベクタ、および第 2 オペランドベクタを指定します。

Dd, Dn, Dm ダブルワード演算で使用するデスティネーションベクタ、第 1 オペランドベクタ、および第 2 オペランドベクタを指定します。

Qd, Dn, Dm Long 演算で使用するデスティネーションベクタ、第 1 オペランドベクタ、および第 2 オペランドベクタを指定します。

5.8.2 V{Q}ABS および V{Q}NEG

VABS (ベクタ絶対値) は、ベクタの各要素の絶対値を取得して、その結果を 2 番目のベクタに返します (浮動小数点バージョンは、符号ビットをクリアするだけです)。

VNEG (ベクタ否定) は、ベクタの各要素を否定して、その結果を 2 番目のベクタに返します (浮動小数点バージョンは、符号ビットを反転するだけです)。

いずれの命令にもサチュレートバージョンがあります。サチュレーションが発生した場合は、ステイッキー QC フラグ (FPSCR bit[27]) が設定されます。

構文

$V\{Q\}op\{cond\}.datatype\ Qd, Qm$

$V\{Q\}op\{cond\}.datatype\ Dd, Dm$

各パラメータには以下の意味があります。

Q このパラメータが指定されている場合、オーバーフローした結果はサチュレートされます。

op ABS または NEG にします。

cond 任意の条件コードを指定します ('条件コード' (P. 5-10) を参照)。

datatype 以下のいずれかを指定します。

S8、S16、S32 VABS、VNEG、VQABS、または VQNEG の場合

F32 VABS および VNEG の場合のみ

Qd, Qm クワッドワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。

Dd, Dm ダブルワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。

5.8.3 V{Q}ADD、VADDL、VADDW、V{Q}SUB、VSUBL、VSUBW

VADD (ベクタ加算) は、2つのベクタのそれぞれ対応する要素を加算し、その結果をデスティネーションベクタに返します。

VSUB (ベクタ減算) は、あるベクタの要素を対応する別のベクタの要素から減算し、その結果をデスティネーションベクタに返します。

いずれの命令にもサチュレートバージョン、Long バージョン、および Wide バージョンがあります。サチュレーションが発生した場合は、ステイッキー QC フラグ (FPSCR bit[27]) が設定されます。

構文

$V\{Q\}op\{cond\}.datatype\ Qd, Qn, Qm$

$V\{Q\}op\{cond\}.datatype\ Dd, Dn, Dm$

$VopL\{cond\}.datatype\ Qd, Dn, Dm$

$VopW\{cond\}.datatype\ Qd, Qn, Dm$

各パラメータには以下の意味があります。

Q このパラメータが指定されている場合、オーバーフローした結果はサチュレートされます。

op ADD または SUB にします。

$cond$ 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

$datatype$ 以下のいずれかを指定します。

I8、I16、I32、I64、F32 VADD または VSUB の場合

S8、S16、S32 VQADD、VQSUB、VADDL、VADDW、VSUBL、または VSUBW の場合

U8、U16、U32 VQADD、VQSUB、VADDL、VADDW、VSUBL、または VSUBW の場合

S64、U64 VQADD または VQSUB の場合

Qd, Qn, Qm クワッドワード演算で使用するデスティネーションベクタ、第 1 オペランドベクタ、および第 2 オペランドベクタを指定します。

Dd, Dn, Dm ダブルワード演算で使用するデスティネーションベクタ、第 1 オペランドベクタ、および第 2 オペランドベクタを指定します。

Qd, Dn, Dm Long 演算で使用するデスティネーションベクタ、第 1 オペランドベクタ、および第 2 オペランドベクタを指定します。

Qd, Qn, Dm Wide 演算で使用するデスティネーションベクタ、第 1 オペランドベクタ、および第 2 オペランドベクタを指定します。

5.8.4 V{R}ADDHN および V{R}SUBHN

V{R}ADDH (上位半分を選択するベクタ加算および Narrow) は、2つのベクタのそれぞれ対応する要素を加算し、その結果の上位半分を選択して、最終的な結果をデスティネーションベクタに返します。結果は丸めるか、切り捨てるすることができます。

V{R}SUBH (上位半分を選択するベクタ減算および Narrow) は、あるベクタの要素を対応する別のベクタの要素から減算し、その結果の上位半分を選択して、最終的な結果をデスティネーションベクタに返します。結果は丸めるか、切り捨てるすることができます。

構文

V{R}opHN{cond}.datatype Dd, Qn, Qm

各パラメータには以下の意味があります。

R このパラメータが指定されている場合、結果は丸められます。指定されていない場合、結果は切り捨てられます。

op ADD または SUB にします。

cond 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

datatype I16、I32、または I64 のいずれかにします。

Dd, Qn, Qm デスティネーションベクタ、第1オペランドベクタ、および第2オペランドベクタを指定します。

5.8.5 V{R}HADD および VHSUB

VHADD（ベクタ二分加算）は、2つのベクタのそれぞれ対応する要素を加算し、結果を1ビット右にシフトして、その結果をデスティネーションベクタに返します。結果は丸めるか、切り捨てるすることができます。

VHSUB（ベクタ二分減算）は、あるベクタの要素を対応する別のベクタの要素から減算し、結果を1ビット右にシフトして、その結果をデスティネーションベクタに返します。結果は常に切り捨てられます。

構文

`V{R}HADD{cond}.datatype Qd, Qn, Qm`

`V{R}HADD{cond}.datatype Dd, Dn, Dm`

`VHSUB{cond}.datatype Qd, Qn, Qm`

`VHSUB{cond}.datatype Dd, Dn, Dm`

各パラメータには以下の意味があります。

R このパラメータが指定されている場合、結果は丸められます。指定されていない場合、結果は切り捨てられます。

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

datatype S8、S16、S32、U8、U16、または U32 のいずれかにします。

Qd, Qn, Qm クワッドワード演算で使用するデスティネーションベクタ、第1オペランドベクタ、および第2オペランドベクタを指定します。

Dd, Dn, Dm ダブルワード演算で使用するデスティネーションベクタ、第1オペランドベクタ、および第2オペランドベクタを指定します。

5.8.6 VPADD{L}、VPADAL

VPADD（ベクタペアワイズ加算）は、2つのベクタの隣接する要素のペアを加算し、その結果をデスティネーションベクタに返します。

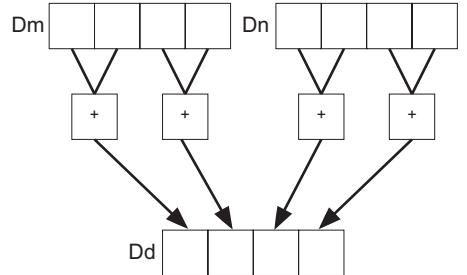


図 5-5 VPADD 演算の例（データ型 I16 の場合）

VPADDL（ベクタペアワイズ加算 Long）は、あるベクタ、符号、またはゼロの隣接する要素のペアを加算し、結果を元の幅の2倍にして、最終的な結果をデスティネーションベクタに返します。

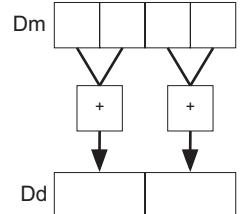


図 5-6 ダブルワード VPADDL 演算の例（データ型 S16 の場合）

VPADAL（ベクタペアワイズ加算累積 Long）は、あるベクタの隣接する要素のペアを加算し、その結果の絶対値をデスティネーションベクタの要素に累積します。

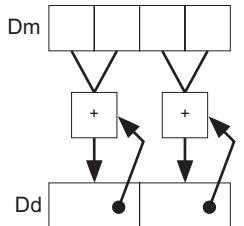


図 5-7 VPADAL 演算の例（データ型 S16 の場合）

構文

`VPADD{cond}.datatype Dd, Dn, Dm`

`VPopL{cond}.datatype Qd, Qm`

`VPopL{cond}.datatype Dd, Dm`

各パラメータには以下の意味があります。

op ADD または ADA にします。

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

datatype 以下のいずれかを指定します。

I8、I16、I32、F32 VPADD の場合

S8、S16、S32 VPADDL または VPADAL の場合

U8、U16、U32 VPADDL または VPADAL の場合

Dd, Dn, Dm VPADD 命令で使用するデスティネーションベクタ、第 1 オペランドベクタ、および第 2 オペランドベクタを指定します。

Qd, Qm クワッドワードの VPADDL または VPADAL で使用するデスティネーションベクタとオペランドベクタを指定します。

Dd, Dm ダブルワードの VPADDL または VPADAL で使用するデスティネーションベクタとオペランドベクタを指定します。

5.8.7 VMAX、VMIN、VPMAX、VPMIN

VMAX (ベクタ最大値) は、2つのベクタのそれぞれ対応する要素を比較し、各ペアの大きい方の要素をデスティネーションベクタの対応する要素にコピーします。

VMIN (ベクタ最小値) は、2つのベクタのそれぞれ対応する要素を比較し、各ペアの小さい方の要素をデスティネーションベクタの対応する要素にコピーします。

VPMAX (ベクタペアワイズ最大値) は、2つのベクタの隣接する要素のペアを比較し、各ペアの大きい方の要素をデスティネーションダブルワードベクタの対応する要素にコピーします。

VPMIN (ベクタペアワイズ最小値) は、2つのベクタの隣接する要素のペアを比較し、各ペアの小さい方の要素をデスティネーションダブルワードベクタの対応する要素にコピーします。

ペアワイズ演算の図については、P. 5-57 図 5-5 を参照して下さい。

—————注—————

ペアワイズ最大値とペアワイズ最小値の演算対象は、ダブルワードベクタのみです。

構文

Vop{cond}.datatype Qd, Qn, Qm

V{P}op{cond}.datatype Dd, Dn, Dm

各パラメータには以下の意味があります。

op MAX または MIN にします。

cond 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

datatype S8、S16、S32、U8、U16、U32、または F32 のいずれかにします。

Qd, Qn, Qm クワッドワード演算で使用するデスティネーションベクタ、第 1 オペランドベクタ、および第 2 オペランドベクタを指定します。

Dd, Dn, Dm ダブルワード演算で使用するデスティネーションベクタ、第 1 オペランドベクタ、および第 2 オペランドベクタを指定します。

浮動小数点の最大値と最小値

$$\max(+0.0, -0.0) = +0.0$$

$$\min(+0.0, -0.0) = -0.0$$

いずれかの入力が NaN の場合、対応する結果要素はデフォルトの NaN になります。

5.8.8 VCLS、VCLZ、VCNT

VCLS (ベクタ先行符号ビットカウント) は、あるベクタの各要素に含まれる最上位ビットに続く連続ビット (最上位ビットと同じ) の数をカウントし、その結果を 2 番目のベクタに返します。

VCLS (ベクタ先行ゼロカウント) は、あるベクタの各要素に含まれる連続するゼロの数をカウントし (最上位ビットから開始)、その結果を 2 番目のベクタに返します。

VCNT (ベクタセットビットカウント) は、あるベクタの各要素に含まれるビットの数をカウントし、その結果を 2 番目のベクタに返します。

構文

$\text{Vop}\{\text{cond}\}.\text{datatype } Qd, Qm$

$\text{Vop}\{\text{cond}\}.\text{datatype } Dd, Dm$

各パラメータには以下の意味があります。

op CLS、CLZ、または CNT のいずれかにします。

cond 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

datatype 以下のいずれかを指定します。

- CLS の場合、S8、S16、または S32
- CLZ の場合、I8、I16、または I32
- CNT の場合、I8

Qd, Qm クワッドワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。

Dd, Dm ダブルワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。

5.8.9 VRECPE および VRSQRTE

VRECPE（ベクタ逆数の推定）は、あるベクタに含まれる各要素の逆数の近似値を見つけ、その結果を 2 番目のベクタに返します。

VRSQRTE（ベクタ逆平方根の推定）は、あるベクタに含まれる各要素の逆平方根の近似値を見つけ、その結果を 2 番目のベクタに返します。

構文

Vop{cond}.datatype Qd, Qm

Vop{cond}.datatype Dd, Dm

各パラメータには以下の意味があります。

op RECPE または RSQRTE にします。

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

datatype U32 または F32 にします。

Qd, Qm クワッドワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。

Dd, Dm ダブルワード演算で使用するデスティネーションベクタとオペランドベクタを指定します。

有効範囲外の入力を指定した場合の結果

表 5-7 に、有効範囲外の入力値を指定した場合の結果を示します。

表 5-7 有効範囲外の入力を指定した場合の結果

オペランド要素 (VRECPE)	オペランド要素 (VRSQRTE)	結果要素
整数 $\leq 0x7FFFFFFF$	$\leq 0x3FFFFFFF$	$0xFFFFFFFF$
浮動小数点 NaN	NaN、負の正規、負の無限大	デフォルトの NaN
負の 0、負の非正規	負の 0、負の非正規	負の無限大 ^a
正の 0、正の非正規	正の 0、正の非正規	正の無限大 ^a
正の無限大	正の無限大	正の 0
負の無限大		負の 0

a. FPSCR (FPSCR[1]) では、ゼロによる除算の例外ビットが設定されます。

5.8.10 VRECPS および VRSQRTS

VRECPS (ベクタ逆数のステップ) は、あるベクタの要素を対応する別のベクタの要素で乗算し、その結果を 2 から減算して、最終的な結果をデスティネーションベクタの要素に返します。

VRSQRTS (ベクタ逆平方根のステップ) は、あるベクタの要素を対応する別のベクタの要素で乗算し、その結果を 3 から減算して 2 で除算し、最終的な結果をデスティネーションベクタの要素に返します。

構文

$\text{Vop}\{\text{cond}\}.\text{F32 } Qd, Qn, Qm$

$\text{Vop}\{\text{cond}\}.\text{F32 } Dd, Dn, Dm$

各パラメータには以下の意味があります。

op RECPS または RSQRTS にします。

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

Qd, Qn, Qm クワッドワード演算で使用するデスティネーションベクタ、第 1 オペランドベクタ、および第 2 オペランドベクタを指定します。

Dd, Dn, Dm ダブルワード演算で使用するデスティネーションベクタ、第 1 オペランドベクタ、および第 2 オペランドベクタを指定します。

有効範囲外の入力を指定した場合の結果

表 5-8 に、有効範囲外の入力値を指定した場合の結果を示します。

表 5-8 有効範囲外の入力を指定した場合の結果

第 1 オペランド要素	第 2 オペランド要素	結果要素 (VRECPS)	結果要素 (VRSQRTS)
NaN	-	デフォルトの NaN	デフォルトの NaN
-	NaN	デフォルトの NaN	デフォルトの NaN
+/- 0.0 または非正規	+/- 無限大	2.0	1.5
+/- 無限大	+/- 0.0 または非正規	2.0	1.5

使用法

ニュートンラプソン法による反復計算：

$$x_{n+1} = x_n(2-dx_n)$$

VRECPE を d に適用した結果が x_0 である場合に $(1/d)$ に収束します。

ニュートンラプソン法による反復計算：

$$x_{n+1} = x_n(3-dx_n^2)/2$$

VRSQRTE を d に適用した結果が x_0 である場合に $(1/\sqrt{d})$ に収束します。

5.9 NEON 乗算命令

このセクションは以下のサブセクションから構成されています。

- $VMUL\{L\}$ 、 $VMLA\{L\}$ 、 $VMLS\{L\}$ (P. 5-65)
ベクタ乗算、積和、および積差
- $VMUL\{L\}$ 、 $VMLA\{L\}$ 、 $VMLS\{L\}$ (スカラによる) (P. 5-66)
ベクタ乗算、積和、および積差 (スカラによる)
- $VQDMULL$ 、 $VQDMLAL$ 、 $VQDMLSL$ (ベクタまたはスカラによる) (P. 5-67)
ベクタサチュレートダブル乗算、積和、および積差 (ベクタまたはスカラによる)
- $VQ\{R\}DMULH$ (ベクタまたはスカラによる) (P. 5-68)
上位半分を返すベクタサチュレートダブル乗算 (ベクタまたはスカラによる)

5.9.1 VMUL{L}、VMLA{L}、VMLS{L}

VMUL (ベクタ乗算) は、2つのベクタのそれぞれ対応する要素を乗算し、その結果をデスティネーションベクタに返します。

VMLA (ベクタ積和) は、2つのベクタのそれぞれ対応する要素を乗算し、その結果をデスティネーションベクタの要素に累積します。

VMLS (ベクタ積差) は、2つのベクタのそれぞれ対応する要素を乗算し、その結果をデスティネーションベクタの対応する要素から減算して、最終的な結果をデスティネーションベクタに返します。

構文

Vop{cond}.datatype Qd, Qn, Qm

Vop{cond}.datatype Dd, Dn, Dm

VopL{cond}.datatype Qd, Dn, Dm

各パラメータには以下の意味があります。

<i>op</i>	以下のいずれかを指定します。
MUL	乗算
MLA	積和
MLS	積差
<i>cond</i>	任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。
<i>datatype</i>	以下のいずれかを指定します。
I8、I16、I32、F32	MUL、MLA、または MLS の場合
S8、S16、S32	MULL、MLAL、または MLSL の場合
U8、U16、U32	MULL、MLAL、または MLSL の場合
P8	MUL または MULL の場合
データ型 P8 については、「{0,1} を超える多項式算術演算」(P. 5-16) を参照して下さい。	
<i>Qd, Qn, Qm</i>	クワッドワード演算で使用するデスティネーションベクタ、第1オペランドベクタ、および第2オペランドベクタを指定します。
<i>Dd, Dn, Dm</i>	ダブルワード演算で使用するデスティネーションベクタ、第1オペランドベクタ、および第2オペランドベクタを指定します。
<i>Qd, Dn, Dm</i>	Long 演算で使用するデスティネーションベクタ、第1オペランドベクタ、および第2オペランドベクタを指定します。

5.9.2 VMUL{L}、VMLA{L}、VMLS{L} (スカラによる)

VMUL (スカラによるベクタ乗算) は、あるベクタの各要素をスカラで乗算し、その結果をデスティネーションベクタに返します。

VMLA (ベクタ積和) は、2つのベクタのそれぞれ対応する要素を乗算し、その結果をデスティネーションベクタの要素に累積します。

VMLS (ベクタ積差) は、2つのベクタのそれぞれ対応する要素を乗算し、その結果をデスティネーションベクタの対応する要素から減算して、最終的な結果をデスティネーションベクタに返します。

構文

`Vop{cond}.datatype Qd, Qn, Dm[x]`

`Vop{cond}.datatype Dd, Dn, Dm[x]`

`VopL{cond}.datatype Qd, Dn, Dm[x]`

各パラメータには以下の意味があります。

<i>op</i>	以下のいずれかを指定します。
MUL	乗算
MLA	積和
MLS	積差
<i>cond</i>	任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。
<i>datatype</i>	以下のいずれかを指定します。
I16、I32、F32	MUL、MLA、または MLS の場合
S16、S32	MULL、MLAL、または MSL 的場合
U16、U32	MULL、MLAL、または MSL 的場合
<i>Qd, Qn</i>	クワッドワード演算で使用するデスティネーションベクタと第1オペランドベクタを指定します。
<i>Dd, Dn</i>	ダブルワード演算で使用するデスティネーションベクタと第1オペランドベクタを指定します。
<i>Qd, Dn</i>	Long 演算で使用するデスティネーションベクタと第1オペランドベクタを指定します。
<i>Dm[x]</i>	第2オペランドを保持するスカラです。

5.9.3 VQDMULL、VQDMLAL、VQDMLSL（ベクタまたはスカラによる）

ベクタサチュレートダブル乗算命令は、オペランドを乗算し、その結果を 2 倍にします。VQDMULL は、結果をデスティネーションレジスタに返します。VQDMLAL は、結果をデスティネーションレジスタの値に加算します。VQDMLSL は、結果をデスティネーションレジスタの値から減算します。

オーバーフローした結果はサチュレートされます。サチュレーションが発生した場合は、ステッキ QC フラグ (FPSCR bit[27]) が設定されます。

構文

`VQDopL{cond}.datatype Qd, Dn, Dm`

`VQDopL{cond}.datatype Qd, Dn, Dm[x]`

各パラメータには以下の意味があります。

op 以下のいずれかを指定します。

MUL	乗算
MLA	積和
MLS	積差

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

datatype S16 または S32 にします。

Qd, Dn デスティネーションベクタと第 1 オペランドベクタを指定します。

Dm ベクタによる演算の第 2 オペランドを保持するベクタです。

Dm[x] スカラによる演算の第 2 オペランドを保持するスカラです。

5.9.4 VQ{R}DMULH (ベクタまたはスカラによる)

ベクタサチュレートダブル乗算命令は、オペランドを乗算し、その結果を 2 倍にします。結果の上位半分しか返されません。

オーバーフローした結果はサチュレートされます。サチュレーションが発生した場合は、ステイッキー QC フラグ (FPSCR bit[27]) が設定されます。

構文

```
VQ{R}DMULH{cond}.datatype Qd, Qn, Qm
VQ{R}DMULH{cond}.datatype Dd, Dn, Dm
VQ{R}DMULH{cond}.datatype Qd, Qn, Dm[x]
VQ{R}DMULH{cond}.datatype Dd, Dn, Dm[x]
```

各パラメータには以下の意味があります。

R このパラメータが指定されている場合、結果は丸められます。指定されていない場合、結果は切り捨てられます。

cond 任意の条件コードを指定します ('条件コード' (P. 5-10) を参照)。

datatype S16 または S32 にします。

Qd, Qn クワッドワード演算で使用するデスティネーションベクタと第 1 オペランドベクタを指定します。

Dd, Dn ダブルワード演算で使用するデスティネーションベクタと第 1 オペランドベクタを指定します。

Qm または Dm ベクタによる演算の第 2 オペランドを保持するベクタです。

Dm[x] スカラによる演算の第 2 オペランドを保持するスカラです。

5.10 NEON 要素と構造体のロード / ストア命令

このセクションは以下のサブセクションから構成されています。

- インターリープ。
- 要素と構造体のロード / ストア命令における境界調整の制約 (P. 5-70)。
- $VLDn$ および $VSTn$ (1 レーンへの 1 つの n 要素構造体) (P. 5-71)。
これは、ほとんどすべてのデータアクセスで使用されます。正規ベクタをロードできます ($n = 1$)。
- $VLDn$ (全レーンへの 1 つの n 要素構造体) (P. 5-73)。
- $VLDn$ および $VSTn$ (複数の n 要素構造体) (P. 5-75)。

5.10.1 インターリープ

このグループの命令の多くは、構造体をメモリに保存するときにインターリープを可能にし、構造体をメモリからロードするときにインターリープの解除を可能にします。図 5-8 に、インターリープの解除の例を示します。インターリープはこの逆のプロセスです。

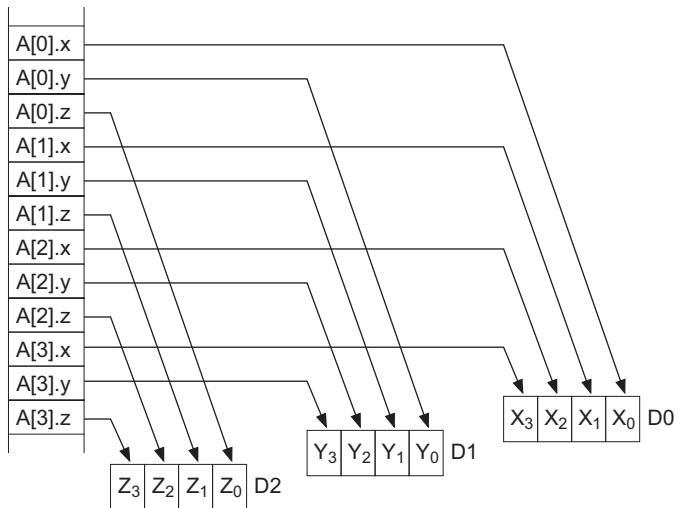


図 5-8 3 要素構造体から成る配列のインターリープの解除

5.10.2 要素と構造体のロード / ストア命令における境界調整の制約

通常、これらの命令を使用すると、メモリ境界調整の制約を指定できます。命令で境界調整が指定されていない場合、境界調整の制約は、次のように A ビット (CP15 レジスター 1 ビット [1]) によって制御されます。

- A ビットが 0 の場合、境界調整の制約はありません（ただし、厳密に順序を指定されたメモリやデバイスマモリなど、アクセスが整列要素である場合を除きます。この場合は予測しない結果が返されます）。
- A ビットが 1 の場合、アクセスは整列要素である必要があります。

アドレスが正しく整列されていないと、境界調整エラーが発生します。

5.10.3 VLD n および VST n (1 レーンへの 1 つの n 要素構造体)

1 レーンへの 1 つの n 要素構造体のベクタロードを実行すると、1 つの n 要素構造体がメモリから NEON レジスタにロードされます。ロードされなかったレジスタの要素は変更されません。

構文

$Vopn\{cond\}.datatype\ list, [Rn@\align]\{!\}$

$Vopn\{cond\}.datatype\ list, [Rn@\align], Rm$

各パラメータには以下の意味があります。

op LD または ST にします。

n 1、2、3、または 4 のいずれかにします。

$cond$ 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

$datatype$ P. 5-72 表 5-9 を参照して下さい。

$list$ NEON レジスタリストを指定します。オプションについては、P. 5-72 表 5-9 を参照して下さい。

Rn ベースアドレスを保持する ARM レジスタです。 Rn を R15 にすることはできません。

$align$ オプションの境界調整を指定します。オプションについては、P. 5-72 表 5-9 を参照して下さい。

! ! が指定されている場合、 Rn は (Rn + 命令によって転送されるバイト数) に更新されます。

Rm ベースアドレスからのオフセットを保持する ARM レジスタです。 Rm が指定されている場合、メモリにアクセスするためにアドレスが使用された後で、 Rn は ($Rn + Rm$) に更新されます。 Rm を R13 または R15 にすることはできません。

表 5-9 パラメータの有効な組み合わせ

<i>n</i>	datatype	list^a	align^b	alignment
1	8	{Dd[x]}	-	標準のみ
	16	{Dd[x]}	@16	2 バイト
	32	{Dd[x]}	@32	4 バイト
2	8	{Dd[x], D(d+1)[x]}	@16	2 バイト
	16	{Dd[x], D(d+1)[x]}	@32	4 バイト
		{Dd[x], D(d+2)[x]}	@32	4 バイト
	32	{Dd[x], D(d+1)[x]}	@64	8 バイト
		{Dd[x], D(d+2)[x]}	@64	8 バイト
3	8、16、または 32	{Dd[x], D(d+1)[x], D(d+2)[x]}	-	標準のみ
		{Dd[x], D(d+2)[x], D(d+4)[x]}	-	標準のみ
4	8	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@32	4 バイト
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@32	4 バイト
	16	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@64	8 バイト
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@64	8 バイト
32		{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@64 または @128	8 バイトまたは 16 バイト
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@64 または @128	8 バイトまたは 16 バイト

a. 表内のすべてのレジスタは D0 ~ D31 の範囲内にある必要があります。

b. *align* は省略できます。省略した場合は、標準の境界調整ルールが適用されます。「要素と構造体のロード / ストア命令における境界調整の制約」(P. 5-70) を参照して下さい。

5.10.4 VLDn (全レーンへの 1 つの n 要素構造体)

全レーンへの 1 つの n 要素構造体のベクタロードを実行すると、1 つの n 要素構造体の複数のコピーがメモリから NEON レジスタにロードされます。

構文

`VLDn{cond}.datatype list, [Rn{@align}]{!}`

`VLDn{cond}.datatype list, [Rn{@align}], Rm`

各パラメータには以下の意味があります。

n 1、2、3、または 4 のいずれかにします。

$cond$ 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

$datatype$ P. 5-74 表 5-10 を参照して下さい。

$list$ NEON レジスタリストを指定します。オプションについては、P. 5-74 表 5-10 を参照して下さい。

Rn ベースアドレスを保持する ARM レジスタです。 Rn を R15 にすることはできません。

$align$ オプションの境界調整を指定します。オプションについては、P. 5-74 表 5-10 を参照して下さい。

! !が指定されている場合、 Rn は (Rn + 命令によって転送されるバイト数) に更新されます。

Rm ベースアドレスからのオフセットを保持する ARM レジスタです。 Rm が指定されている場合、メモリにアクセスするためにアドレスが使用された後で、 Rn は ($Rn + Rm$) に更新されます。 Rm を R13 または R15 にすることはできません。

表 5-10 パラメータの有効な組み合わせ

<i>n</i>	<i>datatype</i>	<i>list^a</i>	<i>align^b</i>	<i>alignment</i>
1	8	{Dd[]}	-	標準のみ
		{Dd[], D(d+1)[]}	-	標準のみ
16		{Dd[]}	@16	2 バイト
		{Dd[], D(d+1)[]}	@16	2 バイト
32		{Dd[]}	@32	4 バイト
		{Dd[], D(d+1)[]}	@32	4 バイト
2	8	{Dd[], D(d+1)[]}	@8	バイト
		{Dd[], D(d+2)[]}	@8	バイト
16		{Dd[], D(d+1)[]}	@16	2 バイト
		{Dd[], D(d+2)[]}	@16	2 バイト
32		{Dd[], D(d+1)[]}	@32	4 バイト
		{Dd[], D(d+2)[]}	@32	4 バイト
3	8、16、または 32	{Dd[], D(d+1)[], D(d+2)[]}	-	標準のみ
		{Dd[], D(d+2)[], D(d+4)[]}	-	標準のみ
4	8	{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}	@32	4 バイト
		{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}	@32	4 バイト
16		{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}	@64	8 バイト
		{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}	@64	8 バイト
32		{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}	@64 または @128	8 バイトまたは 16 バイト
		{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}	@64 または @128	8 バイトまたは 16 バイト

a. 表内のすべてのレジスタは D0 ~ D31 の範囲内にある必要があります。

b. *align* は省略できます。省略した場合は、標準の境界調整ルールが適用されます。「要素と構造体のロード/ストア命令における境界調整の制約」(P. 5-70) を参照して下さい。

5.10.5 VLDn および VSTn (複数の n 要素構造体)

複数の n 要素構造体のベクタロードを実行すると、複数の n 要素構造体がメモリから NEON レジスタにロードされ、インターリーブが解除されます ($n = 1$ の場合を除きます)。各レジスタのすべての要素がロードされます。

複数の n 要素構造体のベクタストアを実行すると、複数の n 要素構造体が NEON レジスタからメモリに保存され、インターリーブされます ($n = 1$ の場合を除きます)。各レジスタのすべての要素が保存されます。

構文

Vopn{cond}.datatype list, [Rn{@align}]{}!

Vopn{cond}.datatype list, [Rn{@align}], Rm

各パラメータには以下の意味があります。

op LD または ST にします。

n 1、2、3、または4のいずれかにします。

cond 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

datatype オプションについては、P. 5-76 表 5-11 を参照して下さい。

list NEON レジスタリストを指定します。オプションについては、P. 5-76 表 5-11 を参照して下さい。

Rn ベースアドレスを保持する ARM レジスタです。*Rn* を R15 にすることはできません。

align オプションの境界調整を指定します。オプションについては、P. 5-76 表 5-11 を参照して下さい。

! ! が指定されている場合、*Rn* は (*Rn* + 命令) によって転送されるバイト数) に更新されます。

Rm ベースアドレスからのオフセットを保持する ARM レジスタです。*Rm* が指定されている場合、メモリにアクセスするためにアドレスが使用された後で、*Rn* は (*Rn* + *Rm*) に更新されます。*Rm* を R13 または R15 にすることはできません。

表 5-11 パラメータの有効な組み合わせ

<i>n</i>	<i>datatype</i>	<i>list^a</i>	<i>align^b</i>	<i>alignment</i>
1	8、16、32、または 64	{Dd}	@64	8 バイト
		{Dd, D(d+1)}	@64 または @128	8 バイトまたは 16 バイト
		{Dd, D(d+1), D(d+2)}	@64	8 バイト
		{Dd, D(d+1), D(d+2), D(d+3)}	@64、@128、または @256	8 バイト、16 バイト、または 32 バイト
2	8、16、または 32	{Dd, D(d+1)}	@64 または @128	8 バイトまたは 16 バイト
		{Dd, D(d+2)}	@64 または @128	8 バイトまたは 16 バイト
		{Dd, D(d+1), D(d+2), D(d+3)}	@64、@128、または @256	8 バイト、16 バイト、または 32 バイト
3	8、16、または 32	{Dd, D(d+1), D(d+2)}	@64	8 バイト
		{Dd, D(d+2), D(d+4)}	@64	8 バイト
4	8、16、または 32	{Dd, D(d+1), D(d+2), D(d+3)}	@64、@128、または @256	8 バイト、16 バイト、または 32 バイト
		{Dd, D(d+2), D(d+4), D(d+6)}	@64、@128、または @256	8 バイト、16 バイト、または 32 バイト

a. 表内のすべてのレジスタは D0 ~ D31 の範囲内にある必要があります。

b. *align* は省略できます。省略した場合は、標準の境界調整ルールが適用されます。「要素と構造体のロード/ストア命令における境界調整の制約」(P. 5-70) を参照して下さい。

5.11 NEON 擬似命令

このセクションは以下のサブセクションから構成されています。

- *VAND* および *VORN* (イミディエート) (P. 5-78)
- *VACLE* および *VACLT* (P. 5-79)
- *VCLE* および *VCLT* (P. 5-80)

5.11.1 VAND および VORN (イミディエート)

VAND (ビット単位論理積イミディエート) は、デスティネーションベクタの各要素を取得し、イミディエート定数を使用してビット単位論理積を実行し、デスティネーションベクタに結果を返します。

VORN (ビット単位否定論理和イミディエート) は、デスティネーションベクタの各要素を取得し、イミディエート定数を使用してビット単位論理和補数を求め、デスティネーションベクタに結果を返します。

——注——

逆アセンブル時に、これらの擬似命令は、相補イミディエート定数を使用して対応する VBIC 命令と VORR 命令に逆アセンブルされます。

構文

`Vop{cond}.datatype Qd, #imm`

`Vop{cond}.datatype Dd, #imm`

各パラメータには以下の意味があります。

op VAND または VORN にします。

cond 任意の条件コードを指定します (「条件コード」(P. 5-10) を参照)。

datatype I16 または I32 にします。

Qd または *Dd* 結果を保持する NEON レジスタです。

imm イミディエート定数です。

イミディエート定数

datatype が I16 の場合、イミディエート定数は以下のいずれかの形式に対応する必要があります。

- 0xFFXY
- 0xXYFF。

datatype が I32 の場合、イミディエート定数は以下のいずれかの形式に対応する必要があります。

- 0xFFFFFFFFXY
- 0xFFFFXYFF
- 0xFFXYFFFF
- 0xXYFFFFFF。

5.11.2 VACLE および VACLT

ベクタ絶対値比較は、ベクタの各要素の絶対値を取得し、2番目のベクタの対応する要素の絶対値と比較します。条件が True の場合、デスティネーションベクタの対応する要素はすべて 1 に設定されます。それ以外の場合は、すべて 0 に設定されます。

——注——

逆アセンブル時に、これらの擬似命令は、反転したオペランドを使用して対応する VACGT 命令と VACGE 命令に逆アセンブルされます。

構文

`VACOp{cond}.datatype Qd, Qn, Qm`

`VACOp{cond}.datatype Dd, Dn, Dm`

各パラメータには以下の意味があります。

op 以下のいずれかを指定します。

LE 以下（絶対値）

LT 未満（絶対値）

cond 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

datatype F32 にする必要があります。

Qd または *Dd* 結果を保持する NEON レジスタです。

結果の *datatype* は I32 です。

Qn または *Dn* 第 1 オペランドを保持する NEON レジスタです。

Qm または *Dm* 第 2 オペランドを保持する NEON レジスタです。

5.11.3 VCLE および VCLT

ベクタ比較はベクタの各要素の値を取得し、2番目のベクタの対応する要素の値または 0 と比較します。条件が True の場合、デスティネーションベクタの対応する要素はすべて 1 に設定されます。それ以外の場合は、すべて 0 に設定されます。

—————注—————

逆アセンブル時に、これらの擬似命令は、反転したオペランドを使用して対応する VCGT 命令と VCGE 命令に逆アセンブルされます。

構文

`VCop{cond}.datatype Qd, Qn, Qm`

`VCop{cond}.datatype Dd, Dn, Dm`

各パラメータには以下の意味があります。

op 以下のいずれかを指定します。

LE 以下

LT 未満

cond 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

datatype S8、S16、S32、U8、U16、U32、または F32 のいずれかにします。

Qd または *Dd* 結果を保持する NEON レジスタです。

結果の *datatype* は次のようにになります。

- I32 (オペランドのデータ型が I32、S32、U32、または F32 の場合)
- I16 (オペランドのデータ型が I16、S16、または U16 の場合)
- I8 (オペランドのデータ型が I8、S8、または U8 の場合)

Qn または *Dn* 第 1 オペランドを保持する NEON レジスタです。

Qm または *Dm* 第 2 オペランドを保持する NEON レジスタです。

5.12 ベクタ浮動小数点コプロセッサ

ベクタ浮動小数点 (VFP) コプロセッサと、そのサポートコードを使用することにより、ANSI/IEEE 規格 754-1985 「IEEE Standard for Binary Floating-Point Arithmetic (二進浮動小数点演算に関する IEEE 規格)」で定義されているように、単精度および倍精度の浮動小数点演算を実行できます。本章では、この規格を「IEEE 754 規格」と呼びます。この規格の概要については、*RealView Compilation Tools v3.0 Compiler and Libraries Guide* の浮動小数点に関する章を参照して下さい。

最大 8 個の単精度数値または最大 4 個 (VFPv3 では 8 個) の倍精度数値から成るショートベクタは、特に効率的に処理されます。これらのベクタでは大半の算術命令を使用でき、1 つの命令で複数のデータを並列に処理 (SIMD) できます。また、浮動小数点ロード / ストア命令には複数のレジスタ形式があり、ベクタをメモリとの間で効率的に転送できます。

VFP コプロセッサの詳細については、ARM アーキテクチャリファレンスマニュアルを参照して下さい。

5.13 VFP レジスタ

VFP コプロセッサには、s0 ~ s31 の 32 本の単精度レジスタがあります。各レジスタは、単精度浮動小数点値または 32 ビット整数を保持できます。

これら 32 本のレジスタは、d0 ~ d15 の 16 本の倍精度レジスタとしても使用されます。 d_n は、 $s(2n)$ および $s(2n+1)$ と同じハードウェアを使用します。

VFPv3 は、d16 ~ d31 の 16 本の倍精度レジスタをさらに追加することにより、VFP レジスタセットを拡張します。これらのレジスタは、いずれの単精度 VFP レジスタとも重複しません。

————— 注 —————

使用するプロセッサに NEON と VFP の両方がある場合、すべての NEON レジスタが VFP レジスタと重複します。「NEON / VFP レジスタバンク」(P. 5-8) を参照して下さい。

以下のように使用できます。

- 一部のレジスタを単精度値用に使用しながら、同時に他のレジスタを倍精度値用に使用する。
- 別々のタイミングで、単精度値と倍精度値に同じレジスタを使用する。

対応する単精度レジスタと倍精度レジスタを同時に使用しないで下さい。レジスタが破損することはありませんが、有意な結果が得られなくなります。

5.13.1 レジスタバンク

VFP レジスタは次のように並んでいます。

- 8 本の単精度レジスタから成る 4 つのバンク ($s_0 \sim s_7$ 、 $s_8 \sim s_{15}$ 、 $s_{16} \sim s_{23}$ 、および $s_{24} \sim s_{31}$)
- 4 本の倍精度レジスタから成る 8 つ (VFPv2 では 4 つ) のバンク ($d_0 \sim d_3$ 、 $d_4 \sim d_7$ 、 $d_8 \sim d_{11}$ 、 $d_{12} \sim d_{15}$ 、 $d_{16} \sim d_{19}$ 、 $d_{20} \sim d_{23}$ 、 $d_{24} \sim d_{27}$ 、および $d_{28} \sim d_{31}$)
- 単精度レジスタと倍精度レジスタの任意の組み合わせ

詳細については、図 5-9 および図 5-10 を参照して下さい。

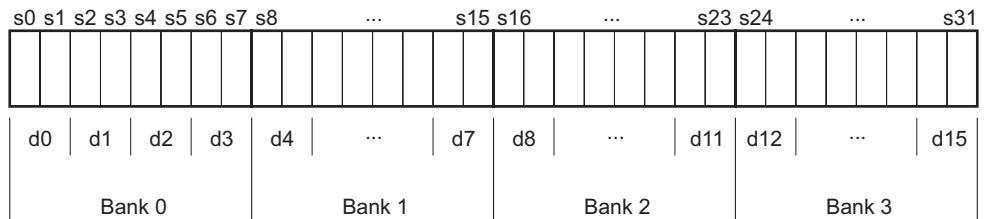


図 5-9 VFPv2 レジスタバンク

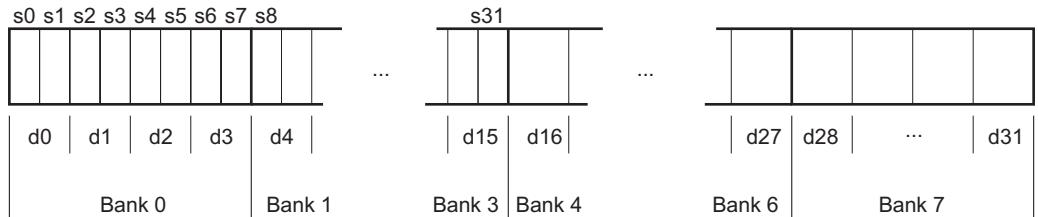


図 5-10 VFPv3 レジスタバンク

5.13.2 ベクタ

ベクタには、同じバンクから、最大 8 本の単精度レジスタまたは最大 4 本の倍精度レジスタを使用できます。ベクタによって使用されるレジスタの数は、FPSCR の LEN ビットによって制御されます（「FPSCR : 浮動小数点ステータス / 制御レジスタ」(P. 5-87) を参照）。

ベクタはどのレジスタからでも開始できます。ベクタによって使用される最初のレジスタは、各命令のレジスタフィールドで指定されます。

ベクタのラップアラウンド

ベクタがバンクの最終位置からはみ出ると、以下のように同じバンクの開始位置にラップアラウンドされます。

- s5 で始まる長さ 6 のベクタは {s5, s6, s7, s0, s1, s2} となります。
- s15 で始まる長さ 3 のベクタは {s15, s8, s9} となります。
- s22 で始まる長さ 4 のベクタは {s22, s23, s16, s17} となります。
- d7 で始まる長さ 2 のベクタは {d7, d4} となります。
- d10 で始まる長さ 3 のベクタは {d10, d11, d8} となります。

1 つのベクタが複数のバンクのレジスタを保持することはできません。

ベクタのストライド

上記の例が示すように、ベクタには連続するレジスタを使用できますが、1つおきのレジスタを使用することもできます。これは、FPSCR の STRIDE ビットによって制御されます（「FPSCR : 浮動小数点ステータス / 制御レジスタ」(P. 5-87) を参照）。以下に例を示します。

- s1 で始まる、長さ 3、ストライド 2 のベクタは {s1, s3, s5} となります。
- s6 で始まる、長さ 4、ストライド 2 のベクタは {s6, s0, s2, s4} となります。
- d1 で始まる、長さ 2、ストライド 2 のベクタは {d1, d3} となります。

ベクタの長さに関する制限

1 つのベクタに同じレジスタを 2 回使用することはできません。ベクタのラップアラウンドを可能にするには、以下のようなベクタは使用できません。

- 長さ > 4、ストライド = 2 の単精度ベクタ
- 長さ > 4、ストライド = 1 の倍精度ベクタ
- 長さ > 2、ストライド = 2 の倍精度ベクタ

5.14 VFP ベクタ演算とスカラ演算

VFP 算術命令を使用すると、以下を演算の対象にすることができます。

- スカラ
- ベクタ
- スカラとベクタ

ベクタの長さは、FPSCR の LEN ビットを使用して制御します（「FPSCR：浮動小数点データバス / 制御レジスタ」(P. 5-87) を参照）。

LEN が 1 の場合、すべての演算はスカラになります。

5.14.1 スカラ演算、ベクタ演算、および混合演算の制御

LEN が 1 より大きい場合、算術演算の動作は、デスティネーションレジスタとオペランドレジスタが配置されているレジスタバンクによって異なります（「レジスタバンク」(P. 5-83) を参照）。

以下の汎用形式の命令を指定するとします。

$$\begin{array}{l} \text{Op } Fd, Fn, Fm \\ \text{Op } Fd, Fm \end{array}$$

この場合、動作は以下のとおりです。

- Fd が最初または 5 番目のレジスタバンク ($s0 \sim s7$ 、 $d0 \sim d3$ 、または $d16 \sim d19$) にある場合、スカラ演算になります。
- Fm が最初または 5 番目のレジスタバンクにあるが、 Fd はない場合、混合演算になります。
- Fd と Fm のいずれも最初または 5 番目のレジスタバンクにない場合、ベクタ演算になります。

スカラ演算

Op は、 Fm の値と、 Fn が指定されている場合はその値に対する演算を行います。演算結果は Fd に返されます。

ベクタ演算

Op は、 Fm で始まるベクタ内の値と、 Fn が指定されている場合はその値で始まるベクタ内の値に対する演算を行います。演算結果は、 Fd で始まるベクタに返されます。

スカラ / ベクタ混合演算

オペランドが 1 つしかない命令の場合、 Op は、 Fm の 1 つの値に対して演算を行います。演算結果の LEN 個のコピーは、 Fd で始まるベクタに返されます。

オペランドが複数ある命令の場合、 Op は、 Fm の 1 つの値と Fn で始まるベクタの値に対して演算を行います。演算結果は、 Fd で始まるベクタに返されます。

5.15 VFP / NEON システムレジスタ

VFP と NEON のあらゆる実装で、以下の 3 つの VFP / NEON システムレジスタにアクセスできます。

- *FPSCR* : 浮動小数点ステータス / 制御レジスタ
- *FPEXC* : 浮動小数点例外レジスタ (P. 5-89)
- *FPSID* : 浮動小数点システム ID レジスタ (P. 5-89)
- NEON / VFP システムレジスタの各ビットの変更 (P. 5-90)

NEON または VFP の一部の実装では、追加のレジスタを指定することができます（使用する VFP コプロセッサ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）。

5.15.1 FPSCR : 浮動小数点ステータス / 制御レジスタ

FPSCR には、ユーザレベルの NEON / VFP ステータス / 制御ビットがすべて保持されています。NEON はビット [31:27] のみを使用します。ビットは以下のように使用されます。

ビット [31:28] N, Z, C、および V の各フラグです。これらは NEON / VFP ステータス フラグです。これらのフラグを CPSR のステータスフラグにコピーするまでは、これらを使用して条件実行を制御することはできません（「条件コード」(P. 5-10) を参照）。

ビット [27] QC (累積サチュレート) フラグです。NEON / VFP サチュレート命令でサチュレーションが発生した場合に設定されます。

ビット [24] ゼロクリアモード制御ビットです。

- | | |
|----------|-------------------|
| 0 | ゼロクリアモードは無効になります。 |
| 1 | ゼロクリアモードは有効になります。 |

使用するハードウェアとソフトウェアによっては、ゼロクリアモードを使用すると、範囲情報が失われる代わりにパフォーマンスが向上します（「ゼロクリアモード」(P. 5-91) を参照）。

——注——

NEON は、このビットに関係なくゼロクリアモードを常に使用します。

IEEE 754 との互換性が必要な場合、ゼロクリアモードを使用しないで下さい。

ビット [23:22] 丸めモードを以下のように制御します。

- | | |
|-------------|----------------------|
| 0b00 | 近似値への丸め (RN) モード |
| 0b01 | 正の無限大方向への丸め (RP) モード |
| 0b10 | 負の無限大方向への丸め (RM) モード |
| 0b11 | ゼロ方向への丸め (RZ) モード |

ビット [21:20] STRIDE は、ベクタ内で連続する値の間の距離です（「ベクタ」(P. 5-84) を参照）。ストライドは以下のように制御されます。

0b00	ストライド = 1
0b11	ストライド = 2

ビット [18:16] LEN は、各ベクタによって使用されるレジスタの数です（「ベクタ」(P. 5-84) を参照）。 $1 + \text{ビット}[18:16]$ の値になります。

0b000	LEN = 1
...	
0b111	LEN = 8

ビット [12:8] 例外のトラップイネーブルビットです。

IXE	不正確例外イネーブル
UFE	アンダーフロー例外イネーブル
OFE	オーバーフロー例外イネーブル
DZE	ゼロ除算例外イネーブル
IOE	無効演算例外イネーブル

本書では、浮動小数点例外トラップの使用について説明しません。詳細については、使用している VFP コプロセッサ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。

ビット [4:0] 累積例外ビットです。

IXC	不正確例外
UFC	アンダーフロー例外
OFC	オーバーフロー例外
DZC	ゼロ除算例外
IOC	無効演算例外

累積例外ビットは、対応する例外が発生すると設定されます。これらの例外ビットは、FPSCR への直接の書き込みによってクリアされるまで、設定されたままとなります。

その他すべてのビット 基本的な NEON / VFP 仕様では使用されません。これらのビットは特定の実装で使用できます（使用している VFP コプロセッサ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）。特定の実装で使用されている場合を除き、これらのビットを変更しないで下さい。

他のビットに影響を及ぼすことなく一部のビットだけを変更するには、読み出し - 変更 - 書き込みプロシージャを使用します（「NEON / VFP システムレジスタの各ビットの変更」(P. 5-90) を参照）。

5.15.2 FPEXC : 浮動小数点例外レジスタ

FPEXC には、特権モードでのみアクセスできます。FPEXC には以下のビットが保持されます。

ビット [31] EX ビットです。NEON / VFP のあらゆる実装で読み出すことができます。一部の実装では、このビットへの書き込みが可能な場合もあります。

値が 0 の場合、NEON / VFP システムは、汎用レジスタの内容に FPSCR および FPEXC を加えた状態になります。

値が 1 の場合は、状態を保存するには実装固有の情報が必要となります（使用している VFP コプロセッサ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）。

ビット [30] EN ビットです。NEON / VFP のあらゆる実装で読み出しと書き込みが可能です。

値が 1 で、NEON と VFP が存在する場合、これらは有効になり、正常に動作します。

値が 0 の場合、NEON と VFP は無効になります。無効になっている場合、FPSID レジスタまたは FPEXC レジスタの読み出しや書き込みを行うことはできますが、他の NEON / VFP 命令は未定義命令として処理されます。

ビット [29:0] VFP の特定の実装で使用される場合があります。本章で説明するすべての VFP 機能は、これらのビットにアクセスしなくとも使用できます。

特定の実装で使用されている場合を除き、これらのビットを変更しないで下さい（使用している VFP コプロセッサ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）。

他のビットに影響を及ぼすことなく一部のビットだけを変更するには、読み出し - 変更 - 書き込みプロシージャを使用します（「NEON / VFP システムレジスタの各ビットの変更」（P. 5-90）を参照）。

5.15.3 FPSID : 浮動小数点システム ID レジスタ

FPSID は、読み出し専用のレジスタです。このレジスタを読み出すと、使用しているプログラムが動作中の NEON / VFP アーキテクチャの実装を確認できます。

5.15.4 NEON / VFP システムレジスタの各ビットの変更

他のビットに影響を及ぼすことなく NEON / VFP システムレジスタの一部のビットだけを変更するには、以下の例に示すような読み出し - 変更 - 書き込みプロシージャを使用します。

```
FMRX    r10,FPSCR          ; copy FPSCR into r10
BIC     r10,r10,#0x00370000 ; clears STRIDE and LEN
ORR     r10,r10,#0x00030000 ; sets STRIDE = 1, LEN = 4
FMXR    FPSCR,r10         ; copy r10 back into FPSCR
```

「*MRS および MSR*」(P. 5-23) を参照して下さい。

5.16 ゼロクリアモード

VFP の一部の実装では、非正規化数を処理するためのサポートコードが使用されます。これらのシステムでは、非正規化数を計算する際のパフォーマンスは、正規計算を行う場合より大きく低下します。

ゼロクリアモードでは、非正規化数が 0 に置換されます。この処理は、IEEE 754 の演算に準拠していませんが、状況によってはパフォーマンスの大幅な向上につながります。

NEON と VFPv3 のゼロクリアでは符号ビットが維持され、VFPv2 のゼロクリアでは +0 にクリアされます。

NEON では、ゼロクリアモードを常に使用します。

5.16.1 ゼロクリアモードをいつ使用すべきか

以下の条件がすべて該当する場合は、ゼロクリアモードを選択して下さい。

- IEEE 754 への準拠がシステム要件ではない。
- 使用しているアルゴリズムによって非正規化数が生成される場合がある。
- 使用しているシステムで非正規化数を処理するサポートコードが使用されている。
- 使用しているアルゴリズムが保存される非正規化数の正確さに依存しない。
- 非正規化数を 0 に置換した結果、使用しているアルゴリズムで例外が頻繁に生成されない

コードの各部で要件が異なる場合は、ゼロクリアモードと標準モードをいつでも切り替えることができます。モードを切り替えて、レジスタに既に格納されている数値は影響を受けません。

5.16.2 ゼロクリアモードの使用による影響

一部の例外を除き（「ゼロクリアモードの影響を受けない演算」(P. 5-92) を参照）、ゼロクリアモードを使用すると、浮動小数点演算に以下のような影響があります。

- 非正規化数が浮動小数点演算への入力として使用されるとき、この非正規化数は 0 として処理されます。ソースレジスタは変更されません。
- 単精度浮動小数点演算の丸め前の結果が $-2^{-126} \sim +2^{-126}$ の範囲にある場合、この結果は 0 に置換されます。
- 倍精度浮動小数点演算の丸め前の結果が $-2^{-1022} \sim +2^{-1022}$ の範囲にある場合、この結果は 0 に置換されます。

非正規化数がオペランドとして使用されると不正確例外が発生するか、結果がゼロにクリアされます。アンダーフロー例外はゼロクリアモードでは発生しません。

5.16.3 ゼロクリアモードの影響を受けない演算

以下の NEON 演算と VFP 演算は、ゼロクリアモードにおいても結果をゼロにクリアすることなく非正規化数に対して実行できます。

- コピー、絶対値、および否定 (*/VMOV (レジスタ)* (P. 5-37) , */FABS*、*FCPY*、*FNEG* (P. 5-94) および */V{Q}ABS* および *V{Q}NEG* (P. 5-53) を参照)。
- 複製 (*/VDUP* (P. 5-34) を参照)。
- スワップ (*/VSWP* (P. 5-40) を参照)。
- ロードおよびストア (*/VLDR* および *VSTR* (P. 5-18) を参照)。
- 多重ロードおよび多重ストア (*/VLDM* および *VSTM* (P. 5-19) を参照)。
- NEON / VFP レジスタと ARM 汎用レジスタ間の転送 (*/VMOV (2 本の ARM レジスタと NEON / VFP の間)* (P. 5-20)、*/VMOV (1 本の ARM レジスタと NEON / VFP の間)* (P. 5-21)、および */VMOV (1 本の ARM レジスタと 単精度 VFP の間)* (P. 5-22) を参照)。

5.17 VFP 命令

このセクションは以下のサブセクションから構成されています。

- *FABS*、*FCPY*、*FNEG* (P. 5-94)
浮動小数点絶対値、コピー、および否定
- *FADD* および *FSUB* (P. 5-95)
浮動小数点加算および減算
- *FCMP* (P. 5-96)
浮動小数点比較
- *FCVTDS* (P. 5-97)
単精度浮動小数点から倍精度浮動小数点への変換
- *FCVTS*D (P. 5-98)
倍精度浮動小数点から単精度浮動小数点への変換
- *FDIV* (P. 5-99)
浮動小数点除算
- *FMAC*、*FNMAC*、*FMSC*、*FNMSC* (P. 5-100)
浮動小数点積和命令
- *FMUL*、*FNMUL* (P. 5-101)
浮動小数点乗算および否定乗算
- *FSITO*、*FUITO* (P. 5-102)
符号付き整数から浮動小数点への変換と、符号なし整数から浮動小数点への変換
- *FSQRT* (P. 5-103)
浮動小数点平方根
- *FTOSI*、*FTOUI* (P. 5-104)
浮動小数点から符号付き整数への変換と、浮動小数点から符号なし整数への変換
- *FCONSTS*、*FCONSTD* (P. 5-105)
単精度または倍精度レジスタへの浮動小数点定数の挿入
- *FSHTOS*、*FSHTOD*、*FSLTOS*、*FSLTOD*、*FUHTOS*、*FUHTOD*、*FULTOS*、*FULTOD* (P. 5-106)
固定小数点から単精度または倍精度浮動小数点への変換
- *FTOSHS*、*FTOSHD*、*FTOSLS*、*FTOSLD*、*FTOUHS*、*FTOUHD*、*FTOULS*、*FTOULD* (P. 5-107)
単精度または倍精度浮動小数点から固定小数点への変換

5.17.1 FABS、FCPY、FNEG

浮動小数点のコピー、絶対値、および否定

これらの命令では、スカラ、ベクタ、または混合演算を実行できます（「VFP ベクタ演算とスカラ演算」(P. 5-85) を参照）。

構文

`<op><precision>{cond} Fd, Fm`

各パラメータには以下の意味があります。

`<op>` FCPY、FABS、または FNEG のいずれかにします。

`<precision>` 単精度の場合は S、倍精度の場合は D にします。

`cond` 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

`Fd` 結果を保持する VFP レジスタです。

`Fm` オペランドを保持する VFP レジスタです。

`Fd` および `Fm` の精度は、`<precision>` で指定した精度と一致している必要があります。

使用法

FCPY 命令は、`Fm` の内容を `Fd` にコピーします。

FABS 命令は、`Fm` の内容を取得し、符号ビットをクリアして、その結果を `Fd` に返します。これにより、絶対値が得られます。

FNEG 命令は、`Fm` の内容を取得し、符号ビットを変更して、その結果を `Fd` に返します。これにより、値の否定が得られます。

オペランドが NaN の場合、符号ビットは上記の各ケースで決定されますが、例外は生成されません。

例外

これらの命令によって例外が生成されることはありません。

例

```
FABSD d3, d5
FNEGSMI s15, s15
```

5.17.2 FADD および FSUB

浮動小数点加算および減算

FADD および FSUB では、スカラ、ベクタ、または混合演算を実行できます（「VFP ベクタ演算とスカラ演算」（P. 5-85）を参照）。

構文

`FADD<precision>{cond} Fd, Fn, Fm`

`FSUB<precision>{cond} Fd, Fn, Fm`

各パラメータには以下の意味があります。

`<precision>` 単精度の場合は S、倍精度の場合は D にします。

`cond` 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

`Fd` 結果を保持する VFP レジスタです。

`Fn` 第 1 オペランドを保持する VFP レジスタです。

`Fm` 第 2 オペランドを保持する VFP レジスタです。

`Fd`、`Fn`、および `Fm` の精度は、`<precision>` で指定した精度と一致している必要があります。

使用法

FADD 命令は、`Fn` と `Fm` の値を加算して、その結果を `Fd` に返します。

FSUB 命令は、`Fm` の値を `Fn` の値から減算して、その結果を `Fd` に返します。

例外

FADD 命令と FSUB 命令は、無効演算例外、オーバーフロー例外、または不正確例外を生成する場合があります。

例

<code>FSUBSEQ</code>	<code>s2, s4, s17</code>
<code>FADDGT</code>	<code>d4, d0, d12</code>
<code>FSUBD</code>	<code>d0, d0, d12</code>

5.17.3 FCMP

浮動小数点比較

FCMP では常にスカラ演算が実行されます。

構文

`FCMP{E}{<precision>}{cond} Fd, Fm`

`FCMP{E}Z<precision>{cond} Fd`

各パラメータには以下の意味があります。

`E` 任意に指定できるパラメータです。`E` が指定されると、いずれかのオペランドが NaN の場合に例外が発生します。このパラメータが指定されていない場合は、いずれかのオペランドがシグナル型 NaN の場合にのみ例外が発生します。

`Z` ゼロとの比較を指定するパラメータです。

`<precision>` 単精度の場合は S、倍精度の場合は D にします。

`cond` 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

`Fd` 第 1 オペランドを保持する VFP レジスタです。

`Fm` 第 2 オペランドを保持する VFP レジスタです。ゼロとの比較命令の場合は `Fm` を省略します。

`Fd` および `Fm` の精度は、`<precision>` で指定した精度と一致している必要があります。

使用法

FCMP 命令は、`Fm` の値を `Fd` の値から減算し、その結果に基づいて VFP 条件フラグを設定します（「条件コード」（P. 5-10）を参照）。

例外

FCMP 命令は、無効演算例外を生成することができます。

例

```
FCMPS      s3, s0
FCMPEDNE   d5, d13
FCMPZSEQ   s2
```

5.17.4 FCVTDS

単精度浮動小数点から倍精度浮動小数点への変換

FCVTDS では常にスカラ演算が実行されます。

構文

FCVTDS{cond} Dd, Sm

各パラメータには以下の意味があります。

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

Dd 結果を保持する倍精度 VFP レジスタです。

Sm オペランドを保持する単精度 VFP レジスタです。

使用法

FCVTDS 命令は、Sm の単精度値を倍精度値に変換し、その結果を Dd に返します。

例外

FCVTDS 命令は、無効演算例外を生成することがあります。

例

FCVTDS	d5, s7
FCVTDSGT	d0, s4

5.17.5 FCVTSD

倍精度浮動小数点から单精度浮動小数点への変換

FCVTSD では常にスカラ演算が行われます。

構文

FCVTSD{*cond*} *Sd*, *Dm*

各パラメータには以下の意味があります。

cond 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

Sd 結果を保持する单精度 VFP レジスタです。

Dm オペランドを保持する倍精度 VFP レジスタです。

使用法

FCVTSD 命令は、*Dm* の倍精度値を单精度値に変換し、その結果を *Sd* に返します。

例外

FCVTSD 命令は、無効演算例外、オーバーフロー例外、アンダーフロー例外、または不正確例外を生成する場合があります。

例

```
FCVTSD      s3, d14
FCVTSDMI   s0, d1
```

5.17.6 FDIV

浮動小数点除算。

FDIV では、スカラ、ベクタ、または混合演算を実行できます（「VFP ベクタ演算とスカラ演算」（P. 5-85）を参照）。

構文

`FDIV<precision>{cond} Fd, Fn, Fm`

各パラメータには以下の意味があります。

`<precision>` 単精度の場合は S、倍精度の場合は D にします。

`cond` 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

`Fd` 結果を保持する VFP レジスタです。

`Fn` 第 1 オペランドを保持する VFP レジスタです。

`Fm` 第 2 オペランドを保持する VFP レジスタです。

`Fd, Fn`、および `Fm` の精度は、`<precision>` で指定した精度と一致している必要があります。

使用法

FDIV 命令は、`Fn` の値を `Fm` の値で除算し、その結果を `Fd` に返します。

例外

FDIV 演算は、ゼロ除算例外、無効演算例外、オーバーフロー例外、アンダーフロー例外、または不正確例外を生成する場合があります。

例

<code>FDIVS</code>	<code>s8, s0, s12</code>
<code>FDIVSNE</code>	<code>s2, s27, s28</code>
<code>FDIVD</code>	<code>d10, d2, d10</code>

5.17.7 FMAC、FNMAC、FMSC、FNMSC

浮動小数点積和、否定積和、積差、および否定積差。

これらの命令では、スカラ、ベクタ、または混合演算を実行できます（「VFP ベクタ演算とスカラ演算」（P. 5-85）を参照）。

構文

`<op><precision>{cond} Fd, Fn, Fm`

各パラメータには以下の意味があります。

`<op>` FMAC、FNMAC、FMSC、または FNMSC のいずれかにします。

`<precision>` 単精度の場合は S、倍精度の場合は D にします。

`cond` 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

`Fd` 結果を保持する VFP レジスタです。

`Fn` 第 1 オペランドを保持する VFP レジスタです。

`Fm` 第 2 オペランドを保持する VFP レジスタです。

`Fd`、`Fn`、および `Fm` の精度は、`<precision>` で指定した精度と一致している必要があります。

使用法

FMAC 命令は、 $Fd + Fn * Fm$ を計算し、その結果を `Fd` に返します。

FNMAC 命令は、 $Fd - Fn * Fm$ を計算し、その結果を `Fd` に返します。

FMSC 命令は、 $-Fd + Fn * Fm$ を計算し、その結果を `Fd` に返します。

FNMSC 命令は、 $-Fd - Fn * Fm$ を計算し、その結果を `Fd` に返します。

例外

これらの演算は、無効演算例外、オーバーフロー例外、アンダーフロー例外、または不正確例外を生成する場合があります。

例

FMACD	d8, d0, d8
FMACS	s20, s24, s28
FNMSCSLE	s6, s0, s26

5.17.8 FMUL、FNMUL

浮動小数点乗算および否定乗算。

FMUL および FNMUL では、スカラ、ベクタ、または混合演算を実行できます（「VFP ベクトル演算とスカラ演算」（P. 5-85）を参照）。

構文

`FMUL<precision>{cond} Fd, Fn, Fm`

`FNMUL<precision>{cond} Fd, Fn, Fm`

各パラメータには以下の意味があります。

`<precision>` 単精度の場合は S、倍精度の場合は D にします。

`cond` 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

`Fd` 結果を保持する VFP レジスタです。

`Fn` 第 1 オペランドを保持する VFP レジスタです。

`Fm` 第 2 オペランドを保持する VFP レジスタです。

`Fd`、`Fn`、および `Fm` の精度は、`<precision>` で指定した精度と一致している必要があります。

使用法

FMUL 命令は、`Fn` の値と `Fm` の値を乗算し、その結果を `Fd` に返します。

FNMUL 命令は、`Fn` の値と `Fm` の値を乗算し、その結果の否定を `Fd` に返します。

例外

FMUL および FNMUL 演算は、無効演算例外、オーバーフロー例外、アンダーフロー例外、または不正確例外を生成する場合があります。

例

<code>FNMUL S</code>	<code>s10, s10, s14</code>
<code>FMULD LT</code>	<code>d0, d7, d8</code>

5.17.9 FSITO、FUITO

符号付き整数から浮動小数点への変換と、符号なし整数から浮動小数点への変換

FSITO と FUITO では常にスカラ演算が実行されます。

構文

`FSITO<precision>{cond} Fd, Sm`

`FUITO<precision>{cond} Fd, Sm`

各パラメータには以下の意味があります。

`<precision>` 単精度の場合は S、倍精度の場合は D にします。

`cond` 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

`Fd` 結果を保持する VFP レジスタです。Fd の精度は、`<precision>` で指定した精度と一致している必要があります。

`Sm` 整数オペランドを保持する単精度 VFP レジスタです。

使用法

FSITO 命令は、`Sm` の符号付き整数値を浮動小数点に変換し、その結果を `Fd` に返します。

FUITO 命令は、`Sm` の符号なし整数値を浮動小数点に変換し、その結果を `Fd` に返します。

例外

FSITOS 命令と FUITOS 命令は、不正確例外を生成することができます。

FSITOD 命令と FUITOD 命令は、いずれの例外も生成しません。

例

<code>FUITOD</code>	<code>d3, s31 ; unsigned integer to double-precision</code>
<code>FSITOD</code>	<code>d5, s16 ; signed integer to double-precision</code>
<code>FSITOSNE</code>	<code>s2, s2 ; signed integer to single-precision</code>

5.17.10 FSQRT

浮動小数点平方根命令

この命令では、スカラ、ベクタ、または混合演算を実行できます（「VFP ベクタ演算とスカラ演算」（P. 5-85）を参照）。

構文

`FSQRT<precision>{cond} Fd, Fm`

各パラメータには以下の意味があります。

<precision> 単精度の場合は S、倍精度の場合は D にします。

cond 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

Fd 結果を保持する VFP レジスタです。

Fm オペランドを保持する VFP レジスタです。

Fd および *Fm* の精度は、*<precision>* で指定した精度と一致している必要があります。

使用法

FSQRT 命令は、*Fm* の内容の値の平方根を計算し、その結果を *Fd* に返します。

例外

FSQRT 演算は、無効演算例外または不正確例外を生成する場合があります。

例

FSQRTS	s4, s28
FSQRTD	d14, d6
FSQRTSNE	s15, s13

5.17.11 FTOSI、FTOUI

浮動小数点から符号付き整数への変換と、浮動小数点から符号なし整数への変換。

FTOSI と FTOUI では常にスカラ演算が実行されます。

構文

`FTOSI{Z}{precision}{cond} Sd, Fm`

`FTOUI{Z}{precision}{cond} Sd, Fm`

各パラメータには以下の意味があります。

Z ゼロ方向への丸めを指定するオプションのパラメータです。このパラメータが指定されている場合、FPSCR で現在指定されている丸めモードはオーバーライドされます。FPSCR は変更されません。

<precision> 単精度の場合は S、倍精度の場合は D にします。

cond 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

Sd 整数結果を保持する単精度 VFP レジスタです。

Fm オペランドを保持する VFP レジスタです。*Fm* の精度は、*<precision>* で指定した精度と一致している必要があります。

使用法

FTOSI 命令は、*Fm* の浮動小数点値を符号付き整数に変換し、その結果を *Sd* に返します。

FTOUI 命令は、*Fm* の浮動小数点値を符号なし整数に変換し、その結果を *Sd* に返します。

例外

FTOSI 命令と FTOUI 命令は、無効演算例外または不正確例外を生成する場合があります。

例

<code>FTOSID</code>	<code>s10, d2</code>
<code>FTOUID</code>	<code>s3, d1</code>
<code>FTOSIZS</code>	<code>s3, s31</code>

5.17.12 FCONSTS、FCONSTD

単精度または倍精度レジスタへの浮動小数点定数の挿入。

FCONSTS と FCONSTD では常にスカラ演算が実行されます。

構文

`FCONST<precision>{cond} Fd, #imm8`

各パラメータには以下の意味があります。

<precision> 単精度の場合は S、倍精度の場合は D にします。

cond 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

Fd デステイネーションレジスタです。Fd の精度は、*<precision>* で指定した精度と一致している必要があります。

imm8 浮動小数点定数を生成するために使用されるイミディエート値です。

定数値

以下の定数が生成されます。

$(-1)^{0bA} * (2 + (0bEFGH / 8)) * 2^{0bBCD}$

ここで、0bABCDEFGH は *imm8* の値になります。また、0bBCD は符号付きの 2 の補数値として取得され、0bEFGH は符号なしバイナリ値として取得されます。

例外

なし

アーキテクチャ

これらの命令は VFPv3 で使用できます。

5.17.13 FSHTOS、FSHTOD、FSLTOS、FSLTOD、FUHTOS、FUHTOD、FULTOS、FULTOD

固定小数点から単精度または倍精度浮動小数点への変換。

これらの命令では常にスカラ演算が実行されます。固定小数点数をレジスタから取得し、浮動小数点数に変換して、その結果を同じレジスタに返します。

構文

`F<source>T0<dest>{cond} Fd, #fbits`

各パラメータには以下の意味があります。

`<source>` ソースを指定します。以下のいずれかになります。

SH	16 ビットの符号付きハーフワード
SL	32 ビットの符号付きロングワード
UH	16 ビットの符号なしハーフワード
UL	32 ビットの符号なしロングワード

`<dest>` 単精度の場合は S、倍精度の場合は D にします。

`cond` 任意の条件コードを指定します（「条件コード」(P. 5-10) を参照）。

`Fd` ソースレジスタとデスティネーションレジスタです。`Fd` の精度は、`<dest>` で指定した精度と一致している必要があります。ソースは、`Fd` の最下位 16 ビットまたは 32 ビットです。

`fbits` ソース内の小数部ビットの数を指定します。

例外

不正確

アーキテクチャ

これらの命令は VFPv3 で使用できます。

注

ゼロ 入力の固定小数点値がゼロの場合、デスティネーション精度の結果は +0.0 になります。

丸め これらの変換では、近似値への丸めモードが使用されます。

ベクタ これらの命令は、FPSCR の LEN フィールドにかかわらず、常にスカラ演算を指定します。

5.17.14 FTOSHS、FTOSHD、FTOSLS、FTOSLD、FTOUHS、FTOUHD、FTOULS、FTOULD

単精度または倍精度浮動小数点から固定小数点への変換。

これらの命令では常にスカラ演算が実行されます。浮動小数点数をレジスタから取得し、固定小数点数に変換して、その結果を同じレジスタに返します。

構文

`FTO<dest><source>{cond} Fd, #fbits`

各パラメータには以下の意味があります。

`<dest>` ソースを指定します。以下のいずれかになります。

SH	16 ビットの符号付きハーフワード
SL	32 ビットの符号付きロングワード
UH	16 ビットの符号なしハーフワード
UL	32 ビットの符号なしロングワード

`<source>` 単精度の場合は S、倍精度の場合は D にします。

`cond` 任意の条件コードを指定します（「条件コード」（P. 5-10）を参照）。

`Fd` ソースレジスタとデスティネーションレジスタです。`Fd` の精度は、`<dest>` で指定した精度と一致している必要があります。デスティネーションは、`Fd` の最下位 16 ビットまたは 32 ビットです。

`fbits` 結果の小数部ビットの数を指定します。

例外

無効演算、不正確

アーキテクチャ

これらの命令は VFPv3 で使用できます。

5.18 VFP 擬似命令

VFP 擬似命令は 1 つあります。

5.18.1 VLDR 擬似命令

VLDR 擬似命令は、VFP 浮動小数点レジスタと単精度または倍精度の浮動小数点定数をロードします。

————— 注 —————

VLDR を使用できるのは、コマンドラインオプション `--fpu` が `vfpv2` に設定されている場合のみです。

このセクションでは、VLDR 擬似命令についてのみ説明します。VLDR 命令については、「*VLDR および VSTR*」(P. 5-18) を参照して下さい。

構文

`VLDR{cond} fp-register,=fp-literal`

各パラメータには以下の意味があります。

`<precision>` 単精度の場合は S、倍精度の場合は D にします。

`cond` 任意の条件コードを指定します。

`fp-register` ロードする浮動小数点レジスタです。

`fp-literal` 単精度または倍精度の浮動小数点リテラルを指定します（「*浮動小数点リテラル*」(P. 3-30) を参照）。

使用法

アセンブラーは、定数をリテラルプールに配置し、この定数をリテラルプールから読み出すプログラム相対 VLDR 命令を生成します。単精度定数のストアにはリテラルプール内の 1 ワードが使用され、倍精度定数のストアには 2 ワードが使用されます。

PC から定数までのオフセットは、1KB 未満にする必要があります。リテラルプールが範囲内にあることを必ず確認して下さい。詳細については、「*LTORG*」(P. 7-19) を参照して下さい。

5.19 VFP ディレクティブとベクタ表記

このセクションの説明は、`armasm` にのみ該当します。C コンパイラおよび C++ コンパイラのインラインアセンブラーでは、このセクションで説明するディレクティブやベクタ表記を使用できません。

コード内で VFP ベクタの長さとストライドに関する情報をアサートし、アセンブラーで自動的にチェックすることができます。以下を参照して下さい。

- *VFPASSERT SCALAR* (P. 5-110)
- *VFPASSERT VECTOR* (P. 5-111)

`VFPASSERT` ディレクティブを使用する場合は、すべての VFP データ処理命令でベクタ情報を指定する必要があります。ベクタ表記については、「ベクタ表記」で説明します。`VFPASSERT` ディレクティブを使用しない場合は、この表記を使用しないで下さい。

5.19.1 ベクタ表記

VFP データ処理命令では、以下のようにかぎ括弧を使用して VFP レジスタのベクタを指定します。

- sn は、単精度スカラレジスタ n です。
- $sn\langle>$ は、長さとストライドが現在のベクタの長さとストライドによって指定される、レジスタ n で始まる単精度ベクタです。
- $sn\langle L \rangle$ は、レジスタ n で始まる、長さ L 、ストライド 1 の単精度ベクタです。
- $sn\langle L:S \rangle$ は、レジスタ n で始まる、長さ L 、ストライド S の単精度ベクタです。
- dn は、倍精度スカラレジスタ n です。
- $dn\langle>$ は、長さとストライドが現在のベクタの長さとストライドによって指定される、レジスタ n で始まる倍精度ベクタです。
- $dn\langle L \rangle$ は、レジスタ n で始まる、長さ L 、ストライド 1 の倍精度ベクタです。
- $dn\langle L:S \rangle$ は、レジスタ n で始まる、長さ L 、ストライド S の倍精度ベクタです。

このベクタ表記には、`DN` ディレクティブと `SN` ディレクティブを使用して定義された名前を使用できます（`/DN, SN`）(P. 7-15) を参照）。

このベクタ表記を `DN` ディレクティブや `SN` ディレクティブの中で使用しないで下さい。

5.19.2 VFPASSERT SCALAR

VFPASSERT SCALAR ディレクティブは、後続の VFP 命令がスカラモードであることをアセンブラーに通知します。

構文

VFPASSERT SCALAR

使用法

VFPASSERT SCALAR ディレクティブを使用して、VFP モードが VECTOR である任意のコードブロックの終わりをマークできます。

VFPASSERT SCALAR ディレクティブは、モード変更が発生する命令の直後に配置します。これは一般に FMXR 命令ですが、BL 命令の場合もあります。

関数の終了時に VFP がベクタモードになることが予測される場合は、VFPASSERT SCALAR ディレクティブを最後の命令の直後に配置します。このような関数は AAPCS に準拠しません。詳細については、*install_directory\Documentation\Specifications\...* にある *Procedure Call Standard for the ARM Architecture* (aapcs.pdf) を参照して下さい。

以下も参照して下さい。

- ベクタ表記 (P. 5-109)
- VFPASSERT VECTOR (P. 5-111)

—— 注 ——

このディレクティブはプログラマによるアサートにすぎず、コードは生成されません。これらのアサートが互いに一致していない場合や、VFP データ処理命令のベクタ表記と一致しない場合は、アセンブラーによってエラーメッセージが生成されます。

ベクタの長さが 1 である場合でも、VFPASSERT SCALAR ディレクティブに続く VFP データ処理命令のベクタ表記が検出されると、アセンブラーによってエラーが生成されます。

例

```
VFPASSERT SCALAR          ; scalar mode
faddd    d4, d4, d0        ; okay
fadds    s4<3>, s0, s8<3> ; ERROR, vector in scalar mode
fabss    s24<1>, s28<1>   ; ERROR, vector in scalar mode
                           ; (even though length==1)
```

5.19.3 VFPASSERT VECTOR

VFPASSERT VECTOR ディレクティブは、後続の VFP 命令がベクタモードであることをアセンブラーに通知します。ベクタの長さとストライドを指定することもできます。

構文

`VFPASSERT VECTOR[<[n[:s]]>]`

各パラメータには以下の意味があります。

n 1 ~ 8 でベクタの長さを指定します。

s 1 ~ 2 でベクタのストライドを指定します。

使用法

VFPASSERT VECTOR ディレクティブを使用して、VFP モードが VECTOR となる命令ブロックの開始位置と、ベクタの長さまたはストライドの変更をマークできます。

VFPASSERT VECTOR ディレクティブは、変更が発生する命令の直後に配置します。これは一般に FMXR 命令ですが、BL 命令の場合もあります。

関数のエントリ処理で VFP がベクタモードになることが予測される場合は、VFPASSERT VECTOR ディレクティブを最初の命令の直前に配置します。このような関数は AAPCS に準拠しません。詳細については、

install_directory\Documentation\Specifications\... にある *Procedure Call Standard for the ARM Architecture* (aapcs.pdf) を参照して下さい。

以下も参照して下さい。

- ベクタ表記 (P. 5-109)
- *VFPASSERT SCALAR* (P. 5-110)

注

このディレクティブはプログラマによるアサートにすぎず、コードは生成されません。これらのアサートが互いに一致しない場合や、VFP データ処理命令のベクタ表記と一致しない場合は、アセンブラーによってエラーメッセージが生成されます。

例

```

FMRX    r10,FPSCRFLEx1m
ORR     r10,r10,#0x00020000 ; set length = 3, stride = 1
FMXR    FPSCR,r10

VFPASSERT VECTOR          ; assert vector mode, unspecified length & stride
faddd  d4, d4, d0          ; ERROR, scalar in vector mode
fadds  s16<3>, s0, s8<3> ; okay
fabss  s24<1>, s28<1>    ; wrong length, but not faulted (unspecified)

FMRX    r10,FPSCR
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00030000 ; set length = 4, stride = 1
FMXR    FPSCR,r10

VFPASSERT VECTOR<4>       ; assert vector mode, length 4, stride 1
fadds  s24<4>, s0, s8<4> ; okay
fabss  s24<2>, s24<2>    ; ERROR, wrong length

FMRX    r10,FPSCR
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00130000 ; set length = 4, stride = 2
FMXR    FPSCR,r10

VFPASSERT VECTOR<4:2>      ; assert vector mode, length 4, stride 2
fadds  s8<4>, s0, s16<4> ; ERROR, wrong stride
fabss  s16<4:2>, s28<4:2> ; okay
fadds  s8<>, s2, s16<>   ; okay (s8 and s16 both have
                           ; length 4 and stride 2.
                           ; s2 is scalar.)

```

第 6 章

ワイヤレス MMX テクノロジの命令

本章では、ARM® アセンブラーである `armasm` でのワイヤレス MMX™ テクノロジの命令の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。

- [はじめに \(P. 6-2\)](#)
- [ワイヤレス MMX テクノロジに対する ARM のサポート \(P. 6-3\)](#)
- [命令の概要 \(P. 6-7\)](#)

6.1 はじめに

ワイヤレス MMX テクノロジは SIMD (1 つの命令で複数のデータを並列に処理する) 命令のセットで、マルチメディアアプリケーションのパフォーマンスを向上させる一部の XScale プロセッサで使用できます。ワイヤレス MMX テクノロジでは 64 ビットレジスタを使用して、パック形式の複数のデータ要素で演算できるようになります。

ワイヤレス MMX テクノロジでは ARM コプロセッサ 0 および 1 を使用して、その命令セットとデータ型をサポートしています。ワイヤレス MMX テクノロジの命令を使用するソースコードをアセンブルして、PXA270 プロセッサで実行できます。

ARM アセンブラーを使用するときは、以下の点に注意してください。

- ワイヤレス MMX テクノロジの命令は、サポートされているプロセッサ (`armasm --cpu PXA270`) を指定した場合にのみアセンブルされます。
- PXA270 プロセッサでは、ARM または Thumb® で記述されたコードのみをサポートします。
- ARM フラグの状態によっては、ワイヤレス MMX テクノロジの命令は条件実行できます。これに対する例外は、制御レジスタ (`wC*`) へのロード命令およびストア命令です。

ワイヤレス MMX テクノロジの条件コードは、ARM の条件コードと同じです。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*を参照して下さい。

本章では、RVCT における ARM アセンブラーによるワイヤレス MMX テクノロジのサポートについて説明します。ワイヤレス MMX テクノロジについては詳しく説明していません。プログラマモデル、およびワイヤレス MMX テクノロジの命令セットの詳細な説明については、*Wireless MMX Technology Developer Guide*を参照して下さい。

6.2 ワイヤレス MMX テクノロジに対する ARM のサポート

このセクションでは、ワイヤレス MMX テクノロジに対するアセンブラーのサポートについて説明します。このセクションは、以下のサブセクションから構成されています。

- レジスタ
- ディレクティブ、*WRN* と *WCN* (P. 6-4)
- *Frame* ディレクティブ (P. 6-4)
- ロード命令およびストア命令 (P. 6-5)
- ワイヤレス MMX テクノロジの命令と *XScale* の命令 (P. 6-6)

6.2.1 レジスタ

ワイヤレス MMX テクノロジでは、次の 2 つのタイプのレジスタをサポートしています。

ステータスレジスタおよび制御レジスタ

制御レジスタではコプロセッサ 1 にマップし、汎用レジスタ *wCGR0* ~ *wCGR3* および SIMD フラグを含んでいます。これらのレジスタの詳細については、表 6-1 を参照して下さい。

これらのレジスタに対する読み出しおよび書き込みには、ワイヤレス MMX テクノロジの命令 *TMCR* および *TMRC* を使用します。

表 6-1 ステータスレジスタおよび制御レジスタ

タイプ	ワイヤレス MMX テクノロジのレジスタ	CP1 レジスタ
コプロセッサ ID	<i>wCID</i>	<i>c0</i>
制御	<i>wCon</i>	<i>c1</i>
サチュレーション SIMD フラグ	<i>wCSSF</i>	<i>c2</i>
算術 SIMD フラグ	<i>wCASF</i>	<i>c3</i>
予約	-	<i>c4</i> ~ <i>c7</i>
汎用	<i>wCGR0</i> ~ <i>wCGR3</i>	<i>c8</i> ~ <i>c11</i>
予約	-	<i>c12</i> ~ <i>c15</i>

SIMD データレジスタ

データレジスタ (wR0 ~ wR15) ではコプロレッサ 0 にマップし、16 ビット × 64 ビットのパック化データを保持します。これらのレジスタと ARM レジスタ間でデータを移動するには、ワイヤレス MMX テクノロジの疑似命令である TMRRC および TMCRR を使用します。

レジスタの詳細については、*Wireless MMX Technology Developer Guide* を参照して下さい。

ワイヤレス MMX テクノロジの命令をアセンブルする場合、アセンブラーは次のようなレジスタの仕様を受け入れます。

- ワイヤレス MMX テクノロジの仕様と完全に一致する、大文字と小文字の混在。
wR0、wCID、wCon など。
- すべて小文字。
wr0、wcid、wcon など。
- すべて大文字。
WR0、WCID、WCON など。

レジスタ名を指定するときは、WRN ディレクティブおよび WCN ディレクティブをサポートします（「ディレクティブ、WRN と WCN」を参照）。

6.2.2 ディレクティブ、WRN と WCN

次のディレクティブを使用して、ワイヤレス MMX テクノロジをサポートできます。

WCN 指定された制御レジスタ名を定義します。例は以下の通りです。
`speed WCN wcgr0 ; defines speed as a symbol for control reg 0`

WRN 指定された SIMD データレジスタ名を定義します。例は以下の通りです。
`rate WRN wr6 ; defines rate as a symbol for data reg 6`

同一レジスタに複数の名前を付けて矛盾を生じさせるような使用方法は避けて下さい。「定義済みのレジスタおよびコプロセッサ名」(P. 3-18) に記載されている事前定義された名前や、「レジスタ」(P. 6-3) に記載されているレジスタ名を使用しないで下さい。

6.2.3 Frame ディレクティブ

ワイヤレス MMX テクノロジのレジスタを通常の方法で FRAME ディレクティブとともに使用して、オブジェクトファイルにデバッグ情報を追加できます（詳細については、「フレームディレクティブ」(P. 7-41) を参照）。次の制限に注意して下さい。

- ワイヤレス MMX テクノロジのレジスタである wR0 ~ wR9 または wCGR0 ~ wCGR3 をスタックにプッシュしようとすると警告が表示されます（「FRAME PUSH」(P. 7-45) を参照）。
- ワイヤレス MMX テクノロジのレジスタは、アドレスオフセットとして使用できません（「FRAME ADDRESS」(P. 7-43) および「FRAME RETURN ADDRESS」(P. 7-49) を参照）。

6.2.4 ロード命令およびストア命令

アセンブラーでは、次の形式のワイヤレス MMX テクノロジのロード命令およびストア命令をサポートします。

```
op{B,H,W,D}{cond} wRd, [Rn {, #offset}]{!}
opW wCd, [Rn {, #offset}]{!}
op{B,H,W,D}{cond} wRd, [Rn], #{-}offset
opW wCd, [Rn], #{-}offset
```

以下に例を示します。

```
WLDRW wr3, [r4, #-0x3fc]
WLDRH wr9, [r10], #0x45
WLDRW wc1, [r2, #0]
WSTRH wr1, [r2, #0]
WSTRD wr11, [r12], #-0x2f4
WSTRW wc3, [r4, #-0x3fc]
```

ロード（またはストア）がワードまたはダブルワードであるプログラム相対シンボルまたはレジスタ相対シンボルを指定できます。例は次の通りです。

```
WLDRW wr0, localdatal ; loads a word located at label localdatal
```

アセンブラーで WLDR または WSTR のターゲットを非境界整列することを決める場合は、警告が表示されます。

SIMD レジスタへの定数のロード

アセンブラーでは、WLDRW リテラルロード疑似命令もサポートします。例は次の通りです。

```
WLDRW wr0, =0x114
```

次の点に注意して下さい。

- アセンブラーでは、バイトおよびハーフワードリテラルをロードできません。これらをロードすると降格エラーが生成されます。降格されると、命令は WLDRW に変換され、32 ビットリテラルが生成されます。これはバイトリテラルのロードと同様ですが、32 ビットワードを使用する点が異なります。
- ロードするリテラルがゼロで、デスティネーションが SIMD データレジスタである場合、アセンブラーでは命令を WZERO に変換します。
- 8 バイト整列ではないダブルワードロードは予測不能です。

6.2.5 ワイヤレス MMX テクノロジの命令と XScale の命令

ワイヤレス MMX テクノロジの命令は XScale の命令と重複します。競合を避けるために、アセンブラーには次の制約があります。

- XScale の命令とワイヤレス MMX テクノロジの命令と同じアセンブリに混在させることはできません。
- ワイヤレス MMX テクノロジの TMIA 命令には、XScale の MIA 命令と重複する MIA ニーモニックがあります。次の点に注意して下さい。
 - MIA acc0, Rm, Rs は XScale では使用できますが、ワイヤレス MMX テクノロジではエラーになります。
 - MIA wR0, Rm, Rs および TMIA wR0, Rm, Rs はワイヤレス MMX テクノロジで使用できます。
 - TMIA acc0, Rm, Rs は XScale ではエラーになります (XScale には TMIA 命令はありません)。

XScale 命令の詳細については、「その他の命令」(P. 4-136) を参照して下さい。

6.3 命令の概要

表 6-2 では、ワイヤレス MMX テクノロジの命令セットの概要を説明します。この表を使用して、*Wireless MMX Technology Developer Guide* で説明されている個々の命令を配置します。「疑似命令」(P. 6-11 表 6-3) も参照して下さい。

このセクションでは、ワイヤレス MMX テクノロジのレジスタは *wRn*、*wRd*、ARM のレジスタは *Rn*、*Rd* と示されます。

特に明示されない限り、すべての命令は条件実行できますが、ARM フラグの状態によって異なります。

表 6-2 ワイヤレス MMX テクノロジの命令

二モニック	概要	例
TANDC	SIMD PSR (<i>wCASF</i>) の値の論理積を求めます。結果を ARM CPSR に入れます。フラグを設定するバイト、ハーフワード、またはワード演算の後に実行できます。	TANDCB r15
TBCST	ソースレジスタ <i>Rn</i> からデスティネーションレジスタ <i>wRd</i> の各 SIMD 位置に値を伝達します。8、16、および 32 ビット値で演算できます。	TBCSTB wr15, r1
TEXTRC	SIMD PSR (<i>wCASF</i>) から 3 ビットのイミディエート値によって指定された 4、8、または 16 ビットデータを抽出し、ARM CPSR に移動します。 <i>r15</i> 以外のデスティネーションレジスタを指定すると、未定義命令例外が発生します。	TEXTRCB r15, #0
TEXTRM	ソースレジスタ <i>wRn</i> から 3 ビットのイミディエート値によって指定された 8、16、または 32 ビットデータを抽出し、デスティネーションレジスタ <i>Rd</i> に移動します。 <i>Rd</i> に <i>r15</i> を使用しないで下さい。	TEXTRMUBCS r3, wr7, #7
TINSR	ソースレジスタ <i>Rn</i> から 8、16、または 32 ビットデータを 3 ビットのイミディエート値によって指定されているデスティネーションレジスタ <i>wRd</i> の位置に移動します。	TINSRB wr6, r11, #0
TMIA、 TMIAPH、 TMIAxy	ソースレジスタ <i>Rm</i> および <i>Rs</i> の符号付き整数を積和し、結果をデスティネーションレジスタ <i>wRd</i> と累積します (XScale コプロセッサ 0 命令 <i>/MIA, MIAPH, MIAxy/</i> (P. 4-97) を参照)。 <i>Rm</i> または <i>Rs</i> に <i>r15</i> を使用しないで下さい。先行 T のない二モニックを使用できます ('ワイヤレス MMX テクノロジの命令と XScale の命令' (P. 6-6) も参照)。	TMIANE wr1, r2, r3 TMIAPH wr4, r5, r6 TMIABB wr4, r5, r6 MIAPHNE wr4, r5, r6
TMOVMSK	ソースレジスタ <i>wRn</i> の各 SIMD フィールドの最上位ビットをデスティネーションレジスタ <i>Rd</i> の最下位 8、4、または 2 ビットに移動します。8、16、および 32 ビット値で演算できます。 <i>Rd</i> に <i>r15</i> を使用しないで下さい。	TMOVMSKNE r14, wr15

表 6-2 ワイヤレス MMX テクノロジの命令（続き）

ニーモニック	概要	例
TORC	SIMD PSR (wCASF) の値の論理和を求めます。結果を ARM CPSR に入れます。フラグを設定するバイト、ハーフワード、またはワード演算の後に実行できます。r15 以外のデスティネーションレジスタを指定すると、未定義命令例外が発生します。	TORCB r15
WACC	ソースレジスタ wRn の値間で符号なし累積を行います。結果をデスティネーションレジスタ wRd に入れます。バイト、ハーフワード、ワードで実行できます。	WACCBGE wr1, wr2
WADD	8、16、または 32 ビット符号付きまたは符号なしデータのベクタに対し、ソースレジスタ wRn および wRm のベクタ加算を行います。結果をデスティネーションレジスタ wRd に入れます。サチュレーションは、符号付き、符号なし、またはサチュレーションなしとして指定できます。	WADDBGE wr1, wr2, wr13
WALIGNI、 WALIGNR	2つの 64 ビットソースレジスタ wRn および wRm から 64 ビット値を抽出します。結果をデスティネーションレジスタ wRd に入れます。 WALIGNI では 3 ビットのイミディエート値を、 WALIGNR では所定の汎用レジスタの 3 ビット値を使用して、抽出する値のバイトオフセットを指定します。	WALIGNI wr7, wr6, wr5,#3 WALIGNR wr4, wr8, wr12
WAND、WANDN	ソースレジスタ wRn および wRm の値に対し、ビット単位論理積または否定論理積を実行します。結果をデスティネーションレジスタ wRd に入れます。	WAND wr1, wr2, wr3 WANDN wr5, wr5, wr9
WAVG2	8 または 16 ビットデータの符号なしベクタに対し、ソースレジスタ wRn および wRm の 2 ピクセル平均を行います。オプションで、+1 で丸めます。結果をデスティネーションレジスタ wRd に入れます。	WAVG2B wr3, wr6, wr9 WAVG2BR wr4, wr7, wr10
WCMPEQ	8、16、または 32 ビットデータのベクタに対し、ソースレジスタ wRn および wRm のベクタが同じかどうかを比較します。対応するビットが $wRn = wRm$ であるときは、デスティネーションレジスタ wRd の対応するビットをすべて 1 に設定します。それ以外はすべてゼロです。	WCMPEQB wr0, wr4, wr2
WCMPGT	8、16、または 32 ビットデータのベクタに対し、ソースレジスタ wRn および wRm のベクタの大きさを比較します。対応するビットが $wRn > wRm$ であるときは、デスティネーションレジスタ wRd の対応するビットをすべて 1 に設定します。それ以外はすべてゼロです。符号付きデータまたは符号なしデータに実行できます。	WCMPGTUB wr0, wr4, wr2

表 6-2 ワイヤレス MMX テクノロジの命令（続き）

ニーモニック	概要	例
WLDR	データレジスタ wRd または制御レジスタ wCx のいずれかにメモリからロードを行います。データレジスタでは 8、16、32、および 64 ビットデータ値をロードできます。32 ビット値のみ、制御レジスタにロードできます。ロードされたデータはゼロ拡張です。 制御レジスターへのロードは条件実行できません。 この命令の ARM の実装の詳細については、「ロード命令およびストア命令」(P. 6-5) を参照して下さい。	WLDRB wr1, [r2, #0]
WMAC	ソースレジスタ wRn および wRm のベクタ乗算を実行します。結果を 16 ビットデータのみのベクタの wRd と累積できます。	WMACU wr3, wr4, wr5
WMADD	16 ビットベクタ乗算を実行します。 wRd の下位ワードに下位の積 2つを、 wRd の上位ワードに上位の積 2つを合計します。中間の積は 32 ビットです。符号付きデータまたは符号なしデータのいずれかに対して演算できます。結果が結果フィールドよりも大きい場合は切り捨てられます。	WMADDU wr3, wr4, wr5
WMAX、WMIN	8、16、または 32 ビットデータのベクタに対し、ソースレジスタ wRn および wRm のベクタ最大値、または最小値、要素の選択を実行します。結果をデスティネーションレジスタ wRd に入れます。符号付きデータまたは符号なしデータで演算できます。	WMAXUB wr0, wr4, wr2 WMINSB wr0, wr4, wr2
WMUL	16 ビットデータのみのベクタに対し、ソースレジスタ wRn および wRm にベクタ乗算を行います。 符号付きデータまたは符号なしデータに実行できます。	WMULUL wr4, wr2, wr3
WOR	ソースレジスタ wRn および wRm 間でビット単位論理和を実行します。 結果をデスティネーションレジスタ wRd に入れます。	WOR wr3, wr1, wr4
WPACK	ソースレジスタ wRn および wRm のデータを結合し、サチュレートされた結果を生成します。16、32、または 64 ビットデータのベクタに対し、 wRm をデスティネーションレジスタ wRd の上位半分に、 wRn を下位半分にパックします。符号付きの値を使用して、符号付きまたは符号なしのサチュレーションを生成します。	WPACKHUS wr2, wr7, wr1
WROR	16、32、または 64 ビットデータのベクタに対し、 wRm によるソースレジスタ wRn のベクタ論理右ロテートを行います。結果をデスティネーションレジスタ wRd に入れます。シフト値の汎用データレジスタを指定できます。	WRORH wr3, wr1, wr4

表 6-2 ワイヤレス MMX テクノロジの命令（続き）

ニーモニック	概要	例
WSAD	ソースレジスタ wRn および wRm の絶対差を合計します。結果を wRd と累積します。8 または 16 ビット符号なしデータベクタに対して演算できます。結果を 32 ビットのみに累積します。	WSADB wr3, wr5, wr8
WSHUFH	8 ビットのイミディエート値で指定した、ソースレジスタ wRn の 16 ビットフィールドから、デスティネーションレジスタ wRd の 16 ビットデータ値をシャッフルします。	WSHUFH wr8, wr15, #17
WSLL, WSRL	16、32、または 64 ビットデータのベクタに対し、 wRm によるソースレジスタ wRn のベクタ論理左シフト、または右シフトを行います。結果をデスティネーションレジスタ wRd に入れます。シフト値の汎用データレジスタを指定できます。	WSLLH wr3, wr1, wr4 WSRLHG wr3, wr1, wcgr0
WSRA	16、32、または 64 ビットデータのベクタに対し、 wRm によるソースレジスタ wRn のベクタ算術右シフトを実行します。結果をデスティネーションレジスタ wRd に入れます。シフト値の汎用データレジスタを指定できます。	WSRAH wr3, wr1, wr4 WSRAHG wr3, wr1, wcgr0
WSTR	ソースデータレジスタ wRm 、または制御レジスタ wCx のいずれかから Rn レジスタおよび対応するアドレス修飾子によってアドレスが指定されているメモリ位置にストアを実行します。データレジスタは 8、16、32、および 64 ビットデータ値を保存できます。32 ビットのみ、制御レジスタから保存できます。この命令の ARM 実装の詳細については、「ロード命令およびストア命令」(P. 6-5) を参照して下さい。	WSTRB wr1, [r2, #0] WSTRW wc1, [r2, #0]
WSUB	8、16、または 32 ビット符号付きまたは符号なしデータのベクタに対し、ソースレジスタ wRn および wRm のベクタ減算を行います。結果をデスティネーションレジスタ wRd に入れます。サチュレーションは、符号付き、符号なし、またはサチュレーションなしとして指定できます。	WSUBBGE wr1, wr2, wr13
WUNPCKEH, WUNPCKEL	ソースレジスタ wRn の上位半分、または下位半分から 8、16、または 32 ビットデータを展開します。各フィールドをゼロ拡張または符号拡張します。結果をデスティネーションレジスタ wRd に入れます。	WUNPCKEHUB wr0, wr4 WUNPCKELSB wr0, wr4
WUNPCKIH, WUNPCKIL	ソースレジスタ wRn の上位半分または下位半分から 8、16、または 32 ビットデータを展開します。 wRm の上位半分または下位半分でインターリープします。結果をデスティネーションレジスタ wRd に入れます。	WUNPCKIHB wr0, wr4, wr2 WUNPCKILH wr1, wr5, wr3
WXOR	ソースレジスタ wRn および wRm の間でビット単位排他的論理和を行います。結果をデスティネーションレジスタ wRd に入れます。	WXOR wr3, wr1, wr4

6.3.1 疑似命令

表 6-3 では、ワイヤレス MMX テクノロジの疑似命令の概要を説明します。この表を使用して、*Wireless MMX Technology Developer Guide* および「第 4 章 ARM 命令と Thumb 命令」で説明されている命令を配置します。

表 6-3 ワイヤレス MMX テクノロジの疑似命令

ニーモニック	概要	例
TMCR	ソースレジスタ Rn の内容を制御レジスタ wCn に移動します。 ARM MCR コプロセッサ命令にマップします (P. 4-128)。	TMCR $wC1, r10$
TMCRR	2つのソースレジスタ $RnLo$ および $RnHi$ の内容をデスティネーションレジスタ wRd に移動します。 $RnLo$ または $RnHi$ のいずれかに $r15$ を使用しないで下さい。ARM MCRR コプロセッサ命令にマップします (P. 4-128)。	TMCRR $wR4, r5, r6$
TMRC	制御レジスタ wCn の内容をデスティネーションレジスタ Rd に移動します。 Rd に $r15$ を使用しないで下さい。ARM MRC コプロセッサ命令にマップします (P. 4-130)。	TMRC $r1, wc2$
TMRRC	ソースレジスタ wRn の内容を 2つのデスティネーションレジスタ $RdLo$ および $RdHi$ に移動します。いずれかのデスティネーションレジスタに $r15$ を使用しないで下さい。 $RdLo$ および $RdHi$ は別のレジスタである必要があります。それ以外の場合は結果が予測できないものになるため、信頼できません。 ARM MRRC コプロセッサ命令にマップします (P. 4-130)。	TMRRC $r1, r0, wr2$
WMOV	ソースレジスタ wRn の内容をデスティネーションレジスタ wRd に移動します。この命令は、WOR の形式です (P. 6-7 表 6-2 を参照)。	WMOV $wR1, wr8$
WZERO	デスティネーションレジスタ wRd をゼロにします。この命令は、WANDN の形式です (P. 6-7 表 6-2 を参照)。	WZERO $wR1$

第7章

ディレクティブリファレンス

本章では、ARM[®]アセンブラー armasm に使用できるディレクティブ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。

- ディレクティブの一覧 (アルファベット順) (P. 7-2)
- シンボル定義ディレクティブ (P. 7-4)
- データ定義ディレクティブ (P. 7-17)
- アセンブリ制御ディレクティブ (P. 7-32)
- フレームディレクティブ (P. 7-41)
- 通知ディレクティブ (P. 7-56)
- 命令セットと構文選択のディレクティブ (P. 7-61)
- その他のディレクティブ (P. 7-63)

——注——

本章で説明するディレクティブは、ARM C コンパイラおよび C++ コンパイラのインラインアセンブラーには使用できません。

7.1 ディレクティブの一覧（アルファベット順）

表 7-1 は、ディレクティブの一覧を示しています。この表を使用して、各ディレクティブの説明が記載されているページに移動できます。

表 7-1 各ディレクティブの参照先

ディレクティブ	ページ	ディレクティブ	ページ	ディレクティブ	ページ
ALIGN	P. 7-64	EQU	P. 7-70	LCLA、LCLL、LCLS	P. 7-7
ARM、CODE32	P. 7-62	EXPORT、GLOBAL	P. 7-71	LTORG	P. 7-19
AREA	P. 7-66	EXPORTAS	P. 7-73	MACRO、MEND	P. 7-33
ASSERT	P. 7-56	EXTERN	P. 7-74	MAP	P. 7-20
CN	P. 7-13	FIELD	P. 7-21	MEND (MACRO 参照)	P. 7-33
CODE16	P. 7-62	FRAME ADDRESS	P. 7-43	MEXIT	P. 7-36
COMMON	P. 7-31	FRAME POP	P. 7-44	NOFP	P. 7-81
CP	P. 7-14	FRAME PUSH	P. 7-45	OPT	P. 7-58
DATA	P. 7-31	FRAME REGISTER	P. 7-47	PRESERVE8 (REQUIRE8 参照)	P. 7-82
DCB	P. 7-23	FRAME RESTORE	P. 7-48	PROC (FUNCTION 参照)	P. 7-54
DCD、DCDU	P. 7-24	FRAME SAVE	P. 7-50	QN	P. 7-16
DCDO	P. 7-25	FRAME STATE REMEMBER	P. 7-51	RELOC	P. 7-9
DCF、DCFUDU	P. 7-26	FRAME STATE RESTORE	P. 7-52	REQUIRE	P. 7-81
DCFS、DCFSU	P. 7-27	FRAME UNWIND ON、FRAME UNWIND OFF	P. 7-53	REQUIRE8、PRESERVE8	P. 7-82
DCI	P. 7-28	FUNCTION、PROC	P. 7-54	RLIST	P. 7-12
DCQ、DCQU	P. 7-29	GBLA、GBLL、GBLS	P. 7-5	RN	P. 7-11
DCW、DCWU	P. 7-30	GET、INCLUDE	P. 7-76	ROUT	P. 7-84
DN、SN	P. 7-15	GLOBAL (EXPORT 参照)	P. 7-71	SETA、SETL、SETS	P. 7-8
ELIF、ELSE (IF 参照)	P. 7-37	IF、ELSE、ENDIF、ELIF	P. 7-37	SN (DN 参照)	P. 7-15
END	P. 7-68	IMPORT	P. 7-77	SPACE	P. 7-22
ENDFUNC、ENDP	P. 7-55	INCBIN	P. 7-79	THUMB	P. 7-62

表 7-1 各ディレクティブの参照先 (続き)

ディレクティブ	ページ	ディレクティブ	ページ	ディレクティブ	ページ
ENDIF (IF 参照)	P. 7-37	INCLUDE (GET 参照)	P. 7-76	THUMBX	P. 7-62
ENTRY	P. 7-69	INFO	P. 7-57	TTL、SUBT	P. 7-60
-	-	KEEP	P. 7-80	WHILE、WEND	P. 7-40

7.2 シンボル定義ディレクティブ

このセクションでは、以下のディレクティブについて説明します。

- *GBLA*、*GBLL*、*GBLS* (P. 7-5)
グローバル算術変数、論理変数、または文字列変数を宣言します。
- *LCLA*、*LCLL*、*LCLS* (P. 7-7)
ローカル算術変数、論理変数、または文字列変数を宣言します。
- *SETA*、*SETL*、*SETS* (P. 7-8)
算術変数、論理変数、または文字列変数の値を設定します。
- *RELOC* (P. 7-9)
オブジェクトファイルで ELF の再配置をエンコードします。
- *RN* (P. 7-11)
指定されたレジスタの名前を定義します。
- *RLIST* (P. 7-12)
汎用レジスタセットの名前を定義します。
- *CN* (P. 7-13)
コプロセッサレジスタ名を定義します。
- *CP* (P. 7-14)
コプロセッサ名を定義します。
- *DN*、*SN* (P. 7-15)
倍精度または単精度の VFP レジスタ名を定義します。
- *QN* (P. 7-16)
NEON™ Q レジスタ名を定義します。

7.2.1 GBLA、GBLL、GBLS

GBLA ディレクティブは、グローバル算術変数を宣言し、その値をゼロに初期化します。

GBLL ディレクティブは、グローバル論理変数を宣言し、その値を {FALSE} に初期化します。

GBLS ディレクティブは、グローバル文字列変数を宣言し、その値を NULL 文字列 "" に初期化します。

構文

<gblx> variable

パラメータの説明 :

<gblx> GBLA、GBLL、または GBLS のいずれかを指定します。

variable 変数の名前を指定します。*variable* は、ソースファイル内のシンボルの中で一意である必要があります。

使用法

定義済みの変数に対して上記のディレクティブのいずれかを使用すると、その変数は上記に記載されている値に再初期化されます。

変数の有効範囲は、その変数を含むソースファイル内に制限されています。

変数の値は、SETA、SETL、または SETS のいずれかのディレクティブを使用して設定します（/SETA、SETL、SETS/ (P. 7-8) 参照）。

ローカル変数の宣言については、/LCLA、LCLL、LCLS/ (P. 7-7) を参照して下さい。

グローバル変数は、アセンブラーのコマンドラインオプション -predefine を使用して設定することもできます。詳細については、「コマンド構文」(P. 3-2) を参照して下さい。

例

例 7-1 では、変数 `objectsize` が宣言され、`objectsize` の値に `0xFF` が設定されます。その後、その値が `SPACE` ディレクティブで使用されます。

例 7-1

```
        GBLA    objectsize ; declare the variable name
objectsize SETA    0xFF   ; set its value
.
.
.
SPACE   objectsize ; quote the variable
```

例 7-2 は、`armasm` を呼び出す場合の変数の宣言方法と設定方法を示しています。アセンブリ時に変数の値を設定する場合は、この方法を使用して下さい。`--pd` は `--predefine` と同じ意味です。

例 7-2

```
armasm --predefine "objectsize SETA 0xFF" sourcefile -o objectfile
```

7.2.2 LCLA、LCLL、LCLS

LCLA ディレクティブは、ローカル算術変数を宣言し、その値をゼロに初期化します。

LCLL ディレクティブは、ローカル論理変数を宣言し、その値を {FALSE} に初期化します。

LCLS ディレクティブは、ローカル文字列変数を宣言し、その値を NULL 文字列 "" に初期化します。

構文

<lclx> variable

パラメータの説明：

<lclx> LCLA、LCLL、または LCLS のいずれかを指定します。

variable 変数の名前を指定します。*variable* は、マクロ内で一意である必要があります。

使用法

定義済みの変数に対して上記のディレクティブのいずれかを使用すると、その変数は上記に記載されている値に再初期化されます。

変数の有効範囲は、その変数を含むマクロの特定のインスタンスに制限されています (/MACRO、MEND) (P. 7-33) 参照)。

変数の値は、SETA、SETL、または SETS のいずれかのディレクティブを使用して設定します (/SETA、SETL、SETS) (P. 7-8) 参照)。

グローバル変数の宣言については、/GBLA、GBLL、GBLS (P. 7-5) を参照して下さい。

例

```

MACRO ; Declare a macro
$label message $a ; Macro prototype line
          LCLS   err ; Declare local string
                      ; variable err.
err      SETS    "error no: " ; Set value of err
$label  ; code
INFO    0, "err":CC::STR:$a ; Use string
MEND

```

7.2.3 SETA、SETL、SETS

SETA ディレクティブは、ローカル算術変数またはグローバル算術変数の値を設定します。

SETL ディレクティブは、ローカル論理変数またはグローバル論理変数の値を設定します。

SETS ディレクティブは、ローカル文字列変数またはグローバル文字列変数の値を設定します。

構文

`variable <setx> expr`

パラメータの説明：

`<setx>` SETA、SETL、または SETS のいずれかを指定します。

`variable` GBLA、GBLL、GBLS、LCLA、LCLL、または LCLS のいずれかのディレクティブで宣言される変数の名前を指定します。

`expr` 以下の式を指定します。

- SETA の場合は、数値式（「数値式」（P. 3-28）参照）
- SETL の場合は、論理式（「論理式」（P. 3-31）参照）
- SETS の場合は、文字列式（「文字列式」（P. 3-27）参照）

使用法

いずれかのディレクティブを使用する前に、グローバル宣言ディレクティブまたはローカル宣言ディレクティブを使用して `variable` を宣言する必要があります。詳細については、「*GBLA、GBLL、GBLS*」（P. 7-5）および「*LCLA、LCLL、LCLS*」（P. 7-7）を参照して下さい。

また、変数名はコマンドラインで事前に定義することもできます。詳細については、「コマンド構文」（P. 3-2）を参照して下さい。

例

	GBLA	VersionNumber
VersionNumber	SETA	21
	GBLL	Debug
Debug	SETL	{TRUE}
	GBLS	VersionString
VersionString	SETS	"Version 1.0"

7.2.4 RELOC

RELOC ディレクティブは、オブジェクトファイルで ELF の再配置を明示的にエンコードします。

構文

`RELOC n, symbol`

`RELOC n`

パラメータの説明：

n 0 ~ 255 の範囲内の値を指定する必要があります。

symbol 任意のプログラム相対ラベルを指定できます。

使用法

`RELOC n, symbol` を使用し、*symbol* というラベルの付いたアドレスを基準にして再配置を行います。

RELOC ディレクティブを ARM 命令または Thumb 命令の直後に使用すると、その命令で再配置が行われます。RELOC ディレクティブを DCB、DCW、DCD、またはその他のデータ生成ディレクティブの直後に使用すると、データの開始位置で再配置が行われます。適用される加数は、命令または DCI か DCD でエンコードする必要があります。

アセンブラーによって同じ場所で既に再配置が行われている場合、RELOC ディレクティブの設定内容で再配置が更新されます。以下に例を示します。

```
DCD      sym2 ; R_ARM_ABS32 to sym32
RELOC    55   ; ... makes it R_ARM_ABS32 NOI
```

RELOC は、データ生成ディレクティブ以外のディレクティブ、LTORG、ALIGN の後に使用したり、AREA の最初の項目として使用したりした場合など、その他すべての場合に失敗します。

RELOC *n* を使用し、匿名のシンボル（シンボルテーブルのシンボル 0）を基準に再配置を行います。以前にアセンブラーによって再配置が行われていない場合に RELOC *n* を使用すると、匿名のシンボルを基準に再配置が行われます。

例

```
IMPORT  impsym
LDR    r0,[pc,#-8]
RELOC  4, impsym

DCD    0
RELOC  2, sym

DCD    0,1,2,3,4 ; the final word is relocated
RELOC  38,sym2   ; R_ARM_TARGET1
```

7.2.5 RN

RN ディレクティブは、指定されたレジスタのレジスタ名を定義します。

構文

name RN expr

パラメータの説明：

name レジスタに割り当てる名前を指定します。*name* には、「定義済みのレジスタおよびコプロセッサ名」(P. 3-18) に記載された定義済みの名前と同じ名前を指定できません。

expr 0 ~ 15 の Q レジスタ番号を求める式を指定します。

使用法

RN を使用して適切な名前をレジスタに割り当てることにより、各レジスタの使用目的を明確にすることができます。同一レジスタに複数の名前を付けて矛盾を生じさせるような使用方法は避けて下さい。

例

regname RN 11 ; defines regname for register 11

sqr4 RN r6 ; defines sqr4 for register 6

7.2.6 RLIST

RLIST (レジスタリスト) ディレクティブは、汎用レジスタセットの名前を指定します。

構文

```
name RLIST {list-of-registers}
```

パラメータの説明：

name レジスタセットに割り当てる名前を指定します。*name* には、「定義済みのレジスタおよびプロセッサ名」(P. 3-18) に記載された定義済みの名前と同じ名前を指定できません。

list-of-registers

レジスタ名またはレジスタ範囲、あるいはその両方をコンマで区切って並べたリストを指定します。レジスタリストは括弧で囲む必要があります。

使用法

RLIST を使用して、LDM 命令または STM 命令によって転送されるレジスタセットに名前を付けます。

LDM および STM は、LDM 命令または STM 命令に指定された順序に関係なく、最も小さい物理レジスタ番号の内容を、メモリ内の最下位アドレスに格納します。シンボルレジスタ名を既に定義している場合は、レジスタリストが昇順で指定されているかどうかが分かりにくいことがあります。

アセンブラーオプション `--diag_warning 1206` を使用して、レジスタリスト内のレジスタが昇順で指定されていることを確認して下さい。レジスタが昇順で指定されていない場合には、警告が生成されます。

例

```
Context RLIST {r0-r6,r8,r10-r12,r15}
```

7.2.7 CN

CN ディレクティブは、コプロセッサレジスタの名前を定義します。

構文

name CN *expr*

パラメータの説明：

- | | |
|-------------|---|
| <i>name</i> | コプロセッサレジスタに定義する名前を指定します。 <i>name</i> には、「定義済みのレジスタおよびコプロセッサ名」(P.3-18) に記載された定義済みの名前と同じ名前を指定できません。 |
| <i>expr</i> | 0 ~ 15 のコプロセッサレジスタ番号を求める式を指定します。 |

使用法

CN を使用して適切な名前をレジストリに割り当てることにより、各レジスタの使用目的を明確にすることができます。

—— 注 ——

同一レジスタに複数の名前を付けて矛盾を生じさせるような使用方法は避けて下さい。

c0 ~ c15 という名前が事前に定義されています。

例

```
power    CN  6      ; defines power as a symbol for
                      ; coprocessor register 6
```

7.2.8 CP

CP ディレクティブは、指定されたコプロセッサの名前を定義します。コプロセッサ番号には、0 ~ 15 の番号を指定する必要があります。

構文

name CP expr

パラメータの説明：

name コプロセッサに割り当てる名前を指定します。*name* には、「定義済みのレジスタおよびコプロセッサ名」(P. 3-18) に記載された定義済みの名前と同じ名前を指定できません。

expr 0 ~ 15 のコプロセッサ番号を求める式を指定します。

使用法

CP を使用して適切な名前をコプロセッサに割り当てることにより、各コプロセッサの使用目的を明確にすることができます。

—————注—————

同一コプロセッサに複数の名前を付けて矛盾を生じさせるような使用方法は避けて下さい。

コプロセッサ 1 ~ 15 には、p0 ~ p15 という名前が事前に定義されています。

例

```
dmu    CP  6      ; defines dmu as a symbol for  
           ; coprocessor 6
```

7.2.9 DN、SN

DN ディレクティブは、指定された倍精度 VFP レジスタの名前を定義します。VFPv2 用には d0 ~ d15 と D0 ~ D15、VFPv3 用には d16 ~ d31 と D16 ~ D31 の名前が事前に定義されています。

SN ディレクティブは、指定された単精度 VFP レジスタの名前を定義します。s0 ~ s31 と S0 ~ S31 という名前が事前に定義されています。

構文

name DN expr

name SN expr

パラメータの説明：

name VFP レジスタに割り当てる名前を指定します。*name* には、「定義済みのレジスタおよびコプロセッサ名」(P. 3-18) に記載された定義済みの名前と同じ名前を指定できません。

expr 必要に応じて、0 ~ 15 (VFPv2 の場合) または 0 ~ 31 (VFPv3 の場合) の倍精度 VFP レジスタ番号を求める式、または 0 ~ 31 の単精度 VFP レジスタ番号を求める式を指定します。

使用法

DN または SN を使用して適切な名前を VFP レジスタに割り当てるにより、各レジスタの使用目的を明確にすることができます。

注

同一レジスタに複数の名前を付けて矛盾を生じさせるような使用方法は避け下さい。

DN ディレクティブまたは SN ディレクティブではベクタ長を指定できません（「VFP ディレクティブとベクタ表記」(P. 5-109) 参照）。

例

```
energy DN 6 ; defines energy as a symbol for
; VFP double-precision register 6
```

```
mass SN 16 ; defines mass as a symbol for
; VFP single-precision register 16
```

7.2.10 QN

QN ディレクティブは、指定された NEON クワッドワードレジスタの名前を定義します。q0 ~ q15 および Q0 ~ Q15 の名前は NEON 用に事前定義されています。

構文

name QN *expr*

パラメータの説明：

name Q レジスタに割り当てる名前を指定します。*name* には、「定義済みのレジスタおよびコプロセッサ名」(P. 3-18) に記載された事前定義されている名前と同じ名前を指定できません。

expr 0 ~ 15 の Q レジスタ番号を求める式を指定します。

使用法

QN を使用して Q レジスタに適切な名前を割り当てることにより、各レジスタの使用目的が分かりやすくなります。

—————注—————

同一レジスタに複数の名前を付けて矛盾を生じさせるような使用方法は避けて下さい。

例

speed QN 5 ; defines speed as a symbol for Q register 5

7.3 データ定義ディレクティブ

このセクションでは、メモリの割り当て、データ構造の定義、およびメモリの初期内容の設定を行う以下のディレクティブについて説明します。

- *LTORG* (P. 7-19)
リテラルプールの起点を設定します。
- *MAP* (P. 7-20)
記憶域マップの起点を設定します。
- *FIELD* (P. 7-21)
記憶域マップ内のフィールドを定義します。
- *SPACE* (P. 7-22)
ゼロ初期化されるメモリブロックを割り当てます。
- *DCB* (P. 7-23)
バイト単位でメモリを割り当てて、初期内容を指定します。
- *DCD*, *DCDU* (P. 7-24)
ワード単位でメモリを割り当てて、初期内容を指定します。
- *DCDO* (P. 7-25)
ワード単位でメモリを割り当てて、スタティックベースレジスタからのオフセットとして初期内容を指定します。
- *DCFD*, *DCFDU* (P. 7-26)
ダブルワード単位でメモリを割り当てて、倍精度浮動小数点数として初期内容を指定します。
- *DCFS*, *DCFSU* (P. 7-27)
ワード単位でメモリを割り当てて、单精度浮動小数点数として初期内容を指定します。
- *DCI* (P. 7-28)
ワード単位でメモリを割り当てて、初期内容を指定します。メモリ位置をデータではなくコードとしてマークします。

- *DCQ*、*DCQU* (P. 7-29)
ダブルワード単位でメモリを割り当てて、64 ビット整数として初期内容を指定します。
- *DCW*、*DCWU* (P. 7-30)
ハーフワード単位でメモリを割り当てて、初期内容を指定します。
- *COMMON* (P. 7-31)
ブロック単位でメモリをシンボルに割り当てて、境界調整を指定します。
- *DATA* (P. 7-31)
コードセクション内のデータをマークします。このディレクティブは、下位互換性を維持する目的だけでサポートされています。

7.3.1 LTORG

LTORG ディレクティブは、現在のリテラルプールをすぐにアセンブルするようアセンブラーに指示します。

構文

LTORG

使用法

アセンブラーは、各コードセクションの終了位置で現在のリテラルプールをアセンブルします。コードセクションの終了位置は、次のセクションの先頭にある AREA ディレクティブによって決まるか、またはアセンブリの終了位置となります。

このようなデフォルトのリテラルプールは、LDR、FLDD、および FLDS 擬似命令の範囲外になる場合があります。詳細については、「*LDR 擬似命令*」(P. 4-160) を参照して下さい。リテラルプールが範囲内でアセンブルされていることを確認するには、LTORG を使用します。大きなプログラムでは、複数のリテラルプールが必要になる場合があります。

定数がプロセッサによって命令として実行されないように、LTORG ディレクティブは、無条件分岐またはサブルーチン復帰命令の後に配置して下さい。

リテラルプール内のデータは、アセンブラーによってワード境界で整列されます。

例

```

        AREA   Example, CODE, READONLY
start  BL     func1

func1           ; function body
; code
LDR    r1,=0x55555555 ; => LDR R1, [pc, #offset to Literal Pool 1]
; code
MOV    pc,lr       ; end function
LTORG          ; Literal Pool 1 contains literal &55555555.

data   SPACE  4200      ; Clears 4200 bytes of memory,
; starting at current location.
END               ; Default literal pool is empty.

```

7.3.2 MAP

MAP ディレクティブは、指定されたアドレスを記憶域マップの起点として設定します。記憶域マップの位置カウンタ {VAR} には、同じアドレスが設定されます。^ は MAP と同じ意味です。

構文

MAP *expr{,base-register}*

パラメータの説明：

expr 数値式またはプログラム相対式を指定します。

- *base-register* が指定されていない場合は、*expr* によって記憶域マップの開始アドレスが求められます。記憶域マップの位置カウンタには、同じアドレスが設定されます。
- *expr* にプログラム相対式が指定されている場合、マップ内でラベルを使用するには、ラベルを事前に定義しておく必要があります。マップでは、アセンブラーの最初のパスでラベルを定義する必要があります。

base-register

レジスタを指定します。*base-register* が指定されている場合、記憶域マップの開始アドレスは、*expr* と、ランタイムの *base-register* の値の合計になります。

使用法

記憶域マップを記述するには、MAP ディレクティブを FIELD ディレクティブと組み合わせて使用します。

base-register を指定してレジスタ相対ラベルを定義します。次の MAP ディレクティブが出現するまで、このベースレジスタが、後続の FIELD ディレクティブで定義されるすべてのラベル内で暗黙的に使用されます。レジスタ相対ラベルは、ロード命令とストア命令で使用できます。この例については、「FIELD」(P. 7-21) を参照して下さい。

MAP ディレクティブを複数回使用することにより、複数の記憶域マップを定義できます。最初の MAP ディレクティブが使用されるまで、{VAR} カウンタにはゼロが設定されます。

例

```
MAP      0,r9
MAP      0xffff,r9
```

7.3.3 FIELD

FIELD ディレクティブは、MAP ディレクティブを使用して定義された記憶域マップ内の空間を定義します。# は FIELD と同じ意味です。

構文

{*label*} FIELD *expr*

パラメータの説明：

label ラベルを指定します（省略可）。指定されている場合は、*label* に記憶域マップの位置カウンタ {VAR} の値が割り当てられます。その後、記憶域位置マップのカウンタは、*expr* の値でインクリメントされます。

expr 記憶域マップの位置カウンタをインクリメントするバイト数を求める式を指定します。

使用法

記憶域マップが *base-register* を指定する MAP ディレクティブによって設定されている場合は、次の MAP ディレクティブが出現するまで、このベースレジスタが、後続の FIELD ディレクティブで定義されるすべてのラベル内で暗黙的に使用されます。これらのレジスタ相対ラベルは、ロード命令とストア命令で使用できます（/MAP/ (P. 7-20) 参照）。

例

以下の例は、MAP ディレクティブと FIELD ディレクティブを使用してレジスタ相対ラベルを定義する方法を示しています。

```

MAP      0,r9          ; set {VAR} to the address stored in r9
FIELD    4              ; increment {VAR} by 4 bytes
Lab FIELD  4           ; set Lab to the address [r9 + 4]
                      ; and then increment {VAR} by 4 bytes
LDR      r0,Lab         ; equivalent to LDR r0,[r9,#4]

```

7.3.4 SPACE

SPACE ディレクティブは、ゼロ初期化されるメモリブロックを予約します。% は SPACE と同じ意味です。

構文

```
{label} SPACE expr
```

パラメータの説明：

expr ゼロ初期化される確保バイト数を求める式を指定します（「*数値式*」（P. 3-28）参照）。

使用法

SPACE ディレクティブの後のコードを整列させるには、ALIGN ディレクティブを使用します。詳細については、「*ALIGN*」（P. 7-64）を参照して下さい。

以下も参照して下さい。

- DCB (P. 7-23)
- DCD、DCDU (P. 7-24)
- DCDO (P. 7-25)
- DCW、DCWU (P. 7-30)

例

```
AREA   MyData, DATA, READWRITE  
data1  SPACE   255      ; defines 255 bytes of zeroed store
```

7.3.5 DCB

DCB ディレクティブは、バイト単位でメモリを割り当てて、ランタイム時のメモリの初期内容を定義します。= は DCB と同じ意味です。

構文

{*label*} DCB *expr*{,*expr*}...

パラメータの説明：

- | | |
|-------------|----------------|
| <i>expr</i> | 以下のいずれかを指定します。 |
|-------------|----------------|
- $-128 \sim 255$ の範囲の整数を求める数値式（「数値式」（P. 3-28）参照）。
 - 引用符で囲まれた文字列。文字列中の文字はストアの連続したバイトにロードされます。

使用法

DCB の後に命令が続く場合、ALIGN ディレクティブを使用して命令を整列させて下さい。 詳細については、「ALIGN」（P. 7-64）を参照して下さい。

以下も参照して下さい。

- *DCD*、*DCDU*（P. 7-24）
- *DCQ*、*DCQU*（P. 7-29）
- *DCW*、*DCWU*（P. 7-30）
- *SPACE*（P. 7-22）

例

C 言語の文字列とは異なり、ARM アセンブラーの文字列の終端は NULL ではありません。 終端が NULL の C 言語の文字列は、DCB を以下のように使用して作成できます。

```
C_string    DCB  "C_string",0
```

7.3.6 DCD、DCDU

DCD ディレクティブは、ワード単位でメモリを割り当てて 4 バイト境界で整列させ、ランタイム時のメモリの初期内容を定義します。

& は DCD と同じ意味です。

メモリの境界調整が任意である点を除き、DCDU は DCD と同じ意味です。

構文

{*label*} DCD{U} *expr*{,*expr*}

パラメータの説明：

expr 以下のいずれかを指定します。

- 数値式（「数値式」(P. 3-28) 参照）
- プログラム相対式

使用法

DCD は、必要に応じて、最初に定義されたワードの前に最大 3 バイトのパディングを挿入して、境界調整を 4 バイトにします。

境界調整が不要な場合は、DCDU を使用して下さい。

以下も参照して下さい。

- DCB (P. 7-23)
- DCW、DCWU (P. 7-30)
- DCQ、DCQU (P. 7-29)
- SPACE (P. 7-22)

例

```
data1  DCD    1,5,20      ; Defines 3 words containing
                           ; decimal values 1, 5, and 20

data2  DCD    mem06 + 4   ; Defines 1 word containing 4 +
                           ; the address of the label mem06

                           AREA  MyData, DATA, READWRITE
                           DCB   255          ; Now misaligned ...
data3  DCDU   1,5,20      ; Defines 3 words containing
                           ; 1, 5 and 20, not word aligned
```

7.3.7 DCD0

DCD0 ディレクティブは、ワード単位でメモリを割り当てて、4 バイト境界で整列させ、ランタイム時のメモリの初期内容をスタティックベースレジスタ sb (r9) からのオフセットとして定義します。

構文

```
{Label} DCD0 expr{,expr}...
```

パラメータの説明 :

<i>expr</i>	レジスタ相対式またはラベルを指定します。ベースレジスタには sb を指定する必要があります。
-------------	--

使用法

DCD0 を使用して、スタティックベースレジスタと相対的な再配置可能アドレスのためにメモリ内の空間を割り当てます。

例

```
IMPORT externsym
DCD0    externsym ; 32-bit word relocated by offset of
; externsym from base of SB section.
```

7.3.8 DCFD、DCFDU

DCFD ディレクティブは、ワード境界で整列された倍精度浮動小数点数にメモリを割り当てる、ランタイム時のメモリの初期内容を定義します。倍精度浮動小数点数には 2 ワードが使用されます。この 2 ワードを算術演算で使用するには、ワード境界で整列させる必要があります。

メモリの境界調整が任意である点を除き、DCDFU は DCFD と同じ意味です。

構文

```
{label} DCFD{U} fpliteral{,fpliteral}...
```

パラメータの説明：

fpliteral 倍精度浮動小数点リテラルを指定します（「浮動小数点リテラル」(P. 3-30) 参照）。

使用法

アセンブラーは、必要に応じて、最初に定義された数値の前に最大 3 バイトのパディングを挿入して、境界調整を 4 バイトにします。

境界調整が不要な場合は、DCFDU を使用して下さい。

fpliteral を内部形式に変換する場合に使用されるワードの順序は、選択された浮動小数点アーキテクチャによって制御されます。--fpu none オプションを選択すると、DCFD および DCFDU は使用できません。

倍精度数値の範囲は以下のとおりです。

- 最大 : 1.79769313486231571e+308
- 最小 : 2.22507385850720138e-308

「DCFS、DCFSU」(P. 7-27) も参照して下さい。

例

```
DCFD    1E308,-4E-100
DCFDU   10000,-.1,3.1E26
```

7.3.9 DCFS、DCFSU

DCFS ディレクティブは、ワード境界で整列された単精度浮動小数点数にメモリを割り当て、ランタイム時のメモリの初期内容を定義します。単精度浮動小数点数には 1 ワードが使用されます。この 1 ワードを算術演算で使用するには、ワード境界で整列させる必要があります。

メモリの境界調整が任意である点を除き、DCDSU は DCFS と同じ意味です。

構文

{*label*} DCFS{U} *fpliteral*{, *fpliteral*}...

パラメータの説明：

fpliteral 単精度浮動小数点リテラルを指定します（「浮動小数点リテラル」(P. 3-30) 参照）。

使用法

DCFS は、必要に応じて、最初に定義された数値の前に最大 3 バイトのパディングを挿入して、境界調整を 4 バイトにします。

境界調整が不要な場合は、DCFSU を使用して下さい。

単精度数値の範囲は以下のとおりです。

- 最大 : 3.40282347e+38
- 最小 : 1.17549435e-38

「DCFD、DCFDU」(P. 7-26) も参照して下さい。

例

```
DCFS      1E3,-4E-9
DCFSU    1.0,-.1,3.1E6
```

7.3.10 DCI

ARM コードでは、DCI ディレクティブは、ワード単位でメモリを割り当てて、4 バイト境界で整列させ、ランタイム時のメモリの初期内容を定義します。

Thumb コードでは、DCI ディレクティブは、ハーフワード単位でメモリを割り当てて、2 バイト境界で整列させ、ランタイム時のメモリの初期内容を定義します。

構文

```
{label} DCI{.W} expr{,expr}
```

パラメータの説明：

expr 数値式を指定します（「数値式」（P. 3-28）参照）。

.W 指定されている場合は、Thumb コードに 4 バイトを挿入する必要があることを示します。

使用法

DCI ディレクティブは、DCD ディレクティブや DCW ディレクティブとよく似ていますが、メモリ位置はデータではなくコードとしてマークされます。使用しているアセンブラーのバージョンでサポートされていない新しい命令のマクロを記述する場合は、DCI を使用して下さい。

ARM コードでは、DCI は、必要に応じて、最初に定義されたワードの前に最大 3 バイトのパディングを挿入して、境界調整を 4 バイトにします。Thumb コードでは、DCI は、必要に応じて、先頭のバイトのパディングを挿入して、境界調整を 2 バイトにします。

DCI を使用して、ビットパターンを命令ストリームに挿入できます。例えば、以下を使用します。

```
DCI 0x46c0
```

Thumb 演算 MOV r8,r8 が挿入されます。

「DCD、DCDU」（P. 7-24）および「DCW、DCWU」（P. 7-30）も参照して下さい。

例

```
MACRO          ; this macro translates newinstr Rd,Rm
               ; to the appropriate machine code
newinst      $Rd,$Rm
DCI          0xe16f0f10 :OR: ($Rd:SHL:12) :OR:$Rm
MEND
```

Thumb-2 の例

```
DCI.W 0xf3af8000 ; inserts 32-bit NOP, 2-byte aligned.
```

7.3.11 DCQ、DCQU

DCQ ディレクティブは、8 バイト単位でメモリブロックを割り当てて、4 バイト境界で整列させ、ランタイム時のメモリの初期内容を定義します。

メモリの境界調整が任意である点を除き、DCQU は DCQ と同じ意味です。

構文

```
{label} DCQ{U} {-}literal{,{,-}literal}...
```

パラメータの説明：

literal 64 ビットの数値リテラルを指定します（「数値リテラル」(P. 3-29) 参照）。

この値の有効範囲は 0 ~ $2^{64}-1$ です。

数値リテラルで通常使用できる文字に加え、*literal* の先頭にマイナス符号を付けることができます。この場合の値の有効範囲は $-2^{63} \sim -1$ となります。

$-n$ を指定した場合は、 $2^{64}-n$ を指定した場合と同じ結果が得られます。

使用法

DCQ は、必要に応じて、最初に定義された 8 バイトのブロックの前に最大 3 バイトのpadding を挿入して、境界調整を 4 バイトにします。

境界調整が不要な場合は、DCQU を使用して下さい。

以下も参照して下さい。

- DCB (P. 7-23)
- DCD、DCDU (P. 7-24)
- DCW、DCWU (P. 7-30)
- SPACE (P. 7-22)

例

```
AREA   MiscData, DATA, READWRITE
data   DCQ    -225,2_101      ; 2_101 means binary 101.
          number+4        ; number must already be defined.
```

7.3.12 DCW、DCWU

DCW ディレクティブは、ハーフワード単位でメモリを割り当てて、2バイト境界で整列させ、ランタイム時のメモリの初期内容を定義します。

メモリの境界調整が任意である点を除き、**DCWU** は **DCW** と同じ意味です。

構文

```
{label} DCW{U} expr{,expr}...
```

パラメータの説明：

expr -32768 ~ 65535 の範囲の整数を求める数値式を指定します（「数値式」(P. 3-28) 参照）。

使用法

DCW は、必要に応じて、最初に定義されたハーフワードの前に 1 バイトのパディングを挿入して、境界調整を 2 バイトにします。

境界調整が不要な場合は、**DCWU** を使用して下さい。

以下も参照して下さい。

- *DCB* (P. 7-23)
- *DCD*、*DCDU* (P. 7-24)
- *DCQ*、*DCQU* (P. 7-29)
- *SPACE* (P. 7-22)

例

```
data    DCW    -225,2*number ; number must already be defined
```

```
          DCWU   number+4
```

7.3.13 COMMON

COMMON ディレクティブは、定義されたサイズのメモリブロックを、指定したシンボルで割り当てます。メモリの整列方法を指定します。境界調整を省略した場合、デフォルトの境界調整は 4 になります。また、サイズを省略した場合、デフォルトのサイズは 0 になります。

このメモリには他のメモリと同様にアクセスできますが、オブジェクトファイルに領域は割り当てられません。

構文

```
COMMON symbol{, size{, alignment}}
```

パラメータの説明：

symbol シンボル名を指定します。シンボル名では大文字と小文字が区別されます。

size 予約するバイト数を指定します。

alignment 境界調整を指定します。

使用法

リンクは、リンクステージ中に、必要な領域をゼロで初期化されたメモリとして割り当てます。

例

```
COMMON xyz,255,4 ; defines 255 bytes of ZI store, word-aligned
```

7.3.14 DATA

DATA ディレクティブは必要なくなりました。このディレクティブはアセンブラーによって無視されます。

7.4 アセンブリ制御ディレクティブ

このセクションでは、条件付きアセンブリ、ループ、インクルード、およびマクロを制御する以下のディレクティブについて説明します。

- *MACRO*, *MEND* (P. 7-33)
- *MEXIT* (P. 7-36)
- *IF*, *ELSE*, *ENDIF*, *ELIF* (P. 7-37)
- *WHILE*, *WEND* (P. 7-40)

7.4.1 ネスティングディレクティブ

以下の構造は合計で 256 の深さまでネストできます。

- *MACRO* 定義
- *WHILE...WEND* ループ[†]
- *IF...ELSE...ENDIF* 条件構造
- *INCLUDE* ファイルインクルード

これらの構造がどのようにネストされているかに関係なく、上記の制限はネストされた構造全体に適用されます。各構造の制限が 256 ではありません。

7.4.2 MACRO、MEND

MACRO ディレクティブは、マクロの定義の開始位置をマークします。マクロ拡張は、MEND ディレクティブで終了します。詳細については、「マクロの使用」(P. 2-43) を参照して下さい。

構文

マクロの定義には 2 つのディレクティブを使用します。構文は以下のとおりです。

```
MACRO
{$label}  macroname {$parameter{,$parameter}...}
; code
MEND
```

パラメータの説明：

\$label マクロが呼び出されたときに、指定されたシンボルが代入されるパラメータを指定します。通常、このシンボルはラベルです。

macroname マクロの名前を指定します。命令またはディレクティブの名前で始まる名前は付けられません。

\$parameter マクロが呼び出されたときに値が代入されるパラメータを指定します。パラメータのデフォルト値は、以下の形式を使用して設定できます。

\$parameter="default value"

デフォルト値にスペースが含まれているか、または値の前後のいずれかにスペースがある場合は、二重引用符を使用する必要があります。

使用法

マクロ内で WHILE...WEND ループまたは IF...ENDIF 条件を始める場合は、MEND ディレクティブに到達する前にこのループまたは条件を閉じる必要があります。ループ内からの終了時など、マクロからの早期終了を可能にする必要がある場合は、「*MEXIT*」(P. 7-36) を参照して下さい。

マクロボディ内では、\$label や \$parameter などのパラメータは他の変数と同じように使用できます（「アセンブリ時の変数代入」(P. 3-22) 参照）。これらのパラメータには、マクロが呼び出されるたびに新しい値が渡されます。パラメータを通常のシンボルと区別するには、先頭に \$ を付ける必要があります。パラメータはいくつでも使用できます。

\$label は省略可能ですが、マクロによって内部ラベルが定義される場合に役立ちます。このラベルは、マクロへのパラメータとして処理されますが、必ずしも、マクロ拡張の最初の命令を表す必要はありません。マクロでは、すべてのラベルの位置が定義されます。

パラメータのデフォルト値を使用するには、|を引数として使用します。この引数が省略されている場合は、空の文字列が使用されます。

複数の内部ラベルを使用するマクロでは、各内部ラベルを、異なる接尾文字の付いたベースラベルとして定義すると役立ちます。

拡張にスペースが不要な場合には、パラメータとそれに続くテキストの間、またはパラメータとパラメータの間にはドット(.)を使用します。ただし、パラメータとその前のテキストの間にはドットは使用しないで下さい。

マクロでは、ローカル変数の有効範囲を定義します(*/LCLA,LCLL,LCLS/*(P. 7-7)参照)。

マクロはネストできます(「ネスティングディレクティブ」(P. 7-32) 参照)。

例

```
; macro definition
      MACRO          ; start macro definition
$label    xmac   $p1,$p2
          ; code
$label.loop1  ; code
          ; code
          BGE    $label.loop1
$label.loop2  ; code
          BL     $p1
          BGT    $label.loop2
          ; code
          ADR    $p2
          ; code
          MEND   ; end macro definition

; macro invocation
abc      xmac   subr1,de   ; invoke macro
          ; code
          ; code
          ; code
          BGE    abcloop1  ; this is what is
          ; is produced when
          ; the xmac macro is
          ; expanded
abcloop2  ; code
          BL     subr1
          BGT    abcloop2
          ; code
          ADR    de
          ; code
```

マクロを使用したアセンブリ時の診断を以下に示します。

```
MACRO          ; Macro definition
diagnose $param1"default" ; This macro produces
INFO      0,"$param1"       ; assembly-time diagnostics
MEND          ; (on second assembly pass)

; macro expansion

diagnose        ; Prints blank line at assembly-time
diagnose "hello"   ; Prints "hello" at assembly-time
diagnose |         ; Prints "default" at assembly-time
```

7.4.3 MEXIT

MEXIT ディレクティブは、マクロ定義の終了位置に到達する前にマクロを終了する場合に使用します。

使用法

MEXIT は、マクロ本体の中から終了する必要がある場合に使用します。マクロ本体の中で閉じられていない WHILE...WEND ループまたは IF...ENDIF 条件は、マクロが終了する前にアセンブラーによって閉じられます。

「MACRO、MEND」(P. 7-33) も参照して下さい。

例

```
MACRO
$abc    macro abc      $param1,$param2
        ; code
        WHILE condition1
            ; code
            IF condition2
                ; code
                MEXIT
            ELSE
                ; code
            ENDIF
        WEND
        ; code
MEND
```

7.4.4 IF、ELSE、ENDIF、ELIF

IF ディレクティブは、命令またはディレクティブ、あるいはその両方のシーケンスをアセンブルするかどうかを決める条件を定義します。[は IF と同じ意味です。

ELSE ディレクティブは、前の条件が満たされなかった場合にアセンブルされる命令またはディレクティブ、あるいはその両方のシーケンスの開始位置をマークします。| は ELSE と同じ意味です。

ENDIF ディレクティブは、条件付きでアセンブルされる命令またはディレクティブ、あるいはその両方のシーケンスの終了位置をマークします。] は ENDIF と同じ意味です。

ELIF ディレクティブは、条件をネストまたは反復することなく、ELSE IF と同じ構造を作成します。詳細については、「/ELIF の使用」(P. 7-38) を参照して下さい。

構文

```
IF logical-expression      ...      {ELSE      ...}      ENDIF
```

パラメータの説明 :

logical-expression

{TRUE} または {FALSE} を求める式を指定します。

詳細については、「関係演算子」(P. 3-38) を参照して下さい。

使用法

指定された条件下においてのみアセンブルや実行される命令またはディレクティブ、あるいはその両方のシーケンスには、IF と ENDIF に加え、必要に応じて ELSE を組み合わせて使用します。

IF...ENDIF 条件はネストできます（「ネスティングディレクティブ」(P. 7-32) 参照）。

ELIF の使用

ELIF を使用せずに、以下のようなネストされた条件付き命令セットを作成できます。

```
IF logical-expression
    instructions
ELSE
    IF logical-expression2
        instructions
    ELSE
        IF logical-expression3
            instructions
        ENDIF
    ENDIF
ENDIF
```

このようなネスト構造は、256 の深さまでネストできます。

ELIF を使用すると、同じ構造をより簡単に記述できます。

```
IF logical-expression
    instructions
ELIF logical-expression2
    instructions
ELIF logical-expression3
    instructions
ENDIF
```

この構造では、現在のネストの深さに IF...ENDIF の 1 レベルだけが追加された深さになります。

例

例 7-3 では、`NEVERVERSION` が定義されている場合は最初の命令群がアセンブルされ、定義されていない場合は 2 番目の命令群がアセンブルされます。

例 7-3 定義される変数に基づく条件付きアセンブリ

```
IF :DEF:NEVERVERSION
    ; first set of instructions/directives
ELSE
    ; alternative set of instructions/directives
ENDIF
```

`armasm` を以下のように呼び出した場合には `NEVERVERSION` が定義されるため、最初の命令群とディレクティブ群がアセンブルされます。

`armasm --predefine "NEVERVERSION SETL {TRUE}" test.s`

`armasm` を以下のように呼び出した場合には `NEVERVERSION` が定義されていないため、2 番目の命令群とディレクティブ群がアセンブルされます。

`armasm test.s`

例 7-4 では、`NEVERVERSION` の値が `{TRUE}` の場合に最初の命令群がアセンブルされ、それ以外の場合は 2 番目の命令群がアセンブルされます。

例 7-4 変数の値に基づく条件付きアセンブリ

```
IF NEVERVERSION = {TRUE}
    ; first set of instructions/directives
ELSE
    ; alternative set of instructions/directives
ENDIF
```

`armasm` を以下のように呼び出した場合には、最初の命令群とディレクティブ群がアセンブルされます。

`armasm --predefine "NEVERVERSION SETL {TRUE}" test.s`

`armasm` を以下のように呼び出した場合には、2 番目の命令群とディレクティブ群がアセンブルされます。

`armasm --predefine "NEVERVERSION SETL {FALSE}" test.s`

7.4.5 WHILE、WEND

WHILE ディレクティブは、繰り返しアセンブルされる命令やディレクティブのシーケンスを開始します。このシーケンスは WEND ディレクティブで終了します。

構文

`WHILE logical-expression`

`code`

`WEND`

パラメータの説明：

logical-expression

{TRUE} または {FALSE} を求める式を指定します（「論理式」(P. 3-31) 参照）。

使用法

命令のシーケンスを複数回アセンブルするには、WHILE ディレクティブを WEND ディレクティブと組み合わせて使用します。反復回数はゼロにすることもできます。

IF...ENDIF 条件は WHILE...WEND ループ内で使用できます。

WHILE...WEND ループはネストできます（「ネスティングディレクティブ」(P. 7-32) 参照）。

例

```
count  SETA   1           ; you are not restricted to
                  WHILE  count <= 4       ; such simple conditions
count  SETA   count+1     ; In this case,
                  ; code          ; this code will be
                  ; code          ; repeated four times
                  WEND
```

7.5 フレームディレクティブ

このセクションでは、以下のディレクティブについて説明します。

- *FRAME ADDRESS* (P. 7-43)
- *FRAME POP* (P. 7-44)
- *FRAME PUSH* (P. 7-45)
- *FRAME REGISTER* (P. 7-47)
- *FRAME RESTORE* (P. 7-48)
- *FRAME RETURN ADDRESS* (P. 7-49)
- *FRAME SAVE* (P. 7-50)
- *FRAME STATE REMEMBER* (P. 7-51)
- *FRAME STATE RESTORE* (P. 7-52)
- *FRAME UNWIND ON* (P. 7-53)
- *FRAME UNWIND OFF* (P. 7-53)
- *FUNCTION, PROC* (P. 7-54)
- *ENDFUNC, ENDP* (P. 7-55)

上記のディレクティブを使用することにより、以下の処理を行うことができます。

- `armlink --callgraph` オプションを使用して、アセンブラー関数によるスタック使用量を計算できます。
スタック使用量の特定には、以下のルールが使用されます。
 - 関数が *PROC* または *ENDP* でマークされていない場合、スタック使用量は特定できません。
 - 関数が *PROC* または *ENDP* でマークされていても *FRAME PUSH* または *FRAME POP* でマークされていない場合、スタック使用量はゼロであると見なされます。つまり、*FRAME PUSH 0* または *FRAME POP 0* を手動で追加する必要はありません。
 - 関数が *PROC* または *ENDP* でマークされ、かつ *FRAME PUSH n* または *FRAME POP n* でマークされている場合、スタック使用量は *n* バイトと見なされます。
- 特に既存のコードを変更する場合に、関数の構造内でのエラーを回避できます。
- アセンブラーが関数の構造内のエラーについて警告メッセージを生成できます。
- デバッグ中に、関数呼び出しのバックトレースを行うことができます。
- デバッガを使用して、アセンブラー関数のプロファイルを取得することができます。

アセンブラー関数のプロファイルを必要としていても、他の目的にフレームディレクティブを必要としない場合、以下のようになります。

- `FUNCTION` ディレクティブと `ENDFUNC` ディレクティブ、または `PROC` ディレクティブと `ENDP` ディレクティブを使用する必要があります。
- 他の `FRAME` ディレクティブは省略できます。
- `FUNCTION` ディレクティブと `ENDFUNC` ディレクティブは、プロファイルの取得の対象となる関数に対してのみ使用する必要があります。

DWARF での標準構造フレームアドレスは、割り込み関数のコールフレームがある場所を指定する、スタック上のアドレスです。

7.5.1 FRAME ADDRESS

FRAME ADDRESS ディレクティブは、後続の命令の標準構造フレームアドレスの計算方法を記述します。このディレクティブは、FUNCTION ディレクティブと ENDFUNC ディレクティブを含む関数または PROC ディレクティブと ENDP ディレクティブを含む関数でのみ使用できます。

構文

FRAME ADDRESS *reg[,offset]*

パラメータの説明：

reg 標準構造フレームアドレスのベースとなるレジスタを指定します。関数で別のフレームポインタを使用しない限り、このレジスタが sp になります。

offset *reg* から標準構造フレームアドレスまでのオフセットを指定します。*offset* がゼロの場合は省略できます。

使用法

FRAME ADDRESS ディレクティブは、コードによって標準構造フレームアドレスのベースとなるレジスタが変更される場合またはレジスタから標準構造フレームアドレスまでのオフセットが変更される場合に使用します。FRAME ADDRESS ディレクティブは、標準構造フレームアドレスの計算方法を変更する命令の直後に使用する必要があります。

注

コードでレジスタを保存してスタックポインタを変更する1つの命令を使用する場合、FRAME ADDRESS と FRAME SAVE の両方を使用する代わりに FRAME PUSH を使用できます（「FRAME PUSH」（P. 7-45）参照）。

また、コードでレジスタをロードしてスタックポインタを変更する1つの命令を使用する場合は、FRAME ADDRESS と FRAME RESTORE の両方を使用する代わりに FRAME POP を使用できます（「FRAME POP」（P. 7-44）参照）。

例

```
_fn      FUNCTION          ; CFA (Canonical Frame Address) is value
        ; of sp on entry to function
        PUSH    {r4,fp,ip,lr,pc}
        FRAME PUSH {r4,fp,ip,lr,pc}
        SUB    sp,sp,#4           ; CFA offset now changed
        FRAME ADDRESS sp,24       ; - so we correct it
        ADD    fp,sp,#20
        FRAME ADDRESS fp,4         ; New base register
        ; code using fp to base call-frame on, instead of sp
```

7.5.2 FRAME POP

FRAME POP ディレクティブを使用して、被発呼側がレジスタをリロードするタイミングをアセンブラーに通知します。このディレクティブは、FUNCTION ディレクティブと ENDFUNC ディレクティブを含む関数または PROC ディレクティブと ENDP ディレクティブを含む関数でのみ使用できます。

このディレクティブは、関数内の最後の命令の後で必ず使用する必要があるものではありません。

構文

FRAME POP には以下の 3 つの構文があります。

FRAME POP {reglist}

FRAME POP {reglist},n

FRAME POP n

パラメータの説明 :

reglist 関数へのエントリで保持していた値に復元されるレジスタリストを指定します。リストには少なくとも 1 つのレジスタを指定する必要があります。

n スタックポインタが移動するバイト数を指定します。

使用法

FRAME POP は、FRAME ADDRESS ディレクティブおよび FRAME RESTORE ディレクティブを使用することと同じです。このディレクティブを使用すると、1 つの命令でレジスタをロードし、スタックポインタを変更することができます。

FRAME POP は、このディレクティブが参照する命令の直後に配置する必要があります。

n が指定されていない場合やゼロの場合、アセンブラーによって、{*reglist*} から標準構造フレームアドレスまでの新しいオフセットが計算されます。アセンブラーは以下を前提としています。

- ポップされる各 ARM レジスタによってスタック上の 4 バイトが占有されています。
- ポップされる各 VFP 単精度レジスタによってスタック上の 4 バイトが占有され、さらにリストごとに 4 バイトワードが占有されています。
- ポップされる各 VFP 倍精度レジスタによってスタック上の 8 バイトが占有され、さらにリストごとに 4 バイトワードが占有されています。

詳細については、「FRAME ADDRESS」(P. 7-43) および「FRAME RESTORE」(P. 7-48) を参照して下さい。

7.5.3 FRAME PUSH

FRAME PUSH ディレクティブを使用して、通常は関数エントリにおいて、被発呼側がレジスタを保存するタイミングをアセンブラーに通知します。このディレクティブは、FUNCTION ディレクティブと ENDFUNC ディレクティブを含む関数または PROC ディレクティブと ENDP ディレクティブを含む関数内でのみ使用できます。

構文

FRAME PUSH には以下の 3 つの構文があります。

FRAME PUSH {reglist}

FRAME PUSH {reglist},n

FRAME PUSH n

パラメータの説明 :

reglist 標準構造フレームアドレスの下位方向に連続してストアされるレジスタのリストを指定します。リストには少なくとも 1 つのレジスタを指定する必要があります。

n スタックポインタが移動するバイト数を指定します。

使用法

FRAME PUSH は、FRAME ADDRESS ディレクティブおよび FRAME SAVE ディレクティブを使用することと同じです。このディレクティブを使用すると、1 つの命令でレジスタを保存し、スタックポインタを変更することができます。

FRAME PUSH は、このディレクティブが参照する命令の直後に配置する必要があります。

n が指定されていない場合やゼロの場合、アセンブラーによって、{*reglist*} から標準構造フレームアドレスまでの新しいオフセットが計算されます。アセンブラーは以下を前提としています。

- プッシュされる各 ARM レジスタによってスタック上の 4 バイトが占有されています。
- プッシュされる各 VFP 単精度レジスタによってスタック上の 4 バイトが占有され、さらにリストごとに 4 バイトワードが占有されています。
- ポップされる各 VFP 倍精度レジスタによってスタック上の 8 バイトが占有され、さらにリストごとに 4 バイトワードが占有されています。

詳細については、「FRAME ADDRESS」(P. 7-43) および「FRAME SAVE」(P. 7-50) を参照して下さい。

例

```
p PROC ; Canonical frame address is sp + 0
EXPORT p
PUSH {r4-r6,lr}
; sp has moved relative to the canonical frame address,
; and registers r4, r5, r6 and lr are now on the stack
FRAME PUSH {r4-r6,lr}
; Equivalent to:
; FRAME ADDRESS sp,16      ; 16 bytes in {r4-r6,lr}
; FRAME SAVE   {r4-r6,lr},-16
```

7.5.4 FRAME REGISTER

FRAME REGISTER ディレクティブを使用して、レジスタに保持されている関数引数の位置を管理します。このディレクティブは、FUNCTION ディレクティブと ENDFUNC ディレクティブを含む関数または PROC ディレクティブと ENDP ディレクティブを含む関数内でのみ使用できます。

構文

FRAME REGISTER *reg1, reg2*

パラメータの説明：

reg1 関数へのエントリで引数を保持するレジスタを指定します。

reg2 その値を保持するレジスタを指定します。

使用法

FRAME REGISTER ディレクティブは、関数へのエントリで別のレジスタに保持されていた引数を保持するレジスタを使用する場合に使用します。

7.5.5 FRAME RESTORE

FRAME RESTORE ディレクティブを使用して、指定されたレジスタの内容が、関数へのエントリで保持されていた値に復元されたことをアセンブラーに通知します。このディレクティブは、FUNCTION ディレクティブと ENDFUNC ディレクティブを含む関数または PROC ディレクティブと ENDP ディレクティブを含む関数でのみ使用できます。

構文

```
FRAME RESTORE {reglist}
```

パラメータの説明：

reglist 内容が復元されたレジスタのリストを指定します。リストには少なくとも 1 つのレジスタを指定する必要があります。

使用法

FRAME RESTORE は、被発呼側がスタックからレジスタをリロードした直後に使用します。このディレクティブは、関数内の最後の命令の後で必ず使用する必要があるものではありません。

reglist には、整数レジスタまたは浮動小数点レジスタのいずれかを指定できますが、両方を混在させることはできません。

—— 注 ——

コードでレジスタをロードしてスタックポインタを変更する 1 つの命令を使用する場合は、FRAME RESTORE と FRAME ADDRESS の両方を使用する代わりに FRAME POP を使用できます（「FRAME POP」(P. 7-44) 参照）。

7.5.6 FRAME RETURN ADDRESS

FRAME RETURN ADDRESS ディレクティブによって、r14 以外のレジスタを復帰アドレスに使用する関数を定義できます。このディレクティブは、FUNCTION ディレクティブと ENDFUNC ディレクティブを含む関数または PROC ディレクティブと ENDP ディレクティブを含む関数でのみ使用できます。

——注——

r14 以外のレジスタを復帰アドレスに使用する関数は AAPCS に準拠していません。このような関数はエクスポートしないで下さい。

構文

FRAME RETURN ADDRESS *reg*

パラメータの説明：

reg 復帰アドレスに使用するレジスタを指定します。

使用法

FRAME RETURN ADDRESS ディレクティブは、復帰アドレスに r14 を使用しない関数で使用します。このディレクティブを使用しないと、デバッガはその関数のバックトレースを行うことができません。

FRAME RETURN ADDRESS は、r14 を使用しない関数の開始位置をマークする FUNCTION ディレクティブまたは PROC ディレクティブの直後に使用します。

7.5.7 FRAME SAVE

FRAME SAVE ディレクティブは、標準構造フレームアドレスからの相対位置に保存されるレジスタの位置を記述します。このディレクティブは、FUNCTION ディレクティブと ENDFUNC ディレクティブを含む関数または PROC ディレクティブと ENDP ディレクティブを含む関数でのみ使用できます。

構文

```
FRAME SAVE {reglist}, offset
```

パラメータの説明：

reglist 標準構造フレームアドレスからの *offset* から連続してストアされるレジスタのリストを指定します。リストには少なくとも 1 つのレジスタを指定する必要があります。

使用法

FRAME SAVE は、被発呼側がスタックにレジスタをストアした直後に使用します。

reglist には、バックトレースに不要なレジスタを含めることができます。DWARF ユールフレーム情報として記録する必要のあるレジスタはアセンブラーによって決定されます。

—— 注 ——

コードでレジスタを保存してスタックポインタを変更する 1 つの命令を使用する場合、FRAME SAVE と FRAME ADDRESS の両方を使用する代わりに FRAME PUSH を使用できます（/FRAME PUSH/ (P. 7-45) 参照）。

7.5.8 FRAME STATE REMEMBER

FRAME STATE REMEMBER ディレクティブは、標準構造フレームアドレスと、保存されるレジスタ値の位置の計算方法に関する現在の情報を保存します。このディレクティブは、FUNCTION ディレクティブと ENDFUNC ディレクティブを含む関数または PROC ディレクティブと ENDP ディレクティブを含む関数でのみ使用できます。

構文

FRAME STATE REMEMBER

使用法

インラインの終了シーケンスでは、標準構造フレームアドレスと、保存されるレジスタ値の位置の計算に関する情報が変更される場合があります。終了シーケンスの後、以前と同じ情報を使用して別の分岐を継続できます。この情報を保存するには FRAME STATE REMEMBER を使用し、この情報を復元するには FRAME STATE RESTORE を使用します。

これらのディレクティブはネストできます。各 FRAME STATE RESTORE ディレクティブには、対応する FRAME STATE REMEMBER ディレクティブが必要です。詳細については、以下を参照して下さい。

- *FRAME STATE RESTORE* (P. 7-52)
- *FUNCTION, PROC* (P. 7-54)

例

```
; function code
FRAME STATE REMEMBER
    ; save frame state before in-line exit sequence
    POP    {r4-r6,pc}
        ; do not have to FRAME POP here, as control has
        ; transferred out of the function
    FRAME STATE RESTORE
        ; end of exit sequence, so restore state
exitB   ; code for exitB
    POP    {r4-r6,pc}
ENDP
```

7.5.9 FRAME STATE RESTORE

FRAME STATE RESTORE ディレクティブは、標準構造フレームアドレスと、保存されるレジスタ値の位置の計算方法に関する現在の情報を復元します。このディレクティブは、FUNCTION ディレクティブと ENDFUNC ディレクティブを含む関数または PROC ディレクティブと ENDP ディレクティブを含む関数でのみ使用できます。

構文

FRAME STATE RESTORE

使用法

以下を参照して下さい。

- *FRAME STATE REMEMBER* (P. 7-51)
- *FUNCTION, PROC* (P. 7-54)

7.5.10 FRAME UNWIND ON

FRAME UNWIND ON ディレクティブは、この関数とその後に続く関数の *unwind* テーブルを生成するようアセンブラーに指示します。

構文

FRAME UNWIND ON

使用法

このディレクティブは関数の外部で使用できます。この場合のアセンブラーは、FRAME UNWIND OFF ディレクティブに到達するまで、後続のすべての関数の *unwind* テーブルを生成します。

「例外テーブル生成を制御する」(P. 3-15) も参照して下さい。

7.5.11 FRAME UNWIND OFF

FRAME UNWIND OFF ディレクティブは、この関数と後に続く関数の *nounwind* テーブルを生成するようアセンブラーに指示します。

構文

FRAME UNWIND OFF

使用法

このディレクティブは関数の外部で使用できます。この場合のアセンブラーは、FRAME UNWIND ON ディレクティブに到達するまで、後続のすべての関数の *nounwind* テーブルを生成します。

「例外テーブル生成を制御する」(P. 3-15) も参照して下さい。

7.5.12 FUNCTION、PROC

FUNCTION ディレクティブは、関数の開始位置をマークします。PROC は FUNCTION と同じ意味です。

構文

```
label FUNCTION [{reglist1} [, {reglist2}]]
```

パラメータの説明：

reglist1 被発呼側によって保存される ARM レジスタのリストを指定できます（省略可）。*reglist1* が指定されていない場合、デバッガはレジスタの使用状況をチェックするときに AAPCS が使用されていると見なします。

reglist2 被発呼側によって保存される VFP レジスタのリストを指定できます（省略可）。

使用法

FUNCTION ディレクティブを使用して関数の開始位置をマークします。ELF の DWARF コールフレーム情報を生成するとき、アセンブラーは FUNCTION を使用して関数の開始位置を識別します。

FUNCTION によって標準構造フレームアドレスは r13 (sp) に設定され、フレームのスタックは空になります。

各 FUNCTION ディレクティブには、対応する ENDFUNC ディレクティブが必要です。FUNCTION/ENDFUNC の対をネストしたり、これらの対に PROC ディレクティブや ENDP ディレクティブを含めたりすることはできません。

別のプロシージャコール標準を使用している場合は、オプションの *reglist* パラメータを使用して、その情報をデバッガに通知できます。ただし、すべてのデバッガがこの機能をサポートしているわけではありません。詳細については、デバッガのマニュアルを参照して下さい。

また、「FRAME ADDRESS」(P. 7-43) ~ 「FRAME STATE RESTORE」(P. 7-52) も参照して下さい。

注

FUNCTION では、ワード境界（Thumb の場合はハーフワード境界）への境界調整は自動的には行われません。境界調整を行う必要がある場合は ALIGN を使用します。このディレクティブを使用しないと、コールフレームによって関数の開始位置が示されない場合があります。詳細については、「ALIGN」(P. 7-64) を参照して下さい。

例

```
ALIGN      ; ensures alignment
dadd      FUNCTION ; without the ALIGN directive, this might not be word-aligned
          EXPORT dadd
          PUSH   {r4-r6,lr}    ; this line automatically word-aligned
          FRAME PUSH {r4-r6,lr}
          ; subroutine body
          POP    {r4-r6,pc}
ENDFUNC

func6    PROC {r4-r8,r12},{D1-D3} ; non-AAPCS-conforming function
...
ENDP
```

7.5.13 ENDFUNC、ENDP

ENDFUNC ディレクティブは、AAPCS 準拠の関数の終了位置をマークします (*/FUNCTION, PROC/* (P. 7-54) 参照)。ENDP は ENDFUNC と同じ意味です。

7.6 通知ディレクティブ

このセクションでは、以下のディレクティブについて説明します。

- *ASSERT*
アセンブリ中にアサーションが偽になる場合にエラーメッセージを生成します。
- *INFO* (P. 7-57)
アセンブリ中に診断情報を生成します。
- *OPT* (P. 7-58)
リストオプションを設定します。
- *TTL*, *SUBT* (P. 7-60)
リストにタイトルとサブタイトルを挿入します。

7.6.1 ASSERT

ASSERT ディレクティブは、指定されたアサーションが偽になると、アセンブリの第 2 パスでエラーメッセージを生成します。

構文

ASSERT logical-expression

パラメータの説明：

logical-expression

{TRUE} または {FALSE} を返すアサーションを指定します。

使用法

ASSERT を使用して、必要な条件がアセンブリ中に満たされているかどうかを確認します。

アサーションが偽の場合、エラーメッセージが生成され、アセンブルに失敗します。

/INFO/ (P. 7-57) も参照して下さい。

例

```
ASSERT label1 <= label2 ; Tests if the address
; represented by label1
; is <= the address
; represented by label2.
```

7.6.2 INFO

INFO ディレクティブは、アセンブルのどちらのパスに関する診断情報の生成もサポートします。

!は INFO とよく似ていますが、通知される情報の詳細度は低くなります。

構文

`INFO numeric-expression, string-expression`

パラメータの説明：

numeric-expression

アセンブリ時に評価される数値式を指定します。この式の結果がゼロになった場合、以下のようになります。

- 第 1 パスでは何も行われません。
- 第 2 パスでは *string-expression* が出力されます。

この式の結果がゼロ以外の場合は、*string-expression* がエラーメッセージとして出力され、アセンブルに失敗します。

string-expression

文字列を返す式を指定します。

使用法

INFO ディレクティブを使用すると、独自のエラーメッセージを柔軟に作成することができます。数値式と文字列式に関する追加情報については、「数値式」(P. 3-28) および「文字列式」(P. 3-27) を参照して下さい。

「ASSERT」(P. 7-56) も参照して下さい。

例

```
INFO    0, "Version 1.0"

IF endofdata <= label1
    INFO    4, "Data overrun at label1"
ENDIF
```

7.6.3 OPT

OPT ディレクティブは、ソースコード内からリストオプションを設定します。

構文

`OPT n`

パラメータの説明：

- `n` OPT ディレクティブの設定を指定します。表 7-2 は、利用可能な設定を示しています。

表 7-2 OPT ディレクティブの設定

OPT n	効果
1	通常のリストイングが有効になります。
2	通常のリストイングが無効になります。
4	ページ区切りが行われます。すぐに改ページが行われ、新しいページが始まります。
8	行番号カウンタがゼロにリセットされます。
16	SET、GBL、およびLCL ディレクティブのリストイングが有効になります。
32	SET、GBL、およびLCL ディレクティブのリストイングが無効になります。
64	マクロ拡張のリストイングが有効になります。
128	マクロ拡張のリストイングが無効になります。
256	マクロ呼び出しのリストイングが有効になります。
512	マクロ呼び出しのリストイングが無効になります。
1024	第1パスのリストイングが有効になります。
2048	第1パスのリストイングが無効になります。
4096	条件ディレクティブのリストイングが有効になります。
8192	条件ディレクティブのリストイングが無効になります。
16384	MEND ディレクティブのリストイングが有効になります。
32768	MEND ディレクティブのリストイングが無効になります。

使用法

リストイングを有効にするには、**--list=** アセンブラーオプションを指定します。

デフォルトでは、**--list=** オプションを指定すると、変数宣言、マクロ拡張、呼び出し条件ディレクティブ、MEND ディレクティブを含む通常のリストが生成されます。このリストは第 2 パスのみで生成されます。コード内でデフォルトのリストオプションを修正するには、OPT ディレクティブを使用します。**--list=** オプションについては、「リストをファイルに出力する」(P. 3-12) を参照して下さい。

OPT ディレクティブを使用して、コードのリストのフォーマットを設定できます。例えば、関数とセクションの前に新しいページを指定できます。

例

```
AREA Example, CODE, READONLY
start ; code
      ; code
      BL    func1
      ; code
      OPT 4           ; places a page break before func1
func1  ; code
```

7.6.4 TTL、SUBT

TTL ディレクティブは、リストイングファイルの各ページの先頭にタイトルを挿入します。このタイトルは、新しい TTL ディレクティブが発行されるまで各ページに出力されます。

SUBT ディレクティブは、リストイングファイルのページにサブタイトルを挿入します。このサブタイトルは、新しい SUBT ディレクティブが発行されるまで各ページに出力されます。

構文

`TTL title`

`SUBT subtitle`

パラメータの説明：

`title` タイトルを指定します。

`subtitle` サブタイトルを指定します。

使用法

TTL ディレクティブを使用して、リストイングファイルの各ページの最上部にタイトルを挿入します。最初のページにタイトルを出力する場合は、ソースファイルの 1 行目に TTL ディレクティブを配置する必要があります。

タイトルを変更するには、別の TTL ディレクティブを使用します。新しい TTL ディレクティブの設定内容は、次のページから反映されます。

SUBT ディレクティブを使用して、リストイングファイルのページの最上部にサブタイトルを挿入します。サブタイトルは、タイトルの次の行に出力されます。最初のページにサブタイトルを出力する場合は、ソースファイルの 1 行目に SUBT ディレクティブを配置する必要があります。

サブタイトルを変更するには、別の SUBT ディレクティブを使用します。新しい SUBT ディレクティブの設定内容は、次のページから反映されます。

例

```

TTL    First Title   ; places a title on the first
          ; and subsequent pages of a
          ; listing file.
SUBT   First Subtitle ; places a subtitle on the
          ; second and subsequent pages
          ; of a listing file.

```

7.7 命令セットと構文選択のディレクティブ

このセクションでは、以下のディレクティブについて説明します。

- *ARM*, *THUMB*, *THUMBX*, *CODE16*, *CODE32* (P. 7-62)

7.7.1 ARM、THUMB、THUMBX、CODE16、CODE32

ARM ディレクティブと CODE32 ディレクティブは同義語です。これらのディレクティブは、後続の命令を ARM 命令として解釈するようにアセンブラーに指定します。次のワード境界で整列させるために、必要に応じて、最大 3 バイトのパディングを挿入します。

このモードのアセンブラーでは、最新のアセンブリ言語と以前のバージョンのアセンブリ言語の両方を使用できます。

構文

```
ARM
THUMB
THUMBX
CODE16
CODE32
```

使用法

異なる命令セットを使用したコードを含むファイルでは、以下のようになります。

- ARM は ARM コードの前に配置する必要があります。CODE32 は ARM と同じ意味です。
- THUMB は、新しい構文で記述された Thumb コードの前に配置する必要があります。
- THUMBX は、新しい構文で記述された Thumb-2EE コードの前に配置する必要があります。
- CODE16 は、古い構文で記述された Thumb コードの前に配置する必要があります。

これらのディレクティブをアセンブルすることで、状態を変更する命令が生成されるわけではありません。これらのディレクティブは、ARM、Thumb-2、Thumb-2EE、または Thumb の命令を適切にアセンブルするようアセンブラーに指定し、必要に応じてパディングを挿入するだけです。

例

この例では、ARM および CODE16 を使用して ARM 命令から 16 ビット Thumb 命令に分岐する方法を示します。

```
AREA ToThumb, CODE, READONLY      ; Name this block of code
ENTRY                           ; Mark first instruction to execute
ARM                             ; Subsequent instructions are ARM
start
    ADR    r0, into_thumb + 1     ; Processor starts in ARM state
    BX     r0                     ; Inline switch to Thumb state

    THUMB                         ; Subsequent instructions are Thumb
into_thumb
    MOVS   r0, #10                ; New-style Thumb instructions
```

7.8 その他のディレクティブ

このセクションでは、以下のディレクティブについて説明します。

- *ALIGN* (P. 7-64)
- *AREA* (P. 7-66)
- *END* (P. 7-68)
- *ENTRY* (P. 7-69)
- *EQU* (P. 7-70)
- *EXPORT*, *GLOBAL* (P. 7-71)
- *EXPORTAS* (P. 7-73)
- *EXTERN* (P. 7-74)
- *GET*, *INCLUDE* (P. 7-76)
- *IMPORT* (P. 7-77)
- *INCBIN* (P. 7-79)
- *KEEP* (P. 7-80)
- *NOFP* (P. 7-81)
- *REQUIRE* (P. 7-81)
- *REQUIRE8*, *PRESERVE8* (P. 7-82)
- *ROUT* (P. 7-84)

7.8.1 ALIGN

ALIGN ディレクティブは、ゼロでパディングすることにより、現在の位置を指定された境界で整列させます。

構文

```
ALIGN {expr{,offset{,pad }}}}
```

パラメータの説明：

expr $2^0 \sim 2^{31}$ の 2 の累乗を返す数値式を指定します。

offset 任意の数値式を指定できます。

pad 任意の数値式を指定できます。

演算

現在の位置は、以下の形式の次のアドレスで整列されます。

*offset + n * expr*

expr が指定されていない場合、ALIGN によって現在の位置に次のワード（4 バイト）境界が設定されます。

以前の位置と現在の新しい位置との間にある未使用的バイトは、*pad* の最下位バイトで埋められるか、*pad* が指定されていない場合はゼロで埋められます。

使用法

ALIGN を使用して、データとコードを適切な境界で整列させます。このディレクティブは、一般的に以下の状況で必要となります。

- ADR Thumb 擬似命令は、ワード境界で整列されたアドレスしかロードできませんが、Thumb コード内のラベルはワード境界で整列されていない場合があります。このような場合は、ALIGN 4 を使用して、Thumb コード内のアドレスを 4 バイト境界で境界調整させることができます。
- ALIGN を使用すると、一部の ARM プロセッサに搭載されているキャッシュを利用することができます。例えば、ARM940T には、16 バイトのラインから構成されるキャッシュがあります。このような場合は、ALIGN 16 を使用して、関数エントリを 16 バイト境界で整列させてキャッシュの効果を最大限に高めることができます。

- LDRD および STRD のダブルワードデータ転送命令は、8 バイト境界で整列させる必要があります。LDRD または STRD を使用してデータにアクセスする場合には、DCQ（「データ定義ディレクティブ」(P. 7-17) 参照）などのメモリ割り当てディレクティブの前に ALIGN 8 を使用します。
- ラベルだけの行は任意の境界で整列されます。それに続く ARM コードはワード境界で整列されます (Thumb コードはハーフワード境界で整列されます)。そのため、このラベルではコードが正しくアドレス指定されません。このような場合は、ラベルの前に ALIGN 4 (Thumb の場合は ALIGN 2) を使用します。

境界調整は、ルーチンが配置される ELF セクションの開始位置から相対的に行われます。ELF セクションは、同じ境界またはそれよりも粗い境界で整列させる必要があります。AREA ディレクティブの ALIGN 属性は異なる方法で指定されます (AREA (P. 7-66) および例参照)。

例

```

AREA cacheable, CODE, ALIGN=3
rout1 ; code ; aligned on 8-byte boundary
        ; code
        MOV pc,lr ; aligned only on 4-byte boundary
        ALIGN 8 ; now aligned on 8-byte boundary
rout2 ; code

AREA OffsetExample, CODE
DCB 1 ; This example places the two
ALIGN 4,3 ; bytes in the first and fourth
DCB 1 ; bytes of the same word.

AREA Example, CODE, READONLY
start LDR r6,label1
        ; code
        MOV pc,lr
label1 DCB 1 ; pc now misaligned
        ALIGN ; ensures that subroutine1 addresses
subroutine1 ; the following instruction.
        MOV r5,#0x5

```

7.8.2 AREA

AREA ディレクティブは、新しいコードセクションまたはデータセクションをアセンブルするようアセンブラーに指定します。これらのセクションは、リンクによって操作される、名前付きで独立した分割不可能なコードまたはデータのかたまりです。詳細については、「ELF セクションと AREA ディレクティブ」(P. 2-14) を参照して下さい。

構文

```
AREA sectionname{,attr}{,attr}...
```

パラメータの説明：

sectionname

セクションに割り当てる名前を指定します。

セクションには任意の名前を指定できます。ただし、数字で始まる名前は縦棒で囲む必要があります。これに反すると、セクション名エラーが生成されます。例えば、|1_DataArea| のように記述します。

定型の名前は従来どおりです。例えば、|.text| は、C コンパイラによって生成されるコードセクション、または C ライブラリに関連付けられたコードセクションに使用されます。

attr

1 つ以上のセクション属性をコンマで区切って指定します。有効な属性は以下のとおりです。

ALIGN=expression

デフォルトでは、ELF セクションは 4 バイト境界で整列されます。*expression* には 0 ~ 31 の任意の整数値を指定できます。このセクションは $2^{expression}$ バイト境界で整列されます。例えば、*expression* に 10 が指定されている場合、このセクションは 1KB 境界で整列されます。

これは、ALIGN ディレクティブの指定方法とは異なります。詳細については、「ALIGN」(P. 7-64) を参照して下さい。

——注——

コードセクションには ALIGN=0 または ALIGN=1 を使用しないで下さい。

ASSOC=section

section には、関連する ELF セクションを指定します。*sectionname* は、*section* を含むすべてのリンクに含まれている必要があります。

CODE

マシン命令を保持します。READONLY がデフォルトです。

COMDEF	共通セクションの定義です。この ELF セクションは、コードまたはデータを保持できます。このセクションは、他のソースファイル内にある同じ名前の付いた他のセクションと同じである必要があります。 同じ名前の付いた同一の ELF セクションは、リンクによって同じメモリセクションにオーバーレイされます。これらのセクションが異なっていると、リンクによって警告メッセージが生成され、セクションのオーバーレイは行われません。詳細については、 <i>RealView Compilation Tools v3.0 Linker and Utilities Guide</i> の基本的なリンクの機能に関する章を参照して下さい。
COMMON	共通データセクションです。このセクション内ではコードまたはデータは定義できません。このセクションは、リンクによってゼロに初期化されます。同じ名前の付いたすべての共通セクションは、リンクによって同じメモリセクションにオーバーレイされます。すべてのセクションが同じサイズである必要はありません。リンクは、同じ名前の付いたセクションのうち最も大きな共通セクションに必要とされるだけの空間を割り当てます。
DATA	命令ではなくデータを保持します。READWRITE がデフォルトです。
NOALLOC	ターゲットシステム上のメモリがこのエリアに割り当てられないことを示します。
NOINIT	データセクションが初期化されていないか、またはゼロに初期化されていることを示します。この属性に含まれるのは、空間予約ディレクティブ SPACE か、初期値がゼロに設定された DCB、DCD、DCDU、DCQ、DCQU、DCW、または DCWU ディレクティブだけです。エリアを初期化しないか、またはゼロで初期化するかはリンク時に決めることができます。詳細については、 <i>RealView Compilation Tools v3.0 Linker and Utilities Guide</i> の基本的なリンクの機能に関する章を参照して下さい。
READONLY	このセクションへの書き込みが禁止されていることを示します。コードエリアの場合はこれがデフォルトです。
READWRITE	このセクションに対する読み出しと書き込みが可能なことを示します。データエリアの場合はこれがデフォルトです。

使用法

AREA ディレクティブを使用して、ソースファイルを複数の ELF セクションに分割します。複数の AREA ディレクティブで同じ名前を使用できます。同じ名前の付いたエリアはすべて、同じ ELF セクションに配置されます。特定の名前の付いた最初の AREA ディレクティブの属性だけが適用されます。

通常、コードとデータには別々の ELF セクションを使用する必要があります。大きなプログラムは、扱いやすいように複数のコードセクションに分割できます。また、大きな独立データセットも別々のセクションに配置するのが適切です。

ローカルラベルの有効範囲は、AREA ディレクティブによって定義され、必要な場合は ROUT ディレクティブによって分割されます（「ローカルラベル」(P. 3-25) および「ROUT」(P. 7-84) 参照）。

アセンブリには少なくとも 1 つの AREA ディレクティブが必要です。

例

以下の例では、Example という名前の読み出し専用のコードセクションを定義します。

```
AREA      Example, CODE, READONLY    ; An example code section.
; code
```

7.8.3 END

END ディレクティブは、ソースファイルの終わりに到達したことをアセンブラーに通知します。

構文

```
END
```

使用法

すべてのアセンブリ言語のソースファイルは、END ディレクティブが単独で記述された行で終了する必要があります。

GET ディレクティブによってソースファイルがペアレントファイルにインクルードされている場合、アセンブラーはペアレントファイルに戻り、GET ディレクティブの次の行からアセンブルを継続します。詳細については、「GET、INCLUDE」(P. 7-76) を参照して下さい。

トップレベルのソースファイル内で、エラーが発生することなく第 1 パスで END に到達すると、第 2 パスが開始されます。

トップレベルのソースファイル内で、第 2 パスで END に到達すると、アセンブラーはアセンブリを終了して適切な出力を行います。

7.8.4 ENTRY

ENTRY ディレクティブは、プログラムへのエントリポイントを宣言します。

構文

ENTRY

使用法

プログラムには少なくとも 1 つの ENTRY ポイントを指定する必要があります。ENTRY が存在しない場合は、リンク時に警告メッセージが生成されます。

1 つのソースファイル内で複数の ENTRY ディレクティブを使用することはできません。また、すべてのソースファイル内に ENTRY ディレクティブを指定する必要があるわけではありません。1 つのソースファイル内に複数の ENTRY が存在すると、アセンブリ時にエラーメッセージが生成されます。

例

```
AREA      ARMex, CODE, READONLY
ENTRY          ; Entry point for the application
```

7.8.5 EQU

EQU ディレクティブは、数値定数、レジスタ相対値、またはプログラム相対値にシンボル名を割り当てます。* は EQU と同じ意味です。

構文

```
name EQU expr{, type}
```

パラメータの説明 :

name 値に割り当てるシンボル名を指定します。

expr レジスタ相対アドレス、プログラム相対アドレス、絶対アドレス、または 32 ビット整数定数のいずれかを指定します。

type これはオプションです。**type** には、以下のいずれかを指定できます。

- ARM
- THUMB
- CODE32
- CODE16
- DATA

type は、**expr** が絶対アドレスの場合にのみ使用できます。**name** がエクスポートされる場合、オブジェクトファイル内のシンボルテーブルに含まれる **name** エントリは、**type** に指定されている値に基づいて ARM、THUMB、CODE32、CODE16、または DATA のいずれかとしてマークされます。この情報はリンクによって使用されます。

使用法

EQU を使用して定数を定義します。これは、C 言語で #define を使用して定数を定義する方法と似ています。

シンボルのエクスポートについては、「KEEP」(P. 7-80) および「EXPORT、GLOBAL」(P. 7-71) を参照して下さい。

例

```
abc EQU 2           ; assigns the value 2 to the symbol abc.  
xyz EQU label+8    ; assigns the address (label+8) to the  
; symbol xyz.  
fiq EQU 0x1C, CODE32 ; assigns the absolute address 0x1C to  
; the symbol fiq, and marks it as code
```

7.8.6 EXPORT、GLOBAL

EXPORT ディレクティブは、個別のオブジェクトファイルとライブラリファイルに含まれるシンボルへの参照を解決するためにリンクが使用できるシンボルを宣言します。GLOBAL は EXPORT と同じ意味です。

構文

```
EXPORT {symbol}{[WEAK,attr]}
```

パラメータの説明：

<i>symbol</i>	エクスポートするシンボル名を指定します。シンボル名では大文字と小文字が区別されます。 <i>symbol</i> が省略されている場合は、すべてのシンボルがエクスポートされます。
[WEAK]	他のソースによって <i>symbol</i> の別のインスタンスがエクスポートされていない場合にのみ、このインスタンスを他のソースにインポートすることを意味します。 <i>symbol</i> を指定せずに [WEAK] を指定すると、エクスポートされるすべてのシンボルが WEAK になります。
[attr]	動的なコンポーネントにリンクされるときにシンボルを可視化します。デフォルトでは、シンボルのバインディングによって可視化が定義されます。つまり、他のコンポーネントでは、グローバルシンボルおよび WEAK シンボルは認識され、ローカルシンボルは非表示になります。 有効な属性は以下のとおりです。
DYNAMIC	<i>symbol</i> は他のコンポーネントで認識され、他のコンポーネントで再定義できます。
HIDDEN	<i>symbol</i> は、定義されているコンポーネント外部からは直接的にも間接的にも参照できません。 また、リンクでは INTERNAL も使用できます。これは、現在、HIDDEN として処理されます。両方を指定すると、以下のようにになります。 EXPORT SymA[WEAK,INTERNAL,HIDDEN] この場合、アセンブラーは最も制約の大きいもの (INTERNAL) を選択します。
PROTECTED	<i>symbol</i> は他のコンポーネントで認識されますが、他のコンポーネントで再定義することはできません。

使用法

`EXPORT` を使用して、他のファイルのコードが現在のファイルのシンボルにアクセスできるようにします。

[`WEAK`] 属性を使用して、*symbol* の別のインスタンスが別のソースに存在する場合は、そのインスタンスがこのインスタンスに優先されることをリンクに通知します。[`WEAK`] 属性は、任意のシンボルの可視化属性と共に使用できます。

エクスポートが重複する場合、シンボルの可視化はオーバーライドされることがあります。以下の例では、最後の `EXPORT` のバインドと可視化が優先されます。

```
EXPORT  SymA[WEAK]      ; Export as weak-hidden
EXPORT  SymA[DYNAMIC]    ; SymA becomes non-weak dynamic.
```

/IMPORT/ (P. 7-77) も参照して下さい。

例

```
AREA   Example, CODE, READONLY
EXPORT DoAdd           ; Export the function name
                  ; to be used by external
                  ; modules.
DoAdd  ADD    r0,r0,r1
```

7.8.7 EXPORTAS

EXPORTAS ディレクティブを使用すると、ソースファイル内の別のシンボルに対応する、シンボルをオブジェクトファイルにエクスポートできます。

構文

```
EXPORTAS symbol11, symbol12
```

パラメータの説明：

symbol11 ソースファイル内のシンボルの名前を指定します。*symbol11* は事前に定義されている必要があります。この名前には、エリア名、ラベル、または定数を含むシンボルを指定できます。

symbol12 オブジェクトファイルで使用するシンボルの名前を指定します。

シンボル名では大文字と小文字が区別されます。

使用法

EXPORTAS を使用すると、ソースファイル内の各インスタンスを変更する必要なく、オブジェクトファイル内のシンボルを変更することができます。

「*EXPORT*, *GLOBAL*」(P. 7-71) も参照して下さい。

例

```
AREA data1, DATA      ;; starts a new area data1
AREA data2, DATA      ;; starts a new area data2
EXPORTAS data2, data1 ;; the section symbol referred to as data2 will
                      ;; appear in the object file string table as data1.

one EQU 2
EXPORTAS one, two
EXPORT one             ;; the symbol 'two' will appear in the object
                      ;; file's symbol table with the value 2.
```

7.8.8 EXTERN

EXTERN ディレクティブは、現在のアセンブリに定義されていない名前をアセンブリに渡します。

EXTERN は **IMPORT** とよく似ています。ただし、名前への参照が現在のアセンブリに検出されない場合、その名前はインポートされません (*/IMPORT* (P. 7-77) および */EXPORT*, *GLOBAL* (P. 7-71) 参照)。

構文

```
EXTERN symbol {[WEAK,attr]}
```

パラメータの説明：

symbol 別々にアセンブルされたソースファイル、オブジェクトファイル、またはライブラリ内で定義されているシンボルの名前を指定します。シンボル名では大文字と小文字が区別されます。

[WEAK] シンボルがどこにも定義されていない場合に、リンクによってエラーメッセージが生成されるのを回避します。また、このオプションを指定すると、リンクはインクルードされていないライブラリの検索も行いません。

[attr] 動的なコンポーネントにリンクされるときにシンボルを可視化します。デフォルトでは、シンボルのバインディングによって可視化が定義されます。つまり、他の外部オブジェクトでは、グローバルシンボルおよびWEAKシンボルは認識され、ローカルシンボルは非表示になります。

有効な属性は以下のとおりです。

DYNAMIC *symbol* は他のコンポーネントで認識され、他のコンポーネントで再定義できます。

HIDDEN *symbol* は、定義されているコンポーネント外部からは直接的にも間接的にも参照できません。

また、リンクでは **INTERNAL** も使用できます。これは、現在、**HIDDEN** として処理されます。両方を指定すると、以下のようにになります。

EXTERN SymA [WEAK,INTERNAL,HIDDEN]

この場合、アセンブリは最も制約の大きいもの (**INTERNAL**) を選択します。

PROTECTED *symbol* は他のコンポーネントで認識されますが、他のコンポーネントで再定義することはできません。

使用法

別のオブジェクトファイル内で定義されたシンボルへのリンク時に、シンボル名が解決されます。このシンボルはプログラムのアドレスとして処理されます。[WEAK] が指定されていない場合、対応するシンボルがリンク時に検出されないと、リンクによってエラーが生成されます。

[WEAK] が指定されており、対応するシンボルがリンク時に検出されない場合は、以下のようになります。

- 参照が B 命令または BL 命令のデスティネーションである場合、シンボルの値が次の命令のアドレスとして使用されます。したがって、この B 命令または BL 命令は NOP となります。
- それ以外の場合、シンボルの値はゼロと見なされます。

例

この例では、C++ ライブラリがリンクされていて、演算結果に基づいて条件分岐が発生するかどうかを確認しています。

```

AREA      Example, CODE, READONLY
EXTERN    __CPP_INITIALIZE[WEAK] ; If C++ library linked, gets the address of
                                ; __CPP_INITIALIZE function.
LDR      r0,=__CPP_INITIALIZE   ; If not linked, address is zeroed.
CMP      r0,#0                  ; Test if zero.
BEQ      nocplusplus           ; Branch on the result.

```

7.8.9 GET、INCLUDE

GET ディレクティブは、アセンブル中のファイル内に別のファイルをインクルードします。インクルードされるファイルは、GET ディレクティブの位置でアセンブルされます。INCLUDE は GET と同じ意味です。

構文

`GET filename`

パラメータの説明：

`filename` アセンブリ時にインクルードされるファイルの名前を指定します。アセンブラーは UNIX 形式または MS-DOS 形式のパス名を認識します。

使用法

GET は、アセンブリ時にマクロ定義、EQU、および記憶域マップをインクルードするのに便利です。インクルードされたファイルのアセンブルが完了すると、アセンブラーは GET ディレクティブの次の行からアセンブルを継続します。

デフォルトでは、アセンブラーは、現在の場所でインクルードされるファイルを検索します。現在の場所とは、呼び出し元のファイルが存在するディレクトリを指します。検索パスにディレクトリを追加するには、アセンブラーの -i コマンドラインオプションを使用します。スペースを含むファイル名およびディレクトリ名を二重引用符 (" ") で囲まないで下さい。

インクルードされるファイル内で別の GET ディレクティブを使用して、他のファイルをインクルードできます（「ネスティングディレクティブ」(P. 7-32) 参照）。

インクルードされるファイルが現在の場所とは異なるディレクトリにある場合は、インクルードされるファイルの終わりまで、そのディレクトリが現在の場所となります。その後、前の場所が復元されます。

GET を使用して、オブジェクトファイルをインクルードすることはできません（`/INCBIN` (P. 7-79) 参照）。

例

```
AREA Example, CODE, READONLY
GET file1.s           ; includes file1 if it exists
                      ; in the current place.
GET c:\project\file2.s ; includes file2
GET c:\Program files\file3.s ; space is permitted
```

7.8.10 IMPORT

IMPORT ディレクティブは、現在のアセンブリに定義されていない名前をアセンブラーに渡します。

指定された名前が現在のアセンブリに参照されるかどうかに関係なくインポートされる点を除き、**IMPORT** は **EXTERN** とよく似ています (*'EXTERN'* (P. 7-74) および *'EXPORT, GLOBAL'* (P. 7-71) 参照)。

構文

```
IMPORT symbol {[WEAK,attr]}
```

パラメータの説明：

symbol 別々にアセンブルされたソースファイル、オブジェクトファイル、またはライブラリ内で定義されているシンボルの名前を指定します。シンボル名では大文字と小文字が区別されます。

[WEAK] シンボルがどこにも定義されていない場合に、リンクによってエラーメッセージが生成されるのを回避します。また、このオプションを指定すると、リンクはインクルードされていないライブラリの検索も行いません。

[attr] 動的なコンポーネントにリンクされるときにシンボルを可視化します。デフォルトでは、シンボルのバイディングによって可視化が定義されます。つまり、他のコンポーネントでは、グローバルシンボルおよび **WEAK** シンボルは認識され、ローカルシンボルは非表示になります。

有効な属性は以下のとおりです。

DYNAMIC *symbol* は他のコンポーネントで認識され、他のコンポーネントで再定義できます。

HIDDEN *symbol* は、定義されているコンポーネント外部からは直接的にも間接的にも参照できません。

また、リンクでは **INTERNAL** も使用できます。これは、現在、**HIDDEN** として処理されます。両方を指定すると、以下のようにになります。

IMPORT SymA[WEAK,INTERNAL,HIDDEN]

この場合、アセンブラーは最も制約の大きいもの (**INTERNAL**) を選択します。

PROTECTED *symbol* は他のコンポーネントで認識されますが、他のコンポーネントで再定義することはできません。

使用法

別のオブジェクトファイル内で定義されたシンボルへのリンク時に、シンボル名が解決されます。このシンボルはプログラムのアドレスとして処理されます。[WEAK] が指定されていない場合、対応するシンボルがリンク時に検出されないと、リンクによってエラーが生成されます。

[WEAK] が指定されており、対応するシンボルがリンク時に検出されない場合は、以下のようになります。

- 参照が B 命令または BL 命令のデスティネーションである場合、シンボルの値が次の命令のアドレスとして使用されます。したがって、この B 命令または BL 命令は NOP となります。
- それ以外の場合、シンボルの値はゼロと見なされます。

リンク時に検出されないシンボルへのアクセスを回避するには、*/EXTERN/* (P. 7-74) の例で示したようなコードを使用して下さい。

7.8.11 INCBIN

INCBIN ディレクトリは、アセンブル中のファイル内に別のファイルをインクルードします。このファイルはアセンブルされずにそのままインクルードされます。

構文

`INCBIN filename`

パラメータの説明：

`filename` アセンブリ時にインクルードされるファイルの名前を指定します。アセンブリは UNIX 形式または MS-DOS 形式のパス名を認識します。

使用法

INCBIN を使用して、実行可能ファイル、リテラル、または任意のデータをインクルードできます。ファイルの内容は、解釈されることなく、現在の ELF セクションにバイト単位で追加されます。アセンブルは、INCBIN ディレクトリの次の行から継続されます。

デフォルトでは、アセンブリは、現在の場所でインクルードされるファイルを検索します。現在の場所とは、呼び出し元のファイルが存在するディレクトリを指します。検索パスにディレクトリを追加するには、アセンブリの -i コマンドラインオプションを使用します。スペースを含むファイル名およびディレクトリ名を二重引用符 (" ") で囲まないで下さい。

例

```
AREA Example, CODE, READONLY
INCBIN file1.dat           ; includes file1 if it
                           ; exists in the
                           ; current place.
INCBIN "c:\project\file2.txt"; includes file2
```

7.8.12 KEEP

KEEP ディレクティブは、オブジェクトファイル内のシンボルテーブルにローカルシンボルを保持しておくようアセンブラに指定します。

構文

```
KEEP {symbol}
```

パラメータの説明：

symbol1 保持するローカルシンボルの名前を指定します。*symbol1* が指定されていない場合は、レジスタ相対シンボルを除くすべてのローカルシンボルが保持されます。

使用法

デフォルトでは、アセンブラによって出力されるオブジェクトファイル内に記述されるシンボルは以下のシンボルのみです。

- エクスポートされるシンボル
- 再配置されるシンボル

KEEP を使用して、デバッグに使用できるローカルシンボルを保持します。保持されたシンボルは、ARM デバッガおよびリンクマップファイルに出力されます。

KEEP では、レジスタ相対シンボルを保持できません（/MAP/ (P. 7-20) 参照）。

例

```
label  ADC    r2,r3,r4
      KEEP   label    ; makes label available to debuggers
      ADD    r2,r2,r5
```

7.8.13 NOFP

NOFP ディレクティブは、アセンブリ言語ソースファイル内の浮動小数点命令を禁止します。

構文

NOFP

使用法

NOFP を使用して、ソフトウェアまたはターゲットハードウェアで浮動小数点命令がサポートされていない場合などに、浮動小数点命令が使用されることを防ぎます。

NOFP ディレクティブより後に浮動小数点命令が出現すると、Unknown opcode エラーが生成され、アセンブルに失敗します。

浮動小数点命令より後に NOFP ディレクティブが検出されると、アセンブラーによって以下のエラーが生成され、

`Too late to ban floating point instructions`

アセンブルに失敗します。

7.8.14 REQUIRE

REQUIRE ディレクティブは、セクション間の依存関係を指定します。

構文

REQUIRE *label*

パラメータの説明：

label 必要なラベルの名前を指定します。

使用法

REQUIRE を使用すると、関連するセクションが直接呼び出されない場合でも、そのセクションをインクルードすることができます。REQUIRE ディレクティブを含むセクションがリンクにインクルードされる場合、リンクは、指定されたラベルの定義を含むセクションもインクルードします。

7.8.15 REQUIRE8、PRESERVE8

REQUIRE8 ディレクティブは、現在のファイルに 8 バイト境界で境界調整されたスタックが必要であることを指定します。このディレクティブは、このことをリンクに通知するために REQ8 ビルド属性を設定します。

PRESERVE8 ディレクティブは、現在のファイルが 8 バイト境界で境界調整されたスタックを保持していることを指定します。このディレクティブは、このことをリンクに通知するために PRES8 ビルド属性を設定します。

リンクは、8 バイト境界で境界調整されたスタックを必要とするコードが、8 バイト境界で境界調整されたスタックを保持するコードによってのみ直接的または間接的に呼び出されることをチェックします。

構文

```
REQUIRE8 {bool}
```

```
PRESERVE8 {bool}
```

パラメータの説明：

<i>bool</i>	任意でブール定数 ({TRUE} または {FALSE}) を指定します。
-------------	---------------------------------------

使用法

コードで 8 バイト境界で境界調整されたスタックを保持する場合は、必要に応じて、**PRESERVE8** を使用してファイルに PRES8 ビルド属性を設定します。コードで 8 バイト境界で境界調整されたスタックを保持しない場合は、**PRESERVE8 {FALSE}** を使用して PRES8 ビルド属性が設定されないようにします。

——注——

PRESERVE8 と **PRESERVE8 {FALSE}** の両方を省略すると、アセンブラーは、sp を変更する命令を調べて、PRES8 ビルド属性を設定するかどうかを決定します。**PRESERVE8** は明示的に指定することをお勧めします。

以下のように警告メッセージをイネーブルできます。

```
armasm --diag_warning 1546
```

詳細については、「コマンド構文」(P. 3-2) を参照して下さい。

警告メッセージをイネーブルすると、以下のような警告メッセージが表示されます。

```
"test.s", line 37: Warning: A1546W: Stack pointer update potentially
                     breaks 8 byte stack alignment
      37 00000044      STMFD   sp!,{r2,r3,lr}
```

例

```
REQUIRE8
REQUIRE8 {TRUE}      ; equivalent to REQUIRE8
REQUIRE8 {FALSE}     ; equivalent to absence of REQUIRE8
PRESERVE8 {TRUE}    ; equivalent to PRESERVE8
PRESERVE8 {FALSE}   ; NOT exactly equivalent to absence of PRESERVE8
```

7.8.16 ROUT

ROUT ディレクティブは、ローカルラベルの有効範囲の境界をマークします（「ローカルラベル」(P. 3-25) 参照）。

構文

{*name*} ROUT

パラメータの説明：

name 有効範囲に割り当てる名前を指定します。

使用法

ROUT ディレクティブを使用して、ローカルラベルの有効範囲を制限します。これにより、誤って違うラベルが参照されるのを簡単に防ぐことができます。ローカルラベルの有効範囲内に ROUT ディレクティブが存在しない場合は、ローカルラベルの有効範囲はエリア全体となります（「AREA」(P. 7-66) 参照）。

正しいローカルラベルへの参照が行われるようにするには、*name* オプションを使用します。ラベルの名前またはラベルへの参照が、その前の ROUT ディレクティブに指定されている名前と一致しない場合は、アセンブラーによってエラーメッセージが生成され、アセンブルに失敗します。

例

```
; code
routineA    ROUT           ; ROUT is not necessarily a routine
; code
3routineA   ; code          ; this label is checked
; code
        BEQ    %4routineA  ; this reference is checked
; code
        BGE    %3           ; refers to 3 above, but not checked
; code
4routineA   ; code          ; this label is checked
; code
otherstuff  ROUT           ; start of next scope
```