ARM アーキテクチャ リファレンスマニュアル

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ARM DDI 0100HJ-00

ARM アーキテクチャリファレンスマニュアル

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited.All rights reserved.

リリース情報

このドキュメントには、以下の変更が加えられています。

改訂履歴

日付	変更箇所	変更内容
1996 年 2 月	A	初版
1997 年 7 月	B	更新と索引の追加
1998 年 4 月	C	更新
2000 年 2 月	D	ARM アーキテクチャ v5 に対応した更新
2000 年 6 月	E	ARM アーキテクチャ v5TE に対応した更新とパート B の修正
2004 年 7 月	F	ARM アーキテクチャ v6 に対応した更新（非公開）
2004 年 12 月	G	誤記の修正
2005 年 3 月	H	誤記の修正

著作権表記

ARM、ARM Powered ロゴ、Thumb、StrongARM は ARM 社の登録商標です。

ARM ロゴ、AMBA、Angel、ARMulator、EmbeddedICE、ModelGen、Multi-ICE、PrimeCell、ARM7TDMI、ARM7TDMI-S、ARM9TDMI、ARM9E-S、ETM7、ETM9、TDMI、STRONG は ARM 社の商標です。

このドキュメントに表記されている他の製品やサービスは、対応する所有者の商標の場合があります。

このドキュメントに説明されている製品は、継続的に開発と改良が行われています。このドキュメントにある製品とその使用法に関する記載事項について、ARM は保証しません。

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

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

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

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

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

Copyright ©1996-1998, 2000, 2004, 2005 ARM Limited.

110 Fulbourn Road Cambridge, England CB1 9NJ

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

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

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

目次
ARM アーキテクチャリファレンスマニュアル

序章

本書について ... xii
アーキテクチャのバージョンとバリエーション ... xiii
本書の使用法 ... xviii
表記規則 ... xxi
参考資料 ... xxiii
ご意見・ご質問 ... xxiv

パート A CPU のアーキテクチャ

第 A1 章 ARM アーキテクチャの概要

A1.1 ARM アーキテクチャについて ... A1-2
A1.2 ARM 命令セット ... A1-6
A1.3 Thumb 命令セット ... A1-11

第 A2 章 プログラマモデル

A2.1 データタイプ ... A2-2
A2.2 プロセッサモード ... A2-3
A2.3 レジスタ ... A2-4
A2.4 汎用レジスタ ... A2-6
A2.5 プログラムステータスレジスタ ... A2-11

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.6 例外 A2-16
A2.7 エンディアンのサポート A2-30
A2.8 アンアラインドアクセスのサポート A2-38
A2.9 同期化基本命令 A2-44
A2.10 Jazelle 拡張 A2-53
A2.11 飽和整数算術演算 A2-69

第 A3 章 ARM 命令セット

A3.1 命令セットのエンコード A3-2
A3.2 条件フィールド A3-3
A3.3 分岐命令 A3-5
A3.4 データ処理命令 A3-7
A3.5 乗算命令 A3-10
A3.6 並列加算命令と並列減算命令 A3-14
A3.7 拡張命令 A3-16
A3.8 その他の算術演算命令 A3-17
A3.9 その他の命令 A3-18
A3.10 ステータスレジスタアクセス命令 A3-19
A3.11 ロード / ストア命令 A3-21
A3.12 複数ロード / ストア命令 A3-26
A3.13 セマフォ命令 A3-28
A3.14 例外生成命令 A3-29
A3.15 コプロセッサ命令 A3-30
A3.16 命令セットの拡張 A3-32

第 A4 章 ARM 命令

A4.1 ARM 命令のアルファベット順リスト A4-2
A4.2 ARM 命令とアーキテクチャのバージョン A4-287

第 A5 章 ARM アドレシングモード

A5.1 アドレシングモード 1 - データ処理オペランド A5-2
A5.2 アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア A5-18
A5.3 アドレシングモード 3 - その他のロードとストア A5-33
A5.4 アドレシングモード 4 - 複数ロード / ストア A5-41
A5.5 アドレシングモード 5 - コプロセッサのロード / ストア A5-49

第 A6 章 Thumb 命令セット

A6.1 Thumb 命令セットについて A6-2
A6.2 命令セットのエンコード A6-4
A6.3 分岐命令 A6-6
A6.4 データ処理命令 A6-8
A6.5 レジスタロード / ストア命令 A6-15
A6.6 複数ロード / ストア命令 A6-18
A6.7 例外生成命令 A6-20
A6.8 未定義命令空間 A6-21

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 A7 章 Thumb 命令

A7.1 Thumb 命令のアルファベット順一覧 A7-2
A7.2 Thumb 命令とアーキテクチャのバージョン A7-125

パート B メモリとシステムアーキテクチャ

第 B1 章 メモリとシステムアーキテクチャの概要

B1.1 メモリシステムの概要 B1-2
B1.2 メモリ階層 B1-4
B1.3 L1 キャッシュ B1-6
B1.4 L2 キャッシュ B1-7
B1.5 ライトバッファ B1-8
B1.6 密結合メモリ B1-9
B1.7 非同期な例外 B1-10
B1.8 セマフォ B1-12

第 B2 章 メモリオーダモデル

B2.1 メモリオーダモデルの概要 B2-2
B2.2 読み出しと書き込みの定義 B2-4
B2.3 ARMv6 以前のメモリ属性 B2-7
B2.4 ARMv6 のメモリ属性の概要 B2-8
B2.5 メモリアクセスの順序に関する条件 B2-16
B2.6 メモリバリア B2-18
B2.7 メモリのコヒーレンシとアクセスの考慮点 B2-20

第 B3 章 システム制御コプロセッサ

B3.1 システム制御コプロセッサについて B3-2
B3.2 レジスタ B3-3
B3.3 レジスタ 0: ID コード B3-7
B3.4 レジスタ 1: 制御レジスタ B3-12
B3.5 レジスタ 2 から 15 まで B3-18

第 B4 章 仮想メモリシステムアーキテクチャ

B4.1 VMSA の概要 B4-2
B4.2 メモリアクセスのシーケンス B4-4
B4.3 メモリアクセスの制御 B4-8
B4.4 メモリ領域属性 B4-11
B4.5 アボート B4-14
B4.6 フォルトアドレスレジスタとフォルトステータスレジスタ B4-19
B4.7 ハードウェアページテーブル変換 B4-23
B4.8 ファインページテーブルとタイニーページのサポート B4-35
B4.9 CP15 のレジスタ B4-39

第 B5 章 保護メモリシステムアーキテクチャ

B5.1 PMSA の概要 B5-2

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.2 メモリアクセスのシーケンス B5-4
B5.3 メモリアクセスの制御 B5-8
B5.4 メモリアクセスの属性 B5-10
B5.5 メモリアボート（PMSAv6） B5-13
B5.6 フォルトステータスレジスタとフォルトアドレスレジスタのサポート B5-16
B5.7 CP15 のレジスタ B5-18

第 B6 章 キャッシュとライトバッファ
B6.1 キャッシュとライトバッファの概要 B6-2
B6.2 キャッシュの構成 B6-4
B6.3 キャッシュのタイプ B6-7
B6.4 L1 キャッシュ B6-10
B6.5 キャッシュのレベルの追加に関する考慮事項 B6-12
B6.6 CP15 のレジスタ B6-13

第 B7 章 密結合メモリ
B7.1 TCM について B7-2
B7.2 TCM の構成と制御 B7-3
B7.3 TCM とキャッシュへのアクセス B7-7
B7.4 レベル 1（L1）DMA モデル B7-8
B7.5 CP15 のレジスタ 11 を使用した L1 DMA 制御 B7-9

第 B8 章 高速コンテキストスイッチ拡張機能
B8.1 FCSE の概要 B8-2
B8.2 修飾仮想アドレス B8-3
B8.3 FCSE の許可 B8-5
B8.4 デバッグとトレース B8-6
B8.5 CP15 のレジスタ B8-7

パート C ベクタ浮動小数点アーキテクチャ

第 C1 章 ベクタ浮動小数点アーキテクチャの概要
C1.1 ベクタ浮動小数点アーキテクチャの概要 C1-2
C1.2 VFP アーキテクチャの概要 C1-4
C1.3 IEEE754 標準への準拠 C1-9
C1.4 IEEE 754 実装上の選択 C1-10

第 C2 章 VFP プログラマモデル
C2.1 浮動小数点形式 C2-2
C2.2 丸め C2-9
C2.3 浮動小数点例外 C2-10
C2.4 Flush-to-Zero モード C2-14
C2.5 デフォルト NaN モード C2-16
C2.6 浮動小数点汎用レジスタ C2-17
C2.7 システムレジスタ C2-21
C2.8 リセット時の動作と初期化 C2-29

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 C3 章 VFP 命令セットの概要

C3.1 データ処理命令 ... C3-2
C3.2 ロードとストア命令 ... C3-14
C3.3 単一レジスタ転送命令 ... C3-18
C3.4 2 レジスタ転送命令 ... C3-22

第 C4 章 VFP 命令

C4.1 VFP 命令のアルファベット順一覧 ... C4-2

第 C5 章 VFP のアドレシングモード

C5.1 アドレシングモード 1 - 単精度ベクタ（非単項） ... C5-2
C5.2 アドレシングモード 2 - 倍精度ベクタ（非単項） ... C5-8
C5.3 アドレシングモード 3 - 単精度ベクタ（単項） ... C5-14
C5.4 アドレシングモード 4 - 倍精度ベクタ（単項） ... C5-18
C5.5 アドレシングモード 5 - VFP 複数ロード / ストア ... C5-22

パート D デバッグアーキテクチャ

第 D1 章 デバッグアーキテクチャの概要

D1.1 概要 ... D1-2
D1.2 トレース ... D1-4
D1.3 デバッグと ARMv6 ... D1-5

第 D2 章 デバッグイベントと例外

D2.1 概要 ... D2-2
D2.2 モニタデバッグモード ... D2-5
D2.3 ホールトデバッグモード ... D2-8
D2.4 外部デバッグインターフェース ... D2-13

第 D3 章 コプロセッサ 14: デバッグコプロセッサ

D3.1 コプロセッサ 14 のデバッグレジスタ ... D3-2
D3.2 コプロセッサ 14 のデバッグ命令 ... D3-5
D3.3 デバッグレジスタリファレンス ... D3-8
D3.4 CP14 のデバッグレジスタのリセット時の値 ... D3-24
D3.5 外部デバッグインターフェースから CP14 のデバッグレジスタへのアクセス ... D3-25

用語集

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

序章

本章では、ARM® アーキテクチャの各バージョンとこのマニュアルの内容について説明し、使用する表記規則と用語の一覧を示します。

- *本書について* : P. xii
- *アーキテクチャのバージョンとバリエーション* : P. xiii
- *本書の使用法* : P. xviii
- *表記規則* : P. xxi
- *参考資料* : P. xxiii
- *ご意見・ご質問* : P. xxiv

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

本書について

本書では、ARM 命令セットアーキテクチャについて解説します。これには、コード密度の高い Thumb®サブセットと、3 つの標準コプロセッサ拡張機能が含まれます。

- 標準のシステム制御コプロセッサ（コプロセッサ 15）はキャッシュ、ライトバッファ、メモリ管理ユニット、保護ユニットなどのメモリシステムコンポーネントの制御に使用されます。
- *ベクタ浮動小数点*（VFP）アーキテクチャはコプロセッサ 10 と 11 を使用し、高パフォーマンスの浮動小数点命令セットを提供します。
- ARMv6 のアーキテクチャには、デバッグアーキテクチャインターフェース（コプロセッサ 14）が正式に追加され、ARM コアのデバッグ機能（ブレークポイントやウォッチポイントの制御など）にソフトウェアでアクセス可能になりました。

32 ビットの ARM 命令セットと 16 ビットの Thumb 命令セットについては、パート A で別々に解説されています。これらの命令の正確な効果と、使用に関する制限について説明されています。この情報は、コンパイラ、アセンブラ、その他 ARM マシンコードを生成するプログラムの作成者に特に重要です。

本書で解説されている命令のほとんどにはアセンブラ構文が記載され、命令をテキスト形式で指定できるようになっています。

ただし、このマニュアルは ARM アセンブラ言語のチュートリアルを意図したものではなく、ARM アセンブラ言語を極めて基本的なレベルで解説したものでしかありません。ARM アセンブラ言語を効率的に使用するには、使用するアセンブラに付属のドキュメントを参照して下さい。

メモリとシステムのアーキテクチャの定義は、ARM アーキテクチャバージョン 6（最新バージョン）で大きく改良されています。それ以前は、使用するデバイスのテクニカルリファレンスマニュアルにある詳細な実装固有の情報で補完することが一般に必要でした。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アーキテクチャのバージョンとバリエーション

ARM 命令セットアーキテクチャは、最初に開発されて以降大幅に拡張されており、将来も開発が続けられる予定です。今日までに命令セットの 6 つのメジャーバージョンが定義されており、これらにはバージョン番号として 1 から 6 までが割り振られています。これらのうち、元の 26 ビットアーキテクチャを含む最初の 3 バージョン（32 ビットアーキテクチャは ARMv3 で導入されました）は現在では使用されていません。26 ビット機能で使用されていたすべてのビットとエンコードは、ARM 社による将来の拡張用に予約されています。

バージョンには、アーキテクチャの拡張機能に含まれている追加命令群を示すバリエーション文字が付加されていることがあります。拡張機能は一般に、次のバージョン番号でベースアーキテクチャに盛り込まれます。ただし ARMv5T は注意すべき例外です。ある機構を搭載していないことを示すために、次のバージョン 5 の概要にある xP バリエーションのように、バリエーション文字の前に x がつけられています。

——— 注 ———

xM バリエーションは、ロング乗算（32 ビット× 32 ビットで 64 ビットの結果を求める乗算）をサポートしていないことを示しますが、これは廃止されました。

———————

有効なアーキテクチャバリエーションは次のとおりです（かっこ内のバリエーションは、過去に存在したものです）。

ARMv4、ARMv4T、ARMv5T、（ARMv5TExP）、ARMv5TE、ARMv5TEJ、ARMv6

次のアーキテクチャバリエーションは、現在は使用されていません。

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

現在使用されていないバージョンの詳細については、ARM にお問合せ下さい。

ARM 命令セットと Thumb 命令セットについては、アーキテクチャバリエーションごとにそれぞれP. A4-287「*ARM 命令とアーキテクチャのバージョン*」と P. A7-125「*Thumb 命令とアーキテクチャのバージョン*」に要約されています。ARMv4 以降に導入された主要な変更点は次のとおりです。

バージョン 4 での Thumb 命令セットの導入（T バリエーション）

Thumb 命令セットは、ARM 命令セットを再エンコードしたサブセットです。Thumb 命令は独自のプロセッサステートで実行され、ARM 状態と Thumb 状態の間で移行を行うために必要な機構はアーキテクチャにより定義されます。主要な相違点は、Thumb 命令のサイズが ARM 命令の半分だということです（Thumb 命令は 16 ビット、ARM 命令は 32 ビット）。Thumb 命令セットを使用すると、普通は ARM 命令セットと比較してコード密度を高くすることができます。ただし、Thumb 命令セットにはいくつかの制約があります。

- Thumb コードは一般に、同じ作業を実行するためにより多くの命令が必要なため、実行時間が重要なコードでは ARM コードを使用した方が高いパフォーマンスを実現できます。
- 例外の処理には ARM 状態と、いくつかの関連する ARM 命令が必要です。

Thumb 命令セットは常に、いずれかのバージョンの ARM 命令セットと組み合わせて使用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

バージョン 5T で導入された新機能

このバージョンでは、アーキテクチャのバージョン 4T に次のような拡張が加えられています。

- ARM/Thumb のインタワーキングの効率が改善されました。
- 先行ゼロカウント命令（CLZ、ARM のみ）と、ソフトウェアブレークポイント命令（BKPT、ARM と Thumb の両方）が追加されました。
- コプロセッサ設計者向けのオプションが追加されました（コプロセッサは ARM でのみサポートされます）。
- 乗算でのフラグ設定がより厳密に定義されました（ARM と Thumb の両方）。
- 新たに E バリエーションが追加され、特定のデジタル信号処理（DSP）アルゴリズムで ARM プロセッサのパフォーマンスを向上させる ARM 命令が追加されました。
  - 16 ビットデータアイテムを操作する、いくつかの乗算および積和演算命令
  - 符号付き飽和演算を実行する加算および減算命令。飽和演算では、計算が通常の整数の範囲からオーバフローした場合、結果がラップアラウンドするのではなく、正または負の最大値に結果が設定されます。
  - 2 ワードのデータを操作するロード（LDRD）、ストア（STRD）、コプロセッサレジスタ転送（MCRR と MRRC）命令
  - データプリロード命令 PLD
- 新たに J バリエーションが追加され、Jazelle® アーキテクチャ拡張機能をサポートするために必要な BXJ 命令と他の機構が追加されました。

——— 注 ———

E バリエーションの初期の実装の一部には、LDRD、STRD、MCRR、MRCC、PLD 命令が存在しません。これらは ExP バリエーションに準拠すると指定されています。このバリエーションは過去のアーキテクチャのためにのみ定義されています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

バージョン 6 で導入された新機能

次の ARM 命令が追加されています。

- 例外処理を改善する CPS、SRS、RFE 命令
- バイト反転命令 : REV、REV16、REVSH
- 改訂されたエンディアン（メモリ）モデル用の SETEND
- 排他アクセス命令 : LDREX、STREX
- バイト / ハーフワード拡張命令 : SXTB、SXTH、UXTB、UXTH
- SIMD（並列処理）メディア命令のセット
- 64 ビットの結果を生成する、新しい形式の累積付き乗算命令

次の Thumb 命令が追加されています。

- CPS、CPY（MOV の一形式）、REV、REV16、REVSH、SETEND、SXTB、SXTH、UXTB、UXTH

その他、ARMv6 では次の点が変更されています。

- アーキテクチャ名 ARMv6 は、従来の機能をすべて含み、ARMv5TEJ 準拠であることを意味します。
- 仮想メモリと保護メモリのシステムアーキテクチャが改訂されています。
- 密結合メモリモデルの定義
- アラインされていないワードとハーフワードアクセスのハードウェアサポート
- 外部とコプロセッサ 14 ベースのインターフェースによるデバッグアーキテクチャの正式な採用
- ARMv6 以前は、パート B の第 3 章で説明されているシステム制御コプロセッサ（CP15）は推奨のみでした。ARMv6 では、このプロセッサのサポートは必須になっています。
- 歴史的な理由から、PC に書き込まれたアラインされていない値に関する規則は、ARMv6 以前はやや複雑でした。ARMv6 では、これらの規則がより単純で一貫したものになっています。
- ARMv6 以前は、*ハイベクタ*拡張はアーキテクチャのオプション（実装定義）部分でした。ARMv6 では、この拡張は必須です。
- ARMv6 以前は、プロセッサは 2 つのアボートモデルのいずれかを使用可能でした。ARMv6 では、*ベース復元アボートモデル*（BRAM）を使用する必要があります。従来サポートされていたアボートモデルは次の 2 つです。
  - — BRAM: メモリシステムアボートを発生した有効なロード / ストア命令のベースレジスタはすべて、その命令が実行される前の値に復元されます。
  - — *ベース更新アボートモデル*（BUAM）: メモリシステムアボートを発生した有効なロード / ストア命令のベースレジスタは、その命令のベースレジスタライトバック（存在する場合）により変更されている場合があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- 乗算のデスティネーションレジスタがソースレジスタと異なっている必要があるという制約は、ARMv6 では削除されています。
- ARMv5 では、ARM 命令の LDM (2) と STM (2) には、直後の命令でバンクレジスタを使用する場合の制限があります。これらの制限は、ARMv6 では削除されています。
- MSR 命令でどの PSR ビットが更新されるかを決定する規則が明確になり、ARMv6 で定義された新しい PSR ビットをカバーするように拡張されています。
- ARMv5 では、Thumb の MOV 命令の動作は使用されるレジスタによって異なっています（注を参照して下さい）。ARMv6 では、2 つの変更が加えられています。
  - MOV (3) 命令のエンコードで低位のレジスタ番号を使用する場合の制限が削除されています。
  - 既存のアセンブラソースの意味を変更せず、アセンブラ言語プログラマが、他の部分に変更を加えない新しい MOV 命令を使えるようにするため、新しいアセンブラ構文 CPY Rd,Rn が追加されています。この構文は、Rd と Rn が高位レジスタか低位レジスタかにかかわらず、常に MOV (3) 命令にアセンブルされます。

——— 注 ———

ARMv5 では、Thumb の MOV Rd,Rn 命令は次のような性質があります。

- Rd と Rn がどちらも低位レジスタの場合、命令は MOV (2) 命令です。この命令は転送される値に従って N フラグと Z フラグをセットし、C フラグと V フラグは 0 に設定されます。
- Rd と Rn のいずれかが高位レジスタの場合、命令は MOV (3) 命令です。この命令では、条件フラグは変更されません。

このため、使用されるレジスタによって動作が異なります。また、MOV (2) がフラグに与える影響の関係で、コンパイラがグローバルレジスタアロケータで擬似レジスタを使用する場合に制限があります。

ARM/Thumb アーキテクチャバージョンの名称

ARM/Thumb アーキテクチャの正確なバージョンとバリエーションを名前で示すため、次の文字列が付加されます。

1. 文字列 ARMv
2. ARM 命令セットのバージョン番号
3. 含まれるバリエーションを示すバリエーション文字
4. これらに加え、ARMv5TExP バリエーションではいくつかの命令が除外されていることを示すため、x の後に文字 P が使用されています。

本書で説明されている現行の（現在でも使用されている）ARM/Thumb アーキテクチャバージョンの標準名の一覧を、P. -xvii の表*「アーキテクチャのバージョン」*に示します。これらの名前によって、ARM プロセッサに実装されている正確な命令セットを簡単に識別できます。ただし、本書ではアーキテクチャ名の一覧を並べることを避けるため、*アーキテクチャバージョン 4 以降の T バリエーション*などの説明的な語句を一般に使用します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ARMv4 以前のアーキテクチャ名はすべて、現在は*使用されていません*。本書を通して、**すべて**という用語は、ARMv4 以後のすべてのアーキテクチャバージョンを指します。

アーキテクチャのバージョン

名前	ARM 命令セットのバージョン	Thumb 命令セットのバージョン	注
ARMv4	4	なし	-
ARMv4T	4	1	-
ARMv5T	5	2	-
ARMv5TExP	5	2	LDRD、MCRR、MRRC、PLD、STRD 以外の拡張 DSP 命令
ARMv5TE	5	2	拡張 DSP 命令
ARMv5TEJ	5	2	ARMv5TE に加え、BXJ命令とJazelle拡張機能のサポート
ARMv6	6	3	追加の命令の一覧を P. A4-287 表 A4-2 と P. A7-125 表 A7-1 に示します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

本書の使用法

本書に記載されている情報は次の 4 つのパートに分かれています。

パート A - CPU のアーキテクチャ

パート A は ARM と Thumb の命令セットの説明で、次の章に分かれています。

第 A1 章 ARM アーキテクチャと、ARM と Thumb の命令セットの概要について説明します。

第 A2 章 ARM 命令で扱う値のタイプ、それらの値を保持する汎用レジスタ、プログラムステータスレジスタについて説明します。この章では、ARM プロセッサが割り込みや他の例外を処理する方法、エンディアンとアンアラインドアクセスのサポート、+ 同期化基本命令の情報、Jazelle® 拡張機能についても説明します。

第 A3 章 ARM 命令セットについて、命令タイプごとに説明します。

第 A4 章 各 ARM 命令の詳細なリファレンス資料。命令はニーモニックのアルファベット順に解説されています。

第 A5 章 ARM 命令で使用されるアドレシングモードの詳細なリファレンス資料。本書では*アドレシングモード*という用語を広義に解釈し、多くの異なる命令によって共有される、その命令で使用する値を生成する手順を指しています。この章で説明しているアドレシングモードのうち 4 つでは、生成される値はメモリアドレスです。これはアドレシングモードの伝統的な役割です。最後のアドレシングモードは、データ処理命令でオペランドとして使用される値を生成します。

第 A6 章 Thumb 命令セットについて、命令タイプごとに説明します。この章では、ARM と Thumb の命令セットの切り替えを行う方法と、Thumb 状態での実行中に発生した例外の処理方法についても説明します。

第 A7 章 各 Thumb 命令の詳細なリファレンス資料。命令はニーモニックのアルファベット順に解説されています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

パート B - メモリとシステムアーキテクチャ

パート B では、ARM ベースのシステムでシステム制御コプロセッサ（コプロセッサ 15）により一般に実装される、標準的なメモリシステム機能について説明します。パートBは次の章に分かれています。

第 B1 章 メモリとシステムアーキテクチャの概要について説明します。

第 B2 章 メモリオーダモデルの概要について説明します。

第 B3 章 システム制御コプロセッサとその用法について全般的に説明します。

第 B4 章 *メモリ管理ユニット*（MMU）をベースとした*仮想メモリシステムアーキテクチャ*（VMSA）を使用する標準のARMメモリとシステムアーキテクチャについて説明します。

第 B5 章 *メモリ保護ユニット*（MPU）をベースとした、より単純な*保護メモリシステムアーキテクチャ*（PMSA）について説明します。

第 B6 章 ARM メモリシステムでキャッシュとライトバッファを制御する、標準的な方法について説明します。この章は、MMU をベースとしたシステムと、MPU をベースとしたシステムの両方に適用されます。

第 B7 章 レベル1メモリの*密結合メモリ*(TCM)アーキテクチャオプションについて説明します。

第 B8 章 高速コンテキストスイッチ拡張機能とコンテキスト ID サポート（ARMv6 のみ）について説明します。

パート C - ベクタ浮動小数点アーキテクチャ

パート C では、*ベクタ浮動小数点*（VFP）アーキテクチャについて説明します。これは、一般的なグラフィックや DSP アルゴリズムで高性能の浮動小数点演算を行うように設計された ARM アーキテクチャのコプロセッサ拡張機能です。

第 C1 章 VFP アーキテクチャの概要と、IEE 754-1985 浮動小数点演算標準への準拠について説明します。

第 C2 章 VFP 命令セットでサポートされる浮動小数点形式、それらの値を保持する浮動小数点汎用レジスタ、VFP システムレジスタについて説明します。

第 C3 章 VFP コプロセッサ命令セットについて、命令タイプごとに説明します。

第 C4 章 VFP コプロセッサ命令セットの詳細なリファレンス資料。命令はニーモニックのアルファベット順に解説されています。

第 C5 章 VFP 命令で使用されるアドレシングモードの詳細なリファレンス資料。1 つは従来のアドレシングモードで、ロード / ストア命令のアドレスを生成します。他のアドレシングモードは、浮動小数点汎用レジスタと命令を使用して浮動小数点数値のベクタを保持し、計算を実行する方法を指定します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

パート D - デバッグアーキテクチャ

パート D では、デバッグアーキテクチャについて説明します。これは ARM アーキテクチャのコプロセッサ拡張機能で、構成、ブレークポイントとウォッチポイントのサポート、デバッグホストへの*デバッグ通信チャネル*（DCC）を提供するように設計されています。

第 D1 章 デバッグアーキテクチャの概要について説明します。

第 D2 章 デバッグアーキテクチャの主要な機能について説明します。

第 D3 章 デバッグアーキテクチャのコプロセッサデバッグレジスタサポート（CP14）について説明します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表記規則

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

書体の一般的な規則

`typewriter` アセンブラ構文の表記、命令の擬似コードの表記、ソースコードの例に使用されています。アセンブラ構文と擬似コードの表記では、以下に示す追加の表記規則も使用されます。

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

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

太字 説明の一覧、その他適切な場所で強調のため使用されます。

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

命令の擬似コード表記

命令の動作を正確に説明するため、擬似コード形式が使用されています。この擬似コードは `typewriter` フォントで表記され、明確で簡潔にするために次の表記規則が使用されています。

- 構造を示すために段下げが使用されています。たとえば、`for` 文でループが繰り返される文の範囲は、`for` 文から、`for` 文と同じまたはより低い段下げレベルにある次の文までです（両端を含みません）。
- コメントは C 言語と同様に `/*` と `*/` で囲まれています。
- 他の方法で記述することが困難な機能を記述するため、コメントの外側で英語のテキストが使用されていることがあります。
- 擬似コードで使用されるキーワードと特別な関数は、すべて*用語集*で解説されています。
- 代入と等価テストは、C 言語と同様に代入が `=`、等価テストが `==` で区別されています。
- 命令のフィールドは、命令のエンコード図に示されている名前で参照されます。命令のフィールドがレジスタを示している場合、その場所で特に記載がない限り、それへの参照はレジスタ番号ではなくそのレジスタの値を意味します。たとえば、`Rn == 0` テストは、指定のレジスタの値が 0 かどうかのチェックを意味しますが、`Rd is R15` テストは、指定されたレジスタがレジスタ 15 かどうかのチェックを意味します。
- 命令があるアドレシングモードを使用している場合、そのアドレシングモードの擬似コードは、その命令の擬似コードで使用される 1 つ以上の値を生成します。たとえば、P. A4-8*「AND」*で説明されている AND 命令は ARM のアドレシングモード 1 を使用します。詳細については、P. A5-2*「アドレシングモード 1 - データ処理オペランド」*を参照して下さい。アドレシングモードの擬似コードは、`shifter_operand` と `shifter_carry_out` の 2 つの値を生成し、これらの値は AND 命令の擬似コードで使用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アセンブラ構文の記述

本書には、アセンブラ命令と、アセンブラ命令のコンポーネントを示すため、多数の構文の記述が含まれています。これらは typewriter フォントで、次のように記述されます。

< >
: < と > で囲まれたアイテムは、その位置にユーザによって指定される値のタイプの短い説明です。通常、アイテムの詳細な説明は以降のテキストにあります。このようなアイテムは多くの場合、命令のエンコード図に示されている同様の名前のフィールドに対応しています。この対応が、整数値やレジスタ番号のバイナリエンコードを命令のエンコードに置き換えるだけの場合、明示的な説明はありません。たとえば、ARM 命令のアセンブラ構文に <Rn> というアイテムが含まれており、命令のエンコード図に Rn という名前の 4 ビットフィールドが含まれている場合、アセンブラ構文で指定されたレジスタ番号は、バイナリとして命令フィールドにエンコードされます。

  アセンブラ構文のアイテムと命令エンコードとの対応が、整数やレジスタ番号を単純にバイナリエンコードするよりも複雑な場合、アイテムの説明にエンコードの方法が記載されています。

{ }
: { と } で囲まれているアイテムはオプションです。通常、アイテムの説明と、このアイテムの存在 / 非存在が命令にエンコードされる方法は、以降のテキストにあります。

|
: 代替可能な文字列を示します。たとえば、LDM|STM は LDM または STM を意味します。

スペース
: 1 つのスペースは、アイテムを分離して読みやすくするために使用されます。アセンブラ構文でスペースが必須な場合、連続した 2 つ以上のスペースが使用されます。

+/-
: オプションの + または - 符号を示します。どちらもコードに指定されていない場合、+ と見なされます。

*
: <immed_8> * 4 などの組み合わせで使用されている場合、イミディエート値で、かつある数値範囲からの値の指定倍である必要があることを示します。この場合、数値の範囲は 0 ～ 255（8 ビットイミディエートとして表現可能な値）で、指定の倍数は 4 なため、この値は 4 * 0 = 0 から 4 * 255 = 1020 までの範囲で、4 の倍数の必要があります。

他の文字はすべて、アセンブラ構文に表記されているとおり正確にエンコードされる必要があります。{ と } を除いて、上に説明されている特殊文字は、本書に記載されているアセンブラ命令の基本形には登場しません。{ と } の文字は、いくつかの場所で変数アイテムの一部としてエンコードされる必要があります。この場合、変数アイテムの詳細説明に、それらに指定された使用方法の説明があります。

——— 注 ———

本書では、アセンブラ命令構文の最も基本的な形式のみについて説明します。一般に実際のアセンブラは、はるかに広範な命令の構文、アセンブリ処理、シンボリックな操作およびマクロ展開などの追加機能を制御する各種ディレクティブを認識します。これらはすべて、本書の範囲外です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

参考資料

このセクションでは、ARM ファミリのプロセッサに関する関連情報が記載されている、ARM 社やサードパーティからの出版物を紹介します。

ARM は定期的にドキュメントの更新と修正を行います。最新の誤記シートと補遺、ARM によく寄せられる質問（FAQ）は `http://www.arm.com` で参照できます。

ARM の刊行物

ARM 外部デバッグインターフェース仕様

他の出版物

次の書籍は本書で参照されているか、関連情報が記載されています。

- *IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors*, IEEE Std 1596.5-1993, (ISBN 1-55937-354-7、IEEE)
- *The Java™ Virtual Machine Specification* Second Edition, Tim Lindholm and Frank Yellin, Addison Wesley より出版 (ISBN 0-201-43294-3)
- JTAG Specification IEEE 1149.1

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ご意見・ご質問

ARM 社では、本書に関するご意見・ご質問をお待ちしています。

本書に関するご意見

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

- 資料名
- 資料番号
- ご意見のあるページ番号
- 問題の詳しい説明

補足または改善すべき点についてのご提案もお待ちしています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

パート A

CPU のアーキテクチャ

第 A1 章
ARM アーキテクチャの概要

本章ではARM® アーキテクチャの概要を紹介します。本章は以下のセクションから構成されています。

- *ARM アーキテクチャについて*：P. A1-2
- *ARM 命令セット*：P. A1-6
- *Thumb 命令セット*：P. A1-11

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A1.1 ARM アーキテクチャについて

ARM アーキテクチャの進化により、広範な性能範囲の製品への実装が可能になりました。これまでに出荷された ARM プロセッサは 20 億個を超え、市場の多くの分野で最も有力なアーキテクチャの地位を占めるようになっています。ARM プロセッサはアーキテクチャが単純なため、従来から非常に小型の実装を実現しており、消費電力が極めて低いデバイスの製造が可能です。実装サイズが小さく、高性能で、消費電力が極めて低い点は、現在でも ARM アーキテクチャの重要な利点です。

ARM は*縮小命令セットコンピュータ*(RISC) で、RISC アーキテクチャに典型的な次の特徴があります。

- 大型で均一なレジスタファイル。
- *ロード / ストア*アーキテクチャ。データ処理はレジスタの内容に対してのみ実行され、メモリの内容が直接処理されることはありません。
- 単純なアドレシングモード。ロード / ストアのアドレスはすべて、レジスタの内容と命令のフィールドから計算されます。
- 命令フィールドが単一形式で固定長なため、命令のデコードが簡素化されています。

また、ARM アーキテクチャには次のような特徴があります。

- ほとんどのデータ処理命令で*算術論理回路*（ALU）とシフタを使用できるため、これらの機能を最大限に活用できます。
- 自動インクリメント / 自動デクリメントアドレシングモードにより、プログラムループを最適化できます。
- 複数ロード / ストア命令を使用して、最大のデータスループットを得られます。
- ほとんどの命令を条件付きで実行可能なため、最大の実行スループットを実現できます。

ARM プロセッサは、基本的な RISC アーキテクチャに上記のような拡張を加えることで、高性能、小さなコードサイズ、低消費電力、小さなシリコン面積を、バランス良く実現しています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A1.1.1 ARM レジスタ

ARM には、31 個の汎用 32 ビットレジスタがあります。どの時点でも、これらのレジスタの中で 16 個が参照可能です。他のレジスタは、例外処理の高速化に使用されます。ARM 命令に含まれるレジスタ指定子はすべて、参照可能な 16 個のレジスタのどれでも指定できます。

レジスタ 16 個のメインバンクは、すべての非特権コードによって使用されます。これらはユーザモードレジスタです。ユーザモードは他のモードと違い、特権を持ちません。これは、次のことを意味します。

- ユーザモードは、例外の生成によってのみ他のプロセッサモードに切り替え可能です。SWI 命令はプログラムコントロールからこの機能を提供します。
- ユーザモードでは、メモリとコプロセッサ機能へのアクセスが、特権モードに比べて制限されます。

16 個の可視レジスタのうち 3 つは、特定の役割が指定されています。

スタックポインタ ソフトウェアは通常、R13 を*スタックポインタ*（SP）として使用します。R13 は、T バリエーションの PUSH 命令と POP 命令、ARMv6 以降の SRS 命令と RFE 命令で使用されます。

リンクレジスタ レジスタ 14 は*リンクレジスタ*（LR）です。サブルーチンコールを実行するリンク付き分岐（BL または BLX）命令を実行した後で、次の命令のアドレスがこのレジスタに保持されます。また、このレジスタは例外モードへのエントリ時に、復帰アドレス情報を保持する目的でも使用されます。これらの場合以外は常に、R14 は汎用レジスタとして使用できます。

プログラムカウンタ レジスタ 15 は*プログラムカウンタ*（PC）です。このレジスタはほとんどの命令において、現在実行されている命令の 2 つ後の命令を指すポインタとして使用できます。ARM ステートでは、すべての ARM 命令が 4 バイト長（32 ビットワード 1 つ）で、常にワード境界にアラインしています。つまり、PC の最下位 2 ビットは常に 0 となるため、PC 上位 30 ビットのみが変化します。アーキテクチャの一部のバージョンでは、これ以外に 2 つのプロセッサステートが使用されます。T バリエーションでは Thumb® ステート、J バリエーションでは Jazelle® ステートが使用されます。PC は、Thumb 状態ではハーフワード（16 ビット）、Jazelle 状態ではバイトでアラインされます。

他の 13 個のレジスタは、ハードウェアで特定の目的に使用することはありません。これらのレジスタの用途は、ソフトウェアによって決められます。レジスタの詳細については、P. A2-4 *「レジスタ」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A1.1.2 例外

ARM には 7 種類の例外があり、それぞれに対応する特権処理モードがあります。例外のタイプは次の 7 つです。

- リセット
- 未定義命令の実行
- ソフトウェア割り込み（`SWI`）命令 : オペレーティングシステムコールに使用されます。
- プリフェッチアボート : 命令フェッチメモリアボート
- データアボート : データアクセスメモリアボート
- IRQ : 通常割り込み
- FIQ : 高速割り込み

例外が発生すると、標準レジスタのうちいくつかは例外モード専用のレジスタに置き換わります。すべての例外モードには、R13 と R14 を置き換える*バンクレジスタ*があります。高速割り込みモードでは、これらに加えて高速割り込み処理用のバンクレジスタが使用されます。

例外ハンドラへのエントリ時に、例外処理の復帰アドレスは R14 に保持されています。復帰アドレスは、例外処理の完了後に復帰するアドレスで、例外を発生した命令の参照にも使用されます。

レジスタ 13 はそれぞれの例外ハンドラのプライベートスタックポインタとなるため、すべての例外モードで別のバンクが使用されます。高速割り込みモードではレジスタ 8 ～ 12 も別のバンクに切り替えられるため、これらのレジスタの保存や復元の必要なしに割り込み処理を開始できます。

システムモードは、ユーザモードレジスタを使用する 6 番目の特権処理モードです。このモードは、メモリ、コプロセッサ、または両方への特権アクセスを必要とするタスクの実行に使用され、どの例外がそのタスクの間に発生できるかに関する制約がありません。

これに加え、リセットも SWI と同じ特権モードを使用します。

例外の詳細については、P. A2-16*「例外」*を参照して下さい。

例外処理

例外が発生すると、ARM プロセッサは所定の方法で実行を停止し、*例外ベクタ*と呼ばれる、メモリ内の多数の固定アドレスのうちの 1 つから実行を開始します。ベクタのアドレスは、リセットを含め、例外ごとに異なっています。システムが通常に実行されている場合（セクション A2.6 を参照）と、デバッグイベント（パート D の第 3 章*コプロセッサ 14 : デバッグコプロセッサ*参照）の場合について、それぞれの動作が定められています。

オペレーティングシステムは、初期化時にすべての例外のハンドラをインストールします。オペレーティングシステム内で例外が発生してもステートが消失しないようにするため、特権オペレーティングシステムタスクは一般にシステムモードで実行されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A1.1.3 ステータスレジスタ

汎用レジスタの内容以外のプロセッサステータスはすべて、*ステータスレジスタ*に保持されます。現在のプロセッサステータスは、*カレントプログラムステータスレジスタ*（CPSR）にあります。CPSR には以下の情報が保持されます。

- 4 つの条件コードフラグ（ネガティブ、ゼロ、キャリー、オーバフロー）
- 1 つのスティッキー (Q) フラグ（ARMv5 以降のみ）: このフラグは、飽和が飽和算術演算命令で発生したのか、特定の積和命令における符号付きオーバフローで発生したのかを示しています。
- 4 つの GE（大きいかより等しい）フラグ（ARMv6 以降のみ）: これらのフラグには、並列命令の各演算について次のような条件がエンコードされます。
  - — 符号付き演算の結果が非負であるか。
  - — 符号なし演算によりキャリーまたは桁上がりが発生したか。
- 各タイプの割り込み（ARMv5 以前では 2 つ）に 1 つ、合計 2 つの割り込みディセーブルビット
- 1 つの不正確なアボートマスクビット (A)（ARMv6 以降）
- 現在のプロセッサモードをエンコードする 5 ビット
- 現在実行されているオペコードが ARM 命令、Thumb 命令、Jazelle 命令のどれかをエンコードする 2 ビット
- ロード / ストア操作のエンディアンを制御する 1 ビット（ARMv6 以降のみ）

各例外モードには、例外発生直前のタスクの CPSR を保持する*保存プログラムステータスレジスタ*（SPSR）もあります。CPSR と SPSR には、専用の命令でアクセスします。

ステータスレジスタの詳細については、P. A2-11「*プログラムステータスレジスタ*」を参照して下さい。

表 A1-1 ステータスレジスタの概要

フィールド	説明	アーキテクチャ
N Z C V	条件コードフラグ	すべて
J	Jazelle 状態フラグ	5TEJ 以降
GE[3:0]	SIMD 条件フラグ	6
E	ロード / ストアのエンディアン形式	6
A	不正確なアボートマスク	6
I	IRQ 割り込みマスク	すべて
F	FIQ 割り込みマスク	すべて
T	Thumb 状態フラグ	4T 以降
モード [4:0]	プロセッサモード	すべて

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A1.2 ARM 命令セット

ARM 命令セットに含まれる命令は、大きく 6 つのタイプに分類されます。

- *分岐命令*
- *データ処理命令* : P. A1-7
- *ステータスレジスタ転送命令* : P. A1-8
- *ロード/ストア命令* : P. A1-8
- *コプロセッサ命令* : P. A1-10
- *例外生成命令* : P. A1-10

ほとんどのデータ処理命令と、1 つのタイプのコプロセッサ命令は、操作の結果に基づいて、CPSR の 4 つの条件コードフラグ（ネガティブ、ゼロ、キャリー、オーバフロー）を更新します。

ほとんどの ARM 命令には、4 ビットの*条件*フィールドが含まれています。このフィールドの値のうち 1 つは、命令が無条件に実行されることを指定します。

他の 14 の値は、命令が*条件付き実行*であることを指定します。命令の実行開始時に、対応する条件が真であることが条件コードフラグにより示されている場合、命令は通常どおり実行されます。それ以外の場合、命令は実行されません。これらの 14 の条件を使用して、以下のような操作を行えます。

- 等価および非等価のテスト
- 符号付き、符号なしの両方の算術演算における不等価 <、<=、>、>= のテスト
- 各条件コードフラグの個別テスト

条件フィールドの 16 番目の値は、代替命令をエンコードするものです。これらの代替命令の条件付き実行はできません。ARMv5 以前では、これらの命令の動作は予測不能です。

A1.2.1 分岐命令

多くのデータ処理命令やロード命令を使用して PC に書き込みを行い、制御フローを変更できます。その他に標準の分岐命令があり、24 ビットの符号付きワードオフセットを使用して、前方と後方のどちらにも最大 32MB までの分岐が可能です。

リンク付き分岐（BL）命令も使用できます。この命令では分岐の実行後、分岐の次にある命令のアドレスが R14（LR）に保存されます。この命令はサブルーチンコールに使用され、LR の内容を PC にコピーすることでサブルーチンから復帰できます。

命令セットを切り替え、分岐先で Thumb 命令セットまたは Jazelle オペコードを使って実行を継続する分岐命令もあります。Thumb に対応している場合、ARM コードが Thumb サブルーチンを呼び出すことも、ARM サブルーチンから Thumb の呼び出し元に戻ることも可能です。Thumb 命令セットにも同様の命令があり、Thumb → ARM の切り替えが可能です。Thumb 命令セットの概要については、第 A6 章「*Thumb 命令セット*」を参照して下さい。

ARMv5 の J バリエーションで導入され、ARMv6 にも存在する BXJ 命令は、プロセッサを Jazelle ステートに移行し、それに伴って CPSR の J フラグをアサートします。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A1.2.2 データ処理命令

データ処理命令は、汎用レジスタの内容に対して計算を実行します。データ処理命令には、5 つのタイプがあります。

- *算術 / 論理命令*
- *比較命令*
- *SIMD（複数データ並列処理）命令*
- *乗算命令* : P. A1-8
- *その他のデータ処理命令* : P. A1-8

算術 / 論理命令

以下の算術 / 論理命令には共通の命令フォーマットがあります。これらの命令は、2 つまでのソースオペランドに対して算術演算または論理演算を実行し、結果をデスティネーションレジスタに書き込みます。結果に基づき、必要であれば条件コードフラグを更新することもできます。

ソースオペランド 2 つのうち、

- 1 つは常にレジスタです。
- もう 1 つは、次の 2 つのいずれかです。
  - — イミディエート値
  - — オプションでシフトされるレジスタ値

オペランドがシフトしたレジスタの場合、シフトの数はイミディエート値または別のレジスタの値で指定されます。シフトは 5 タイプを指定可能です。したがって、算術 / 論理命令はすべて、算術 / 論理演算とシフト演算を実行できます。このため、ARM にシフト専用命令はありません。

プログラムカウンタ（PC）は汎用レジスタなため、算術 / 論理命令の結果を直接 PC に書き込むことができます。このため、各種のジャンプ命令を簡単に実装できます。

比較命令

比較命令は、算術 / 論理命令と同じ命令フォーマットを使用します。比較命令は、2 つまでのソースオペランドに対して算術演算または論理演算を実行しますが、結果をレジスタに書き込みません。比較命令は、常に結果に基づいて条件フラグを更新します。

比較命令のソースオペランドは算術 / 論理命令と同じ形式で、シフト演算を組み込むこともできます。

SIMD（複数データ並列処理）命令

この加算命令と減算命令は、各オペランドを 2 つの並列な 16 ビット数、または 4 つの並列な 8 ビット数として扱います。これらの数は、符号付きとしても、符号なしとしても扱うことができます。飽和演算とラップアラウンド演算のどちらも実行でき、オーバフローを防ぐために結果を半分にすることも可能です。

これらの命令は、ARMv6 で追加されたものです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

乗算命令

乗算命令にはいくつかの種類があり、それぞれアーキテクチャに追加された時期が異なっています。詳細については、P. A3-10「*乗算命令*」を参照して下さい。

その他のデータ処理命令

その他の命令には、先行ゼロカウント（CLZ）命令、符号なし差の絶対値の合計命令、オプションの累算付き符号なし差の絶対値の合計命令（USAD8、USADA8）があります。

A1.2.3　ステータスレジスタ転送命令

ステータスレジスタ転送命令は、CPSR または SPSR と汎用レジスタとの間で内容を転送します。CPSR への書き込みにより、以下のような操作が行えます。

- 条件コードフラグの値のセット
- 割り込みイネーブルビットの値のセット
- プロセッサのモードとステートのセット
- ロード / ストア操作のエンディアン形式の変更

A1.2.4　ロード / ストア命令

ロード / ストア命令には、次のような種類があります。

- *レジスタのロード / ストア*
- *複数レジスタのロード / ストア*：P. A1-9
- *レジスタの排他ロード / ストア*：P. A1-9

スワップ命令とスワップバイト命令もありますが、ARMv6 での使用は推奨されません。すべてのソフトウェアについて、レジスタ排他ロード / ストア命令に移行することをお勧めします。

レジスタのロード / ストア

レジスタロード命令は、64 ビットのダブルワード、32 ビットのワード、16 ビットのハーフワード、8 ビットのバイトのいずれかを、メモリから 1 つ以上のレジスタにロードします。バイトとハーフワードをロードする場合、ロード時に自動的にゼロ拡張または符号拡張を実行できます。

レジスタストア命令は、64 ビットのダブルワード、32 ビットのワード、16 ビットのハーフワード、8 ビットのバイトのいずれかを、1 つ以上のレジスタからメモリにストアします。

ARMv6 以降は、指定されたバイトアドレスにアクセスする、ワードとハーフワードのアンアラインドロード / ストアにも対応しています。ARMv6 以前は、32 ビットのアンアラインドロードを行った場合、ロードされるデータがローテートされ、32 ビットストアは常にアラインされてストアされます。他の関連命令の動作は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

レジスタロード / ストア命令には、3 つの基本アドレシングモードがあります。これらはすべて、命令で指定されている*ベースレジスタ*と*オフセット*を使用します。

- *オフセットアドレシング*では、ベースレジスタの値にオフセットを加算または減算してメモリアドレスが計算されます。
- *プリインデクスアドレシングでは*、オフセットアドレシングと同様にメモリアドレスが計算されます。同時に、計算されたメモリアドレスがベースレジスタに書き戻されます。
- *ポストインデクスアドレシングでは*、メモリアドレスはベースレジスタ値です。アドレス計算後に、ベースレジスタの値にオフセットが加算または減算され、結果がベースレジスタに書き戻されます。

どのケースでも、オフセットはイミディエート値、または*インデクスレジスタ*の値です。レジスタベースのオフセットも、シフト演算でスケールを変更可能です。

PC は汎用レジスタとして扱われるため、32 ビット値を直接 PC にロードし、4GB のメモリ空間の任意のアドレスにジャンプを実行することが可能です。

複数レジスタのロード / ストア

複数ロード（LDM）命令と複数ストア（STM）命令は、任意の数の汎用レジスタをメモリとの間でブロック転送します。これらの命令のアドレシングモードは 4 つです。

- プリインクリメント
- ポストインクリメント
- プリデクリメント
- ポストデクリメント

ベースアドレスはレジスタの値によって指定されます。必要に応じて、この値を転送後に更新することもできます。サブルーチン復帰アドレスと PC の値が汎用レジスタにあるため、LDM と STM を使用して、非常に効率的なサブルーチンのエントリシーケンスとイグジット（終了）シーケンスを構築できます。

- サブルーチンのエントリ部分にある 1 つの STM 命令で、レジスタ内容と復帰アドレスをスタックにプッシュし、同時にスタックポインタを更新できます。
- サブルーチンのイグジット（終了）部分にある 1 つの LDM 命令で、スタックからレジスタ内容を復元し、PC に復帰アドレスをロードし、スタックポインタを更新できます。

LDM 命令と STM 命令を使用すると、ブロックコピーや同様のデータ移動アルゴリズムについても、非常に効率的なコードを作成できます。

レジスタの排他ロード / ストア

これらの命令は、協調的なメモリ同期化をサポートしています。これらの命令の目的は、ロードフェーズとストアフェーズの間ですべてのシステムリソースをロックすることなく、セマフォに必要とされるアトミックな動作を実現することです。詳細については、P. A4-53「*LDREX*」と P. A4-203「*STREX*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A1.2.5 コプロセッサ命令

コプロセッサ命令には 3 つのタイプがあります。

データ処理命令

コプロセッサ固有の内部演算を開始します。

データ転送命令

コプロセッサとメモリとの間でデータを転送します。転送のアドレスは、ARM プロセッサにより計算されます。

レジスタ転送命令

コプロセッサのレジスタと、ARM のレジスタまたはレジスタのペアとの間で値を転送します。

A1.2.6 例外生成命令

特定の例外を発生させる命令には、2 つのタイプがあります。

ソフトウェア割り込み命令

`SWI` 命令は、ソフトウェア割り込み例外を発生させます。この命令は通常、OS で定義されているサービスを要求するオペレーティングシステムコールに使用されます。`SWI` 命令により、例外へのエントリと同時にプロセッサが特権モードに移行します。これにより、特権のないタスクが特権機能にアクセスできますが、アクセスは OS の許容する方法に限定されます。

ソフトウェアブレークポイント命令

`BKPT` 命令は、アボート例外を発生させます。アボートベクタに適切なデバッガソフトウェアがインストールされている場合、この形式で発生したアボート例外はブレークポイントとして扱われます。システムにデバッグハードウェアが存在する場合は、`BKPT` 命令を直接ブレークポイントとして扱うため、アボート例外は発生しません。

前に述べたほか、以下のタイプの命令は未定義命令例外を引き起こします。

- どのハードウェアコプロセッサによっても認識されないコプロセッサ命令
- ARM 命令として割り当てられていないほとんどの命令語

どちらの場合も、発生する例外は、対応するエラーを発生するか、命令のソフトウェアエミュレーションを開始します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A1.3 Thumb 命令セット

Thumb 命令セットは ARM 命令セットのサブセットで、各命令は 32 ビットではなく 16 ビットでエンコードされています。詳細については、第 A6 章「*Thumb 命令セット*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 A2 章
プログラマモデル

本章では、ARM® のプログラマモデル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。

- *データタイプ*：P. A2-2
- *プロセッサモード*：P. A2-3
- *レジスタ*：P. A2-4
- *汎用レジスタ*：P. A2-6
- *プログラムステータスレジスタ*：P. A2-11
- *例外*：P. A2-16
- *エンディアンのサポート*：P. A2-30
- *アンアラインドアクセスのサポート*：P. A2-38
- *同期化基本命令*：P. A2-44
- *Jazelle 拡張*：P. A2-53
- *飽和整数算術演算*：P. A2-69

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.1 データタイプ

ARM プロセッサは、次のデータタイプをサポートしています。

バイト　8 ビット

ハーフワード　16 ビット

ワード　32 ビット

—— 注 ——

- ハーフワードのサポートは、バージョン 4 で導入されたものです。
- ARMv6　では、ワードとハーフワードについて、アンアラインドデータがサポートされています。詳細については、P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。
- データタイプが*符号なし*と記載されている場合、N ビットのデータ値は、通常のバイナリ形式で $0 \sim +2^{N} - 1$ の範囲にある負でない整数です。
- データタイプが*符号付き*と記載されている場合、N ビットのデータ値は、2 の補数形式で $-2^{N-1} \sim +2^{N-1} - 1$ の範囲にある整数です。
- `ADD`　など、ほとんどのデータ演算は、ワード単位で実行されます。ロング乗算は、累算付き、または累算なしの 64 ビットの結果をサポートしています。ARMv5TE では、いくつかのハーフワード乗算が追加されています。ARMv6 では、ハーフワード 2 つまたはバイト 4 つを同時に処理する各種の SIMD 命令が追加されています。
- ロード / ストア操作は、バイト、ハーフワード、ワードをレジスタとメモリとの間で転送します。バイトまたはハーフワードのロード時には、自動的にゼロ拡張または符号拡張が行われます。2 つ以上のワードをメモリとの間で転送するロード / ストア操作も存在します。
- ARM 命令は常に 1 ワードで、4 バイト境界にアラインされています。Thumb® 命令は常にハーフワードで、2 バイト境界にアラインされています。Jazelle® オペコードの長さは可変数のバイトで、任意のアライメントに配置されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.2 プロセッサモード

ARM アーキテクチャは、表 A2-1 に示されている 7 つのプロセッサモードに対応しています。

表 A2-1 ARM プロセッサモード

プロセッサモード		モード番号	説明
ユーザ	usr	`0b10000`	通常のプログラム実行モード
FIQ	fiq	`0b10001`	高速なデータ転送またはチャネルプロセスに対応
IRQ	irq	`0b10010`	汎用割り込み処理に使用
スーパバイザ	svc	`0b10011`	オペレーティングシステム用の保護モード
アボート	abt	`0b10111`	仮想メモリ、メモリ保護、または両方を実装
未定義	und	`0b11011`	ハードウェアコプロセッサのソフトウェアエミュレーションに使用
システム	sys	`0b11111`	特権オペレーティングシステムタスクを実行（ARMv4 以降）

モード変更は、ソフトウェア制御のもとで行われる場合と、外部割り込みまたは例外処理が原因で発生する場合があります。

ほとんどのアプリケーションプログラムは、ユーザモードで実行されます。プロセッサがユーザモードのとき、実行中のプログラムが一部の保護されているシステムリソースにアクセスしたり、モードを変更したりするには、例外を発生する必要があります（P. A2-16 *「例外」*参照）。これにより、オペレーティングシステムが適切に記述されていれば、システムリソースの使用を制御できます。

ユーザモード以外のモードは、*特権モード*と呼ばれます。特権モードはシステムリソースにフルアクセスが可能で、モードも自由に変更できます。特権モードのうち 5 種類は、*例外モード*と呼ばれます。

- FIQ
- IRQ
- スーパバイザ
- アボート
- 未定義

特定の例外が発生すると、対応する例外モードへのエントリが行われます。各例外モードには、例外発生時にユーザモード状態が破壊されないよう、いくつかの追加レジスタが用意されています。詳細については、P. A2-4 *「レジスタ」*を参照して下さい。

他に、システムモードと呼ばれるモードがあります。システムモードは例外によってエントリされるものではなく、レジスタはユーザモードと完全に同じです。ただし、システムモードは特権モードであり、ユーザモードの制約を受けません。システムモードは、システムリソースへのアクセスを必要とすると同時に、例外モードの追加レジスタの使用を希望しないオペレーティングシステムタスクで使用することを目的としています。追加レジスタを使用しないため、例外が発生してもタスク状態は破壊されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

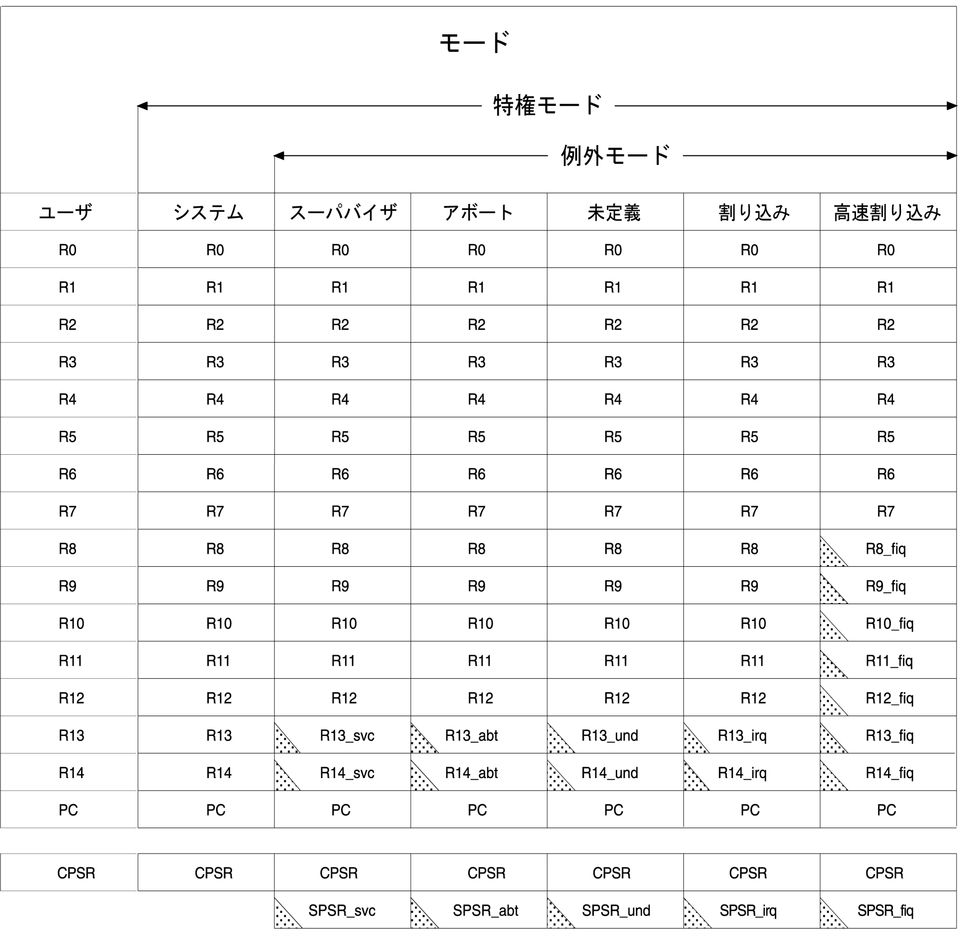
A2.3 レジスタ

ARM プロセッサには、合計 37 個のレジスタがあります。

- プログラムカウンタを含む、31 個の汎用レジスタ。汎用レジスタは 32 ビット幅です。詳細については、P. A2-6 *「汎用レジスタ」*を参照して下さい。
- 6 つのステータスレジスタ。ステータスレジスタも 32 ビット幅ですが、32 ビットのうち割当てられている、または実装する必要があるのは一部だけです。どの部分が割り当てられているかは、アーキテクチャのバリエーションによって異なります。ステータスレジスタについては、P. A2-11 *「プログラムステータスレジスタ」*を参照して下さい。

レジスタは、部分的にオーバラップしたバンクに配置されています。使用可能なバンクは、現在のプロセッサモードにより決定されます。プロセッサモードとレジスタバンクの関係を P. A2-5 図 A2-1 に示します。どの時点でも、15 個の汎用レジスタ（R0 ～ R14）、1 つまたは 2 つのステータスレジスタ、プログラムカウンタが使用できます。P. A2-5 図 A2-1 の各列は、各プロセッサモードで使用できる汎用レジスタとステータスレジスタを示しています。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.



モード

	特権モード					
		例外モード				
ユーザ	システム	スーパバイザ	アボート	未定義	割り込み	高速割り込み
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC

ユーザ	システム	スーパバイザ	アボート	未定義	割り込み	高速割り込み
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

ユーザまたはシステムモードで使用される通常のレジスタが、例外モードに固有の代替レジスタに置き換えられていることを示します。

図 A2-1 レジスタの構成

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.4 汎用レジスタ

R0 ～ R15 の汎用レジスタは、3 つのグループに分けられます。各グループは、バンク切り替えの方法が異なっており、一部のグループは特別な用途が割り当てられています。

- *バンクなしレジスタ : R0 ～ R7*
- *バンクレジスタ : R8 ～ R14*
- レジスタ 15（PC）については、P. A2-9「*レジスタ 15 とプログラムカウンタ*」を参照して下さい。

A2.4.1 バンクなしレジスタ : R0 ～ R7

レジスタ R0 ～ R7 は、*バンクなしレジスタ*です。これらのレジスタは、すべてのプロセッサモードで同じ 32 ビットの物理レジスタを参照します。これらは完全な汎用レジスタで、アーキテクチャで特定の用途に割り当てられていないため、汎用レジスタを指定できる命令で常に使用可能です。

A2.4.2 バンクレジスタ : R8 ～ R14

レジスタ R8 ～ R14 は、*バンクレジスタ*です。これらのレジスタが参照する物理レジスタは、現在のプロセッサモードによって異なります。現在のプロセッサモードに関係なく、特定の物理レジスタを指す場合、以下で説明するような特定の名称が使用されます。ほとんどの命令では、汎用レジスタの使用が可能であれば、バンクレジスタの使用も可能です。

——— 注 ———

ARMv6 以前のプロセッサでは、この規則にいくつかの例外があります。これらの例外は各命令の説明に記載されています。バンクレジスタの使用に制限がある場合、常に R8 ～ R14 のすべてに適用されます。例えば、FIQ モードが一切使用されず、R8 ～ R12 の物理バージョンが 1 つしか使用されないシステムでも、これらのレジスタには制限が適用されます。

———————

R8 ～ R12 には、それぞれ 2 つのバンク物理レジスタがあります。1 つは FIQ 以外のすべてのプロセッサモードで使用され、もう 1 つは FIQ モードで使用されます。参照するバージョンを明確に指定する必要がある場合、物理レジスタの最初のグループは R8_usr ～ R12_usr、2 番目のグループは R8_fiq ～ R12_fiq と呼ばれます。

レジスタ R8 ～ R12 には、アーキテクチャで特定の用途が割り当てられていません。しかし、レジスタ R8 ～ R14 だけを使用して処理できる単純な割り込みの場合、これらのレジスタの FIQ モードバージョンが別に存在することで、非常に高速な割り込み処理が可能となります。

R13 ～ R14 には、それぞれ 6 つのバンク物理レジスタがあります。1 つはユーザモードとシステムモードで、他の 5 つは 5 つの例外モードのそれぞれで使用されます。参照するバージョンを明確に指定する必要がある場合は、次の形式の名前を使用します。

```
R13_<mode>
R14_<mode>
```

ここで、`<mode>` は、`usr`、`svc`（スーパバイザモード）、`abt`、`und`、`irq`、`fiq` のうち、モードに対応するものです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

レジスタ R13 は一般にスタックポインタとして使用され、SP とも呼ばれます。ARMv6 で導入された SRS 命令は、R13 を特殊な方法で使用する唯一の ARM 命令です。Thumb 命令セットには他にも同様の命令があります。詳細については、第 A6 章「*Thumb 命令セット*」を参照して下さい。

各例外モードには、それぞれ専用の R13 のバンクバージョンがあります。R13 のバンクバージョンの適切な用途は、アーキテクチャバージョンによって異なります。

- ARMv6 以前のアーキテクチャバージョンでは、R13 の各バンクバージョンは、一般にその例外モード専用のスタックを指すように初期化されます。例外ハンドラは通常、使用する他のレジスタの値をエントリ時にこのスタックにストアします。これらの値は復帰時にレジスタにリロードされ、例外発生時に実行されていたプログラムの状態が破壊されることを防ぎます。

  これよりも例外処理スタックの数を減らす必要のあるシステムでは、別の例外モードとスタックに移行するときに、例外モード用の R13 のバンクバージョンを、一時的なストレージとして使用する小さなメモリ領域を指すように初期化することが可能です。たとえば、IRQ ハンドラが、スーパバイザモードスタックを使用して SPSR_irq、R0 〜 R3、R12、R14_irq をストアした後、IRQ を許可したスーパバイザモードで実行される必要があるとします。この場合、R13_irq が 4 ワードの一時記憶領域を指すように初期化し、次のコードシーケンスを使用してハンドラを開始します。

```
STMIA  R13, (R0-R3)    ; Put R0-R3 into temporary storage
MRS    R0, SPSR        ; Move banked SPSR and R12-R14 into
MOV    R1, R12         ; unbanked registers
MOV    R2, R13
MOV    R3, R14
MRS    R12, CPSR       ; Use read/modify/write sequence
BIC    R12, R12, #0x1F ; on CPSR to switch to Supervisor
ORR    R12, R12, #0x13 ; mode
MSR    CPSR_c, R12
STMFD  R13!, (R1,R3)   ; Push original {R12, R14_irq}, then
STR    R0, [R13,#-20]! ; SPSR_irq with a gap for R0-R3
LDMIA  R2, {R0-R3}     ; Reload R0-R3 from temporary storage
BIC    R12, R12, #0x80 ; Modify and write CPSR again to
MSR    CPSR_c, R12     ; re-enable IRQs
STMIB  R13, {R0-R3}    ; Store R0-R3 in the gap left on the
                       ; stack for them
```

- ARMv6 以降では、OS 設計者がシステム内で必要な例外処理スタックの数を決定し、各スタックを使用する例外を処理するのに適切なプロセッサモードを選択する必要があります。例えば、例外処理スタックの 1 つはアボートや優先度の高い割り込みに使用するために実メモリに固定する必要があり、もう 1 つは SWI、未定義命令、優先度の低い割り込みに使用するため仮想メモリを使用できるとします。この場合、プロセッサモードとしてそれぞれアボートモードとスーパバイザモードを使用するのが適切です。

  この場合、選択した各モード用の R13 のバンクバージョンは対応するスタックを指すように初期化され、R13 の他のバンクバージョンは通常は使用されません。例外ハンドラは、最初の SRS 命令で例外復帰情報を適切なスタックにストアしてから、（必要に応じて）CPS 命令で適切なモードに切替え、場合によっては割り込みを再許可します。その後で、他のレジスタをスタッ

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

クに保存できます。したがって上記の例で、直ちに割り込みを再許可したい未定義命令ハンドラは、次の 2 つの命令で開始します。

```
SRSFD    #svc_mode!
CPSIE   i, #svc_mode
```

ハンドラはこの後、R13_svc の指す仮想メモリスタックを使用して、すべてスーパバイザモードで実行されます。

レジスタ R14（*リンクレジスタ*または LR とも呼ばれます）には、アーキテクチャ上 2 つの特殊機能があります。

- 各モードで、モード独自のバージョンの R14 は、サブルーチン復帰アドレスの保持に使用されます。`BL` 命令または `BLX` 命令によりサブルーチンコールが実行されるとき、サブルーチン復帰アドレスは R14 に保存されます。サブルーチンから復帰するには、R14 をプログラムカウンタに再度コピーします。これは通常、次のいずれかの方法で実行されます。
  - `BX LR` 命令を実行します。

    ——— 注 ———

    `MOV PC,LR` 命令は、復帰する先のコードが現在の命令セットを使用している場合、`BX LR` 命令と同じ機能を果たします。ただし、Thumb コードから ARM サブルーチンが呼び出されている場合、または ARM コードから呼び出された Thumb サブルーチンが呼び出されている場合は、この命令で正しく復帰できません。この理由から、サブルーチンからの復帰に `MOV PC,LR` 命令を使用することは推奨されません。

    ———————
  - サブルーチンへのエントリ時に、次の形の命令で R14 をスタックにストアします。

    ```
    STMFD SP!,{<registers>,LR}
    ```

    復帰時には、対応する次の命令を使用します。

    ```
    LDMFD SP!,{<registers>,PC}
    ```
- 例外が発生すると、その例外モードに対応するバージョンの R14 に例外復帰アドレスが保存されます（一部の例外の場合、オフセットとして小さな定数が加えられます）。例外からの復帰はサブルーチンからの復帰と同様に行いますが、例外発生時に実行されていたプログラムの状態を完全に復元するため、少し異なる命令が使用されます。詳細については、P. A2-16*「例外」*を参照して下さい。

その他の場合は常に、R14 を汎用レジスタとして扱うことができます。

——— 注 ———

ネストした例外の発生が可能な場合、2 つの特殊用途が競合することがあります。たとえば、プログラムがユーザモードで実行されている間に IRQ 割り込みが発生した場合、それだけではどのユーザモードレジスタも破壊されません。しかし、IRQ モードで動作する割り込みハンドラが IRQ 割り込みを再許可した後で、ネストしたIRQ割り込みが発生すると、そのとき外側の割り込みハンドラがR14_irq に保持している値は、ネストした割り込みの復帰アドレスによって上書きされます。

システムプログラマは、これらの相互作用に注意する必要があります。これに対処する一般的な方法は、ネストした例外が発生する可能性がある場合、該当するバージョンの R14 が重要な値を保持しないようにすることです。この条件が簡単に実現不能な場合、例外ハンドラへのエントリ時に割り込み

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

を再度許可する、または他の形でネストした例外の発生を許可する前に、別のプロセッサモードに変更するのが最適です（ARMv4 以降の場合、一般にこの目的にはシステムモードが最適です）。

A2.4.3 レジスタ 15 とプログラムカウンタ

レジスタ 15（R15）を他の汎用レジスタの代わりに使用すると、多くの場合に特殊な動作が引き起こされます。このような動作は命令によって異なるため、詳細については各命令の説明を参照して下さい。

また、R15 の使用に関しては、各命令に特有の多くの制限があります。これらについても、各命令の説明の中で説明します。通常、R15 がこれらの制限に従わない方法で使用された場合、命令の動作は予測不能です。

各命令の説明の中に、R15 が使用される場合の特別な効果や、R15 の使用に関する制限が記載されていない場合、以下のセクションで説明されているように、R15 は*プログラムカウンタ*（PC）の読み出しと書き込みに使用されます。

- *プログラムカウンタの読み出し*
- *プログラムカウンタへの書き込み*：P. A2-10

プログラムカウンタの読み出し

命令によって PC を読み出す場合、読み出される値は、命令がどの命令セットのものかによって異なります。

- ARM 命令の場合、読み出される値は、（その命令 + 8 バイト）のアドレスです。ARM 命令は常にワード境界でアラインしているので、この値のビット [1:0] は常に 0 です。
- Thumb 命令の場合、読み出される値は、（その命令 + 4）バイトのアドレスです。Thumb 命令は常にハーフワード境界でアラインしているので、この値のビット [0] は常に 0 です。

このような PC の読み出しは、プログラム内のポジション・インディペンデントの分岐など、近くの命令やデータに対する迅速でポジション・インディペンデントのアドレス指定に主に使用されます。

ARM 命令の STR または STM 命令で R15 をストアする場合は、上記の規則の例外となります。これらの命令は、R15 を読み出す他の命令と同様に、（その命令 + 8 バイト）、または（その命令 + 12 バイト）のアドレスをストアできます。オフセットとして 8 と 12 のどちらが使用されるかは実装定義です。同じ実装では、R15 をストアするすべての ARM 命令の STR 命令と STM 命令について同じオフセットを使用する必要があります。場合によって、8 と 12 のオフセットを使い分けることはできません。

この例外があるため、通常は、R15 をストアする STR、STM 命令は使用しないことをお勧めします。それが難しい場合は、プログラム内で適切な命令シーケンスを使用し、実装でどちらのオフセットが使用されているかを確認します。例えば、R0 がメモリ内の使用可能なワードを指す場合、以下の命令は、実装のオフセットを R0 に置きます。

```
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
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

――― 注 ―――

R15 の読み出しに関するこの規則は、命令による読み出しにのみ適用されます。特に、命令フェッチ中にハードウェアアドレスバスに置かれる値については、必ずしも当てはまりません。ハードウェアインタフェースに関する他の詳細と同様に、そのような値は実装定義です。

―――――――――

プログラムカウンタへの書き込み

命令によって PC への書き込みを行う場合、PC に書き込まれる値は通常は命令アドレスとして扱われ、そのアドレスへ分岐が発生します。

ARM 命令はワード境界でアラインしている必要があるため、ARM 命令が PC に書き込む値は通常、ビット [1:0] == 0b00 です。同様に、Thumb 命令はハーフワード境界でアラインしている必要があるため、Thumb 命令が PC に書き込む値は通常、ビット [0] == 0 です。

正確な規則は、現在の命令セットの状態とアーキテクチャのバージョンによって異なります。

- ARMv6 以降のすべてのバリエーションと ARMv4 以降のの T バリエーションでは、命令の説明に特に記載されていない限り、Thumb ステートで R15 に書き込まれた値のビット [0] は無視されます。PC のビット [0] が実装されている場合（Jazelle 拡張機能の実装の有無と実装の方法に依存）、どのような値が書き込まれたかにかかわらず、R15 のビット [0] には 0 が書き込まれる必要があります。
- ARMv6 以降では、ARM ステートで R15 に書き込まれる値のビット [1:0] は、命令の説明に特記がない限り無視されます。PC のビット [1] には、R15 のビット [1] に書き込まれる値にかかわらず、必ず 0 を書き込む必要があります。PC のビット [0] が実装されている場合（Jazelle 拡張機能の実装方法に依存）、必ず 0 を書き込む必要があります。
- ARMv4 と ARMv5 のすべてのバリエーションで、ARM ステートで R15 に書き込まれる値のビット [1:0] は常に 0b00 です。この値が 0b00 でない場合、結果は予測不能です。

いくつかの命令は、R15 に書き込まれた値を独自の規則で解釈します。例えば、BX など、ARM ステートと Thumb ステートの間で遷移を行う命令は、値のビット [0] を使用して、分岐先のコードを ARM ステートと Thumb ステートのどちらで実行するか判断します。このタイプの特別な規則は、各命令のページに説明されており、このセクションに記載されている一般的な規則より優先されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.5 プログラムステータスレジスタ

カレントプログラムステータスレジスタ（CPSR）は、すべてのプロセッサモードでアクセス可能です。CPSR には、条件コードフラグ、割り込みディセーブルビット、カレントプロセッサモード、その他のステータスおよび制御情報が含まれています。各例外モードには、*保存プログラムステータスレジスタ*（SPSR）もあります。これは、所定の例外が発生した場合に、CPSR の値を保存するために使用されます。

――― 注 ―――

ユーザモードとシステムモードは例外モードでないため、対応する SPSR は存在しません。ユーザモードまたはシステムモードで実行された場合、SPSR に読み出しまたは書き込みを行う命令はすべて、動作は予測不能です。

―――――――

CPSR と SPSR の形式は以下のとおりです。

31	30	29	28	27	26 25	24	23 20	19 16	15 10	9	8	7	6	5	4 0
N	Z	C	V	Q	Res	J	予約	GE[3:0]	予約	E	A	I	F	T	M[4:0]

A2.5.1 PSR ビットのタイプ

PSR ビットは、更新の方法によって 4 つのカテゴリに分けられます。

予約ビット
将来の拡張用に予約されています。実装は、これらのビットから 0 が読み出され、書き込みは無視されるように設計する必要があります。アーキテクチャの将来の拡張と最大限の互換性を持たせるため、予約ビットには同じビットから読み出した値を書き込む必要があります。

ユーザ書き込み可能ビット
どのモードからでも書き込むことができます。N、Z、C、V、Q、GE[3:0]、E はユーザ書き込み可能ビットです。

特権ビット
どの特権モードからでも書き込むことができます。ユーザモードでは、特権ビットへの書き込みは無視されます。A、I、F、M[4:0] は特権ビットです。

実行ステートビット
どの特権モードからでも書き込むことができます。ユーザモードでは、実行ステートビットへの書き込みは無視されます。実行ステートビットは J ビットと T ビットで、ARM ステートでは常に 0 です。

CPSR の実行ステートビットに書き込む特権 `MSR` 命令は、実行ステートビットの変更を防ぐため、0 を書き込む必要があります。実行ステートビットのいずれか、または両方に 1 を書き込んだ場合、動作は予測不能です。この制限は、CPSR の実行ステートビットだけに適用され、SPSR の実行ステートビットには適用されません。

A2.5.2 条件コードフラグ

N、Z、C、V（ネガティブ、ゼロ、キャリー、オーバフロー）ビットは、集合的に*条件コードフラグ*と呼ばれ、多くの場合*フラグ*と呼ばれます。ほとんどの命令では CPSR の条件コードフラグに対してテストが行われ、その結果によって命令が実行されるかどうかが判断されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

条件コードフラグは通常、以下の動作によって変更されます。

- 比較命令（CMN、CMP、TEQ、TST）の実行
- 命令のデスティネーションレジスタが R15 でない一部の算術、論理、移動命令の実行。このような命令のほとんどには、フラグ保存バージョンとフラグセットバージョンがあり、命令のニーモニックに S 修飾子を追加すると後者が選択されます。一部の命令には、フラグ保存バージョンしか存在しません。これについては、各命令の説明を参照して下さい。

どちらの場合も、新しい条件コードフラグ（命令実行後）は通常、次のような意味です。

N 命令の結果のビット 31 を示しています。この結果が、2 の補数の符号付き整数と見なされる場合、結果が負であれば N = 1、結果が正または 0 であれば N = 0 です。

Z 命令の結果が 0 の場合は 1 にセットされ（一般に、比較の結果が *等しい* であることを意味します）、それ以外の場合は 0 にセットされます。

C 以下の 4 つのいずれかの方法でセットされます。

- 比較命令の CMN を含め、加算の場合に、加算によってキャリーが生じた場合（符号なしのオーバフロー）C が 1 にセットされ、それ以外は 0 にセットされます。
- 比較命令の CMP を含め、減算の場合に、減算によって桁下がりが生じた場合（符号なしのアンダーフロー）C が 0 にセットされ、それ以外は 1 にセットされます。
- シフト演算を含む加算 / 減算の場合、シフタによって値の外にシフトされた最後のビットが C にセットされます。
- 他の加算 / 減算以外の命令では通常、C は変更されません（特殊なケースについては各命令の説明に記載されています）。

V 以下の 2 つのいずれかの方法でセットされます。

- 加算または減算で、オペランドと結果を 2 の補数の符号付き整数と見なした場合に符号付きオーバフローが発生すると、V が 1 にセットされます。
- その他の加算 / 減算については通常、V は変更されません（特殊なケースについては各命令の説明に記載されています）。

フラグは、以下の場合にも変更されます。

- MSR 命令の実行。命令の機能の一部として、CPSR または SPSR に新しい値が書き込まれます。
- R15 をデスティネーションレジスタとする MRC 命令の実行。このような命令の目的は、コプロセッサが生成した条件コードフラグの値を ARM プロセッサに移すことです。
- LDM 命令のうち、一部のバリエーションの実行。これらのバリエーションは、SPSR を CPSR にコピーします。主に、例外からの復帰に使用されます。
- 特権モードにおける RFE 命令の実行。新しい値をメモリから CPSR にロードします。
- 算術命令と論理命令のうち、R15 をデスティネーションレジスタとし、フラグをセットするバリエーションの実行。これらも SPSR を CPSR にコピーし、例外からの復帰に使用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.5.3 Q フラグ

ARMv5 以降の E バリエーションでは、CPSR のビット [27] は Q フラグと呼ばれ、一部の DSP 指向命令でオーバフローや飽和が発生したかどうかを示します。同様に、各 SPSR のビット [27] も Q フラグで、例外が発生した場合、CPSR の Q フラグの保存と復元に使用されます。詳細については、P. A2-69 *「飽和整数算術演算」*を参照して下さい。

ARMv5 以前のバージョンのアーキテクチャと、ARMv5 の E バリエーション以外では、CPSR および SPSR のビット [27] は、予約ビットとして扱う必要があります。詳細については、P. A2-11 *「PSR ビットのタイプ」*を参照して下さい。

A2.5.4 GE[3:0] ビット

ARMv6 の場合、SIMD 命令は、結果の各バイトまたはハーフワードの ≧ (GE) フラグとして、ビット [19:16] を使用します。GE フラグは、以後の SEL 命令の制御に使用できます。詳細については、P. A4-128 *「SEL」*を参照して下さい。

ハーフワードを操作する命令では、次の動作が実行されます。

- 最上位ハーフワードの計算結果に基づいて、GE[3:2] を共にセットまたはクリアします。
- 最下位ハーフワードの計算結果に基づいて、GE[1:0] を共にセットまたはクリアします。

バイトを操作する命令では、次の動作が実行されます。

- 最上位バイトの計算結果に基づいて、GE[3] をセットまたはクリアします。
- 2 番目のバイトの計算結果に基づいて、GE[2] をセットまたはクリアします。
- 3 番目のバイトの計算結果に基づいて、GE[1] をセットまたはクリアします。
- 最下位バイトの計算結果に基づいて、GE[0] をセットまたはクリアします。

対応する計算の結果が以下の場合、対応するビットがセットされます。それ以外の場合、ビットはクリアされます。

- 符号なしバイト加算で、結果が 2^8 以上の場合
- 符号なしハーフワード加算で、結果が 2^{16} 以上の場合
- 符号なし減算で、結果がゼロ以上の場合
- 符号付き算術演算で、結果がゼロ以上の場合

ARMv6 以前のバージョンのアーキテクチャでは、CPSR および SPSR のビット [19:16] は予約ビットとして扱う必要があります。詳細については、P. A2-11 *「PSR ビットのタイプ」*を参照して下さい。

A2.5.5 E ビット

ARMv6 以降は、ビット [9] はデータ処理のロード / ストアのエンディアン形式を制御します。詳細については、P. A2-36 *「CPSR の E ビットを変更する命令」*を参照して下さい。このビットは、命令フェッチ時には無視されます。

ARMv6 以前のバージョンのアーキテクチャバージョンでは、CPSR および SPSR のビット [9] は予約ビットとして扱う必要があります。詳細については、P. A2-11 *「PSR ビットのタイプ」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.5.6 割り込みディセーブルビット

A、I、F は割り込みディセーブルビットです。

A ビット セットすると、不正確なデータアボートが禁止されます。このビットは、ARMv6 以降にのみ存在します。ARMv6 以前バージョンのアーキテクチャでは、CPSR および SPSR のビット [8] は予約ビットとして扱う必要があります。詳細については、P. A2-11 「*PSR ビットのタイプ*」を参照して下さい。

I ビット セットすると、IRQ 割り込みが禁止されます。

F ビット セットすると、FIQ 割り込みが禁止されます。

A2.5.7 モードビット

M[4:0] は、モードビットです。モードビットは、プロセッサが動作するモードを決定します。モードビットの解釈については、表 A2-2 を参照して下さい。

表 A2-2 モードビット

M[4:0]	モード	アクセス可能なレジスタ
0b10000	ユーザ	PC、R14 ～ R0、CPSR
0b10001	FIQ	PC、R14_fiq ～ R8_fiq、R7 ～ R0、CPSR、SPSR_fiq
0b10010	IRQ	PC、R14_irq、R13_irq、R12 ～ R0、CPSR、SPSR_irq
0b10011	スーパバイザ	PC、R14_svc、R13_svc、R12 ～ R0、CPSR、SPSR_svc
0b10111	アボート	PC、R14_abt、R13_abt、R12 ～ R0、CPSR、SPSR_abt
0b11011	未定義	PC、R14_und、R13_und、R12 ～ R0、CPSR、SPSR_und
0b11111	システム	PC、R14 ～ R0、CPSR（ARMv4 以降）

モードビットのすべての組み合わせが、有効なプロセッサモードを定義しているわけではありません。明記されている組み合わせのみが使用可能です。モードビット M[4:0] に他の値がプログラムされた場合、結果は予測不能です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.5.8 T ビットと J ビット

T ビットと J ビットは、表 A2-3 に示すように、現在の命令セットの選択に使用します。

表 A2-3 T ビットと J ビット

J	T	命令セット
0	0	ARM
0	1	Thumb
1	0	Jazelle
1	1	予約

T ビットは、ARMv4 の T バリエーションと、ARMv5 以降のすべてのバリエーションに存在します。ARMv4 の T 以外のバリエーションでは、T ビットは予約ビットとして扱う必要があります。詳細については、P. A2-11 *「PSR ビットのタイプ」*を参照して下さい。

Thumb 命令セットは、ARMv4 と ARMv5 の T バリエーション、ARMv6 以降の全バリエーションに実装されています。ARM ステートと Thumb ステートの実行を切り替える命令は、これらのアーキテクチャの実装で自由に使用できます。

ARMv5 の T 以外のバリエーションには、Thumb 命令セットが実装されていません。これらのアーキテクチャのバリエーションで、T == 1 にセットし、Thumb 命令セットを選択すると、次に実行される命令が未定義命令例外（P. A2-19 *「未定義命令例外」*参照）を発生します。ARM ステートと Thumb ステートの実行を切り替える命令は、ARMv5 の T 以外のアーキテクチャバリエーションを採用している実装でも使用できますが、プログラムが ARM ステートを維持する場合にのみ正常に動作します。プログラムが Thumb 状態への切り替えを試みると、切替えの後で最初に実行される命令が未定義命令例外を発生します。この場合、未定義命令例外へのエントリ後にプロセッサは ARM 状態に戻されます。例外ハンドラは、SPSR_und の T ビットがセットされていることで、例外の原因を検出できます。

J ビットは、ARMv5TEJ と、ARMv6 以降の全バリエーションに存在します。ARMv4 と、ARMv5TEJ 以外の ARMv5 のバリエーションでは、J ビットは予約ビットとして扱われる必要があります。詳細については、P. A2-11 *「PSR ビットのタイプ」*を参照して下さい。

Jazelle オペコード実行のハードウェアアクセラレーションは、ARMv5TEJ と、ARMv6 以降で実装可能です。これらのアーキテクチャバリエーションでは、ハードウェアアクセラレータが存在し、許可されている場合、ARM ステートから Jazelle ステートへの切り替えに `BXJ` 命令が使用されます。ハードウェアアクセラレータが存在しない、または禁止されている場合、`BXJ` 命令は `BX` 命令として動作し、J ビットはクリアのままとなります。詳細については、P. A2-53 *「Jazelle 拡張」*を参照して下さい。

A2.5.9 その他のビット

プログラムステータスレジスタのその他のビットは、将来の拡張用に予約されています。通常、プログラマは、これらの予約ビットが変更されないように注意してコードを作成する必要があります。これらのビットを変更した場合、将来のアーキテクチャバージョンで、そのコードが予期しない効果を引き起こす可能性があります。詳細については、P. A2-11 *「PSR ビットのタイプ」*、および P. A4-77 に記載された `MSR` 命令の使用上の注意を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.6 例外

例外は、内部ソースと外部ソースによって発生します。例外により、プロセッサは、外部で生成された割り込みや、未定義命令の実行などのイベントに対して処理を実行します。例外ルーチンの完了後に元のプログラムが再開できるようにするため、通常は例外を処理する直前のプロセッサステートが保存されます。同時に複数の例外が発生することもあります。

ARM アーキテクチャは、7 タイプの例外に対応します。表 A2-4 は、例外のタイプと、各タイプの処理に使用されるプロセッサモードを示しています。例外が発生すると、例外のタイプに従い、メモリの固定アドレスから実行が開始されます。このような固定アドレスは、*例外ベクタ*と呼ばれます。

——— 注 ———

アドレス `0x00000014` の通常ベクタと、アドレス `0xFFFF0014` の上位ベクタは、将来の拡張用に予約されています。

表 A2-4 例外処理モード

例外タイプ	モード	VE[a]	通常アドレス	上位ベクタアドレス
リセット	スーパバイザ		0x00000000	0xFFFF0000
未定義命令	未定義		0x00000004	0xFFFF0004
ソフトウェア割り込み（SWI）	スーパバイザ		0x00000008	0xFFFF0008
プリフェッチアボート（命令フェッチ中に発生するメモリアボート）	アボート		0x0000000C	0xFFFF000C
データアボート（データアクセス中に発生するメモリアボート）	アボート		0x00000010	0xFFFF0010
IRQ（割り込み）	IRQ	0	0x00000018	0xFFFF0018
		1	実装定義	
FIQ（高速割り込み）	FIQ	0	0x0000001C	0xFFFF001C
		1	実装定義	

a. VE = ベクタ割り込み許可（CP15 制御）。実装されていない場合は、0 が読み出されます（RAZ）。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外が発生すると、その例外モードに対応するバンクの R14 および SPSR が、状態の保存に使用されます。以下を参照して下さい。

```
R14_<exception_mode> = return link
SPSR_<exception_mode> = CPSR
CPSR[4:0] = exception mode number
CPSR[5] = 0                             /* Execute in ARM state */
if <exception_mode> == Reset or FIQ then
    CPSR[6] = 1                         /* Disable fast interrupts */
/* else CPSR[6] is unchanged */
CPSR[7] = 1                             /* Disable normal interrupts */
if <exception_mode> != UNDEF or SWI then
    CPSR[8] = 1                         /* Disable imprecise aborts (v6 only) */
/* else CPSR[8] is unchanged */
CPSR[9] = CP15_reg1_EEbit               /* Endianness on exception entry */
PC = exception vector address
```

例外処理の復帰時には、SPSR が CPSR に移動され、R14 が PC に移動されます。これは、以下の 2 つの方法で、アトミックに行われます。

- S ビットがセットされている PC をデスティネーションとするデータ処理命令
- P. A4-41「*LDM (3)*」に記載されている、CPSR 復元付き複数ロード命令

また、ARMv6 では、RFE 命令（P. A4-114「*RFE*」参照）を使用し、メモリから CPSR と PC をロードできます。これによって、例外から、前にメモリに保存されていた CPSR と PC にアトミックに復帰します。

これらの機構は全体として、例外からの復帰を実行する機構の全てを定義します。

以降のセクションでは、例外が発生したときに自動的に起きる処理、および各例外からの復帰時に推奨されるデータ処理命令について説明します。この命令は常に、PC をデスティネーションとする MOVS または SUBS 命令です。

注

推奨されるデータ処理命令が SUBS で、CPSR 復元付き複数ロード命令を使用して例外ハンドラから復帰する場合でも、その減算の実行は必要です。通常、減算は例外ハンドラの開始時、復帰リンクがメモリにストアされる前に行われます。

例えば、復帰リンクをスタックにストアする割り込みハンドラは、エントリポイントで次の形の命令を使用できます。

```
    SUB    R14, R14, #4
    STMFD  SP!, {<other_registers>, R14}
```

復帰には次の命令を使用します。

```
    LDMFD  SP!, {<other_registers>, PC}^
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.6.1 ARMv6 における例外モデルの拡張

ARMv6 以降では、例外モデルが以下のように拡張されています。

- 不正確なデータアボート機構により、いくつかのタイプのデータアボートを非同期に扱うことが可能です。結果として生じる例外は、割り込みのように動作しますが、アボートモードとそのバンクレジスタが使用される点が異なります。この機構には、不正確なデータアボートが別のアボートの処理中に発生しないよう、PSR にマスクビット（A ビット）があります。この機構については、P. A2-23 *「不正確なデータアボート」*を参照して下さい。
- システム制御コプロセッサの VE ビットにより制御されるベクタ割り込みのサポート（P. A2-26 *「ベクタ割り込みのサポート」*参照）。以前のバージョンのアーキテクチャがこの機構に対応しているかどうかは、実装定義です。
- システム制御コプロセッサの FI ビットにより制御される低割込みレイテンシ構成のサポート（P. A2-27 *「低割込みレイテンシ構成」*参照）。以前のバージョンのアーキテクチャがこの機構に対応しているかどうかは、実装定義です。
- ひとつの共通モードで、複数の異なる例外のネストスタック処理を改善するための、3 つの新しい命令（CPS、SRS、RFE）。CPS 命令は、1 つのモード内で、または特権モードから他のモードへの遷移中に、割り込みと不正確なアボートマスクを効率的に許可または禁止するためにも使用できます。説明については、P. A2-28 *「例外処理を改善する新しい命令」*を参照して下さい。

A2.6.2 リセット

リセット入力がプロセッサ上でアサートされると、ARM プロセッサは、ただちに現在の命令の実行を停止します。リセットがアサート解除されると、以下のアクションが実行されます。

```
R14_svc    = UNPREDICTABLE value
SPSR_svc   = UNPREDICTABLE value
CPSR[4:0] = 0b10011              /* Enter Supervisor mode */
CPSR[5]    = 0                   /* Execute in ARM state */
CPSR[6]    = 1                   /* Disable fast interrupts */
CPSR[7]    = 1                   /* Disable normal interrupts */
CPSR[8]    = 1                   /* Disable Imprecise Aborts (v6 only) */
CPSR[9]    = CP15_reg1_EEbit     /* Endianness on exception entry */
if high vectors configured then
    PC     = 0xFFFF0000
else
    PC     = 0x00000000
```

リセットの後、ARM プロセッサは、割り込みを禁止したスーパバイザモードで、アドレス 0x00000000 または 0xFFFF0000 から実行を開始します。

―― 注 ――

リセットから復帰するための、アーキテクチャで定義された方法はありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.6.3 未定義命令例外

コプロセッサ命令を実行するときに、ARM プロセッサは、いずれかの外部コプロセッサから命令の実行可能の応答が返されるのを待ちます。応答するコプロセッサがなければ、未定義命令例外が発生します。

未定義の命令の実行を試みると、未定義命令例外が発生します（P. A3-32「*命令セットの拡張*」参照）。

未定義命令例外は、物理的なコプロセッサ（ハードウェア）を持たないシステムにおけるコプロセッサのソフトウェアエミュレーション、またはソフトウェアエミュレーションによる汎用命令セットの拡張に使用できます。

未定義命令例外が発生すると、次の動作が実行されます。

```
R14_und    = address of next instruction after the Undefined instruction
SPSR_und   = CPSR
CPSR[4:0] = 0b11011              /* Enter Undefined Instruction mode */
CPSR[5]    = 0                   /* Execute in ARM state */
                                 /* CPSR[6] is unchanged */
CPSR[7]    = 1                   /* Disable normal interrupts */
                                 /* CPSR[8] is unchanged */
CPSR[9]    = CP15_reg1_EEbit     /* Endianness on exception entry */
if high vectors configured then
    PC     = 0xFFFF0004
else
    PC     = 0x00000004
```

未定義命令のエミュレートから復帰するには、次の命令を使用します。

```
    MOVS PC,R14
```

この命令は、PC（R14_und から）と CPSR（SPSR_und から）を復元し、未定義命令の次の命令に復帰します。

一部のコプロセッサ設計では、1 つのコプロセッサ命令によって引き起こされた内部的な例外条件が、後続のコプロセッサ命令への応答を拒否するという形で、*不正確に*通知されることがあります。この場合、未定義命令ハンドラは、例外条件をクリアするのに必要な動作を実行した後、2 番目のコプロセッサ命令に戻ります。このためには、次の命令を使用します。

```
    SUBS PC,R14,#4
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.6.4 ソフトウェア割り込み例外

ソフトウェア割り込み例外（SWI）は、スーパバイザモードを開始し、特定のスーパバイザ（オペレーティングシステム）機能を要求します。SWI が実行されると、以下の動作が起こります。

```
R14_svc   = address of next instruction after the SWI instruction
SPSR_svc  = CPSR
CPSR[4:0] = 0b10011             /* Enter Supervisor mode */
CPSR[5]   = 0                   /* Execute in ARM state */
                                /* CPSR[6] is unchanged */
CPSR[7]   = 1                   /* Disable normal interrupts */
                                /* CPSR[8] is unchanged */
CPSR[9]   = CP15_reg1_EEbit     /* Endianness on exception entry */
if high vectors configured then
    PC    = 0xFFFF0008
else
    PC    = 0x00000008
```

SWI 処理を実行した後で、次の命令を使用して PC（R14_svc から）と CPSR（SPSR_svc から）を復元し、SWI の次の命令に復帰します。

```
    MOVS PC,R14
```

A2.6.5 プリフェッチアボート（命令フェッチ中に発生するメモリアボート）

メモリアボートは、メモリシステムによって通知されます。命令フェッチに対してアボートを起動することにより、フェッチされた命令は無効としてマークされます。プリフェッチアボート例外は、プロセッサが無効な命令の実行を試みたときに生成されます。その命令が実行されない場合（パイプライン内で分岐が発生した場合など）は、プリフェッチアボートが発生しません。

ARMv5 以降では、BKPT 命令を実行した結果としてプリフェッチ例外が発生することもあります。詳細については、P. A4-14「*BKPT*」（ARM 命令）と P. A7-24「*BKPT*」（Thumb 命令）を参照して下さい。

アボートされた命令の実行を試みると、次の動作が引き起こされます。

```
R14_abt   = address of the aborted instruction + 4
SPSR_abt  = CPSR
CPSR[4:0] = 0b10111             /* Enter Abort mode */
CPSR[5]   = 0                   /* Execute in ARM state */
                                /* CPSR[6] is unchanged */
CPSR[7]   = 1                   /* Disable normal interrupts */
CPSR[8]   = 1                   /* Disable Imprecise Data Aborts (v6 only) */
CPSR[9]   = CP15_reg1_EEbit     /* Endianness on exception entry */
if high vectors configured then
    PC    = 0xFFFF000C
else
    PC    = 0x0000000C
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アボートが発生した理由を修正した後で、次の命令を使用して例外から復帰します。

```
SUBS PC,R14,#4
```

これにより、PC（R14_abt から）と CPSR（SPSR_abt から）の両方が復元され、アボートした命令への復帰が行われます。

A2.6.6 データアボート（データアクセス中に発生するメモリアボート）

メモリアボートは、メモリシステムによって通知されます。データアクセス（ロードまたはストア）に対してアボートが起動すると、そのデータは無効にマークされます。データアボート例外は、後に続く命令や例外によって CPU の状態が変更される前に発生します。次の動作が実行されます。

```
R14_abt   = address of the aborted instruction + 8
SPSR_abt  = CPSR
CPSR[4:0] = 0b10111             /* Enter Abort mode */
CPSR[5]   = 0                   /* Execute in ARM state */
                                /* CPSR[6] is unchanged */
CPSR[7]   = 1                   /* Disable normal interrupts */
CPSR[8]   = 1                   /* Disable Imprecise Data Aborts (v6 only) */
CPSR[9]   = CP15_reg1_EEbit     /* Endianness on exception entry */
if high vectors configured then
    PC    = 0xFFFF0010
else
    PC    = 0x00000010
```

アボートが発生した原因を修正した後で、次の命令を使用して例外から復帰します。

```
SUBS PC,R14,#8
```

これにより、PC（R14_abt から）と CPSR（SPSR_abt から）の両方が復元され、アボートが発生した命令が再実行されます。

アボートが発生した命令を再実行する必要がない場合、次の命令を使用します。

```
SUBS PC,R14,#4
```

データアボートが発生した命令の影響

データメモリにアクセスする命令は、1 つまたは複数の値をストアすることで、メモリを変更する場合があります。そのような命令でデータアボートが発生すると、命令がストアする各メモリロケーションの値は次のようになります。

- メモリシステムがそのメモリロケーションへの書き込みアクセスを許可しない場合、メモリの値は変更されません。
- それ以外の場合、メモリの値は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

データメモリにアクセスする命令は、以下の方法でレジスタを変更する場合があります。

- PC を含む、1 つ以上の汎用レジスタへの値のロードで
- アドレス計算に使用されたベースレジスタに変更された値が書き込まれる*ベースレジスタライトバック*を指定することで、どのような命令でも、ベースレジスタライトバックが指定され、ベースレジスタが PC の場合は、結果は予測不能です。したがって、事実上はこの方法で変更できるのは PC 以外の汎用レジスタのみです。
- コプロセッサレジスタへの値のロードで
- CPSR の変更で

データアボートが発生した場合、これらのレジスタに残る値は、以下の規則に従って決定されます。

1. データアボートハンドラへのエントリ時の PC の値は `0x00000010` または `0xFFFF0010` で、R14_abt の値は、アボートが発生した命令のアドレスによって決定されます。命令によって指定された PC ロードの結果には、どちらも影響を受けません。

2. ベースレジスタライトバックが指定されていない場合、ベースレジスタの値は変更されません。これは、アボートの発生したアクセスより前に、命令が自分のベースレジスタをロードしベースレジスタをロードするメモリアクセスを行った場合でも、当てはまります。

   例えば、次のような命令を考えます。

   ```
   LDMIA R0,{R0,R1,R2}
   ```

   この場合、新しい R0 値、新しい R1 値、最後に新しい R2 値がロードされます。いずれかのアクセスにおいてデータアボートが発生した場合、命令のベースレジスタ R0 の値は変更されません。これは、アボートが発生したのが R0 のロードではなく、R1 または R2 のロードの場合も同じです。

3. ベースレジスタライトバックが指定されている場合、ベースレジスタに残る値は、実装の*アボートモデル*によって決定されます。詳細については、P. A2-23「*アボートモデル*」を参照して下さい。

4. 命令が汎用レジスタを 1 つしかロードしない場合、そのレジスタの値は変更されません。

5. 命令が複数の汎用レジスタをロードする場合、PC とベースレジスタを除く命令のデスティネーションレジスタに残る値は予測不能です。

6. 命令がコプロセッサレジスタをロードする場合、コプロセッサの命令セットの説明に明記されていない限り、デスティネーションのコプロセッサレジスタに残る値は予測不能です。

7. 例外へのエントリ時に更新済みと定義されていない CPSR ビットは、現行の値を維持します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アボートモデル

各 ARM 実装で使用されるアボートモデルは実装定義で、以下のいずれかです。

ベース復元アボートモデル

ベースレジスタライトバックを指定する命令で正確なデータアボートが発生した場合、ベースレジスタの値は変更されません。ARMv6 以降では、このアボートモデルのみが許容されます。

ベース更新アボートモデル

ベースレジスタライトバックを指定する命令で正確なデータアボートが発生した場合も、ベースレジスタライトバックが発生します。ARMv6 以降ではこのモデルは使用できません。

どちらの場合も、すべての命令について同じアボートモデルが適用されます。一部の命令にベース復元アボートモデルを使用し、一部の命令にベース更新アボートモデルを使用する実装は存在しません。

A2.6.7 不正確なデータアボート

不正確なデータアボートとは、ライトバッファに保持されている書き込みに対する外部エラーなどによって発生するアボートで、原因となった命令の実行とは非同期であり、実際には、メモリアクセスを引き起こした命令の終了から何サイクルも後で発生することがあります。このため、不正確なデータアボートは、正確なデータアボートが原因でプロセッサがアボートモードにあるとき、またはアボートモードで割り込みを処理しているときにも発生する可能性があります。

このような場合に、アボートモード状態（R14 および SPSR_abt）が失われ、プロセッサが回復不能な状態に陥るのを防ぐには、アボートモードに安全なエントリが可能になるまで、システムが不正確なデータアボートを保留する必要があります。

ARMv6 以降は、不正確なアボートを許可できないことを示すマスク（CPSR[8]）が CPSR に追加されています。このビットは、A ビットと呼ばれます。不正確なデータアボートがマスクされていない場合、不正確なデータアボートはデータアボートを引き起こします。不正確なデータアボートがマスクされている場合、マスクがクリアされてアボートが開始するまで、実装が不正確なデータアボートを保留する必要があります。複数の不正確なアボートを保留できるかどうかは、実装定義です。

A ビットは、アボートモード、FIQ モード、IRQ モードへのエントリ時、およびリセット時に自動的にセットされます。

A ビットは、特権モードでのみ変更できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.6.8 割り込み要求（IRQ）例外

IRQ 例外は、プロセッサの IRQ 入力をアサートすることにより、外部で生成されます。IRQ 例外の優先度は FIQ より低く（P. A2-25 表 A2-5 参照）、FIQ シーケンスへのエントリ時にマスクされます。

CPSR の I ビットがセットされている場合、割り込みは禁止されます。I ビットがクリアの場合、ARM は命令境界で IRQ があるかどうかを確認します。

——— 注 ———

I ビットは、特権モードでのみ変更できます。

IRQ が検出されると、次の動作が実行されます。

```
R14_irq   = address of next instruction to be executed + 4
SPSR_irq  = CPSR
CPSR[4:0] = 0b10010              /* Enter IRQ mode */
CPSR[5]   = 0                    /* Execute in ARM state */
                                 /* CPSR[6] is unchanged */
CPSR[7]   = 1                    /* Disable normal interrupts */
CPSR[8]   = 1                    /* Disable Imprecise Data Aborts (v6 only) */
CPSR[9]   = CP15_reg1_EEbit      /* Endianness on exception entry */
if VE==0 then
  if high vectors configured then
      PC    = 0xFFFF0018
  else
      PC    = 0x00000018
else
  PC = IMPLEMENTATION DEFINED    /* see P. A2-26 */
```

割り込み処理の完了後、次の命令で復帰します。

```
    SUBS PC,R14,#4
```

これにより、PC（R14_irq から）と CPSR（SPSR_irq から）の両方が復元され、割り込まれたコードの実行が再開されます。

A2.6.9 高速割り込み要求（FIQ）例外

FIQ 例外は、プロセッサの FIQ 入力をアサートすることにより、外部で生成されます。FIQ は、データ転送またはチャネル処理を目的とし、レジスタを保存しなくても済むよう十分な専用レジスタを持つため、コンテキストスイッチに必要なオーバヘッドを最小限に抑えられます。

CPSR の F ビットがセットされている場合、高速割り込みは禁止されます。F ビットがクリアの場合、ARM は命令境界で FIQ があるかどうかを確認します。

——— 注 ———

F ビットは、特権モードでのみ変更できます。

FIQ が検出されると、次の動作が実行されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

```
R14_fiq   = address of next instruction to be executed + 4
SPSR_fiq  = CPSR
CPSR[4:0] = 0b10001              /* Enter FIQ mode */
CPSR[5]   = 0                    /* Execute in ARM state */
CPSR[6]   = 1                    /* Disable fast interrupts */
CPSR[7]   = 1                    /* Disable normal interrupts */
CPSR[8]   = 1                    /* Disable Imprecise Data Aborts (v6 only) */
CPSR[9]   = CP15_reg1_EEbit      /* Endianness on exception entry */
if VE==0 then
  if high vectors configured then
      PC    = 0xFFFF001C
  else
      PC    = 0x0000001C
else
  PC = IMPLEMENTATION DEFINED    /* see P. A2-26 */
```

割り込み処理の完了後、次の命令で復帰します。

```
    SUBS PC, R14,#4
```

これにより、PC（R14_fiq から）と CPSR（SPSR_fiq から）の両方が復元され、割り込まれたコードの実行が再開されます。

FIQ 例外ハンドラソフトウェアを直接 `0x0000001C` または `0xFFFF001C` に置くことができ、ベクタから分岐命令を使わずに済むよう、FIQ ベクタは意図的に最後のベクタとなっています。

A2.6.10 例外の優先度

表 A2-5 は例外の優先度を示しています。

表 A2-5 例外の優先度

優先度		例外
最高	1	リセット
	2	データアボート（データ TLB ミスを含む）
	3	FIQ
	4	IRQ
	5	不正確なアボート（外部アボート）: ARMv6
	6	プリフェッチアボート（プリフェッチ TLB ミスを含む）
最低	7	未定義命令 SWI

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

未定義命令割り込みとソフトウェア割り込みは、それぞれ現在の命令に関する固有の（オーバラップしない）デコードに対応しているため、同時に発生することはありません。プリフェッチアボートは有効な命令がフェッチされなかったことを示すので、未定義命令割り込みとソフトウェア割り込みは、どちらもプリフェッチアボートより優先度が低くなければなりません。

データアボート例外の優先度は、FIQ より高くなっています。このため、データアボートハンドラは FIQ ハンドラより前に開始されます（FIQ ハンドラの完了後にデータアボートが解決）。

A2.6.11 ハイベクタ

ハイベクタは、一部の ARMv4 実装に導入されており、ARMv6 実装には必須です。ハイベクタを使用すると、例外ベクタロケーションを、32 ビットアドレス空間の最下位にある通常のアドレス範囲、つまり `0x00000000` ～ `0x0000001C` から、最上位に近い代替アドレス範囲、つまり `0xFFFF0000` ～ `0xFFFF001C` に移動できます。この代替ロケーションを*ハイベクタ*と呼ばれます。

ARMv6 以前は、ハイベクタがサポートされているかどうかは実装定義です。ハイベクタがサポートされている場合、ハードウェアコンフィギュレーション入力により、通常ベクタとハイベクタのどちらを使用するかがリセット時に選択されます。

ARM 命令セットには、通常ベクタとハイベクタのどちらで構成するかを直接変更する命令はありません。ただし、ハイベクタをサポートする ARM プロセッサに標準的なシステム制御コプロセッサが接続されていれば、コプロセッサ 15 のレジスタ 1 のビット [13] 使用してて、通常ベクタとハイベクタを切り替えることができます。詳細については、P. B3-12 *「レジスタ 1: 制御レジスタ」*を参照して下さい。

A2.6.12 ベクタ割り込みのサポート

従来、IRQ 例外ベクタと FIQ 例外ベクタは、ハイベクタが許可されているかどうかに影響を受け、許可されていない場合は固定でした。このため、割り込みハンドラの最初には、割り込みの原因を判断し、割り込みを処理するルーチンに分岐する命令シーケンスが必要です。ベクタ割り込みをサポートしている場合、割り込みコントローラが割り込みの優先度を判断し、必要な割り込みハンドラのアドレスをコアに直接通知することが可能です。ベクタ割り込みの動作は、システムコプロセッサ CP15 のレジスタ 1 の VE ビットをセットすることで明示的に許可されます。詳細については、P. B3-12 *「レジスタ 1: 制御レジスタ」*を参照して下さい。下位互換性を維持するため、リセット時には、ベクタ割り込み機構が禁止されます。ベクタ割り込みをサポートするハードウェアの詳細は、実装定義です。

ベクタ割り込みコントローラ（VIC）は、割り込みを再許可する前に、割り込みハンドラを使わずに割り込みのソースを特定し応答することにより、実際の割り込みレイテンシを大幅に短縮します。また、割り込みハンドラルーチンへのエントリ時に、VIC とコアが適切なハンドシェイクを実行すれば、ハンドラに対応する割り込みソースと、より優先度が低いソースは、VIC によって自動的にマスクされます。これにより、割り込みの復帰情報（R14 と SPSR の値）が保存されると同時に、プロセッサコアに対して割り込みが再度許可されるため、優先度の高い割り込みの禁止時間が短縮されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.6.13 低割込みレイテンシ構成

システム制御レジスタ（CP15 のレジスタ 1）の FI ビット（ビット [21]）は、実装の割り込みレイテンシコンフィギュレーション回路の動作を許可します。詳細については、P. B3-12 *「レジスタ 1: 制御レジスタ」*を参照して下さい。このコンフィギュレーションの目的は、プロセッサの割り込みレイテンシを短縮することです。具体的な機構は実装定義です。

通常のコンフィギュレーションと低割込みレイテンシコンフィギュレーションの切替えを正しく同期するため、FI ビットは必ず実装で定義された状況で変更する必要があります。ソフトウェアシステムでは、割り込みが禁止されているリセット直後にのみ FI ビットを変更することをお勧めします。

割り込みレイテンシの短縮は、全体的な性能の低下につながる恐れがあります。このような機構の例には、コアにおけるヒットアンダーミス機能の禁止と、再開可能な外部アクセスの破棄があります。これらは、保留中の割り込みにコアが高速に応答するためのものです。低割込みレイテンシコンフィギュレーションは、メモリシステムまたはプロセッサコアのどこかに、実装定義の影響を与える場合があります。低割込みレイテンシコンフィギュレーションでは、再開可能なメモリロケーションにストアする前に割り込みが開始したと見なすこともできますが、メモリは更新されてしまうこともあります。

低割込みレイテンシコンフィギュレーションでは、ソフトウェアは、複数ワードのロード / ストア命令を完全に再開可能な方法でのみ使用する必要があります。これにより、低レイテンシ割込みコンフィギュレーションで、複数ワード命令を割り込み可能とすることができます（必ずしも割り込み可能にする必要はありません）。現在、この規則が当てはまる複数アクセス命令は次のものです。

ARM LDC、LDM（すべての形式）、LDRC、STC、STM（すべての形式）、STRD

Thumb LDMIA、PUSH、POP、STMIA

—— 注 ——

命令が完了する前に割り込みが発生した場合、1 つ以上のワードが 2 回アクセスされることがあります。システムの正当性を維持するには、同じ情報を複数回読み書きしても同じシステム結果を示すメモリが必要です。

ARMv6 では、通常属性のメモリは必ず冪等に動作しますが、デバイスまたはストロングオーダにマークされたメモリは別です（FIFO など）。低レイテンシ割込みコンフィギュレーションで、デバイスおよびストロングオーダタイプのメモリに対する複数ワードアクセスがサポートされているかどうかは、実装定義です。

————————

回復可能な形でアボートできるメモリロケーションにアクセスする複数ワードのロード / ストア命令についても、同様の状況があります。これは、アクセスされたワードのうち 1 つにアボートが発生した場合、前にアクセスされたワードが 2 回アクセスされる可能性があるためです。1 回目はアボートの前、2 回目はアボートハンドラからの復帰後です。このような場合、同じ情報を複数回読み書きしても同じシステム結果を示すか、またはアボートが最初にアクセスされるワードで発生するか、一切発生しないことが必要条件になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.6.14 例外処理を改善する新しい命令

ARMv6 には、プロセッサモードの変更や、割り込みの禁止と許可を簡素化する命令が追加されています。元のモードのスタックを使用する必要をなくし、例外エントリモードと異なるモードで例外を処理するコストを削減するための新しい命令も追加されています。次に 2 つの例を示します。

- IRQ を再許可し、同時に BL 命令を使用可能にするため、IRQ ルーチンをシステムモードまたはスーパバイザモードで実行することを考えます。IRQ モードでは、ネストした IRQ により BL の復帰リンクが破壊される可能性があるため、この動作は不可能です。新しい命令を使用すると、システムは、R13_irq や IRQ スタックを使わず、復帰状態（R14 リンクレジスタと SPSR_irq）をシステム / ユーザ、またはスーパバイザモードのスタックに保存し、システムまたはスーパバイザモードに切替えた後、IRQ を効率的に再度許可できます。
- FIQ モードは、R8_fiq ～ R13_fiq をグローバル変数として使用し、単一のオーナによる効率的な使用を目的としています。また、IRQ と異なり、FIQ は他の例外によって（リセットを除く）禁止されません。このため、仮想メモリや命令エミュレーションなど他の例外が一般的に使用される条件での、リアルタイム割り込みに適しています。IRQ は、このような処理の実行中に長時間禁止されることがあるため、リアルタイム割り込みに対応できない可能性があります。

  ただし、複数のリアルタイム割り込みソースが必要な場合、割り込みの競合が発生します。新しい機構は複数の FIQ ソースに対応しており、FIQ を禁止する時間を最小限に抑え、割り込みレイテンシのペナルティを大幅に低減しています。FIQ モードレジスタは、最も優先度の高い FIQ に単一オーナとして割り当てることができます。

SRS: 復帰状態ストア

この命令は、ベースアドレスを供給するために、指定したモードの R13 のバンクレジスタを使用して（ベースレジスタライトバックが指定されている場合は、ライトバックも実行）、R14_<current_mode> と SPSR_<current_mode> をシーケンシャルアドレスにストアします。これによって、例外ハンドラは、例外エントリシーケンスにより自動的に指定されたもの以外のスタックに復帰状態をストアできます。

使用されるアドレシングモードは、ARM アドレシングモード 4 のバージョンの 1 つ（P. A5-41 *「アドレシングモード 4 - 複数ロード / ストア」*参照）で、命令のビットマスクで指定されるリストの代わりに {R14,SPSR} レジスタリストを使用するように変更されています。これによって、SRS 命令は、スタックアクセスを行う STM 命令の通常の用法に矛盾しない形で、スタックにアクセスできます。命令の詳細については、P. A4-175 *「SRS」*を参照して下さい。

RFE: 例外からの復帰

この命令は、シーケンシャルアドレスから PC と CPSR をロードします。この命令は、SRS 命令で例外の復帰状態を保存した場合の復帰に使用されます。前と同様に ARM アドレシングモード 4 のバージョンの 1 つを使用しますが、この場合は {PC,CPSR} レジスタリストが使用されます。命令の詳細については、P. A4-114 *「RFE」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

CPS: プロセッサ状態の変更

この命令は、CPSR の割り込みマスク、モードビット、または両方に新しい値を提供し、初期のアーキテクチャバリエーションよりも、リード・モディファイ・ライト命令シーケンスを短縮し、高速化するためのものです。例外ハンドラは、SRS 命令とこの命令の両方を使用して、復帰情報を別のモードのスタックに保存した後、その別のモードに切り替えることができます。元のモードに属するスタックを変更したり、新しいモードのスタックポインタ以外のレジスタを変更したりすることはありません。

この命令は、他のコードで割り込みマスク処理とモード切り替えを合理化するためにも使用できます。また特に、開始時に割り込みを禁止し、終了時に再度許可することで、単一プロセッサシステムでのアトミックなコードシーケンスを短く、効率的に作成できます。命令の詳細については、P. A4-29「*CPS*」を参照して下さい。

現在のモードでマスクの更新を可能にする CPS Thumb 命令も提供されています。詳細については、P. A7-39「*CPS*」を参照して下さい。

——— 注 ———

Thumb 命令は、命令スペース使用上の制約により、モードを変更できません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.7 エンディアンのサポート

このセクションでは、ARM プロセッサの実装で前提となるエンディアンの観点から、メモリとメモリマップ I/O について説明します。

ARMv6 では、ハードウェアで混在エンディアンのアクセスに対応するために、アーキテクチャにいくつかの拡張が加えられています。

- バイトリバース命令 : ワード、符号付きハーフワード、符号なしハーフワードのデータをサポートするため、汎用レジスタの内容を操作します。
- 命令とデータで独立したエンディアン形式 : 命令はリトルエンディアン形式固定で、命令サイズでアラインされますが、従来の 32 ビットのワード不変バイナリイメージ /ROM もサポートします。
- PSR のエンディアン制御フラグ（E ビット）: データをレジスタファイルにロードしたり、レジスタファイルから再度ストアしたりするときに、ロードおよびストア命令スペース全体に使用されるバイト順序を決定します。以前のアーキテクチャでは、PSR のこのビットには 0 が指定されており、ARMv6 以前のアーキテクチャ用に作成された従来のコードでは決して 1 にセットされません。
- E ビットを明示的にセット / クリアする ARM 命令と Thumb 命令
- 細粒度ビッグエンディアン / リトルエンディアン共有データ構造に対応するバイト固定型アドレシング方式 : IEEE 標準 1596.5-1993（ISBN 1-55937-354-7, IEEE）の*スケーラブルコヒーレントインターフェース（SCI）プロセッサに最適化した共有データ形式向け IEEE 標準*に準拠しています。
- バスインタフェースのエンディアン形式は実装定義です。ただし、アラインされていないワードおよびハーフワードのデータアクセスについて、バイトレーン制御に対応している必要があります。

A2.7.1 アドレス空間

ARM アーキテクチャは、2^{32} 個の 8 ビットバイトで構成される、単一のフラットなアドレス空間を使用します。バイトアドレスは、0 ～ 2^{32} - 1 までの符号なし数値として扱われます。

このアドレス空間は、2^{30} 個の 32 ビットワードで構成されていると見なされます。これらの各ワードのアドレスはワード境界でアラインしているため、4 で割り切れます。アドレスが A で、ワード境界でアラインしているワードは、アドレス A、A + 1、A + 2、A + 3 の 4 バイトで構成されます。

ARMv4 以降では、アドレス空間は 2^{31} 個の 16 ビットハーフワードで構成されていると見なされることもあります。各ハーフワードはハーフワード境界でアラインしているため、アドレスは 2 で割り切れます。アドレスが A で、ハーフワード境界でアラインしているハーフワードは、アドレス A、A + 1 の 2 バイトで構成されます。

ARMv5E 以降では、アドレス空間は 64 ビットのダブルワード操作に対応しています。ダブルワード演算は、2 ワードのロード / ストア操作と考えることができます。各ワードのアドレスは次のとおりです。

- 最初のワードは A、A + 1、A + 2、A + 3
- 2 番目のワードは A + 4、A + 5、A + 6、A + 7

ARMv6 以前は、ダブルワード境界でアラインしたアドレスは常にサポートされますが、ワード境界でアラインしたダブルワード操作は予測不能です。ARMv6 では、ダブルワードのモジュロ 4、モジュロ8の両方のアライメントでの対応が必須で、アンアラインドワードおよびアンアラインドハーフワードのデータアクセスにも対応しています。これらはすべて、標準のシステム制御コプロセッサにより制御されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ARM アーキテクチャのバリエーション v5J で導入された Jazelle ステート（P. A2-15 *「T ビットと J ビット」*参照）では、バイトアドレシングがサポートされています。

アドレス計算は通常、普通の整数命令で行われます。つまり、アドレス空間をオーバフローしたり、アンダーフローした場合、通常はラップアラウンドが行われます。このため、計算結果はモジュロ 2^{32} だけ減少します。

通常のシーケンシャルな命令実行では、各命令の後で次の値が計算されます。

```
(address_of_current_instruction) + 4
```

各命令の後で、この計算によって次に実行される命令が判断されます。この計算がアドレス空間の最上位をオーバフローした場合、結果は予測不能です。言い換えれば、プログラムが、アドレス `0xFFFFFFFC` にある命令の次に、アドレス `0x00000000` にある命令をシーケンシャルに実行することはできません。

この規則は実行される命令にのみ適用されますが、これには条件コードチェックに失敗した命令も含まれます。ほとんどの ARM 実装は、現在実行中の命令よりも前方の命令のプリフェッチを行います。このプリフェッチがアドレス空間の最上位をオーバフローしても、プリフェッチした命令が実際に実行されるまでは、実装の動作は予測不能にはなりません。

LDC、LDM、LDRD、POP、PUSH、STC、STRD、STM 命令は、メモリアドレス増加方向で一連のワードにアクセスし、各ロード / ストアごとにメモリアドレスを 4 だけインクリメントします。この計算がアドレス空間の最上位をオーバフローした場合、結果は予測不能です。つまり、プログラムはアドレス `0xFFFFFFFC` にあるワードに続いてアドレス `0x00000000` にあるワードにアクセスするような命令を使うことはできません。

アンアラインドのロード / ストア命令で、アドレス計算の結果、同じ命令でアドレス `0xFFFFFFFF` にあるバイトとアドレス `0x00000000` にあるバイトの両方にアクセスが行われる場合、常に結果は予測不能です。

A2.7.2 エンディアン形式の概要

P. A2-30 *「アドレス空間」*で説明した規則から、ワード境界でアラインしたアドレス A には次の条件が必要です。

- アドレス A のワードは、アドレス A、A + 1、A + 2、A + 3 にあるバイトで構成されます。
- アドレス A のハーフワードは、アドレス A、A + 1 にあるバイトで構成されます。
- アドレス A + 2 のハーフワードは、アドレス A + 2、A + 3 にあるバイトで構成されます。
- したがって、アドレス A のワードは、アドレス A、A + 2 にあるハーフワードで構成されます。

ただしこの条件でワード、ハーフワード、バイトのマッピングを完全に指定しているわけではありません。

メモリシステムは、以下の 2 つのマッピング方式を使用します。この選択は、メモリシステムのエンディアン形式と呼ばれます。

*リトルエンディアン*のメモリシステムは、次の形式です。

- ワード境界でアラインしたアドレスにあるバイトまたはハーフワードは、そのアドレスにあるワード内の最下位バイトまたはハーフワードです。
- ハーフワード境界でアラインしたアドレスにあるバイトは、そのアドレスにあるハーフワード内の最下位バイトです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ビッグエンディアンのメモリシステムは、次の形式です。

- ワード境界でアラインしたアドレスにあるバイトまたはハーフワードは、そのアドレスにあるワード内の最上位バイトまたはハーフワードです。
- ハーフワード境界でアラインしたアドレスにあるバイトは、そのアドレスにあるハーフワード内の最上位バイトです。

表 A2-6 と表 A2-7 は、ワード境界でアラインしたアドレス A について、アドレス A のワード、アドレス A、A + 2 のハーフワード、A、A + 1、A + 2、A + 3 のバイトが、各エンディアン形式で互いにどのように対応するかを示しています。

表 A2-6 ビッグエンディアンのメモリシステム

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

表 A2-7 リトルエンディアンのメモリシステム

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

従来の ARM アーキテクチャは、32 ビットより広いメモリシステムについて、ワード不変メモリモデルをサポートしてきました。これは、ワード境界でアラインしたアドレスは、ビッグエンディアンとリトルエンディアンの両方のシステムで同じデータをフェッチするということを意味します。表 A2-8 と P. A2-33 表 A2-9 は、このモデルが 64 ビットデータパスでどのように扱われるかを示したものです。

表 A2-8 ビッグエンディアンのワード不変モデル

63　　32		31　　0	
アドレス A + 4 のワード		アドレス A のワード	
アドレス A + 4 の ハーフワード	アドレス A + 6 の ハーフワード	アドレス A の ハーフワード	アドレス A + 2 の ハーフワード

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 A2-9 リトルエンディアンのワード不変モデル

63	32	31	0
アドレス A + 4 のワード		アドレス A のワード	
アドレス A + 6 の ハーフワード	アドレス A + 4 の ハーフワード	アドレス A + 2 の ハーフワード	アドレス A の ハーフワード

ARMv6 で導入された新しい機構

ARMv6 は、混在エンディアンと呼ばれる新しい構成をサポートしています。この構成では、バイト固定アドレスモデルを使用して、ARM レジスタとの間でバイトが転送される順序を決定します。バイト固定とは、メモリ内のあるバイトのアドレスが、そのバイトへのアクセスがビッグエンディアンとリトルエンディアンのどちらの形式で行われても変化しないことを意味します。

バイト、ハーフワード、ワードのアクセスは、ビッグエンディアンとリトルエンディアンのどちらの構成でも、メモリ内の同じ 1、2、4 バイトに対して実行されます。ARM アーキテクチャにおいて、ダブルワードおよび複数ワードアクセスは、昇順のワードアドレスからの連続したワードアクセスとして扱われ、これらのアクセスについても各ワードの返すバイトは同じです。

——— 注 ———

実装が混在エンディアンモードに構成されている場合、この構成はデータアクセスと、データがレジスタファイルとの間でロード / ストアされる方法にのみ影響します。命令フェッチは常に、リトルエンディアンのバイトオーダモデルで実行されます。

- ビッグエンディアンのロード / ストア用に構成されている場合、最下位アドレスの内容が、要求されるワードまたはハーフワードの最上位バイトになります。LDRD/STRD の場合、これは、アクセスされる最初のワードの最上位バイトです。
- リトルエンディアンのロード / ストア用に構成されている場合、最下位アドレスの内容が、要求されるワードまたはハーフワードの最下位バイトになります。LDRD/STRD の場合、これは、アクセスされる最初のワードの最下位バイトです。

———————

本書では、各エンディアンモデルを以下のように区別します。

- ワード不変のビッグエンディアンモデルは、BE-32 と呼びます。
- バイト固定のビッグエンディアンモデルは、BE-8 と呼びます。
- リトルエンディアンデータはどちらのモデルでも同一で、LE と呼ばれます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.7.3 エンディアンの構成と制御

ARMv6 以前のバージョンでは、B ビットという 1 つのビットでエンディアンが制御されます。ARMv5 またはそれ以前の実装では、リトルエンディアン、ビッグエンディアンのどちらのメモリシステムをサポートしているか、または両方をサポートしているかは実装定義です。標準的なシステム制御コプロセッサが、B ビットをサポートする ARM 実装に接続されている場合、システム制御コプロセッサのレジスタ 1 のビット [7] に書き込みを行ってこの構成入力を変更できます。詳細については P. B3-12 *「レジスタ 1: 制御レジスタ」*を参照して下さい。実装によっては、リセット時に B ビットのプリセットが行われる場合があります。ARM プロセッサが、リセット時にリトルエンディアン処理に構成され、ビッグエンディアンのメモリシステムに接続されている場合、リセットハンドラは最初に以下のような命令シーケンスを使用して、構成をビッグエンディアンに変更する必要があります。

```
MRC    p15, 0, r0, c1, c0    ; r0 := CP15 register 1
ORR    r0, r0, #0x80         ; Set bit[7] in r0
MCR    p15, 0, r0, c1, c0    ; CP15 register 1 := r0
```

この操作は、バイトまたはハーフワードのデータアクセスの発生や、Thumb または Jazelle 状態での命令の実行の前に行う必要があります。

ARMv6 は、ビッグエンディアン、リトルエンディアン、バイト固定のハイブリッドシステムをサポートしています。LE 形式と BE-8 形式のサポートは必須となっています。BE-32 のサポートは実装定義です。

システム制御コプロセッサと CPSR/SPSR には、ハイブリッド処理をサポートする機能が搭載されています。使用されるシステム制御コプロセッサレジスタ（CP15 のレジスタ 1）と CPSR のビットは、次のとおりです。

- ビット [1]: A ビット。アライメントチェックの許可に使用されます。常にリセットで 0 になります（アライメントチェック OFF）。
- ビット [7]: B ビット。オプションで、下位互換性のために保持されています。
- ビット [22]: U ビット。ARMv6 でアンアラインドデータのサポートを許可し、ビット [1]（A ビット）とともに、アライメントチェック動作を決定します。
- ビット [25]: EE ビット。例外エンディアンビット。
- CPSR/SPSR[9]: E ビット。ロード / ストアのエンディアン制御。

U ビットと A ビットの各値に対応するメモリシステムの動作を表 A2-10 に示します。

表 A2-10

U	A	説明
0	0	レガシー（32 ビットワード不変のみ）
0	1	モジュロ 8 のアライメントチェック : LDRD/STRD（8 ビットと 32 ビットの不変メモリモデル）
1	0	アンアラインドアクセスのサポート（8 ビットバイト固定データアクセスのみ）
1	1	モジュロ 4 のアライメントチェック : LDRD/STRD（8 ビットと 32 ビットの不変メモリモデル）

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

EE ビットの値は、例外エントリおよびページテーブル参照時に、CPSR_E ビットの上書きに使用されます。これらは、CPSR E ビットの通常制御に対して非同期のイベントです。

BIGENDINIT コンフィギュレーションピンの代わりに、2 ビットの構成（CFGEND[1:0]）によってリセット時にハードウェアシステムの構成が行われます。CFGEND[1] は U ビットにマップし、CFGEND[0] はリセット時に B ビットまたは EE ビットと CPSR_E をセットします。

表 A2-11 は、CFGEND[1:0] のエンコードと関連する構成を定義したものです。

表 A2-11

CFGEND[1:0]	コプロセッサ 15 システム制御レジスタ（レジスタ 1）				CPSR/SPSR
	EE ビット [25]	U ビット [22]	A ビット [1]	B ビット [7]	E ビット
00	0	0	0	0	0
01[a]	0	0	0	1	0
10	0	1	0	0	0
11	1	1	0	0	1

a. この構成は、BE-32 をサポートしない実装では予約されています。その場合、B ビットからは 0 が読み出される必要があります（RAZ）。

実装にコンフィギュレーションピンが含まれていない場合、U ビットと A ビットはリセット時にクリアされます。

B ビットと E ビットの各値に対する U ビットと A ビットの使用モデルを表 A2-12 に示します。BE-32 がサポートされていない場合、B ビットからは 0 が読み出される必要があり、`B == 1` で示されるすべてのエントリは予約されています。制御ビットとデータアライメントの関係については、P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

表 A2-12 エンディアンとアライメント制御ビットの用法

U	A	B	E	命令エンディアン	データエンディアン	アンアラインドでの動作	説明
0	0	0	0	LE	LE	ローテートされた LDR	レガシー LE/ プログラムされた BE 構成
0	0	0	1	-	-	-	予約（レガシーコードには E ビットなし）
0	0	1	0	BE-32	BE-32	ローテートされた LDR	レガシー BE（32 ビットワード不変）
0	0	1	1	-	-	-	予約（レガシーコードには E ビットなし）
0	1	0	0	LE	LE	データアボート	モジュロ 8 LDRD/STRD ダブルワードアライメントチェック、LE データ

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 A2-12 エンディアンとアライメント制御ビットの用法（続き）

U	A	B	E	命令エンディアン	データエンディアン	アンアラインドでの動作	説明
0	1	0	1	LE	BE-8	データアボート	モジュロ 8 LDRD/STRD ダブルワードアライメントチェック、BE データ
0	1	1	0	BE-32	BE-32	データアボート	モジュロ 8 LDRD/STRD ダブルワードアライメントチェック、レガシー BE
0	1	1	1	-	-	-	予約
1	0	0	0	LE	LE	アンアラインド	LE 命令、LE 混在エンディアンデータ、アンアラインドアクセス許可
1	0	0	1	LE	BE-8	アンアラインド	LE 命令、BE 混在エンディアンデータ、アンアラインドアクセス許可
1	0	1	x	-	-	-	予約
1	1	0	0	LE	LE	データアボート	モジュロ 4 のアライメントチェック、LE データ
1	1	0	1	LE	BE-8	データアボート	モジュロ 4 のアライメントチェック、BE データ
1	1	1	0	BE-32	BE-32	データアボート	モジュロ 4 のアライメントチェック、レガシー BE
1	1	1	1	-	-	-	予約

BE-32 と BE-8 の定義を P. A2-31「*エンディアン形式の概要*」に示します。データアボートが発生すると、システムコプロセッサのフォルトステータスレジスタにアライメントエラーが報告されます。

——— 注 ———

U、A、B ビットはシステム制御コプロセッサのビット、E ビットは CPSR/SPSR フラグです。

E ビットをセットした結果予約状態となる場合、`SETEND` 命令（または、CPSR を変更する他の命令）の動作は予測不能です。

A2.7.4 CPSR の E ビットを変更する命令

E ビットを効率的にセット / クリアする ARM 命令と Thumb 命令は、次のとおりです。

SETEND BE CPSR E ビットのセット

SETEND LE CPSR E ビットのリセット

これらは無条件命令です。詳細については、ARM P. A4-130「*SETEND*」と Thumb P. A7-95「*SETEND*」を参照して下さい。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.7.5 汎用レジスタのバイトを反転する命令

アプリケーションドライバまたはデバイスドライバが、メモリマップされたペリフェラルレジスタや共有メモリ DMA 構造とのインターフェースとして動作する必要があり、それらのエンディアン形式が内部のデータ構造やオペレーティングシステムと異なっている場合、データのエンディアン形式を明示的に変換する効率的な方法が必要となります。

ARMv6 の ARM および Thumb 命令セットには、この機能が含まれています。

- ビッグエンディアンおよびリトルエンディアンの 32 ビット表現を変換するワード（4 バイト）レジスタ反転 : ARM P. A4-110「*REV*」と Thumb P. A7-88「*REV*」を参照して下さい。
- 符号付き 16 ビット表現を変換する、ハーフワードの反転と符号拡張 : ARM P. A4-112「*REVSH*」と Thumb P. A7-90「*REVSH*」を参照して下さい。
- ビッグエンディアンおよびリトルエンディアンの 16 ビット表現を変換する、レジスタ内のパックハーフワード反転 : ARM P. A4-111「*REV16*」と Thumb P. A7-89「*REV16*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.8 アンアラインドアクセスのサポート

ARM アーキテクチャは従来から、すべてのメモリアクセスが適切にアラインされていることを前提にしています。通常、ハーフワードアクセスに使用されるアドレスはハーフワード境界で、ワードアクセスに使用されるアドレスはワード境界でアラインしている必要があります。

ARMv6 以前は、アドレスがダブルワード境界でアラインしていない場合、メモリへのダブルワード（LDRD/STRD）アクセスの結果は予測不能です。また、アラインしていないワードおよびハーフワードのデータへのデータアクセスは、メモリインタフェースの観点からはアラインしているものとして扱われます。これは、次のようなことを意味します。

- アドレスの下位ビットが切り捨てられます。ワードアクセスの場合はアドレスビット [1:0] が 0 であると見なされ、ハーフワードアクセスの場合はアドレスビット [0] が 0 であると見なされます。
- 単一ワードをロードする ARM 命令では、ワード境界にアラインしていないアドレスによって転送されたワード境界でアラインしたデータは、アドレスビットの最下位 2 ビットの値に応じて、1、2、3 バイトのいずれかだけ右にローテートするようにアーキテクチャ上で定義されています。
- システム制御コプロセッサをサポートする実装については、CP15 のレジスタ 1 の A ビットを使用してアライメントチェックが定義されています。このビットがセットされている場合、アンアラインドアクセスが発生すると、アライメントフォルトを示すデータアボートが報告されます。

ARMv6 には、アンアラインドワードおよびハーフワードのロード / ストアデータアクセスのサポートが追加されています。これが許可されている場合、プロセッサは、トランザクションが実装定義のキャッシュライン、バス幅、ページ境界の条件を超えた場合は、1 つ以上のメモリアクセスを生成し、連続したバイトに対して要求された転送を実行します。これは、プログラマから見た場合、アクセスタイムにペナルティが発生する可能性があること以外は無視できます。この構成でも、ダブルワードアクセスはワード境界でアラインしている必要があります。

A2.8.1 アンアラインド命令フェッチ

命令フェッチは、常に境界アラインしている必要があります。具体的には、次の規則が適用されます。

- ARM ステートでは、ワード境界でアラインしている必要があります。
- Thumb ステートでは、ハーフワード境界でアラインしている必要があります。

R15 にアンアラインドアドレスを書き込んだ場合、結果は予測不能です。ただし、Thumb ステートから ARM ステートへの遷移に関連する命令の場合に限り、ビット [1] は Thumb ステートに遷移したときの有効なアドレスビットとなり、ビット [0] は遷移の発生が必要かどうかを示します。ARM ステートの BX 命令（P. A4-20「*BX*」参照）と Thumb ステートの POP 命令（P. A7-82「*POP*」参照）は、状態遷移をサポートする命令の例です。

プログラムカウンタの読み出しと書き込みに関する一般的な規則は、P. A2-9「*レジスタ 15 とプログラムカウンタ*」に定義されています。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.8.2 ARMv6 システムでのアンアラインドデータアクセス

ARMv6 は、U ビット（CP15 のレジスタ 1 のビット [22]）と A ビット（CP15 のレジスタ 1 のビット [1]）を使用し、以下のアンアラインドメモリアクセスをサポートする構成を可能にします。

- LDRH、LDRSH、STRH のアンアラインドハーフワードアクセス
- LDR、LDRT、STR、STRT のアンアラインドワードアクセス

U ビットと A ビットは、エンディアンサポートの構成にも使用されます。詳細については、P. A2-34 *「エンディアンの構成と制御」*を参照して下さい。他のすべての複数バイトロード / ストアアクセスは、ワード境界でアラインしている必要があります。

命令は常に、次に示す境界に、リトルエンディアン形式でアラインしている必要があります。

- ARM 命令は、ワード境界でアラインしている必要があります。
- Thumb 命令は、ハーフワード境界でアラインしている必要があります。

また、ARMv6 システムは、CFGEND[1:0] 条件にリセットされます。詳細については、P. A2-35 表 A2-11 を参照して下さい。

ARMv6 では、あるアクセスに対してアライメントフォルトが発生しなければならない場合と、あるアクセスの動作がアーキテクチャ的に予測不能な場合を、P. A2-40 表 A2-14 に定義しています。ここには、有効なアクセスに対して戻されるメモリロケーションも正確に記載されています。

本セクションで使用されるアクセスタイプの説明は、ロード / ストア命令から決定されます。詳細については、表 A2-13 を参照して下さい。

表 A2-13

アクセスタイプ	ARM 命令	Thumb 命令
バイト	LDRB LDRBT LDRSB STRB STRBT SWPB（いずれかのアクセス）	LDRB LDRSB STRB
ハーフワード	LDRH LDRSH STRH	LDRH LDRSH STRH
ワードロード	LDR LDRT SWP（U == 0 の場合、ロードアクセス）	LDR
ワードストア	STR STRT SWP（U == 0 の場合、ストアアクセス）	STR
ワード同期	LDREX STREX SWP（U == 1 の場合、いずれかのアクセス）	-
2 ワード	LDRD STRD	-
複数ワード	LDC LDM RFE SRS STC STM	LDMIA POP PUSH STMIA

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アクセスされるメモリロケーションの説明に、以下の用語が使用されます。

バイト [X] 現在のエンディアン形式モデルで、アドレスが X のバイトを意味します。エンディアン形式モデル間の対応として、LE エンディアン形式モデルのバイト [A]、BE-8 エンディアン形式モデルのバイト [A]、BE-32 エンディアン形式モデルのバイト [A EOR 3] が、メモリ内で同じバイトとなります。

ハーフワード [X]

現在のエンディアン形式モデルでアドレスが X、X + 1 のバイトで構成されるハーフワードを意味します。これらのバイトは、LE エンディアン形式モデルではリトルエンディアン順序、BE-8 または BE-32 エンディアン形式モデルではビッグエンディアン順序で結合し、ハーフワードを形成します。

ワード [X] 現在のエンディアン形式モデルで、アドレスが X、X + 1、X + 2、X + 3 のバイトで構成されるワードを意味します。これらのバイトは、LE エンディアン形式モデルではリトルエンディアン順序、BE-8 または BE-32 エンディアン形式モデルではビッグエンディアン順序で結合し、ワードを形成します。

——— 注 ———

これらの定義により、X がワード境界でアラインしている場合、ワード [X] は、LE および BE-32 エンディアン形式モデルの同じ順序で並んだ同じ 4 バイトのメモリで構成されます。

アライン [X] （`X AND 0xFFFFFFFC`）を意味します。つまり、X の最下位 2 ビットが 0 にクリアされ、ワード境界にアラインされます。

——— 注 ———

アドレス [1:0] == 0b00 の行では、アドレスとアライン（アドレス）は同一です。実装では、これを利用し、いつ最下位ビットを 0 にクリアするかの制御を簡素化できます。

---

2 ワードおよび複数ワードのアクセスタイプでは、「アクセスされるメモリ」の列に、アクセスされる最下位ワードのみが記載されています。それ以降のワードは、最下位ワードを 4 ずつインクリメントしたアドレスで構成され、最下位ワードと同じエンディアン形式モデルが使用されます。

表 A2-14 ARMv6 システムでのデータアクセス動作

U	A	アドレス [2:0]	アクセスタイプ	動作	アクセスされるメモリ	注
0	0					レガシー、アライメントフォルトなし
0	0	xxx	バイト	通常	バイト [アドレス]	-
0	0	xx0	ハーフワード	通常	ハーフワード [アドレス]	-
0	0	xx1	ハーフワード	予測不能	-	-

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 A2-14 ARMv6 システムでのデータアクセス動作（続き）

U	A	アドレス [2:0]	アクセスタイプ	動作	アクセスされるメモリ	注
0	0	xxx	ワードロード	通常	ワード [アライン（アドレス）]	8 * アドレス [1:0] ビットだけ右ローテートされたデータがロードされる
0	0	xxx	ワードストア	通常	ワード [アライン（アドレス）]	この処理はアドレス [1:0] の影響を受けない
0	0	x00	ワード同期	通常	ワード [アドレス]	-
0	0	xx1、x1x	ワード同期	予測不能	-	-
0	0	xxx	複数ワード	通常	ワード [アライン（アドレス）]	この処理はアドレス [1:0] の影響を受けない
0	0	000	2 ワード	通常	ワード [アドレス]	-
0	0	xx1、x1x、1xx	2 ワード	予測不能	-	-
1	0					新しい ARMv6 アンアラインドサポート
1	0	xxx	バイト	通常	バイト [アドレス]	-
1	0	xxx	ハーフワード	通常	ハーフワード [アドレス]	-
1	0	xxx	ワードロード ワードストア	通常	ワード [アドレス]	-
1	0	x00	ワード同期 複数ワード 2 ワード	通常	ワード [アドレス]	-
1	0	xx1、x1x	ワード同期 複数ワード 2 ワード	アライメントフォルト	-	-
x	1					フルアライメントフォルト
x	1	xxx	バイト	通常	バイト [アドレス]	-
x	1	xx0	ハーフワード	通常	ハーフワード [アドレス]	-
x	1	xx1	ハーフワード	アライメントフォルト	-	-

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 A2-14 ARMv6 システムでのデータアクセス動作（続き）

U	A	アドレス [2:0]	アクセスタイプ	動作	アクセスされるメモリ	注
x	1	x00	ワードロード ワードストア ワード同期 複数ワード	通常	ワード [アドレス]	-
x	1	xx1、x1x	ワードロード ワードストア ワード同期 複数ワード	アライメントフォルト	-	-
x	1	000	2 ワード	通常	ワード [アドレス]	-
0	1	100	2 ワード	アライメントフォルト	-	-
1	1	100	2 ワード	通常	ワード [アドレス]	-
x	1	xx1、x1x	2 ワード	アライメントフォルト	-	-

アンアラインドアクセスが予測不能となる他の理由

P. A2-40 表 A2-14 に記載されている動作には以下の例外があり、これらの場合に発生するアンアラインドアクセスは予測不能です。

- PC をロードし、アドレス [1:0] != 0b00 で、通常の動作を行うことが表に記載されている LDR 命令の動作は予測不能になります。

  ——— 注 ———

  これが LDR にのみ適用される理由は、PC がデスティネーションレジスタに指定されている場合、他のほとんどのロード命令はアライメントに関係なく予測不能なためです。例外は、LDM、RFE、Thumb の POP です。これらの命令でアドレス [1:0] != 0b00 の場合、A == 0 かつ U ==0 であると、転送の実効アドレスの最下位 2 ビットは強制的に 0 になります。それ以外の場合、表に記載されている動作は、デスティネーションレジスタに関係なく、予測不能またはアライメントフォルトのいずれかです。

  ————————

- ストロングオーダまたはデバイスのメモリ属性でメモリにアクセスし、アドレス [1:0] != 0b00 で、通常の動作を行うことが表に記載されているすべてのワードロード、ワードストア、ワード同期、2 ワード、複数ワード命令の動作は予測不能になります。
- ストロングオーダまたはデバイスのメモリ属性でメモリにアクセスし、アドレス [0] != 0 で、通常の動作を行うことが表に記載されているすべてのハーフワード命令の動作は予測不能になります。

上記の理由のいずれかが該当する場合は、表に記載された動作より優先されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

——— 注 ———

上記の理由が、アライメントフォルト動作に優先することはありません。

———

下位アドレスビットが 0 でなく、アクセスがアンアラインドとなる場合、ARM 実装がアドレスをメモリに送信するときにそれらの下位アドレスビットを必ずクリアする必要はありません。代わりに、ロード / ストア命令によって計算されたアドレスをそのままメモリに送信し、メモリシステムがハーフワードアクセスのアドレス [0] とワードアクセスのアドレス [1:0] を無視することを要求してもかまいません。

命令で、アンアラインドアクセスを生成する下位アドレスビットが無視される場合、命令の説明にある擬似コードは、それらのビットを明示的にマスクしません。代わりに、擬似コードに使用されているメモリ [< アドレス >、< サイズ >] 関数によってこれらのビットは暗黙的にマスクされます。

ARMv6 のアンアラインドデータアクセスに関する制約

ARMv6 には、アンアラインドデータアクセスに関して以下の制約があります。

- アンアラインドアクセスは、アトミックに処理されるとは限りません。ロックされたトランザクションサイクルを保証することなく、共有メモリシステム内にある境界でアラインされた一連の操作から合成されることもあります。
- アンアラインドアクセスは通常、境界でアラインした転送に比べ、完了までに多数のサイクルが必要です。このため、リアルタイム性に関する影響を慎重に分析し、場合によっては、パフォーマンスを最適化するよう主要なデータ構造のアライメントを調整する必要があります。
- アクセスがページ境界をまたいで発生する場合、2 分されたアクセスのどちらか、または両方でアボートが発生することがあります。データアボートハンドラは、アライメントフォルトステータスコードが通知された後で、再開可能アボートを慎重に処理する必要があります。

したがって、共有メモリ方式では、バイト幅を超えるデータ項目のロード、ストア、スワップで、境界でアラインしていないデータが一律に更新されることを前提とはできません。

アンアラインドアクセス操作は、デバイスのメモリマップされたレジスタへのアクセスには使用できません。また、境界でアラインしたセマフォや同期変数によって保護される共有メモリ構造では、注意して使用する必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.9 同期化基本命令

従来、共有メモリの同期化は、レジスタの内容とメモリの内容をスワップする「読み出し - ロック - 書き込み」命令、つまり、P. A4-213 *「SWP」*と P. A4-215 *「SWPB」*に記載されている SWP 命令と SWPB 命令によってサポートされていました。これらの命令は、基本的な「ビジー / 開放」のセマフォ機構に対応していますが、読み出しフェーズと書き込みフェーズの間でセマフォの計算を必要とする機構はサポートしていません。ARMv6 は、新しい機構により、マルチプロセッサシステムの設計にも拡大適用できる包括的なノンブロッキング共有メモリ同期化基本命令をサポートしています。

―――― 注 ――――

ARMv6 では、スワップ命令、スワップバイト命令は推奨されません。すべてのソフトウェアについて、新しい同期化基本命令に移行することをお勧めします。

――――――――

ARM 命令セットには、2 つの命令が導入されました。

- P. A4-53 *「LDREX」*に説明されている排他ロード命令
- P. A4-203 *「STREX」*に説明されている排他ストア命令

これらの命令は、ステートマシンおよび関連のメモリアクセス用システム制御機能を提供するアドレスモニタと協調して動作します。メモリが共有可、非共有のいずれのメモリ属性を持つかによって、モニタモデルは 2 つあります。詳細については、P. B2-12 *「共有属性」*を参照して下さい。単一プロセッサシステムは、最小限のハードウェアオーバヘッドで同期化基本命令をサポートできるよう、非共有メモリモデルのみをサポートすれば十分です。最小限のシステムの例については、図 A2-2 を参照して下さい。

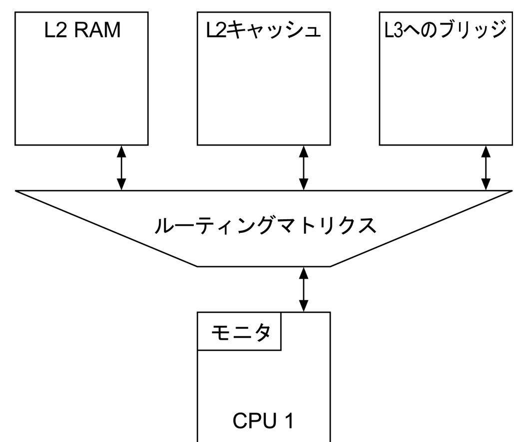


図 A2-2 単一プロセッサ（非共有）モニタの例

マルチプロセッサシステムでは、各プロセッサにアドレスモニタを実装する必要があります。メモリシステム階層内でモニタが存在する場所と、モニタがすべての共有アクセスに対して可視なプロセッサごとに単一エンティティとして実装されているか、分散したエンティティとして実装されているかは、実装定義です。P. A2-45 図 A2-3 は、モニタが共有、非共有の両方の場合についてのステートマシンをサポートする単一エンティティ方式を示しています。共有属性の場合のみ、スヌープが必要です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

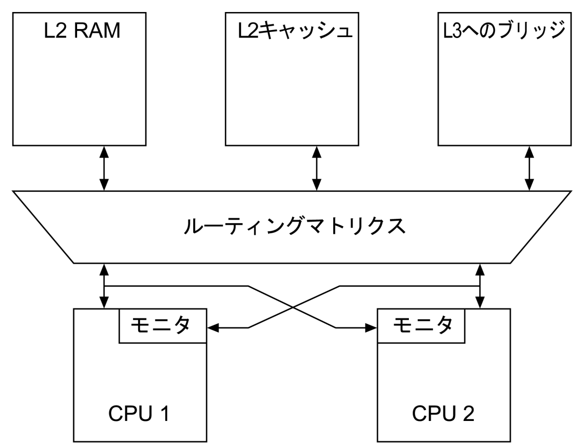


図 A2-3 書き込みスヌープモニタ方式

図 A2-4 は、ローカルモニタがプロセッサブロックにあり、グローバルモニタがターゲットに分散している分散モデルを示しています。

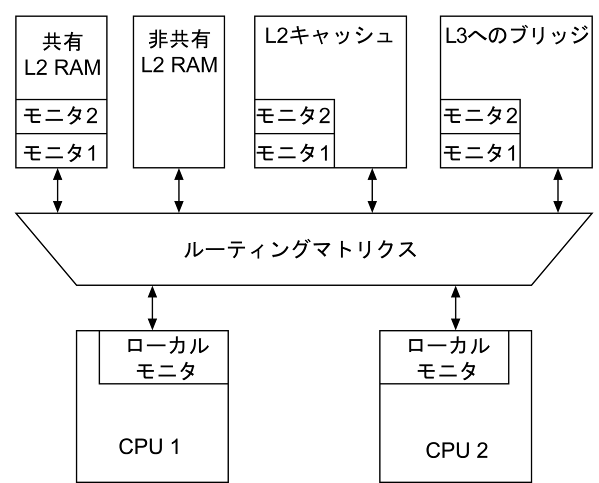


図 A2-4 ターゲットモニタ方式

A2.9.1 排他アクセス命令：非共有メモリ

*共有*という TLB 属性を持たないメモリ領域の場合、排他アクセス命令は、排他ロードが実行されたという事実にタグをつける機能に依存しています。排他ロードを実行したプロセッサが排他ストアを使用してどのアドレスでも変更を試みた場合、アボートが発生しない限り、このタグは必ずクリアされます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

―――― 注 ――――

非共有メモリでは、タグの付いた物理アドレスへのストアによってタグがクリアされるかどうかは、その物理アドレスにタグを付けたプロセッサ以外のプロセッサによるストアの場合、予測不能です。

――――――――

排他ロード命令は、メモリからのロードを実行、その実行するプロセッサは、非共有メモリに対して未解決のタグ付き物理アドレスがあるというタグを付けます。モニタの状態は、排他アクセスに遷移します。

排他ストア命令は、メモリへの条件付きストアを実行します。ストアは、実行するプロセッサのローカルモニタが排他アクセス状態にある場合のみ、実行されます。0b0 という状態値がレジスタに戻され、実行するプロセッサのモニタがオープンアクセス状態に遷移します。ストアが妨げられた場合、命令で定義されたレジスタに 0b1 という値が戻されます。

プロセッサが、排他ストア以外の命令を使用して、ローカルモニタの対象となっていない物理アドレスへの書き込みを行っても、ローカルモニタの状態は変わりません。モニタの対象となっている物理アドレスへの書き込み（排他ストア以外）がローカルモニタの状態に影響するかどうかは、実装定義です。

プロセッサが、排他ロードを最後に実行したアドレス以外の、非共有メモリ中の何らかのアドレスに対して排他ストアを実行し、モニタが排他状態にある場合、ストアが成功するかどうかは実装定義です。この機構は、コンテキストスイッチに使用されています（セクション P. A2-48 *「コンテキストスイッチのサポート」*参照）。これは、どのような場合も、ソフトウェアプログラミングエラーとして扱う必要があります。

関連するデータモニタのステートマシンを図 A2-5 に示します。

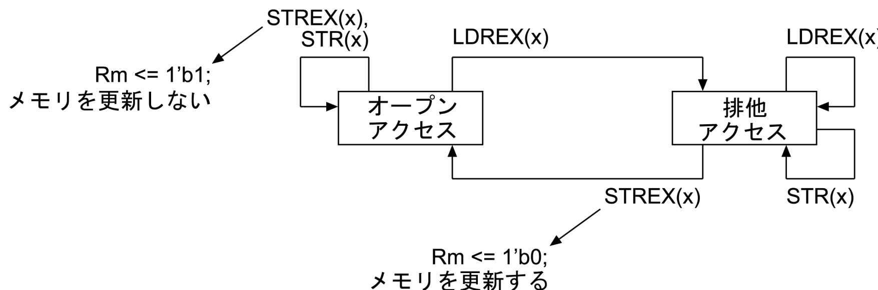


図 A2-5 状態図：ローカルモニタ

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

—— 注 ——

ローカルモニタ用の実装定義オプションは、物理アドレスを保持せず全てのアクセスを以前の LDREX のアドレスと一致するものとして扱うように構成されているローカルモニタと、一貫性があります。

示されている動作は、LDREX、STREX、STR 命令を発行するプロセッサに関連するローカルアドレスモニタのものです。排他アクセスからオープンアクセスへの遷移は、異なるプロセッサが STR または STREX を発行した場合、予測不能となります。他のプロセッサからのトランザクションは、このモニタにとって可視の必要はありません。

A2.9.2 排他アクセス命令：共有メモリ

*共有*という TLB 属性を持つメモリ領域の場合、排他アクセス命令は、物理アドレスにタグを付け、特定プロセッサの排他アクセスであることを示すグローバルモニタの機能に依存しています。このタグは後で、そのアドレスに排他ストアを発生させる必要があるかどうか判断するのに使用されます。プロセッサがそのアドレスの変更を試み、アボートが発生しなかった場合、このタグは必ずクリアされます。

グローバルモニタは、P. A2-45 図 A2-3 に示すようにプロセッサブロックにあるか、P. A2-45 図 A2-4 に示すように、二次モニタとしてメモリインターフェースにあります。グローバルモニタとローカルモニタの機能は、実装では 1 つのモニタに統合することができます。

*共有*メモリからの排他ロードは、メモリからのロードを実行し、要求を行うプロセッサの排他アクセスであることを示すタグを物理アドレスに付けます。これにより、要求を行うプロセッサによりタグ付けされた他の物理アドレスについて、排他アクセスを示すタグがクリアされます。共有可能メモリに対する未解決の排他アクセスは、プロセッサにつき 1 つしかサポートされません。

排他ストアは、メモリへの条件付きストアを実行します。このストアは、要求を行うプロセッサの排他アクセスであることを示すタグが物理アドレスに付いている場合のみ、実行が保証されます。排他アクセスを示すタグのついたアドレスがない場合、ストアは成功しません。要求を行うプロセッサの排他アクセスを示すタグが、異なる物理アドレスに付けられている場合、ストアが成功するかどうかは実装定義です。ストアの成功を示す応答として、ステータス値 0b0 がレジスタに戻されます。成功しない場合は、値 0b1 が戻されます。要求を行うプロセッサの排他アクセスを示すタグが物理アドレスに付いている場合、排他モニタの状態はオープンアクセス状態に遷移し、モニタが既にオープンアクセス状態の場合はその状態を維持します。タグが付けられていない場合、モニタが排他アクセス状態を維持するか、オープンアクセス状態に遷移するかは実装定義です。

共有メモリシステムのプロセッサ（または独立した DMA エージェント）はすべて、専用のアドレスモニタを必要とします。マルチプロセシング環境において、プロセッサ (n) に関連するグローバルアドレスモニタのステートマシンは、次のような自分にとって可視であるすべてのメモリアクセスに反応します。

- 関連するプロセッサ (n) が生成したトランザクション
- 共有メモリシステムの他のプロセッサ (!n) に関連するトランザクション

動作については、P. A2-48 図 A2-6 を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

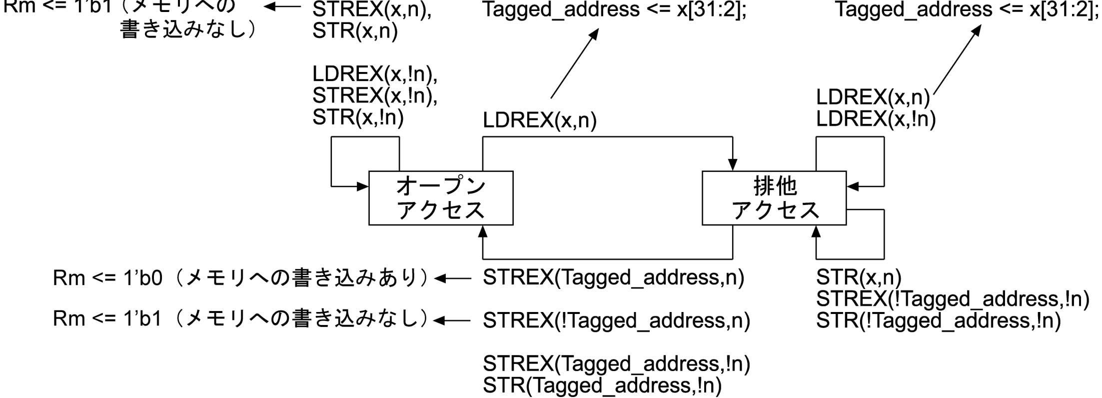


図 A2-6 状態図：グローバルモニタ

——— 注 ———

STREX がメモリの更新に成功するかどうかは、関連するグローバルモニタとタグアドレスが一致するかどうかに依存します。したがって、(!n) エントリは、ステートマシンの状態遷移にどのように影響するかという点のみが示されています。同様に、LDREX は、関連するグローバルモニタのタグだけを更新できます。

———————

A2.9.3 コンテキストスイッチのサポート

コンテキストスイッチでは、コンテキストスイッチの後、ローカルモニタが必ずオープンアクセス状態になるようにして下さい。このためには、この目的で割り当てたメモリ内のアドレスに対してダミーの STREX を実行する必要があります。

パフォーマンス上の理由から、この排他ストア命令は排他ロード命令から数命令以内に置くことをお勧めします。これにより、コンテキストスイッチオーバヘッドの可能性、またはマルチプロセッサのアクセスの競合によって排他ストアが失敗し、ロード / ストアシーケンスの再実行が必要となる可能性が最小限に抑えられます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.9.4 操作の概要

排他アクセス操作は、以下の擬似関数を使用して記述できます。

- TLB(<Rm>)
- Shared(<Rm>)
- ExecutingProcessor()
- MarkExclusiveGlobal(<physical_address>,<processor_id>,<size>)
- MarkExclusiveLocal(<physical address>,<processor_id>,size>)
- IsExclusiveGlobal(<physical_address>,<processor_id>,<size>)
- IsExclusiveLocal(<physical_address>,<processor_id>,<size>)
- ClearExclusiveByAddress(<physical_address>,<processor_id>,<size>)
- ClearExclusiveLocal(<processor_id>).

1. CP15 のレジスタ 1 のビット [0]（M ビット）がセットされている場合、TLB(<Rm>) は、実行プロセッサのカレントプロセス ID および TLB エントリについて、Rm が示す仮想アドレスに対応する物理アドレスを戻します。M ビットがセットされていない場合、またはシステムが仮想から物理への変換を実装していない場合は、Rm にある値を戻します。

2. CP15 レジスタ 1 のビット [0]（M ビット）がセットされている場合、Shared(<Rm>) は、VMSA、PMSA 領域記述子に対する実行プロセッサのカレントプロセス ID および TLB エントリについて、Rm が示す仮想アドレスに対応する共有メモリ領域属性を返します。M ビットがセットされていない場合、戻される値は、メモリシステム動作の関数です（第 B4 章 *「仮想メモリシステムアーキテクチャ」*、第 B5 章 *「保護メモリシステムアーキテクチャ」* 参照）。

3. ExecutingProcessor() は、そのシステム内にあるすべてのプロセッサの中で、その操作をしているプロセッサを固有に識別可能な値を返します。

4. MarkExclusiveGlobal(<physical_address>,<processor_id>,<size>) は、プロセッサ <processor_id> が、アドレス <physical_address> から少なくとも <size> バイトの範囲を対象とする排他アクセスを要求したことを記録します。排他とマークされた領域のサイズは、実装定義で、最大 128 バイトの制限があり、<size> より小さくはなりません。また、アドレス空間で領域のサイズにアラインしています。これによって、以前に同じプロセッサが他のアドレスに対して要求した排他アクセスがクリアされるかどうかは、予測不能です。

5. MarkExclusiveLocal(<physical_address>,<processor_id>,<size>) は、プロセッサ <processor_id> が、<physical_address> から少なくとも <size> バイトの範囲を対象とするアドレスに排他アクセスを要求したことを、ローカルレコードに記録します。排他とマークされた領域のサイズは実装定義で、最大でメモリ全体を対象とする場合がありますが、<size> より小さくはなりません。また、アドレス空間で領域のサイズにアラインしています。これにより MarkExclusiveGlobal(<physical_address>,<processor_id>,<size>) も実行されるかどうかは、実装定義です。

6. プロセッサ <processor_id> がグローバルレコードにおいて、アドレス <physical_address> から少なくとも <size> バイトを含むアドレス範囲を、排他アクセス要求にマークしている場合、IsExclusiveGlobal(<physical_addess>,<processor_id>,<size>) は、TRUE を戻します。グローバルレコードで、他のアドレスに排他アクセスの要求がマークされている場合、TRUE、FALSE のどちらが返されるかは実装定義です。排他アクセスとしてマークされているアドレスがグローバルレコードにない場合、IsExclusiveGlobal(<physical_addess>,<processor_id>,<size>) は FALSE を返します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

7. プロセッサ <processor_id> が、アドレス <physical_address> から少なくとも <size> バイトを含むアドレス範囲を、排他アクセス要求にマークしている場合、IsExclusiveLocal(<physical_addess>,<processor_id>,<size>) は TRUE を返します。排他アクセス要求にマークされたアドレスが、アドレス <physical_address> から <size> バイトの全体を含んでいない場合、この関数が TRUE、FALSE のどちらを戻すかは実装定義です。排他アクセス要求にマークされているアドレスがない場合、この関数は FALSE を返します。この結果と、IsExclusiveGlobal(<physical_addess>,<processor_id>,<size>) の結果の論理積を取るかどうかは実装定義です。

8. ClearExclusiveByAddress(<physical_address,<processor_id>,<size>) は、少なくとも <physical_address> と（<physical_address> + <size> - 1) の間のすべてのバイトを含むアドレスに対して排他アクセスの要求があったとして、<processor_id> 以外のすべてのプロセッサのグローバルレコードをクリアします。クリアされる領域のサイズは実装定義で、最大 128 バイトの制限があり、<size> より小さくはなりません。また、アドレス空間で領域のサイズにアラインしています。プロセッサ（1 つ以上）のローカルレコードもクリアされるかどうかは、実装定義です。

   <physical_address> と（<physical_address> + <size> - 1）の間のすべてのバイトについて排他アクセスが要求されている場合、または排他アクセスが要求されているアドレスが他にある場合、プロセッサ <processor_id> のグローバルレコードもクリアされるかどうかは、実装定義です。

9. ClearExclusiveLocal(<processor_id>) は、アドレスが排他アクセスの要求を受けたとして、プロセッサ <processor_id> のローカルレコードをクリアします。この操作が、アドレスが排他アクセスの要求を受けたとして、プロセッサ <processor_id> のグローバルレコードもクリアするかどうかは、実装定義です。

この定義において、*プロセッサ*は、仮想システムコンポーネントを含め、メモリトランザクションを生成できるシステムコンポーネントと定義されます。*processor_id* は、プロセッサ固有の識別子として定義されます。

他のストア操作に対する影響

実行されたすべてのストア操作は、擬似コード処理に以下の機能的動作を加えます。

```
if (Shared(address))
    physical_address = TLB(address)
    ClearExclusiveByAddress(physical_address,processor_id,size)
```

ロード / ストア操作

排他アクセスは、レジスタファイルの使い方の点から説明することができます。

- Rd: ロード時のデータ、ストア時のステータスに使用するデスティネーションレジスタ
- Rm: ストアに使用するソースデータレジスタ
- Rn: ロードとストアに使用するメモリアドレスレジスタ

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

擬似表現は次のとおりです。

LDREX 操作

```
if ConditionPassed (cond) then
    processor_id = ExecutingProcessor()
    Rd = Memory[Rn,4]
    physical_address = TLB(Rn)
    if Shared(Rn) == 1 then
        MarkExclusiveGlobal(physical_address,processor_id,4)
    MarkExclusiveLocal(physical_address,processor_id,4)
```

STREX 操作

```
if ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    physical_address = TLB(Rn)
    if IsExclusiveLocal(physical_address,processor_id,4) then
        if Shared(Rn) == 1 then
            if IsExclusiveGlobal(physical_address,processor_id,4) then
                Memory[Rn,4] = Rm
                Rd = 0
                ClearExclusiveByAddress(physical_address,processor_id,4)
            else
                Rd = 1
        else
            Memory[Rn,4] =Rm
            Rd = 0
    else
        Rd = 1
    ClearExclusiveLocal(processor_id)
```

——— 注 ———

排他操作をサポートしていない（排他モニタを実装していないなど）共有メモリ領域における STREX の動作は、予測不能です。

命令の動作の完全な定義については、P. A4-53「*LDREX*」と P. A4-203「*STREX*」を参照して下さい。

使用上の制限

LDREX 命令と STREX 命令は、協調して動作するよう設計されています。これらの機能の多様な実装に対応するため、以下の注釈と制限に従って下さい。

1. 排他操作において、サポートされる未解決の排他アクセスは、実行される各プロセッサスレッドにつき 1 つです。アーキテクチャは、IsExclusiveLocal() 関数の一部としてアドレスチェックやサイズチェックを義務化しないというかたちで、このことを利用します。STREX のターゲットアドレスが、同じ実行スレッド内の先行する LDREX と異なる場合、予測不能な動作につながることがあります。このため、LDREX/STREX のペアが最終的に成功すると保証できるのは、同じアドレスに対して実行される場合のみです。コンテキストスイッチや例外によって実行ス

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

レッドが変更される可能性がある場合は、P. A2-48 *「コンテキストスイッチのサポート」* に記載されているダミー STREX 命令を実行し、望ましくない影響を防ぐ必要があります。この場合のみ、STREX を前に実行された LDREX と異なるアドレスに対してプログラムします。

2. メモリへの明示的なストアは、他のプロセッサに関連する排他モニタをクリアする可能性があります。このため、LDREX と STREX の間にストアを実行すると、排他ロックが失敗する原因になります。したがって、1 つのコードシーケンス内で LDREX と STREX の間に明示的なストアを置かないようなコードにする必要があります。

3. LDREX を間にはさまないで 2 つの STREX 命令が実行される場合も、2 番目の STREX が FALSE を返す結果になります。このため、1 つの実行スレッドにおいて、STREX の前に常に LDREX が先行する必要があります。しかし、LDREX の後に必ず STREX が実行される必要はありません。

4. 実装によっては、キャッシュ追い出しなどが原因で、明らかに不法な排他モニタのクリアが、LDREX と STREX の間に起こることがあります。このような実装で動作するためのコードでは、LDREX 命令と STREX 命令の間に、明示的なメモリトランザクションやキャッシュ保守操作を入れないようにする必要があります。

5. 実装によっては、コードシーケンス内の LDREX 操作と STREX 操作を近くに置く方が有利です。こうすると、排他モニタ状態が不法にクリアされる可能性が減少します。また、最高のパフォーマンスを得るためには、1 つのコードシーケンスにおける LDREX 命令と STREX 命令の間を 128 バイト以内に制限することが強く推奨されます。

6. コヒーレントなプロトコルを持つ実装、またはシングルマスタの実装では、1 つのプロセッサについてローカルモニタとグローバルモニタが統合される場合があります。P. A2-49 *「操作の概要」* に記載されている定義のうち、実装定義、予測不能の部分は、この動作を説明するために設けられています。

7. アーキテクチャは、排他とマークされる可能性のある領域に 128 バイトの上限を設定します。したがって、パフォーマンス上、ソフトウェアでは排他アクセスを受けるオブジェクトを 128 バイト以上離しておくことが推奨されます。これは機能的な要件ではなく、パフォーマンス上の指針です。

8. LDREX 操作と STREX 操作は、ノーマルメモリ属性をサポートするメモリに対してのみ実行します。

9. モニタの状態に関するデータアボートの影響は、予測不能です。アボート処理コードでダミーの STREX 命令を実行し、モニタ状態をクリアすることをお勧めします。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.10 Jazelle 拡張

Jazelle 拡張は、ARMv5 のバリエーションである ARMv5TEJ で導入されたもので、ARMv6 では必須です。Jazelle 拡張により、Java 仮想マシン（JVM）によるオペコード実行のハードウェアアクセラレーションが、アーキテクチャでサポートされます。Jazelle 拡張により、プロセッサに高速オペコード実行機能が存在するかどうかに関係なく、かつ存在する場合はそのすべてを自動的に利用するよう JVM を作成することができます。最も単純な実装では、プロセッサはオペコードの実行のアクセラレーションは行わず、すべてのオペコードがソフトウェアルーチンで実行されます。これは、Jazelle 拡張のトリビアル実装と呼ばれ、Jazelle 拡張をまったく実装しない場合に比べ、最小限のコストで済みます。Jazelle 拡張の非トリビアル実装では、通常、ハードウェア実装が単純で Jazelle 実行時間の大部分を占めるオペコードを選んだサブセットが、ハードウェアに実装されます。

非トリビアル実装で必須の機能は次のとおりです。

- CPSR および各 SPSR への状態ビット（J ビット）追加
- Jazelle ステートを開始する新しい命令（BXJ）
- 完全な 32 ビットバイトアドレシングに対応する PC 拡張
- 例外モデルの変更
- JVM が Jazelle 拡張ハードウェアを具体的なニーズに合わせて構成するための機構
- OS が Jazelle 拡張ハードウェアの使用を統制するための機構

トリビアル実装で必須の機能は次のとおりです。

- 存在する実行ステートが、ARM ステートと Thumb ステートのみであること。J ビットは常に 0 として読み出し書き込みが許されます。J ビットが 1 に更新された場合、以下の命令の実行は未定義です。
- BXJ 命令が BX 命令として動作すること。
- インタフェースを永久的に禁止する構成のサポート。

オペコード実行のハードウェアアクセラレーションを自動的に利用するよう作成された JVM を、イネーブルド JVM（EJVM）と呼びます。

A2.10.1 サブアーキテクチャ

Jazelle 拡張を含む ARM 実装では、Jazelle ステート実行が開始および終了する際、ARM プロセッサの汎用レジスタと他のリソースが呼び出し規則に従う必要があります。例えば、特定の汎用レジスタは、現在のオペコードを指すポインタとして使うために予約される可能性があります。EJVM または関連するデバッグサポートが正しく機能するためには、Jazelle ステート実行の開始点と終了点で、アクセラレーションハードウェアが期待する呼び出し規則に従うよう記述されている必要があります。

EJVM は呼び出し規則に依存しますが、他のシステムソフトウェアは一般にこの規則に依存しません。これにより、オペコード実行の大幅なパフォーマンス向上など十分な技術的利点が得られるのであれば、規則を変更するコストに見合う価値があると考えられます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

各種の規則は、実装のサブアーキテクチャと集合的に呼ばれます。本書では、サブアーキテクチャに関する説明は行いません。サブアーキテクチャを利用できるのは、上記に説明した EJVM 実装、およびデバッグソフトウェア、同様のソフトウェアだけです。その他のソフトウェアは、このセクションで説明する Jazelle 拡張の一般的なアーキテクチャ定義にだけ従うようにして下さい。特定のサブアーキテクチャを識別するには、P. A2-62「*Jazelle ID レジスタ*」に記載されている Jazelle ID レジスタの説明を参照して下さい。

A2.10.2 Jazelle ステート

Jazelle 拡張機能は、プロセッサステータスレジスタ（CPSR、バンク SPSR）に追加された状態ビット (J) を利用します。これは、それぞれのレジスタのビット [24] です。

31	30	29	28	27	26 25	24	23 20	19 16	15 10	9	8	7	6	5	4 0
N	Z	C	V	Q	予約	J	予約	GE[3:0]	予約	E	A	I	F	T	モード

その他のビットフィールドについては、P. A2-11「*プログラムステータスレジスタ*」を参照して下さい。

——— 注 ———

J ビットがフラグバイトに配置されているのは、ARMv5TE 以前のプロセッサで実行されるコードにおける状態バイトまたは拡張バイトの使用を避けるためです。これにより、推奨されない CPSR、SPSR、CPSR_all または SPSR_all 文法を `MSR` 命令のデスティネーションに使用するように作成された OS コードが動作しなくなるのは、ARMv6 に導入された機能、つまり E、A、GE のビットフィールドが使用される場合のみであることが保証されます。

また、`MSR` 命令が実行される際、J は常に 0 です。これにより、`MSR CPSR_f,#0xF0000000` など、既知状態にフラグを置く既存命令が他の部分に影響を与えないことが保証されます。

———————

J ビットは、T ビットとともにプロセッサの実行状態を示します。詳細については、表 A2-15 を参照して下さい。

表 A2-15

J	T	実行状態
0	0	32 ビット ARM 命令を実行する ARM ステート
0	1	16 ビット Thumb 命令を実行する Thumb ステート
1	0	可変長の Jazelle オペコードを実行する Jazelle ステート
1	1	未定義、将来の拡張用に予約済み

J ビットは、以下の点で T ビットと同様に扱われます。

- 例外エントリの際、どちらのビットも CPSR から例外モードの SPSR にコピーされた後、CPSR でクリアされて、プロセッサを ARM ステートにします。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- Rd = R15 で、S ビットがセットされたデータ処理命令により、両方のビットは SPSR から CPSR にコピーされ、その結果移行するステートで実行が再開されます。これによって、これらの命令が通常の、例外からの復帰機能を果たすことが保証されます。

  これらの例外からの復帰は、プロセッサ例外エントリが生成した SPSR と R14 の値を使用し、該当する例外の復帰命令を使用する必要があります。詳細については、P. A2-16 *「例外」*を参照して下さい。戻り値が、SPSR 値の J == 1、T == 0 で使用される場合、結果はサブアーキテクチャ定義です。

- 同様に、PC がレジスタリストにあり、「^」指定の `LDM` 命令の場合（P. A4-41 *「LDM (3)」*に記載されている `LDM` (3) 命令）、両方のビットは SPSR から CPSR にコピーされ、その結果移行する状態で実行が再開されます。これらの命令は例外からの復帰にも使用され、同様に前の項目の内容が適用されます。

- 特権モードの場合、CPSR の J ビットまたは T ビットを 1 にセットする `MSR` 命令の実行は、予測不能の結果を招きます。

- 非特権（ユーザ）モードの場合、CPSR の J ビットまたは T ビットを 1 にセットする `MSR` 命令を実行しても、これらのビットは変更されません。

- J == 1、T == 1 のセットは、Thumb に対応していないプロセッサで T == 1 をセットするのと同様の効果があります。つまり、次の命令を実行すると、未定義命令例外が発生します。例外ハンドラへのエントリ時に、プロセッサは再度 ARM ステートに移行し、ハンドラは、J と T の両方が SPSR_und でセットされていることから、それが例外の原因であったことを検出します。

Jazelle ステートでは、プロセッサがオペコードプログラムを実行します。オペコードプログラムは、Lindholm と Yellin 著の『*The Java Virtual Machine Specification 2nd Edition*』で定義されているように、1 つ以上のクラスファイルで構成される、または 1 つ以上のクラスファイルから派生し、機能的に同等な実行可能オブジェクトと定義されます。Jazelle ステートでは、PC はプログラムカウンタとして機能し、次に実行される JVM オペコードを特定します。JVM オペコードは、Lindholm と Yellin の著書に定義されているオペコード、または機能的に同等の変形版です。

Lindholm と Yellin の著書に記載されている Jazelle 拡張の*ネイティブメソッド*では、ARM 命令セットまたは Thumb 命令セットのみ、もしくは両方のセットを使って機能を指定する必要があります。

このセクションおよび Lindholm と Yellin の著書に従い、Jazelle 拡張の実装は、オペコードプログラムの高速処理を除いて、Jazelle 状態で何らかのタスクを実行すると説明書に記載したり、宣伝したりすることはできません。

PC の 32 ビットへの拡張

PC が任意のオペコードを指せるようにするため、PC の 32 ビットはすべて、非トリビアル実装で定義されています。PC のビット [0] は、ARM ステートでも Thumb ステートでも常に 0 として読み出されます。ビット [1] は、ARM 命令のワードアライメント、Thumb 命令のハーフワードアライメントを反映します。PC におけるビット [0] の存在が ARM ステートまたは Thumb ステートで可視となるのは、Jazelle ステートで発生する例外の理由のみで、この場合例外からの復帰アドレスは奇数バイトでアラインされています。

このためアーキテクチャでは、例外ハンドラが R15 の 32 ビットすべてを常に復元することを保証する必要があります。例外からの復帰を処理するのに推奨されている方法は、正しく動作します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.10.3 新しい Jazelle ステート開始命令（BXJ）

新たに追加された、BX に類似した ARM 命令です。BXJ 命令には、ターゲットの実行ステート（ARM または Thumb）と、Jazelle ステートを開始できない場合に使用する分岐先アドレスを指定する 1 つのレジスタオペランドがあります。詳細については、P. A4-21 *「BXJ」*を参照して下さい。

Jazelle を使用した Java の実行には、BXJ 命令を使用する EJVM、標準 ARM レジスタの使用モデル、P. A2-62 *「構成と制御」*に記載されている Jazelle 拡張機能制御およびコンフィギュレーションレジスタが使用されます。

Jazelle 拡張機能許可時の BXJ 実行

JE ビットが 1 の場合に BXJ 命令を実行すると、Jazelle 拡張機能ハードウェアが Jazelle ステートを開始し、直接オペコードの実行を開始します。Jazelle ステートの実行が開始する状況は、実装定義です。Jazelle ステートの実行が開始しない場合、命令は、サブアーキテクチャ定義のレジスタ使用モデルに対し、BX 命令と同じ方法で実行されます。これは、Jazelle 拡張機能ハードウェアと EJVM ソフトウェアとで有効に通信を行うために必要となります。同様に、Jazelle ステートの実行が終了し、ARM または Thumb ステートの実行が再開するときは、各種のレジスタにサブアーキテクチャ定義の値が保持されます。そのような条件に影響を受ける具体的なレジスタ群は、プロセスレジスタのサブアーキテクチャ定義のサブセットで、次のように定義されます。

- ARM 汎用レジスタの R0 〜 R14
- PC
- CPSR
- VFP 汎用レジスタである S0 〜 S31 および D0 〜 D15（使用については VFP アーキテクチャの制限を受け、VFP アーキテクチャの存在を前提とします）
- FPSCR（VFP アーキテクチャの存在を前提とします）

Jazelle ステートの実行によって変更される可能性のあるすべてのプロセッサ状態は、プロセッサ例外やプロセススワップの発生時に正しく保存され復元されるよう、プロセスレジスタに保持する必要があります。コンフィギュレーション状態（Jazelle ステートの実行に影響を与えるが、それによって変更されない状態）は、プロセスレジスタまたはコンフィギュレーションレジスタに保持できます。

EJVM の実装は、プロセッサの Jazelle 拡張機能サブアーキテクチャがプロセスレジスタの使用に適合すると判断した後、JE == 1 だけをセットします。適合しない場合は、JE == 0 のままとし、ハードウェアアクセラレーションなしで実行します。

Jazelle 拡張機能禁止時の BXJ 実行

JE ビットが 0 で BXJ 命令が実行される場合は、同じレジスタオペランドの BX 命令と同様に実行されます。

したがって、JE ビットが 0 の場合は、BXJ 命令を自由に実行できます。特に、Jazelle 拡張機能の実装がトリビアルであるプロセッサで実行される、または互換性のないサブアーキテクチャを使用していると EJVM が判断した場合は、JE == 0 をセットし、Jazelle ハードウェアアクセラレーションを利用せずに正しく実行します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

Jazelle ステートの終了

プロセッサは、実装定義の状況で Jazelle ステートを終了します。これは一般に、実装がハードウェアで処理できないオペコード、または Jazelle 例外を生成するオペコード（NULL ポインタ例外など）の実行を試みたときに発生します。この状況が発生した場合、各種のプロセッサレジスタはサブアーキテクチャ定義の値を保持し、EJVM がオペコードプログラムのソフトウェア実行を正しく再開できるようにします。

プロセッサは、プロセッサ例外が発生したときも Jazelle ステートを終了します。CPSR は通常どおり、例外モードのバンク SPSR にコピーされるため、バンク SPSR が J == 1 と T == 0 を保持し、例外からの復帰時には SPSR が CPSR に戻され、Jazelle ステートが復元されます。この条件と、プロセスレジスタだけが Jazelle 状態実行によって変更されるという制限により、プロセッサ例外ハンドラによってすべてのレジスタが正しく保存、復元されることが保証されます。コンフィギュレーションレジスタと制御レジスタは例外ハンドラ自体で変更されることがあります。詳細については、P. A2-62 *「構成と制御」*を参照して下さい。

プロセッサ例外については、オペコード実行に固有の問題を考慮する必要があります。詳細については、P. A2-58 *「Jazelle 拡張機能の例外処理」*を参照して下さい。

Jazelle ステートの実行の間に変更され、Jazelle ステートの実行の間プロセスレジスタ外に保持される状態が、Jazelle 拡張機能ハードウェアに含まれるかどうかは、実装定義です。そのような状態が存在する場合、実装は次の動作を行います。

- BXJ 命令の実行またはプロセッサ例外からの復帰により Jazelle ステートが開始する時は必ず、1 つ以上のプロセスレジスタから状態を初期化します。
- プロセッサ例外の実行または実装定義の状況により Jazelle ステートが終了するときは必ず、1 つ以上のプロセスレジスタに状態を書き込みます。
- これらの状態をプロセッサ例外の実行時にプロセスレジスタに書き込み、例外からの復帰時にプロセスレジスタから初期化する方法を確保し、例外の間、状態が正しく保存され、復元されるようにします。

Jazelle ステートに関するその他の制限

Jazelle 拡張機能ハードウェアは、以下の制限に従う必要があります。

- 標準的な ARM プロセッサ例外のいずれかを実行する場合以外は、プロセッサモードを変更できません。
- 開始したプロセッサモードに属するもの以外のバンクレジスタにはアクセスできません。
- 予測不能な命令に対して不正な動作は実行できません。これは、セキュリティループホールを生成したり、プロセッサや他のシステム部分を停止、ハングさせたりしてはいけないという意味です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

このような条件の結果、OS のセキュリティを侵害することなく、ユーザモードから Jazelle ステートに入れます。また、次のような制限も適用されます。

- FIQ モードから Jazelle ステートを開始した場合、結果は予測不能です。
- Jazelle 拡張機能のサブアーキテクチャおよび実装は、割り当てられないはずの CPSR または SPSR ビットを使用できません。これらのビットはすべて、ARM および Thumb アーキテクチャの将来の拡張用に予約されています。

A2.10.4 Jazelle 拡張機能の例外処理

すべての例外は、J ビットを CPSR から SPSR にコピーします。SPSR から CPSR にコピーすることで他の部分を変更する命令はすべて、その他の全ビットとともに J ビットをコピーする必要があります。

Jazelle ステートで例外が発生すると、例外モードの R14 レジスタが以下のように計算されます。

IRQ/FIQ 割り込みからの復帰時に実行されるオペコードのアドレス＋ 4

プリフェッチアボート

アボートを引き起こしたオペコードのアドレス＋ 4

データアボート

アボートを引き起こしたオペコードのアドレス＋ 8

未定義命令 発生してはいけません。詳細については、P. A2-60「*未定義命令例外*」を参照して下さい。

SWI 発生してはいけません。詳細については、P. A2-60「*SWI 例外*」を参照して下さい。

割り込み（IRQ、FIQ）

割り込みを処理する標準的な機構が正しく動作するよう、Jazelle 例外のハードウェア実装は、Jazelle 状態実行中に割り込みが起こる際に必ず、以下のいずれかが発生するようにする必要があります。

- 実行がオペコード命令境界に到達した。これは、1 つのオペコードを実装するのに必要なすべての演算が完了し、かつ次のオペコードを実装するのに必要な演算が 1 つも完了していないことを指します。割り込みハンドラ開始時の R14 の値は、次のオペコード＋ 4 のアドレスの必要があります。
- 現在のオペコードの実行開始から割り込みが発生できる任意の点までに実行される動作シーケンスは、同じ情報を複数回読み書きしても同じシステム結果を示す。すなわち、オペコード実行の全体的な結果を変更することなく、最初から繰り返すことができる。割り込みハンドラ開始時の R14 の値は、現在のオペコード＋ 4 のアドレスの必要があります。
- オペコードの実行中に割り込みが発生すると、直接 Jazelle 拡張機能ハードウェアにより、または EJVM におけるサブアーキテクチャ定義のハンドラ呼び出しにより、是正措置が行われます。そして、その是正措置が、オペコードを最初から再実行できる状況を再作成します。割り込みハンドラ開始時の R14 の値は、オペコード＋ 4 のアドレスの必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

データアボート

データアボート時に R14_abt に保存される値により、仮想メモリデータアボートハンドラは、システムコプロセッサ (CP15) のフォルトステータスレジスタおよびフォルトアドレスレジスタを読み出し、アボートの理由を修正し、`SUBS PC、R14、#8` または同等の命令を使用して復帰できます。アボートを引き起こした命令や、その命令が実行されていた状態を参照する必要はありません。

——— 注 ———

これは、データアボートを引き起こしたオペコードに復帰し再試行することを意図していると仮定しています。アボートを引き起こしたオペコードの後のオペコードに復帰することを意図している場合、アボートを引き起こしたオペコードの長さだけ、復帰アドレスを変更する必要があります。

データアボートを処理する標準的な機構が正しく動作するには、オペコードによりデータアボートが生成される可能性がある場合に、Jazelle 例外のハードウェア実装が以下のいずれかに該当する必要があります。

- オペコードの実行開始からデータアボートが発生する点まで実行される操作のシーケンスが、同じ情報を複数回読み書きしても同じシステム結果を示すとなること。これは、オペコード実行の全体的な結果を変更することなく、最初から繰り返すことができることを意味します。
- オペコードの実行中にデータアボートが発生すると、直接 Jazelle 拡張機能ハードウェアにより、または EJVM におけるサブアーキテクチャ定義のハンドラ呼び出しにより、是正措置が行われます。そして、その是正措置が、オペコードを最初から再実行できる状況を再作成します。

——— 注 ———

ARMv6 では、ベース更新アボートモデルが許容されなくなりました（P. A2-23 *「アボートモデル」* 参照）。これにより、最初のソリューションに障害が生じる可能性がなくなります。

プリフェッチアボート

プリフェッチアボート時に R14_abt に保存された値により、仮想メモリプリフェッチアボートハンドラは、実行が試みられたときの状態を参照することなく、アボートを引き起こした命令の開始位置を簡単に特定することができます。これは、常にアドレス（R14_abt - 4）にあります。

しかし、マルチバイトオペコードはページ境界を越える場合があり、この場合、ARM プロセッサのプリフェッチアボートハンドラは、2 つのページのどちらがアボートを引き起こしたかを直接判断できません。この状況がどう処理されるかは、サブアーキテクチャ定義です。ARM プロセッサのプリフェッチアボートハンドラを呼び出すことによって処理する場合は、（R14_abt - 4）がそのオペコードの最初のバイトを指す必要があります。

サブアーキテクチャに依存しないようにするため、OS 設計者は、オペコードがまたがる 2 ページのどちらでプリフェッチアボートが発生しても処理できるようにプリフェッチアボートハンドラを作成する必要があります。簡単なテクニックを紹介します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

```
IF the page pointed to by (R14_abt - 4) is not mapped
    THEN map the page
    ELSE map the page following the page including (R14_abt - 4)
ENDIF
retry the instruction
```

SWI 例外

SWI 例外は、以下の理由から、Jazelle 状態の実行中に発生できません。

- ARM アーキテクチャでは、ARM 状態および Thumb 状態の SWI がサポートされています。オペコード SWI は、SWI 使用モデルを複雑にするため、サポートされていません。
- さらに複雑なオペコードを実行するためには、Jazelle 拡張機能サブアーキテクチャと実装が、ARM または Thumb 状態のハンドラに戻る機構を持つ必要があります。オペコードが OS 呼び出しを実行する必要がある場合は、この機構を利用し、OS 呼び出し自体のコストに比べて相対的に少ないオーバヘッドで、ARM または Thumb の `SWI` 命令を実行させることができます。
- SWI 呼び出し規則は OS 依存度が高く、サブアーキテクチャが OS を認識していることが必要な場合もあります。

未定義命令例外

未定義命令例外は、Jazelle 状態の実行中に発生できません。

Jazelle 拡張機能ハードウェアは、コプロセッサ命令を合成し、ハードウェアコプロセッサ（多くの場合は VFP コプロセッサ）に渡します。このとき、コプロセッサが命令を拒否した場合に ARM プロセッサの未定義命令トラップの発生を許可すると、状況が非常に複雑になります。例えば、以下のような事態が考えられます。

- コプロセッサ命令をメモリからロードできなくなります（ほとんどの未定義命令ハンドラは、これをロードできることに依存しています）。
- Jazelle 拡張機能ハードウェアの実装とサブアーキテクチャに関する詳細な知識がなければ、メモリからロード可能なオペコードによりコプロセッサ命令を判断することは通常はできません。
- 一般に、コプロセッサが生成する未定義命令例外（特に、VFP が生成する例外）は、正確（(R14_und - 4）にある命令が原因で発生）または不正確（以前の命令が生成した保留中の例外条件が原因で発生し、(R14_und - 4）にある命令とは無関係）です。

  正確な未定義命令例外は、一般に（R14_und - 4）にある命令のエミュレートと、その次の命令への復帰によって処理する必要があります。不正確な未定義命令例外は、一般にコプロセッサから例外条件、以前の命令、もしくは両方の詳細を取得し、何らかの方法で修正した後、(R14_und - 4）にある命令に復帰することによって処理される必要があります。

  すなわち、復帰アドレスには 2 種類あり、メモリ内のコプロセッサ命令を処理するときのように、必ずしも互いからの固定オフセットに基づくわけではありません。このため、未定義命令ハンドラの開始時の R14_und の値を定義するのが困難です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- 未定義命令ハンドラの復帰アドレスは、Jazelle 拡張機能ハードウェアが実行する操作シーケンスについて、冪等、完了、もしくは両方の条件を課します。つまり、コプロセッサ演算が完了すれば、オペコードの実行に必要なすべてが終了したことになります。

  この制限により、Jazelle アクセラレーションとコプロセッサの両方の設計者の協力が必要となり、設計の自由が制限されます。

未定義例外の必要性をなくすため、以下のような Jazelle 拡張機能ハードウェアのコプロセッサインタワーキングモデルが適用されます。

コプロセッサインタワーキング

Jazelle 状態で実行中、Jazelle 拡張機能ハードウェアがコプロセッサ命令を合成し、ハードウェアコプロセッサに渡して実行させる場合、Jazelle 拡張機能ハードウェアは、コプロセッサが命令を拒否した場合の準備をしておく必要があります。コプロセッサが Jazelle 拡張機能ハードウェアの発行した命令を拒否した場合、Jazelle 拡張機能ハードウェアとコプロセッサは協調して以下の動作を実行する必要があります。

- コプロセッサが ARM 状態のコプロセッサ命令の発生を拒否した場合に起こる未定義命令例外を防止する。
- サブアーキテクチャ定義の適切な是正措置を行う。多くの場合は Jazelle 状態を終了し、他のコプロセッサ命令を含む適切な ARM コードハンドラを実行することになります。

これを実用的なテクニックとし、コプロセッサによる命令の拒否を不十分あるいは過度に処理しないようにするため、Jazelle 拡張機能で使用するように設計されているコプロセッサは、以下の動作を行う必要があります。

- 以前の命令が生成した例外条件がある場合、コプロセッサは、未定義命令ハンドラによって例外条件がクリアされるまで、その例外条件に常に注意を払い、コプロセッサ命令の実行が試みられるたびに、不正確な未定義命令例外の発生を試みます。
- 現在実行を要求されているコプロセッサ命令に関する理由で正確な未定義命令例外の発生を試みる場合、コプロセッサはメモリを使わない方法で動作する必要があります。つまり、異なるコプロセッサ命令を続けて実行するよう要求された場合、最初に正確な拒否を試みた命令を無視し、新しい命令を正確に拒否する必要があるかどうかを判断します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.10.5 構成と制御

Jazelle 拡張機能に関連するすべてのレジスタは、コプロセッサ 14（CP14）の一部としてコプロセッサ空間に実装されます。これらのレジスタは、MCR（P. A4-63「*MCR*」）命令と MRC（P. A4-71「*MRC*」）命令を使用してアクセスされます。

Jazelle 拡張機能の制御と構成に関する一般的な命令形式は、次のとおりです。

```
MCR{<cond>} p14, 7, <Rd>, CRn, CRm{,  opcode_2}*
MRC{<cond>} p14, 7, <Rd>, CRn, CRm{,  opcode_2}*
```

*opcode_2 == 0 の場合、opcode_2 は省略できます。

Jazelle 拡張機能の制御レジスタとコンフィギュレーションレジスタには、以下の規則が適用されます。

- サブアーキテクチャ定義のすべてのコンフィギュレーションレジスタは、<opcode_1> を 7 にセットした状態で、コプロセッサ 14 の MRC 命令と MCR 命令によってアクセスされます。
- コンフィギュレーションレジスタが保持する値は、MCR 命令の実行によってのみ変更されます。Jazelle 状態のオペコード実行によっては変更されません。
- 必要なレジスタに関するアクセス方針は、レジスタの説明に詳しく定義されています。Jazelle ID レジスタに対するすべての MCR アクセス、および特権モードだけに限定されている MRC または MCR アクセスは、ユーザモードで実行される場合、未定義です。

  その他のコンフィギュレーションレジスタに関するアクセス方針は、サブアーキテクチャ定義です。
- コンフィギュレーションレジスタが読み出し可能の場合、読み出しの結果は、最後に書き込まれた値となり、他の部分に影響はありません。コンフィギュレーションレジスタが読み出し不能の場合、読み出そうとする試みの結果は予測不能です。
- コンフィギュレーションレジスタに書き込める場合、その結果は同じ情報を複数回読み書きしても同じシステム結果を示す必要があります。つまり、値を 2 回以上書き込んだときの結果は、1 回書き込んだときの結果と同じの必要があります。

非トリビアル実装には、最低 3 つのレジスタが必要です。サブアーキテクチャ定義のレジスタが追加されることもあります。

Jazelle ID レジスタ

Jazelle ID レジスタにより、EJVM は、EJVM が動作しているアーキテクチャとサブアーキテクチャを判断できます。これは、MRC 命令によってアクセスされる、コプロセッサ 14 の読み出し専用レジスタです。

```
MRC{<cond>}  p14, 7, <Rd>, c0, c0 {, 0}   ;<Rd>:= Jazelle Identity register
```

Jazelle ID レジスタは通常、特権モードからもユーザモードからもアクセス可能です。ユーザモードでのアクセス制限については、P. A2-64「*オペレーティングシステム（OS）制御レジスタ*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

Jazelle ID レジスタの形式は次のとおりです。

31 28	27 20	19 12	11 0
アーキテクチャ	実装者	サブアーキテクチャ	サブアーキテクチャ定義

ビット **[31:28]** アーキテクチャコードを保持します。これは、コプロセッサ 15 のメイン ID レジスタにあるアーキテクチャコードと同じです。

ビット **[27:20]** サブアーキテクチャ設計者の実装者コードを保持します。これは、コプロセッサ 15 のメイン ID レジスタにある実装者コードと同じです。詳細については、P. B3-7 *「メイン ID レジスタ」*を参照して下さい。

特例として、Jazelle 拡張機能のトリビアル実装が使用されている場合、この実装者コードは 0x00 となります。

ビット **[19:12]** サブアーキテクチャコードを保持します。以下のサブアーキテクチャコードが定義されています。

0x00 = Jazelle V1 サブアーキテクチャ、または実装者コードが 0x00 の場合は Jazelle 拡張機能のトリビアル実装。

ビット **[11:0]** 詳しいサブアーキテクチャ定義の情報を保持します。

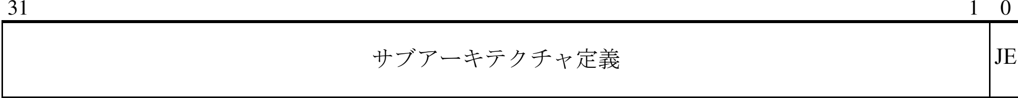
メインコンフィギュレーションレジスタ

Jazelle 拡張機能を制御するために、メインコンフィギュレーションレジスタが追加されています。これは、MRC 命令と MCR 命令によってアクセスされるコプロセッサ 14 のレジスタです。

```
MRC{<cond>}  p14, 7, <Rd>, c2, c0 {, 0}     ; <Rd> := Main Configuration
                                            ; register
MCR{<cond>}  p14, 7, <Rd>, c2, c0 {, 0}     ; Main Configuration
                                            ; register := <Rd>
```

このレジスタは通常、ユーザモードからは書き込み専用です。ユーザモードでの他のアクセス制限については、P. A2-64 *「オペレーティングシステム（OS）制御レジスタ」*を参照して下さい。

メインコンフィギュレーションレジスタの形式は、次のとおりです。



ビット **[31:1]** サブアーキテクチャ定義の情報。

ビット **[0]** Jazelle イネーブル（JE）ビットで、リセット時に 0 にクリアされはす。

JE ビットが 0 の場合、Jazelle 拡張機能は禁止され、BXJ 命令を実行しても Jazelle 状態は開始されず、BX 命令と完全に同じに動作します。詳細については、P. A4-21 *「BXJ」*を参照して下さい。

JE ビットが 1 の場合、Jazelle 拡張機能は許可されています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

オペレーティングシステム（OS）制御レジスタ

Jazelle OS 制御レジスタは、オペレーティングシステムが Jazelle 拡張機能のプロセス使用を制御するためのものです。これは、MRC 命令と MCR 命令によってアクセスされる、コプロセッサ 14 のレジスタです。

```
MRC{<cond>}  p14, 7, <Rd>, c1, c0 {, 0}      ; <Rd> := Jazelle OS
                                              ; Control register
MCR{<cond>}  p14, 7, <Rd>, c1, c0 {, 0}      ; Jazelle OS Control
                                              ; register := <Rd>
```

このレジスタは、特権モードからのみアクセス可能です。上記の命令がユーザモードで実行された場合は、未定義です。EJVM は通常、Jazelle OS 制御レジスタにアクセスしません。ユーザモードで動作するよう意図されている EJVM は、アクセスできません。

Jazelle OS 制御レジスタの主な目的は、OS がサブアーキテクチャに依存しない形で Jazelle 拡張機能ハードウェアへのアクセスを制御できるようにすることです。OS 制御レジスタは、メインコンフィギュレーションレジスタの JE ビットとの併用を前提としています。

Jazelle OS 制御レジスタの形式は、次のとおりです。

31 ... 2	1	0
予約（常に 0）	CV	CD

ビット [31:2] 将来の拡張用に予約されています。拡張が行われるまでの間は、0 が読み出される必要があります。将来の互換性を最大限に確保するため、ソフトウェアは、リード・モディファイ・ライトの手法を使用して他の制御ビットを更新し、予約ビットの内容を保存する必要があります。

CV ビット [1] コンフィギュレーション有効（CV）ビットは、オペレーティングシステムが EJVM に、構成をコンフィギュレーションレジスタに再書き込みする必要があることを通知するのに使えます。CV == 0 の場合、次にオペコードを実行する前に、コンフィギュレーションレジスタの再書き込みが必要です。CV == 1 の場合、次にオペコードが実行される前に起きる事が確実な再書き込みの他は、コンフィギュレーションレジスタに再書き込みを行う必要ありません。

CD ビット [0] コンフィギュレーションディセーブルビットは、オペレーティングシステムがコンフィギュレーションレジスタおよび Jazelle ID レジスタへのユーザモードアクセスの監視、制御、もしくは両方を行うために使用されます。CD == 0 の場合、コンフィギュレーションレジスタに書き込む MCR 命令と、Jazelle ID レジスタを読み出す MRC 命令が、正常に実行されます。CD == 1 の場合、これらの命令はすべて、特権モードで実行されたときのみ正常に動作します。ユーザモードで実行された場合は未定義です。

メインコンフィギュレーションレジスタの JE ビットが 0 の場合、Jazelle OS 制御レジスタは、BXJ 命令の実行に影響を与えません。BXJ 命令は、常に BX 命令として実行されます。

メインコンフィギュレーションレジスタの JE ビットが 1 の場合、CV ビットは BXJ 命令に以下の影響を与えます。

- CV == 1 の場合、Jazelle 拡張機能ハードウェアのコンフィギュレーションは許可され、有効であると見なされ、プロセッサは Jazelle 状態を開始し、P. A2-56 *「Jazelle 拡張機能許可時の BXJ 実行」* に示されているとおりにオペコードを実行できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- CV == 0 の場合、もし CV == 1 だったらであれば JAZELLE 拡張機能ハードウェアが JAZELLE 状態に入っただろうと思われる実装定義の状況すべてにおいて、コンフィギュレーション無効ハンドラが開始され、CV に 1 がセットされます。コンフィギュレーション無効ハンドラは、MCR 命令を含む ARM 命令のシーケンスであり、EJVM で必要とされる構成を書き込みます。シーケンスの最後は、オペコードを再試行する BXJ 命令です。コンフィギュレーション無効ハンドラのアドレスが決定される方法、開始条件、終了条件はすべてサブアーキテクチャ定義です。

  もし CV == 1 であったら Jazelle 拡張機能ハードウェアは Jazelle 状態に入らなかっただろうと思われる状況では、コンフィギュレーション無効ハンドラが前述のように開始するか、それとも BXJ 命令が BX 命令として扱われるか（この場合、サブアーキテクチャ定義の制限が付けられることがあります）は実装定義です。

CV ビットの意図された用法では、プロセススワップが発生した場合、CV はオペレーティングシステムにより 0 にセットされます。このため、新しいプロセスは、Jazelle 拡張機能ハードウェアでオペコードを実行する前に、コンフィギュレーション無効ハンドラを実行する必要があります。これにより、Jazelle 拡張機能ハードウェアのコンフィギュレーションレジスタが、EJVM に対して常に正常に動作します。CV ビットは、コンフィギュレーション無効ハンドラの開始時に 1 にセットされるため、無効コンフィギュレーションハンドラが実行を再試行する際に、オペコードがハードウェアで実行されます。

——— 注 ———

CV ビットが、コンフィギュレーションレジスタの書き込み完了後ではなく、コンフィギュレーション無効ハンドラの開始時に 1 にセットされるのは、自然でないように見えますが、この方式を使用しない場合、プロセススワップが発生する前に、コンフィギュレーション無効ハンドラがハードウェアを部分的に設定し、EJVM を使用する別のプロセスがハードウェアにコンフィギュレーションを書き込む可能性があります。

元のプロセスが再開するとき、CV はオペレーティングシステムによってクリアされます（CV == 0）。この例で、ハンドラが構成をハードウェアに書き込んだ後、CV を 1 にセットすると、ハードウェアが 2 つの構成のハイブリッドに設定された状態で、オペコードが実行されます。

コンフィギュレーション無効ハンドラの開始時に CV を 1 にセットすることにより、オペコード実行が再試行されるときには CV == 0 で、プロセススワップの発生なしに実行を完了するまで、コンフィギュレーション無効ハンドラが繰り返し実行されます（必要であれば再帰的に）。

———————————

CD ビットには、Jazelle 拡張機能ハードウェアへのユーザモードアクセスの監視および制御を行うための複数の用途があります。以下に例を挙げます。

- CD == 1、JE == 0 にセットすると、OS は、Jazelle 拡張機能ハードウェアへのすべてのユーザモードアクセスを防ぐことができます。つまり、BXJ 命令の使用を試みた場合、常に BX 命令と同じ結果となり、ハードウェアの構成を試みると（JE ビットのセットなど）未定義命令例外が発生します。

- 他のプロセスによるハードウェアの矛盾する使用から EJVM を保護しつつ、ユーザモードから Jazelle 拡張機能ハードウェアに簡単にアクセスできるようにするため、OS は CD == 0 をセットし、プロセススワップの際にメインコンフィギュレーションレジスタの保存と復元を行い、新しいプロセス用にこのレジスタの値を 0 に初期化する必要があります。また、プロセススワップのたびに CV ビットを 0 にセットし、EJVM が必要に応じて Jazelle 拡張機能ハードウェアを再設定するようにする必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- 上記の手法では、Jazelle 拡張機能ハードウェアを使用するプロセスが非常に少ない場合、そのハードウェアの不要な再構成が大量に実行される場合があります。これは、EJVM を使用するユーザモードプロセスを OS が把握することによって改善できます。

  OS は、新しいプロセスの作成時、もしくは EJVM を使用しない既存のプロセスへのコンテキストスイッチ時に、CD == 1、JE == 0 をセットする必要があります。コンフィギュレーションレジスタにアクセスを試みるユーザモード命令は、すべて未定義例外を発生します。この場合、未定義命令ハンドラは EJVM の必要を識別し、プロセスが EJVM を使用することをマークした上で、復帰してから CD == 0 で命令を再試行します。

  さらに改善を行う方法としては、コンテキストの切り替え先のプロセス EJVM を使用し、かつそのプロセスが、最後に EJVM を使用していたプロセスと異なる場合にのみ、CV ビットを 0 にクリアします。これによって、ハードウェアの無駄な再構成が防止できます。つまり、オペレーティングシステムが「現在 Jazelle 拡張機能ハードウェアを所有しているプロセス」変数を保持し、それが EJVM を使用するプロセスへの切り替え時に process_ID で更新されます。process_ID が更新された結果、保存されている変数が変更される場合、コンテキストスイッチソフトウェアは CV を 0 にセットします。

  CV ビット方式を実装するコンテキストスイッチソフトウェアは、プロセススワップで EJVM を使用するプロセスが変更される場合、メインコンフィギュレーションレジスタ（全体）の保存と復元も行う必要があります。これにより、復元された EJVM が、独自の目的で JE ビットを確実に使用できるようになります。

  ——— 注 ———

  この手法は、EJVM を使用する特権プロセスを識別しません。しかし、オペレーティングシステムは、特権プロセスの要求を認識していることが前提です。

  ———

- OS は、構成をハードウェアに書き込み、CD == 1、JE == 1 にセットすることにより、すべてのユーザモードコードに対して単一の Jazelle 拡張機能の構成を適用することができます。

CV ビットと CD ビットは、両方ともリセット時に 0 にセットされます。これにより、いくつか条件を満たしていれば、Jazelle 拡張機能をサポートしていない OS でも EJVM が正常に動作できるようになります。主な条件は、コンフィギュレーションレジスタについて異なる設定を必要とする、EJVM を使用する 2 つのプロセスの間で、プロセススワップが一切発生しないことです。これは例えば、以下の 2 つのいずれかにおいて発生します。

- システム内に EJVM を使用するプロセスが 1 つしかない場合
- システム内の EJVM を使用するプロセスすべてについて、コンフィギュレーションレジスタのスタティック設定が同一の場合

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.10.6 EJVM の動作

このセクションでは、アーキテクチャの要求を満たすには、EJVM がどのように動作すべきかを説明します。

初期化

EJVM は初期化時に、最初に Jazelle ID レジスタから読み出した値に含まれる実装者コードとサブアーキテクチャコードを使用して、どのサブアーキテクチャが存在しているかを確認します。

EJVM がサブアーキテクチャと互換でない場合は、JE == 0 の値をメインコンフィギュレーションレジスタに書き込むか、（オペコードをアクセラレーションなしで実行することが許容されない場合）エラーを生成します。

EJVM がサブアーキテクチャと互換の場合は、要求される構成をメインコンフィギュレーションレジスタおよび他のコンフィギュレーションレジスタに書き込みます。EJVM は、Jazelle OS 制御レジスタの CV ビットが 0 となることを想定して、この手順を省略することはできません。Jazelle 拡張機能ハードウェアがオペコードを実行する前に CV == 0 がコンフィギュレーション無効ハンドラをトリガすることを前提とはできません。

オペコードの実行

EJVM は、意図されたサブアーキテクチャが指定するオペコードと例外条件のそれぞれに対応するハンドラを含む必要があります（例外条件は常にコンフィギュレーション無効を含みます）。EJVM は BXJ 命令を実行し、オペコードの実行を初期化します。この BXJ 命令のレジスタオペランドにはプログラムの最初にあるオペコード用のオペコードハンドラのターゲットアドレスを指定し、プロセスレジスタはサブアーキテクチャ定義のレジスタ使用モデルに対応して設定します。

オペコードハンドラは、オペコードが要求するデータ処理操作を実行し、次に実行するオペコードのアドレスを判断し、そのオペコードに対応するハンドラのアドレスを判断し、BXJ 命令をそのハンドラアドレスに対して実行します。この BXJ 命令も同様に、サブアーキテクチャ定義のレジスタ使用モデルに対応してレジスタを設定した状態で実行します。

例外条件ハンドラの開始時におけるレジスタ使用モデルは、サブアーキテクチャ定義であり、BXJ 命令の実行用に定義されたレジスタ使用モデルとは異なる場合があります。その場合、ハンドラは例外条件を解決します。たとえば、コンフィギュレーション無効ハンドラの場合、ハンドラは要求する構成をメインコンフィギュレーションレジスタおよび他のコンフィギュレーションレジスタに再度書き込みます。

その他の考慮事項

アプリケーションを実行し、オペレーティングシステムと正しく通信するために、EJVM はユーザモードで許容された動作のみを実行する必要があります。具体的には、Jazelle ID レジスタの読み出しとコンフィギュレーションレジスタへの書き込みのみを実行します。Jazelle OS 制御レジスタへのアクセスを試みることはできません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.10.7 トリビアル実装

このセクションでは、Jazelle 拡張機能のトリビアル実装で何を実装する必要があるかについて説明します。

- 実装者フィールドとサブアーキテクチャフィールドを 0 にセットし、Jazelle ID レジスタを実装します。レジスタ全体が 0 として読み出されてもかまいません。
- 0 が読み出され、書き込みを無視するようメインコンフィギュレーションレジスタを実装します。
- 読み出しと書き込みが可能だが、結果が無視されるよう Jazelle OS 制御レジスタを実装します。このレジスタは、RAZ（常に 0）/ DNM（書き込み時の修正不可）として実装することもできます。これにより、EJVM をサポートするオペレーティングシステムが正しく実行されます。
- JE ビットが常に 0 である以上、どのような状況でも `BXJ` 命令は `BX` 命令とまったく同じに動作するよう実装します。これは、トリビアル実装では通常、Jazelle 状態が開始しないことを意味します。
- ARMv6 の場合、トリビアル実装は、CPSR/SPSR の J ビットを RAZ（常に 0）/ DNM（書き込み時の修正不可）として実装できます。これが許容されるのは、J ビットをセットし、Jazelle 状態を開始する正当な方法がないため、この動作を試みる復帰ルーチンはすべて予測不能な命令を発行するためです。

  他の方法として、J ビットを CPSR と各 SPSR に実装し、例外が開始したとき、および `MSR` 命令、`MRS` 命令、例外からの復帰が実行されたときに、これらの J ビットの読み出し、書き込み、コピーが正しく行われるようにします。
- どのような場合でも、CPSR で J == 1 の場合、次の命令がフェッチされ、プリフェッチアボートが発生するか、もしくは未定義と見なされるかは、実装定義です。

———— 注 ————

トリビアル実装の場合、PC が 32 ビットに拡張される必要はありません。これは、PC のビット [0] は、Jazelle 状態の実行中にプロセッサ例外が発生しない限り、ARM 状態または Thumb 状態から不可視であり、トリビアル実装では Jazelle 状態の実行が行われないためです。

————————————

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A2.11 飽和整数算術演算

汎用レジスタの内容を符号付きの数値と見なす場合、その値は -2^{31}（または `0x80000000`）～ $+2^{31}-1$（または `0x7FFFFFFF`）の範囲にあります。これらの数値に対して加算または減算が実行され、数学的に正しい結果がその範囲外にある場合は、それを表すのに 32 ビットより多くが必要となります。このような場合、余剰なビットは通常は破棄され、得られる結果は、数学的に正しい結果のモジュロ 2^{32} に等しくなります。

例えば、$+3 \times 2^{29}$ を符号付き整数として表現すると `0x60000000` になります。これを 2 つ足すと、$+3 \times 2^{30}$ となります。これは表現可能な範囲外にありますが、33 ビットの符号付き整数 `0x0C0000000` として表現できます。実際に得られる結果は、右側の 32 ビット、つまり `0xC0000000` です。これは、-2^{30} を表します。これは、数学的に正しい結果より 2^{32} 小さく、正しい結果とは符号も異なります。

この種の不正確さは、多くの DSP アプリケーションにおいて許されません。例えば、音声信号を処理中にこれが発生すると、符号の突然の変更によって「カチッ」という大きな音が発生します。このような影響を防ぐため、多くの DSP アルゴリズムは、*飽和*符号付き算術演算を使用します。これは、通常の整数算術演算の動作とは次のような点で異なっています。

- 数学的に正しい結果が、-2^{31} ～ $+2^{31}-1$ の範囲内にある場合、演算の結果は、数学的に正しい結果と同じです。
- 数学的に正しい結果が $+2^{31}-1$ より大きく、表現可能な範囲の上限を超える場合、演算の結果は $+2^{31}-1$ に等しくなります。
- 数学的に正しい結果が -2^{31} より小さく、表現可能な範囲の下限を超える場合、演算の結果は -2^{31} に等しくなります。

言い換えれば、飽和算術演算の結果は、表現可能な数で、数学的に正しい結果に最も近い数ということになります。

飽和符号付き 32 ビット整数加算および減算をサポートする命令（プリフィックス Q）には、`QADD` 命令と `QSUB` 命令があります。これらの命令（`QDADD`、`QDSUB`）のバリエーションは、飽和加算または減算の前に、オペランドのうち 1 つの飽和倍演算を実行します。

幅 A ビットと B ビットの 2 つの値の積が（A + B）ビットのデスティネーションをオーバフローすることは決して発生しないため、飽和整数乗算はサポートされていません。

A2.11.1 飽和 Q15、Q31 算術演算

32 ビットの符号付き値は、符号ビットの直後に 2 進小数点を持つものとして扱うことができます。これは、符号付き整数値を 2^{31} で割るのと同じなため、この場合は -1 ～ $+1-2^{-31}$ の数を表現できることになります。この方法で、32 ビットの値を使って小数を表現する場合、その値は *Q31 数*と呼ばれます。

飽和加算、減算、倍演算は、飽和整数算術演算と同じ命令を使用して、Q31 数に対して実行ができます。この場合、関係するすべての数値が 2^{-31} 倍だけ小さくなったと考えることができます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

同様に、16 ビット数は、符号ビットの直後に 2 進小数点を持つものとして扱うことができます。これは事実上、符号付き整数値を 2^{15} で割ることになります。この方法で 16 ビット値を使用する場合、-1 〜 $+1 - 2^{-15}$ の数を表現可能となり、この数は *Q15 数*と呼ばれます。

2 つの Q15 数が整数として掛け合わされると、結果の整数は、$2^{-15} \times 2^{-15} == 2^{-30}$ 倍だけ小さくする必要があります。例えば、2 つの Q15 数 `0x8000`（-1 を表現）を掛け合わせると、`0x40000000` となります。この値は、求める結果である +1 の 2^{30} 倍です。

つまり、整数乗算命令の結果は、Q31 形式ではないことになります。Q31 形式にするには、2 倍する必要があるため、必要な倍率は 2^{-31} となります。さらに、倍演算が整数をオーバフローさせることがあるため、結果は飽和付きで 2 倍する必要があります。例えば、2 つの `0x8000` を掛け合わせた結果の `0x40000000` は、飽和付きで 2 倍し、`0x7FFFFFFF` とする必要があります（$-1 \times -1 == +1$ という数学的に正しい結果に最も近い Q31 数）。飽和なしで 2 倍すると、結果が `0x80000000` となり、-1 という Q31 表現になります。

したがって、飽和 $Q15 \times Q15 \rightarrow Q31$ 乗算を実装するには、整数乗算命令の次に飽和整数倍演算を置く必要があります。後者は、乗算結果をそれ自体に加算する QADD 命令によって実行できます。

同様に、飽和 $Q15 \times Q15 + Q31 \rightarrow Q31$ 積和演算は、整数乗算命令と QDADD 命令を使用して実行できます。

Q15 数と Q31 数の算術演算に関する他の例は、各命令の用法を説明したセクションにあります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 A3 章
ARM 命令セット

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

- *命令セットのエンコード* : P. A3-2
- *条件フィールド* : P. A3-3
- *分岐命令* : P. A3-5
- *データ処理命令* : P. A3-7
- *乗算命令* : P. A3-10
- *並列加算命令と並列減算命令* : P. A3-14
- *拡張命令* : P. A3-16
- *その他の算術演算命令* : P. A3-17
- *その他の命令* : P. A3-18
- *ステータスレジスタアクセス命令* : P. A3-19
- *ロード / ストア命令* : P. A3-21
- *複数ロード / ストア命令* : P. A3-26
- *セマフォ命令* : P. A3-28
- *例外生成命令* : P. A3-29
- *コプロセッサ命令* : P. A3-30
- *命令セットの拡張* : P. A3-32

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.1 命令セットのエンコード

図 A3-1 は、ARM 命令セットのエンコードを示しています。

他のビットパターンはすべて、予測不能または未定義です。命令が未定義のケースの説明については、P. A3-32 *「命令セットの拡張」*を参照して下さい。

角括弧内の項目（[1] など）は、図の後に詳しい情報があります。

	31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
データ処理イミディエートシフト	cond [1]	0	0	0	opcode				S	Rn	Rd	シフト量		シフト	0	Rm
その他の命令 図A3-4参照	cond [1]	0	0	0	1	0	x	x	0	x x x x	x x x x	x x x x	x	x x	0	x x x x
データ処理レジスタシフト[2]	cond [1]	0	0	0	opcode				S	Rn	Rd	Rs	0	シフト	1	Rm
その他の命令 図A3-4参照	cond [1]	0	0	0	1	0	x	x	0	x x x x	x x x x	x x x x	0	x x	1	x x x x
乗算 ： 図A3-3参照 拡張ロード / ストア ： 図A3-5参照	cond [1]	0	0	0	x	x	x	x	x	x x x x	x x x x	x x x x	1	x x	1	x x x x
イミディエートデータ処理[2]	cond [1]	0	0	1	opcode				S	Rn	Rd	ローテート	イミディエート			
未定義命令	cond [1]	0	0	1	1	0	x	0	0	x x x x	x x x x	x x x x	x	x x	x	x x x x
イミディエートをステータスレジスタに移動	cond [1]	0	0	1	1	0	R	1	0	マスク	SBO	ローテート	イミディエート			
イミディエートオフセットでのロード/ストア	cond [1]	0	1	0	P	U	B	W	L	Rn	Rd	イミディエート				
レジスタオフセットでのロード/ストア	cond [1]	0	1	1	P	U	B	W	L	Rn	Rd	シフト量		シフト	0	Rm
メディア命令 図A3-2参照	cond [1]	0	1	1	x	x	x	x	x	x x x x	x x x x	x x x x	x	x x	1	x x x x
アーキテクチャ上未定義	cond [1]	0	1	1	1	1	1	1	1	x x x x	x x x x	x x x x	1	1 1	1	x x x x
複数ロード/ストア	cond [1]	1	0	0	P	U	S	W	L	Rn	レジスタリスト					
分岐およびリンク付き分岐	cond [1]	1	0	1	L	24ビットオフセット										
コプロセッサロード/ストアおよび2つのレジスタの転送	cond [3]	1	1	0	P	U	N	W	L	Rn	CRd	cp_num	8ビットオフセット			
コプロセッサデータ処理	cond [3]	1	1	1	0	opcode1				CRn	CRd	cp_num	opcode2		0	CRm
コプロセッサレジスタ転送	cond [3]	1	1	1	0	opcode1			L	CRn	Rd	cp_num	opcode2		1	CRm
ソフトウェア割り込み	cond [1]	1	1	1	1	SWI番号										
無条件命令 図A3-6参照	1 1 1 1	x	x	x	x	x	x	x	x	x x x x	x x x x	x x x x	x	x x	x	x x x x

図 A3-1 ARM 命令セットの概要

1. この行では、cond フィールドに 1111 を使用することはできません。他の行は、命令のビット [31:28] が 1111 であるケースに対応しています。
2. opcode フィールドが 10xx の形式で、かつ S フィールドが 0 の場合は、以降の行のいずれかが代用されます。
3. cond フィールドが 1111 の場合、ARMv5 以前ではこの命令は予測不能です。
4. アーキテクチャ上未定義命令では、これらの命令のエンコードの一部が使用されます。

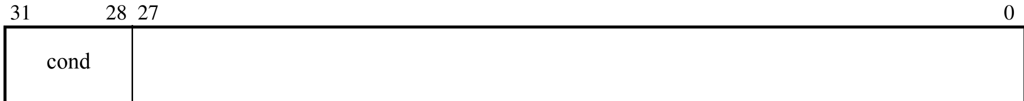
 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.2 条件フィールド

ほとんどの ARM 命令は*条件付きで実行*可能です。これは、CPSR の N、Z、C および V フラグが命令で指定された条件を満たしている場合のみ、その命令はプログラマモデルの状態、メモリ、コプロセッサに対して作用するということです。フラグが条件を満たしていない場合、その命令は NOP として動作します。つまり、命令は正常に実行されたものとして次に進み、割り込みに関連するチェックやプリフェッチアボートが行われますが、それ以外の動作は行われません。

ARMv5 以前は、すべての ARM 命令が条件付きで実行可能でした。新しいバージョンでは、無条件での実行のみが可能ないくつかの命令が追加されています。詳細については、P. A3-41 *「無条件命令拡張空間」*を参照して下さい。

すべての命令で、ビット 31 から 28 には 4 ビットの条件コードが含まれています。



このフィールドには、P. A3-4 表 A3-1 にある 16 個の値のいずれかが含まれます。大部分の命令ニーモニックは、ニーモニック拡張フィールドで定義された文字を使用して拡張可能です。

常時（AL）条件が指定されている場合、命令は条件コードフラグの値にかかわらず実行されます。命令ニーモニックに条件コードがない場合は、AL 条件コードであることを意味しています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.2.1 条件コード 0b1111

条件フィールドが 0b1111 の場合、その動作はアーキテクチャのバージョンによって決まります。

- ARMv4 では、条件フィールドが 0b1111 の命令はいずれも予測不能です。
- ARMv5 以降では、0b1111 の条件フィールドは、無条件でのみ実行可能な各種の追加命令のエンコードに使用されます（P. A3-41*「無条件命令拡張空間」*参照）。ビット [31:28] を cond として示すすべての命令のエンコードダイアグラムは、これらのビットが 0b1111 でない命令とのみ一致します。

表 A3-1 条件コード

オペコード [31:28]	ニーモニック拡張	意味	条件フラグの状態
0000	EQ	=	Z セット
0001	NE	≠	Z クリア
0010	CS/HS	キャリーセット / 符号なし≧	C セット
0011	CC/LO	キャリークリア / 符号なし＜	C クリア
0100	MI	マイナス / 負	N セット
0101	PL	プラス / 正またはゼロ	N クリア
0110	VS	オーバフロー	V セット
0111	VC	オーバフローなし	V クリア
1000	HI	符号なし≧	C セットおよび Z クリア
1001	LS	符号なし≦	C クリアまたは Z セット
1010	GE	符号付き≧	N セットおよび V セット、または N クリアおよび V クリア（N == V）
1011	LT	符号付き＜	N セットおよび V クリア、または N クリアおよび V セット（N != V）
1100	GT	符号付き＞	Z クリアで、N セットおよび V セットか、N クリアおよび V クリアのいずれか（Z == 0、N == V）
1101	LE	符号付き≦	Z セット、または N セットおよび V クリア、または N クリアおよび V セット（Z == 1 または N != V）
1110	AL	常時（無条件）	-
1111	-	*条件コード 0b1111* 参照	-

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.3 分岐命令

ARM プロセッサはすべて、最大 32MB の前方または後方への条件分岐が可能な分岐命令をサポートしています。PC が汎用レジスタ（R15）の 1 つであるため、R15 に値を書き込むことによって分岐またはジャンプを生成することも可能です。

サブルーチンコールは、標準的な分岐命令のバリエーションのひとつで実行できます。最大 32MB の前方または後方への分岐が可能なほか、リンク付き分岐（`BL`）命令によって、分岐後の命令のアドレス（復帰アドレス）が LR（R14）に保存されます。

ARMv4 以降の T バリエーションでは、分岐と状態遷移（`BX`）命令によって、汎用レジスタ Rm の内容が PC にコピーされる（`MOV PC,Rm` 命令と同様）とともに、転送された値のビット [0] が 1 の場合、プロセッサが Thumb® 状態に移行します。対応する Thumb 命令とともに使用することによって、ARM コードと Thumb のコードの間の*インタワーキング*分岐も可能です。

インタワーキングサブルーチンコールは、`BX` と、直前の `MOV LR,PC` 命令などの、適切な復帰アドレスを LR に書き込む命令を結合させることによって生成できます。

ARMv5 以降には、2 種類のリンク付き分岐と状態遷移（`BLX`）命令も追加されています。

- 1 つ目のタイプは、`BX` 命令と同様にレジスタオペランド Rm を使用します。この命令は `BX` 命令と同様に動作しますが、同時に次の命令のアドレスを LR に書き込みます。この命令を使用すると、`MOV LR,PC` と `BX Rm` のシーケンスよりも効率よくインタワーキングサブルーチンコールを実現できます。
- もう 1 つのタイプは、`BL` 命令と同様に、最大 32MB まで前方または後方に分岐し、LR に復帰リンクを書き込みます。ただし、`BL` とは異なり、ARM 状態から Thumb 状態への移行も行います。この命令を使用すると、コール先が Thumb サブルーチンであることが判明しており、かつサブルーチンが 32MB の範囲内にある場合、Rm にサブルーチンのアドレスをロードして `BLX Rm` 命令を実行するよりも効率的に Thumb サブルーチンを呼び出すことができます。

ロード命令を使用すると、4GB のアドレス空間の任意の場所に分岐できます（*ロング分岐*と呼ばれます）。32 ビット値がメモリから PC に直接ロードされ、分岐が行われます。ロング分岐の前に `MOV LR,PC` や、LR に書き込む他の命令を置くと、ロングサブルーチンコールを生成できます。ARMv5 以降では、`BX` 命令によって PC に移動される値のビット [0] と同様に、ロング分岐によってロードされる値のビット [0] により、サブルーチンが ARM 状態と Thumb 状態のどちらで実行されるかが決定されます。ARMv5 以前では、PC にロードされた値のビット [1:0] は無視され、PC へのロードは、ARM 状態でサブルーチンをコールするためにのみ使用されます。

ARMv5 の非 T バリエーションでは、上記の命令によって、Thumb 命令セットが存在しなくても Thumb 状態を開始できます。この場合、分岐先にある命令を実行すると未定義命令例外が発生します。詳細については、P. A2-14 *「割り込みディセーブルビット」*を参照して下さい。

ARMv6 以降および ARMv5 の J バリエーションには、分岐と状態遷移 Jazelle® 命令が存在しています。詳細については、P. A4-21 *「BXJ」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.3.1 例

```
    B       label               ; branch unconditionally to label

    BCC     label               ; branch to label if carry flag is clear

    BEQ     label               ; branch to label if zero flag is set

    MOV     PC, #0              ; R15 = 0, branch to location zero

    BL      func                ; subroutine call to function


func    .
    .
    MOV     PC, LR              ; R15=R14, return to instruction after the BL
    MOV     LR, PC              ; store the address of the instruction
                                ; after the next one into R14 ready to return
    LDR     PC, =func           ; load a 32-bit value into the program counter
```

A3.3.2 分岐命令の一覧

B, BL　　分岐、およびリンク付き分岐。詳細については、P. A4-10「*B, BL*」を参照して下さい。

BLX　　リンク付き分岐と状態遷移。詳細については、P. A4-16「*BLX (1)*」と P. A4-18「*BLX (2)*」を参照して下さい。

BX　　分岐と状態遷移命令セット。詳細については、P. A4-20「*BX*」を参照して下さい。

BXJ　　分岐および Jazelle 状態への切り替え。詳細については、P. A4-21「*BXJ*」を参照して下さい。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.4 データ処理命令

ARM には 16 種類のデータ処理命令があります。命令の一覧を表 A3-2 に示します。

表 A3-2 データ処理命令

オペコード	ニーモニック	演算	アクション
0000	AND	論理積	Rd := Rn AND shifter_operand
0001	EOR	排他的論理和	Rd := Rn EOR shifter_operand
0010	SUB	減算	Rd := Rn - shifter_operand
0011	RSB	逆減算	Rd := shifter_operand - Rn
0100	ADD	加算	Rd := Rn + shifter_operand
0101	ADC	キャリー付き加算	Rd := Rn + shifter_operand + キャリーフラグ
0110	SBC	キャリー付き減算	Rd := Rn - shifter_operand - NOT（キャリーフラグ）
0111	RSC	キャリー付き逆減算	Rd := shifter_operand - Rn - NOT（キャリーフラグ）
1000	TST	テスト	Rn AND shifter_operand の後にフラグを更新
1001	TEQ	等価テスト	Rn EOR shifter_operand の後にフラグを更新
1010	CMP	比較	Rn - shifter_operand の後にフラグを更新
1011	CMN	2 の補数比較	Rn + shifter_operand の後にフラグ更新
1100	ORR	（包含的）論理和	Rd := Rn OR shifter_operand
1101	MOV	移動	Rd := shifter_operand（第 1 オペランド以外）
1110	BIC	ビットクリア	Rd := Rn AND NOT（shifter_operand）
1111	MVN	論理否定	Rd := NOT shifter_operand（第 1 オペランド以外）

ほとんどのデータ処理命令は、2 つのソースオペランドを使用します。ただし、Move と Move Not は 1 つだけです。比較命令とテスト命令は、条件フラグの更新のみを行います。それ以外のデータ処理命令では、レジスタに結果が格納され、必要に応じて条件フラグも更新されます。

2 つのソースオペランドのうち、1 つは常にレジスタです。もう 1 つは*シフタオペランド*で、イミディエート値またはレジスタのいずれかです。第2オペランドがレジスタ値の場合、シフトを適用できます。

`CMP`、`CMN`、`TST`、`TEQ` では、常に条件コードフラグが更新されます。アセンブラでは、命令で S ビットが自動的にセットされます。対応する命令で S ビットがクリアされている場合、その命令はデータ処理命令ではなく、いずれかの命令拡張空間に属します。詳細については、P. A3-32*「命令セットの拡張」*を参照して下さい。それ以外の命令では、命令ニーモニックに S が追加された場合（命令の S ビットがセットされる）にフラグが更新されます。詳細については、P. A2-11*「条件コードフラグ」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.4.1 命令のエンコード

```
<opcode1>{<cond>}{S}  <Rd>, <shifter_operand>
<opcode1> := MOV | MVN
<opcode2>{<cond>}  <Rn>, <shifter_operand>
<opcode2> := CMP | CMN | TST | TEQ
<opcode3>{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
<opcode3> := ADD | SUB | RSB | ADC | SBC | RSC | AND | BIC | EOR | ORR
```

31–28	27	26	25	24–21	20	19–16	15–12	11–0
cond	0	0	I	opcode	S	Rn	Rd	shifter_operand

I ビット イミディエートと <shifter_operand> のレジスタの形式を区別します。

S ビット 命令が条件コードを更新したことを示します。

Rn 最初のソースオペランドレジスタを示します。

Rd デスティネーションレジスタを示します。

shifter_operand 2 番目のソースオペランドを示します。シフタオペランドの詳細については、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」* を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.4.2 データ処理命令の一覧

`ADC`	キャリー付き加算。詳細については、P. A4-4「*ADC*」を参照して下さい。
`ADD`	加算。詳細については、P. A4-6「*ADD*」を参照して下さい。
`AND`	論理積。詳細については、P. A4-8「*AND*」を参照して下さい。
`BIC`	論理ビットクリア。詳細については、P. A4-12「*BIC*」を参照して下さい。
`CMN`	2の補数比較。詳細については、P. A4-26「*CMN*」を参照して下さい。
`CMP`	比較。詳細については、P. A4-28「*CMP*」を参照して下さい。
`EOR`	排他的論理和。詳細については、P. A4-32「*EOR*」を参照して下さい。
`MOV`	移動。詳細については、P. A4-69「*MOV*」を参照して下さい。
`MVN`	論理否定。詳細については、P. A4-83「*MVN*」を参照して下さい。
`ORR`	論理和。詳細については、P. A4-85「*ORR*」を参照して下さい。
`RSB`	逆減算。詳細については、P. A4-116「*RSB*」を参照して下さい。
`RSC`	キャリー付き逆減算。詳細については、P. A4-118「*RSC*」を参照して下さい。
`SBC`	キャリー付き減算。詳細については、P. A4-126「*SBC*」を参照して下さい。
`SUB`	減算。詳細については、P. A4-209「*SUB*」を参照して下さい。
`TEQ`	等価テスト。詳細については、P. A4-229「*TEQ*」を参照して下さい。
`TST`	テスト。詳細については、P. A4-231「*TST*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.5 乗算命令

ARM の乗算命令は、いくつかのクラスに分けられます。

通常 32 ビット x 32 ビット、結果の下位 32 ビットを返します。

ロング 32 ビット x 32 ビット、結果の 64 ビットを返します。

ハーフワード 16 ビット x 16 ビット、結果の 32 ビットを返します。

ワード∞ハーフワード
32 ビット x 16 ビット、結果の上位 32 ビットを返します。

最上位ワード 32 ビット x 32 ビット、結果の上位 32 ビットを返します。

デュアルハーフワード
ふたつの 16 ビット x 16 ビット、結果の 32 ビットを返します。

乗算命令はすべて、乗算器への入力データとして 2 つのレジスタオペランドを使用します。ARM プロセッサでは、シフトと加算、またはシフトと逆減算の命令の効率のため、定数による乗算の命令を直接サポートしていません。

A3.5.1 通常の乗算

結果の下位 32 ビットを生成する 32 ビット x 32 ビット乗算命令が 2 種類あります。

`MUL` 2 つのレジスタの値に対して乗算を行い、結果を 32 ビットに切り捨てて、3 番目のレジスタに結果を格納します。

`MLA` 2 つのレジスタの値に対して乗算を行い、3 番目のレジスタの値を加算し、その結果を 32 ビットに切り捨てて、4 番目のレジスタに結果を格納します。この命令は積和演算の実行に使用されます。

どちらの通常の乗算命令も、必要に応じて N（負）および Z（ゼロ）の条件コードフラグをセットできます。符号付き、符号なしのバリエーションは区別されません。結果の最下位 32 ビットのみがデスティネーションレジスタに格納され、オペランドの符号はこの値には影響しません。

A3.5.2 ロング乗算

64 ビットの結果を生成する 32 ビット x 32 ビット乗算命令が 5 種類あります。

バリエーションのうち 2 種類は、2 つのレジスタの値に対して乗算を行い、結果の 64 ビットを 3 番目と 4 番目のレジスタに格納します。符号付き（SMULL）と符号なし（UMULL）のバリエーションがあります。符号付きのバリエーションでは、ソースオペランドのいずれかまたは両方が負の場合、最上位の 32 ビットに異なる結果が生成されます。

バリエーションのうち 2 種類は、2 つのレジスタの値に対して乗算を行い、3 番目と 4 番目のレジスタの 64 ビット値を加算し、結果の 64 ビットをそれら（3 番目と 4 番目）のレジスタに戻します。符号付き（SMLAL）と符号なし（UMLAL）のバリエーションがあります。これらの命令ではロング乗算と積和命令が実行されます。

UMAAL は、2 つのレジスタの符号なしの値に対して乗算を行い、3 番目と 4 番目のレジスタにある 2 つの符号なし 32 ビット値を加算し、結果の符号なしの 64 ビット値をそれら（3 番目および 4 番目）のレジスタに戻します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

UMAAL 以外のロング乗算命令はすべて、必要に応じて N（負）および Z（ゼロ）の条件コードフラグをセットできます。UMAAL ではどのフラグも更新されません。

UMAAL は ARMv6 以降で使用できます。

A3.5.3 ハーフワード乗算

32 ビットの結果を生成する符号付き 16 ビット x 16 ビット乗算命令が 3 種類あります。

SMULxy　2 つのハーフレジスタの 16 ビット値に対して乗算を行い、結果の符号付き 32 ビット値を 3 番目のレジスタに格納します。

SMLAxy　2 つのハーフレジスタの 16 ビット値に対して乗算を行い、3 番目のレジスタの 32 ビット値を加算し、結果の符号付き 32 ビット値を 4 番目のレジスタに格納します。

SMLALxy　2 つのハーフレジスタの 16 ビット値に対して乗算を行い、3 番目および 4 番目のレジスタの 64 ビット値を加算し、結果の 64 ビット値をそれら（3 番目および 4 番目）のレジスタに戻します。

SMULxy および SMLALxy はどのフラグも更新しません。SMLAxy では、乗算でオーバフローが発生した場合に Q フラグをセットできます。x および y 指定子によって、レジスタの上位 (T) ビットと下位 (B) ビットのどちらがオペランドとして使用されているかが示されます。

これらの命令は ARMv5TE 以降で使用できます。

A3.5.4 ワード × ハーフワード乗算

上位 32 ビットの結果を生成する符号付き乗算命令が 2 種類あります。

SMULWy　最初のレジスタの 32 ビット値と、2 番目のレジスタに含まれるいずれかのハーフワードの 16 ビット値に対して乗算を行い、3 番目のレジスタに、結果の符号付き 48 ビット値の上位 32 ビットを格納します。

SMLAWy　最初のレジスタの 32 ビット値と、2 番目のレジスタに含まれるいずれかのハーフワードの 16 ビットの値に対して乗算を行い、上位 32 ビットを抽出して、3 番目のレジスタの 32 ビット値を加算し、4 番目のレジスタに結果の符号付き 32 ビット値を格納します。

SMLAWy では、乗算でオーバフローが発生した場合に Q フラグがセットされます。SMULWy ではどのフラグも更新されません。

これらの命令は ARMv5TE 以降で使用できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.5.5 最上位ワード乗算

結果の上位 32 ビットを生成する符号付き 32 ビット x 32 ビット乗算命令が 3 種類あります。

SMMUL　2 つのレジスタの 32 ビット値に対して乗算を行い、3 番目のレジスタに結果の符号付き 64 ビット値の上位 32 ビットを格納します。

SMMLA　2 つのレジスタの 32 ビット値に対して乗算を行い、上位 32 ビットを抽出し、3 番目のレジスタにある 32 ビット値を加算し、結果の符号付き 32 ビット値を 4 番目のレジスタに格納します。

SMMLS　2 つのレジスタの 32 ビット値に対して乗算を行い、上位 32 ビットを抽出し、3 番目のレジスタにある 32 ビット値からこの値を減算し、結果の符号付き 32 ビット値を 4 番目のレジスタに格納します。

これらの命令ではどのフラグも更新されません。

これらの命令は、ARMv6 以降で使用できます。

A3.5.6 デュアルハーフワード乗算

デュアル符号付き 16 ビット x 16 ビット乗算命令が 6 種類あります。

SMUAD　2 つのレジスタの上位ハーフワードの値同士と、下位ハーフワードの値同士の乗算を行います。2 つの積を加算して、結果の 32 ビット値を 3 番目のレジスタに格納します。

SMUSD　2 つのレジスタの上位ハーフワードの値同士と、下位ハーフワードの値同士の乗算を行います。一方の積をもう一方から減算して、結果の 32 ビット値を 3 番目のレジスタに格納します。

SMLAD　2 つのレジスタの 32 ビット値に対して乗算を行い、上位 32 ビットを抽出し、3 番目のレジスタの 32 ビット値からこの値を減算し、結果の符号付き 32 ビット値を 4 番目のレジスタに格納します。

SMLSD　2 つのレジスタの 32 ビット値に対して乗算を行い、上位 32 ビットを抽出し、3 番目のレジスタの 32 ビット値を加算し、結果の符号付き 32 ビット値を 4 番目のレジスタに格納します。

SMLALD　2 つのレジスタの 32 ビット値に対して乗算を行い、上位 32 ビットを抽出し、3 番目のレジスタの 32 ビット値からこの値を減算し、結果の符号付き 32 ビット値を 4 番目のレジスタに格納します。

SMLSLD　2 つのレジスタの 32 ビット値に対して乗算を行い、上位 32 ビットを抽出し、3 番目のレジスタの 32 ビット値からこの値を減算し、結果の符号付き 32 ビット値を 4 番目のレジスタに格納します。

SMUAD、SMLAD、SMLSLD では、演算でオーバフローが発生した場合に Q フラグをセットできます。他のすべての命令では、どのフラグも更新されません。

これらの命令は、ARMv6 以降で使用できます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.5.7 例

```
MUL    R4, R2, R1              ; Set R4 to value of R2 multiplied by R1
MULS   R4, R2, R1              ; R4 = R2 x R1, set N and Z flags
MLA    R7, R8, R9, R3          ; R7 = R8 x R9 + R3
SMULL  R4, R8, R2, R3          ; R4 = bits 0 to 31 of R2 x R3

                               ; R8 = bits 32 to 63 of R2 x R3
UMULL  R6, R8, R0, R1          ; R8, R6 = R0 x R1
UMLAL  R5, R8, R0, R1          ; R8, R5 = R0 x R1 + R8, R5
```

A3.5.8 乗算命令の一覧

MLA 積和。詳細については、P. A4-67 「*MLA*」を参照して下さい。

MUL 乗算。詳細については、P. A4-81 「*MUL*」を参照して下さい。

SMLA<x><y> 符号付きハーフワード積和。詳細については、P. A4-142 「*SMLA<x><y>*」を参照して下さい。

SMLAD 符号付きハーフワード積和、デュアル。詳細については、P. A4-145 「*SMLAD*」を参照して下さい。

SMLAL 符号付き積和ロング。詳細については、P. A4-147 「*SMLAL*」を参照して下さい。

SMLAL<x><y> 符号付きハーフワード積和ロング。詳細については、P. A4-149 「*SMLAL<x><y>*」を参照して下さい。

SMLALD 符号付きハーフワード積和ロング、デュアル。詳細については、P. A4-151 「*SMLALD*」を参照して下さい。

SMLAW<y> 符号付きハーフワードとワードの積和。詳細については、P. A4-153 「*SMLAW<y>*」を参照して下さい。

SMLSD 符号付きハーフワード積減算、デュアル。詳細については、P. A4-145 「*SMLAD*」を参照して下さい。

SMLSLD 符号付きハーフワード積減算ロング、デュアル。詳細については、P. A4-151「*SMLALD*」を参照して下さい。

SMMLA 符号付き最上位ワード積和。詳細については、P. A4-159「*SMMLA*」を参照して下さい。

SMMLS 符号付き最上位ワード積減算。詳細については、P. A4-161「*SMMLS*」を参照して下さい。

SMMUL 符号付き最上位ワード乗算。詳細については、P. A4-163「*SMMUL*」を参照して下さい。

SMUAD 符号付きハーフワード乗算、加算、デュアル。詳細については、P. A4-165 「*SMUAD*」を参照して下さい。

SMUL<x><y> 符号付きハーフワード乗算。詳細については、P. A4-167 「*SMUL<x><y>*」を参照して下さい。

SMULL 符号付きロング乗算。詳細については、P. A4-169 「*SMULL*」を参照して下さい。

SMULW<y> 符号付きハーフワードとワードの乗算。詳細については、P. A4-171 「*SMULW<y>*」を参照して下さい。

SMUSD 符号付きハーフワード乗算、減算、デュアル。詳細については、P. A4-173 「*SMUSD*」を参照して下さい。

UMAAL 符号なし上位ロング積和。詳細については、P. A4-248 「*UMAAL*」を参照して下さい。

UMLAL 符号なしロング積和。詳細については、P. A4-250 「*UMLAL*」を参照して下さい。

UMULL 符号なしロング乗算。詳細については、P. A4-252 「*UMULL*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.6 並列加算命令と並列減算命令

通常のデータ処理命令に加えて、ARMv6 では並列加算命令および並列減算命令が導入されました。

6 種類の基本命令があります。

`ADD16` 2つのレジスタの、上位ハーフワード同士を加算して結果の上位ハーフワードを生成し、下位ハーフワード同士を加算して結果の下位ハーフワードを生成します。

`ADDSUBX` 次の処理を実行します。
1. 第 2 オペランドレジスタのハーフワードを交換します。
2. 上位ハーフワード同士の加算と下位ハーフワード同士の減算を行います。

`SUBADDX` 次の処理を実行します。
1. 第 2 オペランドレジスタのハーフワードを交換します。
2. 上位ハーフワード同士の減算と下位ハーフワード同士の加算を行います。

`SUB16` 第 2 オペランドレジスタの上位ハーフワードから第 1 オペランドレジスタの上位ハーフワードを減算して、結果の上位ハーフワードを生成します。

第 1 オペランドレジスタの下位ハーフワードから第 2 オペランドレジスタの下位ハーフワードを減算して、結果の下位ハーフワードを生成します。

`ADD8` 第 2 オペランドレジスタの各バイトを第 1 オペランドレジスタの対応するバイトに加算して、結果の対応するバイトを生成します。

`SUB8` 第 2 オペランドレジスタの各バイトを第 1 オペランドレジスタの対応するバイトから減算して、結果の対応するバイトを生成します。

これらの 6 種類の命令はそれぞれ、以下の接頭文字によるバリエーションで使用できます。

`S` 符号付き演算モジュロ 2^8 または 2^{16}。CPSR GE ビットがセットされます。詳細については、P. A2-13「*GE[3:0] ビット*」を参照して下さい。

`Q` 符号付き飽和演算。

`SH` 符号付き演算で、オーバフローを防ぐために結果を 2 で除算します。

`U` 符号なし演算モジュロ 2^8 または 2^{16}。CPSR GE ビットがセットされます。詳細については、P. A2-13「*GE[3:0] ビット*」を参照して下さい。

`UQ` 符号なし飽和演算。

`UH` 符号なし演算で、オーバフローを防ぐために結果を 2 で除算します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.6.1 並列演算命令の一覧

`QADD16`	デュアル16ビット符号付き飽和加算。詳細については、P. A4-95「*QADD16*」を参照して下さい。
`QADD8`	クワッド8ビット符号付き飽和加算。詳細については、P. A4-96「*QADD8*」を参照して下さい。
`QADDSUBX`	16ビット交換、符号付き飽和加算、減算。詳細については、P. A4-98「*QADDSUBX*」を参照して下さい。
`QSUB16`	デュアル16ビット符号付き飽和減算。詳細については、P. A4-105「*QSUB16*」を参照して下さい。
`QSUB8`	クワッド8ビット符号付き飽和減算。詳細については、P. A4-106「*QSUB8*」を参照して下さい。
`QSUBADDX`	16ビット交換、符号付き飽和減算、加算。詳細については、P. A4-108「*QSUBADDX*」を参照して下さい。
`SADD16`	デュアル16ビット符号付き加算。詳細については、P. A4-120「*SADD16*」を参照して下さい。
`SADD8`	クワッド8ビット符号付き加算。詳細については、P. A4-122「*SADD8*」を参照して下さい。
`SADDSUBX`	16ビット交換、符号付き加算、減算。詳細については、P. A4-124「*SADDSUBX*」を参照して下さい。
`SSUB16`	デュアル16ビット符号付き減算。詳細については、P. A4-181「*SSUB16*」を参照して下さい。
`SSUB8`	クワッド8ビット符号付き減算。詳細については、P. A4-183「*SSUB8*」を参照して下さい。
`SSUBADDX`	16ビット交換、符号付き減算、加算。詳細については、P. A4-185「*SSUBADDX*」を参照して下さい。
`SHADD16`	デュアル16ビット符号付きハーフ加算。詳細については、P. A4-131「*SHADD16*」を参照して下さい。
`SHADD8`	クワッド8ビット符号付きハーフ加算。詳細については、P. A4-132「*SHADD8*」を参照して下さい。
`SHADDSUBX`	16ビット交換、符号付きハーフ加算、減算。詳細については、P. A4-134「*SHADDSUBX*」を参照して下さい。
`SHSUB16`	デュアル16ビット符号付きハーフ減算。詳細については、P. A4-136「*SHSUB16*」を参照して下さい。
`SHSUB8`	クワッド8ビット符号付きハーフ減算。詳細については、P. A4-138「*SHSUB8*」を参照して下さい。
`SHSUBADDX`	16ビット交換、符号付きハーフ減算、加算。詳細については、P. A4-140「*SHSUBADDX*」を参照して下さい。
`UADD16`	デュアル16ビット符号なし加算。詳細については、P. A4-233「*UADD16*」を参照して下さい。
`UADD8`	クワッド8ビット符号なし加算。詳細については、P. A4-234「*UADD8*」を参照して下さい。
`UADDSUBX`	16ビット交換、符号なし加算、減算。詳細については、P. A4-236「*UADDSUBX*」を参照して下さい。
`USUB16`	デュアル16ビット符号なし減算。詳細については、P. A4-270「*USUB16*」を参照して下さい。
`USUB8`	クワッド8ビット符号なし減算。詳細については、P. A4-271「*USUB8*」を参照して下さい。
`USUBADDX`	16ビット交換、符号なし減算、加算。詳細については、P. A4-273「*USUBADDX*」を参照して下さい。
`UHADD16`	デュアル16ビット符号なしハーフ加算。詳細については、P. A4-238「*UHADD16*」を参照して下さい。
`UHADD8`	クワッド8ビット符号なしハーフ加算。詳細については、P. A4-239「*UHADD8*」を参照して下さい。
`UHADDSUBX`	16ビット交換、符号なしハーフ加算、減算。詳細については、P. A4-241「*UHADDSUBX*」を参照して下さい。
`UHSUB16`	デュアル16ビット符号なしハーフ減算。詳細については、P. A4-243「*UHSUB16*」を参照して下さい。
`UHSUB8`	クワッド8ビット符号なしハーフ減算。詳細については、P. A4-244「*UHSUB8*」を参照して下さい。
`UHSUBADDX`	16ビット交換、符号なしハーフ減算、加算。詳細については、P. A4-246「*UHSUBADDX*」を参照して下さい。
`UQADD16`	デュアル16ビット符号なし飽和加算。詳細については、P. A4-254「*UQADD16*」を参照して下さい。
`UQADD8`	クワッド8ビット符号なし飽和加算。詳細については、P. A4-255「*UQADD8*」を参照して下さい。
`UQADDSUBX`	16ビット交換、符号なし飽和加算、減算。詳細については、P. A4-256「*UQADDSUBX*」を参照して下さい。
`UQSUB16`	デュアル16ビット符号なし飽和減算。詳細については、P. A4-258「*UQSUB16*」を参照して下さい。
`UQSUB8`	クワッド8ビット符号なし飽和減算。詳細については、P. A4-259「*UQSUB8*」を参照して下さい。
`UQSUBADDX`	16ビット交換、符号なし飽和減算、加算。詳細については、P. A4-260「*UQSUBADDX*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.7 拡張命令

ARMv6 以降には、バイトをハーフワードまたはワードに、ハーフワードをワードに符号拡張またはゼロ拡張することによってデータをアンパックする命令が複数あります。必要に応じて、その結果を別のレジスタの内容に加算できます。拡張の前に 8 ビットの整数倍でオペランドレジスタをローテートできます。

6 種類の基本命令があります。

`XTAB16` 1 つのレジスタのビット [23:16] とビット [7:0] を 16 ビットに拡張して、対応するハーフワードを別のレジスタの値に加算します。

`XTAB` 1 つのレジスタのビット [7:0] を 32 ビットに拡張して､別のレジスタの値に加算します。

`XTAH` 1 つのレジスタのビット [15:0] を 32 ビットに拡張して､別のレジスタの値に加算します。

`XTB16` ビット [23:16] とビット [7:0] をそれぞれ 16 ビットに拡張します。

`XTB` ビット [7:0] を 32 ビットに拡張します。

`XTH` ビット [15:0] を 32 ビットに拡張します。

これらの 6 種類の命令にはそれぞれ、以下の接頭文字で示されるバリエーションが使用できます。

`S` 符号拡張、加算モジュロ 2^{32} または 2^{16} あり、またはなし。

`U` ゼロ（符号なし）拡張、加算モジュロ 2^{32} または 2^{16} あり、またはなし。

A3.7.1 符号 / ゼロ拡張および加算命令の一覧

`SXTAB16` バイトからハーフワードへの符号拡張、ハーフワードの加算。詳細については、P. A4-219「*SXTAB16*」を参照して下さい。

`SXTAB` バイトからワードへの符号拡張、加算。詳細については、P. A4-217「*SXTAB*」を参照して下さい。

`SXTAH` ハーフワードからワードへの符号拡張、加算。詳細については、P. A4-221「*SXTAH*」を参照して下さい。

`SXTB16` バイトからハーフワードへの符号拡張。詳細については、P. A4-225「*SXTB16*」を参照して下さい。

`SXTB` バイトからワードへの符号拡張。詳細については、P. A4-223「*SXTB*」を参照して下さい。

`SXTH` ハーフワードからワードへの符号拡張。詳細については、P. A4-227「*SXTH*」を参照して下さい。

`UXTAB16` バイトからハーフワードへのゼロ拡張、ハーフワードの加算。詳細については、P. A4-277「*UXTAB16*」を参照して下さい。

`UXTAB` バイトからワードへのゼロ拡張、加算。詳細については、P. A4-275「*UXTAB*」を参照して下さい。

`UXTAH` ハーフワードからワードへのゼロ拡張、加算。詳細については、P. A4-279「*UXTAH*」を参照して下さい。

`UXTB16` バイトからハーフワードへのゼロ拡張。詳細については、P. A4-283「*UXTB16*」を参照して下さい。

`UXTB` バイトからワードへのゼロ拡張。詳細については、P. A4-281「*UXTB*」を参照して下さい。

`UXTH` ハーフワードからワードへのゼロ拡張。詳細については、P. A4-285「*UXTH*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.8 その他の算術演算命令

ARMv5 以降には、その他の算術演算命令がいくつか存在します。

A3.8.1 先行ゼロカウント

ARMv5 以降には先行ゼロカウント（CLZ）命令が存在します。この命令では、オペランド内で最上位にある 1 のビットよりも上位にある 0 のビットの数が返されます（オペランドが 0 の場合、32 が返されます）。この命令の一般的な用途は次の 2 つです。

- オペランドを*正規化*して最上位ビットを 1 にするために何ビットのシフトが必要かの判断（整数除算ルーチンで使用できます）
- ビットマスクの最も優先度の高いビットの特定

詳細については、P. A4-25「*CLZ*」を参照して下さい。

A3.8.2 符号なし差分絶対値和

ARMv6 には、符号なし絶対値の差の合計（USAD8）命令および累算付き符号なし絶対値の差の合計（USADA8）命令があります。

これらの命令は次の演算を行います。

1. 2 つのレジスタから、対応するバイトを取り出します。
2. バイトの各ペアごとに、符号なしの値の絶対値の差を計算します。
3. 4 つの絶対値を加算します。
4. 必要に応じて、絶対値の差の合計と 3 番目のレジスタの値を合計します。

詳細については、P. A4-262「*USAD8*」と P. A4-264「*USADA8*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.9 その他の命令

ARMv6 以降には、既に上げた以外のその他の命令がいくつか存在します。

`PKHBT` (ハーフワード下位上位のパック)第 1 オペランドの下位ハーフワードと、シフトされた第 2 オペランドの上位ハーフワードを結合します。シフトは左に、0 ～ 31 の任意の指定ビット数だけ行われます。

詳細については、P. A4-87「*PKHBT*」を参照して下さい。

`PKHTB` (ハーフワード上位下位のパック)第 1 オペランドの上位ハーフワードと、シフトされた第 2 オペランドの下位ハーフワードを結合します。シフトは算術右シフトで、1 ～ 32 の任意の指定ビット数だけ行われます。

詳細については、P. A4-89「*PKHTB*」を参照して下さい。

`SSAT` (符号付き飽和)符号付きの値を符号付き範囲で飽和します。飽和を発生させるビット位置は選択できます。飽和を発生させる前に値をシフトすることもできます。

詳細については、P. A4-177「*SSAT*」を参照して下さい。

`USAT` (符号なし飽和)符号付きの値を符号なし範囲で飽和します。飽和を発生させるビット位置は選択できます。飽和を発生させる前に値をシフトすることもできます。

詳細については、P. A4-266「*USAT*」を参照して下さい。

`SSAT16` 2 つの符号付き 16 ビット値を符号付き範囲で飽和します。飽和を発生させるビット位置は選択できます。

詳細については、P. A4-179「*SSAT16*」を参照して下さい。

`USAT16` 2 つの符号付き 16 ビット値を符号なし範囲で飽和します。飽和を発生させるビット位置は選択できます。

詳細については、P. A4-268「*USAT16*」を参照して下さい。

`SEL` (選択)GE フラグの値に応じて、第 1 オペランドまたは第 2 オペランドのいずれかから結果の各バイトを選択します。GE フラグは、並列加算または並列減算の結果を記録します。詳細については、P. A3-14「*並列加算命令と並列減算命令*」を参照して下さい。

詳細については、P. A4-128「*SEL*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.10 ステータスレジスタアクセス命令

プログラムステータスレジスタの内容を汎用レジスタとの間で移動する命令が 2 種類あります。CPSR および SPSR の両方にアクセス可能です。

また ARMv6 では、CPSR の特定のビット、またはビットのグループに直接書き込みができる命令が複数存在します。

各ステータスレジスタは従来は 4 つの 8 ビットフィールドに分割され、個別に書き込みが可能でした。

ビット **[31:24]**	フラグフィールド
ビット **[23:16]**	ステータスフィールド
ビット **[15:8]**	拡張フィールド
ビット **[7:0]**	制御フィールド

ARMv6 から、ARM アーキテクチャではステータスフィールドと拡張フィールドが使用されています。ビットフィールドの用法モデルでは、バイト幅の定義は反映されなくなりました。改訂後のカテゴリは、P. A2-11 「*PSR ビットのタイプ*」で定義されています。

A3.10.1 CPSR の値

CPSR の値を変更する目的は 5 つあります。

- 条件コードフラグの値（および、存在する場合は Q フラグの値）を既知の値にセットする。
- 割り込みを有効または無効にする。
- プロセッサモードを変更する（スタックポインタの初期化などのため）。
- ロード / ストア操作のエンディアン形式を変更する。
- プロセッサ状態（J ビットと T ビット）を変更する。

——— 注 ———

T ビットと J ビットは、CPSR への書き込みによって直接変更することはできません。BX 命令、BLX 命令、BXJ 命令、例外復帰用に設計されている命令での暗黙的な SPSR から CPSR への移動によってのみ変更可能です。T ビットまたは J ビットを直接変更して Thumb または Jazelle 状態の開始または終了を試みた場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.10.2 例

これらの例では、ARM プロセッサが既に特権モードになっていることを前提としています。開始時に ARM プロセッサがユーザモードの場合、フラグの更新のみ影響があります。

```
MRS    R0, CPSR                      ; Read the CPSR
BIC    R0, R0, #0xF0000000           ; Clear the N, Z, C and V bits
MSR    CPSR_f, R0                    ; Update the flag bits in the CPSR
                                     ; N, Z, C and V flags now all clear

MRS    R0, CPSR                      ; Read the CPSR
ORR    R0, R0, #0x80                 ; Set the interrupt disable bit
MSR    CPSR_c, R0                    ; Update the control bits in the CPSR
                                     ; interrupts (IRQ) now disabled

MRS    R0, CPSR                      ; Read the CPSR
BIC    R0, R0, #0x1F                 ; Clear the mode bits
ORR    R0, R0, #0x11                 ; Set the mode bits to FIQ mode
MSR    CPSR_c, R0                    ; Update the control bits in the CPSR
                                     ; now in FIQ mode
```

A3.10.3 ステータスレジスタアクセス命令の一覧

`MRS` PSR を汎用レジスタに移動します。詳細については、P. A4-75「*MRS*」を参照して下さい。

`MSR` 汎用レジスタを PSR に移動します。詳細については、P. A4-77「*MSR*」を参照して下さい。

`CPS` プロセッサ状態を変更します。他の CPSR ビットを変更せずに、1 つまたは複数のプロセッサモードと CPSR の割り込みイネーブルビットを変更します。詳細については、P. A4-29「*CPS*」を参照して下さい。

`SETEND` CPSR の他のビットを変更せずに、CPSR のエンディアン形式（E ビット）を変更します。詳細については、P. A4-130「*SETEND*」を参照して下さい。

プロセッサ状態のビットは、PC を更新する各種の分岐命令、ロード命令、復帰命令でも更新できます。これらの命令が Jazelle 状態の開始と終了や、Thumb インターワーキングに使用される場合、変更が発生します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.11 ロード / ストア命令

ARM アーキテクチャには、単一のレジスタまたはレジスタのペアとメモリとの間で値をロード / ストアする、2 種類の命令が存在します。

- 最初のタイプは、32 ビットワードまたは 8 ビットの符号なしバイトをロード / ストアできます。
- 2 番目のタイプは、16 ビットの符号なしハーフワードをロード / ストアでき、16 ビットのハーフワードまたは 8 ビットのバイトを符号拡張付きでロードできます。ARMv5TE 以降では、32 ビットのワードのペアをロード / ストアすることもできます。

A3.11.1 アドレシングモード

どちらのタイプの命令も、アドレシングモードは次の 2 つの要素で構成されます。

- ベースレジスタ
- オフセット

ベースレジスタには、(PC を含む、位置独立コードの PC 相対アドレシングモードが使用可能な) 任意の汎用レジスタが使用できます。

オフセットの形式は、以下の 3 種類のいずれかです。

イミディエート

オフセットは符号なしの数値で、ベースレジスタの値に加算または減算されます。イミディエートオフセットのアドレシングは、構造体フィールド、スタックオフセット、入力 / 出力レジスタなど、データオブジェクトの先頭から一定の位置にあるデータエレメントへのアクセスに便利です。

ワードおよび符号なしバイト命令の場合、イミディエートオフセットは 12 ビット値です。ハーフワードおよび符号付きバイト命令の場合は 8 ビット値です。

レジスタ

オフセットは PC 以外の汎用レジスタで、ベースレジスタの値に加算または減算されます。レジスタオフセットは通常、配列やデータのブロックへのアクセスに便利です。

スケーリング付きレジスタ

オフセットは、イミディエート値によってシフトされた PC 以外の汎用レジスタの値で、ベースレジスタの値に加算または減算されます。データ処理命令で使用されるのと同じシフト演算が使用できます (論理左シフト、論理右シフト、算術右シフト、右ローテート)。ただし、インデクス付き配列の各配列エレメントのサイズによるスケーリングが行える論理左シフトが最も便利です。

スケーリング付きレジスタオフセットは、ワードおよび符号なしバイト命令でのみ使用できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

これら 3 種類のオフセットのいずれかと、ベースレジスタを使用して、次に示す 3 つのアドレシングモードのいずれかを使用してメモリアドレスが生成されます。

オフセット ベースレジスタとオフセットが加算または減算され、メモリアドレスが生成されます。

プリインデクス ベースレジスタとオフセットが加算または減算され、メモリアドレスが生成されます。次に、生成されたアドレスがベースレジスタに書き戻されるため、配列やメモリブロックの各エレメントを自動的にインデクス指定できます。

ポストインデクス ベースレジスタの値がそのままメモリアドレスとして使用されます。ベースレジスタとオフセットが加算または減算され、結果の値がベースレジスタに書き戻されるため、配列やメモリブロックの各要素を自動的にインデクス指定できます。

A3.11.2 ワードまたは符号なしバイトのロード / ストア命令

ロード命令は、メモリから単一の値をロードして、汎用レジスタにその値を書き込みます。

ストア命令は、汎用レジスタから値を読み出して、メモリにその値を格納します。

これらの命令は単一の命令フォーマットです。

```
LDR|STR{<cond>}{B}{T} Rd, <addressing_mode>
```

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	1	I	P	U	B	W	L	Rn	Rd	addressing_mode_specific

I、P、U、W 各種の <addressing_mode> を区別するビットです。詳細については、P. A5-18 *「アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア」*を参照して下さい。

L ビット ロード（L == 1）とストア（L == 0）命令を区別します。

B ビット 符号なしバイト（B == 1）とワード（B == 0）アクセスを区別します。

Rn <addressing_mode> で使用されるベースレジスタを示します。

Rd 内容がロードまたはストアされるレジスタを示します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.11.3 ハーフワードまたはダブルワードのロード / ストアと符号付きバイトのロード命令

ロード命令は、メモリから単一の値をロードして、ひとつの汎用レジスタまたは汎用レジスタのペアにその値を書き込みます。

ストア命令は、ひとつの汎用レジスタまたは汎用レジスタのペアから値を読み出して、メモリにその値を格納します。

これらの命令は単一の命令フォーマットです。

```
LDR|STR{<cond>}D|H|SH|SB  Rd, <addressing_mode>
```

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	P	U	I	W	L	Rn	Rd	addr_mode	1	S	H	1	addr_mode

addr_mode アドレシングモード固有のビット。

I、P、U、W アドレシングモードのタイプを特定するビット。詳細については、P. A5-33 *「アドレシングモード 3 - その他のロードとストア」*を参照して下さい。

L、S、H これらのビットの組み合わせにより、符号付きまたは符号なしのロードまたはストア、およびダブルワード、ハーフワード、バイトアクセスが指定されます。詳細については、P. A5-33 *「アドレシングモード 3 - その他のロードとストア」*を参照して下さい。

Rn アドレシングモードによって使用されるベースレジスタを示します。

Rd 内容がロードまたはストアされるレジスタを示します。

A3.11.4 例

```
LDR     R1, [R0]                ; Load R1 from the address in R0
LDR     R8, [R3, #4]            ; Load R8 from the address in R3 + 4
LDR     R12, [R13, #-4]         ; Load R12 from R13 - 4
STR     R2, [R1, #0x100]        ; Store R2 to the address in R1 + 0x100

LDRB    R5, [R9]                ; Load byte into R5 from R9
                                ;  (zero top 3 bytes)
LDRB    R3, [R8, #3]            ; Load byte to R3 from R8 + 3
                                ;  (zero top 3 bytes)
STRB    R4, [R10, #0x200]       ; Store byte from R4 to R10 + 0x200

LDR     R11, [R1, R2]           ; Load R11 from the address in R1 + R2
STRB    R10, [R7, -R4]          ; Store byte from R10 to addr in R7 - R4

LDR     R11, [R3, R5, LSL #2]   ; Load R11 from R3 + (R5 x 4)
LDR     R1, [R0, #4]!           ; Load R1 from R0 + 4, then R0 = R0 + 4
STRB    R7, [R6, #-1]!          ; Store byte from R7 to R6 - 1,
                                ;  then R6 = R6 - 1

LDR     R3, [R9], #4            ; Load R3 from R9, then R9 = R9 + 4
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

```
STR     R2, [R5], #8            ; Store R2 to R5, then R5 = R5 + 8

LDR     R0, [PC, #40]           ; Load R0 from PC + 0x40 (= address of
                                ;  the LDR instruction + 8 + 0x40)
LDR     R0, [R1], R2            ; Load R0 from R1, then R1 = R1 + R2

LDRH    R1, [R0]                ; Load halfword to R1 from R0
                                ;  (zero top 2 bytes)
LDRH    R8, [R3, #2]            ; Load halfword into R8 from R3 + 2
LDRH    R12, [R13, #-6]         ; Load halfword into R12 from R13 - 6
STRH    R2, [R1, #0x80]         ; Store halfword from R2 to R1 + 0x80

LDRSH   R5, [R9]                ; Load signed halfword to R5 from R9
LDRSB   R3, [R8, #3]            ; Load signed byte to R3 from R8 + 3
LDRSB   R4, [R10, #0xC1]        ; Load signed byte to R4 from R10 + 0xC1

LDRH    R11, [R1, R2]           ; Load halfword into R11 from address
                                ;  in R1 + R2
STRH    R10, [R7, -R4]          ; Store halfword from R10 to R7 - R4

LDRSH   R1, [R0, #2]!           ; Load signed halfword R1 from R0 + 2,
                                ;  then R0 = R0 + 2

LDRSB   R7, [R6, #-1]!          ; Load signed byte to R7 from R6 - 1,
                                ;  then R6 = R6 - 1
LDRH    R3, [R9], #2            ; Load halfword to R3 from R9,
                                ;  then R9 = R9 + 2
STRH    R2, [R5], #8            ; Store halfword from R2 to R5,
                                ;  then R5 = R5 + 8
LDRD    R4, [R9]                ; Load word into R4 from
                                ;  the address in R9
                                ; Load word into R5 from
                                ;  the address in R9 + 4
STRD    R8, [R2, #0x2C]         ; Store R8 at the address in
                                ;    R2 + 0x2C
                                ; Store R9 at the address in
                                ;    R2 + 0x2C+4
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.11.5 ロード / ストア命令の一覧

`LDR`	ワードをロードします。詳細については、P. A4-44「*LDR*」を参照して下さい。
`LDRB`	バイトのロード。詳細については、P. A4-47「*LDRB*」を参照して下さい。
`LDRBT`	ユーザモード権限でのバイトのロード。詳細については、P. A4-49「*LDRBT*」を参照して下さい。
`LDRD`	ダブルワードのロード。詳細については、P. A4-51「*LDRD*」を参照して下さい。
`LDREX`	排他ロード。詳細については、P. A4-53「*LDREX*」を参照して下さい。
`LDRH`	符号なしハーフワードのロード。詳細については、P. A4-55「*LDRH*」を参照して下さい。
`LDRSB`	符号付きバイトのロード。詳細については、P. A4-57「*LDRSB*」を参照して下さい。
`LDRSH`	符号付きハーフワードのロード。詳細については、P. A4-59「*LDRSH*」を参照して下さい。
`LDRT`	ユーザモード権限でのワードのロード。詳細については、P. A4-61「*LDRT*」を参照して下さい。
`STR`	ワードのストア。詳細については、P. A4-194「*STR*」を参照して下さい。
`STRB`	バイトのストア。詳細については、P. A4-196「*STRB*」を参照して下さい。
`STRBT`	ユーザモード権限でのバイトのストア。詳細については、P. A4-198「*STRBT*」を参照して下さい。
`STRD`	ダブルワードのストア。詳細については、P. A4-200「*STRD*」を参照して下さい。
`STREX`	排他ストア。詳細については、P. A4-203「*STREX*」を参照して下さい。
`STRH`	ハーフワードのストア。詳細については、P. A4-205「*STRH*」を参照して下さい。
`STRT`	ユーザモード権限でのワードのストア。詳細については、P. A4-207「*STRT*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.12 複数ロード / ストア命令

複数ロード命令は、メモリから汎用レジスタのサブセットまたは全部をロードします。

複数ストア命令は、汎用レジスタのサブセットまたは全部をメモリにストアします。

複数ロード / ストア命令は単一の命令フォーマットです。

```
LDM{<cond>}<addressing_mode>  Rn{!}, <registers>{^}
STM{<cond>}<addressing_mode>  Rn{!}, <registers>{^}
```

各ビットの説明

```
<addressing_mode> = IA | IB | DA | DB | FD | FA | ED | EA
```

31–28	27	26	25	24	23	22	21	20	19–16	15–0
cond	1	0	0	P	U	S	W	L	Rn	レジスタリスト

レジスタリスト <registers> のリストには、各汎用レジスタに対応するビットが含まれています。ビット 0 は R0、ビット 15 は R15（PC）に対応します。

レジスタ構文のリストは、開始の括弧、カンマで区切られたレジスタのリスト、終了の括弧です。ある範囲のレジスタを指定するには、範囲の最初と最後のレジスタをマイナス記号の前後に指定します。

P、U、W ビット アドレシングモードのタイプを指定します。詳細については、P. A5-41 *「アドレシングモード 4 - 複数ロード / ストア」*を参照して下さい。

S ビット PC をロードする LDM の場合、S ビットは、すべてのレジスタがロードされた後に SPSR から CPSR がロードされることを示します。すべての STM および PC をロードしないすべての LDM では、プロセッサが特権モードの場合、ユーザモードのバンクレジスタが転送され、現在のモードのレジスタは転送されないことを示します。

L ビット ロード（L == 1）とストア命令（L == 0）を区別します。

Rn アドレシングモードによって使用されるベースレジスタを示します。

A3.12.1 例

```
STMFD    R13!, {R0 - R12, LR}
LDMFD    R13!, {R0 - R12, PC}
LDMIA    R0, {R5 - R8}
STMDA    R1!, {R2, R5, R7 - R9, R11}
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.12.2 複数ロード / ストア命令の一覧

`LDM`	複数ロード。詳細については、P. A4-36「*LDM (1)*」を参照して下さい。
`LDM`	ユーザレジスタの複数ロード。詳細については、P. A4-39「*LDM (2)*」を参照して下さい。
`LDM`	CPSR を復元する複数ロード。詳細については、P. A4-41「*LDM (3)*」を参照して下さい。
`STM`	複数ストア。詳細については、P. A4-190「*STM (1)*」を参照して下さい。
`STM`	ユーザレジスタの複数ストア。詳細については、P. A4-192「*STM (2)*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.13 セマフォ命令

ARM 命令セットには、2 種類のセマフォ命令が存在します。

- スワップ（SWP）
- バイトスワップ（SWPB）

これらの命令はプロセスの同期のために用意されています。どちらの命令もアトミックなロードおよびストア操作を生成し、割り込まれることなくメモリセマフォのロードと変更を可能にします。

SWP および SWPB のアドレシングモードは 1 つで、レジスタの内容がそのアドレスになります。ストアする値とロード先の指定には別のレジスタが使用されます。両方に同じレジスタが指定されている場合、SWP によってレジスタの値とメモリの値が交換されます。

セマフォ命令には、比較および条件付き書き込みの機能はありません。必要な場合は、明示的に実行する必要があります。

—— 注 ——

ARMv6 では、スワップ命令とスワップバイト命令は推奨されません。すべてのソフトウェアについて、P. A3-25「*ロード / ストア命令の一覧*」にある新しい LDREX および STREX 同期化基本命令の使用に移行することを推奨します。

A3.13.1 例

```
SWP    R12, R10, [R9]       ; load R12 from address R9 and
                            ; store R10 to address R9

SWPB   R3, R4, [R8]         ; load byte to R3 from address R8 and
                            ; store byte from R4 to address R8

SWP    R1, R1, [R2]         ; Exchange value in R1 and address in R2
```

A3.13.2 セマフォ命令の一覧

SWP　　スワップ。詳細については、P. A4-213「*SWP*」を参照して下さい。

SWPB　　バイトスワップ。詳細については、P. A4-215「*SWPB*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.14 例外生成命令

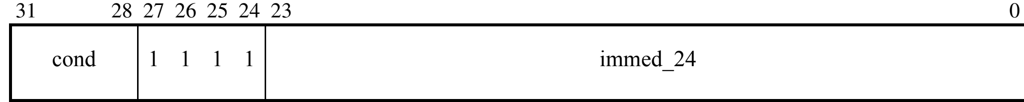
ARM 命令セットには、プロセッサ例外を発生させることを主な目的とした命令が 2 種類存在します。

- ソフトウェア割り込み（`SWI`）命令は SWI 例外の発生に使用されます（詳細については、P. A2-20 *「ソフトウェア割り込み例外」*を参照して下さい）。これは ARM 命令セットの主要な機能で、これにより、ユーザモードのコードで特権モードのオペレーティングシステムのコードを呼び出すことができます。
- ブレークポイント（`BKPT`）命令は、ARMv5 以降でソフトウェアブレークポイントに使用されます。デフォルトの動作は、プリフェッチアボート例外を発生させることです（詳細については、P. A2-20 *「プリフェッチアボート（命令フェッチ中に発生するメモリアボート）」*を参照して下さい）。前もってプリフェッチアボートベクタにインストールしたデバッグ監視プログラムでこの例外を処理できます。

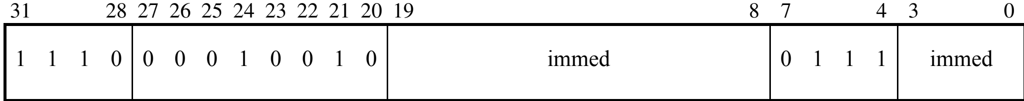
  システム内にデバッグハードウェアが存在している場合、デフォルトの動作をオーバライドできます。この処理が発生するか、どのように発生するかの詳細は、実装定義です。

A3.14.1 命令のエンコード

```
SWI{<cond>}  <immed_24>
```




```
BKPT  <immediate>
```



`SWI`と`BKPT`のいずれの場合も、命令のイミディエートフィールドはARMプロセッサに無視されます。SWI またはプリフェッチアボートハンドラは、必要に応じて、例外を発生させた命令をロードしてこれらのフィールドを抽出するように作成できます。これにより、オペレーティングシステムの呼び出しあるいはハンドラへのブレークポイントに関する情報のやり取りにそのフィールドを使用できます。

A3.14.2 例外生成命令の一覧

`BKPT` ブレークポイント。詳細については、P. A4-14 *「BKPT」*を参照して下さい。

`SWI` ソフトウェア割り込み。詳細については、P. A4-211 *「SWI」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.15 コプロセッサ命令

ARM 命令セットには、コプロセッサとの通信を目的とした命令が 3 種類存在します。これらの命令によって、次のような動作が実行できます。

- ARM プロセッサがコプロセッサにデータ処理動作を開始させる。
- ARM レジスタとコプロセッサレジスタとの間でデータを転送する。
- ARM プロセッサがコプロセッサのロード / ストア命令のアドレスを生成する。

命令セットは、プロセッサ命令ごとに 4 ビットのフィールドを使用して、最大 16 のコプロセッサを区別できます。このため、各プロセッサには固有の数値が割り当てられます。

—— 注 ——

大規模なコプロセッサ命令セットが必要な場合、この 16 の数値のうち複数を 1 つのコプロセッサに割り当てることができます。

———

コプロセッサは ARM と同じ命令ストリームを実行し、ARM 命令と、他のコプロセッサ用のコプロセッサ命令を無視します。コプロセッサハードウェアで実行不能なコプロセッサ命令については未定義命令例外が発生するため、コプロセッサのハードウェアをソフトウェアでエミュレートできます。

コプロセッサはある命令を部分的に実行してから、例外を発生させることができます。この機能は、ゼロによる除算やオーバフローのような、実行時に生成される例外の処理に役立ちます。ただし、その部分的な実行はコプロセッサの内部で行われ、ARM プロセッサからは見えません。ARM プロセッサに関する限り、この命令は実行開始時点で保留され、実行開始が許可されると例外を発生せずに完了します。命令を実行するか例外を発生させるかの判断は、ARM プロセッサが命令の実行開始を許可される前にコプロセッサ内で行われます。

コプロセッサ命令のすべてのフィールドがARMプロセッサによって使用されるわけではありません。コプロセッサレジスタ指定子およびオペコードは、個々のコプロセッサで定義されます。したがって、コプロセッサ命令については汎用命令ニーモニックのみが用意されています。アセンブラマクロを使用して、カスタムコプロセッサニーモニックをこれらの汎用ニーモニックに変換したり、オペコードを手動で再生成したりすることができます。

A3.15.1 例

```
CDP    p5, 2, c12, c10, c3, 4    ; Coproc 5 data operation
                                 ; opcode 1 = 2, opcode 2 = 4
                                 ; destination register is 12
                                 ; source registers are 10 and 3

MRC    p15, 5, R4, c0, c2, 3     ; Coproc 15 transfer to ARM register
                                 ; opcode 1 = 5, opcode 2 = 3
                                 ; ARM destination register = R4
                                 ; coproc source registers are 0 and 2

MCR    p14, 1, R7, c7, c12, 6    ; ARM register transfer to Coproc 14
                                 ; opcode 1 = 1, opcode 2 = 6
                                 ; ARM source register = R7
                                 ; coproc dest registers are 7 and 12
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

```
LDC     p6, CR1, [R4]           ; Load from memory to coprocessor 6
                                ; ARM register 4 contains the address
                                ; Load to CP reg 1

LDC     p6, CR4, [R2, #4]       ; Load from memory to coprocessor 6
                                ; ARM register R2 + 4 is the address
                                ; Load to CP reg 4

STC     p8, CR8, [R2, #4]!      ; Store from coprocessor 8 to memory
                                ; ARM register R2 + 4 is the address
                                ; after the transfer R2 = R2 + 4
                                ; Store from CP reg 8

STC     p8, CR9, [R2], #-16     ; Store from coprocessor 8 to memory
                                ; ARM register R2 holds the address
                                ; after the transfer R2 = R2 - 16
                                ; Store from CP reg 9
```

A3.15.2 コプロセッサ命令の一覧

`CDP` コプロセッサデータ操作。詳細については、P. A4-23「*CDP*」を参照して下さい。

`LDC` コプロセッサのレジスタへのロード。詳細については、P. A4-34「*LDC*」を参照して下さい。

`MCR` ARM レジスタからコプロセッサへのデータの移動。詳細については、P. A4-63「*MCR*」を参照して下さい。

`MCRR` 2 つの ARM レジスタからコプロセッサへのデータの移動。詳細については、P. A4-65「*MCRR*」を参照して下さい。

`MRC` コプロセッサから ARM レジスタへのデータの移動。詳細については、P. A4-71「*MRC*」を参照して下さい。

`MRRC` コプロセッサから 2 つの ARM レジスタへのデータの移動。詳細については、P. A4-73「*MRRC*」を参照して下さい。

`STC` コプロセッサレジスタの内容のストア。詳細については、P. A4-187「*STC*」を参照して下さい。

—— 注 ——

`MCRR` と `MRRC` は、ARMv5TE 以降でのみ使用できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.16 命令セットの拡張

ARM アーキテクチャは、各バージョンごとに各種の点で命令セットを拡張しています。このセクションでは、過去に拡張が行われ、今後も拡張が発生する可能性がある 6 つの領域について説明します。

- *メディア命令空間*：P. A3-33
- *乗算命令拡張空間*：P. A3-35
- *制御命令およびDSP 命令拡張空間*：P. A3-36
- *ロード / ストア命令拡張空間*：P. A3-38
- *アーキテクチャによる未定義命令空間*：P. A3-39
- *コプロセッサ命令拡張空間*：P. A3-40
- *無条件命令拡張空間*：P. A3-41

これらの領域に含まれ、意味がまだ割り当てられていない命令は、未定義または予測不能のいずれかです。判断には、次の規則を使用します。

1. 命令の*デコードビット*は、ビット [27:20] とビット [7:4] に定義されています。

   ARMv5 以降では、ビット [31:28] を AND した結果もデコードビットの一つです。このビットによって条件フィールドが 0b1111 かどうかが判断されます。これは ARMv5 以降で、無条件でのみ実行される各種命令のエンコードに使用されます。詳細については、P. A3-4 *「条件コード 0b1111」*と P. A3-41 *「無条件命令拡張空間」*を参照して下さい。

2. 命令のデコードビットが定義済み命令のデコードビットと同じで、命令全体が定義済み命令ではない場合、その命令は予測不能です。

   たとえば、次の命令を考えます。

   - ビット [31:28] が 0b1111 と等しくない。
   - ビット [27:20] が 0b00010000 と等しい。
   - ビット [7:4] が 0b0000 と等しい。

   ただし、

   - 命令のビット [11] が 1 である。

   この場合、命令は制御命令拡張空間にあり、`MRS` 命令と同じデコードビットですが、`MRS` 命令のビット [11] は常にゼロであるため、この命令は有効な `MRS` 命令ではありません。上記の規則により、この命令は予測不能です。

3. 命令のデコードビットがどの定義済み命令のデコードビットとも等しくない場合、その命令は未定義です。

上記の規則 2 と規則 3 は各 ARM アーキテクチャバージョンに個別に適用されます。その結果、命令の状態は、アーキテクチャバージョン間で異なる場合があります。通常、初期のアーキテクチャバージョンで予測不能または未定義だった命令がその後のバージョンで定義済み命令になった場合にこのケースが発生します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.16.1 メディア命令空間

次のオペコードの命令は、メディア命令空間内に属します。

```
opcode[27:25] = 0b011
opcode[4] = 1
```

31–28	27 26 25	24–5	4	3–0
cond	0 1 1	op x x x x x x x x x x x x x x x x x x	1	x x x x

メディア命令空間にある未割り当て命令は、ARM アーキテクチャのすべてのバージョンで未定義です。

この領域に現在割り当てられている命令の一覧を表 A3-3 に示します。

表 A3-3 メディア命令空間

命令	アーキテクチャバージョン
並列加算、減算、加算と減算。詳細については、P. A3-14 *「並列加算命令と並列減算命令」*を参照して下さい。	ARMv6 以降
PKH、SSAT、SSAT16、USAT、USAT16、SEL 符号 / ゼロ拡張および加算命令にも使用されます。詳細については、P. A3-16 *「拡張命令」*を参照して下さい。	ARMv6 以降
SMLAD、SMLSD、SMLALD、SMUAD、SMUSD	ARMv6 以降
USAD8、USADA8	ARMv6 以降

これらの命令の詳細については、P. A3-34 図 A3-2 を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

<table>
<tr><th></th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>並列加算/減算</td><td colspan="4">cond</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td colspan="3">opc1</td><td colspan="4">Rn</td><td colspan="4">Rd</td><td colspan="4">SBO</td><td colspan="3">opc2</td><td>1</td><td colspan="4">Rm</td></tr>
<tr><td>ハーフワードのパック</td><td colspan="4">cond</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td colspan="4">Rn</td><td colspan="4">Rd</td><td colspan="5">shift_imm</td><td>op</td><td>0</td><td>1</td><td colspan="4">Rm</td></tr>
<tr><td>ワード飽和</td><td colspan="4">cond</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>U</td><td>1</td><td colspan="5">sat_imm</td><td colspan="4">Rd</td><td colspan="5">shift_imm</td><td>sh</td><td>0</td><td>1</td><td colspan="4">Rm</td></tr>
<tr><td>並列ハーフワード飽和</td><td colspan="4">cond</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>U</td><td>1</td><td>0</td><td colspan="4">sat_imm</td><td colspan="4">Rd</td><td colspan="4">SBO</td><td>0</td><td>0</td><td>1</td><td>1</td><td colspan="4">Rm</td></tr>
<tr><td>バイトの選択</td><td colspan="4">cond</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td colspan="4">Rn</td><td colspan="4">Rd</td><td colspan="4">SBO</td><td>1</td><td>0</td><td>1</td><td>1</td><td colspan="4">Rm</td></tr>
<tr><td>符号/ゼロ拡張（加算）</td><td colspan="4">cond</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td colspan="3">op</td><td colspan="4">Rn</td><td colspan="4">Rd</td><td colspan="2">ローテート</td><td colspan="2">SBZ</td><td>0</td><td>1</td><td>1</td><td>1</td><td colspan="4">Rm</td></tr>
<tr><td>乗算（タイプ3）</td><td colspan="4">cond</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td colspan="3">opc1</td><td colspan="4">Rd/RdHi</td><td colspan="4">Rn/RdLo</td><td colspan="4">Rs</td><td colspan="3">opc2</td><td>1</td><td colspan="4">Rm</td></tr>
<tr><td>符号なし絶対値の差の合計</td><td colspan="4">cond</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td colspan="4">Rd</td><td colspan="4">Rn*</td><td colspan="4">Rs</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="4">Rm</td></tr>
<tr><td>符号なし絶対値の差の合計、累算付き</td><td colspan="4">cond</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td colspan="4">Rd</td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="4">Rs</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="4">Rm</td></tr>
</table>

図 A3-2 メディア命令

Rn* Rn != R15

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.16.2 乗算命令拡張空間

次のオペコードの命令は、乗算命令拡張空間に属します。

```
opcode[27:24]  == 0b0000
opcode[7:4]    == 0b1001
opcode[31:28]  != 0b1111  /* Only required for version 5 and above */
```

ここに示すフィールド名は、実装を簡素化するために推奨されるガイドラインです。

31-28	27	26	25	24	23-20	19-16	15-12	11-8	7	6	5	4	3-0
cond	0	0	0	0	op1	Rn	Rd	Rs	1	0	0	1	Rm

この領域に現在割り当てられている命令の一覧を表 A3-4 に示します。

表 A3-4 乗算命令拡張空間

命令	アーキテクチャバージョン
MUL、MULS、MLA、MLAS	すべて
UMULL、UMULLS、UMLAL、UMLALS、SMULL、SMULLS、SMLAL、SMLALS	すべて
UMAAL	ARMv6 以降

これらの命令の詳細については、図 A3-3 を参照して下さい。

	31-28	27-24	23	22	21	20	19-16	15-12	11-8	7-4	3-0
乗算（累算）	cond	0 0 0 0	0	0	A	S	Rd	Rn	Rs	1 0 0 1	Rm
符号なし積和累算ロング	cond	0 0 0 0	0	1	0	0	RdHi	RdLo	Rs	1 0 0 1	Rm
乗算（累算）ロング	cond	0 0 0 0	1	Un	A	S	RdHi	RdLo	Rs	1 0 0 1	Rm

図 A3-3 乗算命令

A 累算
Un 1 = 符号なし、0 = 符号付き
S ステータスレジスタ更新（SPSR => CPSR）

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.16.3 制御命令および DSP 命令拡張空間

次のオペコードの命令は、制御命令空間に属します。

```
opcode[27:26] == 0b00
opcode[24:23] == 0b10
opcode[20]    == 0
opcode[31:28] != 0b1111  /* Only required for version 5 and above */
```

ただし、次の場合を除きます。

```
opcode[25] == 0
opcode[7]  == 1
opcode[4]  == 1
```

ここに示すフィールド名は、実装を簡素化するために推奨されるガイドラインです。

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6–5	4	3–0
cond	0	0	0	1	0	op1		0	Rn	Rd	Rs	op2		0	Rm
cond	0	0	0	1	0	op1		0	Rn	Rd	Rs	0	op2	1	Rm
cond	0	0	1	1	0	R	1	0	Rn	Rd	rotate_imm	immed_8			

この領域に現在割り当てられている命令の一覧を表 A3-5 に示します。

表 A3-5 制御および DSP 拡張空間の命令

命令	アーキテクチャバージョン
MRS	すべて
MSR（レジスタ形式）	すべて
BX	ARMv5 以降および ARMv4 の T バリエーション
CLZ	ARMv5 以降
BXJ	ARMv5EJ 以降
BLX（レジスタ形式）	ARMv5 以降
QADD	ARMv5 以降の E バリエーション
QSUB	ARMv5 以降の E バリエーション
QDADD	ARMv5 以降の E バリエーション

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 A3-5 制御および DSP 拡張空間の命令（続き）

命令	アーキテクチャバージョン
QDSUB	ARMv5 以降の E バリエーション
BKPT	ARMv5 以降
SMLA<x><y>	ARMv5 以降の E バリエーション
SMLAW<y>	ARMv5 以降の E バリエーション
SMULW<y>	ARMv5 以降の E バリエーション
SMLAL<x><y>	ARMv5 以降の E バリエーション
SMUL<x><y>	ARMv5 以降の E バリエーション
MSR（イミディエート形式）	すべて

これらの命令の詳細を図 A3-4 に示します。

	31 30 29 28	27 26 25 24 23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
ステータスレジスタをレジスタに移動	cond	0 0 I 1 0	R	0	0	SBO	Rd	SBZ	0 0 0 0	SBZ
レジスタをステータスレジスタに移動	cond	0 0 0 1 0	R	1	0	マスク	SBO	SBZ	0 0 0 0	Rm
Thumb分岐/交換命令セット	cond	0 0 0 1 0	0	1	0	SBO	SBO	SBO	0 0 0 1	Rm
Java分岐/交換命令セット	cond	0 0 0 1 0	0	1	0	SBO	SBO	SBO	0 0 1 0	Rm
先行ゼロカウント	cond	0 0 0 1 0	1	1	0	SBO	Rd	SBO	0 0 0 1	Rm
Thumbリンク付き分岐/交換命令セット	cond	0 0 0 1 0	0	1	0	SBO	SBO	SBO	0 0 1 1	Rm
飽和加算/減算	cond	0 0 0 1 0	op		0	Rn	Rd	SBZ	0 1 0 1	Rm
ソフトウェアブレークポイント	cond	0 0 0 1 0	0	1	0	immed			0 1 1 1	immed
符号付き乗算（タイプ2）	cond	0 0 0 1 0	op		0	Rd	Rn	Rs	1 y x 0	Rm

図 A3-4 その他の命令

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.16.4 ロード / ストア命令拡張空間

次のオペコードの命令は、ロード / ストア命令拡張空間に属します。

```
opcode[27:25] == 0b000
opcode[7]     == 1
opcode[4]     == 1
opcode[31:28] != 0b1111 /* Only required for version 5 and above */
```

ただし、次の場合を除きます。

```
opcode[24]   == 0
opcode[6:5]  == 0
```

ここに示すフィールド名は、実装を簡素化するために推奨されるガイドラインです。

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6–5	4	3–0
cond	0	0	0	P	U	B	W	L	Rn	Rd	Rs	1	op1	1	Rm

この領域に現在割り当てられている命令の一覧を表 A3-6 に示します。

表 A3-6 ロード / ストア命令

命令	アーキテクチャバージョン
SWP/SWPB	すべて（ARMv6 では推奨されません）
LDREX	ARMv6 以降
STREX	ARMv6 以降
STRH	すべて
LDRD	ARMv5 以降の E バリエーション（ARMv5TExP 以外）
STRD	ARMv5 以降の E バリエーション（ARMv5TExP 以外）
LDRH	すべて
LDRSB	すべて
LDRSH	すべて

これらの拡張ロード / ストア命令の詳細については、P. A3-39 図 A3-5 を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

	31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
スワップ/スワップバイト	cond	0 0 0	1	0	B	0	0	Rn	Rd	SBZ	1	0	0	1	Rm
レジスタ排他ロード/ストア	cond	0 0 0	1	1	0	0	L	Rn	Rd	SBO	1	0	0	1	SBO
レジスタオフセットでのハーフワードのロード/ストア	cond	0 0 0	P	U	0	W	L	Rn	Rd	SBZ	1	0	1	1	Rm
イミディエートオフセットでのハーフワードのロード/ストア	cond	0 0 0	P	U	1	W	L	Rn	Rd	HiOffset	1	0	1	1	LoOffset
イミディエートオフセットでの符号付きハーフワード/バイトのロード	cond	0 0 0	P	U	1	W	1	Rn	Rd	HiOffset	1	1	H	1	LoOffset
レジスタオフセットでの符号付きハーフワード/バイトのロード	cond	0 0 0	P	U	0	W	1	Rn	Rd	SBZ	1	1	H	1	Rm
レジスタオフセットでのダブルワードのロード/ストア	cond	0 0 0	P	U	0	W	0	Rn	Rd	SBZ	1	1	St	1	Rm
イミディエートオフセットでのダブルワードのロード/ストア	cond	0 0 0	P	U	1	W	0	Rn	Rd	HiOffset	1	1	St	1	LoOffset

図 A3-5 拡張ロード / ストア命令

B 1 = バイト、0 = ワード

P、U、I、W アドレスモード用のプリインデクス / ポストインデクスまたはオフセット、アップ / ダウン、イミディエート / レジスタオフセット、ライトバックフィールド。詳細については、第 A5 章「*ARM アドレシングモード*」を参照して下さい。

L 1 = ロード、0 = ストア

H 1 = ハーフワード、0 = バイト

St 1 = ストア、0 = ロード

A3.16.5 アーキテクチャによる未定義命令空間

一般に、未定義命令は将来の ARM 命令セットの拡張に使用されます。ただし、次のエンコードを持つ命令は、将来の拡張を意図したものではありません。

31 28	27 26 25 24 23 22 21 20	19 8	7 6 5 4	3 2 1 0
cond	0 1 1 1 1 1 1 1	x x x x x x x x x x x x	1 1 1 1	x x x x

プログラマがソフトウェア用途で未定義命令を使用する場合、将来のハードウェアでそれらの命令が定義済み命令として扱われるリスクを最小限にするため、このエンコードを持つ命令のいずれかを使用して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.16.6 コプロセッサ命令拡張空間

次のオペコードの命令は、コプロセッサ命令拡張空間に属します。

```
opcode[27:23]  == 0b11000
opcode[21]     == 0
```

ここに示すフィールド名は、実装を簡素化するために推奨されるガイドラインです。

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7–0
cond	1	1	0	0	0	x	0	x	Rn	CRd	cp_num	オフセット

ARMv4 のすべてのバリエーションおよび ARMv5 の E 以外のバリエーションでは、コプロセッサ命令拡張空間のすべての命令は未定義です。ARM プロセッサでこのことを実現する方法は実装定義です。オプションは次のとおりです。

- ARM プロセッサが直接未定義命令例外を発生させる。
- ARM プロセッサが、接続されているコプロセッサでその命令に応答しないことを確認する。この場合、未定義命令例外が発生します。詳細については、P. A2-19「*未定義命令例外*」を参照して下さい。

ARMv5 の E バリエーション以降は、コプロセッサ命令拡張空間の命令は次のように扱われます。

- ビット [22] == 0 の命令は未定義で、E 以外のバリエーションの場合とまったく同じに処理されます。
- ビット [22] ==1 の命令は MCRR および MRRC 命令となります。詳細については、P. A4-65「*MCRR*」と P. A4-73「*MRRC*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A3.16.7 無条件命令拡張空間

ARMv5 以降では、次のオペコードの命令は無条件命令空間に属します。

```
opcode[31:28] == 0b1111
```

31 30 29 28	27 ... 20	19 ... 8	7 ... 4	3 ... 0
1 1 1 1	opcode1	x x x x x x x x x x x x	opcode2	x x x x

この領域に現在割り当てられている命令の一覧を表 A3-7 に示します。

表 A3-7 無条件命令拡張空間

命令	アーキテクチャバージョン
CPS/SETEND	ARMv6 以降
PLD	ARMv5 以上の E バリエーション（ARMv5TExP 以外）
RFE	ARMv6
SRS	ARMv6
BLX（アドレス形式）	ARMv5 以上
MCRR2	ARMv6 以降
MRRC2	ARMv6 以降
STC2	ARMv5 以降
LDC2	ARMv5 以降
CDP2	ARMv5 以降
MCR2	ARMv5 以降
MRC2	ARMv5 以降

無条件命令の詳細については、P. A3-42 図 A3-6 を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

プロセッサ状態の変更	1 1 1 1	0 0 0 1 0 0 0 0	imod	M	0	SBZ	A	I	F	0	mode
エンディアン形式の設定	1 1 1 1	0 0 0 1 0 0 0 0	0 0 0	1	SBZ	E	SBZ	0 0 0 0	SBZ		
キャッシュのプリロード	1 1 1 1	0 1	X	1	U	1 0 1	Rn	1 1 1 1	addr_mode		
復帰状態の保存	1 1 1 1	1 0 0	P	U	1	W	0	1 1 0 1	SBZ	0 1 0 1	SBZ	mode
例外からの復帰	1 1 1 1	1 0 0	P	U	0	W	1	Rn	SBZ	1 0 1 0	SBZ
リンク付き分岐およびThumbへの変更	1 1 1 1	1 0 1	H	24ビットオフセット							
追加コプロセッサ2つのレジスタの転送	1 1 1 1	1 1 0 0 0 1 0	L	Rn	Rd	cp_num	opcode	CRm			
追加コプロセッサレジスタ転送	1 1 1 1	1 1 1 0	opc1	L	CRn	Rd	cp_num	opc2	1	CRm	
未定義命令	1 1 1 1	1 1 1 1	x x x x x x x x x x x x x x x x x x x x x x x x								

図 A3-6 無条件命令

M mmod

X アドレシングモード 2 では、X = 0 の場合はイミディエートのオフセット / インデクス、X = 1 の場合はレジスタベースのオフセット / インデクスを意味します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 A4 章
ARM 命令

本章では、すべての ARM® 命令の構文と用法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。

- *ARM 命令のアルファベット順リスト*：P. A4-2
- *ARM 命令とアーキテクチャのバージョン*：P. A4-287

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1 ARM 命令のアルファベット順リスト

以下のページには、すべての ARM 命令についての説明が含まれています。各命令の説明内容は次のとおりです。

- 命令のエンコード
- 命令の構文
- 命令が有効な ARM アーキテクチャのバージョン
- 適用されるすべての例外
- 命令の動作を示す擬似コードの例
- 用法と特殊なケースについての注記

A4.1.1 一般的な注意点

これらの注記では、命令のページで使用される情報のタイプと省略形について説明します。

アドレシングモード

多くの命令は、第 A5 章「*ARM アドレシングモード*」で説明されているアドレシングモードのいずれかを参照しています。参照されるアドレシングモードの説明は、命令の説明の本質的な部分と考えて下さい。

以下の点は特に重要です。

- アドレシングモードのエンコード図とアセンブラの構文により、命令のエンコード図とアセンブラの構文からは得られない、より多くの詳細を知ることができます。
- アドレシングモードの演算擬似コードは、命令の擬似コードで使用される値を計算し、場合によっては、命令の付加的な影響も示します。
- アドレシングモードの使用上の注意、オペランドの制約条件、その他の注意はすべて、命令に適用されます。

構文の省略形

命令のページでは、以下の省略形を使用しています。

`immed_`*`n`* イミディエート値を示しています。*n* はビット数です。たとえば、8 ビットのイミディエート値は次のように示されます。

`immed_8`

`offset_`*`n`* オフセット値を示しています。*n* はビット数です。たとえば、8 ビットのオフセット値は次のように示されます。

`offset_8`

符号付きオフセットについても同じ構文が使用されます。たとえば、8 ビットの符号付きオフセットは次のように示されます。

`signed_offset_8`

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

エンコード図とアセンブラの構文

使用されている規則については、P. xxii *「アセンブラ構文の記述」*を参照して下さい。

アーキテクチャバージョン

命令が有効なアーキテクチャバージョンの詳細を示します。アーキテクチャバージョンの詳細については、P. xiii *「アーキテクチャのバージョンとバリエーション」*を参照して下さい。

例外

命令の実行中に発生する可能性がある例外の詳細を示します。一般に、プリフェッチアボートは記載されていません。理由は、この例外はどの命令でも発生しうること、および命令のフェッチ中にアボートが発生しても命令のビットパターンが不明なためです。ただし、`BKPT` に対するプリフェッチアボート例外は、この考慮点が適用されないため、記載されています。

動作

これは、命令の実行内容を擬似コードで表現したものです。擬似コードで使用されている表記規則の詳細については、P. xxi *「命令の擬似コード表記」*を参照して下さい。

使用に関する情報

このセクションは、命令を効果的に使用する方法に関するヒントやその他の情報を提供するのに適当な場所に記載されています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.2 ADC

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	0	I	0	1	0	1	S	Rn	Rd	shifter_operand

ADC（キャリー付き加算）は、2 つの値とキャリーフラグを加算します。最初の値はレジスタに含まれています。2 番目の値はイミディエート値またはレジスタの値で、加算を実行する前にシフト可能です。

ADC 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
ADC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S 命令の S ビット（ビット [20]）に 1 をセットし、この命令で CPSR を更新することを指定します。S を省略すると S ビットは 0 にセットされ、CPSR は命令によって変更されません。S が指定された場合の CPSR の更新には、2 タイプあります。

- <Rd> が R15 でない場合、N フラグと Z フラグが加算の結果に従ってセットされます。また、追加によってキャリー（符号なしオーバフロー）と符号付きオーバフローが生成されたかどうかによって、それぞれ C フラグと V フラグがセットされます。CPSR の他のビットは変更されません。
- <Rd> が R15 の場合、現在のモードの SPSR が CPSR にコピーされます。ユーザモードまたはシステムモードには SPSR がないため、これらのモードで実行した場合、この形式の命令は予測不能な結果を招きます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<shifter_operand>

第 2 オペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」* を参照して下さい。

I ビットが 0 で、shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は ADC 命令ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-32 *「命令セットの拡張」* を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャに存在します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = Rn + shifter_operand + C Flag
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + shifter_operand + C Flag)
        V Flag = OverflowFrom(Rn + shifter_operand + C Flag)
```

用法

ADCは、複数ワードの加算の合成に使用します。R0とR1およびR2とR3の各レジスタペアに64ビットの値が保持されている場合（R0 と R2 に最下位ワードが保持されているとします)、次の命令シーケンスを実行すると、64 ビットの加算結果が R4 と R5 に格納されます。

```
ADDS R4,R0,R2
ADC  R5,R1,R3
```

ここで、2 番目の命令が次のものから

```
ADC  R5,R1,R3
```

次のように変更された場合

```
ADCS R5,R1,R3
```

実行の結果セットされるフラグの値は次の意味を持ちます。

N	64 ビットの加算により、負の値が生成されました。
C	符号なしのオーバフローが発生しました。
V	符号付きのオーバフローが発生しました。
Z	最上位の 32 ビットがすべて 0 です。

次の命令では、拡張操作（キャリーフラグを経由した 33 ビットのロテート）を伴う 1 ビットの左ローテートが行われます。

```
ADCS R0,R0,R0
```

右に同様のローテートを行う方法の詳細については、P. A5-17 *「データ処理オペランド - 拡張付き右ローテート」* を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.3 ADD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	0	I	0	1	0	0	S	Rn	Rd	シフタオペランド

ADD は、2 つの値を加算します。最初の値はレジスタに含まれています。2 番目の値はイミディエート値またはレジスタの値で、加算を実行する前にシフト可能です。

ADD 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
ADD{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。詳細については、P. A3-3 *「条件フィールド」*を参照して下さい。<cond> が省略されている場合、AL（常時）条件が使用されます。

S 命令の S ビット（ビット [20]）に 1 をセットし、この命令で CPSR を更新することを指定します。S を省略すると S ビットは 0 にセットされ、CPSR は命令によって変更されません。S が指定された場合の CPSR の更新には、2 タイプあります。

- <Rd> が R15 でない場合、N フラグと Z フラグが加算の結果に従ってセットされます。また、加算によってキャリー（符号なしオーバフロー）と符号付きオーバフローが生成されたかどうかによって、それぞれ C フラグと V フラグがセットされます。CPSR の他のビットは変更されません。
- <Rd> が R15 の場合、現在のモードの SPSR が CPSR にコピーされます。ユーザモードまたはシステムモードには SPSR がないため、これらのモードで実行した場合、この形式の命令は予測不能な結果を招きます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<shifter_operand>

第 2 オペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」*を参照して下さい。

I ビットが 0 で、shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は ADD 命令ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-32 *「命令セットの拡張」*を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャに存在します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = Rn + shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + shifter_operand)
        V Flag = OverflowFrom(Rn + shifter_operand)
```

用法

ADD は 2 つの値の加算に使用します。

Rx の値をインクリメントするには、次の命令を使用します。

```
ADD Rx, Rx, #1
```

次の命令は、Rx を $2^n + 1$ 倍に累乗（定数）算して、Rd に格納します。

```
ADD Rd, Rx, Rx, LSL #n
```

PC 相対アドレスを作成するには、次の命令を使用します。

```
ADD Rd, PC, #offset
```

ここで、offset は求めるアドレスと PC 内に保持されているアドレスの差で、PC（プログラムカウンタ）には ADD 命令自身のアドレスに 8 バイトを加えたアドレスが保持されています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.4 AND

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	0 0 0 0	S	Rn	Rd	shifter_operand

AND は 2 つの値のビット単位の論理積を実行します。最初の値はレジスタに含まれています。2 番目の値はイミディエート値またはレジスタの値で、論理積を実行する前にシフト可能です。

AND 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
AND{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

`<cond>` この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

`S` 命令の S ビット（ビット [20]）に 1 をセットし、この命令で CPSR を更新することを指定します。S を省略すると S ビットは 0 にセットされ、CPSR は命令によって変更されません。S が指定された場合の CPSR の更新には、2 タイプあります。

- <Rd> が R15 でない場合、N フラグと Z フラグが演算の結果によってセットされ、シフタによって生成されたキャリーの出力ビットに C フラグがセットされます（P. A5-2*「アドレシングモード 1 - データ処理オペランド」*参照）。V フラグと CPSR の他のビットは変更されません。
- <Rd> が R15 の場合、現在のモードの SPSR が CPSR にコピーされます。ユーザモードまたはシステムモードには SPSR がないため、これらのモードで実行した場合、この形式の命令は予測不能な結果を招きます。

`<Rd>` デスティネーションレジスタ。

`<Rn>` 最初のオペランドを含むレジスタ。

`<shifter_operand>`

第 2 オペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2*「アドレシングモード 1 - データ処理オペランド」*を参照して下さい。

I ビットが 0 で、shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は AND 命令ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-32*「命令セットの拡張」*を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャに存在します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

操作

```
if ConditionPassed(cond) then
    Rd = Rn AND shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

用法

AND はレジスタから一部のフィールドを抽出するために最も有効で、レジスタに対し、抽出する部分を 1 に、他の部分を 0 にセットしたマスク値の論理積を加えることにより抽出できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.5 B, BL

31–28	27	26	25	24	23–0
cond	1	0	1	L	signed_immed_24

B（分岐）および BL（リンク付き分岐）は目的アドレスに分岐し、条件付きまたは無条件でプログラムのフローを変更します。

BL は、分岐と同時に復帰アドレスをリンクレジスタ R14（LR とも呼ばれます）に格納します。

構文

```
B{L}{<cond>}  <target_address>
```

各項目の説明については以下を参照して下さい。

L　命令の L ビット（ビット 24）に 1 をセットします。このフラグがセットされた命令では、復帰アドレスがリンクレジスタ（R14）に格納されます。L が省略された場合、L ビットは 0 にセットされ、命令は復帰アドレスを格納せず、単に分岐を発生させます。

<cond>　この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<target_address>

分岐先のアドレスを指定します。分岐先アドレスは、次のように計算されます。

1. 24 ビットの符号付き(2 の補数)イミディエート値を 30 ビットに符号拡張します。
2. 結果を左に 2 ビットシフトして、32 ビットの値を生成します。
3. 結果を PC の内容に加算します (PC には、分岐命令アドレス + 8 バイトが含まれています)。

したがって、命令で指定できる分岐範囲は約 ±32MB です。正確な範囲については P. A4-11 *「用法」*を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャに存在します。

例外

なし。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    if L == 1 then
        LR = address of the instruction after the branch instruction
    PC = PC + (SignExtend_30(signed_immed_24) << 2)
```

用法

BL はサブルーチンコールに使用します。サブルーチンからの復帰には、R14 を PC にコピーします。これは通常、次のいずれかの方法で実行されます。

- BX R14 命令を実行する（この命令をサポートしているアーキテクチャのバージョンの場合）。
- MOV PC,R14 命令を実行する。
- 次の形式の命令を使用して、レジスタのグループと R14 をサブルーチンエントリのスタックに格納する。

  ```
  STMFD R13!,{<registers>,R14}
  ```

  復帰時には、次の形式の命令でレジスタの値を元に戻して復帰します。

  ```
  LDMFD R13!,{<registers>,PC}
  ```

signed_immed_24 の正しい値を計算するには、アセンブラ（または他のツールキットコンポーネント）が次の動作を実行する必要があります。

1. 分岐命令のベースアドレスを計算します。これは命令のアドレス + 8 です。つまり、このベースアドレスは命令で使用される PC 値に等しくなります。
2. ターゲットアドレスからベースアドレスを減算して、バイトオフセットを生成します。すべての ARM 命令はワード境界アライメントのため、このオフセットは常に 4 の倍数です。
3. バイトオフセットが –33554432 から +33554428 の範囲外の場合は、別のコード生成手法を使用するか、エラーを生成します。
4. それ以外の場合、命令の signed_immed_24 フィールドをバイトオフセットのビット [25:2] に設定します。

注

メモリバウンド　　アドレス 0 よりも前への分岐と、32 ビットアドレス空間の最後を越えた分岐の結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.6 BIC

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	1	1	1	0	S	Rn	Rd	shifter_operand

`BIC`（ビットクリア）は、最初の値と 2 番目の値の補数に対して、ビット単位の論理積を実行します。最初の値はレジスタに含まれています。2 番目の値はイミディエート値またはレジスタの値で、BIC 演算を実行する前にシフト可能です。

`BIC` 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
BIC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

`<cond>` この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。If `<cond>` が省略されている場合、`AL`（常時）条件が使用されます。

`S` 命令の S ビット（ビット [20]）に 1 をセットし、この命令で CPSR を更新することを指定します。`S` を省略すると S ビットは 0 にセットされ、CPSR は命令によって変更されません。`S` が指定された場合の CPSR の更新には、2 タイプあります。

- `<Rd>` が R15 でない場合、N フラグと Z フラグが演算の結果によってセットされ、シフタによって生成されたキャリーの出力ビットに C フラグがセットされます（P. A5-2 *「アドレシングモード 1 - データ処理オペランド」* 参照）。V フラグと CPSR の他のビットは変更されません。
- `<Rd>` が R15 の場合、現在のモードの SPSR が CPSR にコピーされます。ユーザモードまたはシステムモードには SPSR がないため、これらのモードで実行した場合、この形式の命令は予測不能な結果を招きます。

`<Rd>` デスティネーションレジスタ。

`<Rn>` 最初のオペランドを含むレジスタ。

`<shifter_operand>`

第 2 オペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」* を参照して下さい。

I ビットが 0 で、shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は `BIC` 命令ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-32 *「命令セットの拡張」* を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャに存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = Rn AND NOT shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

用法

BIC は、レジスタで選択したビットをクリアするために使用します。各ビットごとに、BIC で 1 が代入されているとビットがクリアされ、BIC で 0 が代入されているとビットは変更されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.7 BKPT

31–28	27	26	25	24	23	22	21	20	19–8	7–4	3–0
1 1 1 0	0	0	0	1	0	0	1	0	immed	0 1 1 1	immed

BKPT（ブレークポイント）は、ソフトウェアのブレークポイントを発生させます。このブレークポイントは、プリフェッチアボートベクタにインストールされている例外ハンドラにより処理されます。デバッグハードウェアが含まれている実装では、オプションとしてハードウェアがこの動作をオーバライドし、自身でブレークポイントの処理を行うこともできます。この場合、プリフェッチアボート例外のコンテキストがデバッガに表示されます。

構文

```
BKPT  <immed_16>
```

各項目の説明については以下を参照して下さい。

<immed_16> 16 ビットのイミディエート値。<immed_16> の上位 12 ビットは命令のビット [19:8] に配置され、下位 4 ビットは命令のビット [3:0] に配置されます。この値は ARM ハードウェアにより無視されますが、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追加情報を格納できます。

アーキテクチャのバージョン

バージョン 5 以降に存在します。

例外

プリフェッチアボート

動作

```
if (not overridden by debug hardware)
    R14_abt   = address of BKPT instruction + 4
    SPSR_abt  = CPSR
    CPSR[4:0] = 0b10111             /* Enter Abort mode */
    CPSR[5]   = 0                   /* Execute in ARM state */
    /* CPSR[6] is unchanged */
    CPSR[7]   = 1                   /* Disable normal interrupts */
    if high vectors configured then
        PC    = 0xFFFF000C
    else
        PC    = 0x0000000C
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

BKPT の正確な使用方法は、使用されているデバッグシステムによって異なります。デバッグシステムでは、以下の 2 通りの方法で BKPT 命令を使用することができます。

- モニタデバッグモードデバッグハードウェア（ARMv6 以前のオプション）は、BKPT 命令の正常な動作をオーバライドしないため、プリフェッチアボートベクタへのエントリが行われます。デバッグイベントを示すようIFSRを更新すると、他のシステムプリフェッチアボートからBKPT 命令を実行することで、ソフトウェア側でデバッグイベントを識別することができます。

  この方法を行う場合、R14_abt と SPSR_abt が破壊されることを防ぐため、アボートハンドラ内で BKPT 命令を発生させないようにする必要があります。同じ理由から、FIQ ハンドラ内でも避ける必要があります。これは、FIQ 割り込みはアボートハンドラ内で発生する可能性があるためです。

- ホールトデバッグモードデバッグハードウェアは、BKPT 命令の通常の動作をオーバライドし、ソフトウェアブレークポイントを自分で処理します。処理が完了すると、通常は BKPT 命令の次の命令から実行を再開するか、メモリ内の BKPT 命令を他の命令に置き換えてその命令から実行を再開します。

  BKPTがこの方法で使用される場合、R14_abt とSPSR_abt は破壊されないため、アボートとFIQ ハンドラの使用に関して上で説明した制限は適用されません。

注

条件フィールド　BKPT は無条件命令です。命令のビット [31:28] が AL（常時）条件以外の有効な条件をエンコードする場合、この命令の結果は予測不能です。

ハードウェアオーバーライド

実装のデバッグハードウェアは、BKPT 命令の通常の動作をオーバライドすることが許されます。このため、ソフトウェアはこの命令を、使用されているデバッグシステム（存在する場合）のドキュメントに記載されている以外の目的に使用できません。特に、ソフトウェアはプリフェッチアボート例外の発生に依存できません。ただし、システムにデバッグハードウェアが存在しないことが保証されている場合と、デバッグシステムによって発生することが指定されている場合は除きます。

詳細については、使用しているデバッグシステムのドキュメントを参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.8 BLX (1)

31	30	29	28	27	26	25	24	23 ... 0
1	1	1	1	1	0	1	H	signed_immed_24

BLX (1)（リンク付き分岐と状態遷移）は、ARM 命令セットから、命令で指定されたアドレスの Thumb® サブルーチンを呼び出します。

この形式の BLX 命令は無条件命令（必ずプログラムフローの変更が発生する）であり、分岐命令の次にある命令のアドレスがリンクレジスタ（R14）に保存されます。Thumb 命令の実行は、分岐先アドレスで開始されます。

構文

```
BLX  <target_addr>
```

各項目の説明については以下を参照して下さい。

`<target_addr>` 分岐先の Thumb 命令のアドレスを指定します。分岐先アドレスは、次のように計算されます。

1. 24 ビットの符号付き（2 の補数）イミディエート値を 30 ビットに符号拡張します。
2. 結果を左に 2 ビットシフトして、32 ビットの値を生成します。
3. 手順 2 を行った結果のビット [1] を H ビットにセットします。
4. 手順 3 の結果を PC の内容に加算します（PC には分岐命令のアドレス + 8 が含まれています）。

したがって、命令で指定できる分岐範囲は約 ±32MB です。正確な範囲については P. A4-17*「用法」*を参照して下さい。

アーキテクチャのバージョン

バージョン 5 以降に存在します。T バリエーション以外での動作の詳細については、P. A2-15*「T ビットと J ビット」*を参照して下さい。

例外

なし。

動作

```
LR = address of the instruction after the BLX instruction
CPSR T bit = 1
PC = PC + (SignExtend(signed_immed_24) << 2) + (H << 1)
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

BLXを介して呼び出されたThumbサブルーチンからARM呼び出し元へ復帰するには、次のThumb命令を使用します。

```
BX     R14
```

この命令は、P. A7-32「*BX*」の説明どおりに使用するか、以下のサブルーチンエントリで使用します。

```
PUSH {<registers>,R14}
```

上の命令を使用した場合、復帰時には次の命令を使用します。

```
POP  {<registers>,PC}
```

signed_immed_24 の正しい値を計算するには、アセンブラ（または他のツールキットコンポーネント）が次の動作を実行する必要があります。

1. 分岐命令のベースアドレスを計算します。これは命令のアドレス + 8 です。つまり、このベースアドレスは命令で使用される PC 値に等しくなります。
2. ターゲットアドレスからベースアドレスを減算して、バイトオフセットを生成します。ARM 命令はすべてワード境界アライメントで、Thumb 命令はすべてハーフワード境界アライメントなため、このオフセットは常に偶数です。
3. バイトオフセットが –33554432 から +33554430 の範囲外の場合は、別のコード生成手法を使用するか、エラーを生成します。
4. それ以外の場合、命令の signed_immed_24 フィールドをバイトオフセットのビット [25:2] に、H ビットをバイトオフセットのビット [1] に設定します。

注

条件　　他の大部分の ARM 命令とは異なり、この命令は実行付きで実行できません。

ビット [24]　　このビットは、ターゲットアドレスのビット [1] として使用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.9 BLX (2)

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
cond	0	0	0	1	0	0	1	0	SBO	SBO	SBO	0	0	1	1	Rm

BLX (2)はThumb命令セットから、レジスタで指定されたアドレスのARMまたはThumbサブルーチンを呼び出します。

CPSR の T ビットを Rm のビット [0] にセットします。これによって、サブルーチン内で使用される命令セットが選択されます。

分岐先アドレスはレジスタ Rm の値で、このレジスタのビット [0] はゼロに設定されます。

復帰アドレスは R14 にセットされます。サブルーチンから復帰するには、BX R14 命令を使用するか、または R14 をスタックにストアし、ストアした値を PC に再ロードします。

構文

```
BLX{<cond>}  <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rm> ターゲット命令のアドレスを保持しているレジスタ。Rm のビット [0] が 0 の場合はターゲット ARM 命令が選択され、1 の場合はターゲット Thmub 命令が選択されます。<Rm> として R15 を指定した場合、動作は予測不能です。

アーキテクチャのバージョン

バージョン 5 以降に存在します。T バリエーション以外での動作の詳細については、P. A2-15 *「T ビットと J ビット」*を参照して下さい。

例外

なし

動作

```
if ConditionPassed(cond) then
    target = Rm
    LR = address of instruction after the BLX instruction
    CPSR T bit = target[0]
    PC = target AND 0xFFFFFFFE
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ARM と Thumb の状態遷移

ARM 状態ではワード境界でアラインされていないアドレスへの分岐は不可能なため、Rm[1:0] == 0b10 の場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.10 BX

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 0 1 0	SBO	SBO	SBO	0 0 0 1	Rm

BX（分岐と状態遷移）は指定のアドレスに分岐し、スイッチによるオプションとして Thumb 状態への切り替えを行います。

構文

BX{<cond>} <Rm>

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rm> 分岐先アドレスを含むレジスタを指定します。Rm のビット [0] が 0 の場合はターゲット ARM 命令が選択され、1 の場合はターゲット Thmub 命令が選択されます。

アーキテクチャのバージョン

バージョン 5 以降、およびバージョン 4 の T バリエーションに存在します。バージョン 5 の T バリエーション以外での動作の詳細については P. A2-15 *「T ビットと J ビット」*を参照して下さい。

例外

なし

動作

```
if ConditionPassed(cond) then
    CPSR T bit = Rm[0]
    PC = Rm AND 0xFFFFFFFE
```

注

ARM と Thumb の状態遷移

ARM 状態ではワード境界アラインしていないアドレスへの分岐は不可能なため、Rm[1:0] == 0b10 の場合、結果は予測不能です。

R15 の使用 <Rm> にレジスタ 15 を指定できますが、これは推奨されません。

BX R15 命令で、R15 は ARM コードとして通常に読み出されます。つまり、R15 は BX 命令自身のアドレスに 8 を加えたアドレスになります。結果として、ARM 状態で実行しながら、2 ワード先に分岐します。これは、オフセットフィールドが 0 の B 命令、ADD PC,PC,#0、MOV PC,PC 命令を実行した場合に得られるのとまったく同じ結果になります。新しいコードでは、より複雑な BX PC 命令よりも、上述の命令を使用して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.1 1 BXJ

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	1	0	0	1	0	SBO	SBO	SBO	0	0	1	0	Rm

BXJ（分岐および Jazelle® 状態への変更）は、Jazelle 状態が利用可能で許可されている場合、Jazelle 状態に移行します。それ以外の場合、BXJ は BX と同様の動作をします（P. A4-20「*BX*」参照）

構文

```
BXJ{<cond>}  <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rm> Jazelle 状態が利用できない場合に使用する分岐先アドレスの値を保持します。Rm のビット [0] が 0 の場合はターゲット ARM 命令が選択され、1 の場合はターゲット Thmub 命令が選択されます。

アーキテクチャのバージョン

バージョン 6 以降、およびバージョン 5 の J バリエーションに存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    if (JE bit of Main Configuration register) == 0 then
        T Flag = Rm[0]
        PC = Rm AND 0xFFFFFFFE
    else
        jpc = SUB-ARCHITECTURE DEFINED value
        invalidhandler = SUB-ARCHITECTURE DEFINED value
        if (Jazelle Extension accepts opcode at jpc) then
            if (CV bit of Jazelle OS Control register) == 0 then
                PC = invalidhandler
            else
                J Flag = 1
                Start opcode execution at jpc
        else
            if ((CV bit of Jazelle OS Control register) == 0) AND
                            (IMPLEMENTATION DEFINED CONDITION) then
                PC = invalidhandler
            else
                /* Subject to SUB-ARCHITECTURE DEFINED restrictions on Rm: */
                T Flag = Rm[0]
                PC = Rm AND 0xFFFFFFFE
```

用法

この命令は、以下の条件のいずれかが真の場合にのみ使用される必要があります。

- メインコンフィギュレーションレジスタの JE ビットが 0 である。
- 使用されている許可された Java 仮想マシンが、サブアーキテクチャ定義の使用中の Jazelle 拡張ハードウェアのすべての制約条件に適合している。

注

ARM と Thumb の状態遷移

IF（メインコンフィギュレーションレジスタの JE ビット）== 0

AND Rm[1:0] == 0b10 の場合、結果は予測不能です。これは、ARM 状態では、ワード境界にアラインしていないアドレスへの分岐は不可能なためです。

R15 の使用 <Rm> にレジスタ 15 が指定されている場合、結果は予測不能です。

Jazelle オペコードのアドレス

Jazelle オペコードのアドレスはサブアーキテクチャ定義の方法、一般には特別な汎用レジスタである *Jazelle プログラムカウンタ*（jpc）の内容で決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.12 CDP

31–28	27	26	25	24	23–20	19–16	15–12	11–8	7–5	4	3–0
cond	1	1	1	0	opcode_1	CRn	CRd	cp_num	opcode_2	0	CRm

CDP（コプロセッサデータ処理）は、番号が cp_num のコプロセッサに、ARM のレジスタとメモリから独立した演算を実行するように通知します。命令を実行できるコプロセサがない場合は、未定義命令例外が生成されます。

構文

```
CDP{<cond>}  <coproc>, <opcode_1>, <CRd>, <CRn>, <CRm>, <opcode_2>
CDP2         <coproc>, <opcode_1>, <CRd>, <CRn>, <CRm>, <opcode_2>
```

各項目の説明については以下を参照して下さい。

<cond>
この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

CDP2
命令の条件フィールドを 0b1111 にセットします。これによって、コプロセッサ設計者用のオペコード空間が追加されます。生成された命令は無条件でのみ実行できます。

<coproc>
コプロセッサの名前を指定すると､対応するコプロセッサ番号が命令の cp_num フィールドに配置されます。標準の汎用コプロセッサ名は p0、p1、...、p15 です。

<opcode_1>
コプロセッサ固有の方法で、実行するコプロセッサ命令を指定します。

<CRd>
この命令のデスティネーションとなるコプロセッサレジスタを指定します。

<CRn>
最初のオペランドを含むコプロセッサレジスタ。

<CRm>
2 番目のオペランドを含むコプロセッサレジスタ。

<opcode_2>
コプロセッサ固有の方法で、実行するコプロセッサ命令を指定します。

アーキテクチャのバージョン

CDP はバージョン 2 以降でサポートされています。

CDP2 はバージョン 5 以降でサポートされています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

未定義命令

動作

```
if ConditionPassed(cond) then
    Coprocessor[cp_num]-dependent operation
```

用法

CDP は、ARM のレジスタやメインメモリの値に演算を加えないコプロセッサの命令を起動するために使用します。たとえば、浮動小数点プロセッサの浮動小数点乗算命令です。

注

コプロセッサフィールド

命令のビット [31:24]、[11:8]、[4] のみがアーキテクチャで定義されています。他のフィールドは個々の ARM 開発システムに依存しています。

未実装のコプロセッサ命令

コプロセッサ 0 から 13 までのハードウェアコプロセッサのサポートはアーキテクチャのバージョンに関わらずオプションで、ARMv6 以前はコプロセッサ 14 と 15 もオプションです。実装では、コプロセッサ命令のサブセットの実装、またはコプロセッサ命令の未実装を選択できます。ただし、未実装のコプロセッサ命令は未定義命令例外の原因となります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.13 CLZ

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 1 1 0	SBO	Rd	SBO	0 0 0 1	Rm

CLZ（先行ゼロカウント）は値に含まれる最初のバイナリビット '1' に先行するバイナリビット '0' の数を戻します。

CLZ は条件コードフラグを更新しません。

構文

```
CLZ{<cond>}  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> 演算のデスティネーションレジスタ。<Rd> として R15 を指定した場合、結果は予測不能です。

<Rm> 演算のソースレジスタ。<Rm> として R15 を指定した場合、結果は予測不能です。

アーキテクチャのバージョン

バージョン 5 以降に存在します。

例外

なし

動作

```
if Rm == 0
    Rd = 32
else
    Rd = 31 - (bit position of most significant'1' in Rm)
```

用法

CLZ に続けて、結果として得られる Rd 分だけ Rm を左シフトすると、レジスタ Rm の値を正規化できます。この演算により Rm がシフトされ、最も上位の 1 がビット [31] になります。Rm が 0 で最上位の 1 が存在しない特別な場合は、以下のように、MOV ではなく MOVS を使用して Z フラグをセットします。

```
    CLZ   Rd, Rm
    MOVS  Rm, Rm, LSL Rd
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.14 CMN

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 0 1 1	1	Rn	SBZ	shifter_operand

CMN（2 の補数比較）は、最初の値を、2 番目の値の 2 の補数と比較します。最初の値はレジスタに含まれています。2 番目の値はイミディエート値またはレジスタの値で、比較を実行する前にシフト可能です。

CMN は、2 つの値の加算の結果に基づいて条件フラグを更新します。

構文

```
CMN{<cond>}  <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rn> 最初のオペランドを含むレジスタ。

<shifter_operand>

2 番目のオペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」*を参照して下さい。

I ビットが 0 で、かつ shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は CMN ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-35 *「乗算命令拡張空間」*を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    alu_out = Rn + shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = CarryFrom(Rn + shifter_operand)
    V Flag = OverflowFrom(Rn + shifter_operand)
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

CMN は、<shifter_operand> の値をレジスタ <Rn> の値に加算することにより比較を実行し、その結果に基づいて条件コードフラグを更新します。この演算は、1 番目のオペランドから 2 番目のオペランドの符号反転を減算して、演算結果に基づき条件コードフラグをセットするのとほとんど同じです。

相違点は、2 番目のオペランドが 0 または 0x80000000 のときに、生成される条件コードフラグの値が異なる可能性があることです。たとえば、次の命令では常に C フラグ = 1 になります。

```
CMP Rn, #0
```

そして、次の命令では常に C フラグ = 0 になります。

```
CMN Rn, #0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.15 CMP

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	0	I	1	0	1	0	1	Rn	SBZ	shifter_operand

CMP（比較）は、2 つの値を比較します。最初の値はレジスタに含まれています。2 番目の値はイミディエート値またはレジスタの値で、比較を実行する前にシフト可能です。

CMP は、2 番目の値を最初の値から減算した結果に基づいて条件フラグを更新します。

構文

```
CMP{<cond>}  <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rn> 最初のオペランドを含むレジスタ。

<shifter_operand>

2 番目のオペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」*を参照して下さい。

I ビットが 0 で、かつ shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は CMP ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-35 *「乗算命令拡張空間」*を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    alu_out = Rn - shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = NOT BorrowFrom(Rn - shifter_operand)
    V Flag = OverflowFrom(Rn - shifter_operand)
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.16 CPS

31–28	27–20	19–18	17	16	15–9	8	7	6	5	4–0
1 1 1 1	0 0 0 1 0 0 0 0	imod	mmod	0	SBZ	A	I	F	0	mode

CPS（プロセッサ状態変更）は、CPSR のモード A、I、F ビットを 1 つ、または複数変更します。その他の CPSR ビットは変更しません。

構文

```
CPS<effect> <iflags> {, #<mode>}

CPS     #<mode>
```

各項目の説明については以下を参照して下さい。

`<effect>` CPSR の割り込み無効ビット A、I、F にどのような効果をもたせるのか指定します。以下はもたせる効果の 1 つです。

`IE` 割り込み有効。imod == 0b10 でエンコードされます。指定されたビットが 0 にセットされます。

`ID` 割り込み無効。imod == 0b11 でエンコードされます。指定されたビットが 1 にセットされます。

`<effect>` が指定されている場合、影響を受けるビットは `<iflags>` で指定されます。これらのビットは命令の A、I、F ビットでエンコードされます。モードは、`<mode>` にモード値を指定することにより、任意に変更できます。

`<effect>` が指定されていない場合は以下のようになります。

- `<iflags>` は指定されず、A、I、F のマスクの設定は変更されません
- 命令の A、I、F ビットはゼロ
- imod = 0b00
- mmod = 0b1
- `<mode>` は新しいモード値を指定します。

`<iflags>` は以下に示す 1 つまたは複数のビットの並びで、どの割り込み無効フラグが影響を受けるのか指定します。

`a` 命令の A ビットをセットし、CPSR の A（不正確データアボート）ビットに、指定された効果を与えます。

`i` 命令の I ビットをセットし、CPSR の I（IRQ 割り込み）ビットに、指定された効果を与えます。

`f` 命令の F ビットをセットし、CPSR の F（FIQ 割り込み）ビットに、指定された効果を与えます。

`<mode>` 変更後のモードの番号を指定します。mode が指定されている場合、mmod == 1 となり、モード値は命令の mode フィールドでエンコードされます。mode が省略された場合は、mmode == 0 となり、命令の mode フィールドはゼロになります。詳細については、P. A2-14 *「モードビット」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

なし

動作

```
if InAPrivilegedMode() then
    if imod[1] == 1 then
        if A == 1 then CPSR[8] = imod[0]
        if I == 1 then CPSR[7] = imod[0]
        if F == 1 then CPSR[6] = imod[0]
    /* else no change to the mask */
    if mmod == 1 then
        CPSR[4:0] = mode
```

注

ユーザモード CPS はユーザモードでは効果がありません。

無意味なビットの組み合わせ

imod と mmod の以下の組み合わせには意味がありません。

- imod == 0b0、mmod == 0
- imod == 0b01、mmod == 0
- imod == 0b01、mmod == 1

アセンブラはこのような組み合わせを生成することはありません。このような組み合わせが実行時にどのような影響を与えるかは予測不能です。

状態 他のほとんどの ARM の命令と異なり、CPS は条件付きで実行できません。

予約モード モードを予約されている値に変更しようとすると、その結果は予測不能です。

例

```
    CPSIE   a,#31   ; enable imprecise data aborts, change to System mode
    CPSID   if      ; disable interrupts and fast interrupts
    CPS     #16     ; change to User mode
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.17 CPY

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
cond	0	0	0	1	1	0	1	0	SBZ	Rd	0	0	0	0	0	0	0	0	Rm

CPY（コピー）はレジスタ間で値をコピーします。CPY は MOV の同義語で、フラグの設定やシフトはありません。詳細については、P. A4-69 *「MOV」*を参照して下さい。

構文

```
CPY{<cond>}  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<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。

アーキテクチャのバージョン

バージョン 6 以上

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = Rm
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.18 EOR

31-28	27 26	25	24 23 22 21	20	19-16	15-12	11-0
cond	0 0	I	0 0 0 1	S	Rn	Rd	shifter_operand

EOR（排他的論理和）命令は、2 個の値の（排他的な）論理和をビットごとに実行します。最初の値はレジスタに含まれています。2 番目の値は、イミディエート値でもレジスタからの値でもどちらでもよく、排他的論理和演算を行う前にシフトすることも可能です。

EOR 命令では、結果に基づいて、必要であれば条件コードフラグを更新することもできます。

構文

```
EOR{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S 命令の S ビット（ビット [20]）を 1 にセットし、この命令で CPSR を更新するように指定します。S を省略すると S ビットは 0 にセットされ、CPSR は命令によって変更されません。S が指定された場合の CPSR の更新には、2 タイプあります。

- <Rd> が R15 でない場合、N フラグと Z フラグが演算の結果に従ってセットされ、シフタによって生成されたキャリーの出力ビットに C フラグがセットされます（P. A5-2 *「アドレシングモード 1 - データ処理オペランド」*参照）。V フラグと CPSR の残りの部分には影響しません。
- <Rd> が R15 の場合、カレントモードの SPSR は CPSR にコピーされます。ユーザモードまたはシステムモードには SPSR がないため、これらのモードで実行した場合、この形式の命令は予測不能な結果を招きます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<shifter_operand>

2 番目のオペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」*を参照して下さい。

I ビットが 0 で、かつ shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は EOR ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-32 *「命令セットの拡張」*を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = Rn EOR shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

用法

EOR 命令を使用すると、レジスタで選択したビットを反転できます。各ビットは、1 と EOR 演算をするとそのビットが反転され、0 と EOR 演算をするとそのビットは変更されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.19 LDC

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1 1 0	P	U	N	W	1	Rn	CRd	cp_num	8_bit_word_offset

LDC（コプロセッサロード）は、連続したメモリアドレスのシーケンスからメモリデータをコプロセッサにロードします。

命令を実行できるコプロセッサがない場合は、未定義命令例外が生成されます。

構文

```
LDC{<cond>}{L}  <coproc>, <CRd>, <addressing_mode>
LDC2{L}         <coproc>, <CRd>, <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

LDC2 命令の条件フィールドが 0b1111 にセットされます。これによって、コプロセッサ設計者用のオペコード空間が追加されます。生成された命令は無条件でのみ実行できます。

L 命令内の N ビット（ビット [22]）を 1 に設定し、ロングロード（単精度データ転送のかわりに倍精度データ転送を使用するなど）を指定します。L が省略された場合、N ビットは 0 になり、命令はショートロードを指定します。

<coproc> コプロセッサの名前を指定すると、対応するコプロセッサ番号が命令の cp_num フィールドに配置されます。標準の汎用コプロセッサ名は p0、p1、...、p15 です。

<CRd> コプロセッサのデスティネーションレジスタ。

<addressing_mode>

P. A5-49 *「アドレシングモード 5 - コプロセッサのロード / ストア」* を参照して下さい。これによって命令の P、U、Rn、W、8_bit_word_offset の各ビットが決定されます。

<addressing_mode>の全形式の構文にはベースレジスタ<Rn>が含まれます。命令によるベースレジスタ値の変更を指定する形式もあります（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

LDC2 はバージョン 5 以降でサポートされています。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

未定義命令、データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    load Memory[address,4] for Coprocessor[cp_num]
    while (NotFinished(Coprocessor[cp_num]))
        address = address + 4
        load Memory[address,4] for Coprocessor[cp_num]
    assert address == end_address
```

用法

LDC はメモリからコプロセッサのデータをロードする場合に有効です。

注

コプロセッサフィールド

命令ビット [31:23]、ビット [21:16]、ビット [11:0] のみが、ARM アーキテクチャで定義されています。他のフィールド（ビット [22] とビット [15:12]）は、個々の ARM 開発システムに依存しています。

インデクスなしアドレシングモード（P == 0、U == 1、W == 0）の場合、命令ビット [7:0] も ARM アーキテクチャでは定義されていません。このビットは他のコプロセッサオプションの指定に使用できます。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21 *「データアボートが発生した命令の影響」* を参照して下さい。

非ワード境界アライメントアドレス

CP15_reg1_Ubit の値が 0 の場合、コプロセッサロードレジスタ命令はアドレスの最下位の 2 ビットを無視します。システム制御コプロセッサが実装対象に含まれていて（第 B3 章 *「システム制御コプロセッサ」* 参照）、アライメントチェックが有効な場合、ビット [1:0] の値が 0b00 でないとアライメント例外が発生します。

CP15_reg1_Ubit の値が 1 の場合、すべての非ワード境界アライメントアクセスがアライメントフォルトを引き起こします。

未実装のコプロセッサ命令

ハードウェアコプロセッサのサポートは、アーキテクチャのバージョンに関わらずオプションです。実装では、コプロセッサ命令のサブセットの実装、またはコプロセッサ命令の未実装を選択できます。ただし、未実装のコプロセッサ命令は未定義命令例外の原因となります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.20 LDM (1)

31–28	27	26	25	24	23	22	21	20	19–16	15–0
cond	1	0	0	P	U	0	W	1	Rn	register_list

LDM (1)（複数ロード）は、汎用レジスタ内の空ではないサブセット（場合によってはすべて）にシーケンシャルなメモリ位置からロードします。この命令は、ブロックロード、スタック操作、プロシージャのイグジット（終了）シーケンスに有効です。

ロードされる汎用レジスタには PC を含めることができます。PC にロードされた場合、ロードされたワードはアドレスとして処理され、そのアドレスへの分岐が発生します。ARM のバージョン 5 以降では、BX (loaded_value) 命令が実行されたのと同様に、ロードされた値のビット [0] により、分岐後の実行が ARM 状態で実行されるのか Thumb 状態で実行されるのかが決まります（ARMv5 の T バリエーション以外での動作については、P. A2-15 *「T ビットと J ビット」*を参照して下さい）。アーキテクチャの以前のバージョンでは、命令 MOV PC,(loaded_value) が実行されたのと同様に、ロードされた値のビット [1:0] は無視され、ARM 状態での実行が継続されます。

構文

```
LDM{<cond>}<addressing_mode>  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<cond>
: この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<addressing_mode>
: P. A5-41 *「アドレシングモード 4 - 複数ロード / ストア」*を参照して下さい。これによって命令の P、U、W ビットが決定されます。

<Rn>
: <addressing_mode> によって使用されるベースレジスタを指定します。R15 をベースレジスタ <Rn> として使用すると、予測不能な結果が生じます。

!
: W ビットをセットし、命令は修飾済みの値をベースレジスタ Rn にライトバックします。詳細については P. A5-41 *「アドレシングモード 4 - 複数ロード / ストア」*を参照して下さい。! が省略された場合は、W ビットは 0 になり、命令はこの方法でベースレジスタを変更することはありません。ただし、ベースレジスタが <registers> に含まれる場合、値のロードにより変更されます。

<registers>
: カンマで区切られ、{ と } で囲まれたレジスタの一覧を指定します。レジスタのリストは LDM 命令によりロードされるレジスタの組を指定します。

  レジスタは最下位のメモリアドレス（start_address）から最も低い番号のレジスタに、そのあとで順に最上位のメモリアドレス（end_address）から最も高い番号のレジスタへの順序でロードされます。PC がレジルタリストで指定されている場合（オペコードのビット [15] がセットされている）、命令は PC にロードされたアドレス（データ）への分岐を起こします。

  R0 ～ R15 のそれぞれのレジスタが一覧にある場合は命令の register_list フィールドの対応するビット [i] が 1 になり、ない場合は 0 になります。ビット [15:0] がすべてゼロの場合、予測不能な結果を招きます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address

    for i = 0 to 14
        if register_list[i] == 1 then
            Ri = Memory[address,4]
            address = address + 4

    if register_list[15] == 1 then
        value = Memory[address,4]
        if (architecture version 5 or above) then
            pc = value AND 0xFFFFFFFE
            T Bit = value[0]
        else
            pc = value AND 0xFFFFFFFC
        address = address + 4

    if W ==1 then
        Rn = end_address    /* up/down and register-count dependent value*/
```

注

オペランドの制限

<registers> にベースレジスタ <Rn> が指定されていて、かつベースレジスタのライトバックが指定されている場合には、<Rn> の最終的な値は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21*「データアボートが発生した命令の影響」*を参照して下さい。

非ワード境界アライメントアドレス

CP15_reg1_Ubit の値が 0 の場合、複数ロード命令はアドレスの最下位 2 ビットを無視します。システム制御コプロセッサが実装に含まれ（第 B3 章*「システム制御コプロセッサ」*参照)、アライメントチェックが許可されている場合、ビット [1:0] の値が 0b00 でないアドレスではアライメント例外が発生します。

CP15_reg1_Ubit の値が 1 の場合、すべての非ワード境界アライメントアクセスがアライメントフォルトを引き起こします。

ARM 状態と Thumb 状態の遷移（ARM アーキテクチャバージョン 5 以降）

R15 にロードされたビット [1:0] の値が 0b10 の場合、ARM 状態では非ワード境界アライメントのアドレスへの分岐は不可能なため、予測不能な結果を招きます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

時間順序 この命令によって生成される、メモリの個別のワードに対するアクセスの時間順序は、一部の条件でのみ定義されます。詳細については、P. B2-13 *「メモリアクセスの制限」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.21 LDM (2)

31–28	27 26 25	24	23	22	21	20	19–16	15	14–0
cond	1 0 0	P	U	1	0	1	Rn	0	register_list

LDM (2) は、プロセッサが特権モードの状態のとき、ユーザモードレジスタをロードします。これはプロセスのスワップ時や、命令のエミュレートに有用です。LDM (2) は、ユーザモードの汎用レジスタ内の空ではないサブセットをシーケンシャルなメモリ位置からロードします。

構文

```
LDM{<cond>}<addressing_mode>  <Rn>, <registers_without_pc>^
```

各項目の説明については以下を参照して下さい。

<cond>
: この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<addressing_mode>
: P. A5-41 *「アドレシングモード 4 - 複数ロード / ストア」* を参照して下さい。これによって命令の P ビットと U ビットが決定されます。このアドレシングモードは W の値が 0 である場合に限り、この形の LDM 命令で使用できます。

<Rn>
: <addressing_mode> によって使用されるベースレジスタを指定します。R15 をベースレジスタ <Rn> として使用すると、予測不能な結果が生じます。

<registers_without_pc>
: カンマで区切られ、{ と } で囲まれたレジスタの一覧を指定します。このリストには PC を含めないで、LDM 命令でロードされるレジスタの組を指定する必要があります。

  レジスタは最下位のメモリアドレス（start_address）から最も低い番号のレジスタに、その後で順に最上位のメモリアドレス（end_address）から最も高い番号のレジスタへの順序でロードされます。

  R0 ～ R15 のそれぞれのレジスタが一覧にある場合は命令の register_list フィールドの対応するビット [i] が 1 になり、ない場合は 0 になります。ビット [15:0] がすべてゼロの場合、予測不能な結果を招きます。

^
: このことは、PC をロードしない LDM 命令では、ユーザモードレジスタがロードされることを示しています。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to 14
        if register_list[i] == 1
            Ri_usr = Memory[address,4]
            address = address + 4
    assert end_address == address - 4
```

注

ライトバック　ビット [21]（W ビット）をセットすると、予測不能な結果を招きます。

ユーザモードとシステムモード

この形の LDM は、ユーザモードおよびシステムモードにおいて予測不能な結果を招きます。

ベースレジスタモード　ベースレジスタは現在のプロセッサモードのレジスタから読み込まれ、ユーザモードレジスタからは読み込まれません。

データアボート　データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

非ワード境界アライメントアドレス

CP15_reg1_Ubit の値が 0 の場合、複数ロード命令はアドレスの最下位 2 ビットを無視します。システム制御コプロセッサが実装に含まれ（第 B3 章*「システム制御コプロセッサ」*参照）、アライメントチェックが許可されている場合、ビット [1:0] の値が 0b00 でないアドレスではアライメント例外が発生します。

CP15_reg1_Ubit の値が 1 の場合、すべての非ワード境界アライメントアクセスがアライメントフォルトを引き起こします。

時間順序　この命令によって生成される、メモリの個別のワードに対するアクセスの時間順序は、一部の条件でのみ定義されます。詳細については、P. B2-13 *「メモリアクセスの制限」*を参照して下さい。

バンクレジスタ　ARMv6 以前の ARM アーキテクチャでは、この形式の LDM の次にバンクレジスタをアクセスする命令を置くことはできません。この場合、間に NOP を置くことが簡単な解決策です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.22 LDM (3)

31–28	27	26	25	24	23	22	21	20	19–16	15	14–0
cond	1	0	0	P	U	1	W	1	Rn	1	register_list

LDM(3) は、汎用レジスタと PC のサブセットまたはすべてを、シーケンシャルなメモリ位置からロードします。さらに、現在のモードの SPSR は CPSR にコピーされます。これは例外から復帰するのに役に立ちます。

PC にロードされる値はアドレスとして扱われ、そのアドレスへの分岐が発生します。ARM のバージョン 5 以降およびバージョン 4 の T バリエーションでは、SPSR の T ビットから CPSR の T ビットにコピーされた値により、分岐後に ARM 状態で実行を続けるのか、Thumb 状態で実行を続けるのかが決定されます。ARMv5 の T バリエーション以外での動作については、P. A2-15 *「T ビットと J ビット」*を参照して下さい。以前のアーキテクチャバージョンでは、分岐後も ARM 状態（以前のアーキテクチャバージョンでは唯一可能な状態）で実行が継続されます。

構文

```
LDM{<cond>}<addressing_mode>  <Rn>{!}, <registers_and_pc>^
```

各項目の説明については以下を参照して下さい。

<cond>
: この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<addressing_mode>
: P. A5-41 *「アドレシングモード 4 - 複数ロード / ストア」*を参照して下さい。これによって命令の P、U、W ビットが決定されます。

<Rn>
: <addressing_mode> で使用するベースレジスタを指定します。R15 をベースレジスタ <Rn> として使用すると、予測不能な結果が生じます。

!
: W ビットをセットし、命令は修飾済みの値をベースレジスタ Rn に書き戻します。詳細については、P. A5-41 *「アドレシングモード 4 - 複数ロード / ストア」*を参照して下さい。! が省略された場合は、W ビットは 0 になり、命令はこの方法でベースレジスタを変更することはありません。ただし、ベースレジスタが <registers> に含まれる場合、値のロードにより変更されます。

<registers_and_pc>
: カンマで区切られ、{ と } で囲まれたレジスタの一覧を指定します。このリストには、PC が含まれ、LDM 命令でロードされるレジスタの組が指定されている必要があります。
: レジスタは最下位のメモリアドレス（start_address）から最も低い番号のレジスタに、その後で順に最上位のメモリアドレス（end_address）から最も高い番号のレジスタへの順序でロードされます。
: R0 ～ R15 のそれぞれのレジスタが一覧にある場合はそれぞれに対応する命令の register_list フィールドのビット [i] が 1 になり、ない場合は 0 になります。

^
: このことは、PC をロードする LDM 命令では、現行モードの SPSR は CPSR にコピーされることを示しています。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address

    for i = 0 to 14
        if register_list[i] == 1 then
            Ri = Memory[address,4]
            address = address + 4

    if CurrentModeHasSPSR() then
        CPSR = SPSR
    else UNPREDICTABLE

    value = Memory[address,4]
    PC = value
    address = address + 4

    if W ==1 then
        Rn = end_address    /*up/down and register_count dependent value */
```

注

ユーザモードとシステムモード

この命令はユーザモードまたはシステムモードでは予測不能な結果を招きます。

オペランドの制限

`<registers_and_pc>` にベースレジスタ <Rn> が指定されていて、かつベースレジスタのライトバックが指定されている場合には、<Rn> の最終的な値は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21*「データアボートが発生した命令の影響」*を参照して下さい。

非ワード境界アライメントアドレス

CP15_reg1_Ubit の値が 0 の場合、複数ロード命令はアドレスの最下位 2 ビットを無視します。システム制御コプロセッサが実装に含まれ（第 B3 章*「システム制御コプロセッサ」*参照）、アライメントチェックが許可されている場合、ビット [1:0] の値が 0b00 でないアドレスではアライメント例外が発生します。

CP15_reg1_Ubit の値が 1 の場合、すべての非ワード境界アライメントアクセスがアライメントフォルトを引き起こします。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ARM 状態と Thumb 状態の遷移（ARM アーキテクチャバージョン 4T、5 以降）

SPSR の T ビットが 0 で、かつ PC にロードされた値のビット [1] が 1 の場合、結果は予測不能です。これは、非ワード境界アライメントのアドレスにある ARM 命令への分岐は不可能なためです。通常の例外からの復帰では、これに対する特別な対策が必要でないことに注意して下さい。これは、例外エントリは常に SPSR の T ビットを 1 にセットするか、R14 の戻りリンク値のビット [1] を 0 にセットするためです。

時間順序 この命令により生成される、メモリの個々のワードに対するアクセスの時間順序は定義されていません。詳細については、P. B2-13 *「メモリアクセスの制限」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.23 LDR

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	1	I	P	U	0	W	1	Rn	Rd	addr_mode

LDR（レジスタロード）は、メモリアドレスからワードをロードします。

PC が <Rd> として指定された場合、この命令は PC がアドレスとして処理するデータワードをロードし、そのアドレスに分岐します。ARMv5T 以降では、BX (loaded_value) 命令が実行されたのと同様に、ロードされた値のビット [0] により、分岐後の実行が ARM 状態と Thumb 状態のどちらで実行されるかが決定されます。アーキテクチャの以前のバージョンでは、MOVPC,(loaded_value) 命令が実行されたのと同様に、ロードされた値のビット [1:0] は無視され、ARM 状態で実行が継続されます。

構文

```
LDR{<cond>}  <Rd>, <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> ロードされる値のデスティネーションレジスタ。

<addressing_mode>

P. A5-18 *「アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア」* を参照して下さい。これによって命令の I、P、U、W、Rn、addr_mode の各ビットが決定されます。

<addressing_mode>の全形式の構文にはベースレジスタ<Rn>が含まれます。命令によるベースレジスタ値の変更を指定する形式もあります（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0) then
        data = Memory[address,4] Rotate_Right (8 * address[1:0])
    else    /* CP15_reg_Ubit == 1 */
        data = Memory[address,4]
    if (Rd is R15) then
        if (ARMv5 or above) then
            PC = data AND 0xFFFFFFFE
            T Bit = data[0]
        else
            PC = data AND 0xFFFFFFFC
    else
        Rd = data
```

用法

PC をベースレジスタとして使用することで、PC 相対アドレシングが可能になり、ポジション・インディペンデントコードの使用が容易になります。適切なアドレシングモードと組み合わせることで、LDR により 32 ビットのメモリデータを汎用レジスタにロードでき、値の操作が可能になります。デスティネーションレジスタが PC の場合、この命令はメモリから 32 ビットのアドレスをロードし、そのアドレスに分岐します。

リンク付き分岐を合成するには、`LDR` 命令の前に `MOV LR, PC` を置きます。

アライメント

ARMv5 またはそれ以前

アドレスがワード境界アライメントでない場合、ロードされた値は、アドレスのビット [1:0] の値の 8 倍だけ右にローテートされます。リトルエンディアンメモリシステムの場合、このローテートによって、アドレス指定されたバイトがレジスタの最下位バイトになります。ビッグエンディアンメモリシステムの場合、このローテートの結果、アドレスのビット [0] が 0 か 1 かによって、アドレス指定されたバイトがレジスタのビット [31:24] かビット [15:8] になります。

システム制御コプロセッサが実装に含まれていて（第 B3 章*「システム制御コプロセッサ」*参照）、アライメントチェックが許可されている場合、ビット [1:0] の値が 0b00 でないアドレスではアライメント例外が発生します。

ARMv6 以降

ARMv6 から、バイト固定の混在エンディアン形式が、アライメントチェックオプションと共にサポートされています。ARMv6 の疑似コードでは、CPSR の E ビットにより定義されたエンディアン方式の転送で、アンアラインドな混在エンディアンのサポートが構成されていることを前提としています。

エンディアン方式およびアライメントの詳細については、P. A2-30*「エンディアンのサポート」*と P. A2-38*「アンアラインドアクセスのサポート」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21*「データアボートが発生した命令の影響」*を参照して下さい。

オペランドの制限

`<addressing_mode>` でベースレジスタライトバックを指定し、`<Rd>` と `<Rn>` に同じレジスタを指定した場合、結果は予測不能です。

R15 の使用 R15 が `<Rd>` に指定されている場合、ロードされる値のアドレスの値はワード境界にアラインしている必要があります。つまり、アドレス [1:0] は 0b00 の必要があります。また、Thumb のインタワーキングの理由により、R15[1:0] に値 0b10 をロードすることは許されていません。この制約が満たされない場合は、結果は予測不能です。

ARM 状態と Thumb 状態の遷移（ARM アーキテクチャバージョン 5 以降）

R15 にロードされたビット [1:0] の値が 0b10 の場合、ARM 状態では非ワード境界アライメントのアドレスへの分岐は不可能なため、予測不能な結果を招きます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.24 LDRB

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	1	I	P	U	1	W	1	Rn	Rd	addr_mode

LDRB（レジスタロードバイト）は、1 バイトをメモリからロードし、ゼロ拡張して 32 ビットワードにします。

構文

```
LDR{<cond>}B  <Rd>, <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> ロードされる値のデスティネーションレジスタ。<Rd> にレジスタ 15 が指定されている場合、結果は予測不能です。

<addressing_mode>

P. A5-18 *「アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア」* を参照して下さい。これによって命令の I、P、U、W、Rn、addr_mode の各ビットが決定されます。

<addressing_mode>の全形式の構文にはベース*レジスタ*<Rn>が含まれます。命令によるベースレジスタ値の変更を指定する形式もあります（*ベースレジスタライトバック*と呼ばれます)。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

動作

```
if ConditionPassed(cond) then
    Rd = Memory[address,1]
if writeback then
    Rn = address
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

適切なアドレシングモードと組み合わせることで、LDRB により 8 ビットのメモリデータを汎用レジスタにロードでき、その値の操作が可能になります。

PC をベースレジスタとして使用することで、PC 相対アドレシングが可能になり、ポジション・インディペンデントコードの使用が容易になります。

注

オペランドの制限

<addressing_mode> でベースレジスタライトバックを指定し、<Rd> と <Rn> に同じレジスタを指定した場合、結果は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21「*データアボートが発生した命令の影響*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.25 LDRBT

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	0	U	1	1	1	Rn	Rd	addr_mode

LDRBT（変換付きレジスタロードバイト）は、1 バイトをメモリからロードし、ゼロ拡張して 32 ビットワードにします。

プロセッサが特権モードの状態のとき LDRBT が実行されると、メモリシステムは、そのアクセスをプロセッサがユーザモードの場合と同様に処理するよう、シグナルを受けます。

構文

```
LDR{<cond>}BT  <Rd>, <post_indexed_addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> ロードされる値のデスティネーションレジスタ。<Rd> として R15 を指定した場合、結果は予測不能です。

<post_indexed_addressing_mode>

P. A5-18 *「アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア」*を参照して下さい。これによって命令の I、U、Rn、addr_mode の各ビットが決定されます。この命令では、アドレシングモード 2 のポストインデクス形式のみが使用できます。これらの形式では P == 0、W == 0 となり、P および W はそれぞれビット [24] とビット [21] です。この命令では代わりに P == 0、W == 1 を使用しますが、その他のすべての点についてはアドレシングモードは同一です。

<post_indexed_addressing_mode>の全形式の構文には、*ベースレジスタ*<Rn>が含まれます。すべての形式で、命令によるベースレジスタ値の変更が指定されます（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

動作

```
if ConditionPassed(cond) then
    Rd = Memory[address,1]
    Rn = address
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

`LDRBT` は、通常はユーザモードで実行されるメモリアクセス命令をエミュレートしている（特権）例外ハンドラで使用できます。ユーザモード権限の場合と同様に、アクセスは制限されます。

注

ユーザモード この命令をユーザモードで実行すると、通常のユーザモードアクセスが実行されます。

オペランドの制限

`<Rd>` と `<Rn>` に同じレジスタを指定した場合、結果は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.26 LDRD

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0	P	U	I	W	0	Rn	Rd	addr_mode	1 1 0 1	addr_mode

LDRD（レジスタロードダブルワード）は、ARM レジスタのペアに、メモリ内の連続した 2 ワードから値をロードします。このレジスタペアは、偶数番号のレジスタと、その次の奇数番号のレジスタに制限されます（R10 と R11 など）。

2 レジスタの LDM 命令と比較して、より多くのアドレシングモードを使用できます。

構文

```
LDR{<cond>}D  <Rd>, <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。If <cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> <addressing_mode> で指定されるメモリワードからデータをロードする、偶数番号のデスティネーションレジスタ。直後にある奇数番号のレジスタは、次のメモリワードのデスティネーションレジスタです。<Rd> が R14 の場合、2 番目のデスティネーションレジスタに R15 が指定されることになるため、命令の動作は予測不能です。<Rd> が奇数番号のレジスタを指定している場合、この命令は未定義となります。

<addressing_mode>

P. A5-33*「アドレシングモード 3 - その他のロードとストア」*を参照して下さい。これによって命令の P、U、I、W、Rn、addr_mode の各ビットが決定されます。

<addressing_mode>の全形式の構文にはベースレジスタ<Rn>が含まれます。命令によるベースレジスタ値の変更を指定する形式もあります（*ベースレジスタライトバック*と呼ばれます）。

<addressing_mode> で生成されるアドレスは、LDRD 命令でロードされる 2 ワードのうち下位ワードのアドレスです。上位ワードのアドレスは、このアドレス + 4 で生成されます。

アーキテクチャのバージョン

バージョン 5TE 以降に存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if (CP15_reg1_Ubit == 0)
    if ConditionPassed(cond) then
        if (Rd is even-numbered and not R14)
            if (address[2:0] == 0b000) then
                Rd = Memory[address,4]
                R(d+1) = memory[address+4,4]
            else
                UNPREDICTABLE
        else
            UNDEFINED
else    /* CP15_reg1_Ubit == 1 */
    if ConditionPassed(cond) then
        if (Rd is even-numbered and not R14)
            if (address[1:0] == 0b00) then
                Rd = Memory[address,4]
                R(d+1) = memory[address+4,4]
            else
                AlignmentFault()
        else
            UNDEFINED
```

注

オペランドの制限

<addressing_mode> でベースレジスタライトバックを実行し、ベースレジスタ <Rn> がその命令のデスティネーションレジスタの 1 つである場合、結果は予測不能です。

<addressing_mode> がインデックスレジスタ <Rm> を指定し、<Rm> が命令のデスティネーションレジスタの 1 つである場合、結果は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21「*データアボートが発生した命令の影響*」を参照して下さい。

アライメント　ARMv6 以前では、メモリアドレスが 64 ビット境界にアラインしていない場合、メモリから読み出されるデータは予測不能です。アライメントチェック（データアボートの発生）およびビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定の混在エンディアン形式が、モジュロ 4 およびモジュロ 8 のアライメントチェックオプションと共にサポートされています。ARMv6 の疑似コードでは、CPSR の E ビットにより定義されたエンディアン方式の転送で、アンアラインドな混在エンディアンのサポートが構成されていることを前提としています。

エンディアン形式およびアライメントの詳細については、P. A2-30「*エンディアンのサポート*」と P. A2-38「*アンアラインドアクセスのサポート*」を参照して下さい。

時間順序　2 つのメモリワードにアクセスする時間順序は、アーキテクチャで定義されていません。実装では 2 つの 32 ビットメモリにどの順序でもアクセスできます。また、2 つのワードを組み合わせて 1 つの 64 ビットメモリアクセスとすることもできます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.27 LDREX

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	1	1	0	0	1	Rn	Rd	SBO	1	0	0	1	SBO

LDREX（排他的レジスタロード）は、メモリからレジスタに値をロードし、同時に次の処理を実行します。

- アドレスが共有メモリ属性を持つ場合、共有のモニタ内に、実行中のプロセッサがその物理アドレスを排他的にアクセスしていることを示すマークを付けます。
- 実行中のプロセッサが、アクティブで非排他的アクセスをローカルモニタに表示します。

構文

```
LDREX{<cond>} <Rd>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> <Rd> で指定されるアドレスのメモリワードをロードするデスティネーションレジスタ。

<Rn> アドレスを含むレジスタ。

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    Rd = Memory[Rn,4]
    physical_address = TLB(Rn)
    if Shared(Rn) == 1 then
        MarkExclusiveGlobal(physical_address,processor_id,4)
    MarkExclusiveLocal(physical_address,processor_id,4)
    /* See P. A2-49「操作の概要」*/
    /* The notes take precedence over any implied atomicity or
       order of events indicated in the pseudo-code */
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

LDREX 命令と STREX 命令を併用すると、メモリを共有するマルチプロセッサシステムでプロセス間通信を実装できます。詳細については、P. A2-44「*同期化基本命令*」を参照して下さい。この機構は、間にコンテキストスイッチが介在することなく、アトミックのロードとストアが行われるのを保証するために、ローカルに使用することもできます。

注

R15 の使用 <Rd> または <Rn> としてレジスタ 15 を指定した場合、結果は予測不能です。

データアボート

LDREX 処理中にデータアボートが発生した場合、MarkExclusiveGlobal() および MarkExclusiveLocal()処理が実行されるかどうかは予測不能です。Rdは更新されません。

アライメント CP15 のレジスタ 1(A,U) != (0,0) かつ Rd[1:0] != 0b00 の場合、アライメント例外が発生します。

アンアラインドな排他ロードはサポートされません。Rd<1:0> != 0b00 で、(A,U) = (0,0) の場合、結果は予測不能です。

排他的操作のためのメモリサポート

排他的操作をサポートしていない（排他的モニタを実装していないなど）共有メモリ領域における LDREX の動作は、予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.28 LDRH

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	P	U	I	W	1	Rn	Rd	addr_mode	1	0	1	1	addr_mode

LDRH（レジスタロードハーフワード）は、ハーフワードをメモリからロードし、ゼロ拡張して 32 ビットワードにします。

構文

```
LDR{<cond>}H  <Rd>, <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond>　この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd>　ロードされる値のデスティネーションレジスタ。<Rd> として R15 を指定した場合、結果は予測不能です。

<addressing_mode>

P. A5-33*「アドレシングモード 3 - その他のロードとストア」*を参照して下さい。これによって命令の P、U、I、W、Rn、addr_mode の各ビットが決定されます。

<addressing_mode>の全形式の構文にはベースレジスタ<Rn>が含まれます。命令によるベースレジスタ値の変更を指定する形式もあります（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0) then
        if address[0] == 0 then
            data = Memory[address,2]
        else
            data = UNPREDICTABLE
    else    /* CP15_reg1_Ubit == 1 */
        data = Memory[address,2]
    Rd = ZeroExtend(data[15:0])
```

用法

適切なアドレシングモードと組み合わせることで、LDRH により 16 ビットのメモリデータを汎用レジスタにロードでき、値の操作が可能になります。

PC をベースレジスタとして使用することで、PC 相対アドレシングが可能になり、ポジション・インディペンデントコードの使用が容易になります。

注

オペランドの制限

<addressing_mode> でベースレジスタライトバックを指定し、<Rd> と <Rn> に同じレジスタを指定した場合、結果は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント ARMv6 以前は、メモリアドレスがハーフワード境界にアラインしていない場合、メモリから読み出されるデータは予測不能です。アライメントチェック（アドレス [0] != 0 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定の混在エンディアン形式が、アライメントチェックオプションと共にサポートされています。ARMv6 の疑似コードでは、CPSR の E ビットにより定義されたエンディアン形式の転送で、混在エンディアンのサポートが構成されていることを前提としています。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.29 LDRSB

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	P	U	I	W	1	Rn	Rd	addr_mode	1	1	0	1	addr_mode

LDRSB（レジスタロード符号付きバイト）は、1 バイトをメモリからロードし、符号拡張して 32 ビットワードにします。

構文

```
LDR{<cond>}SB  <Rd>, <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> ロードされる値のデスティネーションレジスタ。<Rd> として R15 を指定した場合、結果は予測不能です。

<addressing_mode>

P. A5-33 *「アドレシングモード 3 - その他のロードとストア」* を参照して下さい。これによって命令の P、U、I、W、Rn、addr_mode の各ビットが決定されます。

<addressing_mode>の全形式の構文にはベー*スレジスタ*<Rn>が含まれます。命令によるベースレジスタ値の変更を指定する形式もあります（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

バージョン 4 以降に存在します。

例外

データアボート

動作

```
if ConditionPassed(cond) then
    data = Memory[address,1]
    Rd = SignExtend(data)
    if writeback then
        Rn = address
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

LDRSBを適切なアドレシングモードで使用すると、8ビットの符号付きメモリデータを汎用レジスタにロードでき、その値の操作が可能になります。

PC をベースレジスタとして使用すると、PC 相対アドレシングを実行できます。これにより、位置独立コード（PIC）の使用が容易になります。

注

オペランドの制限

<addressing_mode> でベースレジスタライトバックを指定し、<Rd> と <Rn> に同じレジスタを指定した場合、結果は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21「*データアボートが発生した命令の影響*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.30 LDRSH

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	P	U	I	W	1	Rn	Rd	addr_mode	1	1	1	1	addr_mode

LDRSH（レジスタロード符号付きハーフワード）は、ハーフワードをメモリからロードし、符号拡張して 32 ビットワードにします。

アドレスがハーフワード境界にアラインしていない場合、結果は予測不能です。

構文

```
LDR{<cond>}SH  <Rd>, <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond>　この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd>　ロードされる値のデスティネーションレジスタ。<Rd> として R15 を指定した場合、結果は予測不能です。

<addressing_mode>

P. A5-33 *「アドレシングモード 3 - その他のロードとストア」*を参照して下さい。これによって命令の P、U、I、W、Rn、addr_mode の各ビットが決定されます。

<addressing_mode>の全形式の構文にはベースレジスタ<Rn>が含まれます。命令によるベースレジスタ値の変更を指定する形式もあります（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

バージョン 4 以降に存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0) then
        if address[0] == 0 then
            data = Memory[address,2]
        else
            data = UNPREDICTABLE
    else    /* CP15_reg1_Ubit == 1 */
        data = Memory[address,2]
    Rd = ZeroExtend(data[15:0])
```

用法

適切なアドレシングモードで使用すると、LDRSH により 16 ビットの符号付きメモリデータを汎用レジスタにロードでき、値の操作が可能になります。

PC をベースレジスタとして使用することで、PC 相対アドレシングが可能になり、ポジション・インディペンデントコードの使用が容易になります。

注

オペランドの制限

<addressing_mode> でベースレジスタライトバックを指定し、<Rd> と <Rn> に同じレジスタを指定した場合、結果は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21「*データアボートが発生した命令の影響*」を参照して下さい。

アライメント ARMv6 以前は、メモリアドレスがハーフワード境界にアラインしていない場合、メモリから読み出されるデータは予測不能です。アライメントチェック（アドレス [0] != 0 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定の混在エンディアン形式が、アライメントチェックオプションと共にサポートされています。ARMv6 の疑似コードでは、CPSR の E ビットにより定義されたエンディアン形式の転送で、混在エンディアンのサポートが構成されていることを前提としています。

エンディアン形式とアライメントの詳細については、P. A2-30「*エンディアンのサポート*」と P. A2-38「*アンアラインドアクセスのサポート*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.31 LDRT

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	1	I	0	U	0	1	1	Rn	Rd	addr_mode

LDRT（変換付きレジスタロード）命令は、メモリからワードをロードします。

プロセッサが特権モードのときに LDRT を実行すると、プロセッサがユーザモードの場合と同様にアクセスを処理することを要求するシグナルがメモリシステムに送られます。

構文

```
LDR{<cond>}T  <Rd>, <post_indexed_addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> ロードされる値のデスティネーションレジスタ。<Rd> として R15 を指定した場合、結果は予測不能です。

<post_indexed_addressing_mode>

P. A5-18*「アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア」*を参照して下さい。これによって命令の I、U、Rn、addr_mode の各ビットが決定されます。この命令では、アドレシングモード 2 のポストインデクス形式のみが使用できます。これらの形式では P == 0、W == 0 となり、P および W はそれぞれビット [24] とビット [21] です。この命令では代わりに P == 0、W == 1 を使用しますが、その他のすべての点についてはアドレシングモードは同一です。

<post_indexed_addressing_mode> の全形式の構文には、*ベースレジスタ* <Rn> が含まれます。すべての形式で、命令によるベースレジスタ値の変更が指定されます（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0) then
        data = Memory[address,4] Rotate_Right (8 * address[1:0])
    else    /* CP15_reg1_Ubit == 1 */
        data = Memory[address,4]
    if (Rd is R15) then
        if (ARMv5 or above) then
            PC = data AND 0xFFFFFFFE
            T bit = data[0]
        else
            PC = data AND 0xFFFFFFFC
    else
        Rd = data
```

用法

LDRT は、通常はユーザモードで実行されるメモリアクセス命令をエミュレートしている（特権）例外ハンドラで使用できます。ユーザモード権限の場合と同様に、アクセスは制限されます。

注

ユーザモード　この命令をユーザモードで実行すると、通常のユーザモードアクセスが実行されます。

オペランドの制限

<Rd> と <Rn> に同じレジスタを指定した場合、結果は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21「*データアボートが発生した命令の影響*」を参照して下さい。

アライメント　LDR の場合と同様に、P. A4-44「*LDR*」を参照して下さい。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.32 MCR

31 28	27	26	25	24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
cond	1	1	1	0	opcode_1	0	CRn	Rd	cp_num	opcode_2	1	CRm

MCR（ARM レジスタからコプロセッサへの移動）命令は、レジスタ <Rd> の値を、番号 cp_num のコプロセッサに転送します。

命令を実行できるコプロセッサがない場合は、未定義命令例外が生成されます。

構文

```
MCR{<cond>}  <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
MCR2          <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

MCR2 命令の条件フィールドを 0b1111 にセットします。これによって、コプロセッサ設計者用のオペコード空間が追加されます。生成された命令は無条件でのみ実行できます。

<coproc> コプロセッサの名前を指定すると、対応するコプロセッサ番号が命令の cp_num フィールドに配置されます。標準の汎用コプロセッサ名は p0、p1、...、p15 です。

<opcode_1> コプロセッサ固有のオペコード。

<Rd> コプロセッサに転送される値を含む ARM レジスタ。<Rd> として R15 を指定した場合、結果は予測不能です。

<CRn> デスティネーションコプロセッサレジスタ。

<CRm> 追加のコプロセッサレジスタ（デスティネーションまたはソース）。

<opcode_2> コプロセッサ固有のオペコード。値が省略されている場合、<opcode_2> は 0 と見なされます。

アーキテクチャのバージョン

MCR はすべてのバージョンに存在します。

MCR2 はバージョン 5 以降に存在します。

例外

未定義命令

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    send Rd value to Coprocessor[cp_num]
```

用法

MCR を使用すると、ARM レジスタの値に対するコプロセッサ命令を開始できます。たとえば、浮動小数点コプロセッサを使用する場合に、固定小数点を浮動小数点に変換させる命令などの開始が可能です。

注

コプロセッサフィールド

ARM アーキテクチャで定義されるのは、命令のビット [31:24]、ビット [20]、ビット [15:8]、ビット [4] のみです。他のフィールドは個々の ARM 開発システムに依存しています。

未実装のコプロセッサ命令

コプロセッサ 0 から 13 までのハードウェアコプロセッサのサポートはアーキテクチャのバージョンに関わらずオプションで、ARMv6 以前はコプロセッサ 14 と 15 もオプションです。実装では、コプロセッサ命令のサブセットの実装、またはコプロセッサ命令の未実装を選択できます。ただし、未実装のコプロセッサ命令は未定義命令例外の原因となります。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.33 MCRR

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7–4	3–0
cond	1	1	0	0	0	1	0	0	Rn	Rd	cp_num	opcode	CRm

MCRR（2 つの ARM レジスタからコプロセッサへの移動）命令は、2 つの ARM レジスタの値をコプロセッサに転送します。

命令を実行できるコプロセッサがない場合は、未定義命令例外が生成されます。

構文

```
MCRR{<cond>}  <coproc>, <opcode>, <Rd>, <Rn>, <CRm>
MCRR2  <coproc>, <opcode>, <Rd>, <Rn>, <CRm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

MCRR2 命令の条件フィールドを 0b1111 にセットします。これによって、コプロセッサ設計者用のオペコード空間が追加されます。生成された命令は無条件でのみ実行できます。

<coproc> コプロセッサの名前を指定すると、対応するコプロセッサ番号が命令の cp_num フィールドに配置されます。標準の汎用コプロセッサ名は p0、p1、...、p15 です。

<opcode> コプロセッサ固有のオペコード。

<Rd> コプロセッサに転送される値を含む最初の ARM レジスタ。<Rd> として R15 を指定した場合、結果は予測不能です。

<Rn> コプロセッサに転送される値を含む 2 番目の ARM レジスタ。<Rn> として R15 を指定した場合、または Rn = Rd の場合、結果は予測不能です。

<CRm> デスティネーションコプロセッサレジスタ。

アーキテクチャのバージョン

MCRR はバージョン 5TE 以降に存在します。

MCRR2 はバージョン 6 以降に存在します。

例外

未定義命令

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    send Rd value to Coprocessor[cp_num]
    send Rn value to Coprocessor[cp_num]
```

用法

MCRR を使用すると、2 つの ARM レジスタの値に対するコプロセッサ命令を開始できます。たとえば浮動小数点コプロセッサの場合、2 つの ARM レジスタに保持されている倍精度浮動小数点数を、浮動小数点レジスタに転送する命令として使用できます。

注

コプロセッサフィールド

命令ビット [31:8] のみが ARM アーキテクチャで定義されています。他のフィールドは個々の ARM 開発システムに依存しています。

未実装のコプロセッサ命令

コプロセッサ 0 から 13 までのハードウェアコプロセッサのサポートはアーキテクチャのバージョンに関わらずオプションで、ARMv6 以前はコプロセッサ 14 と 15 もオプションです。実装では、コプロセッサ命令のサブセットの実装、またはコプロセッサ命令の未実装を選択できます。ただし、未実装のコプロセッサ命令は未定義命令例外の原因となります。

転送の順序　コプロセッサは、これらの命令を使用する場合、<Rd> と <Rn> の各値の使用方法を定義します。アーキテクチャでは、2 つのレジスタ転送が特定の順序で発生するという制限はありません。Rd を Rn の前、後、同時のいずれのタイミングで転送するかは実装定義です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.34 MLA

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	0	0	0	1	S	Rd	Rn	Rs	1	0	0	1	Rm

MLA（積和演算）命令は、符号付きまたは符号なしの 2 つの 32 ビット値を乗算し、3 番目の 32 ビット値を加算します。結果の最下位 32 ビットが、デスティネーションレジスタに書き込まれます。

MLA 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
MLA{<cond>}{S}  <Rd>, <Rm>, <Rs>, <Rn>
```

各項目の説明については以下を参照して下さい。

<cond>
: この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S
: 命令の S ビット（ビット [20]）に 1 をセットし、積和の結果に従って CPSR の N フラグと Z フラグをセットすることを指定します。S を省略すると、命令の S ビットは 0 にセットされ、CPSR 全体が命令の影響を受けません。

<Rd>
: デスティネーションレジスタ。

<Rm>
: <Rs> の値に乗算される値を保持します。

<Rs>
: <Rm> の値に乗算される値を保持します。

<Rn>
: <Rs> と <Rm> の積に加算される値を保持します。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    Rd = (Rm * Rs + Rn)[31:0]
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = unaffected in v5 and above, UNPREDICTABLE in v4 and earlier
        V Flag = unaffected
```

注

R15 の使用 <Rd>、<Rm>、<Rs>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

符号付きと符号なし MLA 命令は、64 ビットの積のうち、下位 32 ビットのみを結果として生成します。このため、MLA は、乗算する数が符号付きでも符号なしでも同じ結果を返します。

C フラグ ARMv5 以降では、MLAS 命令は C フラグが変更されないように定義されています。以前のバージョンのアーキテクチャでは、MLAS 命令の実行後の C フラグの値は予測不能です。

オペランドの制限 以前の説明には、<Rd> と <Rm> に同じレジスタを指定した場合は、予測不能な結果を招くことが記載されていました。高性能の乗算器では結果をライトバックする前にすべてのオペランドを読み込むため、ARMv6 では制限は全くなく、該当する ARM の v4 および v5 のどの実装でも、この制限は必要ありません。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.35 MOV

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 1 0 1	S	SBZ	Rd	shifter_operand

MOV（移動）命令は、ディスティネーションレジスタへの値の書き込みを行います。この値はイミディエート値またはレジスタの値で、書き込みを行う前にシフトさせることができます。

MOV 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
MOV{<cond>}{S}  <Rd>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

<cond>
この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S
命令の S ビット（ビット [20]）に 1 をセットし、この命令で CPSR を更新することを指定します。S を省略すると S ビットは 0 にセットされ、CPSR は命令によって変更されません。S が指定された場合の CPSR の更新には、2 タイプあります。

- <Rd> が R15 でない場合、N フラグと Z フラグは移動した値に基づいて（シフトが指定されている場合はシフトの後に）セットされ、C フラグはシフタで生成されたキャリー出力ビットにセットされます（P. A5-2 *「アドレシングモード 1 - データ処理オペランド」* 参照）。V フラグと CPSR の他のビットは変更されません。
- <Rd> が R15 の場合、現在のモードの SPSR が CPSR にコピーされます。ユーザモードまたはシステムモードには SPSR がないため、これらのモードで実行した場合、この形式の命令は予測不能な結果を招きます。

<Rd>
デスティネーションレジスタ。

<shifter_operand>

オペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」* を参照して下さい。

I ビットが 0 で、かつ shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は MOV ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-32 *「命令セットの拡張」* を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    Rd = shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

用法

MOV を使用すると、次のような処理を実行できます。

- レジスタ間で値を移動する。
- レジスタに定数値を書き込む。
- 他の算術演算や論理演算を一切行わずにシフトを実行する。*n* ビットの左シフトによって、2^n による乗算を行う。
- 命令のデスティネーションが PC の場合、分岐が発生します。例として、次の命令は

  `MOV PC, LR`

  サブルーチンからの復帰に使用できます（命令 P. A4-10「*B, BL*」を参照）。アーキテクチャ v4 の T バリエーションおよびアーキテクチャ v5 以降では、`MOV PC, LR` の代わりに、`BX LR` 命令を使用する必要があります。これは、`BX` 命令は必要に応じて自動的に Thumb 状態への切り替えを行うためです（ARM アーキテクチャ v5 の T バリエーション以外で行われる動作について、P. A2-15「*T ビットと J ビット*」も参照して下さい）。
- 命令のデスティネーションが PC で、S ビットがセットされている場合、分岐が発生して、現在のモードの SPSR が CPSR にコピーされます。このことは、`MOVS PC, LR` 命令を使用すると、一部の例外からの復帰が可能であることを意味します（P. A2-16「*例外*」参照）。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.36 MRC

31–28	27	26	25	24	23–21	20	19–16	15–12	11–8	7–5	4	3–0
cond	1	1	1	0	opcode_1	1	CRn	Rd	cp_num	opcode_2	1	CRm

MRC（コプロセッサから ARM レジスタへの移動）命令は、コプロセッサから ARM レジスタまたは条件フラグへの値の転送を行います。

命令を実行できるコプロセッサがない場合は、未定義命令例外が生成されます。

構文

```
MRC{<cond>}  <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
MRC2         <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
```

各項目の説明については以下を参照して下さい。

<cond>
: この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

MRC2
: 命令の条件フィールドを 0b1111 にセットします。これによって、コプロセッサ設計者用のオペコード空間が追加されます。生成された命令は無条件でのみ実行できます。

<coproc>
: コプロセッサの名前を指定すると、対応するコプロセッサ番号が命令の cp_num フィールドに配置されます。標準の汎用コプロセッサ名は p0、p1、...、p15 です。

<opcode_1>
: コプロセッサ固有のオペコード。

<Rd>
: この命令のデスティネーションとなる ARM レジスタを指定します。<Rd> に R15 を指定すると、汎用レジスタではなく条件コードフラグが更新されます。

<CRn>
: 最初のオペランドを含むコプロセッサレジスタ。

<CRm>
: 追加のコプロセッサレジスタ（ソースまたはデスティネーション）。

<opcode_2>
: コプロセッサ固有のオペコード。値が省略されている場合、<opcode_2> は 0 と見なされます。

アーキテクチャのバージョン

MRC はすべての有効なバージョンに存在します。

MRC2 はバージョン 5 以降に存在します。

例外

未定義命令

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    data = value from Coprocessor[cp_num]
    if Rd is R15 then
        N flag = data[31]
        Z flag = data[30]
        C flag = data[29]
        V flag = data[28]
    else /* Rd is not R15 */
        Rd = data
```

用法

MRC 命令の使用目的は 2 つあります。

1. <Rd> に R15 を指定すると、コプロセッサの値の上位 4 ビット（<coproc> で指定）を元に条件コードフラグのビットが更新され、コプロセッサのステータスに基づいた条件付き分岐が可能になります。このとき、他の 28 ビットは無視されます。

   このような使用の例として、浮動小数点コプロセッサによって実行した比較の結果を ARM の条件フラグに転送する場合が挙げられます。

2. それ以外は、レジスタ <Rd> に、<coproc> で指定されたコプロセッサからの値を書き込むために使用されます。

   この場合の例としては、浮動小数点コプロセッサで浮動小数点を整数に変換する命令が挙げられます。

注

コプロセッサフィールド

ARM アーキテクチャで定義されるのは、命令のビット [31:24]、ビット [20]、ビット [15:8]、ビット [4] のみです。他のフィールドは個々の ARM 開発システムに依存しています。

未実装のコプロセッサ命令

コプロセッサ 0 から 13 までのハードウェアコプロセッサのサポートはアーキテクチャのバージョンに関わらずオプションで、ARMv6 以前はコプロセッサ 14 と 15 もオプションです。実装では、コプロセッサ命令のサブセットの実装、またはコプロセッサ命令の未実装を選択できます。ただし、未実装のコプロセッサ命令は未定義命令例外の原因となります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.37 MRRC

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 4	3 0
cond	1 1 0 0 0 1 0 1	Rn	Rd	cp_num	opcode	CRm

MRRC（コプロセッサから 2 つの ARM レジスタへの移動）命令は、コプロセッサから 2 つの ARM レジスタに値を転送します。

命令を実行できるコプロセッサがない場合は、未定義命令例外が生成されます。

構文

```
MRRC{<cond>}  <coproc>, <opcode>, <Rd>, <Rn>, <CRm>
MRRC2  <coproc>, <opcode>, <Rd>, <Rn>, <CRm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

MRRC2 命令の条件フィールドを 0b1111 にセットします。これによって、コプロセッサ設計者用のオペコード空間が追加されます。生成された命令は無条件でのみ実行できます。

<coproc> コプロセッサの名前を指定すると､対応するコプロセッサ番号が命令の cp_num フィールドに配置されます。標準の汎用コプロセッサ名は p0、p1、...、p15 です。

<opcode> コプロセッサ固有のオペコード。

<Rd> 最初のデスティネーションとなる ARM レジスタを指定します。<Rd> として R15 を指定した場合、結果は予測不能です。

<Rn> 2 番目のデスティネーションとなる ARM レジスタを指定します。<Rn> として R15 を指定した場合、結果は予測不能です。

<CRm> 転送されるデータの提供元のコプロセッサレジスタ。

アーキテクチャのバージョン

MRRC はバージョン 5TE 以降に存在します。

MRRC2 はバージョン 6 以降に存在します。

例外

未定義命令

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    Rd = first value from Coprocessor[cp_num]
    Rn = second value from Coprocessor[cp_num]
```

用法

MRRC を使用すると、2 つの ARM レジスタへの値の書き込み動作が実行できます。たとえば浮動小数点コプロセッサの場合、浮動小数点レジスタに保持されている倍精度浮動小数点数を、2 つの ARM レジスタに転送する命令として使用できます。

注

オペランドの制限

<Rd> と <Rn> に同じレジスタを指定した場合、結果は予測不能です。

コプロセッサフィールド

命令ビット [31:8] のみが ARM アーキテクチャで定義されています。他のフィールドは個々の ARM 開発システムに依存しています。

未実装のコプロセッサ命令

コプロセッサ 0 から 13 までのハードウェアコプロセッサのサポートはアーキテクチャのバージョンに関わらずオプションで、ARMv6 以前はコプロセッサ 14 と 15 もオプションです。実装では、コプロセッサ命令のサブセットの実装、またはコプロセッサ命令の未実装を選択できます。ただし、未実装のコプロセッサ命令は未定義命令例外の原因となります。

転送の順序

コプロセッサは、これらの命令を使用する場合、<Rd> と <Rn> にどの値を書き込むかを定義します。アーキテクチャでは、2 つのレジスタ転送が特定の順序で発生するという制限はありません。Rd を Rn の前、後、同時のいずれのタイミングで転送するかは実装定義です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.38 MRS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	0	0	1	0	R	0	0	SBO	Rd	SBZ

MRS（PSR から汎用レジスタへの移動）命令は、現在のモードの CPSR または SPSR の値を、汎用レジスタに移動します。値を移動した汎用レジスタでは、通常のデータ処理命令でその値の評価および操作を行うことができます。

構文

```
MRS{<cond>}  <Rd>, CPSR
MRS{<cond>}  <Rd>, SPSR
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。<Rd> として R15 を指定した場合、結果は予測不能です。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    if R == 1 then
        Rd = SPSR
    else
        Rd = CPSR
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

MRS 命令は一般に、以下の 3 つの目的で使用します。

- PSR を更新する場合に、リード・モディファイ・ライトシーケンスの一部として使用します。詳細については、P. A4-77「*MSR*」を参照して下さい。
- 例外が発生し、さらに同じタイプのネストした例外が発生する可能性がある場合、例外モードの SPSR は破壊される恐れがあります。この問題を解決するには、ネストした例外が発生する前に、例外からの復帰に備えて SPSR 値を保存しておき、後で復元します。SPSR 値の保存には通常、MRS 命令を使用してからストア命令を使用します。SPSR を復元する場合は、保存の時とは逆に、ロード命令の次に MSR 命令の順で使用します。
- プロセススワップコードでは、スワップアウトするプロセスのモデル状態を、関連する PSR の内容を含めて保存し、スワップインするときにはスワップアウトしたのと同じプロセス状態を復元します。この場合も、MRS 命令 - ストア命令と、ロード命令 -MSR 命令のシーケンスを使用します。

注

ユーザモード SPSR ユーザモードまたはシステムモードで SPSR にアクセスすると、予測不能な結果を招きます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.39 MSR

イミディエートオペランド

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	0	0	1	1	0	R	1	0	field_mask	SBO	rotate_imm	8_bit_immediate

レジスタオペランド

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	R	1	0	field_mask	SBO	SBZ	0	0	0	0	Rm

`MSR`（ARM レジスタからステータスレジスタへの移動）命令は、汎用レジスタまたはイミディエート定数の値を現在のモードの CPSR または SPSR に転送します。

構文

```
MSR{<cond>}  CPSR_<fields>, #<immediate>
MSR{<cond>}  CPSR_<fields>, <Rm>
MSR{<cond>}  SPSR_<fields>, #<immediate>
MSR{<cond>}  SPSR_<fields>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<cond>` この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。`<cond>` が省略されている場合、`AL`（常時）条件が使用されます。

`<fields>` 以下の 1 つ以上を順に指定します。

- `c` 制御フィールドマスクビット（ビット 16）をセットします。
- `x` 拡張フィールドマスクビット（ビット 17）をセットします。
- `s` ステータスフィールドマスクビット（ビット 18）をセットします。
- `f` フラグフィールドマスクビット（ビット 19）をセットします。

`<immediate>` CPSR または SPSR に転送されるイミディエート値。イミディエート値には、8 ビットイミディエート（`0x00` ～ `0xFF`）と、その値を 0 ～ 30 の範囲の偶数ビットだけ右ローテートした結果得られる値を使用できます。これらの値は、P. A5-6 *「データ処理オペランド - イミディエート」*で示されているイミディエート形式で許可される値と同じです。

`<Rm>` CPSR または SPSR に転送される汎用レジスタ。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

PSR ビットは、更新のルールに従って 4 つのカテゴリに分けられています。詳細については、P. A2-11 「*PSR ビットのタイプ*」を参照して下さい。

PSR ビットのカテゴリを示すため、擬似コードでは 4 つのビットマスク定数を使用します。これらのマスク値はアーキテクチャのバージョンによって異なります。詳細については表 A4-1 を参照して下さい。

表 A4-1 ビットマスク定数

アーキテクチャのバージョン[a]	UnallocMask	UserMask	PrivMask	StateMask
4	0x0FFFFF20	0xF0000000	0x0000000F	0x00000000
4T、5T	0x0FFFFF00	0xF0000000	0x0000000F	0x00000020
5TE、5TExP	0x07FFFF00	0xF8000000	0x0000000F	0x00000020
5TEJ	0x06FFFF00	0xF8000000	0x0000000F	0x01000020
6	0x06F0FC00	0xF80F0200	0x000001DF	0x01000020

a. アーキテクチャのバリエーション 4、4T、5T、5TExP は、過去のバージョンとしてのみここに記載されています。

```
if ConditionPassed(cond) then
    if opcode[25] == 1 then
        operand = 8_bit_immediate Rotate_Right (rotate_imm * 2)
    else
        operand = Rm
    if (operand AND UnallocMask) !=0 then
        UNPREDICTABLE              /* Attempt to set reserved bits */
    byte_mask = (if field_mask[0] == 1 then 0x000000FF else 0x00000000) OR
                (if field_mask[1] == 1 then 0x0000FF00 else 0x00000000) OR
                (if field_mask[2] == 1 then 0x00FF0000 else 0x00000000) OR
                (if field_mask[3] == 1 then 0xFF000000 else 0x00000000)
    if R == 0 then
        if InAPrivilegedMode() then
            if (operand AND StateMask) != 0 then
                UNPREDICTABLE      /* Attempt to set non-ARM execution state */
            else
                mask = byte_mask AND (UserMask OR PrivMask)
        else
            mask = byte_mask AND UserMask
        CPSR = (CPSR AND NOT mask) OR (operand AND mask)
    else  /* R == 1 */
        if CurrentModeHasSPSR() then
            mask = byte_mask AND (UserMask OR PrivMask OR StateMask)
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

```
        SPSR = (SPSR AND NOT mask) OR (operand AND mask)
    else
        UNPREDICTABLE
```

用法

MSR を使用すると、条件コードフラグや、割り込みイネーブル、プロセッサモードなどの値を更新できます。

通常、PSR の値を更新するには、PSR から汎用レジスタへの移動（MRS 命令を使用）、汎用レジスタの関連するビットの変更、更新された汎用レジスタ値の PSR への復元（MSR 命令を使用）などの方法を使用する必要があります。たとえば、ARM を他の特権モードからスーパバイザモードに切り替えるには、次の方法が適切です。

```
MRS   R0,CPSR                   ; Read CPSR
BIC   R0,R0,#0x1F               ; Modify by removing current mode
ORR   R0,R0,#0x13               ; and substituting Supervisor mode
MSR   CPSR_c,R0                 ; Write the result back to CPSR
```

最大の効率を得るには、MSR 命令の書き込みは変更の可能性のあるフィールドにのみ行う必要があります。例えば、上記コードの最後の命令は、CPSR 制御フィールドだけは変更できますが、他のフィールドのすべてのビットは最初の命令でCPSRから読み出されているため変更されることはありません。そのため、命令の書き込みは CPSR_c に行い、CPSR_fsxc や他の組合せのフィールドには書き込みません。

ただし、MSR 命令による変更ができない*唯一の*理由が、その時点で割り当てられているビットがないための場合は、そのフィールドに書き込みを行い、将来の互換性を保証する必要があります。

イミディエート形式の MSR 命令を使用すると PSR の任意のフィールドを設定できますが、前述のリード・モディファイ・ライトシーケンスの使用には注意が必要です。イミディエート形式の命令は、関連する PSR を読み出し、関連するフィールド内の全ビットをイミディエート定数の対応するビットで置き換え、その結果を PSR にライトバックすることに相当します。このため、イミディエート形式は指定されたフィールド内のビットすべてを変更する目的でのみ使用し、特にそのフィールドにまだ割り当てられていないビットがある場合には使用できません。この規則に従わない場合、ARM アーキテクチャの将来のバージョンで、そのコードが予期しない効果を引き起こす可能性があります。

上記の規則の例外として、イミディエート形式の命令によるフラグバイトの変更が挙げられます。PSR のビット [26:25] には現時点で割り当てられた機能はありませんが、この場合は正当な処理とみなされます。例えば、MSR 命令を使用して以下の 4 つのフラグすべてをセットできます。また、プロセッサに拡張 DSP の機能を実装する場合は Q フラグをクリアすることができます。

```
MSR    CPSR_f,#0xF0000000
```

ARM アーキテクチャの将来のバージョンでビット [26:25] に割り当てられる機能は、そのようなコードが予期しない効果を引き起こさないように設計されます。いくつかのビットは、予約済みの値に変更することはできません。変更した場合、予測不能な結果を招きます。たとえば、予約された値をモードビット（4:0）に書き込むことや、J ビット（24）を変更することはできません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R ビット　命令のビット[22]が0の場合はCPSRが書き込まれ、1の場合はSPSRが書き込まれます。

ユーザモード **CPSR**

特権ビットまたは実行状態ビットへの書き込みは無視されます。

ユーザモード **SPSR**

ユーザモードで SPSR にアクセスした場合、予測不能な結果を招きます。

システムモード **SPSR**

システムモードで SPSR にアクセスした場合、予測不能な結果を招きます。

現在は使用されていないフィールド仕様

CPSR、CPSR_flg、CPSR_ctl、CPSR_all、SPSR、SPSR_flg、SPSR_ctl、SPSR_all 形式の PSR フィールド仕様は、現在は csxf 形式に移行しています（P. A4-77 参照）。

CPSR、SPSR、CPSR_all、SPSR_all は、フィールドマスク 0b1001 を生成します。

CPSR_flg および SPSR_flg は、フィールドマスク 0b1000 を生成します。

CPSR_ctl および SPSR_ctl は、フィールドマスク 0b0001 を生成します。

T ビットまたは **J** ビット

MSR命令を使用してCPSRのTビットまたはJビットを変更しないで下さい。変更しようとすると、予測不能な結果を招きます。

アドレシングモード

イミディエイトおよびレジスタ形式は、アドレシングモード 1 のイミディエート形式およびシフトされていないレジスタ形式の場合と正確に同じ方法で指定されます（P. A5-2 *「アドレシングモード 1 - データ処理オペランド」* 参照）。アドレシングモード 1 の他の形式はすべて、予測不能な結果を引き起こします。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.40 MUL

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	0	0	0	0	S	Rd	SBZ	Rs	1	0	0	1	Rm

MUL（乗算）命令は、符号付きまたは符号なしの 2 つの 32 ビット値を乗算します。結果の最下位 32 ビットが、デスティネーションレジスタに書き込まれます。

MUL 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
MUL{<cond>}{S}  <Rd>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S 命令の S ビット（ビット [20]）に 1 をセットし、乗算の結果に従って CPSR の N フラグと Z フラグをセットすることを指定します。S を省略すると、命令の S ビットは 0 にセットされ、CPSR 全体が命令の影響を受けません。

<Rd> 命令のデスティネーションレジスタ。

<Rm> 乗算する最初の値を含むレジスタを指定します。

<Rs> <Rm> の値に乗算される値を保持します。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = unaffected in v5 and above, UNPREDICTABLE in v4 and earlier
        V Flag = unaffected
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 <Rd>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

符号付きと符号なし MUL は 64 ビットの積のうち下位 32 ビットのみを生成するため、MUL の結果は符号付きと符号なしのどちらの数値についても同じです。

C フラグ ARM アーキテクチャの v5 以降では、MULS 命令は C フラグを変更しません。以前のバージョンのアーキテクチャでは、MULS 命令の実行後の C フラグの値は予測不能です。

オペランドの制限 以前の説明には、<Rd> と <Rm> に同じレジスタを指定した場合は、予測不能な結果を招くことが記載されていました。高性能の乗算器では結果をライトバックする前にすべてのオペランドを読み込むため、ARMv6 では制限は全くなく、該当する ARM の v4 および v5 のどの実装でも、この制限は必要ありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.41 MVN

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	0	I	1	1	1	1	S	SBZ	Rd	shifter_operand

MVN（論理否定）命令は、論理演算により値の 1 の補数を生成します。この値はイミディエート値またはレジスタの値で、MVN 演算の前にシフトさせることができます。

MVN 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
MVN{<cond>}{S}  <Rd>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S 命令の S ビット（ビット [20]）に 1 をセットし、この命令で CPSR を更新することを指定します。S を省略すると S ビットは 0 にセットされ、CPSR は命令によって変更されません。S が指定された場合の CPSR の更新には、2 タイプあります。

- <Rd> が R15 でない場合、N フラグと Z フラグが演算の結果によってセットされ、シフタによって生成されたキャリーの出力ビットに C フラグがセットされます（P. A5-2 *「アドレシングモード 1 - データ処理オペランド」* 参照）。V フラグと CPSR の他のビットは変更されません。
- <Rd> が R15 の場合、現在のモードの SPSR が CPSR にコピーされます。ユーザモードまたはシステムモードには SPSR がないため、これらのモードで実行した場合、この形式の命令は予測不能な結果を招きます。

<Rd> デスティネーションレジスタ。

<shifter_operand>

オペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」* を参照して下さい。

I ビットが 0 で、かつ shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は MVN ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-32 *「命令セットの拡張」* を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    Rd = NOT shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

用法

MVN を使用すると以下の処理を実行できます。

- ビットマスクを形成する。
- 値の 1 の補数を算出する。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.42 ORR

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	1	1	0	0	S	Rn	Rd	shifter_operand

ORR（論理和）命令は、2 つの値の（算入した）論理和をビットごとに実行します。最初の値はレジスタに含まれています。2 番目の値はイミディエート値またはレジスタの値で、論理和を実行する前にシフト可能です。

ORR 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
ORR{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S 命令の S ビット（ビット [20]）に 1 をセットし、この命令で CPSR を更新することを指定します。S を省略すると S ビットは 0 にセットされ、CPSR は命令によって変更されません。S が指定された場合の CPSR の更新には、2 タイプあります。

- <Rd> が R15 でない場合、N フラグと Z フラグが演算の結果によってセットされ、シフタによって生成されたキャリーの出力ビットに C フラグがセットされます（P. A5-2「*アドレシングモード 1 - データ処理オペランド*」参照）。V フラグと CPSR の他のビットは変更されません。
- <Rd> が R15 の場合、現在のモードの SPSR が CPSR にコピーされます。ユーザモードまたはシステムモードには SPSR がないため、これらのモードで実行した場合、この形式の命令は予測不能な結果を招きます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<shifter_operand>

2 番目のオペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2「*アドレシングモード 1 - データ処理オペランド*」を参照して下さい。

I ビットが 0 で、かつ shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は ORR ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-32「*命令セットの拡張*」を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = Rn OR shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

用法

ORR 命令を使用すると、レジスタで選択したビットをセットできます。ビットごとに、1 との論理和ではビットがセットされ、0 との論理和ではビットは変更されません。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.43 PKHBT

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 7	6 4	3 0
cond	0 1 1 0 1 0 0 0	Rn	Rd	shift_imm	0 0 1	Rm

PKHBT 命令は、最初のオペランドの下位（最下位）ハーフワードと、2 番目のオペランドのシフトした上位（最上位）ハーフワードを組み合わせます。シフトは 0 〜 31 の任意の量による左シフトです。

構文

```
PKHBT {<cond>} <Rd>, <Rn>, <Rm> {, LSL #<shift_imm>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。このオペランドのビット [15:0] は、演算結果の [15:0] になります。

<Rm> 2 番目のオペランドを含むレジスタ。これは指定した量で左シフトされ、2 番目のオペランドのビット [31:16] は、演算結果のビット [31:16] となります。

<shift_imm> <Rm> を左にシフトするビット数を指定します。これは 0 〜 31 の値で、シフト指定子が省略されている場合は、シフトするビット数は 0 です。

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[15:0] = Rn[15:0]
    Rd[31:16] = (Rm Logical_Shift_Left shift_imm)[31:16]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

Rd のワードを、最上位ハーフワードはレジスタ Ra の上半分、最下位のハーフワードはレジスタ Rb の下半分となるように構成するには、次のように指定します。

```
PKHBT   Rd, Rb, Ra
```

Rd のワードを、最上位ハーフワードはレジスタ Ra の下半分、最下位のハーフワードはレジスタ Rb の下半分となるように構成するには、次のように指定します。

```
PKHBT   Rd, Rb, Ra, LSL #16
```

注

R15 の使用　<Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.44 PKHTB

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 7	6 4	3 0
cond	0 1 1 0 1 0 0 0	Rn	Rd	shift_imm	1 0 1	Rm

PKHBT 命令は、最初のオペランドの上位（最上位）ハーフワードと、シフトした 2 番目のオペランドの下位（最下位）ハーフワードを組み合わせます。シフトは 1 ～ 32 の任意の量による算術右シフトです。

構文

```
PKHTB {<cond>} <Rd>, <Rn>, <Rm> {, ASR #<shift_imm>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。このオペランドのビット [31:16] は、演算結果の [31:16] になります。

<Rm> 2 番目のオペランドを含むレジスタ。これは、指定した量で算術右シフトされ、結果として生成される 2 番目のオペランドのビット [15:0] が演算結果のビット [15:0] となります。

<shift_imm> <Rm> を右にシフトするビット数を指定します。シフト数が 32 の場合、shift_imm == 0 としてエンコードされます。

シフト指定子が省略されている場合、アセンブラによって命令が変換され、PKHBT Rd, Rm, Rn となります。この結果、0 ビットだけ算術右シフトした場合と同じ効果になります。

——— 注 ———

アセンブラがシフト値として #0 を許可し、シフトなしの場合や LSL #0 と同じように扱う場合、ここで ASR #0 も許可する必要があります。これは、シフト指定子を省略することと同じです。

———————

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

操作

```
if ConditionPassed(cond) then
    if shift_imm == 0 then          /* ASR #32 case */
        if Rm[31] == 0 then
            Rd[15:0] = 0x0000
        else
            Rd[15:0] = 0xFFFF
    else
        Rd[15:0] = (Rm Arithmetic_Shift_Right shift_imm)[15:0]
    Rd[31:16] = Rn[31:16]
```

用法

Rd のワードを、最上位ハーフワードはレジスタ Ra の上半分、最下位ハーフワードはレジスタ Rb の上半分となるように構成するには、次のように指定します。

```
    PKHTB   Rd, Ra, Rb, ASR #16
```

この命令では、Rb の Q31 数を切り捨て、Rd の下半分にその結果を配置することができます。異なるシフト量を使用すれば、Rb 値の位取りが可能です。

Rd のワードを、最上位ハーフワードはレジスタ Ra の上半分、最下位ハーフワードはレジスタ Rb の下半分となるように構成するには、次のように指定します。

```
    PKHTB   Rd, Ra, Rb
```

この命令は、アセンブラにより次のように変換されます。

```
    PKHBT   Rd, Rb, Ra
```

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.45 PLD

31 30 29 28	27 26	25	24	23	22 21 20	19 16	15 14 13 12	11 0
1 1 1 1	0 1	I	1	U	1 0 1	Rn	1 1 1 1	addr_mode

PLD（データのプリロード）命令は、メモリシステムに対して、近い将来に指定されたアドレスからのメモリアクセスが発生する可能性が高いことを通知します。メモリシステムは、そのメモリアクセスが実際に発生した場合に、メモリアクセスの処理速度を上げるると想定されるアクションを行うことで対応できます。高速化を促すアクションとしては、指定したアドレスを含むキャッシュラインをキャッシュにプリロードすることなどが考えられます。PLD は、*ヒント*命令の一種で、メモリシステムのパフォーマンスの最適化を目的としています。アーキテクチャで定義された効果はなく、この最適化をサポートしていないメモリシステムは無視してかまいません。このようなメモリシステムでは、PLD は NOP として動作します。

構文

```
PLD  <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<addressing_mode>

P. A5-18 *「アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア」*を参照して下さい。これによって命令の I、U、Rn、addr_mode の各ビットが決定されます。この命令では、P == 1 および W == 0 が指定されたアドレシングモードのみが使用できます。プリインデックスおよびポストインデックスのアドレシングモードは P == 0 または W == 1 なため、この命令では使用できません。

アーキテクチャのバージョン

バージョン 5TE 以降に存在します。

例外

なし

動作

```
/* No change occurs to programmer's model state, but where
* appropriate, the memory system is signaled that memory accesses
* to the specified address are likely in the near future.
*/
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

条件　他の大部分の ARM 命令とは異なり、PLD は実行付きで実行できません。

ライトバック　ビット [24]（P ビット）のクリアまたはビット [21]（W ビット）のセットを行うと、予測不能な結果を招きます。

データアボート

この命令により、VMSA MMU、PMSA MPU、その他のメモリシステムで生成される正確なデータアボートのシグナルが送られることはありません。この命令によって引き起こされる可能性があるメモリシステムの例外には、この他に不正確なデータアボートの使用や、何らかの例外メカニズムによるものが考えられます。

アライメント　`<addressing_mode>` で生成されるアドレスに対しては、アライメントの制約条件はありません。システム制御コプロセッサが実装に含まれる場合（第 B3 章*「システム制御コプロセッサ」*参照）、いかなる PLD 命令に対しても、アライメント例外を生成できません。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.46 QADD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	1	0	0	0	0	Rn	Rd	SBZ	0	1	0	1	Rm

QADD（飽和加算）命令は、整数の加算を実行します。結果は 32 ビット符号付き整数の範囲である $-2^{31} \leq x \leq 2^{31} - 1$ に飽和します。

飽和が発生すると、QADD 命令により CPSR の Q フラグがセットされます。

構文

```
QADD{<cond>}  <Rd>, <Rm>, <Rn>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> 最初のオペランドを含むレジスタ。

<Rn> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

バージョン 5TE 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = SignedSat(Rm + Rn, 32)
    if SignedDoesSat(Rm + Rn, 32) then
        Q Flag = 1
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

飽和整数と Q31 の加算の実行と同様に、QADD を SMUL<x><y>、SMULW<y>、SMULL 命令と組み合わせて使用することで、Q15 数と Q31 数の乗算を行うことができます。以下に 3 つの例を紹介します。

- R0 と R1 の下位半分にある Q15 数で乗算を行い、R2 に Q31 の結果を配置するには、次の命令を使用します。

```
SMULBB  R2, R0, R1
QADD    R2, R2, R2
```

- R1 の上位半分にある Q15 数と R0 の Q31 数を乗算し、R2 に Q31 の結果を配置するには、次の命令を使用します。

```
SMULWT  R2, R0, R1
QADD    R2, R2, R2
```

- R0 と R1 にある Q31 数の乗算を行い、R2 に Q31 の結果を配置するには、次の命令を使用します。

```
SMULL   R3, R2, R0, R1
QADD    R2, R2, R2
```

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

条件フラグ QADD 命令は、N、Z、C、V フラグを変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.47 QADD16

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	0	1	0	Rn	Rd	SBO	0	0	0	1	Rm

QADD16 は、16 ビット整数の 2 つの加算を実行します。結果は 16 ビット符号付き整数の範囲である $-2^{15} \leq x \leq 2^{15} - 1$ に飽和します。

QADD16 はフラグを変更しません。

構文

```
QADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[15:0]   = SignedSat(Rn[15:0]   + Rm[15:0],  16)
    Rd[31:16] = SignedSat(Rn[31:16] + Rm[31:16], 16)
```

用法

QADD16 命令は、符号付き飽和演算であることを除いて、SADD16 命令と同様に使用します。QADD16 命令は、SEL で使用する GE ビットの設定を行いません。詳細については、P. A4-120「*SADD16*」を参照して下さい。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.48 QADD8

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	0	1	0	Rn	Rd	SBO	1	0	0	1	Rm

QADD8 は、8 ビット整数の 4 つの加算を実行します。結果は 8 ビット符号付き整数の範囲である $-2^7 \leq x \leq 2^7 - 1$ に飽和します。

QADD8 はフラグを変更しません。

構文

```
QADD8{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[7:0]   = SignedSat(Rn[7:0]   + Rm[7:0],   8)
    Rd[15:8]  = SignedSat(Rn[15:8]  + Rm[15:8],  8)
    Rd[23:16] = SignedSat(Rn[23:16] + Rm[23:16], 8)
    Rd[31:24] = SignedSat(Rn[31:24] + Rm[31:24], 8)
```

用法

QADD8 命令は、符号付き飽和演算であることを除いて、SADD8 命令と同様に使用します。QADD8 命令は、SEL で使用する GE ビットの設定を行いません。詳細については、P. A4-122「*SADD8*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 `<Rd>`、`<Rm>`、`<Rn>` のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.49 QADDSUBX

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	0	1	0	Rn	Rd	SBO	0	0	1	1	Rm

QADDSUBX（交換飽和加減算）は、16 ビット整数の加算と 16 ビットの減算を 1 つづつ実行します。結果は 16 ビット符号付き整数の範囲である $-2^{15} \leq x \leq 2^{15} - 1$ に飽和します。QADDSUBX 命令は、演算を実行する前に 2 番目のオペランドの 2 つのハーフワードを交換します。

QADDSUBX はフラグを変更しません。

構文

```
QADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[31:16] = SignedSat(Rn[31:16] + Rm[15:0],  16)
    Rd[15:0]  = SignedSat(Rn[15:0]  - Rm[31:16], 16)
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

QADDSUBX 命令を使用して、16 ビット整数や Q15 数のペアとして格納されている複素数の演算を行うことができます。レジスタの最下位と最上位にそれぞれ複素数の実数部と虚数部がある場合、次の命令を使用します。

```
QADDSUBX  Rd, Ra, Rb
```

この命令は、Rd = (Ra + i * Rb) という複素算術演算を行います。

QADDSUBX 命令は、いずれかの演算で飽和が発生しても、Q フラグのセットを行いません。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.50 QDADD

31–28	27 26 25 24 23 22 21 20	19–16	15–12	11–8	7 6 5 4	3–0
cond	0 0 0 1 0 1 0 0	Rn	Rd	SBZ	0 1 0 1	Rm

QDADD（飽和倍演算＋加算）命令は、2 番目オペランドを 2 倍してから、結果を最初のオペランドに加算します。

倍演算と加算の結果はいずれも、32 ビット符号付き整数の範囲である $-2^{31} \leq x \leq 2^{31} - 1$ に飽和します。

倍演算と加算のどちらかで飽和が発生した場合、CPSR の Q フラグがセットされます。

構文

```
QDADD{<cond>}  <Rd>, <Rm>, <Rn>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> 最初のオペランドを含むレジスタ。

<Rn> 倍演算および飽和演算を実行し、飽和加算の 2 番目のオペランドとして使用される値を格納するレジスタ。

アーキテクチャのバージョン

バージョン 6 以降およびバージョン 5E に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = SignedSat(Rm + SignedSat(Rn*2, 32), 32)
    if SignedDoesSat(Rm + SignedSat(Rn*2, 32), 32) or
        SignedDoesSat(Rn*2, 32) then
        Q Flag = 1
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

この命令は主に、整数乗算命令の後に配置して、Q15 数および Q31 数の積和演算を生成するために使用されます。以下に 3 つの例を紹介します。

- R4 と R5 の上位半分にある Q15 数で乗算を行い、その積を R6 にある Q31 数に加算するには、次の命令を使用します。

```
SMULTT  R0, R4, R5
QDADD   R6, R6, R0
```

- R2 の下位半分にある Q15 数を R3 の Q31 数で乗算し、その積に R7 の Q31 数を加算するには、次の命令を使用します。

```
SMULWB  R0, R3, R2
QDADD   R7, R7, R0
```

- R2 と R3 にある Q31 数を乗算し、その積に R4 の Q31 数を加算するには、次の命令を使用します。

```
SMULL   R0, R1, R2, R3
QDADD   R4, R4, R1
```

注

R15 の使用	<Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。
条件フラグ	QDADD 命令は、N、Z、C、V フラグを変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.51 QDSUB

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
cond	0	0	0	1	0	1	1	0	Rn	Rd	SBZ	0	1	0	1	Rm

QDSUB（飽和倍演算＋減算）命令は、2 番目オペランドを 2 倍してから、結果を第 1 オペランドから減算します。

倍演算と減算の結果はいずれも、32 ビット符号付き整数の範囲である $-2^{31} \le x \le 2^{31} - 1$ に飽和します。

倍演算と減算のどちらかで飽和が発生した場合、CPSR の Q フラグがセットされます。

構文

```
QDSUB{<cond>}  <Rd>, <Rm>, <Rn>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> 最初のオペランドを含むレジスタ。

<Rn> 倍演算および飽和演算を実行し、飽和減算の 2 番目のオペランドとして使用される値を格納するレジスタ。

Rm および Rn は ARM 命令の大半とは異なり、アセンブラの構文では逆順になります。

アーキテクチャのバージョン

バージョン 6 以降およびバージョン 5E に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = SignedSat(Rm - SignedSat(Rn*2, 32), 32)
    if SignedDoesSat(Rm - SignedSat(Rn*2, 32), 32) or
        SignedDoesSat(Rn*2, 32) then
        Q Flag = 1
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

この命令は主に、整数乗算命令の後に配置して、Q15 数および Q31 数の積減算を生成するために使用されます。以下に 3 つの例を紹介します。

- R4 の上位半分と R5 の下位半分にある Q15 数で乗算を行い、その積を R6 にある Q31 数から減算するには、次の命令を使用します。

```
SMULTB  R0, R4, R5
QDSUB   R6, R6, R0
```

- R2 の下位半分にある Q15 数を R3 の Q31 数で乗算し、その積を R7 の Q31 数から減算するには、次の命令を使用します。

```
SMULWB  R0, R3, R2
QDSUB   R7, R7, R0
```

- R2 と R3 にある Q31 数を乗算し、その積を R4 の Q31 数から減算するには、次の命令を使用します。

```
SMULL   R0, R1, R2, R3
QDSUB   R4, R4, R1
```

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

条件フラグ QDSUB 命令は、N、Z、C、V フラグを変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.52 QSUB

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	1	0	0	1	0	Rn	Rd	SBZ	0	1	0	1	Rm

QSUB（飽和減算）は、整数の減算を実行します。結果は 32 ビット符号付き整数の範囲である $-2^{31} \le x \le 2^{31} - 1$ に飽和します。

飽和が発生すると、QSUB 命令により CPSR の Q フラグがセットされます。

構文

```
QSUB{<cond>}  <Rd>, <Rm>, <Rn>
```

各項目の説明については以下を参照して下さい。

<cond>　この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd>　デスティネーションレジスタ。

<Rm>　最初のオペランドを含むレジスタ。

<Rn>　2 番目のオペランドを含むレジスタ。

Rm および Rn は ARM 命令の大半とは異なり、アセンブラの構文では逆順になります。

アーキテクチャのバージョン

バージョン 6 以降およびバージョン 5E に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = SignedSat(Rm - Rn, 32)
    if SignedDoesSat(Rm - Rn, 32) then
        Q Flag = 1
```

注

R15 の使用　<Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

条件フラグ　QSUB 命令は、N、Z、C、V フラグを変更しません。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.53 QSUB16

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	0	1	0	Rn	Rd	SBO	0	1	1	1	Rm

QSUB16 は、16 ビット減算を 2 つ実行します。結果は 16 ビット符号付き整数の範囲である $-2^{15} \leq x \leq 2^{15} - 1$ に飽和します。

QSUB16 はフラグを変更しません。

構文

```
QSUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[15:0]  = SignedSat(Rn[15:0]  - Rm[15:0],  16)
    Rd[31:16] = SignedSat(Rn[31:16] - Rm[31:16], 16)
```

用法

QSUB16 命令は、符号付き飽和演算であることを除いて、SSUB16 命令と同様に使用します。QSUB16 命令は、SEL で使用する GE ビットの設定を行いません。詳細については、P. A4-181 *「SSUB16」* を参照して下さい。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.54 QSUB8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	0	Rn	Rd	SBO	1	1	1	1	Rm

QSUB8 は、8 ビット減算を 4 つ実行します。結果は 8 ビット符号付き整数の範囲である $-2^7 \le x \le 2^7 - 1$ に飽和します。

QSUB8 はフラグを変更しません。

構文

```
QSUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[7:0]   = SignedSat(Rn[7:0]   - Rm[7:0],   8)
    Rd[15:8]  = SignedSat(Rn[15:8]  - Rm[15:8],  8)
    Rd[23:16] = SignedSat(Rn[23:16] - Rm[23:16], 8)
    Rd[31:24] = SignedSat(Rn[31:24] - Rm[31:24], 8)
```

用法

QSUB8 命令は、符号付き飽和演算であることを除いて、SSUB8 と同様に使用します。QSUB8 命令は、SEL で使用する GE ビットの設定を行いません。詳細については、P. A4-183「*SSUB8*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15の使用 <Rd>、<Rm>、<Rn>のいずれかのレジスタとしてR15を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.55 QSUBADDX

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	0	1	0	Rn	Rd	SBO	0	1	0	1	Rm

QSUBADDX（交換飽和加減算）命令は、16 ビットの符号付き整数の加算と減算を 1 つづつ実行し、結果を飽和演算して 16 ビット符号付き整数にします。結果は 16 ビット符号付き整数の範囲である $-2^{15} \le x \le 2^{15} - 1$ に飽和します。QSUBADDX 命令は、演算を実行する前に 2 番目のオペランドの 2 つのハーフワードを交換します。QSUBADDX はフラグを変更しません。

構文

```
QSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[31:16] = SignedSat(Rn[31:16] - Rm[15:0],  16)
    Rd[15:0]  = SignedSat(Rn[15:0]  + Rm[31:16], 16)
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

QSUBADDX 命令を使用して､16 ビット整数や Q15 数のペアとして格納されている複素数の演算を行うことができます。レジスタの最下位と最上位にそれぞれ複素数の実数部と虚数部がある場合、次の命令を使用します。

```
QSUBADDX  Rd, Ra, Rb
```

この命令は、Rd = (Ra - i * Rb) という複素算術演算を行います。

QSUBADDX 命令は、いずれかの演算で飽和が発生しても、Q フラグのセットを行いません。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.56 REV

31–28	27–23	22	21–20	19–16	15–12	11–8	7	6–4	3–0
cond	0 1 1 0 1	0	1 1	SBO	Rd	SBO	0	0 1 1	Rm

REV（ワードのバイト反転）命令は、32 ビットレジスタのバイト順序を逆にします。

構文

```
REV{<cond>} Rd, Rm
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> オペランドレジスタを指定します。

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[31:24] = Rm[ 7: 0]
    Rd[23:16] = Rm[15: 8]
    Rd[15: 8] = Rm[23:16]
    Rd[ 7: 0] = Rm[31:24]
```

用法

REV は、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、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するために使用します。

注

R15 の使用 レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.57 REV16

31 28	27 23	22	21 20	19 16	15 12	11 8	7	6 4	3 0
cond	0 1 1 0 1	0	1 1	SBO	Rd	SBO	1	0 1 1	Rm

REV16（パックハーフワードバイト反転）命令は、32 ビットレジスタの各 16 ビットハーフワードのバイト順序を反転します。

構文

```
REV16{<cond>} Rd, Rm
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> オペランドレジスタを指定します。

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[15: 8] = Rm[ 7: 0]
    Rd[ 7: 0] = Rm[15: 8]
    Rd[31:24] = Rm[23:16]
    Rd[23:16] = Rm[31:24]
```

用法

REV16 は、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、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するために使用します。

注

R15 の使用 レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.58 REVSH

31 28	27 23	22	21 20	19 16	15 12	11 8	7	6 4	3 0
cond	0 1 1 0 1	1	1 1	SBO	Rd	SBO	1	0 1 1	Rm

REVSH（符号付きハーフワードバイト反転）は、32 ビットレジスタの下位 16 ビットハーフワードのバイト順序を反転し、結果を 32 ビットに符号拡張します。

構文

```
REVSH{<cond>} Rd, Rm
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> オペランドレジスタを指定します。

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[15: 8] = Rm[ 7: 0]
    Rd[ 7: 0] = Rm[15: 8]
    if Rm[7] == 1 then
        Rd[31:16] = 0xFFFF
    else
        Rd[31:16] = 0x0000
```

用法

REVSH は、次のような変換に使用します。

- 16 ビット符号付きの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
- 16 ビット符号付きのリトル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

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.59 RFE

31 30 29 28	27 26 25	24	23	22	21	20	19 16	15 12	11 10 9 8	7 0
1 1 1 1	1 0 0	P	U	0	W	1	Rn	SBZ	1 0 1 0	SBZ

RFE（例外からの復帰）命令は、指定したアドレスのワードとその次のワードから、それぞれ PC と CPSR をロードします。

構文

RFE<addressing_mode> <Rn>{!}

各項目の説明については以下を参照して下さい。

<addressing_mode>

LDMおよびSTM命令の<addressing_mode>と同様です。詳細については、P. A5-41「*アドレシングモード4 - 複数ロード/ストア*」を参照して下さい。差異は以下のとおりです。

- ロードするレジスタの数は 2 つです。
- レジスタリストは {PC, CPSR} です。

<Rn> <addressing_mode>で使用するベースレジスタを指定します。ベースレジスタとして R15 を指定した場合、結果は予測不能です。

! 存在する場合、W ビットをセットします。それによって、命令は修飾済みの値を P. A5-41「*アドレシングモード4 - 複数ロード/ストア*」で指定されているのと同様の方法でベースレジスタに書き戻しします。! が省略された場合、W ビットは 0 になり、ベースレジスタが命令によって変更されることはありません。

アーキテクチャのバージョン

バージョン 6 以降に存在します。

例外

データアボート

用法

RFE は多様なベースレジスタをサポートしていますが、R13 に格納されている Rn == sp（スタックポインタ）を使用するのが一般的です。この命令は SRS および CPS の各命令に関連する復帰手法として使用できます。詳細については、P. A2-28「*例外処理を改善する新しい命令*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
address = start_address
value = Memory[address,4]
If InAPrivilegedMode() then
    CPSR = Memory[address+4,4]
else
    UNPREDICTABLE
PC = value

assert end_address == address + 8
```

ここで、`start_address` と `end_address` は、P. A5-41 *「アドレシングモード 4 - 複数ロード / ストア」*の記述に従って決定されます。ただし、`Number_Of_Set_Bits_in(register_list)` は、命令のビット [15:0] にかかわらず 2 として評価されます。

注

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21 *「データアボート（データアクセス中に発生するメモリアボート）」* を参照して下さい。

非ワード境界アライメントアドレス

ARMv6 では、CP15 レジスタ 1 のビット U == 1 またはビット A == 1 の場合に、ビット [1:0] != 0b00 のアドレスがアライメント例外を引き起こし、それ以外の場合は `RFE` によりビット [1:0] = 0b00 として処理されます。

ARMv6 以前のバージョンでは、システム制御コプロセッサが実装に含まれている場合（第 B3 章 *「システム制御コプロセッサ」* 参照）、CP15 のレジスタ 1 のビット A == 1 の場合に、ビット [1:0] != 0b00 のアドレスがアライメント例外を引き起こし、それ以外の場合には `RFE` によりビット [1:0] = 0b00 として処理されます。

時間順序　`RFE` 命令で生成されたメモリの各ワードにアクセスする時間順序は、アーキテクチャ上では定義されていません。この命令は、アクセス順序が関係するメモリマップされた I/O ロケーションには使用しないで下さい。

ユーザモード　`RFE` はユーザモードでは予測不能となります。

条件　他の大部分の ARM 命令とは異なり、`RFE` は実行付きで実行できません。

ARM と Thumb の状態遷移

CPSR の T ビットが 0 で、かつ PC にロードされた値のビット [1] が 1 の場合、結果は予測不能です。これは、非ワード境界アライメントのアドレスにある ARM 命令への分岐は不可能なためです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.60 RSB

31 ... 28	27	26	25	24	23	22	21	20	19 ... 16	15 ... 12	11 ... 0
cond	0	0	I	0	0	1	1	S	Rn	Rd	shifter_operand

RSB（逆減算）命令は、2 番目の値から最初の値を減算します。

最初の値はレジスタに含まれています。2 番目の値はイミディエート値またはレジスタの値で、減算を実行する前にシフト可能です。この命令は、ARM アセンブラ言語の通常のオペランドとは順序が逆になっています。

RSB 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
RSB{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

`<cond>` この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

`S` 命令の S ビット（ビット [20]）に 1 をセットし、この命令で CPSR を更新することを指定します。S を省略すると S ビットは 0 にセットされ、CPSR は命令によって変更されません。S が指定された場合の CPSR の更新には、2 タイプあります。

- <Rd> が R15 でない場合、N フラグと Z フラグが減算の結果に従ってセットされます。また、減算によって桁下がり（符号なしアンダーフロー）と符号付きオーバフローが生成されたかどうかによって、それぞれ C フラグと V フラグがセットされます。CPSR の他のビットは変更されません。
- <Rd> が R15 の場合、現在のモードの SPSR が CPSR にコピーされます。ユーザモードまたはシステムモードには SPSR がないため、これらのモードで実行した場合、この形式の命令は予測不能な結果を招きます。

`<Rd>` デスティネーションレジスタ。

`<Rn>` 2 番目のオペランドを含むレジスタ。

`<shifter_operand>`

最初のオペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」*を参照して下さい。

I ビットが 0 で、shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は RSB 命令ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-32 *「命令セットの拡張」*を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = shifter_operand - Rn
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(shifter_operand - Rn)
        V Flag = OverflowFrom(shifter_operand - Rn)
```

用法

次の命令は、Rx の符号反転（2 の補数）を Rd に格納します。

```
RSB Rd, Rx, #0
```

次の命令は、Rx を 2^n-1 倍に累乗（定数）算して、Rd に格納します。

```
RSB Rd, Rx, Rx, LSL #n
```

注

C フラグ　S が指定されている場合、C フラグは次のようにセットされます。

1	桁下がりが発生しない場合
0	桁下がりが発生する場合

つまり、C フラグは NOT（桁下がり）フラグとして使用されます。桁下がり条件の反転は、次の命令によって使用されます。SBC と RSC では C フラグを NOT（桁下がり）オペランドとして使用して、C == 1 の場合は通常の減算を実行し、C == 0 の場合は通常より 1 多く減算します。

HS（符号なし＞ =）および LO（符号なし＜）の条件は、それぞれ CS（キャリーセット）および CC（キャリークリア）と同等です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.61 RSC

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	0	I	0	1	1	1	S	Rn	Rd	shifter_operand

RSC（キャリー付逆減算）は、ある値を別の値から減算しますが、その際に、前に行われた下位の減算で生じている桁下がりを考慮します。オペランドの順序が通常と逆になるため、シフトレジスタの値またはイミディエート値からの減算が可能です。

RSC 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
RSC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S 命令の S ビット（ビット [20]）に 1 をセットし、この命令で CPSR を更新することを指定します。S を省略すると S ビットは 0 にセットされ、CPSR は命令によって変更されません。S が指定された場合の CPSR の更新には、2 タイプあります。

- <Rd> が R15 でない場合、N フラグと Z フラグが減算の結果に従ってセットされます。また、減算によって桁下がり（符号なしアンダーフロー）と符号付きオーバフローが生成されたかどうかによって、それぞれ C フラグと V フラグがセットされます。CPSR の他のビットは変更されません。
- <Rd> が R15 の場合、現在のモードの SPSR が CPSR にコピーされます。ユーザモードまたはシステムモードには SPSR がないため、これらのモードで実行した場合、この形式の命令は予測不能な結果を招きます。

<Rd> デスティネーションレジスタ。

<Rn> 2 番目のオペランドを含むレジスタ。

<shifter_operand>

最初のオペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」*を参照して下さい。

I ビットが 0 で、shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は RSC 命令ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-32 *「命令セットの拡張」*を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = shifter_operand - Rn - NOT(C Flag)
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(shifter_operand - Rn - NOT(C Flag))
        V Flag = OverflowFrom(shifter_operand - Rn - NOT(C Flag))
```

用法

RSC を使用すると、シフタレジスタの値またはイミディエート値からの減算を実行するためにオペランドの順序を逆にする必要がある場合に、複数ワード減算を合成できます。

例

次のシーケンスは、R0 と R1 にある 64 ビット値の符号反転を行い（最下位ワードは R0 に保持されています)、結果を R2 と R3 に格納します。

```
    RSBS    R2,R0,#0
    RSC     R3,R1,#0
```

注

C フラグ　S が指定されている場合、C フラグは次のようにセットされます。

1	桁下がりが発生しない場合
0	桁下がりが発生する場合

つまり、C フラグは NOT（桁下がり）フラグとして使用されます。桁下がり条件の反転は、次の命令によって使用されます。SBC と RSC では C フラグを NOT（桁下がり）オペランドとして使用して、C == 1 の場合は通常の減算を実行し、C == 0 の場合は通常より 1 多く減算します。

HS（符号なし＞=）および LO（符号なし＜）の条件は、それぞれ CS（キャリーセット）および CC（キャリークリア）と同等です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.62 SADD16

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	0	0	1	Rn	Rd	SBO	0	0	0	1	Rm

SADD16（符号付き加算）は、16 ビット符号付き整数の加算を 2 つ実行します。加算の結果に従って、CPSR の GE ビットをセットします。

構文

```
SADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    sum       = Rn[15:0] + Rm[15:0] /* Signed addition */
    Rd[15:0]  = sum[15:0]
    GE[1:0]   = if sum >= 0 then 0b11 else 0
    sum       = Rn[31:16] + Rm[31:16] /* Signed addition */
    Rd[31:16] = sum[15:0]
    GE[3:2]   = if sum >= 0 then 0b11 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

SADD16 命令を使用して、ハーフワードデータの配列の演算を高速化できます。たとえば、以下のような命令シーケンスを考えます。

```
LDR     R3, [R0], #4
LDR     R5, [R1], #4
SADD16  R3, R3, R5
STR     R3, [R2], #4
```

この命令シーケンスは、次のものと同じ演算を実行します。

```
LDRH    R3, [R0], #2
LDRH    R4, [R1], #2
ADD     R3, R3, R4
STRH    R3, [R2], #2
LDRH    R3, [R0], #2
LDRH    R4, [R1], #2
ADD     R3, R3, R4
STRH    R3, [R2], #2
```

最初のシーケンスで使用する命令の数は、2 番目のシーケンスの半分です。通常は使用するサイクルの数も半分になります。

SADD16命令を使用して、16ビット整数やQ15数のペアとして格納されている複素数の演算を行うこともできます。レジスタの最下位と最上位にそれぞれ複素数の実数部と虚数部がある場合、次の命令を使用します。

```
SADD16  Rd, Ra, Rb
```

この命令は、Rd = Ra + Rb という複素算術演算を行います。

SADD16 命令を実行すると、各加算の結果に従って GE フラグが設定されます。これらは、次の SEL 命令で使用できます。詳細については、P. A4-128 「*SEL*」を参照して下さい。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.63 SADD8

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	0	0	1	Rn	Rd	SBO	1	0	0	1	Rm

SADD8 は、8 ビット符号付き整数の加算を 4 つ実行します。加算の結果に従って、CPSR の GE ビットをセットします。

構文

```
SADD8{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    sum       = Rn[7:0] + Rm[7:0]     /* Signed addition */
    Rd[7:0]   = sum[7:0]
    GE[0]     = if sum >= 0 then 1 else 0
    sum       = Rn[15:8] + Rm[15:8]   /* Signed addition */
    Rd[15:8]  = sum[7:0]
    GE[1]     = if sum >= 0 then 1 else 0
    sum       = Rn[23:16] + Rm[23:16] /* Signed addition */
    Rd[23:16] = sum[7:0]
    GE[2]     = if sum >= 0 then 1 else 0
    sum       = Rn[31:24] + Rm[31:24] /* Signed addition */
    Rd[31:24] = sum[7:0]
    GE[3]     = if sum >= 0 then 1 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

SADD8 命令を使用して、バイトデータの配列の演算を高速化できます。この処理は SADD16 命令の場合と同様です。詳細については、P. A4-120「*SADD16*」の「用法」を参照して下さい。

SADD8 命令を実行すると、各加算の結果に従って GE フラグが設定されます。これらは、次の SEL 命令で使用できます。詳細については、P. A4-128「*SEL*」を参照して下さい。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.64 SADDSUBX

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	0	0	1	Rn	Rd	SBO	0	0	1	1	Rm

SADDSUBX（交換符号付き加減算）は、16 ビット符号付き整数加算と、16 ビット符号付き整数減算を 1 つづつ実行します。演算を実行する前に、2 番目のオペランドにある 2 つのハーフワードの交換を行います。加算の結果に従って、CPSR の GE ビットをセットします。

構文

```
SADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    sum       = Rn[31:16] + Rm[15:0]     /* Signed addition */
    Rd[31:16] = sum[15:0]
    GE[3:2]   = if sum >= 0 then 0b11 else 0
    diff      = Rn[15:0] - Rm[31:16]    /* Signed subtraction */
    Rd[15:0]  = diff[15:0]
    GE[1:0]   = if diff >= 0 then 0b11 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

`SADDSUBX` 命令を使用して､16 ビット整数や Q15 数のペアとして格納されている複素数の演算を行うことができます。レジスタの最下位と最上位にそれぞれ複素数の実数部と虚数部がある場合、次の命令を使用します。

```
SADDSUBX  Rd, Ra, Rb
```

この命令は、Rd = Ra + (i * Rb) という複素算術演算を行います。

`SADDSUBX` 命令を実行すると､演算の結果に従って GE フラグが設定されます｡これらは､次の `SEL` 命令で使用できます。詳細については、P. A4-128「*SEL*」を参照して下さい。

注

R15 の使用 `<Rd>`、`<Rm>`、`<Rn>` のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.65 SBC

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	0	I	0	1	1	0	S	Rn	Rd	shifter_operand

SBC（キャリー付き減算）は、最初のオペランドの値から、2 番目のオペランドの値と NOT（キャリーフラグ）の値を減算します。最初のオペランドはレジスタに含まれています。2 番目のオペランドはイミディエート値またはレジスタの値で、減算を実行する前にシフト可能です。

SBC を使用して複数ワードの減算を合成します。

SBC 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
SBC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

<cond>
: この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S
: 命令の S ビット（ビット [20]）に 1 をセットし、この命令で CPSR を更新することを指定します。S を省略すると S ビットは 0 にセットされ、CPSR は命令によって変更されません。S が指定された場合の CPSR の更新には、2 タイプあります。

  - <Rd> が R15 でない場合、N フラグと Z フラグが減算の結果に従ってセットされます。また、減算によって桁下がり（符号なしアンダーフロー）と符号付きオーバフローが生成されたかどうかによって、それぞれ C フラグと V フラグがセットされます。CPSR の他のビットは変更されません。
  - <Rd> が R15 の場合、現在のモードの SPSR が CPSR にコピーされます。ユーザモードまたはシステムモードには SPSR がないため、これらのモードで実行した場合、この形式の命令は予測不能な結果を招きます。

<Rd>
: デスティネーションレジスタ。

<Rn>
: 最初のオペランドを含むレジスタ。

<shifter_operand>
: 2 番目のオペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」* を参照して下さい。

  I ビットが 0 で、shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は SBC 命令ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-32 *「命令セットの拡張」* を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = Rn - shifter_operand - NOT(C Flag)
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(Rn - shifter_operand - NOT(C Flag))
        V Flag = OverflowFrom(Rn - shifter_operand - NOT(C Flag))
```

用法

R0 と R1 および R2 と R3 の各レジスタペアに 64 ビットの値が保持されている場合（R0 と R2 に最下位ワードが保持されているとします）、次の命令シーケンスを実行すると、64 ビットの差分が R4 と R5 に格納されます。

```
    SUBS    R4,R0,R2
    SBC     R5,R1,R3
```

注

C フラグ　S が指定されている場合、C フラグは次のようにセットされます。

1　桁下がりが発生しない場合

0　桁下がりが発生する場合

つまり、C フラグは NOT（桁下がり）フラグとして使用されます。桁下がり条件の反転は、次の命令によって使用されます。SBC と RSC では C フラグを NOT（桁下がり）オペランドとして使用して、C == 1 の場合は通常の減算を実行し、C == 0 の場合は通常より 1 多く減算します。

HS（符号なし＞ =）および LO（符号なし＜）の条件は、それぞれ CS（キャリーセット）および CC（キャリークリア）と同等です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.66 SEL

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	1	0	0	0	Rn	Rd	SBO	1	0	1	1	Rm

SEL（選択）は、GE フラグの値に基づいて、実行結果の各バイトを最初のオペランドまたは 2 番目のオペランドから選択します。

構文

```
SEL{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[7:0]   = if GE[0] == 1 then Rn[7:0]   else Rm[7:0]
    Rd[15:8]  = if GE[1] == 1 then Rn[15:8]  else Rm[15:8]
    Rd[23:16] = if GE[2] == 1 then Rn[23:16] else Rm[23:16]
    Rd[31:24] = if GE[3] == 1 then Rn[31:24] else Rm[31:24]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

SEL 命令は、GE フラグを設定する命令（SADD8、SADD16、SSUB8、SSUB16、UADD8、UADD16、USUB8、USUB16、SADDSUBX、SSUBADDX、UADDSUBX、USUBADDX など）の後で使用します。たとえば次の命令シーケンスは、Rd の各バイトを、Ra と Rb で対応するバイトの符号なし最小値に設定します。

```
USUB8  Rd, Ra, Rb
SEL    Rd, Rb, Ra
```

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.67 SETEND

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 - 10	9	8	7	6	5	4	3 - 0
1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	SBZ	E	SBZ	0	0	0	0	SBZ

SETEND は CPSR の E ビットを変更します。CPSR の他のビットは変更されません。

構文

```
SETEND <endian_specifier>
```

各項目の説明については以下を参照して下さい。

<endian_specifier>

次のいずれかを指定します。

BE　命令の E ビットをセットします。これにより、CPSR の E ビットがセットされます。

LE　命令の E ビットをクリアします。これにより、CPSR の E ビットがクリアされます。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

CPSR = CPSR の E ビットを指定に応じて変更した値

用法

SETEND は、データアクセスのバイト順序を変更するために使用します。SETEND を使用して、リトルエンディアンを主に使用するアプリケーションの中に一連のビッグエンディアンデータフィールドがある場合や、ビッグエンディアンを主に使用するアプリケーションの中に一連のリトルエンディアンデータフィールドがある場合に、アクセスの効率を改善できます。

注

条件　他の大部分の ARM 命令とは異なり、SETEND は実行付きで実行できません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.68 SHADD16

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 0 1 1	Rn	Rd	SBO	0 0 0 1	Rm

SHADD16（符号付き半加算）は、16 ビット符号付き整数の加算を 2 つ実行して、各演算結果の値を半分にします。GE フラグは変更されません。

構文

```
SHADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    sum       = Rn[15:0] + Rm[15:0] /* Signed addition */
    Rd[15:0]  = sum[16:1]
    sum       = Rn[31:16] + Rm[31:16] /* Signed addition */
    Rd[31:16] = sum[16:1]
```

用法

SHADD16 命令の用途は SADD16 命令（P. A4-120 *「SADD16」* 参照）と類似しています。SHADD16 命令はオペランドの平均値を算出します。オーバフローが発生しないため、この命令によるフラグの設定はありません。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.69 SHADD8

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 0 1 1	Rn	Rd	SBO	1 0 0 1	Rm

SHADD8 は、8 ビット符号付き整数の加算を 4 つ実行して、各演算結果の値を半分にします。GE フラグは変更されません。

構文

```
SHADD8{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    sum       = Rn[7:0] + Rm[7:0] /* Signed addition */
    Rd[7:0]  = sum[8:1]
    sum       = Rn[15:8] + Rm[15:8] /* Signed addition */
    Rd[15:8] = sum[8:1]
    sum       = Rn[23:16] + Rm[23:16] /* Signed addition */
    Rd[23:16]  = sum[8:1]
    sum       = Rn[31:24] + Rm[31:24] /* Signed addition */
    Rd[31:24] = sum[8:1]
```

用法

SHADD8 命令の用途は SADD16 命令（P. A4-120 *「SADD16」* を参照）と類似しています。SHADD8 命令はオペランドの平均値を算出します。オーバフローが発生しないため、この命令によるフラグの設定はありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.70 SHADDSUBX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	1	Rn	Rd	SBO	0	0	1	1	Rm

SHADDSUBX（交換符号付き半加減算）は、16 ビット符号付き整数加算と 16 ビット符号付き整数減算を 1 つづつ実行し、それぞれの結果の値を半分にします。演算を実行する前に、2 番目のオペランドにある 2 つのハーフワードの交換を行います。

SHADDSUBX 命令は GE フラグを変更しません。

構文

```
SHADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    sum       = Rn[31:16] + Rm[15:0]      /* Signed addition */
    Rd[31:16] = sum[16:1]
    diff      = Rn[15:0] - Rm[31:16]      /* Signed subtraction */
    Rd[15:0]  = diff[16:1]
```

用法

SHADDSUBX 命令の用途は SADDSUBX 命令と類似していますが、演算結果の値を半分にする場合に使用します。これらの詳細については、P. A4-124 *「SADDSUBX」* を参照して下さい。

オーバフローが発生しないため、SHADDSUBX 命令によるフラグの設定はありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.71 SHSUB16

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	0	1	1	Rn	Rd	SBO	0	1	1	1	Rm

SHSUB16（符号付き半減算）は、16 ビット符号付き整数の減算を 2 つ実行して、各演算結果の値を半分にします。

SHSUB16 命令は GE フラグを変更しません。

構文

```
SHSUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    diff      = Rn[15:0] - Rm[15:0]    /* Signed subtraction */
    Rd[15:0]  = diff[16:1]
    diff      = Rn[31:16] - Rm[31:16]  /* Signed subtraction */
    Rd[31:16] = diff[16:1]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

SHSUB16 命令を使用して、ハーフワードデータの配列の演算を高速化できます。これは、SADD16 の使用方法と同様です。詳細については、P. A4-120「*SADD16*」の「用法」を参照して下さい。

SHSUB16 命令を使用して、16 ビット整数や Q15 数のペアとして格納されている複素数の演算を行うこともできます。レジスタの最下位と最上位にそれぞれ複素数の実数部と虚数部がある場合、次の命令を使用します。

```
SHSUB16  Rd, Ra, Rb
```

この命令は、Rd = (Ra - Rb) / 2 という複素算術演算を行います。

オーバフローが発生しないため、SHSUB16 命令によるフラグの設定はありません。

注

R15 の使用　　<Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.72 SHSUB8

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	0	1	1	Rn	Rd	SBO	1	1	1	1	Rm

SHSUB8 は、8 ビット符号付き整数の減算を 4 つ実行して、各演算結果の値を半分にします。

SHSUB8 命令は GE フラグを変更しません。

構文

```
SHSUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    diff      = Rn[7:0] - Rm[7:0]       /* Signed subtraction */
    Rd[7:0]   = diff[8:1]
    diff      = Rn[15:8] - Rm[15:8]     /* Signed subtraction */
    Rd[15:8]  = diff[8:1]
    diff      = Rn[23:16] - Rm[23:16]   /* Signed subtraction */
    Rd[23:16] = diff[8:1]
    diff      = Rn[31:24] - Rm[31:24]   /* Signed subtraction */
    Rd[31:24] = diff[8:1]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

SHSUB8 命令を使用して、バイトデータの配列の演算を高速化できます。これは、ハーフワードデータの演算を加速する SADD16 の使用方法と同様です。詳細については、P. A4-120「*SADD16*」の「用法」を参照して下さい。

オーバフローが発生しないため、SHSUB8 命令によるフラグの設定はありません。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.73 SHSUBADDX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	1	Rn	Rd	SBO	0	1	0	1	Rm

SHADDSUBX（交換符号付き半加減算）は、16 ビット符号付き整数減算と 16 ビット符号付き整数加算を 1 つづつ行い、それぞれの結果の値を半分にします。演算を実行する前に、2 番目のオペランドにある 2 つのハーフワードの交換を行います。

SHSUBADDX 命令は GE フラグを変更しません。

構文

```
SHSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    diff      = Rn[31:16] - Rm[15:0]      /* Signed subtraction */
    Rd[31:16] = diff[16:1]
    sum       = Rn[15:0]  + Rm[31:16]     /* Signed addition */
    Rd[15:0]  = sum[16:1]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

SHSUBADDX命令の用途はSSUBADDX命令と類似していますが、演算結果の値を半分にする場合に使用します。これらの詳細については、P. A4-185「*SSUBADDX*」を参照して下さい。

オーバフローが発生しないため、SHSUBADDX 命令によるフラグの設定はありません。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.74 SMLA<x><y>

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
cond	0	0	0	1	0	0	0	0	Rd	Rn	Rs	1	y	x	0	Rm

SMLA<x><y>（符号付き積和 BB、BT、TB、TT）は、符号付き積和演算を実行します。ソースレジスタの下位半分または上位半分のいずれかから取得した 2 つの符号付き 16 ビット値が乗算されます。ソースレジスタの残りの半分は無視されます。32 ビットの積が 32 ビットの累算値に加算され、結果がデスティネーションレジスタに書き込まれます。

累算値の加算によってオーバフローが発生した場合、CPSR の Q フラグがセットされます。乗算中にオーバフローが発生することはありません。

構文

```
SMLA<x><y>{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

各項目の説明については以下を参照して下さい。

<x> 乗算の最初のオペランドとして、ソースレジスタ <Rm> の上位と下位のどちらを使用するかを指定します。<x> が B の場合、命令エンコードに x == 0 が指定され、<Rm> の下位半分（ビット [15:0]）が使用されます。<x> が T の場合、命令エンコードに x == 1 が指定され、<Rm> の上位半分（ビット [31:16]）が使用されます。

<y> 乗算の 2 番目のオペランドとして、ソースレジスタ <Rs> の上位と下位のどちを使用するかを指定します。<y> が B の場合、命令エンコードに y == 0 が指定され、<Rs> の下位半分（ビット [15:0]）が使用されます。<y> が T の場合、命令エンコードに y == 1 が指定され、<Rs> の上位半分（ビット [31:16]）が使用されます。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> 最初のソースレジスタ。このレジスタの下位半分または上位半分（<x> で選択）が乗算の最初のオペランドになります。

<Rs> 2 番目のソースレジスタ。このレジスタの下位半分または上位半分（<y> で選択）が乗算の 2 番目のオペランドになります。

<Rn> 累算値を含むレジスタを指定します。

アーキテクチャのバージョン

ARMv6 以降と ARMv5E に存在します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

```
if ConditionPassed(cond) then

    if (x == 0) then
        operand1 = SignExtend(Rm[15:0])
    else /* x == 1 */
        operand1 = SignExtend(Rm[31:16])

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    Rd = (operand1 * operand2) + Rn
    if OverflowFrom((operand1 * operand2) + Rn) then
        Q Flag = 1
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

整数の積和演算という本来の用途の他に、これらの命令は SMUL<x><y> 命令と QDADD 命令の合成で行われる $Q15 \times Q15 + Q31 \rightarrow Q31$ という積和演算の結果を高速に算出するために使用されることもあります。この用法は、次のような状況で可能です。

- 飽和やオーバフローが演算中に発生しないことが明らかである場合
- 飽和やオーバフローが演算中に発生する可能性があるが、Q フラグによる検知と、発生時の処置が行われる場合

たとえば、次のコードは、R0 および R1 にある 4 つの Q15 数と、R2 および R3 にある 4 つの Q15 数の内積を算出します。

```
SMULBB  R4, R0, R2
QADD    R4, R4, R4
SMULTT  R5, R0, R2
QDADD   R4, R4, R5
SMULBB  R5, R1, R3
QDADD   R4, R4, R5
SMULTT  R5, R1, R3
QDADD   R4, R4, R5
```

飽和が発生しない場合、次のコードを使用するとさらに実行を高速化できます。

```
SMULBB  R4, R0, R2
SMLATT  R4, R0, R2, R4
SMLABB  R4, R1, R3, R4
SMLATT  R4, R1, R3, R4
QADD    R4, R4, R4
```

さらに、2 番目のシーケンスで飽和やオーバフローが発生する場合は、Q フラグが設定されます。これにより、データの値を小さくすることや計算の繰り返しなど、問題に対する処理を行うことができます。

注

R15 の使用 <Rd>、<Rm>、<Rs>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

条件フラグ SMLA<x><y> 命令は、N、Z、C、V フラグを変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.75 SMLAD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	1	0	0	0	0	Rd	Rn	Rs	0	0	X	1	Rm

SMLAD（符号付きデュアル積和演算）は、符号付きの 16 × 16 ビット乗算を 2 つ実行します。その結果に、32 ビット累算オペランドを加算します。

オプションとして、演算を実行する前に 2 番目のオペランドのハーフワードを交換できます。これにより、最上位 x 最下位および最下位 x 最上位の乗算が行われます。

累算によってオーバフローが発生した場合、Q フラグがセットされます。乗算中にオーバフローは発生しません。

構文

```
SMLAD{X}{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

各項目の説明については以下を参照して下さい。

X
: 命令のXビットを1にセットし､最下位x最上位と最上位x最下位の乗算が行われます。

X を省略すると、X ビットは 0 にセットされ、最下位 x 最下位と最上位 x 最上位の乗算が行われます。

<cond>
: この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd>
: デスティネーションレジスタ。

<Rm>
: 最初のオペランドを含むレジスタ。

<Rs>
: 2 番目のオペランドを含むレジスタ。

<Rn>
: 累算オペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
    Rd = Rn + product1 + product2
    if OverflowFrom(Rn + product1 + product2) then
        Q flag = 1
```

用法

SMLAD命令を使用すると、16 ビットデータの積の合計を 32 ビットアキュムレータで累算することができます。この命令を使用してこの処理を行うと、処理速度は他の方法の約 2 倍になります。この方法は、フィルタなど多くのアプリケーションで有効です。

X オプションを使用すると、16 ビットの実数部分と 16 ビットの虚数部分がある複素数に適用される同様のフィルタで使用する虚数部分を計算することができます。

注

R15 の使用 <Rd>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

注

<Rn>レジスタにR15を使用すると、アセンブラによってエラーが生成されます。

エンコード 命令の <Rn> フィールドの値が 0b1111 である場合、この命令は SMUAD 命令（P. A4-165「*SMUAD*」参照）です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

N、Z、C、V フラグ SMLAD 命令は、N、Z、C、V フラグを変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.76 SMLAL

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	0	1	1	1	S	RdHi	RdLo	Rs	1	0	0	1	Rm

SMLAL（符号付き積和演算 long 型）は、符号付きの 32 ビット値を乗算して 64 ビット値にし、この値を 64 ビット値に累算します。

SMLAL 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
SMLAL{<cond>}{S}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S 命令の S ビット（ビット [20]）に 1 をセットし、積和の結果に従って CPSR の N フラグと Z フラグをセットすることを指定します。S を省略すると、命令の S ビットは 0 にセットされ、CPSR 全体が命令の影響を受けません。

<RdLo> <Rm> および <Rs> の積に加算される下位 32 ビット値を含むレジスタで、結果の下位 32 ビットのデスティネーションレジスタでもあります。

<RdHi> <Rm> および <Rs> の積に加算される上位 32 ビット値を含むレジスタで、結果の上位 32 ビットのデスティネーションレジスタでもあります。

<Rm> <Rs> の値に乗算される符号付き値を保持します。

<Rs> <Rm> の値に乗算される符号付き値を保持します。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    RdLo = (Rm * Rs)[31:0] + RdLo /* Signed multiplication */
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

用法

SMLAL 命令を使用すると、符号付き変数が乗算され、64 ビット値が生成されます。この値は 2 つのデスティネーション汎用レジスタにある 64 ビット値に加算されます。結果は 2 つのデスティネーション汎用レジスタに書き戻されます。

注

R15 の使用 <RdHi>、<RdLo>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

オペランドの制限 <RdHi> と <RdLo> が同じレジスタの場合、結果は予測不能です。

以前の説明には、<RdHi> と <Rm> または <RdLo> と <Rm> に同じレジスタを指定した場合は、予測不能な結果を招くことが記載されていました。高性能の乗算器では結果をライトバックする前にすべてのオペランドを読み込むため、ARMv6 では制限は全くなく、該当する ARM の v4 および v5 のどの実装でも、この制限は必要ありません。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

C フラグと V フラグ ARMv5 以降では、SMLALS は C フラグと V フラグが変更されないように定義されています。以前のバージョンのアーキテクチャでは、SMLALS 命令の実行後の C フラグと V フラグの値は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.77 SMLAL<x><y>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	1	0	0	RdHi	RdLo	Rs	1	y	x	0	Rm

SMLAL<x><y>（符号付き積和 long 型 BB、BT、TB、TT）は、符号付き積和演算を実行します。ソースレジスタの下位半分または上位半分のいずれかから取得した2つの符号付き16ビット値が乗算されます。ソースレジスタの残りの半分は無視されます。32 ビットの積は符号拡張され、<RdHi> と <RdLo>に格納されている64ビット累算値に加算されます。その結果は<RdHi>と<RdLo>に書き戻されます。

この命令の実行中にオーバフローが発生する場合がありますが、64 ビット加算の結果として生じるものです。このオーバフローは発生しても検出されません。その代わりに、演算結果はモジュロ 2^{64} にラップアラウンドされます。

構文

```
SMLAL<x><y>{<cond>}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

<x> 乗算の最初のオペランドとして、ソースレジスタ <Rm> の上位と下位のどちらを使用するかを指定します。<x> が B の場合、命令エンコードに x == 0 が指定され、<Rm> の下位半分（ビット [15:0]）が使用されます。<x> が T の場合、命令エンコードに x == 1 が指定され、<Rm> の上位半分（ビット [31:16]）が使用されます。

<y> 乗算の 2 番目のオペランドとして、ソースレジスタ <Rs> の上位と下位のどちを使用するかを指定します。<y> が B の場合、命令エンコードに y == 0 が指定され、<Rs> の下位半分（ビット [15:0]）が使用されます。<y> が T の場合、命令エンコードに y == 1 が指定され、<Rs> の上位半分（ビット [31:16]）が使用されます。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<RdLo> 積に加算する 64 ビット累算値の下位 32 ビットを含むレジスタで、64 ビットの結果の下位 32 ビットのデスティネーションレジスタでもあります。

<RdHi> 積に加算する 64 ビット累算値の上位 32 ビットを含むレジスタで、64 ビットの結果の上位 32 ビットのデスティネーションレジスタでもあります。

<Rm> 最初のソースレジスタ。このレジスタの下位半分または上位半分（<x> で選択）が乗算の最初のオペランドになります。

<Rs> 2 番目のソースレジスタ。このレジスタの下位半分または上位半分（<y> で選択）が乗算の 2 番目のオペランドになります。

アーキテクチャのバージョン

ARMv6 以降と ARMv5E に存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

```
if ConditionPassed(cond) then

    if (x == 0) then
        operand1 = SignExtend(Rm[15:0])
    else /* x == 1 */
        operand1 = SignExtend(Rm[31:16])

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    RdLo = RdLo + (operand1 * operand2)
    RdHi = RdHi + (if (operand1*operand2) < 0 then 0xFFFFFFFF else 0)
                + CarryFrom(RdLo + (operand1 * operand2))
```

用法

これらの命令により、符号付き 16 ビット整数または Q15 数の積和演算の長いシーケンスを実行できます。十分な*ガードビット*があるため、実用上は演算結果が 64 ビットのデスティネーションをオーバフローすることはありません。積和演算を 2^{33} 回以上連続して行わない限り、このようなオーバフローは発生しません。

演算全体を通して符号付き 32 ビット数値からのオーバフローが発生しない場合は、演算結果が <RdLo> に格納されます。

演算中に <RdLo> でオーバフローが発生しているかどうかを簡単に確認するには、次の命令を実行します。

```
CMP     <RdHi>, <RdLo>, ASR #31
```

この命令は演算の最後に実行します。Z フラグがセットされている場合、<RdLo> には正確な最終結果が格納されています。Z フラグがクリアされている場合、最終結果が符号付き 32 ビットデスティネーションからオーバフローしています。

注

R15 の使用 <RdLo>、<RdHi>, <Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

オペランドの制限 <RdLo> と <RdHi> が同じレジスタの場合、結果は予測不能です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

条件フラグ SMLAL<x><y> 命令は、N、Z、C、V、Q フラグを変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.78 SMLALD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	1	0	1	0	0	RdHi	RdLo	Rs	0	0	X	1	Rm

SMLALD（符号付き積和演算 long 型デュアル）は、符号つき 16 × 16 ビット乗算を 2 つ実行します。その結果に、64 ビット累算オペランドを加算します。

オプションとして、演算を実行する前に 2 番目のオペランドのハーフワードを交換できます。これにより、最上位×最下位および最下位×最上位の乗算が行われます。

構文

```
SMLALD{X}{<cond>}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

X
: 命令のXビットを1にセットし､最下位×最上位と最上位×最下位の乗算が行われます。

: X を省略すると、X ビットは 0 にセットされ、最下位×最下位と最上位×最上位の乗算が行われます。

<cond>
: この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<RdLo>
: 積に加算する 64 ビット累算値の下位 32 ビットを含むレジスタで、これが 64 ビットの結果の下位 32 ビットのデスティネーションレジスタでもあります。

<RdHi>
: 積に加算する 64 ビット累算値の上位 32 ビットを含むレジスタで、64 ビットの結果の上位 32 ビットのデスティネーションレジスタでもあります。

<Rm>
: 最初の乗算オペランドを含むレジスタ。

<Rs>
: 2 番目の乗算オペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    accvalue[31:0]  = RdLo
    accvalue[63:32] = RdHi
    product1 = Rm[15:0]  * operand2[15:0]      /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]     /* Signed multiplication */
    result = accvalue + product1 + product2    /* Signed addition */
    RdLo = result[31:0]
    RdHi = result[63:32]
```

用法

SMLALD 命令の用途は SMLAD 命令と類似していますが、32 ビットアキュムレータではなく 64 ビットアキュムレータを使用する場合にこの命令を使用します。通常は、この命令の処理速度の方が遅くなります。詳細については、P. A4-145「*SMLAD*」の「用法」を参照して下さい。

注

R15 の使用　<RdLo>、<RdHi>, <Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

オペランドの制限　<RdLo> と <RdHi> が同じレジスタの場合、結果は予測不能です。

短縮処理　乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

フラグ　SMLALD 命令は、どのフラグも変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.79 SMLAW<y>

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	1	0	0	1	0	Rd	Rn	Rs	1	y	0	0	Rm

SMLAW<y>（符号付きワード積和 B、T）は、符号付き積和演算を実行します。符号付き 32 ビット値と符号付き 16 ビット値が乗算されます。16 ビット値はソースレジスタの下位半分または上位半分のいずれかから取得されます。ソースレジスタの残りの半分は無視されます。48 ビットの積のうち、上位 32 ビットが 32 ビットの累算値に加算され、その結果がデスティネーションレジスタに書き込まれます。48 ビットの積の下位 16 ビットは無視されます。累算値の加算によってオーバフローが発生した場合、CPSR の Q フラグがセットされます。48 ビットの積のうち上位 32 ビットを使用しているため、乗算中にオーバフローは発生しません。

構文

```
SMLAW<y>{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

各項目の説明については以下を参照して下さい。

<y> 乗算の 2 番目のオペランドとして、ソースレジスタ <Rs> の上位と下位のどちらを使用するかを指定します。<y> が B の場合、命令エンコードに y == 0 が指定され、<Rs> の下位半分（ビット [15:0]）が使用されます。<y> が T の場合、命令エンコードに y == 1 が指定され、<Rs> の上位半分（ビット [31:16]）が使用されます。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> 乗算の最初のオペランドである 32 ビット値が格納されているソースレジスタ。

<Rs> 2 番目のソースレジスタ。このレジスタの下位半分または上位半分（<y> で選択）が乗算の 2 番目のオペランドになります。

<Rn> 累算値を含むレジスタを指定します。

アーキテクチャのバージョン

ARMv6 以降と ARMv5E に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    Rd = (Rm * operand2)[47:16] + Rn     /* Signed multiplication */
    if OverflowFrom((Rm * operand2)[47:16] + Rn) then
        Q Flag = 1
```

用法

整数の積和演算という本来の用途の他に、これらの命令は SMULW<y> 命令と QDADD 命令の合成で行われる $Q31 \times Q15 + Q31 \rightarrow$ という積和演算の結果を高速に算出するために使用されることもあります。この用法が使用できる状況とその利点は、SMLA<x><y> 命令の場合と類似しています。詳細については、P. A4-144 *「用法」*を参照して下さい。

注

R15 の使用 <Rd>、<Rm>、<Rs>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

条件フラグ SMLAW<y> 命令は、N、Z、C、V フラグを変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.80 SMLSD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	1	0	0	0	0	Rd	Rn	Rs	0	1	X	1	Rm

SMLSD（符号付き累算積減算デュアル）は、符号つき 16 × 16 ビット乗算を 2 つ実行します。積の差分に、32 ビット累算オペランドを加算します。

オプションとして、演算を実行する前に 2 番目のオペランドのハーフワードを交換できます。これにより、最上位×最下位および最下位×最上位の乗算が行われます。

累算によってオーバフローが発生した場合、Q フラグがセットされます。乗算中および減算中にオーバフローは発生しません。

構文

```
SMLSD{X}{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

各項目の説明については以下を参照して下さい。

X　　命令のXビットを1にセットし､最下位×最上位と最上位×最下位の乗算が行われます。

　　X を省略すると、X ビットは 0 にセットされ、最下位×最下位と最上位×最上位の乗算が行われます。

<cond>　　この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd>　　デスティネーションレジスタ。

<Rm>　　最初の乗算オペランドを含むレジスタ。

<Rs>　　2 番目の乗算オペランドを含むレジスタ。

<Rn>　　累算オペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    product1 = Rm[15:0]  * operand2[15:0]      /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]     /* Signed multiplication */
    diffofproducts = product1 - product2       /* Signed subtraction */
    Rd = Rn + diffofproducts
    if OverflowFrom(Rn + diffofproducts) then
        Q flag = 1
```

用法

SMLSD命令を使用すると、16ビットの実数部分と16ビットの虚数部分がある複素数に適用されている32ビットアキュムレータ付きフィルタで使用する実数部分を計算することができます。

詳細については、P. A4-145「*SMLAD*」の「用法」を参照して下さい。

注

R15 の使用 <Rd>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

——— 注 ———

<Rn>レジスタにR15を使用すると、アセンブラによってエラーが生成されます。

———————

エンコード 命令の <Rn> フィールドの値が 0b1111 である場合、この命令は SMUSD 命令（P. A4-173「*SMUSD*」参照）です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

N、Z、C、V フラグ SMLSD 命令は、N、Z、C、V フラグを変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.81 SMLSLD

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6	5	4	3 0
cond	0 1 1 1 0 1 0 0	RdHi	RdLo	Rs	0 1	X	1	Rm

SMLSLD（符号付き累積乗減算 long 型デュアル）は、符号付き 16 × 16 ビット乗算を 2 つ実行します。積の差分に、64 ビット累算オペランドを加算します。

オプションとして、演算を実行する前に 2 番目のオペランドのハーフワードを交換できます。これにより、最上位×最下位および最下位×最上位の乗算が行われます。

構文

```
SMLSLD{X}{<cond>}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

X
: 命令のXビットを1にセットし､最下位×最上位と最上位×最下位の乗算が行われます。

  X を省略すると、X ビットは 0 にセットされ、最下位×最下位と最上位×最上位の乗算が行われます。

<cond>
: この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<RdLo>
: 積に加算する 64 ビット累算値の下位 32 ビットを含むレジスタで、これが 64 ビットの結果の下位 32 ビットのデスティネーションレジスタでもあります。

<RdHi>
: 積に加算する 64 ビット累算値の上位 32 ビットを含むレジスタで、64 ビットの結果の上位 32 ビットのデスティネーションレジスタでもあります。

<Rm>
: 最初の乗算オペランドを含むレジスタ。

<Rs>
: 2 番目の乗算オペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    accvalue[31:0]  = RdLo
    accvalue[63:32] = RdHi
    product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
    result = accvalue + product1 - product2   /* Signed subtraction */
    RdLo = result[31:0]
    RdHi = result[63:32]
```

用法

この命令の用法は SMLSD 命令（P. A4-155「*SMLSD*」の「用法」を参照）と類似していますが、32 ビットアキュムレータの代わりに 64 ビットアキュムレータを使用する場合にこの命令を使用します。通常の実装では、この命令の結果生成されるフィルタは SMLSD 命令の場合と比べて実行速度が遅くなりますが、オーバフローを防止するガードビットはこの命令の方がはるかに多くなります。

詳細については、P. A4-145「*SMLAD*」の「用法」を参照して下さい。

注

R15 の使用　<Rd>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

オペランドの制限　<RdLo> と <RdHi> が同じレジスタの場合、結果は予測不能です。

短縮処理　乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

フラグ　SMLSD 命令は、どのフラグも変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.82 SMMLA

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	1	0	1	0	1	Rd	Rn	Rs	0	0	R	1	Rm

SMMLA（符号付き最上位ワード積和演算）は、2 つの符号付き 32 ビット値を乗算し、演算結果から最上位の 32 ビットを抽出して、累算値に加算します。

オプションとして、演算結果の切り捨ての代わりに丸めを指定できます。この場合、上位のワードが抽出される前に、定数 0x80000000 が積に加算されます。

構文

```
SMMLA{R}{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

各項目の説明については以下を参照して下さい。

R　　命令の R ビットを 1 にセットします。乗算は丸められます。

R を省略すると R ビットは 0 にセットされ、乗算は切り捨てられます。

<cond>　　この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd>　　デスティネーションレジスタ。

<Rm>　　最初の乗算オペランドを含むレジスタ。

<Rs>　　2 番目の乗算オペランドを含むレジスタ。

<Rn>　　累算オペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    value = Rm * Rs                    /* Signed multiplication */
    if R == 1 then
        Rd = ((Rn<<32) + value + 0x80000000)[63:32]
    else
        Rd = ((Rn<<32) + value)[63:32]
```

用法

32 ビット小数を高速で乗算します。たとえば、Q31 の 2 つの入力値を乗算し、Q30 の結果を生成します（Q*n* は *n* ビットの小数の固定小数点数）。

小数演算の概要については、P. A2-69 *「飽和 Q15、Q31 算術演算」*を参照して下さい。

注

R15 の使用　　<Rd>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

――― 注 ―――

<Rn>レジスタにR15を使用すると、アセンブラによってエラーが生成されます。

――――――――

エンコード　　命令の <Rn> フィールドの値が 0b1111 である場合、この命令は SMMUL 命令（P. A4-163 *「SMMUL」*参照）です。

短縮処理　　乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

フラグ　　SMMLA 命令は、どのフラグも変更しません。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.83 SMMLS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	1	0	1	0	1	Rd	Rn	Rs	1	1	R	1	Rm

SMMLS（符号付き最上位ワード積減算）は、2 つの符号付き 32 ビット値を乗算し、演算結果から最上位の 32 ビットを抽出して、これを累算値から減算します。

オプションとして、演算結果の切り捨ての代わりに丸めを指定できます。この場合、上位のワードが抽出される前に、定数 0x80000000 が累算値に加算されます。

構文

```
SMMLS{R}{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

各項目の説明については以下を参照して下さい。

R　命令の R ビットを 1 にセットします。乗算は丸められます。

R を省略すると R ビットは 0 にセットされ、乗算は切り捨てられます。

<cond>　この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd>　デスティネーションレジスタ。

<Rm>　最初の乗算オペランドを含むレジスタ。

<Rs>　2 番目の乗算オペランドを含むレジスタ。

<Rn>　累算オペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    value = Rm * Rs                    /* Signed multiplication */
    if R == 1 then
        Rd = ((Rn<<32) - value + 0x80000000)[63:32]
    else
        Rd = ((Rn<<32) - value)[63:32]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

32 ビット小数を高速で乗算します。たとえば、Q31 の 2 つの入力値を乗算し、Q30 の結果を生成します（Q*n* は *n* ビットの小数の固定小数点数）。

注

R15 の使用 <Rd>、<Rm>、<Rs>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

フラグ SMMLS 命令は、どのフラグも変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.84 SMMUL

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11–8	7	6	5	4	3–0
cond	0	1	1	1	0	1	0	1	Rd	1	1	1	1	Rs	0	0	R	1	Rm

SMMUL（符号付き最上位ワード乗算）は、2 つの符号付き 32 ビット値を乗算し、演算結果から最上位の 32 ビットを抽出します。

オプションとして、演算結果の切り捨ての代わりに丸めを指定できます。この場合、上位のワードが抽出される前に、定数 0x80000000 が積に加算されます。

構文

```
SMMUL{R}{<cond>}  <Rd>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

R
: 命令の R ビットを 1 にセットします。乗算は丸められます。
: R を省略すると R ビットは 0 にセットされ、乗算は切り捨てられます。

<cond>
: この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd>
: デスティネーションレジスタ。

<Rm>
: 最初の乗算オペランドを含むレジスタ。

<Rs>
: 2 番目の乗算オペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    if R == 1 then
        value = Rm * Rs + 0x80000000  /* Signed multiplication */
    else
        value = Rm * Rs               /* Signed multiplication */
    Rd = value[63:32]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

SMMUL命令をQADD命令またはQDADD命令と併用して、Q31の乗算と積和演算を実行することができます。この方法は、SMULL 命令を QADD 命令または QDADD 命令と併用した場合に比べて、次の 2 つの利点があります。

- 積を丸めることができる。
- 積の下位半分用にスクラッチレジスタが必要ない。

SMMUL 命令は、最適化された高速フーリエ変換などのアルゴリズムで使用することもできます。

注

R15 の使用 <Rd>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

フラグ SMMUL 命令は、どのフラグも変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.85 SMUAD

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11–8	7	6	5	4	3–0
cond	0	1	1	1	0	0	0	0	Rd	1	1	1	1	Rs	0	0	X	1	Rm

SMUAD（符号付き乗加算デュアル）は、符号付き 16 × 16 ビット乗算を 2 つ実行します。2 つの積が加算され、32 ビットの演算結果が生成されます。

オプションとして、演算を実行する前に 2 番目のオペランドのハーフワードを交換できます。これにより、最上位×最下位および最下位×最上位の乗算が行われます。

この命令による加算の結果オーバフローが発生した場合、Q フラグがセットされます。乗算によりオーバフローが発生することはありません。

構文

```
SMUAD{X}{<cond>}  <Rd>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

X	命令のXビットを1にセットし、最下位×最上位と最上位×最下位の乗算が行われます。

X を省略すると、X ビットは 0 にセットされ、最下位×最下位と最上位×最上位の乗算が行われます。

<cond>	この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd>	デスティネーションレジスタ。

<Rm>	最初のオペランドを含むレジスタ。

<Rs>	2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
    Rd = product1 + product2
    if OverflowFrom(product1 + product2) then
        Q flag = 1
```

用法

SMUAD 命令は、シーケンス内にある最初の乗算のペアに対して使用します。シーケンス内の以降の乗算に対しては SMLAD 命令を使用する必要があります。詳細については、P. A4-145「*SMLAD*」を参照して下さい。

X オプションを使用すると、16 ビットの実数部分と 16 ビットの虚数部分がある複素数の積の虚数部分を計算することができます。

注

R15 の使用 <Rd>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

N、Z、C、V フラグ SMUAD 命令は、N、Z、C、V フラグを変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.86 SMUL<x><y>

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	1	0	1	1	0	Rd	SBZ	Rs	1	y	x	0	Rm

SMUL<x><y>（符号付き乗算 BB、BT、TB、TT）は符号付き乗算を実行します。ソースレジスタの下位半分または上位半分のいずれかから取得した 2 つの符号付き 16 ビット値が乗算されます。ソースレジスタの残りの半分は無視されます。この命令でオーバフローは発生しません。

構文

```
SMUL<x><y>{<cond>}  <Rd>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

<x> 乗算の最初のオペランドとして、ソースレジスタ <Rm> の上位と下位のどちらを使用するかを指定します。<x> が B の場合、命令エンコードに x == 0 が指定され、<Rm> の下位半分（ビット [15:0]）が使用されます。<x> が T の場合、命令エンコードに x == 1 が指定され、<Rm> の上位半分（ビット [31:16]）が使用されます。

<y> 乗算の 2 番目のオペランドとして、ソースレジスタ <Rs> の上位と下位のどちを使用するかを指定します。<y> が B の場合、命令エンコードに y == 0 が指定され、<Rs> の下位半分（ビット [15:0]）が使用されます。<y> が T の場合、命令エンコードに y == 1 が指定され、<Rs> の上位半分（ビット [31:16]）が使用されます。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> 最初のソースレジスタ。このレジスタの下位半分または上位半分（<x> で選択）が乗算の最初のオペランドになります。

<Rs> 2 番目のソースレジスタ。このレジスタの下位半分または上位半分（<y> で選択）が乗算の 2 番目のオペランドになります。

アーキテクチャのバージョン

ARMv5TE 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then

    if (x == 0) then
        operand1 = SignExtend(Rm[15:0])
    else /* x == 1 */
        operand1 = SignExtend(Rm[31:16])

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    Rd = operand1 * operand2
```

用法

この命令は、本来の用途である整数乗算の他に、QADD 命令、QDADD 命令、QDSUB 命令と併用して、Q15 数の乗算、積和演算、積減算を実行することもできます。例については、P. A4-94、P. A4-101、P. A4-103 の*用法*を参照して下さい。

注

R15 の使用 <Rd>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

条件フラグ SMUL<x><y> 命令は、N、Z、C、V、Q フラグを変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.87 SMULL

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	0	1	1	0	S	RdHi	RdLo	Rs	1	0	0	1	Rm

SMULL（符号付き乗算 long 型）は、2 つの符号付き 32 ビット値を乗算して、結果の 64 ビット値を生成します。

SMULL 命令では、必要に応じて、64 ビットの結果に基づいて条件コードフラグを更新することもできます。

構文

```
SMULL{<cond>}{S}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S 命令の S ビット（ビット [20]）に 1 をセットし、乗算の結果に従って CPSR の N フラグと Z フラグをセットすることを指定します。S を省略すると、命令の S ビットは 0 にセットされ、CPSR 全体が命令の影響を受けません。

<RdLo> 結果の下位 32 ビットがストアされるレジスタ。

<RdHi> 結果の上位 32 ビットがストアされるレジスタ。

<Rm> <Rs> の値に乗算される符号付き値を保持します。

<Rs> <Rm> の値に乗算される符号付き値を保持します。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    RdHi = (Rm * Rs)[63:32] /* Signed multiplication */
    RdLo = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

用法

SMULL 命令は、符号付き変数を乗算して、64 ビットの結果を 2 つの汎用レジスタに書き込みます。

注

R15 の使用 <RdHi>、<RdLo>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

オペランドの制限 <RdHi> と <RdLo> が同じレジスタの場合、結果は予測不能です。

以前の説明には、<RdHi> と <Rm> または <RdLo> と <Rm> に同じレジスタを指定した場合は、予測不能な結果を招くことが記載されていました。高性能の乗算器では結果をライトバックする前にすべてのオペランドを読み込むため、ARMv6 では制限は全くなく、該当する ARM の v4 および v5 のどの実装でも、この制限は必要ありません。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

C フラグと V フラグ ARMv5 以降では、SMULLS は C フラグと V フラグが変更されないように定義されています。以前のバージョンのアーキテクチャでは、SMULLS 命令の実行後の C フラグと V フラグの値は予測不能です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.88 SMULW<y>

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	1	0	0	1	0	Rd	SBZ	Rs	1	y	1	0	Rm

SMULW<y>（符号付きワード乗算 B、T）は、符号付き乗算を実行します。符号付き 32 ビット値と符号付き 16 ビット値が乗算されます。16 ビット値はソースレジスタの下位半分または上位半分のいずれかから取得されます。ソースレジスタの残りの半分は無視されます。48 ビット積の上位の 32 ビットが、デスティネーションレジスタに書き込まれます。48 ビットの積の下位 16 ビットは無視されます。

この命令でオーバフローは発生しません。

構文

```
SMULW<y>{<cond>}  <Rd>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

<y> 乗算の 2 番目のオペランドとして、ソースレジスタ <Rs> の上位と下位のどちらを使用するかを指定します。<y> が B の場合、命令エンコードに y == 0 が指定され、<Rs> の下位半分（ビット [15:0]）が使用されます。<y> が T の場合、命令エンコードに y == 1 が指定され、<Rs> の上位半分（ビット [31:16]）が使用されます。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> 最初のオペランドである 32 ビット値が格納されているソースレジスタ。

<Rs> 2 番目のソースレジスタ。このレジスタの下位半分または上位半分（<y> で選択）が 2 番目のオペランドになります。

アーキテクチャのバージョン

ARMv5TE 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    Rd = (Rm * operand2)[47:16] /* Signed multiplication */
```

用法

この命令は、本来の用途である整数乗算の他に、`QADD` 命令、`QDADD` 命令、`QDSUB` 命令と併用して、Q31 数と Q15 数の間での乗算、積和演算、積減算を実行することもできます。例については、P. A4-94、P. A4-101、P. A4-103 の*用法*を参照して下さい。

注

R15 の使用 `<Rd>`、`<Rm>`、`<Rs>` のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、`<Rs>` オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

フラグ `SMULW<y>` 命令は、どのフラグも変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.89 SMUSD

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11–8	7	6	5	4	3–0
cond	0	1	1	1	0	0	0	0	Rd	1	1	1	1	Rs	0	1	X	1	Rm

SMUSD（符号付き積減算デュアル）は、符号付き 16 × 16 ビット乗算を 2 つ実行します。一方の積がもう一方の積から減算され、32 ビットの演算結果が生成されます。

オプションとして、演算を実行する前に 2 番目のオペランドのハーフワードを交換できます。これにより、最上位×最下位および最下位×最上位の乗算が行われます。

オーバフローは発生しません。

構文

```
SMUSD{X}{<cond>}  <Rd>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

X
命令のXビットを1にセットし､最下位×最上位と最上位×最下位の乗算が行われます。

X を省略すると、X ビットは 0 にセットされ、最下位×最下位と最上位×最上位の乗算が行われます。

<cond>
この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd>
デスティネーションレジスタ。

<Rm>
最初の乗算オペランドを含むレジスタ。

<Rs>
2 番目の乗算オペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    product1 = Rm[15:0]  * operand2[15:0]      /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]     /* Signed multiplication */
    Rd = product1 - product2                   /* Signed subtraction */
```

用法

SMUSD命令を使用すると、16 ビットの実数部分と 16 ビットの虚数部分がある複素数の積の実数部分を計算することができます。

注

R15 の使用 <Rd>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

フラグ SMUSD 命令は、どのフラグも変更しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.90 SRS

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 12	11 10 9 8	7 5	4 0
1 1 1 1	1 0 0	P	U	1	W	0	1 1 0 1	SBZ	0 1 0 1	SBZ	mode

SRS（復帰状態ストア）は、現在のモードの R14 と SPSR を、指定したアドレスのワードとその次のワードにそれぞれ保存します。アドレスは、指定したモードに属する R13 のバンクバージョンに基づいて決定されます。

構文

```
SRS<addressing_mode> #<mode>{!}
```

各項目の説明については以下を参照して下さい。

`<addressing_mode>`

LDMおよびSTM命令の <addressing_mode>と同様です。詳細については、P. A5-41「*アドレシングモード4 - 複数ロード/ストア*」を参照して下さい。差異は以下のとおりです。

- ベースレジスタ（Rn）は、現在のモードではなく <mode> で指定したモードの R13 のバンクバージョンです。
- ストアするレジスタの数は 2 つです。
- レジスタリストは {R14, SPSR} です。R14 と SPSR は共に現在のモードのバージョンです。

`<mode>` バンクレジスタが<addressing_mode>の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。モード番号は PSR で選択されたモードを 5 ビットにエンコードしたものです。詳細については、P. A2-14「*モードビット*」を参照して下さい。

`!` 存在する場合、W ビットをセットします。それによって、命令は修飾済みの値を P. A5-41「*アドレシングモード 4 - 複数ロード / ストア*」で指定されているものと同様の方法でベースレジスタに書き戻しします。! が省略された場合、W ビットは 0 になり、ベースレジスタが命令によって変更されることはありません。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
address = start_address
Memory[address,4] = R14
if CurrentModeHasSPSR() then
    Memory[address+4,4] = SPSR
else
    UNPREDICTABLE
assert end_address == address + 8
```

start_address と end_address は、P. A5-41「*アドレシングモード 4 - 複数ロード / ストア*」で説明されている方法で決定されます。このとき、次のような修正が行われます。

- Number_Of_Set_Bits_in(register_list) は命令のビット [15:0] にかかわらず 2 に設定されます。
- Rn は現在のモードでの R13 のバージョンではなく、命令により指定されたモードでの R13 のバンクバージョンに設定されます。

注

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21「*データアボート（データアクセス中に発生するメモリアボート）*」を参照して下さい。

非ワード境界アライメントアドレス

SRS では、address の下位 2 ビットが無視されます。

アライメント　システム制御コプロセッサが実装に含まれていて（第 B3 章「*システム制御コプロセッサ*」参照）、アライメントチェックが許可されている場合、ビット [1:0] の値が 0b00 でないアドレスではアライメント例外が発生します。

時間順序　SRS命令で生成されたメモリの各ワードにアクセスする時間順序は、アーキテクチャ上では定義されていません。この命令は、アクセス順序が関係するメモリマップされた I/O ロケーションには使用しないで下さい。

ユーザモードとシステムモード

ユーザモードとシステムモードにはSPSRがないため、SRS命令の結果は予測不能です。

——— 注 ———

ユーザモードでは、SRS で他のモードのバンクレジスタへアクセスしないで下さい。セキュリティホールが発生するおそれがあります。

———————

条件　他の大部分の ARM 命令とは異なり、SRS は条件付きで実行できません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.91 SSAT

31–28	27	26	25	24	23	22	21	20–16	15–12	11–7	6	5	4	3–0
cond	0	1	1	0	1	0	1	sat_imm	Rd	shift_imm	sh	0	1	Rm

SSAT（符号付き飽和）は、符号付き値を符号付きの範囲に飽和させます。飽和するビット位置は選択できます。

飽和の前に値にシフトを適用できます。

演算が飽和すると、Q フラグがセットされます。

構文

```
SSAT{<cond>}  <Rd>, #<immed>, <Rm>{, <shift>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<immed> 飽和させるビット位置を 1 ～ 32 の間で指定します。この値は命令の sat_imm フィールドで、<immed> - 1 としてエンコードされます。

<Rm> 飽和する符号付き値を含むレジスタ。

<shift> オプションのシフトを指定します。指定する場合は、以下のいずれかを選択します。

- LSL #*N*. *N* は 0 ～ 31 の範囲で指定します。
  これは、sh == 0 および shift_imm == *N* としてエンコードされます。
- ASR #*N*. *N* は 1 ～ 32 の範囲で指定します。これは、sh == 1 で、*N* == 32 の場合は shift_imm == 0、それ以外の場合は shift_imm == *N* にエンコードされます。

<shift> が省略されている場合、LSL #0 が使用されます。

戻り値

Rd に戻される値は次のとおりです。

$-2^{(n-1)}$ $X < -2^{(n-1)}$ の場合

X $-2^{(n-1)} <= X <= 2^{(n-1)} - 1$ の場合

$2^{(n-1)} - 1$ $X > 2^{(n-1)} - 1$ の場合

n は <immed>、X は Rm からシフトされた値です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    if shift == 1 then
        if shift_imm == 0 then
            operand = (Rm Artihmetic_Shift_Right 32)[31:0]
        else
            operand = (Rm Artihmetic_Shift_Right shift_imm)[31:0]
    else
        operand = (Rm Logical_Shift_Left shift_imm)[31:0]
    Rd = SignedSat(operand, sat_imm + 1)
    if SignedDoesSat(operand, sat_imm + 1) then
        Q Flag = 1
```

用法

SSAT命令は、符号付きデータのスケーリングと飽和が必要な各種のDSPアルゴリズムで使用できます。

注

R15 の使用 レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.92 SSAT16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	1	0	1	0	sat_imm	Rd	SBO	0	0	1	1	Rm

SSAT16 は、2 つの符号付き 16 ビット値を符号付きの範囲に飽和させます。飽和するビット位置は選択できます。いずれかのハーフワード演算が飽和すると、Q フラグがセットされます。

構文

```
SSAT16{<cond>}  <Rd>, #<immed>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<immed> 飽和のビット位置を指定します。このビット位置は 1 ～ 16 の間で指定します。この値は命令の sat_imm フィールドで、<immed> - 1 としてエンコードされます。

<Rm> 飽和する符号付き値を含むレジスタ。

戻り値

Rd の上位半分と下位半分に戻される値は、それぞれ以下のとおりです。

$-2^{(n-1)}$ $X < -2^{(n-1)}$ の場合

X $-2^{(n-1)} <= X <= 2^{(n-1)} - 1$ の場合

$2^{(n-1)} - 1$ $X > 2^{(n-1)} - 1$ の場合

n は <immed>、X は Rm のいずれかの半分の値です。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    Rd[15:0]  = SignedSat(Rm[15:0],  sat_imm + 1)
    Rd[31:16] = SignedSat(Rm[31:16], sat_imm + 1)
    if SignedDoesSat(Rm[15:0],  sat_imm + 1)
                                  OR SignedDoesSat(Rm[31:16], sat_imm + 1) then
        Q Flag = 1
```

用法

`SSAT16` 命令は、符号付きデータの飽和が必要な各種の DSP アルゴリズムで使用できます。

注

R15 の使用　　レジスタ `<Rd>` または `<Rm>` として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.93 SSUB16

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	0	0	1	Rn	Rd	SBO	0	1	1	1	Rm

SSUB16（符号付き減算）は、16 ビット符号付き整数の減算を 2 つ実行します。減算の結果に従って、CPSR の GE ビットをセットします。

構文

```
SSUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    diff      = Rn[15:0] - Rm[15:0]    /* Signed subtraction */
    Rd[15:0]  = diff[15:0]
    GE[1:0]   = if diff >= 0 then 0b11 else 0
    diff      = Rn[31:16] - Rm[31:16]  /* Signed subtraction */
    Rd[31:16] = diff[15:0]
    GE[3:2]   = if diff >= 0 then 0b11 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

SSUB16 命令を使用して、ハーフワードデータの配列の演算を高速化できます。これは、SADD16 の使用方法と同様です。詳細については、P. A4-120「*SADD16*」の「用法」を参照して下さい。

SSUB16命令を使用すると、16ビット整数やQ15数のペアとして格納されている複素数の演算を行うこともできます。レジスタの最下位と最上位にそれぞれ複素数の実数部と虚数部がある場合、次の命令を使用します。

```
SSUB16  Rd, Ra, Rb
```

この命令は、Rd = Ra - Rb という複素算術演算を行います。

SSUB16 命令を実行すると、各減算の結果に従って GE フラグが設定されます。これらは、次の SEL 命令で使用できます。詳細については、P. A4-128「*SEL*」を参照して下さい。

注

R15 の使用　　<Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.94 SSUB8

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 0 0 1	Rn	Rd	SBO	1 1 1 1	Rm

SSUB8 は、8 ビット符号付き整数の減算を 4 つ実行します。減算の結果に従って、CPSR の GE ビットをセットします。

構文

```
SSUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    diff      = Rn[7:0] - Rm[7:0]       /* Signed subtraction */
    Rd[7:0]   = diff[7:0]
    GE[0]     = if diff >= 0 then 1 else 0
    diff      = Rn[15:8] - Rm[15:8]     /* Signed subtraction */
    Rd[15:8]  = diff[7:0]
    GE[1]     = if diff >= 0 then 1 else 0
    diff      = Rn[23:16] - Rm[23:16]   /* Signed subtraction */
    Rd[23:16] = diff[7:0]
    GE[2]     = if diff >= 0 then 1 else 0
    diff      = Rn[31:24] - Rm[31:24]   /* Signed subtraction */
    Rd[31:24] = diff[7:0]
    GE[3]     = if diff >= 0 then 1 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

SSUB8 命令を使用して、バイトデータの配列の演算を高速化できます。これは、ハーフワードデータの演算を加速する SADD16 の使用方法と同様です。詳細については、P. A4-120「*SADD16*」の「用法」を参照して下さい。

注

R15 の使用　<Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.95 SSUBADDX

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	0	0	1	Rn	Rd	SBO	0	1	0	1	Rm

SSUBADDX（交換符号付き減算 / 加算）は、16 ビット符号付き整数減算と 16 ビット符号付き整数加算を 1 つづつ実行します。演算を実行する前に、2 番目のオペランドにある 2 つのハーフワードの交換を行います。

SSUBADDX 命令を実行すると、演算の結果に従って CPSR の GE ビットが設定されます。

構文

```
SSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    diff       = Rn[31:16] - Rm[15:0]     /* Signed subtraction */
    Rd[31:16] = diff[15:0]
    GE[3:2]    = if diff >= 0 then 0b11 else 0
    sum        = Rn[15:0] + Rm[31:16]     /* Signed addition */
    Rd[15:0]   = sum[15:0]
    GE[1:0]    = if sum >= 0 then 0b11 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

SSUBADDX 命令を使用して、16 ビット整数や Q15 数のペアとして格納されている複素数の演算を行うことができます。レジスタの最下位と最上位にそれぞれ複素数の実数部と虚数部がある場合、次の命令を使用します。

```
SSUBADDX Rd, Ra, Rb
```

この命令は、Rd = Ra - i * Rb という複素算術演算を行います。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.96 STC

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7–0
cond	1	1	0	P	U	N	W	0	Rn	CRd	cp_num	8_bit_word_offset

STC（コプロセッサストア）は、コプロセッサのデータを連続したメモリアドレスのシーケンスにストアします。命令を実行できるコプロセッサがない場合は、未定義命令例外が生成されます。

構文

```
STC{<cond>}{L}  <coproc>, <CRd>, <addressing_mode>
STC2{L}         <coproc>, <CRd>, <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

STC2 命令の条件フィールドを 0b1111 にセットします。これによって、コプロセッサ設計者用のオペコード空間が追加されます。生成された命令は無条件でのみ実行できます。

L 命令内の N ビット（ビット [22]）を 1 に設定し、ロングストア（単精度データ転送の代わりに倍精度データ転送を使用するなど）を指定します。L が省略された場合、N ビットは 0 になり、命令はショートストアを指定します。

<coproc> コプロセッサの名前を指定すると、対応するコプロセッサ番号が命令の cp_num フィールドに配置されます。標準の汎用コプロセッサ名は p0、p1、...、p15 です。

<CRd> ソースのコプロセッサレジスタを指定します。

<addressing_mode>

P. A5-49 *「アドレシングモード 5 - コプロセッサのロード / ストア」* を参照して下さい。これによって命令の P、U、Rn、W、8_bit_word_offset の各ビットが決定されます。

<addressing_mode>の全形式の構文にはベースレジスタ<Rn>が含まれます。命令によるベースレジスタ値の変更を指定する形式もあります（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

STC はすべてのバージョンに存在します。

STC2 は ARMv5 以降に存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

未定義命令、データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    Memory[address,4] = value from Coprocessor[cp_num]
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
        Memory[address,4] = value from Coprocessor[cp_num]
        if (Shared(address))    /* from ARMv6 */
            physical_address = TLB(address)
            ClearExclusiveByAddress(physical_address,processor_id,4)
            /* See P. A2-49「操作の概要」*/
    assert address == end_address
```

用法

STC はコプロセッサのデータをメモリにストアする場合に有効です。L（ロング）オプションは N ビットを制御します。また、浮動小数点ストア命令で単精度転送と倍精度転送を区別する場合にも使用できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

コプロセッサフィールド

ARM アーキテクチャで定義されているのは、命令のビット [31:23]、ビット [21:16]、ビット [11:0] のみです。他のフィールド(ビット [22] とビット [15:12])は、個々の ARM 開発システムに依存しています。

インデクスなしアドレシングモード(P == 0、U == 1、W == 0)の場合、命令ビット [7:0] も ARM アーキテクチャでは定義されていません。このビットは他のコプロセッサオプションの指定に使用できます。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

非ワード境界アラインメントアドレス

CP15_reg1_Ubitの値が0の場合、コプロセッサストアレジスタ命令は`address`の最下位 2 ビットを無視します。CP15_reg1_Ubit の値が 1 の場合、すべての非ワード境界アラインメントアクセスがアライメントフォルトを引き起こします。

アライメント

実装にシステム制御コプロセッサが含まれており(第 B3 章*「システム制御コプロセッサ」*を参照して下さい)、アライメントチェックが有効な場合、ビット [1:0] != 0b00 のアドレスはアライメント例外を発生します。

未実装のコプロセッサ命令

ハードウェアコプロセッサのサポートは、アーキテクチャのバージョンに関わらずオプションです。実装では、コプロセッサ命令のサブセットの実装、またはコプロセッサ命令の未実装を選択できます。ただし、未実装のコプロセッサ命令は未定義命令例外の原因となります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.97 STM (1)

31–28	27	26	25	24	23	22	21	20	19–16	15–0
cond	1	0	0	P	U	0	W	0	Rn	register_list

STM (1)（複数ストア）は、汎用レジスタ内の空ではないサブセット（場合によっては全体）をシーケンシャルなメモリ位置にストアします。

構文

```
STM{<cond>}<addressing_mode>  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

`<cond>` この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。`<cond>` が省略されている場合、AL（常時）条件が使用されます。

`<addressing_mode>`

P. A5-41 *「アドレシングモード 4 - 複数ロード / ストア」* を参照して下さい。これによって命令の P、U、W ビットが決定されます。

`<Rn>` `<addressing_mode>` で使用するベースレジスタを指定します。`<Rn>` として R15 を指定した場合、結果は予測不能です。

`!` W ビットをセットし、命令は修飾済みの値をベースレジスタ Rn にライトバックします。詳細については P. A5-41 *「アドレシングモード 4 - 複数ロード / ストア」* を参照して下さい。! が省略された場合は、W ビットは 0 になり、命令はこの方法でベースレジスタを変更することはありません。

`<registers>` ロードされるレジスタの一覧で、カンマで区切って並べ、{ と } で囲みます。レジスタのリストは STM 命令によりストアされるレジスタの組を指定します。

レジスタは最も低い番号のレジスタから最下位のメモリアドレス（開始アドレス）に、その後順に最も高い番号のレジスタから最上位のメモリアドレス（終了アドレス）への順序でストアされます。

R0 ～ R15 のそれぞれのレジスタが一覧にある場合はそれぞれに対応する命令の register_list フィールドのビット [i] が 1 になり、ない場合は 0 になります。ビット [15:0] がすべてゼロの場合、予測不能な結果を招きます。

`<registers>` として R15 を指定した場合、ストアされる値は実装定義です。詳細については、P. A2-9 *「プログラムカウンタの読み出し」* を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to 15
        if register_list[i] == 1 then
            Memory[address,4] = Ri
            address = address + 4
            if (Shared(address)) then   /* from ARMv6 */
                physical_address = TLB(address)
                ClearExclusiveByAddress(physical_address,processor_id,4)
                /* See P. A2-49「操作の概要」*/
    assert end_address == address - 4
```

用法

STM 命令は、ブロックストア命令（LDM 命令と併用するとブロックコピーとして効果的です）およびスタック操作として有効です。プロシージャのシーケンスで STM 命令を単独で使用すると、復帰アドレスと汎用レジスタの値をスタックにプッシュして、プロセス内のスタックポインタを更新できます。

注

オペランドの制限

<Rn> が <registers> で指定され、ベースレジスタのライトバックが指定されている場合、次の動作が行われます。

- <Rn> が、< レジスタ > に指定されている中で最も番号の小さいレジスタの場合、<Rn> の元の値がストアされます。
- それ以外の場合、ストアされる <Rn> の値は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21*「データアボートが発生した命令の影響」*を参照して下さい。

非ワード境界アラインメントアドレス

CP15_reg1_Ubit の値が 0 の場合、STM[1] 命令は address の最下位 2 ビットを無視します。CP15_reg1_Ubit の値が 1 の場合、すべての非ワード境界アラインメントアクセスがアライメントフォルトを引き起こします。

アライメント　システム制御コプロセッサが実装に含まれていて（第 B3 章*「システム制御コプロセッサ」*参照）、アライメントチェックが許可されている場合、ビット [1:0] の値が 0b00 でないアドレスではアライメント例外が発生します。

時間順序　この命令によって生成される、メモリの個別のワードに対するアクセスの時間順序は、一部の条件でのみ定義されます。詳細については、P. B2-13*「メモリアクセスの制限」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.98 STM (2)

31–28	27	26	25	24	23	22	21	20	19–16	15–0
cond	1	0	0	P	U	1	0	0	Rn	register_list

STM (2) は、ユーザモードの汎用レジスタのサブセット（場合によっては全体）をシーケンシャルなメモリ位置にストアします。

構文

```
STM{<cond>}<addressing_mode>  <Rn>, <registers>^
```

各項目の説明については以下を参照して下さい。

<cond>
: この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<addressing_mode>
: P. A5-41 *「アドレシングモード 4 - 複数ロード / ストア」*を参照して下さい。これによって命令の P ビットと U ビットが決定されます。このアドレシングモードは W == 0 の場合に限り、この形の STM 命令で使用できます。

<Rn>
: <addressing_mode> で使用するベースレジスタを指定します。<Rn> として R15 を指定した場合、結果は予測不能です。

<registers>
: ロードされるレジスタの一覧で、カンマで区切って並べ、{ と } で囲みます。レジスタのリストは STM 命令によりストアされるレジスタの組を指定します。

  レジスタは最も低い番号のレジスタから最下位のメモリアドレス（開始アドレス）に、その後順に最も高い番号のレジスタから最上位のメモリアドレス（終了アドレス）への順序でストアされます。

  R0 ～ R15 のそれぞれのレジスタが一覧にある場合はそれぞれに対応する命令の register_list フィールドのビット [i] が 1 になり、ない場合は 0 になります。ビット [15:0] がすべてゼロの場合、予測不能な結果を招きます。

  <registers> として R15 を指定した場合、ストアされる値は実装定義です。詳細については、P. A2-9 *「プログラムカウンタの読み出し」*を参照して下さい。

^
: STM 命令の場合、ユーザモードのレジスタがストアされることを示します。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to 15
        if register_list[i] == 1
            Memory[address,4] = Ri_usr
            address = address + 4
            if (Shared(address))    /* from ARMv6 */
                physical_address = TLB(address)
                ClearExclusiveByAddress(physical_address,processor_id,4)
                /* See P. A2-49「操作の概要」*/
    assert end_address == address - 4
```

用法

STM (2) 命令を使用して、プロセッサが特権モードの場合にユーザモードのレジスタをストアすることができます（プロセススワップの実行時、および命令エミュレータで有効です）。

注

ライトバック ビット [21]（W ビット）をセットした場合、結果は予測不能です。

ユーザモードとシステムモード
この命令はユーザモードまたはシステムモードでは予測不能な結果を招きます。

ベースレジスタモード
アドレス計算に使用されるベースレジスタは、ユーザモードレジスタではなく現在のプロセッサモードのレジスタから読み込まれます。

データアボート データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

非ワード境界アラインメントアドレス
CP15_reg1_Ubit の値が 0 の場合、STM[2] 命令は address の最下位 2 ビットを無視します。CP15_reg1_Ubit の値が 1 の場合、すべての非ワード境界アラインメントアクセスがアライメントフォルトを引き起こします。

アライメント システム制御コプロセッサが実装に含まれていて（第 B3 章 *「システム制御コプロセッサ」*参照）、アライメントチェックが許可されている場合、ビット [1:0] の値が 0b00 でないアドレスではアライメント例外が発生します。

時間順序 この命令によって生成される、メモリの個別のワードに対するアクセスの時間順序は、一部の条件でのみ定義されます。詳細については、P. B2-13 *「メモリアクセスの制限」*を参照して下さい。

バンクレジスタ ARMv6 以前のバージョンの ARM アーキテクチャでは、この形式の STM 命令の次に、バンクレジスタにアクセスする命令を置くことはできません（間に NOP を置くと、これを回避できます）。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.99 STR

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	1	I	P	U	0	W	0	Rn	Rd	addr_mode

STR（レジスタストア）は、レジスタのワードをメモリにストアします。

構文

```
STR{<cond>}  <Rd>, <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> 演算のソースレジスタ。<Rd> として R15 を指定した場合、ストアされる値は実装定義です。詳細については、P. A2-9 *「プログラムカウンタの読み出し」*を参照して下さい。

<addressing_mode>

P. A5-18 *「アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア」*を参照して下さい。これによって命令の I、P、U、W、Rn、addr_mode の各ビットが決定されます。

<addressing_mode>の全形式の構文にはベースレジスタ<Rn>が含まれます。命令によるベースレジスタ値の変更を指定する形式もあります（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    Memory[address,4] = Rd
    if (Shared(address))    /* from ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,4)
        /* See P. A2-49「操作の概要」*/
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

適切なアドレシングモードで STR 命令を使用すると、汎用レジスタの 32 ビットデータがメモリにストアされます。PC をベースレジスタとして使用することで、PC 相対アドレシングが可能になり、ポジション・インディペンデントコードの使用が容易になります。

注

オペランドの制限

`<addressing_mode>` でベースレジスタライトバックを指定し、`<Rd>` と `<Rn>` に同じレジスタを指定した場合、結果は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21「*データアボートが発生した命令の影響*」を参照して下さい。

アライメント ARMv6 以前のバージョンでは、SRS 命令はアドレスの最下位の 2 ビットを無視します。この動作は LDR 命令の場合とは異なります。アライメントチェック（アドレス [1:0] != 0b00 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定の混在エンディアン形式が、アライメントチェックオプションと共にサポートされています。ARMv6 の疑似コードでは、CPSR の E ビットにより定義されたエンディアン方式の転送で、アンアラインドな混在エンディアンのサポートが構成されていることを前提としています。

エンディアン形式およびアライメントの詳細については、P. A2-30「*エンディアンのサポート*」と P. A2-38「*アンアラインドアクセスのサポート*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.100 STRB

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	1	I	P	U	1	W	0	Rn	Rd	addr_mode

STRB（レジスタストアバイト）は、レジスタの最下位のバイトをメモリにストアします。

構文

```
STR{<cond>}B  <Rd>, <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> 演算のソースレジスタ。<Rd> として R15 を指定した場合、結果は予測不能です。

<addressing_mode>

P. A5-18 *「アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア」*を参照して下さい。これによって命令の I、P、U、W、Rn、addr_mode の各ビットが決定されます。

<addressing_mode>の全形式の構文にはベースレジスタ<Rn>が含まれます。命令によるベースレジスタ値の変更を指定する形式もあります（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

動作

```
if ConditionPassed(cond) then
    Memory[address,1] = Rd[7:0]
    if (Shared(address))      /* from ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,1)
        /* See P. A2-49「操作の概要」*/
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

適切なアドレシングモードで STRB 命令を使用すると、汎用レジスタの最下位のバイトがメモリに書き込まれます。PC をベースレジスタとして使用することで、PC 相対アドレシングが可能になり、ポジション・インディペンデントコードの使用が容易になります。

注

オペランドの制限

<addressing_mode> でベースレジスタライトバックを指定し、<Rd> と <Rn> に同じレジスタを指定した場合、結果は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21*「データアボートが発生した命令の影響」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.101 STRBT

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	1	I	0	U	1	1	0	Rn	Rd	addr_mode

STRBT（変換レジスタストアバイト）は、レジスタの最下位のバイトをメモリにストアします。プロセッサが特権モードのときにこの命令を実行すると、プロセッサがユーザモードにある場合と同様にアクセスを処理することを要求するシグナルがメモリシステムに送られます。

構文

```
STR{<cond>}BT  <Rd>, <post_indexed_addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> 演算のソースレジスタ。<Rd> として R15 を指定した場合、結果は予測不能です。

<post_indexed_addressing_mode>

P. A5-18 *「アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア」* を参照して下さい。これによって命令の I、U、Rn、addr_mode の各ビットが決定されます。この命令では、アドレシングモード 2 のポストインデクス形式のみが使用できます。これらの形式では P == 0、W == 0 となり、P および W はそれぞれビット [24] とビット [21] です。この命令では代わりに P == 0、W == 1 を使用しますが、その他のすべての点についてはアドレシングモードは同一です。

<post_indexed_addressing_mode> の全形式の構文には、*ベースレジスタ* <Rn> が含まれます。すべての形式で、命令によるベースレジスタ値の変更が指定されます（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    Memory[address,1] = Rd[7:0]
    if (Shared(address))      /* from ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,1)
        /* See P. A2-49「操作の概要」*/
```

用法

STRBT は、通常はユーザモードで実行されるメモリアクセス命令をエミュレートしている（特権）例外ハンドラで使用できます。ユーザモード権限の場合と同様に、アクセスは制限されます。

注

ユーザモード　この命令をユーザモードで実行すると、通常のユーザモードアクセスが実行されます。

オペランドの制限

<Rd> と <Rn> に同じレジスタを指定した場合、結果は予測不能です。

データアボート データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.102 STRD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	P	U	I	W	0	Rn	Rd	addr_mode	1	1	1	1	addr_mode

STRD（レジスタストアダブルワード）は、ARM レジスタのペアをメモリ内の連続した 2 ワードにストアします。このレジスタペアは、偶数番号のレジスタと、その次の奇数番号のレジスタに制限されます（R10 と R11 など）。

2 レジスタの STM 命令と比較して、より多くのアドレシングモードを使用できます。

構文

```
STR{<cond>}D  <Rd>, <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> <addressing_mode> で指定されるメモリワードにストアされる、偶数番号のレジスタ。直後にある奇数番号のレジスタは、次のメモリワードに保存されます。<Rd> が R14 の場合、2 番目のソースレジスタに R15 が指定されることになるため、命令の動作は予測不能です。

<Rd> が奇数番号のレジスタを指定している場合、この命令は未定義となります。

<addressing_mode>

P. A5-33 *「アドレシングモード 3 - その他のロードとストア」*を参照して下さい。これによって命令の P、U、I、W、Rn、addr_mode の各ビットが決定されます。

<addressing_mode>の全形式の構文にはベース*レジスタ*<Rn>が含まれます。命令によるベースレジスタ値の変更を指定する形式もあります（*ベースレジスタライトバック*と呼ばれます）。

<addressing_mode> で生成されたアドレスは、STRD 命令でストアされる 2 ワードのうち下位ワードのアドレスです。上位ワードのアドレスは、このアドレス + 4 で生成されます。

アーキテクチャのバージョン

ARMv5TE 以降に存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0)
        if (Rd is even-numbered and not R14)
            if (address[2:0] == 0b000) then
                Memory[address,4] = Rd
                Memory[address+4,4] = R(d+1)
            else
                UNPREDICTABLE
        else
            UNDEFINED
    else  /* CP15_reg1_Ubit == 1 */
        if (Rd is even-numbered and not R14)
            if (address[1:0] == 0b00) then
                Memory[address,4] = Rd
                Memory[address+4,4] = R(d+1)
            else
                AlignmentFault()
        else
            UNDEFINED
    if (Shared(address))    /* ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,4)
    if (Shared(address+4))    /* ARMv6 */
        physical_address = TLB(address+4)
        ClearExclusiveByAddress(physical_address,processor_id,4)
            /* See P. A2-49「操作の概要」*/
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

オペランドの制限

<addressing_mode> でベースレジスタのライトバックを実行し、ベースレジスタの <Rn> が命令の 2 つのソースレジスタの一方である場合、結果は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21*「データアボートが発生した命令の影響」*を参照して下さい。

アライメント ARMv6 以前は、メモリアドレスが 64 ビット境界にアラインしていない場合、命令の動作は予測不能です。アライメントチェック（データアボートの発生）およびビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定の混在エンディアン形式が、モジュロ 4 およびモジュロ 8 のアライメントチェックオプションと共にサポートされています。ARMv6 の疑似コードでは、CPSR の E ビットにより定義されたエンディアン方式の転送で、アンアラインドな混在エンディアンのサポートが構成されていることを前提としています。

エンディアン形式とアライメントの詳細については、P. A2-30*「エンディアンのサポート」*と P. A2-38*「アンアラインドアクセスのサポート」*を参照して下さい。

時間順序 2 つのメモリワードにアクセスする時間順序は、アーキテクチャで定義されていません。実装では 2 つの 32 ビットメモリにどの順序でもアクセスできます。また、2 つのワードを組み合わせて 1 つの 64 ビットメモリアクセスとすることもできます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.103 STREX

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 1 0 0	0	Rn	Rd	SBO	1 0 0 1	Rm

STREX（排他的レジスタストア）は、メモリへの条件付きストアを実行します。このストアは、既存のプロセッサがアドレス指定されたメモリに排他的アクセス権を持っている場合に限り実行されます。

構文

```
STREX{<cond>} <Rd>, <Rm>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> 返されるステータス値のデスティネーションレジスタ。返される値は次の通りです。

- 0 操作によりメモリが更新された場合
- 1 操作によるメモリ更新が失敗した場合

<Rm> メモリにストアされるワードを含むレジスタ。

<Rn> アドレスを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    physical_address = TLB(Rn)
    if IsExclusiveLocal(physical_address, processor_id, 4) then
        if Shared(Rn) == 1 then
            if IsExclusiveGlobal(physical_address, processor_id, 4) then
                Memory[Rn,4] = Rm
                Rd = 0
                ClearExclusiveByAddress(physical_address,processor_id,4)
            else
                Rd = 1
        else
            Memory[Rn,4] = Rm
            Rd = 0
    else
        Rd = 1
    ClearExclusiveLocal(processor_id)
    /* See P. A2-49「操作の概要」*/
    /* The notes take precedence over any implied atomicity or
       order of events indicated in the pseudo-code */
```

用法

STREX 命令と LDREX 命令を併用すると、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。詳細については、P. A4-53 *「LDREX」*を参照して下さい。

注

R15 の使用 <Rd>、<Rn>、<Rm> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

オペランドの制限

<Rd> には、<Rm> と <Rn> のいずれとも異なるレジスタを指定する必要があります。両者のいずれかと同じレジスタを指定した場合、結果は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。STREX 命令の実行中にデータアボートが発生した場合、次のようになります。

- メモリは更新されません。
- <Rd> は更新されません。

アライメント CP15 のレジスタ 1(A,U) != (0,0) かつ Rd<1:0> != 0b00 の場合、アライメント例外が発生します。

アンアラインドな排他ロードはサポートされません。Rd<1:0> != 0b00 で、(A,U) = (0,0) の場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.104 STRH

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	P	U	I	W	0	Rn	Rd	addr_mode	1	0	1	1	addr_mode

STRH（レジスタストアハーフワード）は、レジスタの最下位のハーフワードをメモリにストアします。アドレスがハーフワード境界にアラインしていない場合、結果は予測不能です。

構文

```
STR{<cond>}H  <Rd>, <addressing_mode>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond>が省略されている場合、AL（常時）条件が使用されます。

<Rd> 演算のソースレジスタ。<Rd>としてR15を指定した場合、結果は予測不能です。

<addressing_mode>

P. A5-33*「アドレシングモード3 - その他のロードとストア」*を参照して下さい。これによって命令のP、U、I、W、Rn、addr_modeの各ビットが決定されます。

<addressing_mode>の全形式の構文には*ベースレジスタ*<Rn>が含まれます。命令によるベースレジスタ値の変更を指定する形式もあります（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0) then
        if address[0] == 0b0 then
            Memory[address,2] = Rd[15:0]
        else
            Memory[address,2] = UNPREDICTABLE
    else    /* CP15_reg1_Ubit ==1  */
        Memory[address,2] = Rd[15:0]
    if (Shared(address))    /* ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,2)
        /* See P. A2-49「操作の概要」*/
```

用法

適切なアドレシングモードで STRH 命令を使用すると、汎用レジスタの 16 ビットデータがメモリにストアされます。PC をベースレジスタとして使用することで、PC 相対アドレシングが可能になり、ポジション・インディペンデントコード（PIC）の使用が容易になります。

注

オペランドの制限　`<addressing_mode>` でベースレジスタライトバックを指定し、`<Rd>` と `<Rn>` に同じレジスタを指定した場合、結果は予測不能です。

データアボート　データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント　ARMv6 以前は、メモリアドレスがハーフワード境界にアラインしていない場合、命令の動作は予測不能です。アライメントチェック（アドレス [0] != 0 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定の混在エンディアン形式が、アライメントチェックオプションと共にサポートされています。ARMv6 の疑似コードでは、CPSR の E ビットにより定義されたエンディアン形式の転送で、混在エンディアンのサポートが構成されていることを前提としています。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.105 STRT

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0 1	I	0	U	0	1	0	Rn	Rd	addr_mode

STRT（変換レジスタストア）は、レジスタのワードをメモリにストアします。プロセッサが特権モードにある状態でこの命令を実行すると、プロセッサがユーザモードにある場合と同様にアクセスを処理することを要求するシグナルがメモリシステムに送られます。

構文

```
STR{<cond>}T  <Rd>, <post_indexed_addressing_mode>
```

各項目の説明については以下を参照して下さい。

`<cond>` この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。`<cond>` が省略されている場合、`AL`（常時）条件が使用されます。

`<Rd>` 演算のソースレジスタ。`<Rd>` として R15 を指定した場合、ストアされる値は実装定義です。詳細については、P. A2-9 *「プログラムカウンタの読み出し」* を参照して下さい。

`<post_indexed_addressing_mode>`

P. A5-18 *「アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア」* を参照して下さい。これによって命令の I、U、Rn、addr_mode の各ビットが決定されます。この命令では、アドレシングモード 2 のポストインデクス形式のみが使用できます。これらの形式では P == 0、W == 0 となり、P および W はそれぞれビット [24] とビット [21] です。この命令では代わりに P == 0、W == 1 を使用しますが、その他のすべての点についてはアドレシングモードは同一です。

`<post_indexed_addressing_mode>` の全形式の構文には、*ベースレジスタ* `<Rn>` が含まれます。すべての形式で、命令によるベースレジスタ値の変更が指定されます（*ベースレジスタライトバック*と呼ばれます）。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    Memory[address,4] = Rd
    if (Shared(address))    /* ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,4)
            /* See P. A2-49「操作の概要」*/
```

用法

STRT は、通常はユーザモードで実行されるメモリアクセス命令をエミュレートしている（特権）例外ハンドラで使用できます。ユーザモード権限の場合と同様に、アクセスは制限されます。

注

ユーザモード この命令をユーザモードで実行すると、通常のユーザモードアクセスが実行されます。

オペランドの制限

<Rd> と <Rn> に同じレジスタを指定した場合、結果は予測不能です。

データアボート

データアボートが発生した場合のこの命令の動作の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント STR の場合と同様に、P. A4-194 *「STR」*を参照して下さい。

システム制御コプロセッサが実装に含まれていて（第 B3 章 *「システム制御コプロセッサ」* 参照）、アライメントチェックが許可されている場合、ビット [1:0] の値が 0b00 でないアドレスではアライメント例外が発生します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.106 SUB

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	0 0 1 0	S	Rn	Rd	shifter_operand

SUB（減算）は、2 番目の値から最初の値を減算します。

2 番目の値はレジスタに含まれています。最初の値はイミディエート値またはレジスタの値で、減算を実行する前にシフト可能です。

SUB 命令では、必要に応じて結果に基づいて条件コードフラグを更新することもできます。

構文

```
SUB{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S 命令の S ビット（ビット [20]）に 1 をセットし、この命令で CPSR を更新することを指定します。S を省略すると S ビットは 0 にセットされ、CPSR は命令によって変更されません。S が指定された場合の CPSR の更新には、2 タイプあります。

- <Rd> が R15 でない場合、N フラグと Z フラグが減算の結果に従ってセットされます。また、減算によって桁下がり（符号なしアンダーフロー）と符号付きオーバフローが生成されたかどうかによって、それぞれ C フラグと V フラグがセットされます。CPSR の他のビットは変更されません。
- <Rd> が R15 の場合、現在のモードの SPSR が CPSR にコピーされます。ユーザモードまたはシステムモードには SPSR がないため、これらのモードで実行した場合、この形式の命令は予測不能な結果を招きます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<shifter_operand>

2 番目のオペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」* を参照して下さい。

I ビットが 0 で、shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は SUB 命令ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-32 *「命令セットの拡張」* を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = Rn - shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(Rn - shifter_operand)
        V Flag = OverflowFrom(Rn - shifter_operand)
```

用法

SUB 命令を使用すると、ある値から別の値を減算することができます。Ri レジスタの値を削減するには、次の命令を使用します。

```
    SUB     Ri, Ri, #1
```

SUBS 命令はループカウンタデクリメントとして有効です。ループの分岐により、別々の比較条件を設定しなくても適切な終了条件のフラグをテストできます。

```
    SUBS    Ri, Ri, #1
```

この命令により、Ri のループカウンタの値をデクリメントし、さらにループカウンタが 0 に達したかどうかを確認することができます。

PC をデスティネーションレジスタとして使用し、さらに S ビットが設定されている場合は、SUB 命令を使用して割り込みなどの各種の例外から復帰することができます。詳細については、P. A2-16 *「例外」*を参照して下さい。

注

C フラグ S が指定されている場合、C フラグは次のようにセットされます。

1 桁下がりが発生しない場合

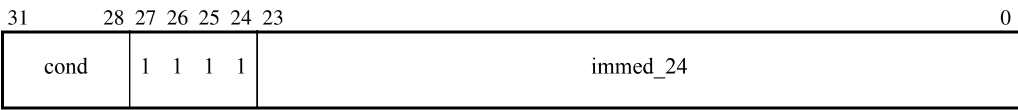
0 桁下がりが発生する場合

つまり、C フラグは NOT（桁下がり）フラグとして使用されます。桁下がり条件の反転は、次の命令によって使用されます。SBC と RSC では C フラグを NOT（桁下がり）オペランドとして使用して、C == 1 の場合は通常の減算を実行し、C == 0 の場合は通常より 1 多く減算します。

HS（符号なし＞ =）および LO（符号なし＜）の条件は、それぞれ CS（キャリーセット）および CC（キャリークリア）と同等です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.107 SWI



SWI（ソフトウェア割り込み）は SWI 例外（P. A2-16「例外」参照）を発生させます。

構文

```
SWI{<cond>}  <immed_24>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「条件フィールド」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<immed_24> 命令のビット [23:0] に設定される、24 ビットのイミディエート値。ARM プロセッサはこの値を無視しますが、オペレーティングシステムの SWI 例外ハンドラがこの値を使用して、要求されているオペレーティングシステムサービスを特定する場合があります（詳細については、この後の P. A4-212「用法」を参照して下さい）。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

ソフトウェア割り込み

動作

```
if ConditionPassed(cond) then
    R14_svc    = address of next instruction after the SWI instruction
    SPSR_svc   = CPSR
    CPSR[4:0] = 0b10011             /* Enter Supervisor mode */
    CPSR[5]   = 0                   /* Execute in ARM state */
    /* CPSR[6] is unchanged */
    CPSR[7]   = 1                   /* Disable normal interrupts */
    if high vectors configured then
        PC    = 0xFFFF0008
    else
        PC    = 0x00000008
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

`SWI` 命令は、オペレーティングシステムサービスの呼び出しに使用します。要求されているオペレーティングシステムサービスを特定する方法は、オペレーティングシステムにより指定されます。オペレーティングシステムの SWI 例外ハンドラは、要求されたサービスを特定して実行します。一般的な方法として、次の 2 つがあります。

- 命令にある 24 ビットのイミディエート値により、要求されているサービスを指定します。選択されたサービスに必要なすべてのパラメータは、汎用レジスタによって渡されます。
- 命令にある 24 ビットのイミディエート値を無視し、汎用レジスタ R0 を使用して必要なサービスを指定します。選択されたサービスに必要なすべてのパラメータは、他の汎用レジスタによって渡されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.108 SWP

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 0 0 0	Rn	Rd	SBZ	1 0 0 1	Rm

SWP（スワップ）命令は、レジスタとメモリの間でワードをスワップします。SWP 命令は、レジスタ <Rn> の値で示されたメモリアドレスからワードをロードします。ロード後に、レジスタ <Rm> の値が <Rn> の値で示されたメモリアドレスにストアされ、ロードされた元の値が <Rd> に書き込まれます。<Rd> と <Rm> に同じレジスタが指定されている場合、この命令はレジスタの値とメモリアドレスに含まれている値とをスワップします。

構文

```
SWP{<cond>}  <Rd>, <Rm>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> 命令のデスティネーションレジスタ。

<Rm> メモリにストアされる値を含むレジスタ。

<Rn> ロードするメモリアドレスを含むレジスタ。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します（ARMv6 では推奨されません）。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0) then
        temp = Memory[address,4] Rotate_Right (8 * address[1:0])
        Memory[address,4] = Rm
        Rd = temp
    else /* CP15_reg1_Ubit ==1 */
        temp = Memory[address,4]
        Memory[address,4] = Rm
        Rd = temp
    if (Shared(address)) then   /* ARMv6 */
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

```
physical_address = TLB(address)
ClearExclusiveByAddress(physical_address,processor_id,4)
/* See P. A2-49「操作の概要」*/
```

用法

SWP 命令を使用してセマフォを実装できます。この命令は、ARMv6 では推奨されません。ソフトウェアは、排他ロード / ストア命令を使用するように移行して下さい。詳細については、P. A2-44 *「同期化基本命令」* を参照して下さい。

注

R15 の使用 <Rd>、<Rn>、<Rm> として R15 を指定した場合、結果は予測不能です。

オペランドの制限

<Rn> と <Rm>、または <Rn> と <Rd> に同じレジスタを指定した場合、結果は予測不能です。

データアボート

ロードアクセスまたはストアアクセスのいずれかで正確なデータアボートが通知された場合、ロードされた値は <Rd> に書き込まれません。正確なデータアボートがロードアクセスで通知された場合、ストアアクセスは発生しません。

アライメント ARMv6 以前は、アライメントの規則は読み出しに関する LDR（P. A4-44 *「LDR」* 参照）と、書き込みに関する STR（P. A4-194 *「STR」* 参照）の規則と同じです。アライメントチェック（アドレス [1:0] != 0b00 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 以降では、CP15 のレジスタ 1(A,U) != (0,0) かつ Rn[1:0] != 0b00 の場合、アライメント例外が発生します。CP15 のレジスタ 1(A,U) == (0,0) の場合、動作は ARMv6 以前と同じです。

エンディアン形式およびアライメントの詳細については、P. A2-30 *「エンディアンのサポート」* と P. A2-38 *「アンアラインドアクセスのサポート」* を参照して下さい。

メモリモデルに関する注意点

スワップはキャッシュの有無を問わず、すべてのアクセスに対してアトミックに動作します。

スワップ処理にはメモリバリアの保証は含まれません。たとえば、マルチプロセッサシステムで重要なライトバッファのフラッシュは保証されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.109 SWPB

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	1	0	1	0	0	Rn	Rd	SBZ	1	0	0	1	Rm

SWPB（スワップバイト）は、レジスタとメモリの間でバイトをスワップします。SWPB 命令は、レジスタ <Rn> の値で示されたメモリアドレスからバイトをロードします。レジスタ <Rm> の最下位バイトの値が <Rn> で示されたメモリアドレスにストアされ、ロードされた元の値が 32 ビットワードにゼロ拡張され、そのワードがレジスタ <Rd> に書き込まれます。<Rd> と <Rm> に同じレジスタが指定されている場合、この命令はレジスタの最下位バイトの値とメモリアドレスに含まれているバイト値とをスワップします。

構文

```
SWP{<cond>}B  <Rd>, <Rm>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> 命令のデスティネーションレジスタ。

<Rm> メモリにストアされる値を含むレジスタ。

<Rn> ロードするメモリアドレスを含むレジスタ。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します（ARMv6 では推奨されません）。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    temp = Memory[address,1]
    Memory[address,1] = Rm[7:0]
    Rd = temp
    if (Shared(address))    /* ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,1)
        /* See P. A2-49「操作の概要」*/
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

SWPB 命令を使用してセマフォを実装できます。この命令は、ARMv6 では推奨されません。ソフトウェアは、排他ロード / ストア命令を使用するように移行して下さい。詳細については、P. A2-44*「同期化基本命令」*を参照して下さい。

注

R15 の使用 <Rd>、<Rn>、<Rm> として R15 を指定した場合、結果は予測不能です。

オペランドの制限 <Rn> と <Rm>、または <Rn> と <Rd> に同じレジスタを指定した場合、結果は予測不能です。

データアボート ロードアクセスまたはストアアクセスのいずれかで正確なデータアボートが通知された場合、ロードされた値は <Rd> に書き込まれません。正確なデータアボートがロードアクセスで通知された場合、ストアアクセスは発生しません。

メモリモデルに関する注意点

スワップはキャッシュの有無を問わず、すべてのアクセスに対してアトミックに動作します。

スワップ処理にはメモリバリアの保証は含まれません。たとえば、マルチプロセッサシステムで重要なライトバッファのフラッシュは保証されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.110 SXTAB

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–10	9–8	7	6	5	4	3–0
cond	0	1	1	0	1	0	1	0	Rn	Rd	rotate	SBZ	0	1	1	1	Rm

SXTAB は、レジスタから 8 ビット値を抽出し、値を 32 ビットに符号拡張して、その結果を他のレジスタの値に加算します。8 ビットの値を抽出する前に、0、8、16、24 ビットのローテートを指定できます。

構文

```
SXTAB{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

<rotation> 以下のいずれかが使用できます。

- ROR #8.rotate フィールドで 0b01 としてエンコードされます。
- ROR #16.rotate フィールドで 0b10 としてエンコードされます。
- ROR #24.rotate フィールドで 0b11 としてエンコードされます。
- 省略。rotate フィールドで 0b00 としてエンコードされます。

—— 注 ——

アセンブラがシフト値として #0 を許可し、シフトなしの場合や LSL #0 と同じように扱う場合、ここで ROR #0 も許可する必要があります。これは、<rotation> の省略と同じです。

——————

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    operand2  = Rm Rotate_Right(8 * rotate)
    Rd = Rn + SignExtend(operand2[7:0])
```

用法

SXTAB命令を使用すると、C/C++の**signed char**値に作用する多数の命令シーケンスに組み込まれた個々の符号拡張命令を削除することができます。

注

R15 の使用　レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

—— 注 ——

<Rn>レジスタにR15を使用すると、アセンブラによってエラーが生成されます。

——————

エンコード　命令の <Rn> フィールドの値が 0b1111 である場合、この命令は SXTB 命令（P. A4-223「*SXTB*」参照）です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.111 SXTAB16

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–10	9–8	7	6	5	4	3–0
cond	0	1	1	0	1	0	0	0	Rn	Rd	rotate	SBZ	0	1	1	1	Rm

SXTAB16 は、レジスタから 2 つの 8 ビット値を抽出し、それぞれの値を 16 ビットに符号拡張して、その結果を他のレジスタの 2 つの 16 ビット値に加算します。8 ビットの値を抽出する前に、0、8、16、24 ビットのローテートを指定できます。

構文

```
SXTAB16{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

<rotation> 以下のいずれかが使用できます。

- ROR #8.rotate フィールドで 0b01 としてエンコードされます。
- ROR #16.rotate フィールドで 0b10 としてエンコードされます。
- ROR #24.rotate フィールドで 0b11 としてエンコードされます。
- 省略。rotate フィールドで 0b00 としてエンコードされます。

—— 注 ——

アセンブラがシフト値として #0 を許可し、シフトなしの場合や LSL #0 と同じように扱う場合、ここで ROR #0 も許可する必要があります。これは、<rotation> の省略と同じです。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    operand2  = Rm Rotate_Right(8 * rotate)
    Rd[15:0]  = Rn[15:0]  + SignExtend(operand2[7:0])
    Rd[31:16] = Rn[31:16] + SignExtend(operand2[23:16])
```

用法

SXTAB16 命令は、符号付きのバイト値の配列を操作するとき、イミディエート値を高精度に保つ必要がある場合に使用します。類似の使用例については、P. A4-277「*UXTAB16*」を参照して下さい。

注

R15 の使用 レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

——— 注 ———

<Rn>レジスタにR15を使用すると、アセンブラによってエラーが生成されます。

———

エンコード 命令の <Rn> フィールドの値が 0b1111 である場合、この命令は SXTB16 命令（P. A4-225「*SXTB16*」参照）です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.112 SXTAH

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–10	9–8	7	6	5	4	3–0
cond	0	1	1	0	1	0	1	1	Rn	Rd	rotate	SBZ	0	1	1	1	Rm

SXTAH 命令は、レジスタから 16 ビット値を抽出し、32 ビットに符号拡張して、その結果を他のレジスタの値に加算します。16ビットの値を抽出する前に、0、8、16、24ビットのローテートを指定できます。

構文

```
SXTAH{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

各項目の説明については以下を参照して下さい。

<cond>
: この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd>
: デスティネーションレジスタ。

<Rn>
: 最初のオペランドを含むレジスタ。

<Rm>
: 2 番目のオペランドを含むレジスタ。

<rotation>
: 以下のいずれかが使用できます。

- ROR #8.rotate フィールドで 0b01 としてエンコードされます。
- ROR #16.rotate フィールドで 0b10 としてエンコードされます。
- ROR #24.rotate フィールドで 0b11 としてエンコードされます。
- 省略。rotate フィールドで 0b00 としてエンコードされます。

——— 注 ———

アセンブラがシフト値として #0 を許可し、シフトなしの場合や LSL #0 と同じように扱う場合、ここで ROR #0 も許可する必要があります。これは、<rotation> の省略と同じです。

———————

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    operand2 = Rm Rotate_Right(8 * rotate)
    Rd       = Rn + SignExtend(operand2[15:0])
```

用法

SXTAH 命令を使用すると、C/C++ の **signed short** 値に作用する多数の命令シーケンスに組み込まれた個々の符号拡張命令を削除することができます。

注

R15 の使用 レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

注

<Rn>レジスタにR15を使用すると、アセンブラによってエラーが生成されます。

エンコード 命令の <Rn> フィールドの値が 0b1111 である場合、この命令は SXTH 命令（P. A4-227「*SXTH*」参照）です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.113 SXTB

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11–10	9–8	7	6	5	4	3–0
cond	0	1	1	0	1	0	1	0	1	1	1	1	Rd	rotate	SBZ	0	1	1	1	Rm

SXTB は、レジスタから 8 ビット値を抽出して、32 ビットに符号拡張します。8 ビットの値を抽出する前に、0、8、16、24 ビットのローテートを指定できます。

構文

```
SXTB{<cond>}  <Rd>, <Rm>{, <rotation>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> オペランドレジスタを指定します。

<rotation> 以下のいずれかが使用できます。

- ROR #8.rotate フィールドで 0b01 としてエンコードされます。
- ROR #16.rotate フィールドで 0b10 としてエンコードされます。
- ROR #24.rotate フィールドで 0b11 としてエンコードされます。
- 省略。rotate フィールドで 0b00 としてエンコードされます。

——— 注 ———

アセンブラがシフト値として #0 を許可し、シフトなしの場合や LSL #0 と同じように扱う場合、ここで ROR #0 も許可する必要があります。これは、<rotation> の省略と同じです。

———————

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    operand2 = Rm Rotate_Right(8 * rotate)
    Rd[31:0] = SignExtend(operand2[7:0])
```

用法

SXTB はバイトからワードへの符号拡張、たとえば C/C++ で **signed char** を操作する命令シーケンスで使用します。

注

R15 の使用　　レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.114 SXTB16

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11–10	9–8	7	6	5	4	3–0
cond	0	1	1	0	1	0	0	0	1	1	1	1	Rd	rotate	SBZ	0	1	1	1	Rm

SXTB16 は、レジスタから 2 つの 8 ビット値を抽出して、それぞれの値を 16 ビットに符号拡張します。8 ビットの値を抽出する前に、0、8、16、24 ビットのローテートを指定できます。

構文

```
SXTB16{<cond>}  <Rd>, <Rm>{, <rotation>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> オペランドレジスタを指定します。

<rotation> 以下のいずれかが使用できます。

- ROR #8.rotate フィールドで 0b01 としてエンコードされます。
- ROR #16.rotate フィールドで 0b10 としてエンコードされます。
- ROR #24.rotate フィールドで 0b11 としてエンコードされます。
- 省略。rotate フィールドで 0b00 としてエンコードされます。

—— 注 ——

アセンブラがシフト値として #0 を許可し、シフトなしの場合や LSL #0 と同じように扱う場合、ここで ROR #0 も許可する必要があります。これは、<rotation> の省略と同じです。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    operand2  = Rm Rotate_Right(8 * rotate)
    Rd[15:0]  = SignExtend(operand2[7:0])
    Rd[31:16] = SignExtend(operand2[23:16])
```

用法

SXTB16 命令は、符号付きのバイト値の配列を操作するとき、イミディエート値を高精度に保つ必要がある場合に使用します。類似の使用例については、P. A4-277「*UXTAB16*」を参照して下さい。

注

R15 の使用 レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.115 SXTH

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11–10	9–8	7	6	5	4	3–0
cond	0	1	1	0	1	0	1	1	1	1	1	1	Rd	rotate	SBZ	0	1	1	1	Rm

SXTH は、レジスタから 16 ビット値を抽出して、32 ビットに符号拡張します。16 ビットの値を抽出する前に、0、8、16、24 ビットのローテートを指定できます。

構文

```
SXTH{<cond>}  <Rd>, <Rm>{, <rotation>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> オペランドレジスタを指定します。

<rotation> 以下のいずれかが使用できます。

- ROR #8.rotate フィールドで 0b01 としてエンコードされます。
- ROR #16.rotate フィールドで 0b10 としてエンコードされます。
- ROR #24.rotate フィールドで 0b11 としてエンコードされます。
- 省略。rotate フィールドで 0b00 としてエンコードされます。

——— 注 ———

アセンブラがシフト値として #0 を許可し、シフトなしの場合や LSL #0 と同じように扱う場合、ここで ROR #0 も許可する必要があります。これは、<rotation> の省略と同じです。

———————

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    operand2 = Rm Rotate_Right(8 * rotate)
    Rd[31:0] = SignExtend(operand2[15:0])
```

用法

SXTH はハーフワードからワードへの符号拡張、たとえば C/C++ で **signed short** 値を操作する命令シーケンスで使用します。

注

R15 の使用 レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.116 TEQ

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 0 0 1	1	Rn	SBZ	shifter_operand

TEQ（等価テスト）は、レジスタの値を別の数値と比較します。2 つの値の排他的論理和の演算結果に基づいて条件フラグが変更されるため、以降の命令を条件付きで実行することができます。

構文

```
TEQ{<cond>}  <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

`<cond>` この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。`<cond>` が省略されている場合、AL（常時）条件が使用されます。

`<Rn>` 最初のオペランドを含むレジスタ。

`<shifter_operand>`

2 番目のオペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」* を参照して下さい。

I ビットが 0 で、shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は TEQ 命令ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-35 *「乗算命令拡張空間」* を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    alu_out = Rn EOR shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = shifter_carry_out
    V Flag = unaffected
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

TEQ 命令を使用して、V フラグへの影響（CMP が行うような）なしに、2 つの値が等しいかどうかをテストできます。多くの場合、C フラグにも影響を与えません。TEQ 命令は、2 つの値の符号が同じかどうかをテストする場合にも役立ちます。比較した後の N フラグは、2 つのオペランドの符号ビットを排他的論理和で演算した結果になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.117 TST

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	0	I	1	0	0	0	1	Rn	SBZ	shifter_operand

`TST`（テスト）命令は、レジスタの値を別の数値と比較します。2 つの値の論理積の演算結果に基づいて条件フラグが変更されるため、以降の命令を条件付きで実行することができます。

構文

```
TST{<cond>}  <Rn>, <shifter_operand>
```

各項目の説明については以下を参照して下さい。

`<cond>` この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。`<cond>` が省略されている場合、`AL`（常時）条件が使用されます。

`<Rn>` 最初のオペランドを含むレジスタ。

`<shifter_operand>`

2 番目のオペランドを指定します。このオペランドのオプションについては、各オプションで I ビット（ビット [25]）および命令の shifter_operand ビット（ビット [11:0]）をセットする方法も含めて、P. A5-2 *「アドレシングモード 1 - データ処理オペランド」*を参照して下さい。

I ビットが 0 で、shifter_operand のビット [7] とビット [4] が共に 1 の場合、この命令は `TST` 命令ではありません。この場合にこの命令がどのような動作を行うかについては、P. A3-35 *「乗算命令拡張空間」*を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    alu_out = Rn AND shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = shifter_carry_out
    V Flag = unaffected
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

TST は、レジスタに含まれる特定のビットサブセットのうち、最低 1 つのビットがセットされているかどうかを判定するために使用します。TST の特に一般的な使用目的は、ある 1 つのビットがセット / クリアされているかどうかのテストです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.118 UADD16

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	1	0	1	Rn	Rd	SBO	0	0	0	1	Rm

UADD16（符号なし加算）命令は、16 ビット符号なし整数の加算を 2 つ実行します。加算のキャリーフラグとして、CPSR の GE ビットをセットします。

構文

```
UADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[15:0]  = Rn[15:0] + Rm[15:0]
    GE[1:0]   = if CarryFrom16(Rn[15:0] + Rm[15:0]) == 1 then 0b11 else 0
    Rd[31:16] = Rn[31:16] + Rm[31:16]
    GE[3:2]   = if CarryFrom16(Rn[31:16] + Rm[31:16]) == 1 then 0b11 else 0
```

用法

UADD16 命令は、SADD16 と同じ結果値を算出します。ただし、GE フラグの値は符号付き演算ではなく符号なし演算に基づいています。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.119 UADD8

<table>
<tr><td>31-28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19-16</td><td>15-12</td><td>11-8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3-0</td></tr>
<tr><td>cond</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>Rn</td><td>Rd</td><td>SBO</td><td>1</td><td>0</td><td>0</td><td>1</td><td>Rm</td></tr>
</table>

UADD8 は、8 ビット符号なし整数の加算を 4 つ実行します。加算のキャリーフラグとして、CPSR の GE ビットをセットします。

構文

```
UADD8{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[7:0]   = Rn[7:0] + Rm[7:0]
    GE[0]     = CarryFrom8(Rn[7:0] + Rm[7:0])
    Rd[15:8]  = Rn[15:8] + Rm[15:8]
    GE[1]     = CarryFrom8(Rn[15:8] + Rm[15:8])
    Rd[23:16] = Rn[23:16] + Rm[23:16]
    GE[2]     = CarryFrom8(Rn[23:16] + Rm[23:16])
    Rd[31:24] = Rn[31:24] + Rm[31:24]
    GE[3]     = CarryFrom8(Rn[31:24] + Rm[31:24])
```

用法

UADD8 命令は、SADD8 と同じ結果値を算出します。ただし、GE フラグの値は符号付き演算ではなく符号なし演算に基づいています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.120 UADDSUBX

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 1 0 1	Rn	Rd	SBO	0 0 1 1	Rm

UADDSUBX（交換符号なし加減算）命令は、16 ビット符号なし整数の加算と 16 ビット符号なし整数の減算を 1 つづつ実行します。演算を実行する前に、2 番目のオペランドにある 2 つのハーフワードの交換を行います。この命令は、加算と減算の結果に従って CPSR の GE ビットをセットします。

構文

```
UADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    sum       = Rn[31:16] + Rm[15:0]    /* unsigned addition */
    Rd[31:16] = sum[15:0]
    GE[3:2]   = if CarryFrom16(Rn[31:16] + Rm[15:0]) then 0b11 else 0
    diff      = Rn[15:0] - Rm[31:16]    /* unsigned subtraction */
    Rd[15:0]  = diff[15:0]
    GE[1:0]   = if BorrowFrom(Rn[15:0] - Rm[31:16]) then 0b11 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

UADDSUBX 命令は、SADDSUBX と同じ結果値を算出します。ただし、GE フラグの値は符号付き演算ではなく符号なし演算に基づいています。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.121 UHADD16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	1	Rn	Rd	SBO	0	0	0	1	Rm

UHADD16（符号なし半加算）命令は、16 ビット符号なし整数の加算を 2 つ実行して、各演算結果の値を半分にします。GE フラグは変更されません。

構文

```
UHADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    sum       = Rn[15:0] + Rm[15:0]   /* Unsigned addition */
    Rd[15:0]  = sum[16:1]
    sum       = Rn[31:16] + Rm[31:16] /* Unsigned addition */
    Rd[31:16] = sum[16:1]
```

用法

UHADD16 命令の用途は UADD16 命令（P. A4-233 *「UADD16」* 参照）と類似しています。UHADD16 命令はオペランドの平均値を算出します。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.122 UHADD8

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 1 1 1	Rn	Rd	SBO	1 0 0 1	Rm

UHADD16 は、8 ビット符号なし整数の加算を 4 つ実行して、各演算結果の値を半分にします。GE フラグは変更されません。

構文

```
UHADD8{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    sum       = Rn[7:0] + Rm[7:0]        /* Unsigned addition */
    Rd[7:0]   = sum[8:1]
    sum       = Rn[15:8] + Rm[15:8]      /* Unsigned addition */
    Rd[15:8]  = sum[8:1]
    sum       = Rn[23:16] + Rm[23:16]    /* Unsigned addition */
    Rd[23:16] = sum[8:1]
    sum       = Rn[31:24] + Rm[31:24]    /* Unsigned addition */
    Rd[31:24] = sum[8:1]
```

用法

UHADD8 命令の用途は UADD8 命令（P. A4-234*「UADD8」*参照）と類似しています。UHADD8 命令はオペランドの平均値を算出します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.123 UHADDSUBX

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	1	1	1	Rn	Rd	SBO	0	0	1	1	Rm

UHADDSUBX（交換符号なし半加減算）命令は、16 ビット符号なし整数の加算と 16 ビット符号なし整数の減算を 1 つづつ実行し、それぞれの結果の値を半分にします。演算を実行する前に、2 番目のオペランドにある 2 つのハーフワードの交換を行います。

GE フラグは変更されません。

構文

```
UHADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    sum       = Rn[31:16] + Rm[15:0]   /* Unsigned addition */
    Rd[31:16] = sum[16:1]
    diff      = Rn[15:0] - Rm[31:16]   /* Unsigned subtraction */
    Rd[15:0]  = diff[16:1]
```

用法

UHADDSUBX 命令の用途は UADDSUBX 命令（P. A4-236 *「UADDSUBX」*参照）と類似しています。UHADDSUBX 命令は、演算結果の値を半分にします。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 `<Rd>`、`<Rm>`、`<Rn>` のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.124 UHSUB16

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	1	1	1	Rn	Rd	SBO	0	1	1	1	Rm

UHSUB16（符号なし半減算）は、16 ビット符号なし整数の減算を 2 つ実行して、各演算結果の値を半分にします。GE フラグは変更されません。

構文

```
UHSUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    diff      = Rn[15:0] - Rm[15:0]     /* Unsigned subtraction */
    Rd[15:0]  = diff[16:1]
    diff      = Rn[31:16] - Rm[31:16]   /* Unsigned subtraction */
    Rd[31:16] = diff[16:1]
```

用法

UHSUB16 命令の用途は USUB16 命令（P. A4-270「*USUB16*」参照）と類似しています。UHSUB16 命令で算出される値は、単なる減算の差ではなく、減算した結果を半分にしたものです。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.125 UHSUB8

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 1 1 1	Rn	Rd	SBO	1 1 1 1	Rm

UHSUB8 は、8 ビット符号なし整数の減算を 4 つ実行して、各演算結果の値を半分にします。GE フラグは変更されません。

構文

```
UHSUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    diff      = Rn[7:0] - Rm[7:0]       /* Unsigned subtraction */
    Rd[7:0]   = diff[8:1]
    diff      = Rn[15:8] - Rm[15:8]     /* Unsigned subtraction */
    Rd[15:8]  = diff[8:1]
    diff      = Rn[23:16] - Rm[23:16]   /* Unsigned subtraction */
    Rd[23:16] = diff[8:1]
    diff      = Rn[31:24] - Rm[31:24]   /* Unsigned subtraction */
    Rd[31:24] = diff[8:1]
```

用法

UHSUB8 命令の用途は USUB8 命令（P. A4-271 *「USUB8」* 参照）と類似しています。UHSUB8 命令で算出される値は、単なる減算の差ではなく、減算した結果を半分にしたものです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.126 UHSUBADDX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	1	Rn	Rd	SBO	0	1	0	1	Rm

UHSUBADDX（交換符号なし半減算 / 加算）命令は、16 ビット符号なし整数の減算と 16 ビット符号なし整数の加算を 1 つづつ実行し、各結果の値を半分にします。演算を実行する前に、2 番目のオペランドにある 2 つのハーフワードの交換を行います。

GE フラグは変更されません。

構文

```
UHSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    diff      = Rn[31:16] - Rm[15:0]      /* Unsigned subtraction */
    Rd[31:16] = diff[16:1]
    sum       = Rn[15:0]  + Rm[31:16]     /* Unsigned addition */
    Rd[15:0]  = sum[16:1]
```

用法

UHSUBADDX 命令の用途は USUBADDX 命令（P. A4-273 *「USUBADDX」* 参照）と類似しています。UHSUBADDX 命令で算出される値は、単なる減算の差と加算の和ではなく、減算の算出値と平均値を半分にしたものです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用

`<Rd>`、`<Rm>`、`<Rn>` のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.127 UMAAL

31 - 28	27	26	25	24	23	22	21	20	19 - 16	15 - 12	11 - 8	7	6	5	4	3 - 0
cond	0	0	0	0	0	1	0	0	RdHi	RdLo	Rs	1	0	0	1	Rm

UMAAL（符号なし積和累算 long 型）命令は、レジスタ <Rm> の符号なし値とレジスタ <Rs> の符号なし値を乗算して、64 ビットの積を算出します。この積は、<RdHi> にある符号なし 32 ビット値と <RdLo> にある符号なし 32 ビット値の両方に加算され、その合計が 64 ビット値として <RdHi> と <RdLo> に書き戻されます。フラグは変更されません。

構文

```
UMAAL{<cond>}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<RdLo> <Rm> と <Rs> の積に加算する 32 ビット値の 1 つを指定します。また、この命令の演算結果の下位 32 ビットが格納されるデスティネーションレジスタでもあります。

<RdHi> <Rm> と <Rs> の積に加算するもう 1 つの 32 ビット値を指定します。また、この命令の演算結果の上位 32 ビットが格納されるデスティネーションレジスタでもあります。

<Rm> <Rs> の値に乗算される符号なし値を保持します。

<Rs> <Rm> の値に乗算される符号なし値を保持します。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    result = Rm * Rs + RdLo + RdHi  /* Unsigned multiplication and additions */
    RdLo = result[31:0]
    RdHi = result[63:32]
```

用法

32 ビット符号なし乗算に対して 2 つの 32 ビット値を加算すると、暗号化アプリケーションで役立つ関数になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 <RdHi>、<RdLo>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

オペランドの制限 <RdLo> と <RdHi> が同じレジスタの場合、結果は予測不能です。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.128 UMLAL

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0 1 0 1	S	RdHi	RdLo	Rs	1 0 0 1	Rm

UMLAL（符号なし積和演算 long 型）は、レジスタ <Rm> の符号なしの値とレジスタ <Rs> の符号なしの値とを乗算して、64 ビットの積を算出します。この積は、<RdHi> と <RdLo> に保持されている 64 ビット値に加算され、その合計が <RdHi> と <RdLo> に書き戻されます。結果に基づいて条件コードフラグを更新します。

構文

```
UMLAL{<cond>}{S}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S 命令の S ビット（ビット [20]）に 1 をセットし、積和の結果に従って CPSR の N フラグと Z フラグをセットすることを指定します。S を省略すると、命令の S ビットは 0 にセットされ、CPSR 全体が命令の影響を受けません。

<RdLo> <Rm> および <Rs> の積に加算される下位 32 ビット値を含むレジスタで、結果の下位 32 ビットのデスティネーションレジスタでもあります。

<RdHi> <Rm> および <Rs> の積に加算される上位 32 ビット値を含むレジスタで、結果の上位 32 ビットのデスティネーションレジスタでもあります。

<Rm> <Rs> の値に乗算される符号付き値を保持します。

<Rs> <Rm> の値に乗算される符号付き値を保持します。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    RdLo = (Rm * Rs)[31:0] + RdLo    /* Unsigned multiplication */
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

用法

UMLAL 命令を使用すると、符号なし変数が乗算され、64 ビット値が生成されます。この値は 2 つのデスティネーション汎用レジスタにある 64 ビット値に加算されます。結果は 2 つのデスティネーション汎用レジスタに書き戻されます。

注

R15 の使用 <RdHi>、<RdLo>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

オペランドの制限 <RdHi> と <RdLo> が同じレジスタの場合、結果は予測不能です。

以前の説明には、<RdHi> と <Rm> または <RdLo> と <Rm> に同じレジスタを指定した場合は、予測不能な結果を招くことが記載されていました。高性能の乗算器では結果をライトバックする前にすべてのオペランドを読み込むため、ARMv6 では制限は全くなく、該当する ARM の v4 および v5 のどの実装でも、この制限は必要ありません。

短縮処理 乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

C フラグと V フラグ ARMv5 以降では、UMLALS は C フラグと V フラグが変更されないように定義されています。以前のバージョンのアーキテクチャでは、UMLALS 命令の実行後の C フラグと V フラグの値は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.129 UMULL

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	1	0	0	S	RdHi	RdLo	Rs	1	0	0	1	Rm

UMULL（符号なし乗算 long 型）命令は、レジスタ <Rm> の符号なし値とレジスタ <Rs> の符号なし値とを乗算して、64 ビットの積を算出します。この演算結果の上位 32 ビットは、<RdHi> に格納されます。下位 32 ビットは、<RdLo> に格納されます。この 64 ビットの結果に基づいて、条件コードフラグを更新します。

構文

```
UMULL{<cond>}{S}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

S 命令の S ビット（ビット [20]）に 1 をセットし、乗算の結果に従って CPSR の N フラグと Z フラグをセットすることを指定します。S を省略すると、命令の S ビットは 0 にセットされ、CPSR 全体が命令の影響を受けません。

<RdLo> 結果の下位 32 ビットがストアされるレジスタ。

<RdHi> 結果の上位 32 ビットがストアされるレジスタ。

<Rm> <Rs> の値に乗算される符号付き値を保持します。

<Rs> <Rm> の値に乗算される符号付き値を保持します。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    RdHi = (Rm * Rs)[63:32]    /* Unsigned multiplication */
    RdLo = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

用法

UMULL 命令は、符号なし変数を乗算して、64 ビットの結果を 2 つの汎用レジスタに書き込みます。

注

R15 の使用　<RdHi>、<RdLo>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

オペランドの制限　<RdHi> と <RdLo> が同じレジスタの場合、結果は予測不能です。

以前の説明には、<RdHi> と <Rm> または <RdLo> と <Rm> に同じレジスタを指定した場合は、予測不能な結果を招くことが記載されていました。高性能の乗算器では結果をライトバックする前にすべてのオペランドを読み込むため、ARMv6 では制限は全くなく、該当する ARM の v4 および v5 のどの実装でも、この制限は必要ありません。

短縮処理　乗算器の実装が短縮処理をサポートする場合、<Rs> オペランドの値に対して実装する必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。

C フラグと V フラグ　ARMv5 以上では、UMULLS 命令は C フラグと V フラグが変更されないように定義されています。以前のバージョンのアーキテクチャでは、UMULLS 命令の実行後の C フラグと V フラグの値は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.130 UQADD16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	0	Rn	Rd	SBO	0	0	0	1	Rm

UQADD16（符号なし飽和加算）は、16 ビット符号なし整数の加算を 2 つ実行します。結果は 16 ビット符号なし整数の範囲である $0 \leq x \leq 2^{16} - 1$ に飽和します。GE フラグは変更されません。

構文

```
UQADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rn[15:0]  + Rm[15:0],  16)
    Rd[31:16] = UnsignedSat(Rn[31:16] + Rm[31:16], 16)
```

用法

UQADD16 命令は、符号なし飽和演算であること除いて、UADD16 と同様に使用します。UQADD16 命令では、SEL で使用する GE ビットはセットされません。詳細については、P. A4-233 *「UADD16」* を参照して下さい。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.131 UQADD8

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	1	1	0	Rn	Rd	SBO	1	0	0	1	Rm

UQADD8 命令は、8 ビット符号なし整数の加算を 4 つ実行します。結果は 8 ビット符号なし整数の範囲である $0 \leq x \leq 2^8 - 1$ に飽和します。GE フラグは変更されません。

構文

```
UQADD8{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[7:0]   = UnsignedSat(Rn[7:0]   + Rm[7:0],   8)
    Rd[15:8]  = UnsignedSat(Rn[15:8]  + Rm[15:8],  8)
    Rd[23:16] = UnsignedSat(Rn[23:16] + Rm[23:16], 8)
    Rd[31:24] = UnsignedSat(Rn[31:24] + Rm[31:24], 8)
```

用法

UQADD8 命令は、符号なし飽和演算であることを除いて、UADD8 と同様に使用します。UQADD8 命令は、SEL で使用する GE ビットのセットを行いません。詳細については、P. A4-234 *「UADD8」* を参照して下さい。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.132 UQADDSUBX

31 … 28	27	26	25	24	23	22	21	20	19 … 16	15 … 12	11 … 8	7	6	5	4	3 … 0
cond	0	1	1	0	0	1	1	0	Rn	Rd	SBO	0	0	1	1	Rm

UQADDSUBX（交換符号なし飽和加減算）命令は、16 ビット符号なし整数の加算と 16 ビット符号なし整数の減算を 1 つづつ実行します。結果は、16 ビット符号なし整数の範囲である $0 \leq x \leq 2^{16} - 1$ に飽和します。演算を実行する前に、2 番目のオペランドの 2 つのハーフワードを交換します。GE フラグは変更されません。

構文

```
UQADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rn[15:0]  - Rm[31:16], 16)
    Rd[31:16] = UnsignedSat(Rn[31:16] + Rm[15:0],  16)
```

用法

UQADDSUBX 命令は、符号なし飽和演算であることを除いて、UADDSUBX 命令と同様に使用します。UQADDSUBX 命令は、SEL で使用する GE ビットのセットを行いません。詳細については、P. A4-236 *「UADDSUBX」* を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 `<Rd>`、`<Rm>`、`<Rn>` のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.133 UQSUB16

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
cond	0	1	1	0	0	1	1	0	Rn	Rd	SBO	0	1	1	1	Rm

UQSUB16（符号なし飽和減算）は、16 ビット符号なしの減算を 2 つ実行します。結果は 16 ビット符号なし整数の範囲である $0 \leq x \leq 2^{16} - 1$ に飽和します。GE フラグは変更されません。

構文

```
UQSUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rn[15:0]  - Rm[15:0],  16)
    Rd[31:16] = UnsignedSat(Rn[31:16] - Rm[31:16], 16)
```

用法

UQSUB16 命令は、符号なし飽和演算であることを除いて、USUB16 と同様に使用します。UQSUB16 命令は、SEL で使用する GE ビットのセットを行いません。詳細については、P. A4-181 *「SSUB16」* を参照して下さい。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.134 UQSUB8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	0	Rn	Rd	SBO	1	1	1	1	Rm

UQSUB8 は、8 ビットの減算を 4 つ実行します。結果は、8 ビット符号なし整数の範囲である $0 \leq x \leq 2^8 - 1$ に飽和します。GE フラグは変更されません。

構文

```
UQSUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[7:0]   = UnsignedSat(Rn[7:0]   - Rm[7:0],   8)
    Rd[15:8]  = UnsignedSat(Rn[15:8]  - Rm[15:8],  8)
    Rd[23:16] = UnsignedSat(Rn[23:16] - Rm[23:16], 8)
    Rd[31:24] = UnsignedSat(Rn[31:24] - Rm[31:24], 8)
```

用法

UQSUB8 命令は、符号なし飽和演算であることを除いて、USUB8 命令と同様に使用します。UQSUB8 命令は、SEL で使用する GE ビットのセットを行いません。詳細については、P. A4-183 *「SSUB8」* を参照して下さい。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.135 UQSUBADDX

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	1	1	0	Rn	Rd	SBO	0	1	0	1	Rm

UQSUBADDX（交換符号なし飽和減算 / 加算）命令は、16 ビットの整数の減算と 16 ビットの整数の加算を 1 つづつ実行します。結果は、16 ビット符号なし整数の範囲である
$0 \leq x \leq 2^{16} - 1$ の範囲に飽和します。演算を実行する前に、2 番目のオペランドの 2 つのハーフワードを交換します。GE フラグは変更されません。

構文

```
UQSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[31:16] = UnsignedSat(Rn[31:16] - Rm[15:0],  16)
    Rd[15:0]  = UnsignedSat(Rn[15:0]  + Rm[31:16], 16)
```

用法

UQSUBADDX 命令は、符号なし飽和演算であることを除いて、USUBADDX 命令と同様に使用します。UQSUBADDX 命令は、SEL で使用する GE ビットのセットを行いません。詳細については、P. A4-236 *「UADDSUBX」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 `<Rd>`、`<Rm>`、`<Rn>` のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.136 USAD8

31 28	27	26	25	24	23	22	21	20	19 16	15	14	13	12	11 8	7	6	5	4	3 0
cond	0	1	1	1	1	0	0	0	Rd	1	1	1	1	Rs	0	0	0	1	Rm

USAD8（符号なし絶対差合計）命令は、符号なし 8 ビット減算を 4 つ実行して、差の絶対値をすべて加算します。

構文

```
USAD8{<cond>}  <Rd>, <Rm>, <Rs>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> 最初のオペランドを含むレジスタ。

<Rs> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then

    if Rm[7:0] < Rs[7:0] then         /* Unsigned comparison */
        diff1 = Rs[7:0] - Rm[7:0]
    else
        diff1 = Rm[7:0] - Rs[7:0]

    if Rm[15:8] < Rs[15:8] then       /* Unsigned comparison */
        diff2 = Rs[15:8] - Rm[15:8]
    else
        diff2 = Rm[15:8] - Rs[15:8]

    if Rm[23:16] < Rs[23:16] then     /* Unsigned comparison */
        diff3 = Rs[23:16] - Rm[23:16]
    else
        diff3 = Rm[23:16] - Rs[23:16]

    if Rm[31:24] < Rs[31:24] then     /* Unsigned comparison */
        diff4 = Rs[31:24] - Rm[31:24]
    else
        diff4 = Rm[31:24] - Rs[31:24]

    Rd = ZeroExtend(diff1) + ZeroExtend(diff2)
                           + ZeroExtend(diff3) + ZeroExtend(diff4]
```

用法

USAD8 を使用すると、ビデオ画像動作予測の計算で最初の 4 バイトを処理することができます。

注

R15 の使用 <Rd>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.137 USADA8

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	1	1	0	0	0	Rd	Rn	Rs	0	0	0	1	Rm

USADA8（符号なし絶対差合計累算）は、符号なし 8 ビット減算を 4 つ実行して、差の絶対値を 32 ビット累算オペランドに加算します。

構文

```
USADA8{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> 最初のメインオペランドを含むレジスタ。

<Rs> 2 番目のメインオペランドを含むレジスタ。

<Rn> 累算オペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then

    if Rm[7:0] < Rs[7:0] then         /* Unsigned comparison */
        diff1 = Rs[7:0] - Rm[7:0]
    else
        diff1 = Rm[7:0] - Rs[7:0]

    if Rm[15:8] < Rs[15:8] then       /* Unsigned comparison */
        diff2 = Rs[15:8] - Rm[15:8]
    else
        diff2 = Rm[15:8] - Rs[15:8]

    if Rm[23:16] < Rs[23:16] then     /* Unsigned comparison */
        diff3 = Rs[23:16] - Rm[23:16]
    else
        diff3 = Rm[23:16] - Rs[23:16]

    if Rm[31:24] < Rs[31:24] then     /* Unsigned comparison */
        diff4 = Rs[31:24] - Rm[31:24]
    else
        diff4 = Rm[31:24] - Rs[31:24]

    Rd = Rn + ZeroExtend(diff1) + ZeroExtend(diff2)
                           + ZeroExtend(diff3) + ZeroExtend(diff4]
```

用法

USADA8 命令は、ビデオ画像動作予測の計算に使用できます。

注

R15 の使用　<Rd>、<Rm>、<Rs> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

エンコード　命令の <Rn> フィールドの値が 0b1111 である場合、この命令は USAD8 命令（P. A4-262「*USAD8*」参照）です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.138 USAT

31–28	27	26	25	24	23	22	21	20–16	15–12	11–7	6	5	4	3–0
cond	0	1	1	0	1	1	1	sat_imm	Rd	shift_imm	sh	0	1	Rm

USAT（符号なし飽和）命令は、符号付きの値を符号なし範囲に飽和させます。飽和するビット位置は選択できます。

飽和の前に値にシフトを適用できます。

演算が飽和すると、Q フラグがセットされます。

構文

```
USAT{<cond>}  <Rd>, #<immed>, <Rm>{, <shift>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<immed> 飽和のビット位置を指定します。このビット位置は 0 ～ 31 の間で指定します。この値は命令の sat_imm フィールドにエンコードされます。

<Rm> 飽和する符号付き値を含むレジスタ。

<shift> オプションのシフトを指定します。指定する場合は、以下のいずれかを選択します。

- LSL #*N*. *N* は 0 ～ 31 の範囲で指定します。
  これは、sh == 0 および shift_imm == *N* としてエンコードされます。
- ASR #*N*. *N* は 1 ～ 32 の範囲で指定します。これは、sh == 1 で、*N* == 32 の場合は shift_imm == 0、それ以外の場合は shift_imm == *N* にエンコードされます。

<shift> が省略されている場合、LSL #0 が使用されます。

戻り値

Rd に戻される値は次のとおりです。

0 X < 0 の場合

X 0 <= X < 2^n の場合

2^n - 1 X > 2^n - 1 の場合

n は <immed>、X は Rm からシフトされた値です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    if shift == 1 then
        if shift_imm == 0 then
            operand = (Rm Artihmetic_Shift_Right 32)[31:0]
        else
            operand = (Rm Artihmetic_Shift_Right shift_imm)[31:0]
    else
        operand = (Rm Logical_Shift_Left shift_imm)[31:0]
    Rd = UnsignedSat(operand, sat_imm)        /* operand treated as signed */
    if UnsignedDoesSat(operand, sat_imm) then
        Q Flag = 1
```

用法

USAT命令は、符号付きデータを符号なしデスティネーションにスケーリングおよび飽和する必要のあるピクセルカラーコンポーネントの計算など、各種の DSP アルゴリズムで使用できます。

注

R15 の使用　　レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.139 USAT16

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	1	1	1	0	sat_imm	Rd	SBO	0	0	1	1	Rm

USAT16 は、2 つの符号付き 16 ビット値を符号なしの範囲に飽和させます。飽和するビット位置は選択できます。いずれかのハーフワード演算が飽和すると、Q フラグがセットされます。

構文

```
USAT16{<cond>}  <Rd>, #<immed>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<immed> 飽和のビット位置を指定します。このビット位置は 0 ～ 15 の間で指定します。この値は命令の sat_imm フィールドにエンコードされます。

<Rm> 飽和する符号付き値を含むレジスタ。

戻り値

Rd の上位半分と下位半分に戻される値は、それぞれ以下のとおりです。

0 $X < 0$ の場合

X $0 <= X < 2^n$ の場合

2^n - 1 $X > 2^n - 1$ の場合

n は <immed>、X は Rm のいずれかの半分の値です。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rm[15:0],  sat_imm) // Rm[15:0]  treated as signed
    Rd[31:16] = UnsignedSat(Rm[31:16], sat_imm) // Rm[31:16] treated as signed
    if UnsignedDoesSat(Rm[15:0], sat_imm)
                                  OR UnsignedDoesSat(Rm[31:16], sat_imm) then
        Q Flag = 1
```

用法

USAT16 命令は、符号付きデータを符号なしデスティネーションに飽和させる必要のあるピクセルカラーコンポーネントの計算など、各種の DSP アルゴリズムで使用できます。

注

R15 の使用　　レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.140 USUB16

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 1 0 1	Rn	Rd	SBO	0 1 1 1	Rm

USUB16（符号なし減算）は、16 ビット符号なし整数の減算を 2 つ実行します。減算の桁下がりビットとして、CPSR の GE ビットをセットします。

構文

```
USUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[15:0]  = Rn[15:0] - Rm[15:0]
    GE[1:0]   = if BorrowFrom(Rn[15:0] - Rm[15:0]) then 0 else 0b11
    Rd[31:16] = Rn[31:16] - Rm[31:16]
    GE[3:2]   = if BorrowFrom(Rn[31:16] - Rm[31:16]) then 0 else 0b11
```

用法

USUB16 命令は、SSUB16 命令（P. A4-181 *「SSUB16」* 参照）と同じ演算結果を算出しますが、GE ビットは符号付き演算ではなく符号なし演算に基づいてセットされます。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.141 USUB8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	0	1	Rn	Rd	SBO	1	1	1	1	Rm

USUB8 命令は、8 ビット符号なし整数の減算を 4 つ実行します。減算の桁下がりビットとして、CPSR の GE ビットをセットします。

構文

```
USUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd[7:0]    = Rn[7:0] - Rm[7:0]
    GE[0]      = NOT BorrowFrom(Rn[7:0] - Rm[7:0])
    Rd[15:8]   = Rn[15:8] - Rm[15:8]
    GE[1]      = NOT BorrowFrom(Rn[15:8] - Rm[15:8])
    Rd[23:16] = Rn[23:16] - Rm[23:16]
    GE[2]      = NOT BorrowFrom(Rn[23:16] - Rm[23:16])
    Rd[31:24] = Rn[31:24] - Rm[31:24]
    GE[3]      = NOT BorrowFrom(Rn[31:24] - Rm[31:24])
```

用法

USUB8 命令は、SSUB8 命令（P. A4-183 *「SSUB8」* 参照）と同じ演算結果を算出しますが、GE ビットは符号付き演算ではなく符号なし演算に基づいてセットされます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 `<Rd>`、`<Rm>`、`<Rn>` のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.142 USUBADDX

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	1	1	0	0	1	0	1	Rn	Rd	SBO	0	1	0	1	Rm

USUBADDX（交換符号なし減算 / 加算）は、16 ビット符号なし整数の減算と 16 ビット符号なし整数の加算を 1 つづつ実行します。

演算を実行する前に、2 番目のオペランドにある 2 つのハーフワードの交換を行います。

桁下がりビットとキャリービットとして、CPSR の GE ビットをセットします。

構文

```
USUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    diff      = Rn[31:16] - Rm[15:0]    /* unsigned subtraction */
    Rd[31:16] = diff[15:0]
    GE[3:2]   = if BorrowFrom(Rn[31:16] - Rm[15:0]) then 0b11 else 0
    sum       = Rn[15:0] + Rm[31:16]    /* unsigned addition */
    Rd[15:0]  = sum[15:0]
    GE[1:0]   = if CarryFrom16(Rn[15:0] + Rm[31:16]) then 0b11 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

USUBADDX 命令は、SSUBADDX 命令（P. A4-185「*SSUBADDX*」を参照）と同じ演算結果を算出しますが、GE ビットは符号付き演算ではなく符号なし演算に基づいてセットされます。

注

R15 の使用 <Rd>、<Rm>、<Rn> のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.143 UXTAB

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–10	9–8	7	6	5	4	3–0
cond	0	1	1	0	1	1	1	0	Rn	Rd	rotate	SBZ	0	1	1	1	Rm

UXTAB は、レジスタから 8 ビット値を抽出し、値を 32 ビットにゼロ拡張して、結果を他のレジスタの値に加算します。8 ビットの値を抽出する前に、0、8、16、24 ビットのローテートを指定できます。

構文

```
UXTAB{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

<rotation> 以下のいずれかが使用できます。

- ROR #8.rotate フィールドで 0b01 としてエンコードされます。
- ROR #16.rotate フィールドで 0b10 としてエンコードされます。
- ROR #24.rotate フィールドで 0b11 としてエンコードされます。
- 省略。rotate フィールドで 0b00 としてエンコードされます。

—— 注 ——

アセンブラがシフト値として #0 を許可し、シフトなしの場合や LSL #0 と同じように扱う場合、ここで ROR #0 も許可する必要があります。これは、<rotation> の省略と同じです。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    operand2  = (Rm Rotate_Right(8 * rotate)) AND 0x000000ff
    Rd = Rn + operand2
```

用法

UXTAB 命令を使用すると、C/C++ の **unsigned char** 値に作用する多数の命令シーケンスに組み込まれた個々の符号拡張命令が必要なくなります。

注

R15 の使用　　レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

—— 注 ——

<Rn>レジスタにR15を使用すると、アセンブラによってエラーが生成されます。

——————

エンコード　　命令の <Rn> フィールドの値が 0b1111 である場合、この命令は UXTB 命令 （P. A4-281 「*UXTB*」参照）です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.144 UXTAB16

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–10	9–8	7	6	5	4	3–0
cond	0	1	1	0	1	1	0	0	Rn	Rd	rotate	SBZ	0	1	1	1	Rm

UXTAB16 は、レジスタから 2 つの 8 ビット値を抽出し、各値を 16 ビットにゼロ拡張して、結果を他のレジスタの 2 つの値に加算します。8 ビットの値を抽出する前に、0、8、16、24 ビットのローテートを指定できます。

構文

```
UXTAB16{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2 番目のオペランドを含むレジスタ。

<rotation> 以下のいずれかが使用できます。

- ROR #8.rotate フィールドで 0b01 としてエンコードされます。
- ROR #16.rotate フィールドで 0b10 としてエンコードされます。
- ROR #24.rotate フィールドで 0b11 としてエンコードされます。
- 省略。rotate フィールドで 0b00 としてエンコードされます。

—— 注 ——

アセンブラがシフト値として #0 を許可し、シフトなしの場合や LSL #0 と同じように扱う場合、ここで ROR #0 も許可する必要があります。これは、<rotation> の省略と同じです。

————

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    operand2  = (Rm Rotate_Right(8 * rotate)) AND 0x00ff00ff
    Rd[15:0]  = Rn[15:0]  + operand2[15:0]
    Rd[31:16] = Rn[31:16] + operand2[23:16]
```

用法

UXTAB16 命令をは、符号なしのバイト値の配列を操作するとき、イミディエート値を高精度に保つ必要がある場合に使用します。

注

R15 の使用 レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

——— 注 ———

<Rn>レジスタにR15を使用すると、アセンブラによってエラーが生成されます。

———————

エンコード 命令の <Rn> フィールドの値が 0b1111 である場合、この命令は UXTB16 命令（P. A4-283「*UXTB16*」参照）です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.145 UXTAH

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–10	9–8	7	6	5	4	3–0
cond	0	1	1	0	1	1	1	1	Rn	Rd	rotate	SBZ	0	1	1	1	Rm

UXTAHは、レジスタから16ビット値を抽出し、32ビットにゼロ拡張して、結果を他のレジスタの値に加算します。16ビットの値を抽出する前に、0、8、16、24ビットのローテートを指定できます。

構文

```
UXTAH{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件はP. A3-3*「条件フィールド」*で定義されています。<cond>が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rn> 最初のオペランドを含むレジスタ。

<Rm> 2番目のオペランドを含むレジスタ。

<rotation> 以下のいずれかが使用できます。

- ROR #8.rotateフィールドで0b01としてエンコードされます。
- ROR #16.rotateフィールドで0b10としてエンコードされます。
- ROR #24.rotateフィールドで0b11としてエンコードされます。
- 省略。rotateフィールドで0b00としてエンコードされます。

——— 注 ———

アセンブラがシフト値として#0を許可し、シフトなしの場合やLSL #0と同じように扱う場合、ここでROR #0も許可する必要があります。これは、<rotation>の省略と同じです。

———————

アーキテクチャのバージョン

ARMv6以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    operand2 = (Rm Rotate_Right(8 * rotate)) AND 0x0000ffff
    Rd       = Rn + operand2
```

用法

UXTAH命令を使用すると、C/C++の **unsigned short** 値に作用する多数の命令シーケンスに組み込まれた個々の符号拡張命令が必要なくなります。

注

R15 の使用 レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

—— 注 ——

<Rn>レジスタにR15を使用すると、アセンブラによってエラーが生成されます。

——————

エンコード 命令の <Rn> フィールドの値が 0b1111 である場合、この命令は UXTH 命令（P. A4-285「*UXTH*」参照）です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.146 UXTB

31 28	27 26 25 24 23 22 21 20	19 18 17 16	15 12	11 10	9 8	7 6 5 4	3 0
cond	0 1 1 0 1 1 1 0	1 1 1 1	Rd	rotate	SBZ	0 1 1 1	Rm

UXTB 命令はレジスタから 8 ビット値を抽出して、32 ビットにゼロ拡張します。8 ビットの値を抽出する前に、0、8、16、24 ビットのローテートを指定できます。

構文

```
UXTB{<cond>}  <Rd>, <Rm>{, <rotation>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> オペランドレジスタを指定します。

<rotation> 以下のいずれかが使用できます。

- ROR #8.rotate フィールドで 0b01 としてエンコードされます。
- ROR #16.rotate フィールドで 0b10 としてエンコードされます。
- ROR #24.rotate フィールドで 0b11 としてエンコードされます。
- 省略。rotate フィールドで 0b00 としてエンコードされます。

——— 注 ———

アセンブラがシフト値として #0 を許可し、シフトなしの場合や LSL #0 と同じように扱う場合、ここで ROR #0 も許可する必要があります。これは、<rotation> の省略と同じです。

———

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    Rd[31:0] = (Rm Rotate_Right(8 * rotate)) AND 0x000000ff
```

用法

UXTBを使用して、たとえばC/C++の**unsigned char**値に作用する命令シーケンスでバイトをワードにゼロ拡張します。

注

R15 の使用　　レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.147 UXTB16

31 28	27 26 25 24 23 22 21 20	19 18 17 16	15 12	11 10	9 8	7 6 5 4	3 0
cond	0 1 1 0 1 1 0 0	1 1 1 1	Rd	rotate	SBZ	0 1 1 1	Rm

UXTB16 は、レジスタから 2 つの 8 ビット値を抽出して、16 ビットにゼロ拡張します。8 ビットの値を抽出する前に、0、8、16、24 ビットのローテートを指定できます。

構文

```
UXTB16{<cond>}  <Rd>, <Rm>{, <rotation>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> オペランドレジスタを指定します。

<rotation> 以下のいずれかが使用できます。

- ROR #8.rotate フィールドで 0b01 としてエンコードされます。
- ROR #16.rotate フィールドで 0b10 としてエンコードされます。
- ROR #24.rotate フィールドで 0b11 としてエンコードされます。
- 省略。rotate フィールドで 0b00 としてエンコードされます。

——— 注 ———

アセンブラがシフト値として #0 を許可し、シフトなしの場合や LSL #0 と同じように扱う場合、ここで ROR #0 も許可する必要があります。これは、<rotation> の省略と同じです。

———————

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    Rd[31:0] = (Rm Rotate_Right(8 * rotate)) AND 0x00ff00ff
```

用法

UXTB16を使用して、たとえばC/C++の**unsigned char**値に作用する命令シーケンスでバイトをハーフワードにゼロ拡張します。

注

R15 の使用 レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.1.148 UXTH

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11–10	9–8	7	6	5	4	3–0
cond	0	1	1	0	1	1	1	1	1	1	1	1	Rd	rotate	SBZ	0	1	1	1	Rm

UXTH 命令はレジスタから 16 ビット値を抽出して、32 ビットにゼロ拡張します。16 ビットの値を抽出する前に、0、8、16、24 ビットのローテートを指定できます。

構文

```
UXTH{<cond>}  <Rd>, <Rm>{, <rotation>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションレジスタ。

<Rm> オペランドレジスタを指定します。

<rotation> 以下のいずれかが使用できます。

- ROR #8.rotate フィールドで 0b01 としてエンコードされます。
- ROR #16.rotate フィールドで 0b10 としてエンコードされます。
- ROR #24.rotate フィールドで 0b11 としてエンコードされます。
- 省略。rotate フィールドで 0b00 としてエンコードされます。

——— 注 ———

アセンブラがシフト値として #0 を許可し、シフトなしの場合や LSL #0 と同じように扱う場合、ここで ROR #0 も許可する必要があります。これは、<rotation> の省略と同じです。

———

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if ConditionPassed(cond) then
    Rd[31:0] = (Rm Rotate_Right(8 * rotate)) AND 0x0000ffff
```

用法

UXTH はハーフワードからワードへのゼロ拡張、たとえば C/C++ で **`unsigned short`** 値を操作する命令シーケンスで使用します。

備考

R15 の使用 レジスタ <Rd> または <Rm> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A4.2 ARM 命令とアーキテクチャのバージョン

現行の ARM アーキテクチャの各バージョンについて、使用できる ARM 命令の一覧を表 A4-2 に示します。

表 A4-2 アーキテクチャのバージョン別の ARM 命令

命令	v4	v4T	v5T	v5TE、v5TEJ、v5TExP	v6
ADC	あり	あり	あり	あり	あり
ADD	あり	あり	あり	あり	あり
AND	あり	あり	あり	あり	あり
B	あり	あり	あり	あり	あり
BIC	あり	あり	あり	あり	あり
BKPT	なし	なし	あり	あり	あり
BL	あり	あり	あり	あり	あり
BLX（両方の形式）	なし	なし	あり	あり	あり
BX	なし	あり	あり	あり	あり
BXJ	なし	なし	なし	v5TEJ のみ	あり
CDP	あり	あり	あり	あり	あり
CDP2	なし	なし	あり	あり	あり
CLZ	なし	なし	あり	あり	あり
CMN	あり	あり	あり	あり	あり
CMP	あり	あり	あり	あり	あり
CPS	なし	なし	なし	なし	あり
CPY	なし	なし	なし	なし	あり
EOR	あり	あり	あり	あり	あり
LDC	あり	あり	あり	あり	あり
LDC2	なし	なし	あり	あり	あり
LDM（すべての形式）	あり	あり	あり	あり	あり
LDR	あり	あり	あり	あり	あり

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 A4-2 アーキテクチャのバージョン別の ARM 命令（続き）

命令	v4	v4T	v5T	v5TE、v5TEJ、v5TExP	v6
LDRB	あり	あり	あり	あり	あり
LDRD	なし	なし	なし	v5TE、v5TEJ のみ	あり
LDRBT	あり	あり	あり	あり	あり
LDREX	なし	なし	なし	なし	あり
LDRH	あり	あり	あり	あり	あり
LDRSB	あり	あり	あり	あり	あり
LDRSH	あり	あり	あり	あり	あり
LDRT	あり	あり	あり	あり	あり
MCR	あり	あり	あり	あり	あり
MCR2	なし	なし	あり	あり	あり
MCRR	なし	なし	なし	v5TE、v5TEJ のみ	あり
MCRR2	なし	なし	なし	なし	あり
MLA	あり	あり	あり	あり	あり
MOV	あり	あり	あり	あり	あり
MRC	あり	あり	あり	あり	あり
MRC2	なし	なし	あり	あり	あり
MRRC	なし	なし	なし	v5TE、v5TEJ のみ	あり
MRRC2	なし	なし	なし	なし	あり
MRS	あり	あり	あり	あり	あり
MSR	あり	あり	あり	あり	あり
MUL	あり	あり	あり	あり	あり
MVN	あり	あり	あり	あり	あり
ORR	あり	あり	あり	あり	あり
PKH（両方の形式）	なし	なし	なし	なし	あり

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 A4-2 アーキテクチャのバージョン別の ARM 命令（続き）

命令	v4	v4T	v5T	v5TE、v5TEJ、v5TExP	v6
PLD	なし	なし	なし	v5TE、v5TEJ のみ	あり
QADD	なし	なし	なし	あり	あり
QADD16	なし	なし	なし	なし	あり
QADD8	なし	なし	なし	なし	あり
QADDSUBX	なし	なし	なし	なし	あり
QDADD	なし	なし	なし	あり	あり
QDSUB	なし	なし	なし	あり	あり
QSUB	なし	なし	なし	あり	あり
QSUB16	なし	なし	なし	なし	あり
QSUB8	なし	なし	なし	なし	あり
QSUBADDX	なし	なし	なし	なし	あり
REV（すべての形式）	なし	なし	なし	なし	あり
RFE	なし	なし	なし	なし	あり
RSB	あり	あり	あり	あり	あり
RSC	あり	あり	あり	あり	あり
SADD（すべての形式）	なし	なし	なし	なし	あり
SBC	あり	あり	あり	あり	あり
SEL	なし	なし	なし	なし	あり
SETEND	なし	なし	なし	なし	あり
SHADD（すべての形式）	なし	なし	なし	なし	あり
SHSUB（すべての形式）	なし	なし	なし	なし	あり
SMLAD	なし	なし	なし	なし	あり
SMLAL	あり	あり	あり	あり	あり
SMLALD	なし	なし	なし	なし	あり

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 A4-2 アーキテクチャのバージョン別の ARM 命令（続き）

命令	v4	v4T	v5T	v5TE、v5TEJ、v5TExP	v6
SMLA<x><y>	なし	なし	なし	あり	あり
SMLAL<x><y>	なし	なし	なし	あり	あり
SMLAW<y>	なし	なし	なし	あり	あり
SMLSD	なし	なし	なし	なし	あり
SMLSLD	なし	なし	なし	なし	あり
SMMLA	なし	なし	なし	なし	あり
SMMLS	なし	なし	なし	なし	あり
SMMUL	なし	なし	なし	なし	あり
SMUAD	なし	なし	なし	なし	あり
SMULL	あり	あり	あり	あり	あり
SMUL<x><y>	なし	なし	なし	あり	あり
SMULW<y>	なし	なし	なし	あり	あり
SMUSD	なし	なし	なし	なし	あり
SRS	なし	なし	なし	なし	あり
SSAT（両方の形式）	なし	なし	なし	なし	あり
SSUB（すべての形式）	なし	なし	なし	なし	あり
STC	あり	あり	あり	あり	あり
STC2	なし	なし	あり	あり	あり
STM（両方の形式）	あり	あり	あり	あり	あり
STR	あり	あり	あり	あり	あり
STRB	あり	あり	あり	あり	あり
STRBT	あり	あり	あり	あり	あり
STRD	なし	なし	なし	v5TE、v5TEJ のみ	あり
STREX	なし	なし	なし	なし	あり

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 A4-2 アーキテクチャのバージョン別の ARM 命令（続き）

命令	v4	v4T	v5T	v5TE、v5TEJ、v5TExP	v6
STRH	あり	あり	あり	あり	あり
STRT	あり	あり	あり	あり	あり
SUB	あり	あり	あり	あり	あり
SWI	あり	あり	あり	あり	あり
SWP	あり	あり	あり	あり	非推奨
SWPB	あり	あり	あり	あり	非推奨
SXT（すべての形式）	なし	なし	なし	なし	あり
TEQ	あり	あり	あり	あり	あり
TST	あり	あり	あり	あり	あり
UADD（すべての形式）	なし	なし	なし	なし	あり
UHADD（すべての形式）	なし	なし	なし	なし	あり
UMAAL	なし	なし	なし	なし	あり
UMLAL	あり	あり	あり	あり	あり
UMULL	あり	あり	あり	あり	あり
UQADD（すべての形式）	なし	なし	なし	なし	あり
UQSUB（すべての形式）	なし	なし	なし	なし	あり
USAD（両方の形式）	なし	なし	なし	なし	あり
USAT（両方の形式）	なし	なし	なし	なし	あり
USUB（すべての形式）	なし	なし	なし	なし	あり
UXT（すべての形式）	なし	なし	なし	なし	あり

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 A5 章
ARM アドレシングモード

本章では、ARM® 命令で使用される 5 つのアドレシングモード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。

- *アドレシングモード 1 - データ処理オペランド* : P. A5-2
- *アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア* : P. A5-18
- *アドレシングモード 3 - その他のロードとストア* : P. A5-33
- *アドレシングモード 4 - 複数ロード / ストア* : P. A5-41
- *アドレシングモード 5 - コプロセッサのロード / ストア* : P. A5-49

——— 注 ———

すべての有効なアーキテクチャバリエーション（バージョン 4 以降、P. xiii *「アーキテクチャのバージョンとバリエーション」*参照）はアドレシングモード 1 から 5 までをサポートしています。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1 アドレシングモード 1 - データ処理オペランド

ARM データ処理命令の `<shifter_operand>` の計算に使用される形式は 11 タイプあります。一般的な構文は以下の通りです。

```
<opcode>{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
```

`<shifter_operand>` には次のいずれかを指定します。

1. `#<immediate>`

   詳細については、P. A5-6「*データ処理オペランド - イミディエート*」を参照して下さい。

2. `<Rm>`

   詳細については、P. A5-8「*データ処理オペランド - レジスタ*」を参照して下さい。

3. `<Rm>, LSL #<shift_imm>`

   詳細については、P. A5-9「*データ処理オペランド - イミディエートによる論理左シフト*」を参照して下さい。

4. `<Rm>, LSL <Rs>`

   詳細については、P. A5-10「*データ処理オペランド - レジスタによる論理左シフト*」を参照して下さい。

5. `<Rm>, LSR #<shift_imm>`

   詳細については、P. A5-11「*データ処理オペランド - イミディエートによる論理右シフト*」を参照して下さい。

6. `<Rm>, LSR <Rs>`

   詳細については、P. A5-12「*データ処理オペランド - レジスタによる論理右シフト*」を参照して下さい。

7. `<Rm>, ASR #<shift_imm>`

   詳細については、P. A5-13「*データ処理オペランド - イミディエートによる算術右シフト*」を参照して下さい。

8. `<Rm>, ASR <Rs>`

   詳細については、P. A5-14「*データ処理オペランド - レジスタによる算術右シフト*」を参照して下さい。

9. `<Rm>, ROR #<shift_imm>`

   詳細については、P. A5-15「*データ処理オペランド - イミディエートによる右ローテート*」を参照して下さい。

10. `<Rm>, ROR <Rs>`

    詳細については、P. A5-16「*データ処理オペランド - レジスタによる右ローテート*」を参照して下さい。

11. `<Rm>, RRX`

    詳細については、P. A5-17「*データ処理オペランド - 拡張付き右ローテート*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.1 エンコード

以下の図は、このアドレシングモードのエンコードを示しています。

32 ビットイミディエート

31–28	27	26	25	24–21	20	19–16	15–12	11–8	7–0
cond	0	0	1	opcode	S	Rn	Rd	rotate_imm	immed_8

イミディエートシフト

31–28	27	26	25	24–21	20	19–16	15–12	11–7	6–5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	shift	0	Rm

レジスタシフト

31–28	27	26	25	24–21	20	19–16	15–12	11–8	7	6–5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	shift	1	Rm

opcode 命令の動作を指定します。

S ビット 命令が条件コードを変更することを示します。

Rd デスティネーションレジスタを指定します。

Rn 最初のソースオペランドレジスタを指定します。

ビット **[11:0]** ビット [11:0] のフィールドはまとめて*シフタオペランド*と呼びます。詳細については、P. A5-4「*シフタオペランド*」を参照して下さい。

ビット **[25]** I ビットと呼ばれ、イミディエートシフタオペランドとレジスタベースのシフタオペランドを区別するために使用されます。

以下の 3 つのビットがすべて以下に示す値の場合、命令はデータ処理命令ではなく、演算またはロード / ストア命令拡張空間に属します。

```
bit[25]   == 0
bit[4]    == 1
bit[7]    == 1
```

詳細については、P. A3-32「*命令セットの拡張*」を参照して下さい。

アドレシングモード 3、`MCRR{2}`、`MRRC{2}`、`STC{2}` はこの空間に存在する命令の例です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.2 シフタオペランド

シフタはシフタオペランドを生成すると同時に、一部の命令がキャリーフラグに書き込むキャリーアウトも生成します。デフォルトのレジスタオペランド（シフトなしで指定されるレジスタ RM）は、ゼロにセットしたイミディエートによるレジスタ左シフト形式を使用します。

シフタオペランドは、以下の 3 つの基本形式を使用します。

イミディエートオペランド値

イミディエートオペランド値は、8 ビットの定数（32 ビットワード内の）を偶数ビット（0、2、4、8...26、28、30）だけローテートして作成します。そのため、各命令には 8 ビットの定数と、その定数に適用される 4 ビットのローテートが含まれています。

有効な定数の例は以下のとおりです。

```
0xFF,0x104,0xFF0,0xFF00,0xFF000,0xFF000000,0xF000000F
```

無効な定数の例は以下のとおりです。

```
0x101,0x102,0xFF1,0xFF04,0xFF003,0xFFFFFFFF,0xF000001F
```

次に例を示します。

```
MOV    R0, #0                    ; Move zero to R0
ADD    R3, R3, #1                ; Add one to the value of register 3
CMP    R7, #1000                 ; Compare value of R7 with 1000
BIC    R9, R8, #0xFF00           ; Clear bits 8-15 of R8 and store in R9
```

レジスタオペランド値

レジスタオペランド値は単純にレジスタの値のことです。レジスタ値は、データ処理命令のオペランドとして直接使用されます。次に例を示します。

```
MOV    R2, R0                    ; Move the value of R0 to R2
ADD    R4, R3, R2                ; Add R2 to R3, store result in R4
CMP    R7, R8                    ; Compare the value of R7 and R8
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

シフト付きレジスタオペランド値

シフト付きレジスタオペランド値は、レジスタの値をシフト（またはローテート）してからデータ処理オペランドとして使用することを意味します。シフトには以下の 5 つのタイプがあります。

`ASR` 算術右シフト

`LSL` 論理左シフト

`LSR` 論理右シフト

`ROR` 右ローテート

`RRX` 拡張付き右ローテート

シフトするビット数は、イミディエートまたはレジスタ値のいずれかで指定します。次に例を示します。

```
MOV    R2,  R0, LSL  #2           ; Shift R0 left by 2, write to R2, (R2=R0x4)
ADD    R9,  R5, R5,  LSL #3       ; R9 = R5 + R5 x 8 or R9 = R5 x 9
RSB    R9,  R5, R5,  LSL #3       ; R9 = R5 x 8 - R5 or R9 = R5 x 7
SUB    R10, R9, R8,  LSR #4       ; R10 = R9 - R8 / 16
MOV    R12, R4, ROR  R3            ; R12 = R4 rotated right by value of R3
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.3 データ処理オペランド - イミディエート

31 28	27 26 25	24 21	20	19 16	15 12	11 8	7 0
cond	0 0 1	opcode	S	Rn	Rd	rotate_imm	immed_8

このデータ処理オペランドは、命令中で定義される定数オペランドをデータ処理命令に提供します。

<shifter_operand> の値は、8 ビットのイミディエート値を、32 ビットワードのうち任意の偶数ビット位置に右ローテートして生成されます。rotate_imm がゼロの場合、シフタからのキャリーアウトは C フラグの値となり、それ以外の場合は、<shifter_operand> の値のビット [31] にセットされます。

構文

```
#<immediate>
```

各項目の説明については以下を参照して下さい。

<immediate> 必要なイミディエート定数を指定します。この定数は 8 ビットのイミディエート（immed_8）と 4 ビットのイミディエート（rotate_imm）としてエンコードされ、<immediate> は immed_8 を（2 × rotate_imm）ビットだけ右ローテートした結果に等しくなります。

動作

```
shifter_operand = immed_8 Rotate_Right (rotate_imm * 2)
if rotate_imm == 0 then
    shifter_carry_out = C flag
else /* rotate_imm != 0 */
    shifter_carry_out = shifter_operand[31]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

正当なイミディエート

すべての 32 ビットイミディエートが正当ではありません。この形式で有効な 32 ビットイミディエートは、8 ビットイミディエートを偶数ビットだけ右ローテートして形成された値のみです。

エンコード `<immediate>` の値には、複数のエンコードが可能なものがあります。たとえば、`0x3F0` は以下のエンコードが可能です。

immed_8 == `0x3F`, rotate_imm == `0xE`

または

immed_8 == `0xFC`, rotate_imm == `0xF`

複数のエンコードが可能な場合、アセンブラでは以下の方法で正しいエンコードを選択する必要があります。

- `<immediate>` が 0 ～ `0xFF` の範囲にある場合、rotate_imm == 0 のエンコードを使用します。アセンブラはこのエンコードを選択する必要があります。別のエンコードを選択すると、一部の命令で C フラグをセットする方法に影響します。
- それ以外の場合、rotate_imm の値が最も小さくなるエンコードを選択することをお勧めします。この選択は、命令の機能に影響しません。

エンコードをより厳密に制御するには、次の構文を使用して命令フィールドを直接指定します。

```
#<immed_8>, <rotate_amount>
```

`<rotate_amount>` は 2 * rotate_imm です。

R15 の使用 レジスタ Rn として R15 が指定されている場合、現在の命令のアドレス + 8 が値として使用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.4 データ処理オペランド - レジスタ

31–28	27	26	25	24–21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	0	0	0	0	0	0	0	0	Rm

このデータ処理オペランドは、レジスタの値を直接使用します。シフタからのキャリーアウトは C フラグです。

構文

```
<Rm>
```

各項目の説明については以下を参照して下さい。

<Rm> このレジスタの値が命令オペランドとして使用されます。

動作

```
shifter_operand = Rm
shifter_carry_out = C Flag
```

注

エンコード この命令は、イミディエート値による論理左シフトで、シフト数がゼロ（shift_imm == 0）（P. A5-9 *「データ処理オペランド - イミディエートによる論理左シフト」*参照）であるものとしてエンコードされます。

R15 の使用 レジスタ Rm または Rn として R15 が指定されている場合、現在の命令のアドレス + 8 が値として使用されます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.5 データ処理オペランド - イミディエートによる論理左シフト

31–28	27	26	25	24–21	20	19–16	15–12	11–7	6	5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	0	0	0	Rm

このデータ処理オペランドは、レジスタ値を直接（P. A5-8 *「データ処理オペランド - レジスタ」*で説明されている単独のレジスタオペランド）使用するか、レジスタの値を左にシフト（2 の累乗倍（定数））してオペランドを算出します。

この命令オペランドは、0 ～ 31 のイミディエート値が示すビット数だけ論理左シフトしたレジスタ Rm の値です。空のビット位置にはゼロが挿入されます。シフタからのキャリーアウトは、シフトアウトされた最後のビット、またはシフトが指定されない場合は C フラグになります。

構文

```
<Rm>, LSL #<shift_imm>
```

各項目の説明については以下を参照して下さい。

`<Rm>` シフトされる値を保持するレジスタを指定します。

`LSL` 論理左シフトを指定します。

`<shift_imm>` シフトのビット数を指定します。値の範囲は 0 ～ 31 です。

動作

```
if shift_imm == 0 then /* Register Operand */
    shifter_operand = Rm
    shifter_carry_out = C Flag
else /* shift_imm > 0 */
    shifter_operand = Rm Logical_Shift_Left shift_imm
    shifter_carry_out = Rm[32 - shift_imm]
```

注

デフォルトのシフト

`<shift_imm>` の値が 0 の場合、オペランドを単に `<Rm>` と書くことができます（P. A5-8 *「データ処理オペランド - レジスタ」* 参照）。

R15 の使用 レジスタ Rm または Rn として R15 が指定されている場合、現在の命令のアドレス + 8 が値として使用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.6 データ処理オペランド - レジスタによる論理左シフト

31–28	27	26	25	24–21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	0	0	1	Rm

このデータ処理オペランドは、レジスタの値を 2 の累乗倍（変数）して値を算出します。

この命令オペランドは、レジスタ Rs の最下位バイトの値が示す数だけ論理左シフトしたレジスタ Rm の値です。空いたビット位置には 0 が挿入されます。シフタからのキャリーアウトは、シフトアウトされた最後のビットです。シフトするビット数が 32 より大きい場合は 0、ビット数が 0 の場合は C フラグになります。

構文

```
<Rm>, LSL <Rs>
```

各項目の説明については以下を参照して下さい。

`<Rm>` シフトされる値を保持するレジスタを指定します。

`LSL` 論理左シフトを指定します。

`<Rs>` シフトの値を含むレジスタを指定します。

動作

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[7:0] < 32 then
    shifter_operand = Rm Logical_Shift_Left Rs[7:0]
    shifter_carry_out = Rm[32 - Rs[7:0]]
else if Rs[7:0] == 32 then
    shifter_operand = 0
    shifter_carry_out = Rm[0]
else /* Rs[7:0] > 32 */
    shifter_operand = 0
    shifter_carry_out = 0
```

注

R15 の使用 Rd、Rm、Rn、Rsのいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.7 データ処理オペランド - イミディエートによる論理右シフト

31–28	27	26	25	24–21	20	19–16	15–12	11–7	6	5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	0	1	0	Rm

このデータ処理オペランドは、右にシフトされたレジスタの符号なしの値を算出します（2 の累乗（定数）で除算）。

この命令オペランドは、1 ～ 32 のイミディエート値が示すビット数だけ論理右シフトしたレジスタ Rm の値です。空のビット位置には 0 が挿入されます。シフタからのキャリーアウトは、シフトアウトされた最後のビットです。

構文

```
<Rm>, LSR #<shift_imm>
```

各項目の説明については以下を参照して下さい。

`<Rm>` シフトされる値を保持するレジスタを指定します。

`LSR` 論理右シフトを指定します。

`<shift_imm>` シフトのビット数を指定します。1 ～ 32 のイミディエート値を指定します（32 ビットのシフトは、shift_imm == 0 にエンコードされます）。

動作

```
if shift_imm == 0 then
    shifter_operand = 0
    shifter_carry_out = Rm[31]
else /* shift_imm > 0 */
    shifter_operand = Rm Logical_Shift_Right shift_imm
    shifter_carry_out = Rm[shift_imm - 1]
```

注

R15 の使用 レジスタ Rm または Rn として R15 が指定されている場合、現在の命令のアドレス + 8 が値として使用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.8 データ処理オペランド - レジスタによる論理右シフト

31 28	27	26	25	24 21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	0	1	1	Rm

このデータ処理オペランドは、右にシフトされたレジスタの符号なしの値を算出します (2 の累乗 (変数) で除算)。

これは、レジスタ Rs の最下位バイトの値が示す数だけ論理右シフトしたレジスタ Rm の値です。空いたビット位置には 0 が挿入されます。シフタからのキャリーアウトは、シフトアウトされた最後のビットです。シフトするビット数が 32 より大きい場合は 0、ビット数が 0 の場合は C フラグになります。

構文

```
<Rm>, LSR <Rs>
```

各項目の説明については以下を参照して下さい。

<Rm> シフトされる値を保持するレジスタを指定します。

LSR 論理右シフトを指定します。

<Rs> シフトの値を含むレジスタを指定します。

動作

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[7:0] < 32 then
    shifter_operand = Rm Logical_Shift_Right Rs[7:0]
    shifter_carry_out = Rm[Rs[7:0] - 1]
else if Rs[7:0] == 32 then
    shifter_operand = 0
    shifter_carry_out = Rm[31]
else /* Rs[7:0] > 32 */
    shifter_operand = 0
    shifter_carry_out = 0
```

注

R15 の使用 Rd、Rm、Rn、Rs のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.9 データ処理オペランド - イミディエートによる算術右シフト

31–28	27	26	25	24–21	20	19–16	15–12	11–7	6	5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	1	0	0	Rm

このデータ処理オペランドは、算術右シフトされたレジスタの符号付きの値を算出します（2 の累乗（定数）で除算）。

この命令オペランドは、1 ～ 32 のイミディエート値が示すビット数だけ算術右シフトしたレジスタ Rm の値です。空のビット位置には Rm (Rm[31]) の符号ビットが挿入されます。シフタからのキャリーアウトは、シフトアウトされた最後のビットです。

構文

```
<Rm>, ASR #<shift_imm>
```

各項目の説明については以下を参照して下さい。

`<Rm>` シフトされる値を保持するレジスタを指定します。

`ASR` 算術右シフトを指定します。

`<shift_imm>` シフトのビット数を指定します。1 ～ 32 のイミディエート値を指定します（32 ビットのシフトは、shift_imm == 0 にエンコードされます）。

動作

```
if shift_imm == 0 then
    if Rm[31] == 0 then
        shifter_operand = 0
        shifter_carry_out = Rm[31]
    else /* Rm[31] == 1 */
        shifter_operand = 0xFFFFFFFF
        shifter_carry_out = Rm[31]
else /* shift_imm > 0 */
    shifter_operand = Rm Arithmetic_Shift_Right <shift_imm>
    shifter_carry_out = Rm[shift_imm - 1]
```

注

R15 の使用 レジスタ Rm または Rn として R15 が指定されている場合、現在の命令のアドレス + 8 が値として使用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.10 データ処理オペランド - レジスタによる算術右シフト

31–28	27	26	25	24–21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	1	0	1	Rm

このデータ処理オペランドは、算術右シフトされたレジスタの符号付きの値を算出します（2 の累乗（変数）で除算）。

この命令オペランドは、レジスタ Rs の最下位バイトの値が示す数だけ算術右シフトしたレジスタ Rm の値です。空いたビット位置には、Rm の符号ビット（Rm[31]）が挿入されます。シフタからのキャリーアウトはシフトアウトされた最後のビットです。シフトするビット数が 32 を超える場合は Rm の符号ビット、ビット数がゼロの場合は C フラグになります。

構文

```
<Rm>, ASR <Rs>
```

各項目の説明については以下を参照して下さい。

`<Rm>` シフトされる値を保持するレジスタを指定します。

`ASR` 算術右シフトを指定します。

`<Rs>` シフトの値を含むレジスタを指定します。

動作

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[7:0] < 32 then
    shifter_operand = Rm Arithmetic_Shift_Right Rs[7:0]
    shifter_carry_out = Rm[Rs[7:0] - 1]
else /* Rs[7:0] >= 32 */
    if Rm[31] == 0 then
        shifter_operand = 0
        shifter_carry_out = Rm[31]
    else /* Rm[31] == 1 */
        shifter_operand = 0xFFFFFFFF
        shifter_carry_out = Rm[31]
```

注

R15 の使用 Rd、Rm、Rn、Rs のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.11 データ処理オペランド - イミディエートによる右ローテート

31 28	27 26 25	24 21	20	19 16	15 12	11 7	6 5 4	3 0
cond	0 0 0	opcode	S	Rn	Rd	shift_imm	1 1 0	Rm

このデータ処理オペランドは、定数値で示されるビット数だけローテートされたレジスタの値を算出します。

この命令オペランドは、1 ～ 31 のイミディエート値が示すビット数だけ右ローテートしたレジスタ Rm の値です。レジスタの右端からはみ出したビットは、左端の空のビット位置に挿入されます。シフタからのキャリーアウトは、右端からはみ出した最後のビットです。

構文

```
<Rm>, ROR #<shift_imm>
```

各項目の説明については以下を参照して下さい。

`<Rm>` 値がローテートされるレジスタを指定します。

`ROR` 右ローテートを指定します。

`<shift_imm>` ローテートするビット数を指定します。イミディエート値で、範囲は 1 ～ 31 です。shift_imm == 0 の場合、RRX 命令（拡張付き右ローテート）が実行されます。詳細については、P. A5-17 *「データ処理オペランド - 拡張付き右ローテート」*を参照して下さい。

動作

```
if shift_imm == 0 then
    See ìData-processing operands - Rotate right with extendî on page A5-17
else /* shift_imm > 0 */
    shifter_operand = Rm Rotate_Right shift_imm
    shifter_carry_out = Rm[shift_imm - 1]
```

注

R15 の使用 レジスタ Rm または Rn として R15 が指定されている場合、現在の命令のアドレス + 8 が値として使用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.12 データ処理オペランド - レジスタによる右ローテート

31–28	27	26	25	24–21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	1	1	1	Rm

このデータ処理オペランドは、変数値で示されるビット数だけローテートされたレジスタの値を算出します。

この命令オペランドは、レジスタ Rs の最下位バイトの値が示す数だけ右ローテートされたレジスタ Rm の値です。ローテートによってレジスタの右端からはみ出したビットは、左端の空いたビット位置に挿入されます。シフタからのキャリーアウトは右端からはみ出した最後のビット、またはシフト量がゼロの場合は C フラグになります。

構文

```
<Rm>, ROR <Rs>
```

各項目の説明については以下を参照して下さい。

<Rm> 値がローテートされるレジスタを指定します。

ROR 右ローテートを指定します。

<Rs> ローテートの値を含むレジスタを指定します。

動作

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[4:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = Rm[31]
else /* Rs[4:0] > 0 */
    shifter_operand = Rm Rotate_Right Rs[4:0]
    shifter_carry_out = Rm[Rs[4:0] - 1]
```

注

R15 の使用 Rd、Rm、Rn、Rs のいずれかのレジスタとして R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.1.13 データ処理オペランド - 拡張付き右ローテート

31–28	27	26	25	24–21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	0	0	0	0	0	1	1	0	Rm

このデータ処理オペランドは、キャリーフラグを 33 番目のビットとして使用して、33 ビットの右ローテートを実行します。

この命令オペランドは、空のビット位置をキャリーフラグに置き換え、1 ビットだけ右シフトされたレジスタ Rm の値です。シフタからのキャリーアウトは、右端からはみ出した最後のビットです。

構文

```
<Rm>, RRX
```

各項目の説明については以下を参照して下さい。

<Rm> 1 ビット右シフトされる値を保持するレジスタを指定します。

RRX 拡張付き右ローテートを示します。

動作

```
shifter_operand = (C Flag Logical_Shift_Left 31) OR (Rm Logical_Shift_Right 1)
shifter_carry_out = Rm[0]
```

注

エンコード 命令のエンコードは、ROR #0 に使用される空間にあります。

R15 の使用 レジスタ Rm または Rn として R15 が指定されている場合、現在の命令のアドレス + 8 が値として使用されます。

ADC 命令 拡張付き左ローテートは、ADC 命令で実行できます。

```
ADC <Rd>, <Rm>
```

ここで、<Rn> == <Rm> の場合は修飾されたオペランドが結果と等しくなります。また

```
ADC <Rd>, <Rn>, <Rm>, LSL #1
```

この場合、拡張付き左ローテートは結果ではなく第 2 オペランドです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.2 アドレシングモード 2 - ワードまたは符号なしバイトのロード / ストア

ワードまたは符号なしバイトのロード / ストア命令のアドレスの計算に使用される形式は 9 タイプあります。一般的な構文は以下の通りです。

```
LDR|STR{<cond>}{B}{T}  <Rd>, <addressing_mode>
```

`<addressing_mode>` は以下の一覧にある 9 つのオプションのうちの 1 つです。

以下の 9 つのオプションはすべて、LDR、LDRB、STR、STRB に使用できます。LDRBT、LDRT、STRBT、STRBT については、*ポストインデクス*オプション（一覧の最後の 3 つ）のみ使用できます。P. A4-91 *「PLD」*の説明にある PLD 命令では、*オフセット*オプション（一覧の最初の 3 つ）のみ使用できます。

1. `[<Rn>, #+/-<offset_12>]`

   詳細については、P. A5-20 *「ワードまたは符号なしバイトのロード / ストア - イミディエートオフセット」*を参照して下さい。

2. `[<Rn>, +/-<Rm>]`

   詳細については、P. A5-21 *「ワードまたは符号なしバイトのロード / ストア - レジスタオフセット」*を参照して下さい。

3. `[<Rn>, +/-<Rm>, <shift> #<shift_imm>]`

   詳細については、P. A5-22 *「ワードまたは符号なしバイトのロード / ストア - スケーリング付きレジスタオフセット」*を参照して下さい。

4. `[<Rn>, #+/-<offset_12>]!`

   詳細については、P. A5-24 *「ワードまたは符号なしバイトのロード / ストア - イミディエートプリインデクス」*を参照して下さい。

5. `[<Rn>, +/-<Rm>]!`

   詳細については、P. A5-25 *「ワードまたは符号なしバイトのロード / ストア - レジスタプリインデクス」*を参照して下さい。

6. `[<Rn>, +/-<Rm>, <shift> #<shift_imm>]!`

   詳細については、P. A5-26 *「ワードまたは符号なしバイトのロード / ストア - スケーリング付きレジスタプリインデクス」*を参照して下さい。

7. `[<Rn>], #+/-<offset_12>`

   詳細については、P. A5-28 *「ワードまたは符号なしバイトのロード / ストア - イミディエートポストインデクス」*を参照して下さい。

8. `[<Rn>], +/-<Rm>`

   詳細については、P. A5-30 *「ワードまたは符号なしバイトのロード / ストア - レジスタポストインデクス」*を参照して下さい。

9. `[<Rn>], +/-<Rm>, <shift> #<shift_imm>`

   詳細については、P. A5-31 *「ワードまたは符号なしバイトのロード / ストア - スケーリング付きレジスタポストインデクス」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.2.1 エンコード

以下の 3 つの図は、このアドレシングモードのエンコードを示しています。

イミディエートオフセット / インデクス

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	1	0	P	U	B	W	L	Rn	Rd	offset_12

レジスタオフセット / インデクス

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	0	1	1	P	U	B	W	L	Rn	Rd	0	0	0	0	0	0	0	0	Rm

スケーリング付きレジスタオフセット / インデクス

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
cond	0	1	1	P	U	B	W	L	Rn	Rd	shift_imm	shift	0	Rm

P ビット　以下の 2 つの意味があります。

P == 0　*ポストインデクスアドレシング*の使用を指定します。ベースレジスタ値はメモリアドレスに使用された後、オフセットがベースレジスタ値に適用され、ベースレジスタに書き戻されます。

P == 1　*オフセットアドレシング*または*プリインデクスアドレシング*の使用を指定します（どちらが使用されるかは W ビットによって決定されます）。メモリアドレスは、ベースレジスタの値にオフセットを適用して生成されます。

U ビット　オフセットをベースに加算するか（U == 1）ベースから減算するか（U == 0）のいずれかを示します。

B ビット　符号なしバイトアクセス（B == 1）とワードアクセス（B == 0）を区別します。

W ビット　以下の 2 つの意味があります。

P == 0　W == 0 の場合、命令は LDR、LDRB、STR、STRB のいずれかになり、通常のメモリアクセスが実行されます。W == 1 の場合、命令は LDRBT、LDRT、STRBT、STRT のいずれかになり、非特権の（ユーザモード）メモリアクセスが実行されます。

P == 1　W == 0 の場合、ベースレジスタは更新されません（オフセットアドレシング）。W == 1 の場合、計算されたメモリアドレスがベースレジスタに書き戻されます（プリインデクスアドレシング）。

L ビット　ロード（L == 1）とストア（L == 0）を区別します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.2.2 ワードまたは符号なしバイトのロード / ストア - イミディエートオフセット

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	0	1	U	B	0	L	Rn	Rd	offset_12

このアドレシングモードでは、イミディエートオフセット値をベースレジスタ Rn の値に加算または減算することによってアドレスを計算します。

構文

```
[<Rn>, #+/-<offset_12>]
```

各項目の説明については以下を参照して下さい。

<Rn> ベースアドレスを保持するレジスタを指定します。

<offset_12> Rn の値と共にアドレスを構成するイミディエートオフセットを指定します。

動作

```
if U == 1 then
    address = Rn + offset_12
else /* U == 0 */
    address = Rn - offset_12
```

用法

このアドレシングモードは、構造体（レコード）フィールドへのアクセスや、スタックフレームのパラメータとローカル変数へのアクセスに使用します。オフセットがゼロの場合、生成されるアドレスはベースレジスタ Rn の変更されていない値になります。

注

0 のオフセット

ポストインデクスアドレシングモードのみを許可する命令（LDRBT、LDRT、STRBT、STRT）以外では、構文 `[<Rn>]` は `[<Rn>, #0]` の省略形として処理されます。

B ビット このビットは、符号なしバイト（B == 1）とワード（B == 0）のアクセスを区別します。

L ビット このビットは、ロード（L == 1）とストア（L == 0）の命令を区別します。

R15 の使用 レジスタ Rn として R15 が指定されている場合、使用される値は命令のアドレス + 8 になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.2.3 ワードまたは符号なしバイトのロード / ストア - レジスタオフセット

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
cond	0	1	1	1	U	B	0	L	Rn	Rd	0	0	0	0	0	0	0	0	Rm

このアドレシングモードでは、インデクスレジスタ Rm の値をベースレジスタ Rn の値に加算または減算することによってアドレスを計算します。

構文

```
[<Rn>, +/-<Rm>]
```

各項目の説明については以下を参照して下さい。

`<Rn>` ベースアドレスを保持するレジスタを指定します。

`<Rm>` Rn に加算または減算する値を含むレジスタを指定します。

動作

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
```

用法

このアドレシングモードはポインタ + オフセット演算に使用され、バイト配列の 1 つのエレメントにアクセスします。

注

エンコード このアドレシングモードは、0 でスケーリングされた LSL のレジスタオフセットとしてエンコードされます。

B ビット このビットは、符号なしバイト（B == 1）とワード（B == 0）のアクセスを区別します。

L ビット このビットは、ロード（L == 1）とストア（L == 0）の命令を区別します。

R15 の使用 レジスタ Rn として R15 が指定されている場合、使用される値は命令のアドレス + 8 になります。レジスタ Rm として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.2.4 ワードまたは符号なしバイトのロード / ストア - スケーリング付きレジスタオフセット

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
cond	0	1	1	1	U	B	0	L	Rn	Rd	shift_imm	shift	0	Rm

これら 5 つのアドレシングモードでは、インデクスレジスタ Rm のシフトまたはローテートされた値を、ベースレジスタ Rn の値に加算または減算することによってアドレスを計算します。

構文

次のいずれかを指定します。

```
[<Rn>, +/-<Rm>, LSL #<shift_imm>]
[<Rn>, +/-<Rm>, LSR #<shift_imm>]
[<Rn>, +/-<Rm>, ASR #<shift_imm>]
[<Rn>, +/-<Rm>, ROR #<shift_imm>]
[<Rn>, +/-<Rm>, RRX]
```

各項目の説明については以下を参照して下さい。

`<Rn>` ベースアドレスを保持するレジスタを指定します。

`<Rm>` Rn に加算または減算するオフセットを含むレジスタを指定します。

`LSL` 論理左シフトを指定します。

`LSR` 論理右シフトを指定します。

`ASR` 算術右シフトを指定します。

`ROR` 右ローテートを指定します。

`RRX` 拡張付き右ローテートを指定します。

`<shift_imm>` シフトまたはローテートを指定します。

- `LSL` 0 ～ 31 の値で、shift_imm フィールドに直接エンコードされます。
- `LSR` 1 ～ 32 の値。シフト数が 32 の場合、shift_imm == 0 としてエンコードされます。他のシフト数は直接エンコードされます。
- `ASR` 1 ～ 32 の値。シフト数が 32 の場合、shift_imm == 0 としてエンコードされます。他のシフト数は直接エンコードされます。
- `ROR` 1 ～ 31 の値で、shift_imm フィールドに直接エンコードされます。shift_imm == 0のエンコードは、RRXオプションの指定に使用します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
case shift of
    0b00 /* LSL */
        index = Rm Logical_Shift_Left shift_imm
    0b01 /* LSR */
        if shift_imm == 0 then /* LSR #32 */
            index = 0
        else
            index = Rm Logical_Shift_Right shift_imm
    0b10 /* ASR */
        if shift_imm == 0 then /* ASR #32 */
            if Rm[31] == 1 then
                index = 0xFFFFFFFF
            else
                index = 0
        else
            index = Rm Arithmetic_Shift_Right shift_imm
    0b11 /* ROR or RRX */
        if shift_imm == 0 then /* RRX */
            index = (C Flag Logical_Shift_Left 31) OR
                    (Rm Logical_Shift_Right 1)
        else /* ROR */
            index = Rm Rotate_Right shift_imm
endcase
if U == 1 then
    address = Rn + index
else /* U == 0 */
    address = Rn - index
```

用法

これらのアドレシングモードは、1 バイトより大きい値の配列の 1 つのエレメントにアクセスするために使用します。

注

B ビット このビットは、符号なしバイト（B == 1）とワード（B == 0）のアクセスを区別します。

L ビット このビットは、ロード（L == 1）とストア（L == 0）の命令を区別します。

R15 の使用 レジスタ Rn として R15 が指定されている場合、使用される値は命令のアドレス + 8 になります。レジスタ Rm として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.2.5 ワードまたは符号なしバイトのロード / ストア - イミディエートプリインデクス

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-0
cond	0	1	0	1	U	B	1	L	Rn	Rd	offset_12

このアドレシングモードでは、イミディエートオフセット値をベースレジスタ Rn の値に加算または減算することによってアドレスを計算します。

命令で指定された条件が条件コードのステータスに一致する場合、計算されたアドレスがベースレジスタ Rn に書き戻されます。条件は P. A3-3 *「条件フィールド」* で定義されています。

構文

```
[<Rn>, #+/-<offset_12>]!
```

各項目の説明については以下を参照して下さい。

`<Rn>` ベースアドレスを保持するレジスタを指定します。

`<offset_12>` Rn の値と共にアドレスを構成するイミディエートオフセットを指定します。

`!` W ビットをセットします。この結果、ベースレジスタが更新されます。

動作

```
if U == 1 then
    address = Rn + offset_12
else /* if U == 0 */
    address = Rn - offset_12
if ConditionPassed(cond) then
    Rn = address
```

用法

このアドレシングモードは、ポインタ値を自動更新しながら配列をポインタアクセスするのに使用します。

注

0 のオフセット

構文 `[<Rn>]` を `[<Rn>, #0]!` の省略形として使用しないで下さい。

B ビット このビットは、符号なしバイト（B == 1）とワード（B == 0）のアクセスを区別します。

L ビット このビットは、ロード（L == 1）とストア（L == 0）の命令を区別します。

R15 の使用 レジスタ Rn として R15 を指定した場合、結果は予測不能です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.2.6 ワードまたは符号なしバイトのロード / ストア - レジスタプリインデクス

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
cond	0	1	1	1	U	B	1	L	Rn	Rd	0	0	0	0	0	0	0	0	Rm

このアドレシングモードでは、インデクスレジスタ Rm の値をベースレジスタ Rn の値に加算または減算することによってアドレスを計算します。

命令で指定された条件が条件コードのステータスに一致する場合、計算されたアドレスがベースレジスタ Rn に書き戻されます。条件は P. A3-3「*条件フィールド*」で定義されています。

構文

```
[<Rn>, +/-<Rm>]!
```

各項目の説明については以下を参照して下さい。

<Rn> ベースアドレスを保持するレジスタを指定します。

<Rm> Rn に加算または減算するオフセットを含むレジスタを指定します。

! W ビットをセットします。この結果、ベースレジスタが更新されます。

動作

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
if ConditionPassed(cond) then
    Rn = address
```

注

エンコード このアドレシングモードは、0 でスケーリングされた LSL のレジスタオフセットとしてエンコードされます。

B ビット このビットは、符号なしバイト（B == 1）とワード（B == 0）のアクセスを区別します。

L ビット このビットは、ロード（L == 1）とストア（L == 0）の命令を区別します。

R15 の使用 レジスタ Rm または Rn として R15 を指定した場合、結果は予測不能です。

オペランドの制限 ARMv6 以降ではオペランドの制限はありません。アーキテクチャの初期のバージョンでは、Rn と Rm に同じレジスタを指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.2.7 ワードまたは符号なしバイトのロード / ストア - スケーリング付きレジスタ プリインデクス

31–28	27 26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
cond	0 1	1	1	U	B	1	L	Rn	Rd	shift_imm	shift	0	Rm

これら 5 つのアドレシングモードでは、インデクスレジスタ Rm のシフトまたはローテートされた値を、ベースレジスタ Rn の値に加算または減算することによってアドレスを計算します。

命令で指定された条件件が条件コードのステータスに一致する場合、計算されたアドレスがベースレジスタ Rn に書き戻されます。条件は P. A3-3 *「条件フィールド」* で定義されています。

構文

次のいずれかを指定します。

```
[<Rn>, +/-<Rm>, LSL #<shift_imm>]!
[<Rn>, +/-<Rm>, LSR #<shift_imm>]!
[<Rn>, +/-<Rm>, ASR #<shift_imm>]!
[<Rn>, +/-<Rm>, ROR #<shift_imm>]!
[<Rn>, +/-<Rm>, RRX]!
```

各項目の説明については以下を参照して下さい。

`<Rn>` ベースアドレスを保持するレジスタを指定します。

`<Rm>` Rn に加算または減算するオフセットを含むレジスタを指定します。

`LSL` 論理左シフトを指定します。

`LSR` 論理右シフトを指定します。

`ASR` 算術右シフトを指定します。

`ROR` 右ローテートを指定します。

`RRX` 拡張付き右ローテートを指定します。

`<shift_imm>` シフトまたはローテートを指定します。

- `LSL` 0 ～ 31 の値で、shift_imm フィールドに直接エンコードされます。
- `LSR` 1 ～ 32 の値。シフト数が 32 の場合、shift_imm == 0 としてエンコードされます。他のシフト数は直接エンコードされます。
- `ASR` 1 ～ 32 の値。シフト数が 32 の場合、shift_imm == 0 としてエンコードされます。他のシフト数は直接エンコードされます。
- `ROR` 1 ～ 31 の値で、shift_imm フィールドに直接エンコードされます。shift_imm == 0 のエンコードは、RRX オプションの指定に使用します。

`!` W ビットをセットします。この結果、ベースレジスタが更新されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
case shift of
    0b00 /* LSL */
        index = Rm Logical_Shift_Left shift_imm
    0b01 /* LSR */
        if shift_imm == 0 then /* LSR #32 */
            index = 0
        else
            index = Rm Logical_Shift_Right shift_imm
    0b10 /* ASR */
        if shift_imm == 0 then /* ASR #32 */
            if Rm[31] == 1 then
                index = 0xFFFFFFFF
            else
                index = 0
        else
            index = Rm Arithmetic_Shift_Right shift_imm
    0b11 /* ROR or RRX */
        if shift_imm == 0 then /* RRX */
            index = (C Flag Logical_Shift_Left 31) OR
                    (Rm Logical_Shift_Right 1)
        else /* ROR */
            index = Rm Rotate_Right shift_imm
endcase
if U == 1 then
    address = Rn + index
else /* U == 0 */
    address = Rn - index
if ConditionPassed(cond) then
    Rn = address
```

注

B ビット このビットは、符号なしバイト（B == 1）とワード（B == 0）のアクセスを区別します。

L ビット このビットは、ロード（L == 1）とストア（L == 0）の命令を区別します。

R15 の使用 レジスタ Rm または Rn として R15 を指定した場合、結果は予測不能です。

オペランドの制限 ARMv6 以降ではオペランドの制限はありません。アーキテクチャの初期のバージョンでは、Rn と Rm に同じレジスタを指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.2.8 ワードまたは符号なしバイトのロード / ストア - イミディエートポストインデクス

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-0
cond	0	1	0	0	U	B	0	L	Rn	Rd	offset_12

このアドレシングモードでは、ベースレジスタRnの値をメモリアクセスのアドレスとして使用します。

命令で指定した条件が条件コードのステータスに一致する場合、ベースレジスタ Rn の値にイミディエートオフセットが加算または減算され、ベースレジスタ Rn に書き戻されます。条件は P. A3-3 *「条件フィールド」* で定義されています。

構文

```
[<Rn>], #+/-<offset_12>
```

各項目の説明については以下を参照して下さい。

`<Rn>` ベースアドレスを保持するレジスタを指定します。

`<offset_12>` Rn の値と共にアドレスを構成するイミディエートオフセットを指定します。

動作

```
address = Rn
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + offset_12
    else /* U == 0 */
        Rn = Rn - offset_12
```

用法

このアドレシングモードは、ポインタ値を自動更新しながら配列をポインタアクセスするのに使用します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ポストインデクスアドレシングモード

LDRBT、LDRT、STRBT、STRT は、ポストインデクスアドレシングモードのみをサポートしています。これらは、上記のビットパターンのビット [21]（W ビット）を 0 ではなく 1 に変更したビットパターンを使用します。

0 のオフセット

ポストインデクスアドレシングモードのみをサポートする命令（LDRBT、LDRT、STRBT、STRT）では、構文 `[<Rn>]` は `[<Rn>],#0` の省略形として処理されます。他の命令ではこの解釈は行われません。

B ビット　このビットは、符号なしバイト（B == 1）とワード（B == 0）のアクセスを区別します。

L ビット　このビットは、ロード（L == 1）とストア（L == 0）の命令を区別します。

R15 の使用　レジスタ Rn として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.2.9 ワードまたは符号なしバイトのロード / ストア - レジスタポストインデクス

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
cond	0	1	1	0	U	B	0	L	Rn	Rd	0	0	0	0	0	0	0	0	Rm

このアドレシングモードでは、ベースレジスタRnの値をメモリアクセスのアドレスとして使用します。

命令で指定した条件が条件コードのステータスに一致する場合、ベースレジスタ Rn の値にインデクスレジスタ Rm の値が加算または減算され、ベースレジスタ Rn に書き戻されます。条件は P. A3-3「*条件フィールド*」で定義されています。

構文

```
[<Rn>], +/-<Rm>
```

各項目の説明については以下を参照して下さい。

`<Rn>` ベースアドレスを保持するレジスタを指定します。

`<Rm>` Rn に加算または減算するオフセットを含むレジスタを指定します。

動作

```
address = Rn
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + Rm
    else /* U == 0 */
        Rn = Rn - Rm
```

注

エンコード
: このアドレシングモードは、0 でスケーリングされた LSL のレジスタオフセットとしてエンコードされます。

ポストインデクスアドレシングモード
: LDRBT、LDRT、STRBT、STRT は、ポストインデクスアドレシングモードのみをサポートしています。これらは、上記のビットパターンのビット [21]（W ビット）を 0 ではなく 1 に変更したビットパターンを使用します。

B ビット
: このビットは、符号なしバイト（B == 1）とワード（B == 0）のアクセスを区別します。

L ビット
: このビットは、ロード（L == 1）とストア（L == 0）の命令を区別します。

R15 の使用
: レジスタ Rn または Rm として R15 を指定した場合、結果は予測不能です。

オペランドの制限
: ARMv6 以降ではオペランドの制限はありません。アーキテクチャの初期のバージョンでは、Rn と Rm に同じレジスタを指定した場合、結果は予測不能です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.2.10 ワードまたは符号なしバイトのロード / ストア - スケーリング付きレジスタポストインデクス

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-7	6-5	4	3-0
cond	0	1	1	0	U	B	0	L	Rn	Rd	shift_imm	shift	0	Rm

このアドレシングモードでは、ベースレジスタRnの値をメモリアクセスのアドレスとして使用します。

命令で指定した条件が条件コードのステータスに一致する場合、ベースレジスタ Rn の値にインデクスレジスタ Rm のシフトまたはローテートされた値が加算または減算され、ベースレジスタ Rn に書き戻されます。条件は P. A3-3 *「条件フィールド」* で定義されています。

構文

次のいずれかを指定します。

```
[<Rn>], +/-<Rm>, LSL #<shift_imm>
[<Rn>], +/-<Rm>, LSR #<shift_imm>
[<Rn>], +/-<Rm>, ASR #<shift_imm>
[<Rn>], +/-<Rm>, ROR #<shift_imm>
[<Rn>], +/-<Rm>, RRX
```

各項目の説明については以下を参照して下さい。

`<Rn>` ベースアドレスを保持するレジスタを指定します。

`<Rm>` Rn に加算または減算するオフセットを含むレジスタを指定します。

`LSL` 論理左シフトを指定します。

`LSR` 論理右シフトを指定します。

`ASR` 算術右シフトを指定します。

`ROR` 右ローテートを指定します。

`RRX` 拡張付き右ローテートを指定します。

`<shift_imm>` シフトまたはローテートを指定します。

- `LSL` 0 ～ 31 の値で、shift_imm フィールドに直接エンコードされます。
- `LSR` 1 ～ 32 の値。シフト数が 32 の場合、shift_imm == 0 としてエンコードされます。他のシフト数は直接エンコードされます。
- `ASR` 1 ～ 32 の値。シフト数が 32 の場合、shift_imm == 0 としてエンコードされます。他のシフト数は直接エンコードされます。
- `ROR` 1 ～ 31 の値で、shift_imm フィールドに直接エンコードされます。shift_imm == 0のエンコードは、RRXオプションの指定に使用します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
address = Rn
case shift of
    0b00 /* LSL */
        index = Rm Logical_Shift_Left shift_imm
    0b01 /* LSR */
        if shift_imm == 0 then /* LSR #32 */
            index = 0
        else
            index = Rm Logical_Shift_Right shift_imm
    0b10 /* ASR */
        if shift_imm == 0 then /* ASR #32 */
            if Rm[31] == 1 then
                index = 0xFFFFFFFF
            else
                index = 0
        else
            index = Rm Arithmetic_Shift_Right shift_imm
    0b11 /* ROR or RRX */
        if shift_imm == 0 then /* RRX */
            index = (C Flag Logical_Shift_Left 31) OR
                    (Rm Logical_Shift_Right 1)
        else /* ROR */
            index = Rm Rotate_Right shift_imm
endcase
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + index
    else /* U == 0 */
        Rn = Rn - index
```

注

W ビット LDRBT、LDRT、STRBT、STRT は、ポストインデクスアドレシングモードのみをサポートしています。これらは、上記のビットパターンのビット [21]（W ビット）を 0 ではなく 1 に変更したビットパターンを使用します。

B ビット 符号なしバイトアクセス（B == 1）とワードアクセス（B == 0）を区別します。

L ビット ロード（L == 1）とストア（L == 0）を区別します。

R15 の使用 レジスタ Rm または Rn として R15 を指定した場合、結果は予測不能です。

オペランドの制限 ARMv6 以降ではオペランドの制限はありません。アーキテクチャの初期のバージョンでは、Rn と Rm に同じレジスタを指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.3 アドレシングモード 3 - その他のロードとストア

ハーフワードのロード / ストア（符号付きまたは符号なし）、符号付きバイトのロード、ダブルワードのロード / ストア命令のアドレスの計算に使用する形式は 6 タイプあります。一般的な構文は以下の通りです。

```
LDR|STR{<cond>}H|SH|SB|D  <Rd>, <addressing_mode>
```

<addressing_mode> は次の 6 つのオプションのうちの 1 つです。

1. [<Rn>, #+/-<offset_8>]

   詳細については、P. A5-35 *「その他のロードとストア - イミディエートオフセット」*を参照して下さい。

2. [<Rn>, +/-<Rm>]

   詳細については、P. A5-36 *「その他のロードとストア - レジスタオフセット」*を参照して下さい。

3. [<Rn>, #+/-<offset_8>]!

   詳細については、P. A5-37 *「その他のロードとストア - イミディエートプリインデクス」*を参照して下さい。

4. [<Rn>, +/-<Rm>]!

   詳細については、P. A5-38 *「その他のロードとストア - レジスタプリインデクス」*を参照して下さい。

5. [<Rn>], #+/-<offset_8>

   詳細については、P. A5-39 *「その他のロードとストア - イミディエートポストインデクス」*を参照して下さい。

6. [<Rn>], +/-<Rm>

   詳細については、P. A5-40 *「その他のロードとストア - レジスタポストインデクス」*を参照して下さい。

A5.3.1 エンコード

以下の図は、このアドレシングモードのエンコードを示しています。

イミディエートオフセット / インデクス

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
cond	0	0	0	P	U	1	W	L	Rn	Rd	immedH	1	S	H	1	ImmedL

レジスタオフセット / インデクス

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
cond	0	0	0	P	U	0	W	L	Rn	Rd	SBZ	1	S	H	1	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

P ビット　以下の 2 つの意味があります。

P == 0　*ポストインデクスアドレシング*の使用を指定します。ベースレジスタ値はメモリアドレスに使用された後、オフセットがベースレジスタ値に適用され、ベースレジスタに書き戻されます。

P == 1　*オフセットアドレシング*または*プリインデクスアドレシング*の使用を指定します（どちらが使用されるかは W ビットによって決定されます）。メモリアドレスは、ベースレジスタの値にオフセットを適用して生成されます。

U ビット　オフセットをベースに加算するか（U == 1）ベースから減算するか（U == 0）のいずれかを示します。

W ビット　以下の 2 つの意味があります。

P == 0　W ビットが 0 以外の場合、命令は予測不能な結果になります。

P == 1　W == 1 の場合、メモリアドレスがベースレジスタに書き戻されます（プリインデクスアドレシング）。W == 0 の場合、ベースレジスタは変更されません（オフセットアドレシング）。

L、S、H ビット

これらのビットを組み合わせて、符号付きまたは符号なしのロードやストア、ダブルワード、ハーフワード、バイトアクセスを指定します。

L = 0、S = 0、H = 1　ハーフワードのストア

L = 0、S = 1、H = 0　ダブルワードのロード

L = 0、S = 1、H = 1　ダブルワードのストア

L = 1、S = 0、H = 1　符号なしハーフワードのロード

L = 1、S = 1、H = 0　符号付きバイトのロード

L = 1、S = 1、H = 1　符号付きハーフワードのロード

バージョン v5TE 以前では、ビットは Load/!Store (L)、Signed/!Unsigned (S)、halfword/!Byte (H) で表現されます。

符号付きバイトとハーフワードは、符号なしの場合と同じ STRB 命令と STRH 命令でストアできるため、符号付きストア命令は個別に用意されていません。

符号なしバイト

S == 0、H == 0 の場合は符号なしバイトを示しているように見えますが、この命令はこのアドレシングモードを使用する命令ではありません。この命令は、乗算命令、SWP / SWPB 命令、LDREX / STREX 命令、演算またはロード / ストア命令の拡張空間に属する未割り当て命令のいずれかです（P. A3-32「*命令セットの拡張*」参照）。

符号なしバイトには LDRB、LDRBT、STRB、STRBT 命令でアクセスし、アドレシングモード 3 ではなく、アドレシングモード 2 を使用します。

符号付きストア

S == 1、L == 0 の場合は符号付きストア命令を示しているように見えますが、このエンコードは H ビットと共に、LDRD (H == 0) と STRD (H == 1) 命令のサポートに使用されます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.3.2 その他のロードとストア - イミディエートオフセット

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	1	U	1	0	L	Rn	Rd	immedH	1	S	H	1	immedL

このアドレシングモードでは、イミディエートオフセット値をベースレジスタ Rn の値に加算または減算することによってアドレスを計算します。

構文

```
[<Rn>, #+/-<offset_8>]
```

各項目の説明については以下を参照して下さい。

`<Rn>` ベースアドレスを保持するレジスタを指定します。

`<offset_8>` Rn の値と共にアドレスを構成するイミディエートオフセットを指定します。このオフセットは、immedH（最上位 4 ビット）および immedL（最下位 4 ビット）にエンコードされます。

動作

```
offset_8 = (immedH << 4) OR immedL
if U == 1 then
    address = Rn + offset_8
else /* U == 0 */
    address = Rn - offset_8
```

用法

このアドレシングモードは、構造体（レコード）フィールドへのアクセスや、スタックフレームのパラメータとローカル変数へのアクセスに使用します。オフセットがゼロの場合、生成されるアドレスはベースレジスタ Rn の変更されていない値になります。

注

0 のオフセット 構文 `[<Rn>]` は、`[<Rn>, #0]` の省略形として処理されます。

L、S、H ビット L、S、H ビットは、P. A5-33 *「エンコード」* で定義されています。

R15 の使用 レジスタ Rn として R15 が指定されている場合、使用される値は命令のアドレス + 8 になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.3.3 その他のロードとストア - レジスタオフセット

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
cond	0	0	0	1	U	0	0	L	Rn	Rd	SBZ	1	S	H	1	Rm

このアドレシングモードでは、インデクスレジスタ Rm の値をベースレジスタ Rn の値に加算または減算することによってアドレスを計算します。

構文

```
[<Rn>, +/-<Rm>]
```

各項目の説明については以下を参照して下さい。

<Rn> ベースアドレスを保持するレジスタを指定します。

<Rm> Rn に加算または減算するオフセットを含むレジスタを指定します。

動作

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
```

用法

このアドレシングモードは、ポインタ + オフセットの演算と、配列の 1 つのエレメントへのアクセスに使用されます。

注

L、S、H ビット L、S、H ビットは、P. A5-33 *「エンコード」* で定義されています。

符号なしバイト S == 0、H == 0 の場合は符号なしバイトを示しているように見えますが、この命令はこのアドレシングモードを使用する命令ではありません。この命令は、乗算命令、SWP / SWPB 命令か、演算またはロード / ストア命令の拡張空間に属する未割り当て命令のいずれかです（P. A3-32 *「命令セットの拡張」* 参照）。

符号なしバイトには LDRB、LDRBT、STRB、STRBT 命令でアクセスし、アドレシングモード 3 ではなく、アドレシングモード 2 を使用します。

R15 の使用 レジスタ Rn として R15 が指定されている場合、使用される値は命令のアドレス+8になります。レジスタRmとしてR15を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.3.4 その他のロードとストア - イミディエートプリインデクス

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	1	U	1	1	L	Rn	Rd	immedH	1	S	H	1	ImmedL

このアドレシングモードでは、イミディエートオフセット値をベースレジスタ Rn の値に加算または減算することによってアドレスを計算します。

命令で指定された条件が条件コードのステータスに一致する場合、計算されたアドレスがベースレジスタ Rn に書き戻されます。条件は P. A3-3 *「条件フィールド」* で定義されています。

構文

```
[<Rn>, #+/-<offset_8>]!
```

各項目の説明については以下を参照して下さい。

<Rn> ベースアドレスを保持するレジスタを指定します。

<offset_8> Rn の値と共にアドレスを構成するイミディエートオフセットを指定します。このオフセットは、immedH（最上位 4 ビット）および immedL（最下位 4 ビット）にエンコードされます。

! W ビットをセットします。この結果、ベースレジスタが更新されます。

動作

```
offset_8 = (immedH << 4) OR immedL
if U == 1 then
    address = Rn + offset_8
else /* U == 0 */
    address = Rn - offset_8
if ConditionPassed(cond) then
    Rn = address
```

用法

このアドレシングモードは、配列にポインタを使用してアクセスし、ポインタ値を自動更新するために使用します。

注

0 のオフセット 構文 [<Rn>] を、[<Rn>, #0]! の省略形として使用しないで下さい。

L、S、H ビット L、S、H ビットは、P. A5-33 *「エンコード」* で定義されています。

R15 の使用 レジスタ Rn として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.3.5 その他のロードとストア - レジスタプリインデクス

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
cond	0	0	0	1	U	0	1	L	Rn	Rd	SBZ	1	S	H	1	Rm

このアドレシングモードでは、インデクスレジスタ Rm の値をベースレジスタ Rn の値に加算または減算することによってアドレスを計算します。

命令で指定された条件が条件コードのステータスに一致する場合、計算されたアドレスがベースレジスタ Rn に書き戻されます。条件は P. A3-3 *「条件フィールド」* で定義されています。

構文

```
[<Rn>, +/-<Rm>]!
```

各項目の説明については以下を参照して下さい。

<Rn> ベースアドレスを保持するレジスタを指定します。

<Rm> Rn に加算または減算するオフセットを含むレジスタを指定します。

! W ビットをセットします。この結果、ベースレジスタが更新されます。

動作

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
if ConditionPassed(cond) then
    Rn = address
```

注

L、S、H ビット L、S、H ビットは、P. A5-33 *「エンコード」* で定義されています。

R15 の使用 レジスタ Rm または Rn として R15 を指定した場合、結果は予測不能です。

オペランドの制限 ARMv6 以降ではオペランドの制限はありません。アーキテクチャの初期のバージョンでは、Rn と Rm に同じレジスタを指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.3.6 その他のロードとストア - イミディエートポストインデクス

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	0	U	1	0	L	Rn	Rd	immedH	1	S	H	1	ImmedL

このアドレシングモードでは､ベースレジスタRnの値をメモリアクセスのアドレスとして使用します。

命令で指定した条件が条件コードのステータスに一致する場合、ベースレジスタ Rn の値にイミディエートオフセットの値が加算または減算され、ベースレジスタ Rn に書き戻されます。条件は P. A3-3 *「条件フィールド」*で定義されています。

構文

```
[<Rn>], #+/-<offset_8>
```

各項目の説明については以下を参照して下さい。

`<Rn>` ベースアドレスを保持するレジスタを指定します。

`<offset_8>` Rn の値と共にアドレスを構成するイミディエートオフセットを指定します。このオフセットは、immedH（最上位 4 ビット）および immedL（最下位 4 ビット）にエンコードされます。

動作

```
address = Rn
offset_8 = (immedH << 4) OR immedL
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + offset_8
    else /* U == 0 */
        Rn = Rn - offset_8
```

用法

このアドレシングモードは、ポインタ値を自動更新しながら配列をポインタアクセスするのに使用します。

注

0 のオフセット 構文 `[<Rn>]` を、`[<Rn>],#0` の省略形として使用しないで下さい。

L、S、H ビット L、S、H ビットは、P. A5-33 *「エンコード」*で定義されています。

R15 の使用 レジスタ Rn として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.3.7 その他のロードとストア - レジスタポストインデクス

31-28	27 26 25 24	23	22 21	20	19-16	15-12	11-8	7	6	5	4	3-0
cond	0 0 0 0	U	0 0	L	Rn	Rd	SBZ	1	S	H	1	Rm

このアドレシングモードでは、ベースレジスタRnの値をメモリアクセスのアドレスとして使用します。

命令で指定した条件が条件コードのステータスに一致する場合、ベースレジスタ Rn の値にインデクスレジスタ Rm の値が加算または減算され、ベースレジスタ Rn に書き戻されます。条件は P. A3-3 *「条件フィールド」* で定義されています。

構文

```
[<Rn>], +/-<Rm>
```

各項目の説明については以下を参照して下さい。

<Rn> ベースアドレスを保持するレジスタを指定します。

<Rm> Rn に加算または減算するオフセットを含むレジスタを指定します。

動作

```
address = Rn
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + Rm
    else /* U == 0 */
        Rn = Rn - Rm
```

注

L、S、H ビット　L、S、H ビットは、P. A5-33 *「エンコード」* で定義されています。

R15 の使用　レジスタ Rm または Rn として R15 を指定した場合、結果は予測不能です。

オペランドの制限　ARMv6 以降ではオペランドの制限はありません。アーキテクチャの初期のバージョンでは、Rn と Rm に同じレジスタを指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.4 アドレシングモード 4 - 複数ロード / ストア

複数ロード命令は、メモリから汎用レジスタのサブセット（場合によってはすべて）をロードします。複数ストア命令は、汎用レジスタのサブセット（場合によってはすべて）をメモリにストアします。

複数ロード / ストアのアドレシングモードは、連続した範囲のアドレスを生成します。最も小さな番号のレジスタが最下位メモリアドレスに、最も大きな番号のレジスタが最上位のメモリアドレスにストアされます。

一般的な構文は以下の通りです。

```
LDM|STM{<cond>}<addressing_mode> <Rn>{!}, <registers>{^}
```

`<addressing_mode>` は次の 4 つのアドレシングモードの 1 つです。

1. `IA`（ポストインクリメント）

   詳細については、P. A5-43 *「複数ロード / ストア - ポストインクリメント」*を参照して下さい。

2. `IB`（プリインクリメント）

   詳細については、P. A5-44 *「複数ロード / ストア - プリインクリメント」*を参照して下さい。

3. `DA`（ポストデクリメント）

   詳細については、P. A5-45 *「複数ロード / ストア - ポストデクリメント」*を参照して下さい。

4. `DB`（プリデクリメント）

   詳細については、P. A5-46 *「複数ロード / ストア - プリデクリメント」*を参照して下さい。

これらのアドレシングモードには別のニーモニックもあり、`LDM` と `STM` でスタックにアクセスするときに有用です。詳細については、P. A5-47 *「複数ロード / ストアアドレシングモード（別名）」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.4.1 エンコード

次の図は、このアドレシングモードのエンコードを示しています。

31–28	27	26	25	24	23	22	21	20	19–16	15–0
cond	1	0	0	P	U	S	W	L	Rn	register list

P ビット 以下の 2 つの意味があります。

P == 0 Rn でアドレスを指定されたワードが、アクセスされるメモリ位置に含まれ、その範囲の最上位（U == 0）または最下位（U == 1）の位置にあることを示します。

P == 1 Rn でアドレスを指定されたワードが、アクセスされるメモリ位置に含まれず、その範囲の最上位より 1 ワード上位（U == 0）または最下位より 1 ワード下位（U == 1）にあることを示します。

U ビット ベースレジスタから前方向（U == 1）または後方向（U == 0）に転送が行われることを示します。

S ビット PC をロードする LDM の場合、SPSR から CPSR がロードされることを S ビットは示します。PC をロードしない LDM およびすべての STM では、S ビットはプロセッサが特権モードの場合、現在のモードのレジスタではなくユーザモードのバンクレジスタが転送されることを示します。

LDM で S ビットをセットすると、ユーザモードまたはシステムモードでは予測不能な結果を招きます。

W ビット ベースレジスタが転送後に更新されることを示します。ベースレジスタは、レジスタリストに含まれるレジスタ数の 4 倍だけインクリメント（U == 1）またはデクリメント（U == 0）されます。

L ビット ロード（L == 1）とストア（L == 0）の命令を区別します。

Register list 命令の register_list フィールドは、各汎用レジスタに 1 ビットが対応しています。レジスタ 0 はビット [0] に対応し、以下同様にレジスタ 15 はビット [15]（PC）に対応しています。どのビットもセットされていない場合、結果は予測不能です。

命令の構文では、ロードまたはストアするレジスタを <registers> 一覧に、カンマで区切り、{ と } で囲んで指定します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.4.2 複数ロード / ストア - ポストインクリメント

31-28	27	26	25	24	23	22	21	20	19-16	15-0
cond	1	0	0	0	1	S	W	L	Rn	register list

このアドレシングモードは複数ロード / ストア命令で使用され、ある範囲のアドレスを生成します。

生成される最初のアドレスは <start_address> で、ベースレジスタ Rn の値です。以降のアドレスは、前のアドレスに 4 を加算して構成されます。<registers> に指定されている各レジスタについて 1 つのアドレスが生成されます。

最後に生成されるアドレスは、<end_address>です。この値は、ベースレジスタの値に<registers> で指定されたレジスタ数の 4 倍を足した値 - 4 です。

命令で指定された条件が条件コードステータスと一致し、W ビットがセットされている場合、Rn は <registers> のレジスタ数の 4 倍だけインクリメントされます。条件は P. A3-3「*条件フィールド*」で定義されています。

構文

```
IA
```

代替構文については、P. A5-47「*複数ロード / ストアアドレシングモード(別名)*」も参照して下さい。

動作

```
start_address = Rn
end_address = Rn + (Number_Of_Set_Bits_In(register_list) * 4) - 4
if ConditionPassed(cond) and W == 1 then
    Rn = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
```

注

L ビット このビットは、複数ロードと複数ストアを区別します。

S ビット PC をロードする LDM の場合、S ビットは SPSR から CPSR がロードされることを示します。PC をロードしない LDM およびすべての STM では、S ビットはプロセッサが特権モードの場合、現在のモードのレジスタではなくユーザモードのバンクレジスタが転送されることを示します。

LDM に S ビットをセットすると、ユーザモードまたはシステムモードでは予測不能な結果を招きます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.4.3 複数ロード / ストア - プリインクリメント

31–28	27	26	25	24	23	22	21	20	19–16	15–0
cond	1	0	0	1	1	S	W	L	Rn	register list

このアドレシングモードは複数ロード / ストア命令で使用され、ある範囲のアドレスを生成します。

生成される最初のアドレスは `<start_address>` で、ベースレジスタ Rn の値 + 4 です。以降のアドレスは、前のアドレスに 4 を加算して生成されます。`<registers>` に指定されている各レジスタについて 1 つのアドレスが生成されます。

最後に生成されるアドレスは、`<end_address>` です。この値は、ベースレジスタの値に `<registers>` で指定されたレジスタ数の 4 倍を足した値です。

命令で指定された条件が条件コードステータスと一致し、W ビットがセットされている場合、Rn は `<registers>` のレジスタ数の 4 倍だけインクリメントされます。条件は P. A3-3 *「条件フィールド」*で定義されています。

構文

```
IB
```

代替構文については、P. A5-47 *「複数ロード / ストアアドレシングモード(別名)」*も参照して下さい。

動作

```
start_address = Rn + 4
end_address = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
if ConditionPassed(cond) and W == 1 then
    Rn = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
```

注

L ビット このビットは、複数ロードと複数ストアを区別します。

S ビット PC をロードする LDM の場合、S ビットは SPSR から CPSR がロードされることを示します。PC をロードしない LDM およびすべての STM では、S ビットはプロセッサが特権モードの場合、現在のモードのレジスタではなくユーザモードのバンクレジスタが転送されることを示します。

LDM に S ビットをセットすると、ユーザモードまたはシステムモードでは予測不能な結果を招きます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.4.4 複数ロード / ストア - ポストデクリメント

31-28	27	26	25	24	23	22	21	20	19-16	15-0
cond	1	0	0	0	0	S	W	L	Rn	register list

このアドレシングモードは複数ロード / ストア命令で使用され、ある範囲のアドレスを生成します。

生成される最初のアドレスは `<start_address>` で、ベースレジスタの値から `<registers>` で指定されたレジスタ数の 4 倍を減算し、4 を加算した値です。以降のアドレスは、前のアドレスに 4 を加算して生成されます。`<registers>` に指定されている各レジスタについて 1 つのアドレスが生成されます。

最後に生成されるアドレスは、`<end_address>` です。これは、ベースレジスタ Rn の値です。

命令で指定された条件が条件コードステータスと一致し、W ビットがセットされている場合、Rn は `<registers>` のレジスタ数の 4 倍だけデクリメントされます。条件は P. A3-3「*条件フィールド*」で定義されています。

構文

```
DA
```

代替構文については、P. A5-47「*複数ロード/ ストアアドレシングモード(別名)*」も参照して下さい。

動作

```
start_address = Rn - (Number_Of_Set_Bits_In(register_list) * 4) + 4
end_address = Rn
if ConditionPassed(cond) and W == 1 then
    Rn = Rn - (Number_Of_Set_Bits_In(register_list) * 4)
```

注

L ビット このビットは、複数ロードと複数ストアを区別します。

S ビット PC をロードする LDM の場合、S ビットは SPSR から CPSR がロードされることを示します。PC をロードしない LDM およびすべての STM では、S ビットはプロセッサが特権モードの場合、現在のモードのレジスタではなくユーザモードのバンクレジスタが転送されることを示します。

LDM に S ビットをセットすると、ユーザモードまたはシステムモードでは予測不能な結果を招きます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.4.5 複数ロード / ストア - プリデクリメント

31–28	27	26	25	24	23	22	21	20	19–16	15–0
cond	1	0	0	1	0	S	W	L	Rn	register list

このアドレシングモードは複数ロード / ストア命令で使用され、ある範囲のアドレスを生成します。

最初に生成されるアドレスは <start_address> で、ベースレジスタの値から <registers> で指定されたレジスタ数の 4 倍を減算した値です。以降のアドレスは、前のアドレスに 4 を加算して構成されます。<registers> に指定されている各レジスタについて 1 つのアドレスが生成されます。

最後に生成されるアドレスは、<end_address> です。これは、ベースレジスタ Rn の値 - 4 です。

命令で指定された条件が条件コードステータスと一致し、W ビットがセットされている場合、Rn は <registers> のレジスタ数の 4 倍だけデクリメントされます。条件は P. A3-3「*条件フィールド*」で定義されています。

構文

```
DB
```

代替構文については、P. A5-47「*複数ロード / ストアアドレシングモード(別名)*」も参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

動作

```
start_address = Rn - (Number_Of_Set_Bits_In(register_list) * 4)
end_address = Rn - 4
if ConditionPassed(cond) and W == 1 then
    Rn = Rn - (Number_Of_Set_Bits_In(register_list) * 4)
```

注

L ビット このビットは、複数ロードと複数ストアを区別します。

S ビット PC をロードする LDM の場合、S ビットは SPSR から CPSR がロードされることを示します。PC をロードしない LDM およびすべての STM では、S ビットはプロセッサが特権モードの場合、現在のモードのレジスタではなくユーザモードのバンクレジスタが転送されることを示します。

LDM に S ビットをセットすると、ユーザモードまたはシステムモードでは予測不能な結果を招きます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.4.6 複数ロード / ストアアドレシングモード（別名）

P. A5-41*「アドレシングモード 4 - 複数ロード / ストア」*で示されている 4 つのアドレシングモードの名前（IA、IB、DA、DB）は複数ロード / ストア命令がブロックデータ転送に使用される場合に最も適しています。これは、複数ロード命令と複数ストア命令はアドレシングモードが同じ場合が多く、データがロードされたときと同じ方法でストアされるためです。

ただし、複数ロードと複数ストアがスタックのアクセスに使用される場合、データはストアに使用されたときと同じアドレシングモードでロードされません。これは、ロード（ポップ）とストア（プッシュ）の操作では逆方向にスタックを調整する必要があるためです。

スタック操作

複数ロードと複数ストアのアドレシングモードでは、スタック操作に適した代替構文を指定できます。

フルスタック	スタックポインタは、最後に使用された位置（フル）を指します。
空スタック	スタックポインタ、未使用（空）な最初の位置を指します。
下降スタック	メモリアドレスの降順（メモリの最下位方向）に増加します。
上昇スタック	メモリアドレスの昇順（メモリの最上位方向）に増加します。

2 つの属性によって、以下の 4 タイプのスタックが定義できます。

- 構文 FD: フル下降
- 構文 ED: 空下降
- 構文 FA: フル上昇
- 構文 EA: 空上昇

——— 注 ———

コプロセッサデータが配置される（または配置される可能性のある）スタックを定義する場合、FD または EA のスタックタイプを使用することをお勧めします。これは、これらのタイプのスタックでは、1 つの STC 命令でコプロセッサデータをプッシュし、1 つの LDC 命令でポップできるためです。コプロセッサが FA スタックまたは ED スタックにアクセスするには、複数の命令シーケンスが必要です。

———

P. A5-48 表 A5-1 および P. A5-48 表 A5-2 では、上記の 4 タイプのスタックと 4 タイプのアドレシングモード、および命令フォーマットの L、U、P ビットの関係について説明しています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 A5-1 は、LDM 命令での関係を示したものです。

表 A5-1 LDM アドレシングモード

非スタックアドレシングモード	スタックアドレシングモード	L ビット	P ビット	U ビット
LDMDA（ポストデクリメント）	LDMFA（フル上昇）	1	0	0
LDMIA（ポストインクリメント）	LDMFD（フル下降）	1	0	1
LDMDB（プリデクリメント）	LDMEA（空上昇）	1	1	0
LDMIB（プリインクリメント）	LDMED（空下降）	1	1	1

表 A5-2 は、STM 命令での関係を示したものです。

表 A5-2 STM アドレシングモード

非スタックアドレシングモード	スタックアドレシングモード	L ビット	P ビット	U ビット
STMDA（ポストデクリメント）	STMED（空下降）	0	0	0
STMIA（ポストインクリメント）	STMEA（空上昇）	0	0	1
STMDB（プリデクリメント）	STMFD（フル下降）	0	1	0
STMIB（プリインクリメント）	STMFA（フル上昇）	0	1	1

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.5 アドレシングモード 5 - コプロセッサのロード / ストア

コプロセッサのロード / ストア命令のアドレスの計算に使用されるアドレシングモードは 4 タイプあります。一般的な構文は以下の通りです。

```
<opcode>{<cond>}{L}  <coproc>,<CRd>,<addressing_mode>
```

`<addressing_mode>` は以下の 4 つのオプションの 1 つです。

1. `[<Rn>,#+/-<offset_8>*4]`

   詳細については、P. A5-51 *「コプロセッサのロード / ストア - イミディエートオフセット」*を参照して下さい。

2. `[<Rn>,#+/-<offset_8>*4]!`

   詳細については、P. A5-52 *「コプロセッサのロード / ストア - イミディエートプリインデクス」*を参照して下さい。

3. `[<Rn>],#+/-<offset_8>*4`

   詳細については、P. A5-53 *「コプロセッサのロード / ストア - イミディエートポストインデクス」*を参照して下さい。

4. `[<Rn>],<option>`

   詳細については、P. A5-54 *「コプロセッサのロード / ストア - インデクスなし」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.5.1 エンコード

次の図は、このアドレシングモードのエンコードを示しています。

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7–0
cond	1	1	0	P	U	N	W	L	Rn	CRd	cp_num	offset_8

P ビット 以下の 2 つの意味があります。

P == 0 *ポストインデクスアドレシング*または*インデクスなしアドレシング*の使用を指定します（どちらが使用されるかは W ビットによって決定されます)。ベースレジスタ値がメモリアドレスに使用されます。

P == 1 *オフセットアドレシング*または*プリインデクスアドレシング*の使用を指定します（どちらが使用されるかは W ビットによって決定されます)。メモリアドレスは、ベースレジスタの値にオフセットを適用して生成されます。

U ビット 以下の 2 つの意味があります。

U == 1 オフセットがベースに加算されることを示します。

U == 0 オフセットがベースから減算されることを示します。

N ビット このビットの意味はコプロセッサによって異なります。転送する値のサイズを区別するために使用することをお勧めします。

W ビット 以下の 2 つの意味があります。

W == 1 メモリアドレスがベースレジスタに書き戻されることを示します。

W == 0 ベースレジスタの値が変更されないことを示します。

また、以下の点に注意して下さい。

- P == 0 の場合、このビットはポストインデクスアドレシング（W == 1）とインデクスなしのアドレシング（W == 0）の区別に使用されます。インデクスなしのアドレシングでは、U の値が 1 以外の場合、未定義または予測不能な結果を招きます（P. A3-40*「コプロセッサ命令拡張空間」*参照)。
- P == 1 の場合、このビットはプリインデクスアドレシング（W == 1）とオフセットアドレシング（W == 0）の区別に使用されます。

L ビット ロード命令（L == 1）とストア命令（L == 0）を区別します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.5.2 コプロセッサのロード / ストア - イミディエートオフセット

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1	1	0	1	U	N	0	L	Rn	CRd	cp_num	offset_8

このアドレシングモードは、連続したアドレスのシーケンスを生成します。最初のアドレスは、ベースレジスタ Rn の値にイミディエートオフセットの値の 4 倍を加算または減算して計算されます。シーケンスの以降のアドレスは、コプロセッサから命令の終了が通知されるまで、前のアドレスを 4 ずつインクリメントして生成されます。これによって、コプロセッサはコプロセッサ定義のサイズのデータにアクセスできます。

コプロセッサでは、16 ワードを超える転送は要求できません。

構文

```
[<Rn>, #+/-<offset_8>*4]
```

各項目の説明については以下を参照して下さい。

<Rn> ベースアドレスを保持するレジスタを指定します。

<offset_8> イミディエートオフセットを指定します。この値を 4 で乗算してから Rn の値に加算または減算して、アドレスが計算されます。

動作

```
if ConditionPassed(cond) then
    if U == 1 then
        address = Rn + offset_8 * 4
    else /* U == 0 */
        address = Rn - offset_8 * 4
    start_address = address
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
    end_address = address
```

注

N ビット コプロセッサに依存します。

L ビット ロード（L == 1）とストア（L == 0）の命令を区別します。

R15 の使用 レジスタ Rn として R15 が指定されている場合、使用される値はその命令のアドレス + 8 になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.5.3 コプロセッサのロード / ストア - イミディエートプリインデクス

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7–0
cond	1	1	0	1	U	N	1	L	Rn	CRd	cp_num	offset_8

このアドレシングモードは、連続したアドレスのシーケンスを生成します。最初のアドレスは、ベースレジスタ Rn の値にイミディエートオフセットの値の 4 倍を加算または減算して計算されます。命令で指定された条件が条件コードのステータスに一致する場合、最初のアドレスがベースレジスタ Rn に書き戻されます。シーケンスの以降のアドレスは、コプロセッサから命令の終了が通知されるまで、前のアドレスを 4 ずつインクリメントして生成されます。これによって、コプロセッサはコプロセッサ定義のサイズのデータにアクセスできます。

コプロセッサでは、16 ワードを超える転送は要求できません。

構文

```
[<Rn>, #+/-<offset_8>*4]!
```

各項目の説明については以下を参照して下さい。

`<Rn>` ベースアドレスを保持するレジスタを指定します。

`<offset_8>` イミディエートオフセットを指定します。この値を 4 で乗算してから Rn の値に加算または減算して、アドレスが計算されます。

`!` W ビットをセットします。この結果、ベースレジスタが更新されます。

動作

```
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + offset_8 * 4
    else /* U == 0 */
        Rn = Rn - offset_8 * 4
    start_address = Rn
    address = start_address
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
    end_address = address
```

注

N ビット コプロセッサに依存します。

L ビット ロード（L == 1）とストア（L == 0）の命令を区別します。

R15 の使用 レジスタ Rn として R15 を指定した場合、結果は予測不能です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.5.4 コプロセッサのロード / ストア - イミディエートポストインデクス

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7–0
cond	1	1	0	0	U	N	1	L	Rn	CRd	cp_num	offset_8

このアドレシングモードは、連続したアドレスのシーケンスを生成します。最初のアドレスはベースレジスタ Rn の値です。シーケンスの以降のアドレスは、コプロセッサから命令の終了が通知されるまで、前のアドレスを 4 ずつインクリメントして生成されます。これによって、コプロセッサはコプロセッサ定義のサイズのデータにアクセスできます。

命令で指定した条件が条件コードのステータスに一致する場合、ベースレジスタ Rn の値にイミディエートオフセットの値の 4 倍を加算または減算した値が Rn に書き戻されます。

コプロセッサでは、16 ワードを超える転送は要求できません。

構文

```
[<Rn>], #+/-<offset_8>*4
```

各項目の説明については以下を参照して下さい。

<Rn> ベースアドレスを保持するレジスタを指定します。

<offset_8> イミディエートオフセットを指定します。この値を 4 で乗算してから Rn の値に加算または減算して、アドレスが計算されます。

動作

```
if ConditionPassed(cond) then
    start_address = Rn
    if U == 1 then
        Rn = Rn + offset_8 * 4
    else /* U == 0 */
        Rn = Rn - offset_8 * 4
    address = start_address
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
    end_address = address
```

注

N ビット コプロセッサに依存します。

L ビット ロード（L == 1）とストア（L == 0）の命令を区別します。

R15 の使用 レジスタ Rn として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A5.5.5 コプロセッサのロード / ストア - インデクスなし

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1 1 0	0	U	N	0	L	Rn	CRd	cp_num	option

このアドレシングモードは、連続したアドレスのシーケンスを生成します。最初のアドレスはベースレジスタ Rn の値です。シーケンスの以降のアドレスは、コプロセッサから命令の終了が通知されるまで、前のアドレスを 4 ずつインクリメントして生成されます。これによって、コプロセッサはコプロセッサ定義のサイズのデータにアクセスできます。

ベースレジスタ Rn は更新されません。そのため、命令のビット [7:0] は、アドレスの計算やベースレジスタの新しい値の計算のどちらにも ARM が使用しないので、コプロセッサに対して追加の命令オプションを指定するために使用できます。

コプロセッサでは、16 ワードを超える転送は要求できません。

構文

```
[<Rn>], <option>
```

各項目の説明については以下を参照して下さい。

<Rn> ベースアドレスを保持するレジスタを指定します。

<option> コプロセッサに対する追加の命令オプションを指定します。命令の構文中では、<option> は 0 ～ 255 の範囲の { と } で囲まれた整数で指定します。

動作

```
if ConditionPassed(cond) then
    start_address = Rn
    address = start_address
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
    end_address = address
```

注

N ビット コプロセッサに依存します。

L ビット ロード（L == 1）とストア（L == 0）の命令を区別します。

R15 の使用 レジスタ Rn として R15 が指定されている場合、使用される値は命令のアドレス + 8 になります。

U ビット ビット [23]（アップ / ダウンビット）がセットされていない場合、未定義または予測不能な結果を招きます（P. A3-40 *「コプロセッサ命令拡張空間」* 参照）。

オプションビット

このアドレシングモードでは ARM により使用されないため、コプロセッサに依存した形式で、命令の追加オプションの要求に使用できます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 A6 章
Thumb 命令セット

本章では、Thumb® 命令セットと、Thumb での ARM® プログラマモデルの使用方法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。

- *Thumb 命令セットについて* : P. A6-2
- *命令セットのエンコード* : P. A6-4
- *分岐命令* : P. A6-6
- *データ処理命令* : P. A6-8
- *レジスタロード / ストア命令* : P. A6-15
- *複数ロード / ストア命令* : P. A6-18
- *例外生成命令* : P. A6-20
- *未定義命令空間* : P. A6-21

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.1 Thumb 命令セットについて

Thumb 命令セットは、ARM 命令セットを再エンコードしたサブセットです。Thumb は、16 ビットまたはそれ以下のバス幅のメモリデータバスを使用する ARM 実装の性能を向上させ、ARM 命令セットよりも高いコード密度を可能にするように設計されています。ARM アーキテクチャの T バリエーションには、32 ビットの ARM 命令セットと 16 ビットの Thumb 命令セットの両方が組み込まれています。Thumb 命令はすべて 16 ビットでエンコードされています。Thumb のサポートは ARMv6 では必須です。

Thumb は、ARM アーキテクチャの基礎となるプログラマモデルの内容を変更しているわけではなく、機能へのアクセスを制限しているだけです。Thumb データ処理命令はすべて 32 ビット値に対して演算を行い、データアクセス命令と命令フェッチのどちらについても 32 ビットのアドレスが生成されます。

プロセッサが Thumb 命令を実行する場合、R0 ～ R7 の 8 つの汎用整数レジスタが使用可能です。これらは ARM 命令を実行する場合の R0 ～ R7 と同じ物理レジスタです。一部の Thumb 命令は、プログラムカウンタ（ARM レジスタ 15）、リンクレジスタ（ARM レジスタ 14）、スタックポインタ（ARM レジスタ 13）にもアクセスします。ARM レジスタ 8 ～ 15（*上位レジスタ*）に制限付きでアクセスを行うための命令も存在します。

R15 を読み出すと、ビット [0] はゼロで、ビット [31:1] に PC が含まれています。R15 に書き込みを行うと、ビット [0] は無視され、ビット [31:1] が PC に書き込まれます。使用される方法に応じて、PC の値は、命令のアドレス + 4、または予測不能のいずれかになります。

Thumb 命令が実行されるかどうかは、CPSR の T ビット（ビット [5]）によって決定されます。

T == 0 32 ビット命令がフェッチされ（PC が 4 インクリメントされる）、ARM 命令として実行されます。

T == 1 16 ビット命令がフェッチされ（PC が 2 インクリメントされる）、Thumb 命令として実行されます。

ARMv6 では、Thumb 命令セットに、`CPS` 命令による CPSR への制限付きアクセス機能があります。SPSR に直接アクセスする機能はありません。以前のバージョンには、CPSR に直接アクセスする機能はありません（ARM 命令セットでは、`MSR` 命令、`MRS` 命令、ARMv6 の `CPS` 命令でこの処理を実行できます）。

A6.1.1 Thumb 状態の開始

Thumb の実行は、一般に ARM `BX` 命令（状態遷移）の実行によって開始されます。この命令は、汎用レジスタに保持されているアドレスに分岐し、そのレジスタのビット [0] が 1 の場合、分岐先アドレスで Thumb 状態の実行が開始されます。分岐先レジスタのビット [0] が 0 の場合、分岐先アドレスで ARM 状態の実行が継続されます。ARMv5T 以降の場合、PC をロードする `BLX` 命令と `LDR`/`LDM` 命令を同様に使用できます。

Thumb の実行は、SPSR に T ビットをセットし、SPSR から CPSR を復元する ARM 命令（S ビットをセットし、PC をデスティネーションとするデータ処理命令、または CPSR 復元付き複数ロード命令）を実行することによっても開始できます。この方法では、プロセスで Thumb コードと ARM コードのどちらを実行しているかに関係なく、オペレーティングシステムが自動的にプロセスを再開できます。

T ビットが CPSR への書き込みによって直接変更された場合、結果は予測不能です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.1.2 例外

Thumb 実行中に例外が発生した場合、（最初の命令がハードウェアベクタにある）例外ハンドラの実行前に、プロセッサは ARM 状態に切り替わります。T ビットの状態は SPSR に保持され、例外が発生したときに ARM と Thumb のどちらの状態で実行が行われていたかにかかわらず通常の復帰命令が正しく動作するように、例外モードの LR が設定されます。表 A6-1 は、Thumb の実行時に生成された例外に対する例外モード LR の値の一覧です。

表 A6-1 例外復帰命令

例外	例外リンクレジスタの値	復帰命令
リセット	予測不能な値	-
未定義	未定義命令のアドレス + 2	`MOVS PC, R14`
SWI	SWI 命令のアドレス + 2	`MOVS PC, R14`
プリフェッチアボート	アボートされた命令フェッチのアドレス + 4	`SUBS PC, R14, #4`
データアボート	アボートを生成した命令のアドレス + 8	`SUBS PC, R14, #8`
IRQ	次に実行される命令のアドレス + 4	`SUBS PC, R14, #4`
FIQ	次に実行される命令のアドレス + 4	`SUBS PC, R14, #4`

—— 注 ——

各例外について表 6-1 に示されている復帰命令は、P. A2-16*「例外」*に示されている命令からの復帰の主要なまたは唯一のメソッドとしては、ARM の実行時に例外が発生した場合に必要な復帰命令と同じです。ただし、以下の 2 タイプの例外には補助的な復帰メソッドがあり、例外が ARM と Thumb のどちらの実行時に発生したかによって、異なる復帰命令が必要です。

- データアボート例外の場合、主要な復帰メソッドによってアボートされた命令から実行が再開され、命令が再実行されます。P. A2-21*「データアボート（データアクセス中に発生するメモリアボート）」*に説明されているように、`SUBS PC,R14,#4` 命令を使用して、アボートされた命令の次の命令に復帰することもできます。Thumb 命令で発生したデータアボートからこのタイプの復帰を行う必要がある場合、復帰命令に `SUBS PC,R14,#6` を使用します。
- 未定義命令例外の場合、主要な復帰メソッドによって、その未定義命令の次の命令から実行が再開されます。P. A2-19*「未定義命令例外」*に説明されているように、命令 `SUBS PC,R14,#4` を使用して、未定義命令自体に復帰することもできます。Thumb の未定義命令で発生した例外からこのタイプの復帰を行う必要がある場合、復帰命令に `SUBS PC,R14,#2` を使用します。ただし、このタイプの復帰は主に一部のコプロセッサ命令に対して使用するものです。Thumb 命令セットにはコプロセッサ命令はないため、この補助的な方法で Thumb 命令に復帰することは通常は必要ありません。

これらの補助的な復帰メソッドが使用される場合、例外ハンドラコードで SPSR の T ビットをテストして、2 つの復帰命令のどちらを使用するか判定する必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.2 命令セットのエンコード

図 A6-1 は、Thumb 命令セットのエンコードを示したものです。角括弧内の項目（[1] など）は、以降のページに注があります。

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
イミディエートによるシフト	0	0	0	opcode [1]		immediate					Rm			Rd		
レジスタ加算/減算	0	0	0	1	1	0	opc	Rm			Rn			Rd		
イミディエート加算/減算	0	0	0	1	1	1	opc	immediate			Rn			Rd		
イミディエート加算/減算/比較/移動	0	0	1	opcode		Rd / Rn			immediate							
レジスタデータ処理	0	1	0	0	0	0	opcode				Rm / Rs			Rd / Rn		
特殊データ処理	0	1	0	0	0	1	opcode [1]		H1	H2	Rm			Rd / Rn		
分岐と状態遷移命令セット[3]	0	1	0	0	0	1	1	1	L	H2	Rm			SBZ		
リテラルプールからロード	0	1	0	0	1	Rd			PC相対オフセット							
レジスタオフセットのロード/ストア	0	1	0	1	opcode			Rm			Rn			Rd		
イミディエートオフセットでのワード/バイトのロード/ストア	0	1	1	B	L	offset					Rn			Rd		
イミディエートオフセットでのハーフワードのロード/ストア	1	0	0	0	L	offset					Rn			Rd		
スタックのロード/ストア	1	0	0	1	L	Rd			SP相対オフセット							
SPまたはPCに加算	1	0	1	0	SP	Rd			immediate							
その他 図6-2参照	1	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x
複数ロード/ストア	1	1	0	0	L	Rn			register list							
条件付き分岐	1	1	0	1	cond [2]				offset							
未定義命令	1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x
ソフトウェア割り込み	1	1	0	1	1	1	1	1	immediate							
無条件分岐	1	1	1	0	0	offset										
BLX接尾文字[4]	1	1	1	0	1	offset										0
未定義命令	1	1	1	0	1	x	x	x	x	x	x	x	x	x	x	1
BL/BLX接頭文字	1	1	1	1	0	offset										
BL接尾文字	1	1	1	1	1	offset										

図 A6-1 Thumb 命令セットの概要

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

1. この行では、opc フィールドに 11 を使用することはできません。他の行は、opc フィールドが 11 であるケースに対応しています。
2. この行では、cond フィールドに 1110 または 1111 を使用することはできません。他の行は、cond フィールドが 1110 または 1111 であるケースに対応しています。
3. `L == 1` の形式は、ARMv5T 以前では予測不能です。
4. ARMv5T 以前では未定義命令です。

A6.2.1 その他の命令

図 A6-2 は、その他の Thumb 命令の一覧です。角括弧内の項目（[1] など）は、図の下に注があります。

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
スタックポインタの調整	1	0	1	1	0	0	0	0	opc	immediate						
符号/ゼロ拡張[2]	1	0	1	1	0	0	1	0	opc		Rn			Rd		
レジスタリストのプッシュ/ポップ	1	0	1	1	L	1	0	R	register list							
エンディアン形式の設定[2]	1	0	1	1	0	1	1	0	0	1	0	1	E	SBZ		
プロセッサ状態の変更[2]	1	0	1	1	0	1	1	0	0	1	1	imod	0	A	I	F
バイト反転[2]	1	0	1	1	1	0	1	0	opc		Rn			Rd		
ソフトウェアブレークポイント[1]	1	0	1	1	1	1	1	0	immediate							

図 A6-2 その他の Thumb 命令

1. ARMv5 以前では未定義命令です。
2. ARMv6 以前では未定義命令です。

——— 注 ———

ビット [15:12] = 1011 で図 A6-2 にない命令は、未定義命令です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.3 分岐命令

Thumb は以下の 6 タイプの分岐命令に対応しています。

- 最大 256 バイト（-256 ～ +254）の前方分岐と後方分岐が可能な条件付き分岐
- 最大 2KB（-2048 ～ +2046）の前方分岐と後方分岐が可能な無条件分岐
- リンク付き分岐（サブルーチンコール）は、最大 4MB（-2^{22}<= offset <= $+2^{22}-2$）の前方分岐と後方分岐が可能な命令のペアで対応しています。
- リンク付き分岐と状態遷移では、リンク付き分岐と同様に命令のペアを使用しますが、さらに ARM コードの実行に切り替わります。
- 状態遷移命令では、レジスタのアドレスに分岐し、必要に応じて ARM コードの実行に切り替わります。
- リンク付き分岐と状態遷移命令の 2 つ目の形式では、レジスタ中のアドレスへのサブルーチンコールを実行し、必要に応じて ARM コードの実行に切り替わります。

これらの命令のエンコードを以下に示します。

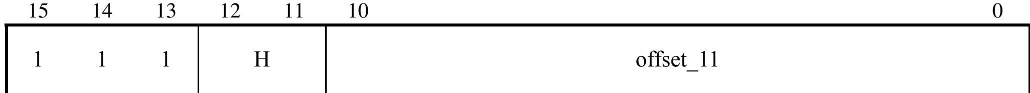
A6.3.1 条件付き分岐

```
B<cond>  <target_address>
```

15	14	13	12	11 - 8	7 - 0
1	1	0	1	cond	8_bit_signed_offset

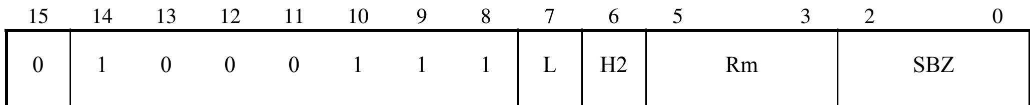
A6.3.2 無条件分岐

```
B    <target_address>
BL   <target_address>            ; Produces two 16-bit instructions
BLX  <target_address>            ; Produces two 16-bit instructions
```



A6.3.3 分岐と状態遷移

```
BX    <Rm>
BLX   <Rm>
```



Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.3.4 例

```
    B    label         ; unconditionally branch to label

    BCC  label         ; branch to label if carry flag is clear

    BEQ  label         ; branch to label if zero flag is set

    BL   func          ; subroutine call to function

func
...                    ; Include body of function here
...
    MOV  PC, LR        ; R15=R14, return to instruction after the BL

    BX   R12           ; branch to address in R12; begin ARM execution if
                       ; bit 0 of R12 is zero; otherwise continue executing
                       ; Thumb code
```

A6.3.5 分岐命令の一覧

以下の命令は前述のフォーマットに従います。

B	条件付き分岐。詳細については、P. A7-19「*B (1)*」を参照して下さい。
B	無条件分岐。詳細については、P. A7-21「*B (2)*」を参照して下さい。
BL	リンク付き分岐。詳細については、P. A7-26「*BL、BLX (1)*」を参照して下さい。
BX	状態遷移命令セット。詳細については、P. A7-32「*BX*」を参照して下さい。
BLX	リンク付き状態遷移命令セット。詳細については、P. A7-26「*BL、BLX (1)*」および P. A7-30「*BLX (2)*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.4 データ処理命令

Thumb データ処理命令は、ARM データ処理命令のサブセットで、2 つのセットに分けられます。1 つ目のセットは下位レジスタ（r0 ～ r7）のみを操作でき、2 つ目のセットは上位レジスタ（r8 ～ r15）、または下位レジスタと上位レジスタの組み合わせを操作できます。

A6.4.1 下位レジスタのデータ処理命令

下位レジスタのデータ処理命令を表 A6-2 に示します。これらの命令の一部は、上位レジスタのデータ処理命令の一覧にもあります。下位レジスタへの操作の場合、この表にあるすべての命令（CPY 以外）で条件コードが設定されます。

表 A6-2 下位レジスタのデータ処理命令

ニーモニック	演算	アクション
`ADC Rd, Rm`	キャリー付き加算	Rd := Rd + Rm + キャリーフラグ
`ADD Rd, Rn, Rm`	加算	Rd := Rn + Rm
`ADD Rd, Rn, #0 to 7`	加算	Rd := Rn + 3 ビットイミディエート
`ADD Rd, #0 to 255`	加算	Rd := Rd + 8 ビットイミディエート
`AND Rd, Rm`	論理積	Rd := Rd AND Rm
`ASR Rd, Rm, #1 to 32`	算術右シフト	Rd := Rm ASR 5 ビットイミディエート
`ASR Rd, Rs`	算術右シフト	Rd := Rd ASR Rs
`BIC Rd, Rm`	ビットクリア	Rd := Rd AND NOT Rm
`CMN Rn, Rm`	2 の補数比較	Rn + Rm の後にフラグ更新
`CMP Rn, #0 to 255`	比較	Rn - 8 ビットイミディエートの後にフラグ更新
`CMP Rn, Rm`	比較	Rn - Rm の後にフラグ更新
`CPY Rd, Rn`	コピー	Rd := Rn
`EOR Rd, Rm`	排他的論理和	Rd := Rd EOR Rm
`LSL Rd, Rm, #0 to 31`	論理左シフト	Rd := Rm LSL 5 ビットイミディエート
`LSL Rd, Rs`	論理左シフト	Rd := Rd LSL Rs
`LSR Rd, Rm, #1 to 32`	論理右シフト	Rd := Rm LSR 5 ビットイミディエート
`LSR Rd, Rs`	論理右シフト	Rd := Rd LSR Rs
`MOV Rd, #0 to 255`	移動	Rd := 8 ビットイミディエート

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 A6-2 下位レジスタのデータ処理命令（続き）

ニーモニック	演算	アクション
`MOV Rd, Rn`	移動	Rd := Rn
`MUL Rd, Rm`	乗算	Rd := Rm x Rd
`MVN Rd, Rm`	論理否定	Rd := NOT Rm
`NEG Rd, Rm`	符号反転	Rd := 0 - Rm
`ORR Rd, Rm`	（包含的）論理和	Rd := Rd OR Rm
`ROR Rd, Rs`	右ローテート	Rd := Rd ROR Rs
`SBC Rd, Rm`	キャリー付き減算	Rd := Rd - Rm - NOT（キャリーフラグ）
`SUB Rd, Rn, Rm`	減算	Rd := Rn - Rm
`SUB Rd, Rn, #0 to 7`	減算	Rd := Rn - 3 ビットイミディエート
`SUB Rd, #0 to 255`	減算	Rd := Rd - 8 ビットイミディエート
`TST Rn, Rm`	テスト	Rn AND Rm の後にフラグ更新

例：

```
ADD     R0, R4, R7          ; R0 = R4 + R7
SUB     R6, R1, R2          ; R6 = R1 - R2
ADD     R0, #255            ; R0 = R0 + 255
ADD     R1, R4, #4          ; R1 = R4 + 4
NEG     R3, R1              ; R3 = 0 - R1
AND     R2, R5              ; R2 = R2 AND R5
EOR     R1, R6              ; R1 = R1 EOR R6
CMP     R2, R3              ; update flags after R2 - R3
CMP     R7, #100            ; update flags after R7 - 100
MOV     R0, #200            ; R0 = 200
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.4.2 上位レジスタ

表 A6-3 に示されているように、ARM レジスタ 8 ～ 14 と PC の演算を行うデータ処理命令は 8 タイプあります。CMP を除き、この表にある命令では条件コードフラグは変更されません。

表 A6-3 上位レジスタのデータ処理命令

ニーモニック	演算	アクション
`MOV Rd, Rn`	移動	Rd := Rn
`CPY Rd, Rn`	コピー	Rd := Rn
`ADD Rd, Rm`	加算	Rd := Rd + Rm
`CMP Rn, Rm`	比較	Rn - Rm の後にフラグ更新
`ADD SP, #0 to 508`	スタックポインタをインクリメント	R13 = R13 + 4*（7 ビットイミディエート）
`SUB SP, #0 to 508`	スタックポインタをデクリメント	R13 = R13 - 4*（7 ビットイミディエート）
`ADD Rd, SP, #0 to 1020`	スタックアドレス生成	Rd = R13 + 4*（8 ビットイミディエート）
`ADD Rd, PC, #0 to 1020`	PC アドレス生成	Rd = PC + 4*（8 ビットイミディエート）

例

```
MOV     R0, R12             ; R0 = R12
ADD     R10, R1             ; R10 = R10 + R1
MOV     PC, LR              ; PC = R14
CMP     R10, R11            ; update flags after R10 - R11
SUB     SP, #12             ; increase stack size by 12 bytes
ADD     SP, #16             ; decrease stack size by 16 bytes
ADD     R2, SP, #20         ; R2 = SP + 20
ADD     R0, PC, #500        ; R0 = PC + 500
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.4.3 フォーマット

データ処理命令では、以下の 8 種類の命令フォーマットを使用します。

フォーマット 1

```
<opcode1>  <Rd>, <Rn>, <Rm>
<opcode1> := ADD | SUB
```

15	14	13	12	11	10	9	8 6	5 3	2 0
0	0	0	1	1	0	op_1	Rm	Rn	Rd

フォーマット 2

```
<opcode2>  <Rd>, <Rn>, #<3_bit_immed>
<opcode2> := ADD | SUB
```

15	14	13	12	11	10	9	8 6	5 3	2 0
0	0	0	1	1	1	op_2	3_bit_immediate	Rn	Rd

フォーマット 3

```
<opcode3>  <Rd>|<Rn>, #<8_bit_immed>
<opcode3> := ADD | SUB | MOV | CMP
```

15	14	13	12 11	10 8	7 0
0	0	1	op_3	Rd|Rn	8_bit_immediate

フォーマット 4

```
<opcode4>  <Rd>, <Rm>, #<shift_imm>
<opcode4> := LSL | LSR | ASR
```

15	14	13	12 11	10 6	5 3	2 0
0	0	0	op_4	shift_immediate	Rm	Rd

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

フォーマット 5

```
<opcode5>  <Rd>|<Rn>, <Rm>|<Rs>
<opcode5> := MVN | CMP | CMN | TST | ADC | SBC | NEG | MUL |
             LSL | LSR | ASR | ROR | AND | EOR | ORR | BIC
```

15	14	13	12	11	10	9 – 6	5 – 3	2 – 0
0	1	0	0	0	0	op_5	Rm|Rs	Rd|Rn

フォーマット 6

```
ADD  <Rd>, <reg>, #<8_bit_immed>
<reg> := SP | PC
```

15	14	13	12	11	10 – 8	7 – 0
1	0	1	0	reg	Rd	8_bit_immediate

フォーマット 7

```
<opcode6>  SP, SP, #<7_bit_immed>
<opcode6> := ADD | SUB
```

15	14	13	12	11	10	9	8	7	6 – 0
1	0	1	1	0	0	0	0	op_6	7_bit_immediate

フォーマット 8

```
<opcode7>  <Rd>|<Rn>, <Rm>
<opcode7> := MOV | ADD | CMP | CPY
```

15	14	13	12	11	10	9 – 8	7	6	5 – 3	2 – 0
0	1	0	0	0	1	opcode	H1	H2	Rm	Rd|Rn

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.4.4 データ処理命令の一覧

以下の命令は前述のフォーマットに従います。

`ADC` キャリー付き加算。詳細については、P. A7-4「*ADC*」を参照して下さい。

`ADD` 加算（イミディエート）。詳細については、P. A7-5「*ADD (1)*」を参照して下さい。

`ADD` 加算（ラージイミディエート）。詳細については、P. A7-6「*ADD (2)*」を参照して下さい。

`ADD` 加算（レジスタ）。詳細については、P. A7-7「*ADD (3)*」を参照して下さい。

`ADD` 上位レジスタを加算。詳細については、P. A7-8「*ADD (4)*」を参照して下さい。

`ADD` 加算（イミディエートをプログラムカウンタに）。詳細については、P. A7-10「*ADD (5)*」を参照して下さい。

`ADD` 加算（イミディエートをスタックポインタに）。詳細については、P. A7-11「*ADD (6)*」を参照して下さい。

`ADD` スタックポインタをインクリメント。詳細については、P. A7-12「*ADD (7)*」を参照して下さい。

`AND` 論理積。詳細については、P. A7-14「*AND*」を参照して下さい。

`ASR` 算術右シフト（イミディエート）。詳細については、P. A7-15「*ASR (1)*」を参照して下さい。

`ASR` 算術右シフト（レジスタ）。詳細については、P. A7-17「*ASR (2)*」を参照して下さい。

`BIC` ビットクリア。詳細については、P. A7-23「*BIC*」を参照して下さい。

`CMN` 補数比較（レジスタ）。詳細については、P. A7-34「*CMN*」を参照して下さい。

`CMP` 比較（イミディエート）。詳細については、P. A7-35「*CMP (1)*」を参照して下さい。

`CMP` 比較（レジスタ）。詳細については、P. A7-36「*CMP (2)*」を参照して下さい。

`CMP` 上位レジスタを比較。詳細については、P. A7-37「*CMP (3)*」を参照して下さい。

`CPY` 上位レジスタまたは下位レジスタをコピー。詳細については、P. A7-41「*CPY*」を参照して下さい。

`EOR` 排他的論理和。詳細については、P. A7-43「*EOR*」を参照して下さい。

`LSL` 論理左シフト（イミディエート）。詳細については、P. A7-64「*LSL (1)*」を参照して下さい。

`LSL` 論理左シフト（レジスタ）。詳細については、P. A7-66「*LSL (2)*」を参照して下さい。

`LSR` 論理右シフト（イミディエート）。詳細については、P. A7-68「*LSR (1)*」を参照して下さい。

`LSR` 論理右シフト（レジスタ）。詳細については、P. A7-70「*LSR (2)*」を参照して下さい。

`MOV` 移動（イミディエート）。詳細については、P. A7-72「*MOV (1)*」を参照して下さい。

`MOV` 下位レジスタを別の下位レジスタに移動。詳細については、P. A7-73「*MOV (2)*」を参照して下さい。

`MOV` 上位レジスタを移動。詳細については、P. A7-75「*MOV (3)*」を参照して下さい。

`MUL` 乗算。詳細については、P. A7-77「*MUL*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

`MVN`	論理否定（レジスタ）。詳細については、P. A7-79「*MVN*」を参照して下さい。
`NEG`	否定（レジスタ）。詳細については、P. A7-80「*NEG*」を参照して下さい。
`ORR`	論理和。詳細については、P. A7-81「*ORR*」を参照して下さい。
`ROR`	右ローテート（レジスタ）。詳細については、P. A7-92「*ROR*」を参照して下さい。
`SBC`	キャリー付き減算（レジスタ）。詳細については、P. A7-94「*SBC*」を参照して下さい。
`SUB`	減算（イミディエート）。詳細については、P. A7-113「*SUB (1)*」を参照して下さい。
`SUB`	減算（ラージイミディエート）。詳細については、P. A7-114「*SUB (2)*」を参照して下さい。
`SUB`	減算（レジスタ）。詳細については、P. A7-115「*SUB (3)*」を参照して下さい。
`SUB`	スタックポインタをデクリメント。詳細については、P. A7-116「*SUB (4)*」を参照して下さい。
`TST`	テスト（レジスタ）。詳細については、P. A7-122「*TST*」を参照して下さい。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.5 レジスタロード / ストア命令

Thumb は、8 タイプのレジスタロード / ストア命令に対応しています。2 種類の基本アドレシングモードがあり、ワード、ハーフワードおよびバイトのロードとストアが可能です。また、符号付きハーフワードとバイトのロードも可能です。

- レジスタ + レジスタ
- レジスタ + 5 ビットイミディエート（符号付きハーフワードと符号付きバイトのロードには使用できません）。

イミディエートオフセットが使用されている場合、ワードアクセスには 4、ハーフワードアクセスには 2 でスケーリングされます。

その他に、3 つの特殊命令が使用可能です。

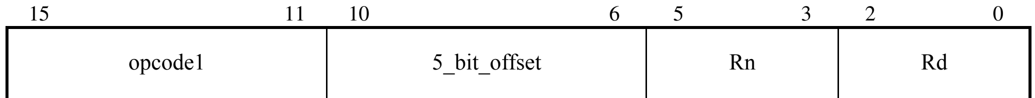
- PC をベースとして使用し、1KB のイミディエートオフセット（ワード境界でアライン）でワードをロードする。
- スタックポインタ（R13）をベースとして、1KB のイミディエートオフセット（ワード境界でアライン）でワードをロードおよびストアする。

A6.5.1 フォーマット

レジスタロード / ストア命令は以下のフォーマットを使用します。

フォーマット 1

```
<opcode1>  <Rd>, [<Rn>, #<5_bit_offset>]
<opcode1> := LDR|LDRH|LDRB|STR|STRH|STRB
```



フォーマット 2

```
<opcode2>  <Rd>, [<Rn>, <Rm>]
<opcode2> := LDR|LDRH|LDRSH|LDRB|LDRSB|STR|STRH|STRB
```

15 – 9	8 – 6	5 – 3	2 – 0
opcode2	Rm	Rn	Rd

フォーマット 3

```
LDR  <Rd>, [PC, #<8_bit_offset>]
```

15	14	13	12	11	10 – 8	7 – 0
0	1	0	0	1	Rd	8_bit_immediate

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

フォーマット 4

```
<opcode3>  <Rd>, [SP, #<8_bit_offset>]
<opcode3>  := LDR | STR
```

15	14	13	12	11	10 - 8	7 - 0
1	0	0	1	L	Rd	8_bit_immediate

例

```
LDR     R4, [R2, #4]          ; Load word into R4 from address R2 + 4
LDR     R4, [R2, R1]          ; Load word into R4 from address R2 + R1
STR     R0, [R7, #0x7C]       ; Store word from R0 to address R7 + 124
STRB    R1, [R5, #31]         ; Store byte from R1 to address R5 + 31
STRH    R4, [R2, R3]          ; Store halfword from R4 to R2 + R3
LDRH    R3, [R6, R5]          ; Load word into R3 from R6 + R5
LDRB    R2, [R1, #5]          ; Load byte into R2 from R1 + 5
LDR     R6, [PC, #0x3FC]      ; Load R6 from PC + 0x3FC
LDR     R5, [SP, #64]         ; Load R5 from SP + 64
STR     R4, [SP, #0x260]      ; Load R5 from SP + 0x260
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.5.2 レジスタロード / ストア命令の一覧

以下の命令は前述のフォーマットに従います。

LDR	ワードのロード（イミディエートオフセット）。詳細については、P. A7-47「*LDR (1)*」を参照して下さい。
LDR	ワードのロード（レジスタオフセット)。詳細については、P. A7-49「*LDR (2)*」を参照して下さい。
LDR	ワードのロード（PC 相対)。詳細については、P. A7-51「*LDR (3)*」を参照して下さい。
LDR	ワードのロード（SP 相対)。詳細については、P. A7-53「*LDR (4)*」を参照して下さい。
LDRB	符号なしバイトのロード（イミディエートオフセット）。詳細については、「P. A7-55「*LDRB (1)*」」を参照して下さい。
LDRB	符号なしバイトのロード（レジスタオフセット）。詳細については、P. A7-56「*LDRB (2)*」を参照して下さい。
LDRH	符号なしハーフワードのロード（イミディエートオフセット）。詳細については、P. A7-57「*LDRH (1)*」を参照して下さい。
LDRH	符号なしハーフワードのロード（レジスタオフセット）。詳細については、P. A7-59「*LDRH (2)*」を参照して下さい。
LDRSB	符号付きバイトのロード（レジスタオフセット)。詳細については、P. A7-61「*LDRSB*」を参照して下さい。
LDRSH	符号付きハーフワードのロード（レジスタオフセット）。詳細については、P. A7-62「*LDRSH*」を参照して下さい。
STR	ワードをストア（イミディエートオフセット）。詳細については、P. A7-99「*STR (1)*」を参照して下さい。
STR	ワードをストア（レジスタオフセット)。詳細については、P. A7-101「*STR (2)*」を参照して下さい。
STR	ワードをストア（SP 相対)。詳細については、P. A7-103「*STR (3)*」を参照して下さい。
STRB	バイトをストア（イミディエートオフセット)。詳細については、P. A7-105「*STRB (1)*」を参照して下さい。
STRB	バイトをストア（レジスタオフセット)。詳細については、P. A7-107「*STRB (2)*」を参照して下さい。
STRH	ハーフワードをストア(イミディエートオフセット)。詳細については、P. A7-109「*STRH (1)*」を参照して下さい。
STRH	ハーフワードをストア（レジスタオフセット)。詳細については、P. A7-111「*STRH (2)*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.6 複数ロード / ストア命令

Thumb は、4 タイプの複数ロード / ストア命令に対応しています。

- 2 つの命令、LDMIA と STMIA は、ブロックコピーに対応するように設計されています。ベースレジスタからの固定ポストインクリメントアドレシングモードがあります。
- 他の 2 つの命令、PUSH と POP にも固定アドレシングモードがあります。フル下降スタックが実装されており、スタックポインタ（R13）がベースレジスタとして使用されます。

4 つの命令すべてで、転送後にベースレジスタが更新され、下位 8 レジスタのいずれかまたはすべてを転送できます。PUSH では復帰アドレスのスタックも可能です。また、POP では PC をロードできます。

A6.6.1 フォーマット

複数ロード / ストア命令は次のフォーマットを使用します。

フォーマット 1

```
<opcode1>  <Rn>!, <registers>
<opcode1>  := LDMIA | STMIA
```

15	14	13	12	11	10-8	7-0
1	1	0	0	L	Rn	register_list

フォーマット 2

```
PUSH  {<registers>}
POP   {<registers>}
```

15	14	13	12	11	10	9	8	7-0
1	0	1	1	L	1	0	R	register_list

A6.6.2 例

```
    LDMIA    R7!, {R0-R3, R5}        ; Load R0 to R3-R5 from R7, add 20 to R7
    STMIA    R0!, {R3, R4, R5}       ; Store R3-R5 to R0: add 12 to R0

function
    PUSH     {R0-R7, LR}             ; push onto the stack (R13) R0-R7 and
                                     ; the return address
...                                  ; code of the function body
...
    POP      {R0-R7, PC}             ; restore R0-R7 from the stack
                                     ; and the program counter, and return
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.6.3 複数ロード / ストア命令の一覧

以下の命令は前述のフォーマットに従います。

`LDMIA` 複数ロード。詳細については、P. A7-44「*LDMIA*」を参照して下さい。

`POP` 複数ポップ。詳細については、P. A7-82「*POP*」を参照して下さい。

`PUSH` 複数プッシュ。詳細については、P. A7-85「*PUSH*」を参照して下さい。

`STMIA` 複数ストア。詳細については、P. A7-96「*STMIA*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.7 例外生成命令

Thumb命令セットには、プロセッサ例外を発生させることを主な目的とする2タイプの命令があります。

- ソフトウェア割り込み（SWI）命令は、SWI 例外の発生に使用されます（詳細については、P. A2-20 *「ソフトウェア割り込み例外」*を参照して下さい）。これは Thumb 命令セットの主要な機構で、これにより、ユーザモードのコードからオペレーティングシステムの特権コードを呼び出すことができます。

- ブレークポイント（BKPT）命令は、ARMv5T 以降でソフトウェアブレークポイントに使用されます。デフォルトの動作は、プリフェッチアボート例外を発生させることです（詳細については、P. A2-20 *「プリフェッチアボート（命令フェッチ中に発生するメモリアボート）」*を参照して下さい）。前もってプリフェッチアボートベクタにインストールしたデバッグ監視プログラムで、この例外を処理できます。

  システムにデバッグハードウェアが存在している場合、デフォルトの動作をオーバライドできます。詳細については、P. A7-24 *「BKPT」の注*を参照して下さい。

A6.7.1 命令のエンコード

```
SWI   <immed_8>
```

15	14	13	12	11	10	9	8	7 … 0
1	1	0	1	1	1	1	1	immed_8

```
BKPT   <immed_8>
```

15	14	13	12	11	10	9	8	7 … 0
1	0	1	1	1	1	1	0	immed_8

SWIとBKPTのいずれの場合も、命令のimmed_8フィールドはARMプロセッサに無視されます。SWIまたはプリフェッチアボートハンドラは、必要に応じて、例外を発生させた命令をロードしてこれらのフィールドを抽出するように作成することができます。これにより、オペレーティングシステムの呼び出しあるいはハンドラへのブレークポイントに関する情報のやり取りにそのフィールドを使用できます。

A6.7.2 例外生成命令の一覧

BKPT　ブレークポイント。詳細については、P. A7-24 *「BKPT」*を参照して下さい。

SWI　ソフトウェア割り込み。詳細については、P. A7-118 *「SWI」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A6.8 未定義命令空間

以下の命令は、Thumb 命令セットでは未定義です。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	1	x	x	x	x	x	x	x	x
1	0	1	1	0	x	1	1	x	x	x	x	x	x	x	x
1	0	1	1	1	0	0	x	x	x	x	x	x	x	x	x
1	0	1	1	1	0	1	0	1	0	x	x	x	x	x	x
1	0	1	1	1	0	1	1	x	x	x	x	x	x	x	x
1	0	1	1	1	1	1	1	x	x	x	x	x	x	x	x
1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x

一般に、これらの命令は将来の Thumb 命令セットの拡張に使用できます。ただし、以下の命令のグループはこのように使用することはできません。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x

将来のハードウェアで未定義命令が定義済み命令として扱われるリスクを最小限にするため、ソフトウェア用途で未定義命令を使用する場合は、これらの命令のいずれかを使用して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 A7 章
Thumb 命令

本章では、Thumb® に含まれるすべての命令の構文と使用法を解説します。本章は以下のセクションから構成されています。

- *Thumb 命令のアルファベット順一覧* : P. A7-2
- *Thumb 命令とアーキテクチャのバージョン* : P. A7-125

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1 Thumb 命令のアルファベット順一覧

各 Thumb 命令について、以下のページで詳しく説明します。各命令の説明には、次の内容が含まれています。

- 命令のエンコード
- 命令の構文
- この命令が有効な ARM® アーキテクチャのバージョン
- 適用される例外
- 命令の動作の擬似コード仕様
- 用法と特別なケースについての注記
- 等価な ARM 命令のエンコード

A7.1.1 一般的な注記

これらの注記は、命令のページで使用されている情報と省略形のタイプについて説明しています。

構文の省略形

命令のページでは、次の省略形が使用されています。

`immed_<n>` `<n>` ビットのイミディエート値です。たとえば、8 ビットのイミディエート値は次のように示されます。

`immed_8`

`signed_immed_<n>`

符号付きイミディエート値です。たとえば、8 ビットの符号付きイミディエート値は次のように示されます。

`signed_immed_8`

アーキテクチャのバージョン

読者の便宜のため、このセクションでは Thumb 命令セットではなく、命令に対応する ARM アーキテクチャのバージョンを示しています。今日までに、Thumb 命令セットアーキテクチャには 3 つのバージョンが存在しています。

THUMBv1 ARM 命令セットアーキテクチャのバージョン 4 の T バリエーションで使用されます。

THUMBv2 ARM 命令セットアーキテクチャのバージョン 5 の T バリエーションで使用されます。

THUMBv3 ARM 命令セットアーキテクチャのバージョン 6 以降で使用されます。

したがって、すべての T バリエーションに存在すると記載されている命令は、THUMBv1、THUMBv2、THUMBv3 に存在します。また、バージョン 6 の T バリエーションに存在すると記載されている命令は、THUMBv3 にのみ存在します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

このセクションでは、等価な ARM 命令の構文とエンコードを示します。正確に等価な命令が存在しない場合、類似の命令と、それが正確に等価でない理由が説明されています。

命令が正確に等価でない主な理由は、その命令が PC の値を読み出すためです。これによって、(命令自体のアドレス + N) が生成されます。ここで、N は ARM 命令では 8、Thumb 命令では 4 です。この相違点は多くの場合、等価な ARM 命令にあるイミディエート定数を調整して補正できます。

等価な命令のエンコードで、名前付きフィールドとビットには Thumb 命令の対応するフィールドやビットの値を入れる必要があります。場合によっては Thumb 命令から派生した値を入れる必要があり、その場合には本文に説明されています。

ARM 命令のフィールドは一般に、対応する Thumb 命令のフィールドと同じ長さですが、重要な例外が 1 つあります。Thumb のレジスタフィールドは一般に 3 ビットで、ARM のレジスタフィールドは一般に 4 ビットです。このため、Thumb レジスタフィールドを ARM レジスタフィールドで置き換える場合は最上位ビットに 0 を付加し、ARM 命令が R0 ～ R7 を参照するようにする必要があります。

使用に関する情報

使用に関する情報は、Thumb 命令の使用法が ARM 命令のものと明確に異なる場合のみ説明されています。Thumb 命令に「用法」のセクションがない場合、使用に関する情報については第 A4 章*「ARM 命令」*にある等価な ARM 命令の説明を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.2 ADC

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	0	1	0	1	Rm	Rd

ADC（キャリー付き加算）は、2 つの値とキャリーフラグを加算します。

ADC を使用して複数ワードの加算を合成します。

ADC は、結果に基づいて条件コードフラグを更新します。

構文

```
ADC  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> 加算の最初の値を保持し、演算のデスティネーションレジスタになります。

<Rm> 加算の第 2 オペランドレジスタを指定します。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rd + Rm + C Flag
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = CarryFrom(Rd + Rm + C Flag)
V Flag = OverflowFrom(Rd + Rm + C Flag)
```

等価な ARM 構文とエンコード

```
ADCS  <Rd>, <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	0	0	0	0	1	0	1	1	Rd	Rd	0	0	0	0	0	0	0	0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.3 ADD (1)

15	14	13	12	11	10	9	8 - 6	5 - 3	2 - 0
0	0	0	1	1	1	0	immed_3	Rn	Rd

ADD (1) は、小さな定数値をレジスタの値に加算し、結果を 2 番目のレジスタにストアします。

結果に基づいて条件コードフラグを更新します。

構文

```
ADD  <Rd>, <Rn>, #<immed_3>
```

各項目の説明については以下を参照して下さい。

<Rd>	演算のデスティネーションレジスタを指定します。
<Rn>	加算のオペランドレジスタを指定します。
<immed_3>	<Rn> の値に加算される 3 ビットのイミディエート値を指定します。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rn + immed_3
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = CarryFrom(Rn + immed_3)
V Flag = OverflowFrom(Rn + immed_3)
```

等価な ARM 構文とエンコード

```
ADDS  <Rd>, <Rn>, #<immed_3>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 - 16	15 - 12	11	10	9	8	7	6	5	4	3	2 - 0
1	1	1	0	0	0	1	0	1	0	0	1	Rn	Rd	0	0	0	0	0	0	0	0	0	immed_3

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.4 ADD (2)

15	14	13	12	11	10 - 8	7 - 0
0	0	1	1	0	Rd	immed_8

ADD (2) は、大きなイミディエート値をレジスタの値に加算し、結果を同じレジスタにストアします。

結果に基づいて条件コードフラグを更新します。

構文

```
ADD  <Rd>, #<immed_8>
```

各項目の説明については以下を参照して下さい。

<Rd> 加算の最初のオペランドを保持し、演算のデスティネーションレジスタになります。

<immed_8> <Rn> の値に加算される 8 ビットのイミディエート値を指定します。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rd + immed_8
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = CarryFrom(Rd + immed_8)
V Flag = OverflowFrom(Rd + immed_8)
```

等価な ARM 構文とエンコード

```
ADDS  <Rd>, <Rd>, #<immed_8>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 - 16	15 - 12	11	10	9	8	7 - 0
1	1	1	0	0	0	1	0	1	0	0	1	Rd	Rd	0	0	0	0	immed_8

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.5 ADD (3)

15	14	13	12	11	10	9	8 6	5 3	2 0
0	0	0	1	1	0	0	Rm	Rn	Rd

ADD (3) は、レジスタの値を 2 番目のレジスタの値に加算し、結果を 3 番目のレジスタにストアします。結果に基づいて条件コードフラグを更新します。

構文

```
ADD  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタを指定します。

<Rn> 加算の最初のオペランドレジスタを指定します。

<Rm> 加算の 2 番目のオペランドレジスタを指定します。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rn + Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = CarryFrom(Rn + Rm)
V Flag = OverflowFrom(Rn + Rm)
```

等価な ARM 構文とエンコード

```
ADDS  <Rd>, <Rn>, <Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
1 1 1 0	0 0 0 0 1 0 0 1	Rn	Rd	0 0 0 0 0 0 0 0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.6 ADD (4)

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	1	0	0	H1	H2	Rm	Rd

ADD (4) は 2 つのレジスタの値を加算するもので、片方または両方が上位レジスタです。

下位レジスタのみを使用する ADD 命令（P. A7-7「*ADD (3)*」参照）とは異なり、この命令ではフラグが変更されません。

構文

```
ADD  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> 加算の最初のオペランドレジスタで、デスティネーションレジスタでもあります。R0 から R15 までの任意のレジスタです。レジスタ番号は命令の H1（最上位ビット）と Rd（他の 3 ビット）にエンコードされます。

<Rm> 加算の 2 番目のオペランドレジスタを指定します。R0 から R15 までの任意のレジスタです。レジスタ番号は命令の H2（最上位ビット）と Rm（他の 3 ビット）にエンコードされます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rd + Rm
```

注

オペランドの制限 <Rd> と Rm の両方に下位レジスタを指定した（H1 == 0 かつ H2 == 0）場合、結果は予測不能です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

完全に等価な命令は存在しません。次の命令が最も等価に近い命令です。

```
ADD  <Rd>, <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18 16	15	14 12	11	10	9	8	7	6	5	4	3	2 0
1	1	1	0	0	0	0	0	1	0	0	0	H1	Rd	H1	Rd	0	0	0	0	0	0	0	0	H2	Rm

ARM コードと Thumb コードを実行しているときの PC の定義が異なるため、命令による PC へのアクセスに多少の相違があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.7 ADD (5)

15	14	13	12	11	10 8	7 0
1	0	1	0	0	Rd	immed_8

ADD (5) はイミディエート値を PC に加算し、結果の PC 相対アドレスをデスティネーションレジスタに書き込みます。イミディエート値は、0 ～ 1020 の範囲に含まれる 4 の倍数です。

条件コードは変化しません。

構文

```
ADD  <Rd>, PC, #<immed_8> * 4
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタを指定します。

PC PC 相対アドレシングを示します。

<immed_8> 4 倍して PC の値に加算される、8 ビットのイミディエート値を指定します。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

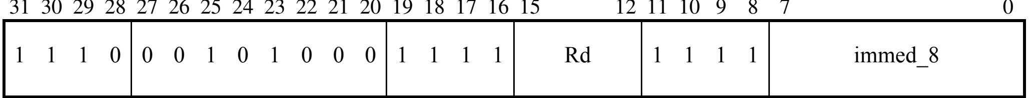
動作

```
Rd = (PC AND 0xFFFFFFFC) + (immed_8 << 2)
```

等価な ARM 構文とエンコード

完全に等価な命令は存在しません。次の命令が最も等価に近い命令です。

```
ADD  <Rd>, PC, #<immed_8> * 4
```



ARM コードと Thumb コードでは PC の定義が異なります。このため、命令の正確な結果に相違が生じます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.8 ADD (6)

15	14	13	12	11	10 8	7 0
1	0	1	0	1	Rd	immed_8

ADD (6) はイミディエート値を SP に加算し、結果の SP 相対アドレスをデスティネーションレジスタに書き込みます。イミディエート値は、0 ～ 1020 の範囲に含まれる 4 の倍数です。

条件コードは変化しません。

構文

```
ADD  <Rd>, SP, #<immed_8> * 4
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタを指定します。

SP SP 相対アドレシングを示します。

<immed_8> 4 倍して SP の値に加算される、8 ビットのイミディエート値を指定します。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = SP + (immed_8 << 2)
```

等価な ARM 構文とエンコード

```
ADD  <Rd>, SP, #<immed_8> * 4
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 12	11 10 9 8	7 0
1 1 1 0	0 0 1 0 1 0 0 0	1 1 0 1	Rd	1 1 1 1	immed_8

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.9 ADD (7)

15	14	13	12	11	10	9	8	7	6 … 0
1	0	1	1	0	0	0	0	0	immed_7

ADD(7)はSPの値を7ビットイミディエート値の4倍だけインクリメントします(4の倍数で、0～508になります)。

条件コードは変化しません。

構文

```
ADD  SP, #<immed_7> * 4
```

各項目の説明については以下を参照して下さい。

`SP` 加算の最初のオペランド。演算のデスティネーションレジスタでもあります。

`<immed_7>` 4 倍して SP の値に加算されるイミディエート値を指定します。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
SP = SP + (immed_7 << 2)
```

用法

Thumb 命令セットで使用するフル下降スタックでは、SP の値をインクリメントすると、スタックの最上部にあるデータが破棄されます。

注

代替構文 この命令は、`ADD SP, SP, #(<immed_7> * 4)` の形式で記述することもできます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
ADD  SP, SP, #<immed_7> * 4
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6 ... 0
1	1	1	0	0	0	1	0	1	0	0	0	1	1	0	1	1	1	0	1	1	1	1	1	0	immed_7

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.10 AND

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	0	0	0	0	Rm	Rd

AND（論理 AND）は、2 つのレジスタの値に対してビット単位の AND を実行します。

AND は、結果に基づいて条件コードフラグを更新します。

構文

```
AND  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> 最初のオペランドレジスタで、デスティネーションレジスタでもあります。

<Rm> 2 番目のオペランドレジスタを指定します。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rd AND Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

等価な ARM 構文とエンコード

```
ANDS  <Rd>, <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	0	0	0	0	0	0	0	1	Rd	Rd	0	0	0	0	0	0	0	0	Rm

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.11 ASR (1)

15	14	13	12	11	10 - 6	5 - 3	2 - 0
0	0	0	1	0	immed_5	Rm	Rd

ASR (1)（算術右シフト）は、レジスタの符号付き数値を 2 の累乗（定数）で除算します。

結果に基づいて条件コードフラグを更新します。

構文

```
ASR  <Rd>, <Rm>, #<immed_5>
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタを指定します。

<Rm> シフトされる値を保持するレジスタを指定します。

<immed_5> シフトするビット数を 1 〜 32 の範囲で指定します。1 〜 31 ビットのシフトは、immed_5 に直接エンコードされます。32 ビットのシフトは immed_5 == 0 としてエンコードされます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
if immed_5 == 0
    C Flag = Rm[31]
    if Rm[31] == 0 then
        Rd = 0
    else /* Rm[31] == 1 */]
        Rd = 0xFFFFFFFF
else /* immed_5 > 0 */
    C Flag = Rm[immed_5 - 1]
    Rd = Rm Arithmetic_Shift_Right immed_5
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
MOVS  <Rd>, <Rm>, ASR #<immed_5>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11 7	6	5	4	3 0
1	1	1	0	0	0	0	1	1	0	1	1	SBZ	Rd	immed_5	1	0	0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.12 ASR (2)

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	0	1	0	0	Rs	Rd

ASR (2) は、レジスタの符号付き数値を 2 の累乗（変数）で除算します。

結果に基づいて条件コードフラグを更新します。

構文

```
ASR  <Rd>, <Rs>
```

各項目の説明については以下を参照して下さい。

<Rd> シフトされる値を保持し、演算のデスティネーションレジスタになります。

<Rs> シフト数を保持するレジスタを指定します。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
if Rs[7:0] == 0 then
    C Flag = unaffected
    Rd = unaffected
else if Rs[7:0] < 32 then
    C Flag = Rd[Rs[7:0] - 1]
    Rd = Rd Arithmetic_Shift_Right Rs[7:0]
else /* Rs[7:0] >= 32 */
    C Flag = Rd[31]
    if Rd[31] == 0 then
        Rd = 0
    else /* Rd[31] == 1 */
        Rd = 0xFFFFFFFF
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
MOVS  <Rd>, <Rd>, ASR <Rs>
```

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
1	1	1	0	0	0	0	1	1	0	1	1	SBZ	Rd	Rs	0	1	0	1	Rd

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.13 B (1)

15	14	13	12	11 8	7 0
1	1	0	1	cond	signed_immed_8

B (1)（分岐）は、ターゲットアドレスへの条件付き分岐を実行します。

構文

```
B<cond>  <target_address>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。

<target_address>

分岐先のアドレスを指定します。分岐先アドレスは、次のように計算されます。

1. 命令の 8 ビット符号付きオフセットフィールドを左に 1 ビットシフトします。
2. 結果を 32 ビットに符号拡張します。
3. 結果を PC の内容に加算します（PC には分岐命令のアドレス + 4 が含まれています）。

したがって、命令で指定できる分岐先は、現在の PC（R15）の値から -256 〜 +254 バイトです。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    PC = PC + (SignExtend(signed_immed_8) << 1)
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

signed_immed_8 の現在の値を計算するには、アセンブラ（または他のツールキットコンポーネント）が次の動作を実行する必要があります。

1. 分岐のベースアドレスを計算します。これは、(分岐命令のアドレス + 4) です。つまり、ベースアドレスはこの命令によって読み出される PC の値と同じです。
2. ターゲットアドレスからベースアドレスを減算し、バイトオフセットを計算します。Thumb 命令はすべてハーフワード境界アラインメントなため、このオフセットは常に偶数です。
3. バイトオフセットが -256 ～ +254 の範囲外の場合、必要に応じて別のコード生成手法を使用するか、エラーを生成します。
4. それ以外の場合、バイトオフセットを 2 で除算した結果を命令の signed_immed_8 フィールドに設定します。

注

メモリバウンド アドレス 0 よりも前への分岐と、32 ビットアドレス空間の最後を越えた分岐の結果は予測不能です。

AL 条件 条件フィールドが `AL` (0b1110) の場合、命令は未定義になります。無条件分岐が必要な場合、P. A7-21 「*B (2)*」で説明されている無条件分岐命令を使用して下さい。

NV 条件 条件フィールドが `NV` (0b1111) の場合、命令は `SWI` になります。詳細についてはP. A7-118 「*SWI*」を参照して下さい。

等価な ARM 構文とエンコード

完全に等価な命令は存在しません。次の命令が最も等価に近い命令です。

```
B<cond>  <target_address>
```

31–28	27	26	25	24	23–8	7–0
cond	1	0	1	0	sign extension of signed_immed_8	signed_immed_8

これは Thumb 命令とは異なります。ARM 命令ではオフセットが PC に加算される前に左に 2 ビットシフトされますが、Thumb 命令では 1 ビットだけ左にシフトされます。また、ARM 命令と Thumb 命令では読み出される PC の値が異なります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.14 B (2)

15	14	13	12	11	10 ... 0
1	1	1	0	0	signed_immed_11

B (2) は、ターゲットアドレスへの無条件分岐を実行します。

構文

```
B  <target_address>
```

各項目の説明については以下を参照して下さい。

`<target_address>`

分岐先のアドレスを指定します。分岐先アドレスは、次のように計算されます。

1. 命令の 11 ビット符号付きオフセットフィールドを左に 1 ビットシフトします。
2. 結果を 32 ビットに符号拡張します。
3. 結果を PC の内容に加算します(PC には分岐命令のアドレス + 4 が含まれています)。

したがって、命令で指定できる分岐先は、現在の PC（R15）の値から -2048 ～ +2046 バイトです。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
PC = PC + (SignExtend(signed_immed_11) << 1)
```

用法

signed_immed_11 の正しい値を計算するには、アセンブラ（または他のツールキットコンポーネント）が次の動作を実行する必要があります。

1. 分岐のベースアドレスを計算します。これは、(分岐命令のアドレス + 4) です。つまり、ベースアドレスはこの命令によって読み出される PC の値と同じです。
2. ターゲットアドレスからベースアドレスを減算し、バイトオフセットを計算します。Thumb 命令はすべてハーフワード境界アラインメントなため、このオフセットは常に偶数です。
3. バイトオフセットが -2048 ～ +2046 の範囲外の場合、必要に応じて別のコード生成手法を使用するか、エラーを生成します。
4. それ以外の場合、バイトオフセットを 2 で除算した結果を命令の signed_immed_11 フィールドに設定します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

メモリバウンド	アドレス 0 よりも前への分岐と、32 ビットアドレス空間の最後を越えた分岐の結果は予測不能です。

等価な ARM 構文とエンコード

完全に等価な命令は存在しません。次の命令が最も等価に近い命令です。

```
B  <target_address>
```

31–28	27	26	25	24	23–11	10–0
1 1 1 0	1	0	1	0	sign extension of signed_immed_11	signed_immed_11

これは Thumb 命令とは異なります。ARM 命令ではオフセットが PC に加算される前に左に 2 ビットシフトされますが、Thumb 命令では 1 ビットだけ左にシフトされます。また、ARM 命令と Thumb 命令では読み出される PC の値が異なります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.15 BIC

15	14	13	12	11	10	9	8	7	6	5 – 3	2 – 0
0	1	0	0	0	0	1	1	1	0	Rm	Rd

BIC（ビットクリア）はレジスタの値と、別のレジスタの値をビット単位で反転した値との間で、ビット単位の AND を実行します。

BIC は、結果に基づいて条件コードフラグを更新します。

構文

```
BIC  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> AND 演算が実行されるレジスタで、演算のデスティネーションレジスタになります。

<Rm> <Rd>とのANDの対象となるレジスタ。このレジスタの値の補数がAND演算の対象となります。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rd AND NOT Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

等価な ARM 構文とエンコード

```
BICS  <Rd>, <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 – 16	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	0	0	0	0	1	1	1	0	1	Rd	Rd	0	0	0	0	0	0	0	0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.16 BKPT

15	14	13	12	11	10	9	8	7 … 0
1	0	1	1	1	1	1	0	immed_8

BKPT（ブレークポイント）は、ソフトウェアブレークポイントを発生させます。このブレークポイントは、プリフェッチアボートベクタにインストールされている例外ハンドラにより処理されます。デバッグハードウェアが含まれている実装では、オプションとしてハードウェアがこの動作をオーバライドし、自身でブレークポイントの処理を行うこともできます。この場合、プリフェッチアボートベクタへの分岐は実行されません。

構文

```
BKPT  <immed_8>
```

各項目の説明については以下を参照して下さい。

`<immed_8>` 8 ビットイミディエート値で、命令のビット [7:0] に置かれます。この値は ARM ハードウェアにより無視されますが、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追加情報を格納できます。

アーキテクチャのバージョン

ARMv5 以降の T バリエーションに存在します。

例外

プリフェッチアボート

動作

```
if (not overridden by debug hardware) /* See notes */
    R14_abt    = address of BKPT instruction + 4
    SPSR_abt   = CPSR
    CPSR[4:0] = 0b10111              /* Enter Abort mode */
    CPSR[5]    = 0                    /* Execute in ARM state */
    /* CPSR[6] is unchanged */
    CPSR[7]    = 1                    /* Disable normal interrupts */
    if high vectors configured then
        PC     = 0xFFFF000C
    else
        PC     = 0x0000000C
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

BKPT 命令の正確な用法は、使用されているデバッグシステムにより異なります。デバッグシステムは、BKPT を 2 つの方法で使用できます。

- デバッグハードウェア（存在している場合）は、通常の BKPT の動作をオーバライドせず、プリフェッチアボートベクタへの分岐が実行されます。システムで、実際のプリフェッチアボートの発生も許容されている場合、プリフェッチアボートハンドラはシステム固有の方法で、ベクタへの分岐が BKPT 命令の結果として発生したのか、実際のプリフェッチアボートの結果として発生したのかを判断し、それに従ってデバッグコードまたはプリフェッチアボートコードに分岐する必要があります。それ以外の場合、プリフェッチアボートハンドラは直ちにデバッグコードに分岐します。

  この方法で使用する場合、アボートハンドラ内で BKPT が発生すると R14_abt と SPSR_abt が破壊されるため、これを避ける必要があります。同じ理由から、FIQ ハンドラ内でも避ける必要があります。これは、FIQ 割り込みはアボートハンドラ内で発生する可能性があるためです。

- デバッグハードウェアが BKPT の通常の動作をオーバライドし、ソフトウェアブレークポイントを自身で処理します。処理が完了したとき、通常は BKPT の次の命令から実行を再開するか、BKPT を別の命令に置き換えてその命令から実行を再開します。

  BKPT がこの方法で使用される場合、R14_abt と SPSR_abt は破壊されないため、アボートと FIQ ハンドラの使用に関して上で説明した制限は適用されません。

注

ハードウェアオーバライド

実装のデバッグハードウェアは、BKPT の通常の動作をオーバライドすることが許されます。このため、ソフトウェアはこの命令を、使用されているデバッグシステム（存在する場合）で許容されている以外の目的に使用できません。特に、ソフトウェアはプリフェッチアボート例外の発生に依存できません。ただし、システムにデバッグハードウェアが存在しないことが保証されている場合と、デバッグシステムによって発生することが指定されている場合は除きます。

ARMv6 の場合、*デバッグステータス / 制御レジスタ*（DSCR）がデバッグハードウェアイネーブルビットを提供し、BKPT 命令が実行されたときに、BKPT 命令が実行されたことがデバッグエントリの方式ステータスフィールドに示されます。詳細については、P. D3-10 *「レジスタ 1: デバッグステータス / 制御レジスタ（DSCR）」*を参照して下さい。

等価な ARM 構文とエンコード

```
BKPT  <immed_8>
```

31 30 29 28	27 26 25 24 23 22 21 10	19 18 17 6 15 14 13 12	11 8	7 4	3 0
1 1 1 0	0 0 0 1 0 0 1 0	0 0 0 0 0 0 0 0	immed_8 [7:4]	0 1 1 1	immed_8 [3:0]

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.17 BL、BLX (1)

15	14	13	12 11	10 ... 0
1	1	1	H	offset_11

BL（リンク付き分岐）は、別の Thumb ルーチンに無条件サブルーチン呼び出しを実行します。サブルーチンからの復帰は通常、次のいずれかの方法で実行されます。

- MOV PC,LR
- BX LR
- PC をロードする POP 命令

BLX (1)（リンク付き分岐と状態遷移）は、ARM ルーチンに無条件サブルーチン呼び出しを実行します。サブルーチンからの復帰は通常、BX LR 命令、または PC をロードする LDR か LDM 命令で実行されます。

ターゲットサブルーチンに対して十分に大きなオフセットを使用できるようにするため、BL 命令と BLX 命令はアセンブラによって自動的に、2 つの 16 ビット Thumb 命令のシーケンスに変換されます。

- 最初の Thumb 命令は H == 10 で、分岐オフセットの上位部分を提供します。この命令はサブルーチン呼び出しをセットアップし、BL と BLX 形式で共有されます。
- 2 番目の Thumb 命令は H == 11（BL の場合）または H == 01（BLX の場合）です。この命令は分岐オフセットの下位部分を提供し、サブルーチン呼び出しを実行します。

構文

```
BL   <target_addr>
BLX  <target_addr>
```

各項目の説明については以下を参照して下さい。

<target_addr> 分岐先のアドレスを指定します。分岐先アドレスは、次のように計算されます。

1. 最初の命令の offset_11 フィールドを左に 12 ビットシフトします。
2. 結果を 32 ビットに符号拡張します。
3. 結果を PC の内容に加算します（PC には最初の命令のアドレス + 4 が含まれています)。
4. 2 番目の命令の offset_11 フィールドを 2 回加算します。BLX の場合、生成されたアドレスのビット [1] が強制的にクリアされ、アドレスがワード境界アラインメントになります。

したがって、命令で指定できる分岐範囲は約 ±4MB です。正確な範囲についてはは P. A7-27 *「用法」*を参照して下さい。

アーキテクチャのバージョン

BL (H == 10 と H == 11 の形式）はすべての T バリエーションに存在します。

BLX（H == 01 の形式）は ARMv5 以降の T バリエーションに存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

なし

動作

```
if H == 10 then
    LR = PC + (SignExtend(offset_11) << 12)

else if H == 11 then
    PC = LR + (offset_11 << 1)
    LR = (address of next instruction) | 1

else if H == 01 then
    PC = (LR + (offset_11 << 1)) AND 0xFFFFFFFC
    LR = (address of next instruction) | 1
    CPSR T bit = 0
```

用法

正しい命令のペアを生成するため、アセンブラ（または他のツールキットコンポーネント）は最初に、次のような手順で分岐オフセットを生成する必要があります。

1. 分岐のベースアドレスを計算します。これは、2 つの Thumb 命令のうち最初の方（H == 10 のもの）で指定されているアドレス + 4 です。つまり、ベースアドレスはこの命令によって読み出される PC の値と同じです。
2. 命令が BLX の場合、ターゲットアドレスのビット [1] を、ベースアドレスのビット [1] と同じに設定します。これは、ARM 命令のアドレスのビット [1:0] は 0b00 であるという規則の例外です。この調整は、命令の H == 01 形式に関連する制限が守られていることを保証するために必要です。
3. ターゲットアドレスからベースアドレスを減算し、オフセットを計算します。

結果として生成されるオフセットは必ず偶数です。オフセットが次の式で示される範囲の外側の場合、

$$-2^{22} <= offset <= +2^{22} - 2$$

ターゲットアドレスはこれらの命令のアドレシング範囲外です。この場合、状況に応じて別のコードを使用するか、エラーを生成します。

オフセットが範囲内の場合、2 つの Thumb 命令を生成する必要があります。これらの命令は次の形式を使用します。

- 最初の命令は H == 10 で、offset_11 = offset[22:12] です。
- 2 番目の命令は H == 11（BL の場合）または H == 01（BLX の場合）です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

エンコード　H == 00 の場合、命令は無条件分岐命令になります。詳細については Thumb 命令 P. A7-21 「*B (2)*」を参照して下さい。

BLX のビット [0]　H == 01 の場合、命令のビット [0] は 0 の必要があります。そうでない場合、命令は未定義です。上の「*用法*」で説明されているオフセットの計算方式は、BLX 命令について計算されるオフセットが 4 の倍数で、この制限に従っていることを保証しています。

メモリバウンド　アドレス 0 よりも前への分岐と、32 ビットアドレス空間の最後を越えた分岐の結果は予測不能です。

命令ペア　これらの Thumb 命令は、常に上で説明したようなペアで実行する必要があります。具体的には次のとおりです。

- アドレス A の Thumb 命令がこの命令の H == 10 形式の場合、アドレス A + 2 の Thumb 命令はこの命令の H == 01 または H == 11 形式の必要があります。
- アドレス A の Thumb 命令がこの命令の H == 01 または H == 11 形式の場合、アドレス A - 2 の Thumb 命令はこの命令の H == 10 形式の必要があります。

また、「*例外*」の下にある注記を除いて、ペアの 2 番目の命令は分岐命令の結果か、PC を変更する他の命令の結果かにかかわらず、どのような分岐のターゲットにもできません。

これらの制限のいずれかに準拠しない場合、動作は予測不能です。

例外　BL または BLX のペアに含まれる 2 つの命令の間でプロセッサ例外が発生可能かどうかは実装定義です。例外が発生可能な場合、例外から復帰するために設計された ARM 命令は、ペアの 2 番目の命令に正しく復帰することが可能な必要があります。このため、BL または BLX のペアの 2 番目の命令に復帰する場合、例外ハンドラが特別な措置を行う必要はありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

これらの命令ペアに最も近い命令は次のものです。

Thumb サブルーチンを呼び出す場合

```
BLX  <target_addr>
```

31	30	29	28	27	26	25	24	23 22 21 20	0
1	1	1	1	1	0	1	L	offset sign	offset[22:2]

ここで、L == オフセット [1] です。

ARM サブルーチンを呼び出す場合

```
BL  <target_addr>
```

ARM コードと Thumb コードでは PC の値が異なるため、これらは Thumb 命令のペアとは多少異なっています。これは、オフセットを 4 調整することで補正可能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.18 BLX (2)

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	1	1	1	1	H2	Rm	SBZ

BLX (2) は Thumb 命令セットから、レジスタで指定されたアドレスの ARM または Thumb サブルーチンを呼び出します。この命令は、分岐を行い、分岐先にある命令をデコードする命令デコーダを選択します。

CPSR の T ビットは、レジスタ Rm の値のビット [0] で更新されます。サブルーチンから呼び出し元に復帰するには BX R14 を使用します。

構文

```
BLX  <Rm>
```

各項目の説明については以下を参照して下さい。

<Rm> 分岐先アドレスを含むレジスタを指定します。R0 から R14 までの任意のレジスタです。レジスタ番号は命令の H2（最上位ビット）と Rm（他の 3 ビット）にエンコードされます。<Rm> として R15 を指定した場合、動作は予測不能です。

アーキテクチャのバージョン

ARMv5 以降の T バリエーションに存在します。

例外

なし

動作

```
target = Rm
LR = (address of the instruction after this BLX) | 1
CPSR T bit = target[0]
PC = target AND 0xFFFFFFFE
```

注

エンコード ビット 7 は、上位レジスタにアクセスする他の命令の一部で使用される H1 ビットです。この命令について、このビットが示されているように 1 ではなく 0 の場合、この命令は BX 命令になります。詳細については P. A7-32「*BX*」を参照して下さい。

ARM と Thumb の状態遷移

ARM 状態ではワード境界アラインしていないアドレスへの分岐は不可能なため、Rm[1:0] == 0b10 の場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
BLX   <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3	2 0
1	1	1	0	0	0	0	1	0	0	1	0	SBO	SBO	SBO	0	0	1	1	H2	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.19 BX

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	1	1	1	0	H2	Rm	SBZ

BX（分岐と状態遷移）は ARM コードと Thumb コードとの間の分岐を実行します。

構文

```
BX  <Rm>
```

各項目の説明については以下を参照して下さい。

<Rm> 分岐先アドレスを含むレジスタを指定します。R0 から R15 までの任意のレジスタです。レジスタ番号は命令の H2（最上位ビット）と Rm（他の 3 ビット）にエンコードされます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
CPSR T bit = Rm[0]
PC = Rm[31:1] << 1
```

用法

Thumb コードの通常のサブルーチン復帰命令は BX R14 です。次のサブルーチン呼び出し命令は、適切な復帰用の値を R14 に残します。

- ARM BLX 命令（P. A4-16「*BLX (1)*」と P. A4-18「*BLX (2)*」参照）
- Thumb BL および BLX 命令（P. A7-26「*BL、BLX (1)*」と P. A7-30「*BLX (2)*」参照）

ARMv4 の T バリエーションでは、ARM ルーチンへのサブルーチン呼び出しは次の形式のコードシーケンスで実行できます。

```
<Put address of routine to call in Ra>
MOV   LR,PC       ; Return to second following instruction
BX    Ra
```

ARM アーキテクチャ ARMv5 以降の T バリエーションでは、ARM ルーチンへのサブルーチン呼び出しは BLX 命令を使用して、より効率的に実行できます。詳細については、P. A7-26「*BL、BLX (1)*」と P. A7-30「*BLX (2)*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

エンコード　ビット 7 は、上位レジスタにアクセスする他の命令の一部で使用される H1 ビットです。この命令について、このビットを 0 ではなく 1 に指定した場合、命令は次のようになります。

- ARMv5 以降の場合、BLX 命令（P. A7-30「*BLX (2)*」参照）
- ARMv5 以前では予測不能

ARM と Thumb の状態遷移

ARM 状態ではワード境界アラインしていないアドレスへの分岐は不可能なため、Rm[1:0] == 0b10 の場合、結果は予測不能です。

R15 の使用　<Rm> にレジスタ 15 を指定することができます。この場合、Thumb コードについては通常に R15 が読み出されます。つまり、BX 命令のアドレス + 4 です。BX 命令がワード境界アラインしたアドレスにある場合、次のワードに分岐し、ARM 状態で実行が継続されます。ただし、BX 命令がワード境界アラインしたアドレスにない場合、命令の結果は予測不能です（R15 から読み出される値のビット [1:0] == 0b10 になるため）。

等価な ARM 構文とエンコード

完全に等価な命令は存在しません。次の命令が最も等価に近い命令です。

```
BX  <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3	0
1	1	1	0	0	0	0	1	0	0	1	0	SBO	SBO	SBO	0	0	0	1	H2	Rm

この ARM 命令は、Thumb 命令と完全に等価ではありません。これは、<Rm> が R15 のときに指定されている動作が異なるためです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.20 CMN

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	1	0	1	1	Rm	Rn

CMN（2 の補数比較）は、レジスタの値を、別のレジスタの値の 2 の補数とを比較します。条件フラグは 2 つのレジスタの値を加算した結果に基づいて更新されるため、以降の命令を条件付きで実行できます（条件付き分岐を使用して）。

構文

```
CMN  <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<Rn>` 比較の最初のオペランドレジスタ。

`<Rm>` 比較の 2 番目のオペランドレジスタ。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
alu_out = Rn + Rm
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = CarryFrom(Rn + Rm)
V Flag = OverflowFrom(Rn + Rm)
```

等価な ARM 構文とエンコード

```
CMN  <Rn>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	0	0	0	1	0	1	1	1	Rn	SBZ	0	0	0	0	0	0	0	0	Rm

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.21 CMP (1)

15	14	13	12	11	10 – 8	7 – 0
0	0	1	0	1	Rn	immed_8

CMP (1)（比較）は、レジスタの値を、大きなイミディエート値と比較します。条件フラグは、レジスタの値から定数を減算した結果に基づいて更新されるため、以降の命令を条件付きで実行できます(条件付き分岐を使用して)。

構文

```
CMP  <Rn>, #<immed_8>
```

各項目の説明については以下を参照して下さい。

<Rn>	比較の最初のオペランドレジスタ。
<immed_8>	比較の 2 番目のオペランドとなる 8 ビット値。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

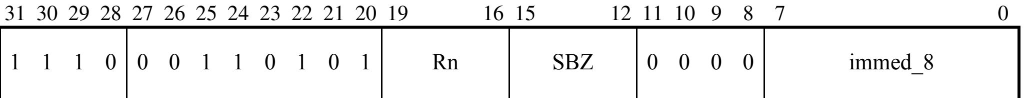
なし

動作

```
alu_out = Rn - immed_8
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - immed_8)
V Flag = OverflowFrom(Rn - immed_8)
```

等価な ARM 構文とエンコード

```
CMP  <Rn>, #<immed_8>
```



 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.22 CMP (2)

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	1	0	1	0	Rm	Rn

CMP (2) は、2 つのレジスタの値を比較します。条件フラグは、最初のレジスタの値から 2 番目のレジスタの値を減算した結果に基づいて更新されるため、以降の命令を条件付きで実行できます（条件付き分岐を使用して）。

構文

```
CMP  <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<Rn>` 比較の最初のオペランドレジスタ。

`<Rm>` 比較の 2 番目のオペランドレジスタ。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
alu_out = Rn - Rm
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - Rm)
V Flag = OverflowFrom(Rn - Rm)
```

等価な ARM 構文とエンコード

```
CMP  <Rn>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	0	0	0	1	0	1	0	1	Rn	SBZ	0	0	0	0	0	0	0	0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.23 CMP (3)

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	1	0	1	H1	H2	Rm	Rn

CMP (3) は 2 つのレジスタの値を比較するもので、片方または両方が上位レジスタです。条件フラグは、最初のレジスタの値から 2 番目のレジスタの値を減算した結果に基づいて更新されるため、以降の命令を条件付きで実行できます（条件付き分岐を使用して）。

構文

```
CMP  <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rn> 最初の値を含むレジスタ。R0 から R14 までの任意のレジスタです。レジスタ番号は命令の H1（最上位ビット）と Rn（他の 3 ビット）にエンコードされます。H1 == 1 で Rn == 0b111 の場合、エンコードは R15 を意味し、命令の結果は予測不能です。

<Rm> 2 番目の値を含むレジスタ。R0 から R15 までの任意のレジスタです。レジスタ番号は命令の H2（最上位ビット）と Rm（他の 3 ビット）にエンコードされます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
alu_out = Rn - Rm
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - Rm)
V Flag = OverflowFrom(Rn - Rm)
```

注

オペランドの制限 <Rn> と <Rm> の両方に下位レジスタを指定した場合（H1 == 0 かつ H2 == 0）、命令の結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

完全に等価な命令は存在しません。次の命令が最も等価に近い命令です。

```
CMP  <Rn>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18 16	15 12	11	10	9	8	7	6	5	4	3	2 0
1	1	1	0	0	0	0	1	0	1	0	1	H1	Rn	SBZ	0	0	0	0	0	0	0	0	H2	Rm

ARM コードと Thumb コードを実行しているときの PC の定義が異なるため、命令による PC へのアクセスに多少の相違があります。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.24 CPS

15			12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	imod	0	A	I	F

CPS（プロセッサ状態変更）は CPSR の A、I、F ビットのうち 1 つ以上を変更します。他の CPSR ビットは変更されません。

構文

```
CPS<effect> <iflags>
```

各項目の説明については以下を参照して下さい。

<effect> CPSR の割り込みディセーブルビット A、I、F にどのような変更を加えるかを指定します。これは次のいずれかです。

- IE 割り込みイネーブル、imod == 0b0 にエンコードされます。指定のビットが 0 にセットされます。
- ID 割り込みディセーブル、imod == 0b1 にエンコードされます。指定のビットが 1 にセットされます。

<iflags> 次のうち 1 つ以上のシーケンスで、どの割り込みディセーブルフラグを変更するかを指定します。

- a A ビット（ビット [2]）をセットし、CPSR の A ビットを指定されたように変更します。
- i I ビット（ビット [1]）をセットし、CPSR の I ビットを指定されたように変更します。
- f F ビット（ビット [0]）をセットし、CPSR の F ビットを指定されたように変更します。

アーキテクチャのバージョン

ARMv6 以降の T バリエーションに存在します。

例外

なし

動作

```
if InAPrivilegedMode() then
    if A == 1 then CPSR[8] = imod
    if I == 1 then CPSR[7] = imod
    if F == 1 then CPSR[6] = imod
/* else no change to interrupt disable bits */
```

注

ユーザモード　この命令はユーザモードでは無効です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
CPS  <effect>, <flags>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18	17	16	15 … 9	8	7	6	5	4 … 0
1 1 1 1	0 0 0 1 0 0 0 0	imod[a]	0	0	SBZ	A	I	F	0	SBZ

a. ARM 構文では imod は厳密には 2 ビットフィールドで、最上位ビットがセットされています（ビット [19] == 1）。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.25 CPY

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	1	1	0	H1	H2	Rm	Rd

CPY（コピー）は、ある上位または下位レジスタの値を別の上位または下位レジスタに移動し、フラグは変更しません。

構文

```
CPY  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタを指定します。R0 ～ R15 の任意のレジスタで、レジスタ番号は命令の H1（最上位ビット）と Rd（他の 3 ビット）にエンコードされます。

<Rm> コピーされる値を含むレジスタを指定します。R0 ～ R15 の任意のレジスタで、レジスタ番号は命令の H2（最上位ビット）と Rm（他の 3 ビット）にエンコードされます。

アーキテクチャのバージョン

ARMv6 以降の T バリエーションに存在します。

例外

なし

動作

```
Rd = Rm
```

用法

呼び出し元が Thumb ルーチンであることが確実ならば、`CPY PC,R14` を使用してサブルーチンから復帰できます。ただし、`BX R14`（P. A7-32「*BX*」参照）を使用する方が便利です。この命令は、呼び出し元が ARM ルーチンか Thumb ルーチンかにかかわらず動作します。

注

エンコード CPY は P. A7-75「*MOV (3)*」と同じ機能を持ち、同じ命令エンコードを使用しますが、アセンブラ構文で両方のオペランドに下位レジスタを使用できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

完全に等価な命令は存在しません。次の命令が最も等価に近い命令です。

```
CPY  <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15	14-12	11	10	9	8	7	6	5	4	3	2-0
1	1	1	0	0	0	0	1	1	0	1	0	SBZ	H1	Rd	0	0	0	0	0	0	0	0	H2	Rm

ARM コードと Thumb コードを実行しているときの PC の定義が異なるため、命令による PC へのアクセスに多少の相違があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.26 EOR

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	0	0	0	1	Rm	Rd

EOR（排他的論理和）は、2 つのレジスタの値に対してビット単位の排他的論理和を実行します。

EOR は、結果に基づいて条件コードフラグを更新します。

構文

```
EOR  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> 最初のオペランドレジスタで、デスティネーションレジスタでもあります。

<Rm> 2 番目のオペランドレジスタを指定します。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rd EOR Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

等価な ARM 構文とエンコード

```
EORS  <Rd>, <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	0	0	0	0	0	0	1	1	Rd	Rd	0	0	0	0	0	0	0	0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.27 LDMIA

15	14	13	12	11	10 - 8	7 - 0
1	1	0	0	1	Rn	register_list

LDMIA(複数ロードポストインクリメント)は、汎用レジスタ R0 ～ R7 の空でないサブセットまたは全部を、連続したメモリロケーションからロードします。

構文

```
LDMIA  <Rn>!, <registers>
```

各項目の説明については以下を参照して下さい。

<Rn> 命令のための開始アドレスを格納しているレジスタ。

! ベースレジスタのライトバックを指定します。オプションではありません。

<registers> ロードされるレジスタの一覧で、カンマで区切って並べ、{ と } で囲みます。一覧は命令の register_list フィールドにエンコードされます。0 ～ 7 までのそれぞれの i について、レジスタ Ri が一覧に含まれる場合はビット [i] が 1 に、そうでない場合は 0 にセットされます。

最低 1 つのレジスタをロードする必要があります。ビット [7:0] がすべて 0 の場合、結果は予測不能です。

レジスタは最下位のメモリアドレス（start_address）から最も低い番号のレジスタに、その後で順に最上位のメモリアドレス（end_address）から最も高い番号のレジスタへの順序でロードされます。

start_address はベースレジスタ <Rn> の値です。以降のアドレスは、前のアドレスに 4 を加算して構成されます。<registers> に指定されている各レジスタについて 1 つのアドレスが生成されます。

end_address の値は、ベースレジスタの値に <registers> で指定されたレジスタ数の 4 倍を足した値 - 4 になります。

最後に、<Rn> が <registers> の一覧に含まれていない場合、ベースレジスタ <Rn> の値は <registers> に含まれているレジスタ数の 4 倍だけインクリメントされます。オペランドの制限を参照して下さい。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
start_address = Rn
end_address = Rn + (Number_Of_Set_Bits_In(register_list) * 4) - 4
address = start_address
for i = 0 to 7
    if register_list[i] == 1
        Ri = Memory[address,4]
        address = address + 4
assert end_address == address - 4
Rn = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
```

用法

LDMIA はブロックロード命令として使用します。STMIA（複数ストア）と組み合わせて、効率的なブロックコピーが実行できます。

注

オペランドの制限

<registers> にベースレジスタ <Rn> が指定されている場合、<Rn> の最終的な値はライトバックされた値ではなく、ロードされた値になります。

データアボート　データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント　実装にシステム制御コプロセッサが含まれており（第 B3 章 *「システム制御コプロセッサ」* 参照）、アライメントチェックが許可されている場合、ビット [1:0] != 0b00 のアドレスはアライメント例外を発生します。

ARMv6 から、アライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - CP15_reg1_Ubit == 0 の場合、アドレスの最下位 2 ビットが命令により無視されます。
  - CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

時間順序　この命令によって生成される、メモリの個別のワードに対するアクセスの時間順序は、一部の条件でのみ定義されます。詳細については、P. B2-13 *「メモリアクセスの制限」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

<Rn> がレジスタ一覧に含まれていない場合（W == 1）

```
LDMIA  <Rn>!, <registers>
```

<Rn> がレジスタ一覧に含まれている場合（W == 0）

```
LDMIA  <Rn>, <registers>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 - 16	15	14	13	12	11	10	9	8	7 - 0
1	1	1	0	1	0	0	0	1	0	W	1	Rn	0	0	0	0	0	0	0	0	register_list

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.28 LDR (1)

15	14	13	12	11	10 ... 6	5 ... 3	2 ... 0
0	1	1	0	1	immed_5	Rn	Rd

LDR (1)（レジスタロード）は、32 ビットのメモリデータを汎用レジスタにロードします。このアドレシングモードは、構造体（レコード）のフィールドへのアクセスに便利です。オフセットが 0 の場合、生成されるアドレスはベースレジスタ `<Rn>` の値と同じです。

構文

```
LDR  <Rd>, [<Rn>, #<immed_5> * 4]
```

各項目の説明については以下を参照して下さい。

`<Rd>` メモリからワードをロードするデスティネーションレジスタを指定します。

`<Rn>` ベースアドレスを含むレジスタ。

`<immed_5>` `<Rn>` の値に 4 倍して加算される 5 ビットのイミディエート値。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
address = Rn + (immed_5 * 4)
if (CP15_reg1_Ubit == 0)
    if address[1:0] == 0b00 then
        data = Memory[address,4]
    else
        data = UNPREDICTABLE
else    /* CP15_reg1_Ubit == 1 */
    data = Memory[address,4]
Rd = data
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート　データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント　ARMv6 以前は、メモリアドレスがワード境界にアラインしていない場合、メモリから読み出されるデータは予測不能です。アライメントチェック（アドレス [1:0] != 0b00 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定混在エンディアン形式とアライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - CP15_reg1_Ubit == 0 の場合、境界アラインしていないアクセスの結果は予測不能です。
  - CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスがサポートされています。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

等価な ARM 構文とエンコード

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6–2	1	0
1	1	1	0	0	1	0	1	1	0	0	1	Rn	Rd	0	0	0	0	0	immed_5	0	0

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.29 LDR (2)

15	14	13	12	11	10	9	8 6	5 3	2 0
0	1	0	1	1	0	0	Rm	Rn	Rd

LDR (2) は、32 ビットのメモリデータを汎用レジスタにロードします。このアドレシングモードは、ポインタ + 大きなオフセットの演算と、配列のエレメントの 1 つにアクセスするために便利です。

構文

```
LDR  <Rd>, [<Rn>, <Rm>]
```

各項目の説明については以下を参照して下さい。

`<Rd>` メモリからワードをロードするデスティネーションレジスタ。

`<Rn>` メモリアドレスを計算するために使用される最初の値を含むレジスタ。

`<Rm>` メモリアドレスを計算するために使用される 2 番目の値を含むレジスタ。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
address = Rn + Rm
if (CP15_reg1_Ubit == 0)
    if address[1:0] == 0b00 then
        data = Memory[address,4]
    else
        data = UNPREDICTABLE
else    /* CP15_reg1_Ubit == 1 */
    data = Memory[address,4]
Rd = data
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート　データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント　ARMv6 以前は、メモリアドレスがワード境界にアラインしていない場合、メモリから読み出されるデータは予測不能です。アライメントチェック（アドレス [1:0] != 0b00 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定混在エンディアン形式とアライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - — CP15_reg1_Ubit == 0 の場合、境界アラインしていないアクセスの結果は予測不能です。
  - — CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスがサポートされています。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

等価な ARM 構文とエンコード

```
LDR  <Rd>, [<Rn>, <Rm>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	0	1	1	1	1	0	0	1	Rn	Rd	0	0	0	0	0	0	0	0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.30 LDR (3)

15	14	13	12	11	10 8	7 0
0	1	0	0	1	Rd	immed_8

LDR (3) は、32 ビットのメモリデータを汎用レジスタにロードします。このアドレシングモードは、PC 相対データへのアクセスに便利です。

構文

```
LDR  <Rd>, [PC, #<immed_8> * 4]
```

各項目の説明については以下を参照して下さい。

<Rd> メモリからワードをロードするデスティネーションレジスタ。

PC プログラムカウンタ。この値は、メモリアドレスの計算に使用されます。PC の値のビット 1 は、この計算で強制的に 0 にセットされるため、アドレスは必ずワード境界アラインメントになります。

<immed_8> 8ビットの値で、この値は4倍してPCの値に加算され、メモリアドレスが計算されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
address = (PC[31:2] << 2) + (immed_8 * 4)
Rd = Memory[address, 4]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント ARMv6 以前は、メモリアドレスがワード境界にアラインしていない場合、メモリから読み出されるデータは予測不能です。アライメントチェック（アドレス [1:0] != 0b00 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定混在エンディアン形式とアライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - — CP15_reg1_Ubit == 0 の場合、境界アラインしていないアクセスの結果は予測不能です。
  - — CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスがサポートされています。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

等価な ARM 構文とエンコード

完全に等価な命令は存在しません。次の命令が最も等価に近い命令です。

```
LDR  <Rd>, [PC, #<immed_8> * 4]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 … 12	11 10	9 … 2	1 0
1 1 1 0	0 1 0 1 1 0 0 1	1 1 1 1	Rd	0 0	immed_8	0 0

PC の定義と、Thumb 命令では PC のビット [1] が無視されることから、動作が多少異なります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.31 LDR (4)

15	14	13	12	11	10 8	7 0
1	0	0	1	1	Rd	immed_8

LDR (4) は、32 ビットのメモリデータを汎用レジスタにロードします。このアドレシングモードは、スタックデータへのアクセスに便利です。

構文

```
LDR  <Rd>, [SP, #<immed_8> * 4]
```

各項目の説明については以下を参照して下さい。

`<Rd>` メモリからワードをロードするデスティネーションレジスタ。

`SP` スタックポインタ。この値は、メモリアドレスの計算に使用されます。

`<immed_8>` 8 ビットの値で、この値は 4 倍して SP の値に加算され、メモリアドレスが計算されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
address = SP + (immed_8 * 4)
if (CP15_reg1_Ubit == 0)
    if address[1:0] == 0b00 then
        data = Memory[address,4]
    else
        data = UNPREDICTABLE
else    /* CP15_reg1_Ubit == 1 */
    data = Memory[address,4]
Rd = data
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート　データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント　ARMv6 以前は、メモリアドレスがワード境界にアラインしていない場合、メモリから読み出されるデータは予測不能です。アライメントチェック（アドレス [1:0] != 0b00 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定混在エンディアン形式とアライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - — CP15_reg1_Ubit == 0 の場合、境界アラインしていないアクセスの結果は予測不能です。
  - — CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスがサポートされています。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

等価な ARM 構文とエンコード

```
LDR  <Rd>, [SP, #<immed_8> * 4]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 12	11 10	9 2	1 0
1 1 1 0	0 1 0 1 1 0 0 1	1 1 0 1	Rd	0 0	immed_8	0 0

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.32 LDRB (1)

15	14	13	12	11	10 6	5 3	2 0
0	1	1	1	1	immed_5	Rn	Rd

LDRB (1)（レジスタロードバイト）は、バイトをメモリからロードし、32 ビットにゼロ拡張し、結果を汎用レジスタに書き込みます。このアドレシングモードは、構造体（レコード）のフィールドへのアクセスに便利です。オフセットが 0 の場合、生成されるアドレスはベースレジスタ <Rn> の値と同じです。

構文

```
LDRB  <Rd>, [<Rn>, #<immed_5>]
```

各項目の説明については以下を参照して下さい。

<Rd> メモリからバイトをロードするデスティネーションレジスタ。

<Rn> ベースアドレスを含むレジスタ。

<immed_5> 5 ビットの値で、この値は <Rn> の値に加算され、メモリアドレスが計算されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
address = Rn + immed_5
Rd = Memory[address,1]
```

注

データアボート データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

等価な ARM 構文とエンコード

```
LDRB  <Rd>, [<Rn>, #<immed_5>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4 0
1	1	1	0	0	1	0	1	1	1	0	1	Rn	Rd	0	0	0	0	0	0	0	immed_5

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.33 LDRB (2)

15	14	13	12	11	10	9	8 6	5 3	2 0
0	1	0	1	1	1	0	Rm	Rn	Rd

LDRB (2) は、バイトをメモリからロードし、32 ビットにゼロ拡張し、結果を汎用レジスタに書き込みます。このアドレシングモードは、ポインタ + 大きなオフセットの演算と、配列のエレメントの 1 つにアクセスするために便利です。

構文

```
LDRB  <Rd>, [<Rn>, <Rm>]
```

各項目の説明については以下を参照して下さい。

`<Rd>` メモリからバイトをロードするデスティネーションレジスタ。

`<Rn>` メモリアドレスを計算するために使用される最初の値を含むレジスタ。

`<Rm>` メモリアドレスを計算するために使用される 2 番目の値を含むレジスタ。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
address = Rn + Rm
Rd = Memory[address,1]
```

注

データアボート データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

等価な ARM 構文とエンコード

```
LDRB  <Rd>, [<Rn>, <Rm>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	0	1	1	1	1	1	0	1	Rn	Rd	0	0	0	0	0	0	0	0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.34 LDRH (1)

15	14	13	12	11	10 6	5 3	2 0
1	0	0	0	1	immed_5	Rn	Rd

LDRH (1)（レジスタロードハーフワード）は、ハーフワード（16 ビット）をメモリからロードし、32 ビットにゼロ拡張し、結果を汎用レジスタに書き込みます。このアドレシングモードは、構造体（レコード）のフィールドへのアクセスに便利です。オフセットが 0 の場合、生成されるアドレスはベースレジスタ <Rn> の値と同じです。

構文

```
LDRH  <Rd>, [<Rn>, #<immed_5> * 2]
```

各項目の説明については以下を参照して下さい。

<Rd> メモリからハーフワードをロードするデスティネーションレジスタ。

<Rn> ベースアドレスを含むレジスタ。

<immed_5> 5 ビットの値で、この値は 2 倍して <Rn> の値に加算され、メモリアドレスが計算されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
address = Rn + (immed_5 * 2)
if (CP15_reg1_Ubit == 0)
    if address[0] == 0b0 then
        data = Memory[address,2]
    else
    data = UNPREDICTABLE
else    /* CP15_reg1_Ubit == 1 */
    data = Memory[address,2]
Rd = ZeroExtend(data[15:0])
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント ARMv6 以前は、メモリアドレスがハーフワード境界にアラインしていない場合、メモリから読み出されるデータは予測不能です。アライメントチェック（アドレス [0] != 0 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定混在エンディアン形式とアライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - CP15_reg1_Ubit == 0 の場合、境界アラインしていないアクセスの結果は予測不能です。
  - CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスがサポートされています。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

等価な ARM 構文とエンコード

```
LDRH  <Rd>, [<Rn>, #<immed_5> * 2]
```

31	30	29	28	27	26	25	24	23	22	21	10	19 16	15 12	11	10	9 8	7	6	5	4	3 1	0
1	1	1	0	0	0	0	1	1	1	0	1	Rn	Rd	0	0	immed [4:3]	1	0	1	1	immed [2:0]	0

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.35 LDRH (2)

15	14	13	12	11	10	9	8 6	5 3	2 0
0	1	0	1	1	0	1	Rm	Rn	Rd

LDRH (2) は、ハーフワード（16 ビット）をメモリからロードし、32 ビットにゼロ拡張し、結果を汎用レジスタに書き込みます。このアドレシングモードは、ポインタ + 大きなオフセットの演算と、配列のエレメントの 1 つにアクセスするために便利です。

構文

```
LDRH  <Rd>, [<Rn>, <Rm>]
```

各項目の説明については以下を参照して下さい。

<Rd> メモリからハーフワードをロードするデスティネーションレジスタ。

<Rn> メモリアドレスを計算するために使用される最初の値を含むレジスタ。

<Rm> メモリアドレスを計算するために使用される 2 番目の値を含むレジスタ。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
address = Rn + Rm
if (CP15_reg1_Ubit == 0)
    if address[0] == 0b0 then
        data = Memory[address,2]
    else
        data = UNPREDICTABLE
else    /* CP15_reg1_Ubit == 1 */
    data = Memory[address,2]
Rd = ZeroExtend(data[15:0])
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート　データアボートが発生した場合の命令への影響の詳細については、P. A2-21「*データアボートが発生した命令の影響*」を参照して下さい。

アライメント　ARMv6 以前は、メモリアドレスがハーフワード境界にアラインしていない場合、メモリから読み出されるデータは予測不能です。アライメントチェック（アドレス [0] != 0 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定混在エンディアン形式とアライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - — CP15_reg1_Ubit == 0 の場合、境界アラインしていないアクセスの結果は予測不能です。
  - — CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスがサポートされています。

エンディアン形式とアライメントの詳細については、P. A2-30「*エンディアンのサポート*」と P. A2-38「*アンアラインドアクセスのサポート*」を参照して下さい。

等価な ARM 構文とエンコード

```
LDRH  <Rd>, [<Rn>, <Rm>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
1	1	1	0	0	0	0	1	1	0	0	1	Rn	Rd	SBZ	1	0	1	1	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.36 LDRSB

15	14	13	12	11	10	9	8 6	5 3	2 0
0	1	0	1	0	1	1	Rm	Rn	Rd

LDRSB（レジスタロード符号付きバイト）は、バイトをメモリからロードし、32 ビットに符号拡張し、結果を汎用レジスタに書き込みます。

構文

```
LDRSB  <Rd>, [<Rn>, <Rm>]
```

各項目の説明については以下を参照して下さい。

<Rd> メモリからバイトをロードするデスティネーションレジスタ。

<Rn> メモリアドレスを計算するために使用される最初の値を含むレジスタ。

<Rm> メモリアドレスを計算するために使用される 2 番目の値を含むレジスタ。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
address = Rn + Rm
Rd = SignExtend(Memory[address,1])
```

注

データアボート データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

等価な ARM 構文とエンコード

```
LDRSB  <Rd>, [<Rn>, <Rm>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
1	1	1	0	0	0	0	1	1	0	0	1	Rn	Rd	SBZ	1	1	0	1	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.37 LDRSH

15	14	13	12	11	10	9	8 6	5 3	2 0
0	1	0	1	1	1	1	Rm	Rn	Rd

LDRSH（レジスタロード符号付きハーフワード）は、ハーフワードをメモリからロードし、32 ビットに符号拡張し、結果を汎用レジスタに書き込みます。

構文

```
LDRSH  <Rd>, [<Rn>, <Rm>]
```

各項目の説明については以下を参照して下さい。

<Rd> メモリからハーフワードをロードするデスティネーションレジスタ。

<Rn> メモリアドレスを計算するために使用される最初の値を含むレジスタ。

<Rm> メモリアドレスを計算するために使用される 2 番目の値を含むレジスタ。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
address = Rn + Rm
if (CP15_reg1_Ubit == 0)
    if address[0] == 0b0 then
        data = Memory[address,2]
    else
        data = UNPREDICTABLE
else    /* CP15_reg1_Ubit == 1 */
    data = Memory[address,2]
Rd = SignExtend(data[15:0])
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート　データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント　ARMv6 以前は、メモリアドレスがハーフワード境界にアラインしていない場合、メモリから読み出されるデータは予測不能です。アライメントチェック（アドレス [0] != 0 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定混在エンディアン形式とアライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - CP15_reg1_Ubit == 0 の場合、境界アラインしていないアクセスの結果は予測不能です。
  - CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスがサポートされています。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

等価な ARM 構文とエンコード

```
LDRSH  <Rd>, [<Rn>, <Rm>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
1	1	1	0	0	0	0	1	1	0	0	1	Rn	Rd	SBZ	1	1	1	1	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.38 LSL (1)

15	14	13	12	11	10 6	5 3	2 0
0	0	0	0	0	immed_5	Rm	Rd

LSL (1)（論理左シフト）は、レジスタの内容に 2 の累乗（定数）を乗算します。シフトにより空いたビット位置には 0 が挿入され、結果に基づいて条件コードフラグが更新されます。

構文

```
LSL  <Rd>, <Rm>, #<immed_5>
```

各項目の説明については以下を参照して下さい。

<Rd> 演算の結果をストアするレジスタ。

<Rm> シフトされる値を含むレジスタ。

<immed_5> シフトするビット数を 0 ～ 31 の範囲で指定します。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
if immed_5 == 0
    C Flag = unaffected
    Rd = Rm
else /* immed_5 > 0 */
    C Flag = Rm[32 - immed_5]
    Rd = Rm Logical_Shift_Left immed_5
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
MOVS  <Rd>, <Rm>, LSL #<immed_5>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 7	6 5 4	3 0
1 1 1 0	0 0 0 1 1 0 1 1	SBZ	Rd	immed_5	0 0 0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.39 LSL (2)

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	0	0	1	0	Rs	Rd

LSL (2)（論理左シフト）は、レジスタの内容に 2 の累乗（変数）を乗算します。空いたビット位置には 0 が挿入されます。

結果に基づいて条件コードフラグを更新します。

構文

```
LSL  <Rd>, <Rs>
```

各項目の説明については以下を参照して下さい。

<Rd> シフトされる値を保持し、演算のデスティネーションレジスタになります。

<Rs> シフトするビット数を保持するレジスタ。値は最下位バイトに保持されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
if Rs[7:0] == 0
    C Flag = unaffected
    Rd = unaffected
else if Rs[7:0] < 32 then
    C Flag = Rd[32 - Rs[7:0]]
    Rd = Rd Logical_Shift_Left Rs[7:0]
else if Rs[7:0] == 32 then
    C Flag = Rd[0]
    Rd = 0
else /* Rs[7:0] > 32 */
    C Flag = 0
    Rd = 0
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
MOVS  <Rd>, <Rd>, LSL <Rs>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 - 16	15 - 12	11 - 8	7	6	5	4	3 - 0
1	1	1	0	0	0	0	1	1	0	1	1	SBZ	Rd	Rs	0	0	0	1	Rd

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.40 LSR (1)

15	14	13	12	11	10 … 6	5 … 3	2 … 0
0	0	0	0	1	immed_5	Rm	Rd

LSR (1)（論理右シフト）は、レジスタの符号なし数値を 2 の累乗（定数）で除算します。空いたビット位置には 0 が挿入されます。

結果に基づいて条件コードフラグを更新します。

構文

```
LSR  <Rd>, <Rm>, #<immed_5>
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタ。

<Rm> シフトされる値を含むレジスタ。

<immed_5> シフトするビット数を 1 ～ 32 の範囲で指定します。1 ～ 31 ビットのシフトは、immed_5 に直接エンコードされます。32 ビットのシフトは immed_5 == 0 としてエンコードされます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
if immed_5 == 0
    C Flag = Rm[31]
    Rd = 0
else /* immed_5 > 0 */
    C Flag = Rm[immed_5 - 1]
    Rd = Rm Logical_Shift_Right immed_5
N Flag = Rd[31] /* 0b0 */
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
MOVS  <Rd>, <Rm>, LSR #<immed_5>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 ... 16	15 ... 12	11 ... 7	6	5	4	3 ... 0
1	1	1	0	0	0	0	1	1	0	1	1	SBZ	Rd	immed_5	0	1	0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.41 LSR (2)

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	0	0	1	1	Rs	Rd

LSR (2) は、レジスタの符号なし数値を 2 の累乗（変数）で除算します。空いたビット位置には 0 が挿入されます。

結果に基づいて条件コードフラグを更新します。

構文

```
LSR  <Rd>, <Rs>
```

各項目の説明については以下を参照して下さい。

<Rd> シフトされる値を保持し、演算のデスティネーションレジスタになります。

<Rs> シフトするビット数を保持するレジスタ。値は最下位バイトに保持されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
if Rs[7:0] == 0 then
    C Flag = unaffected
    Rd = unaffected
else if Rs[7:0] < 32 then
    C Flag = Rd[Rs[7:0] - 1]
    Rd = Rd Logical_Shift_Right Rs[7:0]
else if Rs[7:0] == 32 then
    C Flag = Rd[31]
    Rd = 0
else /* Rs[7:0] > 32 */
    C Flag = 0
    Rd = 0
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
MOVS  <Rd>, <Rd>, LSR <Rs>
```

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
1	1	1	0	0	0	0	1	1	0	1	1	SBZ	Rd	Rs	0	0	1	1	Rd

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.42 MOV (1)

15	14	13	12	11	10 8	7 0
0	0	1	0	0	Rd	immed_8

MOV (1)（移動）は、大きなイミディエート値をレジスタに移動します。

結果に基づいて条件コードフラグを更新します。

構文

```
MOV  <Rd>, #<immed_8>
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタ。

<immed_8> 8 ビットイミディエート値（範囲は 0 ～ 255）で、<Rd> に移動されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = immed_8
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

等価な ARM 構文とエンコード

```
MOVS  <Rd>, #<immed_8>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7 0
1	1	1	0	0	0	1	1	1	0	1	1	SBZ	Rd	0	0	0	0	immed_8

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.43 MOV (2)

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	0	0	1	1	1	0	0	0	0	Rn	Rd

MOV (2) は、下位レジスタの値を別の下位レジスタに移動します。

値に基づいて条件コードフラグを更新します。

構文

```
MOV  <Rd>, <Rn>
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタ。

<Rn> コピーされる値を含むレジスタ。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rn
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = 0
V Flag = 0
```

注

エンコード　この命令は ADD Rd, Rn #0 としてエンコードされます。
詳細については、P. A7-5「*ADD (1)*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
ADDS  <Rd>, <Rn>, #0
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	0	1	0	0	1	Rn	Rd	0	0	0	0	0	0	0	0	0	0	0	0

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.44 MOV (3)

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	1	1	0	H1	H2	Rm	Rd

`MOV` (3) は、上位レジスタに、上位レジスタから、または上位レジスタ間で値を移動します。

下位レジスタのみを使用する `MOV` 命令（P. A7-73「*MOV (2)*」参照）とは異なり、この命令ではフラグが変更されません。

構文

```
MOV   <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<Rd>` 演算のデスティネーションレジスタ。R0 ～ R15 の任意のレジスタで、レジスタ番号は命令の H1（最上位ビット）と Rd（他の 3 ビット）にエンコードされます。

`<Rm>` コピーされる値を含むレジスタ。R0 ～ R15 の任意のレジスタで、レジスタ番号は命令の H2（最上位ビット）と Rm（他の 3 ビット）にエンコードされます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rm
```

用法

呼び出し元が Thumb ルーチンであることが確実ならば、`MOV PC,R14` を使用してサブルーチンから復帰できます。ただし、`BX R14` を使用することを強くお勧めします（P. A7-32「*BX*」参照）。`BX R14` 命令は、呼び出し元が ARM ルーチンか Thumb ルーチンかにかかわらず動作し、一部のプロセッサではパフォーマンスの利点もあります。

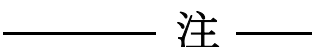
 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

アセンブラ構文 `<Rd>` と `<Rm>` の両方に下位レジスタを指定した場合、アセンブラ構文 `MOV <Rd>, <Rm>` は P. A7-73 で説明されている `MOV` (2) 命令にアセンブルされます。

両方のレジスタが下位の場合

エンコードで H1 == 0 かつ H2 == 0 となる場合、命令は下位レジスタから下位レジスタへの、フラグを設定しないコピーになります。この命令を `MOV` 構文を使用して記述することはできません。これは `MOV <Rd>, <Rm>` はフラグを設定するコピーを生成するためです。ただし、`CPY` ニーモニックを使用して記述することはできます。詳細については、P. A7-41「*CPY*」を参照して下さい。



注

ARMv6 以前では、<Rd> と <Rm> の両方に下位レジスタを指定した場合（H1 == 0 かつ H2 == 0）、結果は予測不能です。

等価な ARM 構文とエンコード

完全に等価な命令は存在しません。次の命令が最も等価に近い命令です。

```
MOV  <Rd>, <Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15	14 12	11 10 9 8 7 6 5 4	3	2 0
1 1 1 0	0 0 0 1 1 0 1 0	SBZ	H1	Rd	0 0 0 0 0 0 0 0	H2	Rm

ARM コードと Thumb コードを実行しているときの PC の定義が異なるため、命令による PC へのアクセスに多少の相違があります。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.45 MUL

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	1	1	0	1	Rm	Rd

MUL（乗算）は、符号付きまたは符号なし変数を乗算し、32 ビットの結果を生成します。

MUL は、結果に基づいて条件コードフラグを更新します。

構文

```
MUL  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> <Rm> の値と掛け合わせられる値を保持し、演算のデスティネーションレジスタになります。

<Rm> <Rd> の値と掛け合わせられる値を保持するレジスタ。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = (Rm * Rd)[31:0]
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected    /* See "C flag" note */
V Flag = unaffected
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

短縮処理	乗算器の実装が短縮処理をサポートしている場合、`<Rd>` オペランドの値について実装されている必要があります。使用される短縮処理のタイプ（符号付きまたは符号なし）は実装定義です。
符号付きと符号なし	`MUL` は 64 ビットの積のうち下位 32 ビットのみを生成するため、`MUL` の結果は符号付きと符号なしのどちらの数値についても同じです。
C フラグ	ARMv5 以降では、`MUL` 命令は C フラグを変更しません。以前のバージョンのアーキテクチャでは、`MUL` 命令の実行後の C フラグの値は予測不能です。
オペランドの制限	ARMv6 以前では、`<Rd>` と `<Rm>` の両方に同じレジスタを指定した場合の結果は予測不能です。

等価な ARM 構文とエンコード

```
MULS  <Rd>, <Rm>, <Rd>
```

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
1	1	1	0	0	0	0	0	0	0	0	1	Rd	SBZ	Rd	1	0	0	1	Rm

注

次の命令は ARM 命令のオペランド制限に違反している（P. A4-81「*MUL*」参照）ため適切な代替命令ではなく、誤った短縮処理動作が行われる可能性があります。

```
MULS  <Rd>, <Rd>, <Rm>
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.46 MVN

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	1	1	1	1	Rm	Rd

MVN（論理否定）はレジスタの値の 1 の補数を計算します。これは多くの場合、ビットマスクの計算に使用されます。

MVN は、結果に基づいて条件コードフラグを更新します。

構文

```
MVN  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタ。

<Rm> このレジスタに含まれている値の 1 の補数が、<Rd> に書き込まれます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = NOT Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

等価な ARM 構文とエンコード

```
MVNS  <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	0	0	0	1	1	1	1	1	SBZ	Rd	0	0	0	0	0	0	0	0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.47 NEG

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	1	0	0	1	Rm	Rd

NEG（符号反転）は、レジスタの値の 2 の補数を計算し、結果を 2 番目のレジスタにストアします。

NEG は、結果に基づいて条件コードフラグを更新します。

構文

```
NEG  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタ。

<Rm> このレジスタの値が 0 から減算されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = 0 - Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = NOT BorrowFrom(0 - Rm)
V Flag = OverflowFrom(0 - Rm)
```

等価な ARM 構文とエンコード

```
RSBS  <Rd>, <Rm>, #0
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	0	0	1	1	1	Rm	Rd	0	0	0	0	0	0	0	0	0	0	0	0

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.48 ORR

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	1	1	0	0	Rm	Rd

ORR（論理 OR）は、2 つのレジスタの値に対してビット単位の OR を実行します。

ORR は、結果に基づいて条件コードフラグを更新します。

構文

```
ORR  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタ。

<Rm> このレジスタの値と、<Rd> の値との OR が計算されます。この演算はビット単位の論理和です。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rd OR Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

等価な ARM 構文とエンコード

```
ORRS  <Rd>, <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	0	0	0	1	1	0	0	1	Rd	Rd	0	0	0	0	0	0	0	0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.49 POP

15	14	13	12	11	10	9	8	7 ... 0
1	0	1	1	1	1	0	R	register_list

POP（複数レジスタポップ）は、汎用レジスタ R0 ～ R7 と PC のサブセットまたは全部を、スタックからロードします。

ロードされる汎用レジスタには PC を含めることができます。その場合、PC にロードされるワードはアドレスと見なされ、そのアドレスに分岐が発生します。ARMv5 以降では、ロードされる値のビット [0] により、この分岐の後の実行が ARM ステートと Thumb ステートのどちらで行われるかが決定されます。これは、次の命令が実行されたのと同様です。

```
BX (loaded_value)
```

ARMv4 の T バリエーションでは、ロードされる値のビット [0] は無視され、実行は Thumb ステートで行われます。これは、次の命令が実行されたのと同様です。

```
MOV PC,(loaded_value)
```

構文

```
POP  <registers>
```

各項目の説明については以下を参照して下さい。

<registers> ロードされるレジスタの一覧で、カンマで区切って並べ、{ と } で囲みます。一覧は命令の register_list フィールドにエンコードされます。0 ～ 7 までのそれぞれの i について、レジスタ Ri が一覧に含まれる場合はビット [i] が 1 に、そうでない場合は 0 にセットされます。PC が一覧に含まれている場合は R ビット（ビット [8]）は 1 に、それ以外の場合は 0 に設定されます。

最低 1 つのレジスタをロードする必要があります。ビット [8:0] がすべて 0 の場合、結果は予測不能です。

レジスタは最下位のメモリアドレス（start_address）から最も低い番号のレジスタに、その後で順に最上位のメモリアドレス（end_address）から最も高い番号のレジスタへの順序でロードされます。レジスタリストに PC が指定されている場合（オペコードのビット [8] がセットされている）、PC にロードされたアドレス（データ）に分岐が実行されます。

<start_address> は SP の値です。

以降のアドレスは、前のアドレスに 4 を加算して構成されます。<registers> に指定されている各レジスタについて 1 つのアドレスが生成されます。

end_address の値は、SP の値に <registers> で指定されたレジスタ数の 4 倍を足した値 - 4 です。

SP レジスタは、<registers> で指定されているレジスタ数の 4 倍だけインクリメントされます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
start_address = SP
end_address = SP + 4*(R + Number_Of_Set_Bits_In(register_list))
address = start_address

for i = 0 to 7
    if register_list[i] == 1 then
        Ri = Memory[address,4]
        address = address + 4

if R == 1 then
    value = Memory[address,4]
    PC = value AND 0xFFFFFFFE
    if (architecture version 5 or above) then
        T Bit = value[0]
    address = address + 4

assert end_address = address
SP = end_address
```

用法

POP はスタック操作に使用します。POP 命令のレジスタリストに PC を含めて、プロシージャのイグジット（終了）を効率的に実行できます。これは、1 つの命令で保存したレジスタの復元、PC への復帰アドレスのロード、スタックポインタの更新を実行できるためです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート　データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

CPSR　POP 命令では、CPSR の T ビットのみが更新されます。他のビットはいずれも変更されません。

アライメント　実装にシステム制御コプロセッサが含まれており（第 B3 章 *「システム制御コプロセッサ」* 参照）、アライメントチェックが許可されている場合、ビット [1:0] != 0b00 のアドレスはアライメント例外を発生します。

ARMv6 から、アライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - CP15_reg1_Ubit == 0 の場合、アドレスの下位 2 ビットが命令により無視されます。
  - CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

ARM と Thumb の状態遷移

ARM アーキテクチャ 5 以降では、ARM 状態でワード境界アラインしていないアドレスへの分岐は不可能なため、R15 にロードされる値のビット [1:0] が 0b10 の場合、結果は予測不能です。

時間順序　この命令によって生成される、メモリの個別のワードに対するアクセスの時間順序は、一部の条件でのみ定義されます。詳細については、P. B2-13 *「メモリアクセスの制限」*を参照して下さい。

等価な ARM 構文とエンコード

```
LDMIA  SP!, <registers>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7 … 0
1	1	1	0	1	0	0	0	1	0	1	1	1	1	0	1	R	0	0	0	0	0	0	0	register_list

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.50 PUSH

15	14	13	12	11	10	9	8	7 … 0
1	0	1	1	0	1	0	R	register_list

PUSH（複数レジスタプッシュ）は汎用レジスタ R0 〜 R7 と LR のサブセットまたは全部を、スタックにプッシュします。

構文

```
PUSH  <registers>
```

各項目の説明については以下を参照して下さい。

`<registers>` ストアされるレジスタの一覧で、カンマで区切って並べ、{ と } で囲みます。一覧は命令の register_list フィールドにエンコードされます。0 〜 7 までのそれぞれの i について、レジスタ Ri が一覧に含まれる場合はビット [i] が 1 に、そうでない場合は 0 にセットされます。LR が一覧に含まれている場合は R ビット（ビット [8]）は 1 に、それ以外の場合は 0 に設定されます。

最低 1 つのレジスタをロードする必要があります。ビット [8:0] がすべて 0 の場合、結果は予測不能です。

レジスタは最も低い番号のレジスタから最下位のメモリアドレス（`start_address`）に、その後で順に最も高い番号のレジスタから最上位のメモリアドレス（`end_address`）への順序でストアされます。

`start_address` は、（SP の値 - ストアするレジスタ数の * 4）です。

以降のアドレスは、前のアドレスに 4 を加算して構成されます。`<registers>` に指定されている各レジスタについて 1 つのアドレスが生成されます。

`end_address` の値は、（SP の元の値 - 4）です。

SP レジスタは、`<registers>` で指定されているレジスタ数の 4 倍だけデクリメントされます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
start_address = SP - 4*(R + Number_Of_Set_Bits_In(register_list))
end_address = SP - 4
address = start_address
for i = 0 to 7
    if register_list[i] == 1
        Memory[address,4] = Ri
        address = address + 4
if R == 1
    Memory[address,4] = LR
    address = address + 4
assert end_address == address - 4
SP = SP - 4*(R + Number_Of_Set_Bits_In(register_list))
if (CP15_reg1_Ubit == 1)        /* ARMv6 */
    if (Shared(Address))
        physical_address = TLB(Address)
        ClearByAddress(physical_address, size)
```

共有メモリと同期化基本命令の詳細については、P. A2-44 *「同期化基本命令」*を参照して下さい。

用法

PUSH はスタック操作に使用します。PUSH 命令のレジスタリストに LR を含めて、プロシージャのエントリを効率的に実行できます。これは、1 つの命令でレジスタのスタックへの保存（復帰アドレスを含む）とスタックポインタの更新を実行できるためです。後で、対応する POP 命令を使用して、プロシージャから復帰できます。

注

データアボート データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント PUSH 命令では、address の下位 2 ビットが無視されます。

実装にシステム制御コプロセッサが含まれており（第 B3 章 *「システム制御コプロセッサ」* 参照）、アライメントチェックが有効な場合、ビット [1:0] != 0b00 のアドレスはアライメント例外を発生します。

ARMv6 から、アライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - CP15_reg1_Ubit == 0 の場合、アドレスの下位 2 ビットが命令により無視されます。
  - CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

エンディアン形式とアライメントの詳細については、P. A2-30「*エンディアンのサポート*」と P. A2-38「*アンアラインドアクセスのサポート*」を参照して下さい。

時間順序

この命令によって生成される、メモリの個別のワードに対するアクセスの時間順序は、一部の条件でのみ定義されます。詳細については、P. B2-13「*メモリアクセスの制限*」を参照して下さい。

等価な ARM 構文とエンコード

```
STMDB  SP!, <registers>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7 ... 0
1	1	1	0	1	0	0	1	0	0	1	0	1	1	0	1	0	R	0	0	0	0	0	0	register_list

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.51 REV

15–12				11–8				7	6	5–3	2–0
1	0	1	1	1	0	1	0	0	0	Rn	Rd

REV (バイト反転ワード) は、32 ビットレジスタのバイト順序を反転します。フラグは変更されません。

構文

```
REV Rd, Rn
```

各項目の説明については以下を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<Rn> オペランドレジスタを指定します。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
Rd[31:24] = Rn[ 7: 0]
Rd[23:16] = Rn[15: 8]
Rd[15: 8] = Rn[23:16]
Rd[ 7: 0] = Rn[31:24]
```

用法

REV は、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、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するために使用します。

等価な ARM 構文とエンコード

```
REV Rd, Rm
```

31–28				27–23					22	21	20	19–16	15–12	11–8	7	6–4			3–0
1	1	1	0	0	1	1	0	1	0	1	1	SBO	Rd	SBO	0	0	1	1	Rm

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.52 REV16

15			12	11			8	7	6	5	3	2	0
1	0	1	1	1	0	1	0	0	1	Rn		Rd	

REV16（バイト反転パックハーフワード）は、32 ビットレジスタの各 16 ビットハーフワードのバイト順序を反転します。フラグは変更されません。

構文

```
REV16 Rd, Rn
```

各項目の説明については以下を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<Rn> オペランドレジスタを指定します。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
Rd[15: 8] = Rn[ 7: 0]
Rd[ 7: 0] = Rn[15: 8]
Rd[31:24] = Rn[23:16]
Rd[23:16] = Rn[31:24]
```

用法

REV16 は、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、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するために使用します。

等価な ARM 構文とエンコード

```
REV16 Rd, Rm
```

31			28	27				23	22	21	20	19 16	15 12	11 8	7	6		4	3 0
1	1	1	0	0	1	1	0	1	0	1	1	SBO	Rd	SBO	1	0	1	1	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.53 REVSH

15			12	11			8	7	6	5	3	2	0
1	0	1	1	1	0	1	0	1	1	Rn		Rd	

REVSH（バイト反転符号付きハーフワード）は、32 ビットレジスタの下位 16 ビットハーフワードのバイト順序を反転し、結果を 32 ビットに符号拡張します。フラグは変更されません。

構文

```
REVSH Rd, Rn
```

各項目の説明については以下を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<Rn> オペランドレジスタを指定します。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
Rd[15: 8] = Rn[ 7: 0]
Rd[ 7: 0] = Rn[15: 8]
if Rn[7] == 1 then
    Rd[31:16] = 0xFFFF
else
    Rd[31:16] = 0x0000
```

用法

REVSH は、次のような変換に使用します。

- 16 ビット符号付きのビッグエンディアンデータを、32 ビット符号付きのリトルエンディアンデータに
- 16 ビット符号付きのリトルエンディアンデータを、32 ビット符号付きのビッグエンディアンデータに

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
REVSH Rd, Rm
```

31-28	27-23	22	21-20	19-16	15-12	11-8	7	6-4	3-0
1 1 1 0	0 1 1 0 1	1	1 1	SBO	Rd	SBO	1	0 1 1	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.54 ROR

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	0	1	1	1	Rs	Rd

ROR（レジスタ右ローテート）は、レジスタの値を変数分だけローテートします。ローテートによって右端からはみ出したビットは、左側の空いたビット位置に挿入されます。

ROR は、結果に基づいて条件コードフラグを更新します。

構文

```
ROR  <Rd>, <Rs>
```

各項目の説明については以下を参照して下さい。

<Rd> ローテートが実行される値を含み、演算のデスティネーションレジスタになります。

<Rs> <Rd> の値に適用されるローテートを保持するレジスタ。ローテートするビット数は最下位バイトに保持されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
if Rs[7:0] == 0 then
    C Flag = unaffected
    Rd = unaffected
else if Rs[4:0] == 0 then
    C Flag = Rd[31]
    Rd = unaffected
else /* Rs[4:0] > 0 */
    C Flag = Rd[Rs[4:0] - 1]
    Rd = Rd Rotate_Right Rs[4:0]
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
MOVS  <Rd>, <Rd>, ROR <Rs>
```

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
1	1	1	0	0	0	0	1	1	0	1	1	SBZ	Rd	Rs	0	1	1	1	Rd

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.55 SBC

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	0	1	1	0	Rm	Rd

SBC（キャリー付き減算）は、最初のオペランドの値から、2 番目のオペランドの値と NOT（キャリーフラグ）を減算します。

SBC は、結果に基づいて条件コードフラグを更新します。

SBC を使用して複数ワードの減算を合成します。

構文

```
SBC  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> 減算の最初のオペランドを保持し、演算のデスティネーションレジスタになります。

<Rm> <Rd> から減算される値を保持します。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rd - Rm - NOT(C Flag)
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rd - Rm - NOT(C Flag))
V Flag = OverflowFrom(Rd - Rm - NOT(C Flag))
```

等価な ARM 構文とエンコード

```
SBCS  <Rd>, <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	0	0	0	0	1	1	0	1	Rd	Rd	0	0	0	0	0	0	0	0	Rm

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.56 SETEND

15	14	13	12	11	10	9	8	7	6	5	4	3	2-0
1	0	1	1	0	1	1	0	0	1	0	1	E	SBZ

SETEND は CPSR の E ビットを変更します。CPSR の他のビットは変更されません。

構文

```
SETEND <endian_specifier>
```

各項目の説明については以下を参照して下さい。

<endian_specifier>

次のいずれかです。

BE　命令の E ビットをセットします。これにより、CPSR の E ビットがセットされます。

LE　命令の E ビットをクリアします。これにより、CPSR の E ビットがクリアされます。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
CPSR = CPSR with specified E bit modification
```

用法

SETEND は、データアクセスのバイト順序を変更するために使用します。SETEND を使用して、リトルエンディアンを主に使用するアプリケーションの中に一連のビッグエンディアンデータフィールドがある場合や、ビッグエンディアンを主に使用するアプリケーションの中に一連のリトルエンディアンデータフィールドがある場合に、アクセスの効率を改善できます。詳細については、P. A2-30「*エンディアンのサポート*」を参照して下さい。

等価な ARM 構文とエンコード

```
SETEND <endian_specifier>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15-10	9	8	7	6	5	4	3-0
1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	SBZ	E	SBZ	0	0	0	0	SBZ

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.57 STMIA

15	14	13	12	11	10 8	7 0
1	1	0	0	0	Rn	register_list

STMIA（複数ストアポストインクリメント）は汎用レジスタの空でないサブセットまたは全部を、連続したメモリロケーションにストアします。

構文

```
STMIA  <Rn>!, <registers>
```

各項目の説明については以下を参照して下さい。

`<Rn>` 命令の開始アドレスを含むレジスタ。

`!` ベースレジスタのライトバックを指定します。オプションではありません。

`<registers>` ストアされるレジスタの一覧で、カンマで区切って並べ、{ と } で囲みます。一覧は命令の register_list フィールドにエンコードされます。0 ～ 7 までのそれぞれの i について、レジスタ Ri が一覧に含まれる場合はビット [i] が 1 に、そうでない場合は 0 にセットされます。

最低 1 つのレジスタをロードする必要があります。ビット [7:0] がすべて 0 の場合、結果は予測不能です。

レジスタは最も低い番号のレジスタから最下位のメモリアドレス（`start_address`）に、その後で順に最も高い番号のレジスタから最上位のメモリアドレス（`end_address`）への順序でストアされます。

`start_address` はベースレジスタ `<Rn>` の値です。以降のアドレスは、前のアドレスに 4 を加算して構成されます。`<registers>` に指定されている各レジスタについて 1 つのアドレスが生成されます。

`end_address` の値は、ベースレジスタの値に `<registers>` で指定されたレジスタ数の 4 倍を足した値 - 4 です。

最後に、ベースレジスタ `<Rn>` は、`<registers>` で指定されているレジスタ数の 4 倍だけインクリメントされます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
start_address = Rn
end_address = Rn + (Number_Of_Set_Bits_In(register_list) * 4) - 4
address = start_address
for i = 0 to 7
    if register_list[i] == 1
        Memory[address,4] = Ri
        address = address + 4
assert end_address == address - 4
Rn = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
if (CP15_reg1_Ubit == 1)        /* ARMv6 */
    if (Shared(Address))
        physical_address = TLB(Address)
        ClearByAddress(physical_address, size)
```

共有メモリと同期化基本命令の詳細については、P. A2-44 *「同期化基本命令」*を参照して下さい。

用法

`STMIA` は、ブロックストア命令として有用です。`LDMIA`（複数ロード）と組み合わせて、効率的なブロックコピーが実行できます。

注

オペランドの制限

`<registers>` に <Rn> が指定されている場合、次の動作が行われます。

- <Rn> が、`<registers>` に指定されている中で最も番号の小さいレジスタの場合、<Rn> の元の値がストアされます。
- それ以外の場合、ストアされる <Rn> の値は予測不能です。

データアボート　データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント　複数ストア命令では、`address` の下位 2 ビットが無視されます。

実装にシステム制御コプロセッサが含まれており（第 B3 章*「システム制御コプロセッサ」*参照）、アライメントチェックが有効な場合、ビット [1:0] != 0b00 のアドレスはアライメント例外を発生します。

ARMv6 から、アライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  — CP15_reg1_Ubit == 0 の場合、アドレスの下位 2 ビットが命令により無視されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

— CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。

エンディアン形式とアライメントの詳細については、P. A2-30*「エンディアンのサポート」*と P. A2-38*「アンアラインドアクセスのサポート」*を参照して下さい。

時間順序 この命令によって生成される、メモリの個別のワードに対するアクセスの時間順序は、一部の条件でのみ定義されます。詳細については、P. B2-13*「メモリアクセスの制限」*を参照して下さい。

等価な ARM 構文とエンコード

```
STMIA  <Rn>!, <registers>
```

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9	8	7–0
1	1	1	0	1	0	0	0	1	0	1	0	Rn	0	0	0	0	0	0	0	0	register_list

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.58 STR (1)

15	14	13	12	11	10 6	5 3	2 0
0	1	1	0	0	immed_5	Rn	Rd

STR (1)（レジスタストア）は、32 ビットのデータを汎用レジスタからメモリにストアします。このアドレシングモードは、構造体（レコード）のフィールドへのアクセスに便利です。オフセットが 0 の場合、生成されるアドレスはベースレジスタ <Rn> の値と同じです。

構文

```
STR  <Rd>, [<Rn>, #<immed_5> * 4]
```

各項目の説明については以下を参照して下さい。

<Rd> メモリにストアされるワードを含むレジスタ。

<Rn> ベースアドレスを含むレジスタ。

<immed_5> 5ビットの値で、この値を4倍して<Rn>の値に加算し、メモリアドレスが計算されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
address = Rn + (immed_5 * 4)
if (CP15_reg1_Ubit == 0)
    if address[1:0] == 0b00 then
        Memory[address,4] = Rd
    else
        Memory[address,4] = UNPREDICTABLE
else                                /* CP15_reg1_Ubit == 1 */
    Memory[address,4] = Rd
    if (Shared(Address))            /* from ARMv6 */
        physical_address = TLB(Address)
        ClearByAddress(physical_address, size)
```

共有メモリと同期化基本命令の詳細については、P. A2-44*「同期化基本命令」*を参照して下さい。

注

データアボート　データアボートが発生した場合の命令への影響の詳細については、P. A2-21*「データアボートが発生した命令の影響」*を参照して下さい。

アライメント　ARMv6 以前は、メモリアドレスがワード境界にアラインしていない場合、命令の動作は予測不能です。アライメントチェック（アドレス [1:0] != 0b00 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定混在エンディアン形式とアライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - CP15_reg1_Ubit == 0 の場合、境界アラインしていないアクセスの結果は予測不能です。
  - CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスがサポートされています。

エンディアン形式とアライメントの詳細については、P. A2-30*「エンディアンのサポート」*と P. A2-38*「アンアラインドアクセスのサポート」*を参照して下さい。

等価な ARM 構文とエンコード

```
STR  <Rd>, [<Rn>, #<immed_5> * 4]
```

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6–2	1	0
1	1	1	0	0	1	0	1	1	0	0	0	Rn	Rd	0	0	0	0	0	immed_5	0	0

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.59 STR (2)

15	14	13	12	11	10	9	8 6	5 3	2 0
0	1	0	1	0	0	0	Rm	Rn	Rd

STR (2) は、32 ビットのデータを汎用レジスタからメモリにストアします。このアドレシングモードは、ポインタ + 大きなオフセットの演算と、配列のエレメントの 1 つにアクセスするために便利です。

構文

```
STR  <Rd>, [<Rn>, <Rm>]
```

各項目の説明については以下を参照して下さい。

<Rd> メモリにストアされるワードを含むレジスタ。

<Rn> メモリアドレスを計算するために使用される最初の値を含むレジスタ。

<Rm> メモリアドレスを計算するために使用される 2 番目の値を含むレジスタ。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
address = Rn + Rm
if (CP15_reg1_Ubit == 0)
    if address[1:0] == 0b00 then
        Memory[address,4] = Rd
    else
        Memory[address,4] = UNPREDICTABLE
else    /* CP15_reg1_Ubit == 1 */
    Memory[address,4] = Rd
    if (Shared(Address))
        physical_address = TLB(Address)
        ClearByAddress(physical_address, size)
```

共有メモリと同期化基本命令の詳細については、P. A2-44 *「同期化基本命令」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント ARMv6 以前は、メモリアドレスがワード境界にアラインしていない場合、命令の動作は予測不能です。アライメントチェック（アドレス [1:0] != 0b00 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定混在エンディアン形式とアライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - — CP15_reg1_Ubit == 0 の場合、境界アラインしていないアクセスの結果は予測不能です。
  - — CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスがサポートされています。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

等価な ARM 構文とエンコード

```
STR  <Rd>, [<Rn>, <Rm>]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
1 1 1 0	0 1 1 1 1 0 0 0	Rn	Rd	0 0 0 0 0 0 0 0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.60 STR (3)

15	14	13	12	11	10 8	7 0
1	0	0	1	0	Rd	immed_8

STR (3) は、32 ビットのデータを汎用レジスタからメモリにストアします。このアドレシングモードは、スタックデータへのアクセスに便利です。この場合、STR はレジスタ <Rd> からメモリに値をストアします。

構文

```
STR  <Rd>, [SP, #<immed_8> * 4]
```

各項目の説明については以下を参照して下さい。

<Rd> メモリにストアされるワードを含むレジスタ。

SP スタックポインタ。この値は、メモリアドレスの計算に使用されます。

<immed_8> 8 ビットの値で、この値は 4 倍して SP の値に加算され、メモリアドレスが計算されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
address = SP + (immed_8 * 4)
if (CP15_reg1_Ubit == 0)
    if address[1:0] == 0b00 then
        Memory[address,4] = Rd
    else
    Memory[address,4] = UNPREDICTABLE
else     /* CP15_reg1_Ubit == 1 */
    Memory[address,4] = Rd
    if (Shared(Address))
        physical_address = TLB(Address)
        ClearByAddress(physical_address, size)
```

共有メモリと同期化基本命令の詳細については、P. A2-44 *「同期化基本命令」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート　データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント　ARMv6 以前は、メモリアドレスがワード境界にアラインしていない場合、命令の動作は予測不能です。アライメントチェック（アドレス [1:0] != 0b00 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定混在エンディアン形式とアライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - CP15_reg1_Ubit == 0 の場合、境界アラインしていないアクセスの結果は予測不能です。
  - CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスがサポートされています。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

等価な ARM 構文とエンコード

```
STR  <Rd>, [SP, #<immed_8> * 4]
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9–2	1	0
1	1	1	0	0	1	0	1	1	0	0	0	1	1	0	1	Rd	0	0	immed_8	0	0

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.61 STRB (1)

15	14	13	12	11	10 6	5 3	2 0
0	1	1	1	0	immed_5	Rn	Rd

STRB (1)（レジスタストアバイト）は、8 ビットのデータを汎用レジスタからメモリにストアします。このアドレシングモードは、構造体（レコード）のフィールドへのアクセスに便利です。

オフセットが 0 の場合、生成されるアドレスはベースレジスタ <Rn> の値と同じです。

構文

```
STRB  <Rd>, [<Rn>, #<immed_5>]
```

各項目の説明については以下を参照して下さい。

<Rd>　　メモリにストアされるバイトを最下位に含むレジスタ。

<Rn>　　ベースアドレスを含むレジスタ。

<immed_5>　　5 ビットの値で、この値が <Rn> の値に加算され、メモリアドレスが計算されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
address = Rn + immed_5
Memory[address,1] = Rd[7:0]
if (CP15_reg1_Ubit == 1)    /* ARMv6 */
    if (Shared(Address))
        physical_address = TLB(Address)
        ClearByAddress(physical_address, size)
```

共有メモリと同期化基本命令の詳細については、P. A2-44 *「同期化基本命令」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

等価な ARM 構文とエンコード

```
STRB  <Rd>, [<Rn>, #<immed_5>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4–0
1	1	1	0	0	1	0	1	1	1	0	0	Rn	Rd	0	0	0	0	0	0	0	immed_5

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.62 STRB (2)

15	14	13	12	11	10	9	8 6	5 3	2 0
0	1	0	1	0	1	0	Rm	Rn	Rd

STRB (2) は、8 ビットのデータを汎用レジスタからメモリにストアします。このアドレシングモードは、ポインタ + 大きなオフセットの演算と、配列のエレメントの 1 つにアクセスするために便利です。

構文

```
STRB  <Rd>, [<Rn>, <Rm>]
```

各項目の説明については以下を参照して下さい。

<Rd> メモリにストアされるバイトを最下位に含むレジスタ。

<Rn> メモリアドレスを計算するために使用される最初の値を含むレジスタ。

<Rm> このレジスタ値は <Rn> の値に加算され、メモリアドレスが計算されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
address = Rn + Rm
Memory[address,1] = Rd[7:0]
if (CP15_reg1_Ubit == 1)    /* ARMv6 */
    if (Shared(Address))
        physical_address = TLB(Address)
        ClearByAddress(physical_address, size)
```

共有メモリと同期化基本命令の詳細については、P. A2-44 *「同期化基本命令」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

等価な ARM 構文とエンコード

```
STRB  <Rd>, [<Rn>, <Rm>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	0	1	1	1	1	1	0	0	Rn	Rd	0	0	0	0	0	0	0	0	Rm

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.63 STRH (1)

15	14	13	12	11	10 ... 6	5 ... 3	2 ... 0
1	0	0	0	0	immed_5	Rn	Rd

STRH (1)（レジスタストアハーフワード）は、16 ビットのデータを汎用レジスタからメモリにストアします。このアドレシングモードは、構造体（レコード）のフィールドへのアクセスに便利です。オフセットが 0 の場合、生成されるアドレスはベースレジスタ <Rn> の値と同じです。

構文

```
STRH  <Rd>, [<Rn>, #<immed_5> * 2]
```

各項目の説明については以下を参照して下さい。

<Rd> メモリにストアされるハーフワードを最下位に含むレジスタ。

<Rn> ベースアドレスを含むレジスタ。

<immed_5> 5 ビットのイミディエート値で、この値を 2 倍して <Rn> の値に加算し、メモリアドレスが計算されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
address = Rn + (immed_5 * 2)
if (CP15_reg1_Ubit == 0)
    if address[0] == 0b0 then
        Memory[address,2] = Rd[15:0]
    else
        Memory[address,2] = UNPREDICTABLE
else                                /* CP15_reg1_Ubit == 1 */
    Memory[address,2] = Rd[15:0]
    if (Shared(Address))            /* from ARMv6 */
        physical_address = TLB(Address)
        ClearByAddress(physical_address, size)
```

共有メモリと同期化基本命令の詳細については、P. A2-44 *「同期化基本命令」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント ARMv6 以前は、メモリアドレスがハーフワード境界にアラインしていない場合、命令の動作は予測不能です。アライメントチェック（アドレス [0] != 0 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定混在エンディアン形式とアライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - — CP15_reg1_Ubit == 0 の場合、境界アラインしていないアクセスの結果は予測不能です。
  - — CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスがサポートされています。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

等価な ARM 構文とエンコード

```
STRH  <Rd>, [<Rn>, #<immed_5> * 2]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10	9 8	7 6 5 4	3 1	0
1 1 1 0	0 0 0 1 1 1 0 0	Rn	Rd	0 0	immed [4:3]	1 0 1 1	immed [2:0]	0

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.64 STRH (2)

15	14	13	12	11	10	9	8 6	5 3	2 0
0	1	0	1	0	0	1	Rm	Rn	Rd

STRH (2) は、16 ビットのデータを汎用レジスタからメモリにストアします。このアドレシングモードは、ポインタ + 大きなオフセットの演算と、配列のエレメントの 1 つにアクセスするために便利です。

構文

```
STRH  <Rd>, [<Rn>, <Rm>]
```

各項目の説明については以下を参照して下さい。

`<Rd>` メモリにストアされるハーフワードを最下位に含むレジスタ。

`<Rn>` メモリアドレスを計算するために使用される最初の値を含むレジスタ。

`<Rm>` このレジスタ値は `<Rn>` の値に加算され、メモリアドレスが計算されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
address = Rn + Rm
if (CP15_reg1_Ubit == 0)
    if address[0] == 0b0 then
        Memory[address,2] = Rd[15:0]
    else
        Memory[address,2] = UNPREDICTABLE
else    /* CP15_reg1_Ubit == 1 */
    Memory[address,2] = Rd[15:0]
    if (Shared(Address))
        physical_address = TLB(Address)
        ClearByAddress(physical_address, size)
```

共有メモリと同期化基本命令の詳細については、P. A2-44 *「同期化基本命令」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

データアボート データアボートが発生した場合の命令への影響の詳細については、P. A2-21 *「データアボートが発生した命令の影響」*を参照して下さい。

アライメント ARMv6 以前は、メモリアドレスがハーフワード境界にアラインしていない場合、命令の動作は予測不能です。アライメントチェック（アドレス [0] != 0 の場合にデータアボートを発生する）と、ビッグエンディアン（BE-32）データ形式のサポートは実装オプションです。

ARMv6 から、バイト固定混在エンディアン形式とアライメントチェックオプションがサポートされています。

- CP15_reg1_Abit == 1 の場合、境界アラインしていないアクセスはデータアボートを発生します（アライメントフォルト）。
- CP15_reg1_Abit == 0 の場合、次の動作が実行されます。
  - — CP15_reg1_Ubit == 0 の場合、境界アラインしていないアクセスの結果は予測不能です。
  - — CP15_reg1_Ubit == 1 の場合、境界アラインしていないアクセスがサポートされています。

エンディアン形式とアライメントの詳細については、P. A2-30 *「エンディアンのサポート」*と P. A2-38 *「アンアラインドアクセスのサポート」*を参照して下さい。

等価な ARM 構文とエンコード

```
STRH  <Rd>, [<Rn>, <Rm>]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 2 1 0
1 1 1 0	0 0 0 1 1 0 0 0	Rn	Rd	SBZ	1 0 1 1	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.65 SUB (1)

15	14	13	12	11	10	9	8 - 6	5 - 3	2 - 0
0	0	0	1	1	1	1	immed_3	Rn	Rd

SUB (1)(減算)は、小さな定数値をレジスタの値から減算し、結果を 2 番目のレジスタにストアします。

結果に基づいて条件コードフラグを更新します。

構文

```
SUB  <Rd>, <Rn>, #<immed_3>
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタ。

<Rn> 減算の第 1 オペランドレジスタ。

<immed_3> <Rn> の値から減算される 3 ビットのイミディエート値(0 ～ 7)。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rn - immed_3
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - immed_3)
V Flag = OverflowFrom(Rn - immed_3)
```

等価な ARM 構文とエンコード

```
SUBS  <Rd>, <Rn>, #<immed_3>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 - 16	15 - 12	11	10	9	8	7	6	5	4	3	2 - 0
1	1	1	0	0	0	1	0	0	1	0	1	Rn	Rd	0	0	0	0	0	0	0	0	0	immed_3

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.66 SUB (2)

15	14	13	12	11	10 … 8	7 … 0
0	0	1	1	1	Rd	immed_8

SUB (2) は、大きなイミディエート値をレジスタの値から減算し、結果を同じレジスタにストアします。

結果に基づいて条件コードフラグを更新します。

構文

```
SUB  <Rd>, #<immed_8>
```

各項目の説明については以下を参照して下さい。

<Rd> 減算の最初のオペランドを保持し、演算のデスティネーションレジスタになります。

<immed_8> <Rn> の値から減算される 8 ビットのイミディエート値（0 ～ 255）。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rd - immed_8
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rd - immed_8)
V Flag = OverflowFrom(Rd - immed_8)
```

等価な ARM 構文とエンコード

```
SUBS  <Rd, <Rd>, #<immed_8>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 … 16	15 … 12	11	10	9	8	7 … 0
1	1	1	0	0	0	1	0	0	1	0	1	Rd	Rd	0	0	0	0	immed_8

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.67 SUB (3)

15	14	13	12	11	10	9	8 6	5 3	2 0
0	0	0	1	1	0	1	Rm	Rn	Rd

SUB (3) は、レジスタの値を2番目のレジスタの値から減算し、結果を3番目のレジスタにストアします。

結果に基づいて条件コードフラグを更新します。

構文

```
SUB  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> 演算のデスティネーションレジスタ。

<Rn> 減算の最初のオペランドレジスタ。

<Rm> <Rn> から減算される値を保持しているレジスタ。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
Rd = Rn - Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - Rm)
V Flag = OverflowFrom(Rn - Rm)
```

等価な ARM 構文とエンコード

```
SUBS  <Rd>, <Rn>, <Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
1 1 1 0	0 0 0 0 0 1 0 1	Rn	Rd	0 0 0 0 0 0 0 0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.68 SUB (4)

15	14	13	12	11	10	9	8	7	6 ... 0
1	0	1	1	0	0	0	0	1	immed_7

SUB (4) はSPの値を7ビットイミディエート値の4倍だけデクリメントします(4の倍数で、0〜508になります)。

条件コードは変化しません。

構文

```
SUB  SP, #<immed_7> * 4
```

各項目の説明については以下を参照して下さい。

SP　スタックポインタ。演算の結果も SP にストアされます。

<immed_7>　7 ビットのイミディエート値で、この値を 4 倍してから、スタックポインタの値から減算されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
SP = SP - (immed_7 << 2)
```

用法

Thumb 命令セットで使用するフル下降スタックでは、SP の値をデクリメントすると、スタックの最上部に追加のメモリ変数が割り当てられます。

注

代替構文　この命令は、`SUB SP, SP, #(<immed_7> * 4)` の形式で記述することもできます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
SUB   SP, SP, #<immed_7> * 4
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6 ... 0
1	1	1	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	0	1	1	1	1	1	0	immed_7

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.69 SWI

15	14	13	12	11	10	9	8	7 ... 0
1	1	0	1	1	1	1	1	immed_8

SWI（ソフトウェア割り込み）は、ソフトウェア割り込み（SWI）を生成します。この割り込みはオペレーティングシステムによって処理されます。詳細については、P. A2-16「*例外*」を参照して下さい。

サービスを提供するオペレーティングシステムサービスを呼び出すために使用します。

構文

```
SWI  <immed_8>
```

各項目の説明については以下を参照して下さい。

<immed_8> 8 ビットイミディエート値で、命令のビット [7:0] に置かれます。この値はプロセッサには無視されますが、オペレーティングシステムの SWI 割り込みハンドラはこの値を使用して、要求されたオペレーティングシステムサービスを判定できます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

ソフトウェア割り込み

動作

```
R14_svc    = address of next instruction after the SWI instruction
SPSR_svc   = CPSR
CPSR[4:0] = 0b10011              /* Enter Supervisor mode */
CPSR[5]    = 0                   /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]    = 1                   /* Disable normal interrupts */
if high vectors configured then
    PC     = 0xFFFF0008
else
    PC     = 0x00000008
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

等価な ARM 構文とエンコード

```
SWI  <immed_8>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7 ... 0
1	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	immed_8

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.70 SXTB

15	14	13	12	11	10	9	8	7	6	5 3	2 0
1	0	1	1	0	0	1	0	0	1	Rm	Rd

SXTB（符号付きバイト拡張）は、オペランドの最下位 8 ビットを抽出し、32 ビットに符号拡張します。フラグは変更されません。

構文

```
SXTB  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> デスティネーションレジスタ。

<Rm> オペランドレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
Rd = SignExtend(Rm[7:0])
```

用法

SXTB はバイトからワードへの符号拡張、たとえば C/C++ で **signed char** を操作する命令シーケンスで使用します。

等価な ARM 構文とエンコード

```
SXTB  <Rd>, <Rm>
```

31 28	27	26	25	24	23	22	21	20	19 18 17 16	15 12	11 10 9 8 7 6 5 4	3 0
1 1 1 0	0	1	1	0	1	0	1	0	1 1 1 1	Rd	0 0 0 0 0 1 1 1	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.71 SXTH

15	14	13	12	11	10	9	8	7	6	5 3	2 0
1	0	1	1	0	0	1	0	0	0	Rm	Rd

SXTH16（符号付きハーフワード拡張）は、オペランドの下位 16 ビットを抽出し、32 ビットに符号拡張します。

SXTH はフラグを変更しません。

構文

```
SXTH  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> デスティネーションレジスタ。

<Rm> オペランドレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
Rd = SignExtend(Rm[15:0])
```

用法

SXTH はハーフワードからワードへの符号拡張、たとえば C/C++ で **signed short** 値を操作する命令シーケンスで使用します。

等価な ARM 構文とエンコード

```
SXTH  <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	0	1	1	0	1	0	1	1	1	1	1	1	Rd	0	0	0	0	0	1	1	1	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.72 TST

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	1	0	0	0	Rm	Rn

TST（テスト）は、レジスタに含まれる特定のビットサブセットのうち、最低 1 つのビットがセットされているかどうかを判定します。TST の特に一般的な使用目的は、ある 1 つのビットがセット / クリアされているかどうかのテストです。

結果に基づいて条件コードフラグを更新します。

構文

```
TST  <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rn> 命令の最初のオペランドを含むレジスタ。

<Rm> このレジスタの値と、<Rn> の値とで論理 AND 演算が実行されます。

アーキテクチャのバージョン

すべての T バリエーションに存在します。

例外

なし

動作

```
alu_out = Rn AND Rm
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

等価な ARM 構文とエンコード

```
TST  <Rn>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	0	0	0	1	0	0	0	1	Rn	SBZ	0	0	0	0	0	0	0	0	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.73 UXTB

15	14	13	12	11	10	9	8	7	6	5 3	2 0
1	0	1	1	0	0	1	0	1	1	Rm	Rd

UXTB（符号なしバイト拡張）は、オペランドの最下位 8 ビットを抽出し、32 ビットにゼロ拡張します。

構文

```
UXTB <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd> デスティネーションレジスタ。

<Rm> オペランドレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
Rd = Rm AND 0x000000FF
```

用法

UXTB はハーフワードからワードへのゼロ拡張、たとえば C/C++ で **unsigned short** 値を操作する命令シーケンスで使用します。

等価な ARM 構文とエンコード

```
UXTB  <Rd>, <Rm>
```

31 28	27	26	25	24	23	22	21	20	19	18	17	16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 0	0	1	1	0	1	1	1	0	1	1	1	1	Rd	0	0	0	0	0	1	1	1	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.1.74 UXTH

15	14	13	12	11	10	9	8	7	6	5　　3	2　　0
1	0	1	1	0	0	1	0	1	0	Rm	Rd

UXTH（符号なしハーフワード拡張）は、オペランドの下位 16 ビットを抽出し、32 ビットにゼロ拡張します。

構文

```
UXTH <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<Rd>　　デスティネーションレジスタ。

<Rm>　　オペランドレジスタ。

アーキテクチャのバージョン

ARMv6 以降に存在します。

例外

なし

動作

```
Rd = Rm AND 0x0000FFFF
```

用法

UXTH はハーフワードからワードへのゼロ拡張、たとえば C/C++ で **unsigned short** 値を操作する命令シーケンスで使用します。

等価な ARM 構文とエンコード

```
UXTH  <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15　　12	11	10	9	8	7	6	5	4	3　　0
1	1	1	0	0	1	1	0	1	1	1	1	1	1	1	1	Rd	0	0	0	0	0	1	1	1	Rm

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

A7.2 Thumb 命令とアーキテクチャのバージョン

現行の ARM アーキテクチャのうち、Thumb をサポートしている各バージョンについて、使用できる Thumb 命令の一覧を表 A7-1 に示します。

表 A7-1 各アーキテクチャで使用可能な Thumb 命令

命令	v4T、v4TxM	v5T、v5TxM	v6
ADC	あり	あり	あり
ADD（すべての形式）	あり	あり	あり
AND	あり	あり	あり
ASR（両方の形式）	あり	あり	あり
B（両方の形式）	あり	あり	あり
BIC	あり	あり	あり
BKPT	なし	あり	あり
BL	あり	あり	あり
BLX（両方の形式）	なし	あり	あり
BX	あり	あり	あり
CMN	あり	あり	あり
CMP（すべての形式）	あり	あり	あり
CPS	なし	なし	あり
CPY	なし	なし	あり
EOR	あり	あり	あり
LDMIA	あり	あり	あり
LDR（すべての形式）	あり	あり	あり
LDRB（両方の形式）	あり	あり	あり
LDRH（両方の形式）	あり	あり	あり
LDRSB	あり	あり	あり
LDRSH	あり	あり	あり
LSL（両方の形式）	あり	あり	あり

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 A7-1 各アーキテクチャで使用可能な Thumb 命令（続き）

命令	v4T、v4TxM	v5T、v5TxM	v6
LSR（両方の形式）	あり	あり	あり
MOV（すべての形式）	あり	あり	あり
MUL	あり	あり	あり
MVN	あり	あり	あり
NEG	あり	あり	あり
ORR	あり	あり	あり
POP	あり	あり	あり
PUSH	あり	あり	あり
REV（すべての形式）	なし	なし	あり
ROR	あり	あり	あり
SBC	あり	あり	あり
SETEND	なし	なし	あり
STMIA	あり	あり	あり
STR（すべての形式）	あり	あり	あり
STRB（両方の形式）	あり	あり	あり
STRH（両方の形式）	あり	あり	あり
SUB（すべての形式）	あり	あり	あり
SWI	あり	あり	あり
SXTB/H	なし	なし	あり
TST	あり	あり	あり
UXTB/H	なし	なし	あり

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

パート B

メモリとシステムアーキテクチャ

第 B1 章
メモリとシステムアーキテクチャの概要

本章では、メモリとシステムアーキテクチャの高レベルな概要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。

- *メモリシステムの概要* : P. B1-2
- *メモリ階層* : P. B1-4
- *L1 キャッシュ* : P. B1-6
- *L2 キャッシュ* : P. B1-7
- *ライトバッファ* : P. B1-8
- *密結合メモリ* : P. B1-9
- *非同期な例外* : P. B1-10
- *セマフォ* : P. B1-12

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B1.1 メモリシステムの概要

ARM® アーキテクチャは、長年にわたって拡張が行われてきました。この期間に出荷された ARM プロセッサは 10 億個を超え、そのほとんどは ARMv4 または ARMv5 に準拠した製品です。これらの製品に対するメモリシステムの要件は、フラットなアドレスマップを持つ単純なメモリブロックから、メモリリソースの活用を最適化するため次のような技術の一部またはすべてを使用するシステムまで、多岐にわたります。

- 複数のタイプのメモリ
- キャッシュ
- 書き込みバッファ
- 仮想メモリなどのメモリ再マップ手法

メモリシステムの制御は、主に*キャッシュ可能*と*バッファ可能*の属性によって記述されていました。これらの属性は、基礎となるハードウェア機構から派生した名前で、プログラマが判断の基礎とできるような、その機構に関連付けられた正式なプロパティの説明は存在していませんでした。さらに、メモリアクセスの順序モデルは定義されていませんでした。初期の実装から暗黙のモデルが作成されていましたが、これは現在開発されているものよりもはるかに単純なシステムです。

より性能の高いシステムと、関連する実装の要求に対応するため、ARMv6 では新しいメモリバリアコマンドを含む、仮想メモリシステムとウィークオーダメモリモデルの規則が導入されました。

メモリの動作は、タイプで分類されるようになりました。

- ストロングオーダ
- デバイス
- 通常

これらの基本タイプに加えて、アクセス機構としてキャッシュ、および共有属もさらに確認されることになります。

ARM *アーキテクチャリファレンスマニュアル*の第 2 版と同様に、必要と思われる事項については、その内容に応じて記述されています。しかし中心となるのは、ARMv6 仮想メモリモデルとその要求される機能です。以前のアーキテクチャで使用されていた仮想メモリをサポートする機構は、下位互換性モデルに記述されています。しかし以前の機能の一部は下位互換性のためだけに残されているもので、新しい設計での使用は推奨されません。

コプロセッサ 15（CP15）は CPU の識別や、他のメモリシステムと機能の制御機能に加えて、仮想メモリシステムの主要な制御のために使用されます。CP15 機構は ARMv6 の必要条件です。

メモリシステムとメモリオーダモデルについては、パート B の次の章で説明されています。

概要

本章。

メモリ階層

基本的なキャッシュ理論や密結合メモリなどを含む概要。

メモリ順序モデル

ARMv6 で導入されたメモリ属性と順序規則。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

システム制御コプロセッサ

提供されている機能とサポートの概要。

仮想メモリシステムアーキテクチャ（**VMSA**）

メモリ管理ユニット（MMU）に基づいて、仮想メモリから物理メモリへのアドレスマッピング、メモリへのアクセス許可、その他のメモリ属性を制御する高度なシステム。修正された ARMv6 モデルと、以前のバージョンのアーキテクチャで使用されていたモデルについて説明しています。

保護メモリシステムアーキテクチャ（**PMSA**）

MMU メモリシステムの提供する完全な機能を必要としない多くのアプリケーションに適切な、より単純な保護機構。修正された ARMV6 と、以前のバージョンのアーキテクチャモデルについて説明しています。

キャッシュとライトバッファ

メモリ階層内でキャッシュとライトバッファの機能を制御する機構。

L1 密結合メモリのサポート

ARMv6 の機構で、関連する DMA と SmartCache モデルを含みます。

高速コンテキストスイッチ拡張機能

高速コンテキストスイッチ拡張機能の説明です。この機能により、別のプロセスブロックで実行されている最大 128 のプロセスを高速にスイッチングできます。各プロセスブロックのサイズは 32MB までです。この機能は ARMv6 では下位互換性の目的でのみサポートされており、使用は推奨されません。

——— 注 ———

パート B では、広い範囲にわたって機能の解説をします。ARMv6 は、メモリモデルと多くのシステムレベル機能を標準化した最初のバージョンのアーキテクチャです。また、システム制御コプロセッサ機構と、ハードウェアおよびソフトウェアの設計でのシステムレベルの整合性を必須とした最初のアーキテクチャです。このため、パート B の説明で、ARMv6 は大きなアーキテクチャの変革として示されています。実装への絶対条件は ARMv6 に準拠した実装に対する要求であり、以前のバージョンのアーキテクチャに関してはシステムガイドラインとしてのみ考慮する必要があります。

バージョン 4 以前のバージョンのアーキテクチャはすべて、現在では使用されていないものです。たとえば、26 ビットモードの説明はすべて削除されています。

ARMv6 以前の ARM プロセッサの一部は、ここで説明しているのとは異なる方法で機能が実装されています。このため、対応する ARM プロセッサのデータシートやテクニカルリファレンスマニュアルが、メモリ、およびシステム制御機能に関する参照資料となります。ガイドラインに従ったプロセッサは、ほとんどの場合既存と将来の ARM ソフトウェアと互換性があります。ARMv6 はシステム設計のベースラインを確立していますが、追加の機能や実装依存のオプションが存在する可能性があります。最適のシステム設計とパフォーマンスを実現するため、システム設計者はアーキテクチャとベンダのデータシートを参照することをお勧めします。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B1.2 メモリ階層

優れたシステムを設計するには、システムのパフォーマンスとコストに関する総合的な目標を達成するため、多くのトレードオフのバランスを考慮する必要があります。この決定手順で重要な要素の 1 つはメモリの設計です。

- メモリのタイプ、たとえば、ROM、フラッシュ、DRAM、SRAM、ディスクベースの記憶域など
- サイズ - 容量とシリコンの面積
- アクセス速度 - メモリの読み出し / 書き込みに必要なコアクロックサイクル数
- アーキテクチャ - ハーバード（命令とデータに独立したメモリを使用）またはフォンノイマン（統合メモリ）

一般的な規則として、メモリのアクセス時間が短いほど、利用可能なリソースの量は制限されます。これは、メモリがプロセッサコアに密に結合される、つまり同じダイ上に存在する必要があるためです。オンチップのメモリでも、タイプやサイズ、電力の制限、物理的なレイアウトでメモリにアクセスするためのクリティカルパスの長さなど各種の要因でタイミングの条件が異なる可能性があります。キャッシュを使用すると、最も速度が速く高価なシステムメモリリソースを、アプリケーションで現在アクティブなプロセススレッド間で共用できます。

システムは種類の異なるメモリを階層化したモデルにより実装されています。これを*メモリ階層*と呼びます。システムは複数のレベルでキャッシュを採用できます。外側の層になるほどレイテンシが長くなり、サイズは大きくなります。システムのすべてのキャッシュは、システムアーキテクチャの一部であるメモリのコヒーレンシポリシーに準拠する必要があります。このように階層化されたシステムでは、通常は階層にレベル 1、レベル 2、…レベル *n* という番号が付けられます。番号が大きくなるほど、コアから離れた階層であることを示し、アクセス時間も長くなります。

I/O も同様にいくつかの階層に分かれて提供されます。いくつかの 0 ウェイトペリフェラルがレベル 1 にあり、メモリ上にマッピングされたペリフェラルはシステムバス上にあります。

図 B1-1 は、メモリ階層の例です。

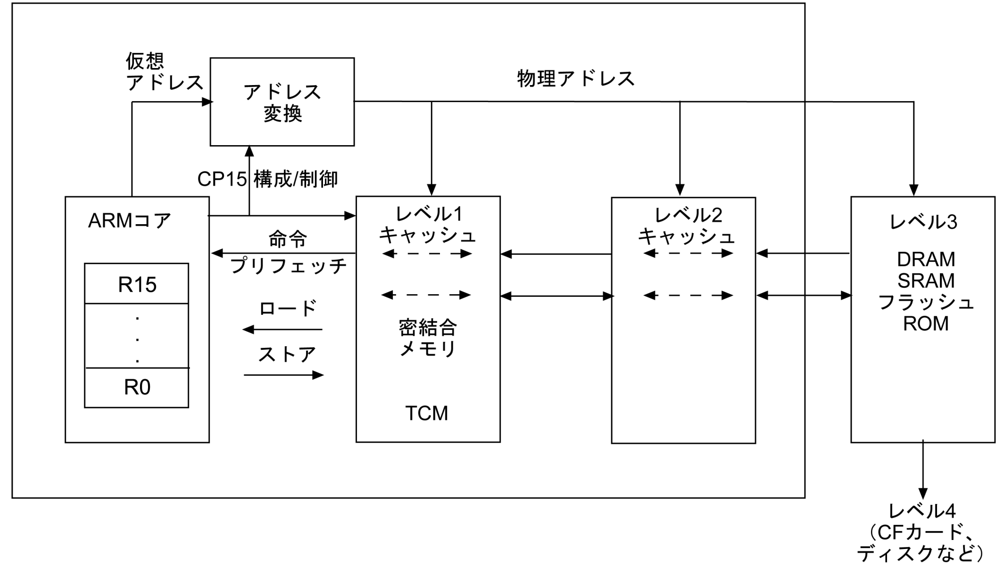


図 B1-1 メモリ階層の例

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ARMv6 では、キャッシュを提供する*レベル 1*（L1）サブシステム、*密結合メモリ*（TCM）、関連する TCM-L1 DMA システムが定義されています。このアーキテクチャでは広範な実装が可能で、ソフトウェアから参照可能な構成レジスタにより、存在するリソースの識別が可能です。L1 サブシステムではオプションとして、メモリ管理ユニット（VMSAv6）またはより単純なメモリ保護ユニット（PMSAv6）のサポートを実装することもできます。

マルチプロセッサ実装とレベル 2（L2）キャッシュをサポートする機構も用意されています。ただし、このドキュメントではこれらについては詳しく説明しません。将来の互換性を保証するため、レベル 2 キャッシュと密結合マルチプロセッサシステムを実装する場合は ARM と密接な共同の上で設計を行うことをお勧めします。

VMSAv6 では、各ページについて定義される内部属性と外部属性が記述されています。これらの属性は、メモリの異なる領域について、異なるキャッシュレベルでキャッシュポリシーを制御するために使用されます。実装では、内部属性と外部属性を使用して、実装定義の手法で他のレベルのキャッシュポリシーを記述できます。アーキテクチャの詳細については、P. B4-11*「メモリ領域属性」*を参照して下さい。すべてのレベルのキャッシュは、適切なキャッシュ管理を持ち、次の機能をサポートしている必要があります。

- キャッシュクリーニング（ライトバックキャッシュのみ）
- キャッシュの無効化（すべてのキャッシュ）

ARM のプロセッサとソフトウェアは、バイトアドレスのメモリに接続するように設計されています。ARMv6 以前のバージョンでは、アドレッシングはワード不変型として定義されていました。メモリへのワードとハーフワードのアクセスはアドレスのバイト境界を無視し、アクセス幅で指定された本来の境界によりアクセスされます。つまりワードアクセスの場合はアドレスビットの 0 と 1、ハーフワードアクセスの場合はビット 0 が無視されます。ARM プロセッサのエンディアン形式は、通常はメモリシステムに合わせて決定されるか、または非ワードアクセスが発生する前に構成されます。

ARMv6 では、次の機能が追加されています。

- バイト固定アドレスモデル
- 境界アラインしていないワードとハーフワードアクセスのサポート
- リトルエンディアンまたはビッグエンディアン形式でデータのロードとストアを行うための追加機能

詳細については、P. A2-30*「エンディアンのサポート」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B1.3 L1 キャッシュ

ARMv6 以前では、ARM キャッシュは一般に仮想インデックスと仮想アドレスタグを持つ、仮想アドレスのキャッシュとして実装されていました。このモデルでは、物理ページは単一の仮想ページにのみマップされ、それ以外の場合には結果は予測不能になります。これらの実装では、単一の物理ページについて複数の仮想コピー間のコヒーレンシが保証されていませんでした。

ARMv6 で想定されているキャッシュアーキテクチャは、物理タグで管理されるキャッシュの動作を前提としています。ARMv6 の L1 キャッシュアーキテクチャは、コンテキスト切り替え時のキャッシュのクリーニングや無効化の条件を少なくし、特定のメモリロケーションに対して複数の仮想アドレスエイリアスをサポートするように設計されています。このサブシステム内でのキャッシュのサイズ、アソシアティビティ、構成に関する柔軟性は、コプロセッサのシステム制御レジスタ（CP15）で提供されています。キャッシュの構成は、独立した命令キャッシュとデータキャッシュを持つハーバードアーキテクチャか、単一の統合キャッシュを持つフォンノイマンアーキテクチャを選択できます。

ハーバードアーキテクチャでは、命令キャッシュとデータキャッシュのコヒーレンシを保証するためのハードウェアサポートを実装に含める必要はありません。このようなサポートが必要になる場合、たとえば自己変更コードでは、問題を回避するためにソフトウェアでキャッシュクリーニング命令を使用する必要があります。

ARMv6 の L1 キャッシュは、ソフトウェアから見て次のように動作している必要があります。

- 異なる仮想アドレスから物理アドレスへのマッピングを使用するソフトウェアが切り替わるときに、キャッシュのエントリをクリーン、もしくは無効化する必要はありません。
- 同じ物理アドレスへのエイリアスが、ページテーブルでキャッシュ可能に指定されているメモリ領域に存在する可能性があります。このエイリアスは、P. B6-11*「ページテーブルのマッピングの制限」*に説明されている 4KB のスモールページの制約に従います。

これらの動作条件に適合すれば、キャッシュは仮想アドレスまたは物理アドレス（インデクス付けを含む）で実装可能です。ARMv6 の L1 キャッシュ管理は仮想アドレスを使用し、以前のアーキテクチャガイドラインおよび実装と一貫性があります。

L1 キャッシュのアーキテクチャの詳細については、第 B6 章*「キャッシュとライトバッファ」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B1.4 L2 キャッシュ

L1 キャッシュは常にコアに密結合されていますが、L2 キャッシュは次のどちらの場合もあります。

- コアに密結合されている。
- システムバス上のメモリマップされたペリフェラルとして実装されている。

L2 キャッシュで推奨されるコマンドの最小セットは、構成と制御のために定義されたものです。密結合された L2 キャッシュは、システム制御コプロセッサで管理する必要があります。L2 キャッシュが制御機能に仮想アドレスと物理アドレスのどちらを使用するかは実装定義です。メモリマップされた L2 キャッシュは、物理アドレスベースの制御を使用する必要があります。

より高いレベルのキャッシュを実装することも可能ですが、ARMv6 では以下の規則に準拠することを除き、そのキャッシュの制御に関して規定はしません。

- P. B4-11 *「メモリ領域属性」*に記述されている内部と外部の属性モデル
- システム制御コプロセッサインターフェースにより複数レベルのキャッシュのコヒーレンシを管理できること。詳細については P. B6-12 *「キャッシュのレベルの追加に関する考慮事項」*を参照して下さい。

L2 キャッシュのアーキテクチャの詳細については、*L2 キャッシュ*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B1.5 ライトバッファ

*ライトバッファ*という用語には複数の異なる動作が含まれます。予期しない結果を防止するため、プログラマはメモリマップされた空間におけるさまざまな使用法でのこれらの動作の影響を理解する必要があります。このため、*バッファ可能*という用語は、メモリシステムで必須の動作を記述する属性としては使用されなくなりました。

ライトバッファは、書き込みトランザクションを、以降のメモリトランザクションの実行と分離するためのものです。また、特定のバッファの実装では、メモリ転送順序の変更、隣接した場所への複数の書き込みの結合、以降の読み出しへの書き込みデータのフォワードなど、他のタスクを実行する場合があります。これらのバッファ動作は、その性質からよりキャッシュに近いものとなります。P. B2-12 *「ストロングオーダメモリ属性」*で記述されている*ストロングオーダ*、*デバイス*、*通常*のメモリ属性は、各メモリ属性の動作が正しく満たされていれば、プログラマが必要な動作を記述し、実装者がシステムに対して最適な構造を自由に選択できるように設計されたものです。

バッファされたメモリ領域への書き込みについては、ライトバッファにデータが転送された時点で検出できる場合のみ、プロセッサに正確なアボート信号を発生できます。例えばメインメモリでの ECC エラーなどデータが後でメインメモリに書き込まれた時点でのみ検出可能な場合は、不正確なアボートか割り込みを発生させて処理する必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B1.6 密結合メモリ

密結合メモリ（TCM）は、L1 キャッシュとともにレベル 1 メモリサブシステムの一部として実装されるメモリの領域です。TCM は物理的にアドレス指定され、各バンクは物理メモリマップで固有の場所を占めます。ARMv6 のオプションとして使用できるスマートキャッシュについては、P. B7-6 *「スマートキャッシュの動作」*を参照して下さい。L1 キャッシュと同様に、TCM の構成は、独立した命令 TCM とデータ TCM を持つハーバードアーキテクチャか、単一の統合 TCM を持つフォンノイマンアーキテクチャを選択できます。

TCM は、プロセッサによって使用できるレイテンシの低いメモリを実現し、同時にキャッシュの予測不能性を回避するように設計されています。このようなメモリは、割り込み処理ルーチンやリアルタイムタスクなど、キャッシュの不確定性が望ましくない重要なルーチンの保持に使用できます。このようなデータの例には、他に次のようなものがあります。

- スクラッチパッドデータ
- 局所性の関係から、キャッシュに適さないデータ
- 割り込みスタックなど重要なデータ

TCM のアーキテクチャの詳細については、第 B7 章*「密結合メモリ」*を参照して下さい。

B1.6.1 密結合メモリとキャッシュメモリの比較

TCM はシステムの物理メモリマップの一部として使用するために設計されており、同じ物理アドレスにある別レベルの外部メモリによって補完されるものではありません。このため、TCM の動作は、ライトスルーキャッシュ可能にマークされているメモリ領域のキャッシュの動作とは異なっています。このような領域では、TCM に格納されているメモリロケーションへの書き込みに対して、外部書き込みは発生しません。

アーキテクチャの必要条件は、そのメモリの場所が TCM またはキャッシュに含まれていることで、両方に含まれていることではありません。特に、TCM とキャッシュの間ではコヒーレンシ機構は一切サポートされていません。このため、TCM のベースアドレスを割り当てる場合は、TCM のアドレス範囲が他の有効なキャッシュエントリとオーバラップしないようにすることが重要です。

B1.6.2 密結合メモリの DMA サポート

ARMv6 には、レジスタによる設定が可能な DMA モデルがあります。これは、関連するプロセッサコア以外に TCM の読み出しと書き込みが可能な唯一の機構です。DMA チャネルは 2 つまで使用可能です。これによって、チェーン動作が可能になります。アーキテクチャの詳細については、P. B7-8 *「レベル 1（L1）DMA モデル」*を参照して下さい。

注

TCM の DMA 機構と、P. B7-6 *「スマートキャッシュの動作」*で説明されている *SmartCache* 機能は排他的な機能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B1.7 非同期な例外

多くの例外は同期イベントで、コアでの命令の実行に関連しています。しかし、次の例外は非同期なイベントを発生させます。

- *リセット*：P. A2-18
- *割り込み*
- *不正確なアボート*：P. B1-11

B1.7.1 リセット

リセットは、ARM アーキテクチャで唯一のマスク不可イベントです。詳細については、P. A2-18*「リセット」*を参照して下さい。

B1.7.2 割り込み

ARM プロセッサは、高速割り込みと通常の割り込みを実装しています。どちらの割り込みも外部からの信号で発生し、多くの実装では例外が発生する前に、割り込みをプロセッサと同期させます。

高速割り込み要求（**FIQ**）

CPSR の I および F ビットをセットし、以降の通常および高速割り込みを禁止します。

マスク不可（ソフトウェアによる）高速割り込み要求

FIQ と同様ですが、CPSR の F ビットは、例外のエントリ時にハードウェアからのみセットすることができます。ソフトウェアでは、割り込み機構を（再度）許可することのみが可能です。

通常割り込み要求（**IRQ**）

CPSR の I ビットをセットし、以降の通常割り込みを禁止します。

一部の実装では、システム制御コプロセッサにより制御される、割り込みベクタを直接コアに返す機構が組み込まれています。この機構は通常は IRQ モードに適用されますが、FIQ モードにも適用可能です。正確な動作は実装定義となります。

割り込みの詳細については、P. A2-24*「割り込み要求（IRQ）例外」*、P. A2-24*「高速割り込み要求（FIQ）例外」*、P. A2-26*「ベクタ割り込みのサポート」*を参照して下さい。

割り込みのキャンセル

割り込みが再度許可される（CPSR の I、F、または両方のビットをクリアする）前に、ソフトウェアは割り込みの発生源がキャンセルされている（プロセッサに割り込みを通知していない）ことを保障する必要があります。割り込みは明示的にデータアクセスを行う任意の命令によりキャンセルできます。これには次のような命令が含まれます。

- 任意のロード命令
- 任意のストア命令
- スワップ命令
- 任意のコプロセッサ命令

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

割り込みをキャンセルするメモリまたはコプロセッサ操作と、CPSR 内の割り込みマスク（I と F）がクリア可能になるまでの間のレイテンシは実装定義です。特に、ARMv6 のメモリには、次の命令の実行前にアクセスが完了することがアーキテクチャ的に保証されているタイプのメモリが含まれていません。その結果、割り込みのキャンセルを保証するアーキテクチャの機構は、各割り込みのキャンセル機構専用の、実装定義のロケーションをポーリングし、割り込みマスクがクリアされる前に割り込みがキャンセルされたことを確認します。

B1.7.3 不正確なアボート

ARMv6 には、不正確なアボートという概念が導入されています。これらのアボートは、アボートを発生した命令の終了後に発生する可能性があります。このため、不正確なアボートは少なくともそれを発生したプロセスに対して致命的であるか、ソフトウェア回復用のアドレス、データ、制御情報を記録するため外部リソースを必要とする場合があります。これらのアボートはほとんどの例外ベクタへのエントリ時にマスクされ、特権ソフトウェアで CPSR_A ビットを使用してマスクすることもできます。詳細については、P. A2-16 *「例外」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B1.8 セマフォ

スワップ（SWP）およびスワップバイト（SWPB）命令は、予測どおりの結果になるよう注意して使用する必要があります。次に 2 つの例を示します。

- 複数のバスマスタを持ち、スワップ命令を使用してセマフォを実装し、異なるバスマスタ間の相互動作を制御しているシステム。

  この場合、メモリのキャッシュされていない領域にセマフォを置き、この機構を使用するすべてのバスマスタについて共通の場所で書き込みのバッファが行われるようにする必要があります。その後で、スワップ命令により、ロックされた読み出し - 書き込みのバストランザクションが発生します。

  このタイプのセマフォは外部からアボート可能です。

- 1 つのプロセッサ上で複数のスレッドを実行しており、スワップ命令を使用してセマフォを実装し、スレッド間の相互動作を制御しているシステム。

  この場合、メモリのキャッシュされている領域にセマフォを置くことができます。ロックされた読み出し - 書き込みバストランザクションは発生することも、しないこともあります。このようなシステムでは、複数のバスマスタを持つシステム（上に述べたようなシステム）と比較して、スワップおよびスワップバイト命令が一般に高いパフォーマンスを持ちます。

  このタイプのセマフォを外部からアボートした場合、結果は予測不能です。

ARMv6 以降、システムパフォーマンス上の理由から、セマフォの実装には排他的なロードとストア命令（LDREX と STREX）を使用する方法が推奨されています。新しい機構は同期化基本命令と呼ばれ、共有メモリモデルケースのすべてのソースから、要求された場所へのアクセスを監視するデータモニタロジックがメモリシステム内に必要です。この命令ではロードとストアの要素間で一定の分離が行われ、ストアに関連するロード以降にその場所に対して他のリソースからの書き込みが行われていない場合のみストアが成功します。詳細については、P. A2-44 *「同期化基本命令」*を参照して下さい。

注

スワップ、およびスワップバイト命令は ARMv6 での使用は推奨されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 B2 章
メモリオーダモデル

本章では、メモリオーダモデルの高レベルな概要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。

- *メモリオーダモデルの概要* : P. B2-2
- *読み出しと書き込みの定義* : P. B2-4
- *ARMv6 以前のメモリ属性* : P. B2-7
- *ARMv6 のメモリ属性の概要* : P. B2-8
- *メモリアクセスの順序に関する条件* : P. B2-16
- *メモリバリア* : P. B2-18
- *メモリのコヒーレンシとアクセスの考慮点* : P. B2-20

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.1 メモリオーダモデルの概要

ARMv6 以前のアーキテクチャでは、明示的なメモリトランザクションで許されるメモリオーダを定義し、以前にそのようなメモリシステムの実装に使用されていたハードウェア手法に従ってメモリの領域を説明することは行われていませんでした。このため、メモリの領域には*ライトスルーキャッシュ可能*、*ライトバックキャッシュ可能*、*キャッシュ不可でバッファ可能*、*キャッシュ不可でバッファ不可*という用語が当てられていました。これらの用語は、コアの従来のハードウェア実装に基づいたもので、メモリトランザクションの正確なプロパティはメモリ名から厳密に特定できません。実装でこれらの名前を異なる方法で解釈しており、互換性のない使用法となっている可能性があります。

同様に、メモリに対してアクセスが行われる順序も定義されておらず、特に、プロセッサにより生成されるメモリトランザクションのオブザーバが確実に利用できる順序の定義が存在していませんでした。実装とシステムが複雑になるにつれ、これらアーキテクチャの未定義の領域は単に標準的なデフォルトに基づいたものから、プロセッサコアおよびシステムレベルで異なる実装間の大きな非互換性の原因となる可能性があるものとなってきました。

ARMv6 では、通常、デバイス、ストロングオーダという一連のメモリタイプが導入され、メモリアクセスプロパティは、元のメモリ領域におけるデファクトの意味に対して大部分下位互換性のある方法で定義されています。I/O 空間へのメモリアクセスが完了したことをプログラムで認識することと、メモリアクセスが他の部分に及ぼす可能性がある影響がシステム全体にわたって可視であることが必要な場合に、ソフトウェアのポーリングポリシーが必要なことから、非互換性が発生することがあります。これは、複雑な高性能システムではメモリアクセスの完了と命令の実行とのリンクを保証することが難しくなってきているのが理由です。

メモリの領域が複数のプロセッサ間で共有される（その結果、オーダリングモデルでキャッシュの透過性が必要となる）ことを示す*共有*メモリ属性も導入されています。この機能を実装する機構は、実装時に自由に選択できます。

メモリオーダモデルに関して重要な考慮点は、対象層により多少異なります。

- ソフトウェアプログラマの場合に重要な点は、メモリに対する書き込みなどを行った結果が反映されるのは、アーキテクチャ的に次の処理に安全に進むことを示すアドレスへのポーリングを行うことでのみ確認できるという点です。
- シリコン実装者の場合は、この章で定義されているストロングオーダとデバイスのメモリ属性により、システム設計者が構成できる設計と、トランザクションの完了をいつ明示できるかということに関して確実な制約を与えることができます。

追加の属性と動作は、メモリシステムのアーキテクチャに関連します。これらの特徴は、本書の他の部分に定義されています。

- MMU をベースとした仮想メモリシステムについては、第 B4 章*「仮想メモリシステムアーキテクチャ」*を参照して下さい。
- MPU をベースとした保護メモリシステムについては、第 B5 章*「保護メモリシステムアーキテクチャ」*を参照して下さい。
- キャッシュとライトバッファについては、第 B6 章*「キャッシュとライトバッファ」*を参照して下さい。
- 密結合メモリ（TCM）については、第 B7 章*「密結合メモリ」*を参照して下さい。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

一部の属性は、ARMv6 の MMU との関連で説明されています。一般に、これらの属性は MPU ベースのシステムにも適用可能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.2 読み出しと書き込みの定義

メモリアクセスは、読み出しまたは書き込みのいずれかです。

B2.2.1 読み出し

読み出しは、読み出しをするメモリ操作です。

ARM® 命令セットには次の命令があります。

- LDM、LDRH、LDRSH、LDRB、LDRSB
- LDM、LDRD、LDRT、LDRBT
- LDC、RFE、SWP、SWPB、LDREX、STREX

Thumb® 命令セットには次の命令があります。

- LDR、LDRH、LDRSH、LDRB、LDRSB
- LDM、POP

ハードウェアで高速化される Jazelle® オペコードでは、オペランドスタックの状態と、Jazelle ハードウェアアクセラレーションの実装により、多数の読み出しが発生することがあります。

B2.2.2 書き込み

書き込みは、書き込みをするメモリ操作です。

ARM 命令セットには次の命令があります。

- STR、STRH、STRB
- STM、STRD、STRT、STRBT
- STC、SRS、SWP、SWPB、STREX

Thumb 命令セットには次の命令があります。

- STR、STRH、STRB
- STM、PUSH

ハードウェアで高速化される Jazelle オペコードでは、オペランドスタックの状態と、Jazelle ハードウェアアクセラレーションの実装により、多数の書き込みが発生することがあります。

B2.2.3 メモリ同期化基本命令

同期化基本命令は、メモリオーダモデル内でシステムセマフォの正しい動作を保証するために必要です。メモリ同期化基本命令は、メモリの同期を保証するために使用される命令として定義されています。

- LDREX、STREX
- SWP、SWPB（ARMv6 では使用は推奨されません）

ARMv6 以前は SWP と SWPB 命令でサポートされていました。ARMv6 では、新しい LDREX と STREX（排他ロード / ストア）命令が導入されています。アーキテクチャの詳細については、P. B2-18 *「メモリバリア」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

LDREX と STREX は、共有メモリと非共有メモリに対してサポートされています。非共有メモリは、同期対象のプロセスが同じプロセッサ上で実行されているときに使用できます。同期対象のプロセスが別のプロセッサ上で実行されているときは、共有メモリを使用する必要があります。

B2.2.4 観測性と完了

観測性の概念はすべてのメモリに適用されますが、グローバルな観測性の概念は共有メモリにのみ適用されます。通常、デバイス、ストロングオーダのメモリは、P. B2-8「*ARMv6 のメモリ属性の概要*」に定義されています。

すべてのメモリについて、次の規則が適用されます。

- メモリのある場所への書き込みについて、その場所に対して同じメモリシステムエージェントにより行われる以後の読み出しについて、書き込み時に書き込まれた値が返される場合、書き込みはメモリシステムエージェントにより観測されていると呼びます。
- メモリのある場所への書き込みについて、その場所に対して任意のメモリシステムエージェントにより行われる以後の読み出しについて、書き込み時に書き込まれた値が返される場合、書き込みはグローバルに観測されていると呼びます。
- メモリのある場所からの読み出しについて、その場所に対して同じメモリシステムエージェントにより行われる以後の書き込みが、読み出し時に返される値に影響しない場合、読み出しはメモリシステムエージェントにより観測されていると呼びます。
- メモリのある場所からの読み出しについて、その場所に対して任意のメモリシステムエージェントにより行われる以後の書き込みが、読み出し時に返される値に影響しない場合、読み出しはグローバルに観測されていると呼びます。

また、ストロングオーダメモリについては次の規則が適用されます。

- ペリフェラルがメモリマップされている場所への読み出し、または書き込みでシステムになんらかの変更が発生した場合、観測されていると呼びます。また、メモリマップされたペリフェラルの状態を変更する可能性があるか、他のペリフェラルデバイス、コア、メモリになんらかの変更を発生させるトリガとなる読み出し、または書き込みについてグローバルに観測されていると呼びます。

すべてのメモリについて、完了の規則は次のとおりです。

- 読み出しまたは書き込みは、グローバルに観測され、その読み出しまたは書き込みに関連するページテーブルウォークが完了した場合、完了と定義されます。
- ページテーブルウォークは、関連するメモリトランザクションがグローバルに観測され、TLB が更新されたときに、完了と定義されます。
- キャッシュ、分岐予測器、TLB の保守操作は、操作の影響がグローバルに観測され、発生したページテーブルウォークがすべて完了したときに完了と定義されます。

——— 注 ———

すべてのメモリマップされたペリフェラルについて、ペリフェラルの動作結果がシステム全体から可視であることが必要な場合、すべての動作が完了したことを判定するために読み出すことができる実装定義の場所をペリフェラルが提供する必要があります。

———————

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ストロングオーダ領域とデバイス領域メモリでのペリフェラル動作完了確認

ペリフェラルの動作が完了したことを判定するには、そのデバイスに関連する場所、たとえばステータスレジスタをポーリングする必要があります。これはメモリオーダモデルアーキテクチャでの重要な点です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.3 ARMv6 以前のメモリ属性

ARMv6 以前は、すべてのメモリは ARM 仮想および保護メモリ管理モデル（VMSA と PMSA）にある 2 つの制御ビットの組み合わせによりタグ付けされていました。これらのビットは次の意味を持ちます。

- バッファ可能 (B) ビット（コアとメモリの間でライトバッファを可能にします）
- キャッシュ可能（C) ビット

これらのビットは、表 B2-1 に示すように、与えられた場所でのメモリの動作を定義するものと解釈されます。

表 B2-1 キャッシュ可能とバッファ可能のビットの解釈

C	B	ライトスルーキャッシュ	ライトバック専用キャッシュ	ライトバック / ライトスルーキャッシュ
0	0	キャッシュなし / バッファなし	キャッシュなし / バッファなし	キャッシュなし / バッファなし
0	1	キャッシュなし / バッファあり	キャッシュなし / バッファあり	キャッシュなし / バッファあり
1	0	実装定義	予測不能	ライトスルーキャッシュ / バッファあり
1	1	キャッシュあり / バッファあり	キャッシュあり / バッファあり	ライトバックキャッシュ / バッファあり

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.4 ARMv6 のメモリ属性の概要

ARMv6 では、システムメモリマップのすべてのメモリとデバイスをサポートするために必要な特性を持つ、一連のメモリ属性が定義されています。メモリ領域におけるアクセス順序も、メモリの属性により定義されています。

メモリ領域を記述するための主要なメモリタイプ属性は 3 つあり、これらは互いに排他です。

- 通常
- デバイス
- ストロングオーダ

通常メモリは、次のような特徴があります。

- 書き込みトランザクションを複数回繰り返しても、システムへの影響はありません。
- 読み出しトランザクションが繰り返し行われた場合、読み出されるリソースに最後に書き込まれた値が返されます。
- トランザクションが割り込まれた場合、再開が可能です。
- 複数バイトのアクセスはアトミックである必要はなく、再開や再実行が可能です。
- アンアラインドアクセスがサポートされています。
- トランザクションは、対象のメモリシステムにアクセスする前に結合可能です。
- 読み出しトランザクションはシステムへの影響なく、追加のメモリロケーションをプリフェッチ可能です。

システムペリフェラル（I/O）は一般に、ARMv6 でストロングオーダまたはデバイスメモリとして定義されている、異なるアクセス規則に準拠します。I/O アクセスの例には次のものがあります。

- 連続したアクセスにより、待機中の値の追加（書き込み）または削除（読み出し）が行われる FIFO
- アクセスを使用して割り込みへの応答を行い、コントローラ自体の状態を変更する割り込みコントローラレジスタ
- 通常メモリの領域について、タイミングとセットアップに使用される、メモリコントローラ構成レジスタ
- メモリロケーションへのアクセスにより、システム内になんらかの変更が発生する、メモリマップされたペリフェラル

システムの正確性を保証するため、アクセス規則は通常のメモリに対するものより厳しくなっています。

- アクセス（読み出しと書き込み）によりシステムに何らかの変更が発生する可能性があります。
- トランザクションは、たとえば例外からの復帰時などに繰り返し発生させることはできません。
- トランザクションの回数、サイズ、順序は変更されないようにする必要があります。

また、共有属性は、メモリが単一のプロセッサ専用のプライベートなものか、複数のプロセッサや DMA 機能を持つインテリジェントペリフェラルなど他のバスマスタリソースからアクセス可能なものかを示します。

P. B2-9 表 B2-2 は、メモリ属性の要約です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B2-2 メモリ属性の要約

メモリタイプ属性	共有属性	他の属性	説明
ストロングオーダ	-		ストロングオーダメモリへのメモリアクセスはすべて、プログラム順に発生します。ストロングオーダアクセスはすべて、共有と見なされます。
デバイス	共有		複数のプロセッサで共有される、メモリマップされたペリフェラルの処理を行うように設計されています。
	非共有		単一のプロセッサによってのみ使用される、メモリマップされたペリフェラルの処理を行うように設計されています。
通常	共有	キャッシュ不可 / ライトスルーキャッシュ可能 / ライトバックキャッシュ可能 /	複数のプロセッサ間で共有される通常メモリを取り扱うように設計されています。
	非共有	キャッシュ不可 / ライトスルーキャッシュ可能 / ライトバックキャッシュ可能 /	単一のプロセッサによってのみ使用される、通常メモリを取り扱うように設計されています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.4.1 通常メモリ属性

この属性は、MMU の各ページについて定義され、さらに共有と非共有に分けられ、システムで使用されるほとんどのメモリを記述します。この属性は、通常メモリに適切なメモリアクセスの順序を提供するように設計されています。このようなメモリは、システムの他の部分に影響を与えることなく、情報を記憶します。通常メモリは読み出し / 書き込み用の場合と、読み出し専用の場合があります。

書き込み可能な通常メモリは、物理アドレスマッピングが変更されない限り、次の性質があります。

- 指定された場所からのロードは、同じプロセッサからその場所に対して最後にストアされたデータを返します。
- 指定された場所から、途中にストアが入らずに 2 回のロードが行われた場合、各ロードについて同じデータが返されます。

読み取り専用の通常メモリには次の性質があります。

- 指定された場所から 2 回のロードが行われた場合、各ロードについて同じデータが返されます。

通常メモリへのアクセスは、メモリ順序のウィークオーダモデルに従います。ウィークオーダモデルの説明は、メモリオーダの考慮点について説明している標準のテキストにあります。Stanford University Technical Report CSL-TR-95-685、Kourosh Gharachorloo 著「*Memory Consistency Models for Shared Memory-Multiprocessors*」の第 2 章の参照をお勧めします。

明示的なアクセスは、P. B2-16 *「メモリアクセスの順序に関する条件」*で説明されているアクセスの順序要件に対応している必要があります。

非共有の通常メモリ

非共有の通常メモリ属性は、単一のプロセッサからのみアクセス可能な通常メモリを記述するために設計されています。

非共有の通常としてマークされたメモリ領域には、キャッシュの効果を透過的にするために必要な条件はありません。非共有でキャッシュ不可としてマークされたメモリ領域では、単一のプロセッサ内で内部バッファに存在する以前のアクセスデータをフォワードする場合、DMB メモリバリアを使用する必要があります。

共有の通常メモリ

共有の通常メモリ属性は、複数のプロセッサや他のシステムマスタからアクセス可能な通常メモリを記述するために設計されています。

共有通常としてマークされたメモリ領域は、メモリシステムでキャッシュを使用する影響がデータアクセスに対して完全に透過的です。命令キャッシュのコヒーレンシを保証するためには依然明示的なソフトウェア管理が必要です。このサポートのため、実装では共有領域へのアクセスをキャッシュの対象にしないという単純な方法も、それらの領域でキャッシュのコヒーレンシを維持するような複雑なハードウェアを使用するなど、多くの方法を使用できます。

共有通常メモリへの書き込みはアトミックである必要はありません。つまり、すべてのオブザーバが同時に書き込みの発生を確認するとは限りません。同じロケーションに対して 2 回の書き込みが発生した場合にコヒーレンシを維持するため、それらの書き込みの順序はどのオブザーバから見ても同じである必要があります。メモリ内でアクセスのサイズに境界アラインされた共有通常メモリからの読み出しはアトミックの必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

キャッシュ可能なライトスルー、キャッシュ可能なライトバック、キャッシュ不可のメモリ

通常メモリの領域を共有または非共有にマークするほか、MMU 内で通常としてマークされているメモリの各ページは、同時に次のいずれかとしてマーク可能です。

- ライトスルーキャッシュ可能
- ライトバックキャッシュ可能
- キャッシュ不可

このマーク付けは、そのメモリ領域が共有と非共有のどちらにマークされているかには無関係です。このマークは、共有データの必要条件を取り扱うためとは別の理由で、データ領域に必要となる取り扱いを示します。このため、キャッシュ可能で共有としてマークされたメモリ領域が、共有領域をデータキャッシュの対象としない実装において、キャッシュに保持しないことが許容されます。

同じメモリ領域が、たとえば仮想アドレスから物理アドレスへのマッピングのシノニムを使用して、異なるキャッシュ属性にマークされている場合、結果は予測不能です。

B2.4.2 デバイスメモリ属性

デバイスメモリ属性は、アクセスによりシステムに何らかの変化が発生する、または実行したロードの回数によりロードで返される値が異なるメモリロケーションについて定義されます。デバイスとしてマークする必要があるメモリ領域の一般的な例には、メモリマップされたペリフェラルやI/Oロケーションがあります。デバイス属性は、MMU で各ページについて定義されます。

プロセッサから、デバイスとしてマークされたメモリ領域への明示的なアクセスは、命令により定義されたサイズと順序で発生します。このようなロケーションに対して発生するアクセスの回数は、プログラムで指定された回数になります。プログラムにアクセスが 1 回しかない場合、このロケーションに対して実装がアクセスを繰り返すことはできません。言い換えれば、このアクセスは*再開可能*ではありません。実装でアクセスの繰り返しが必要になる可能性の例としては、割り込みの前後に、低速なアクセスを中断して割り込みを実行できるようにすることが考えられます。このような実装の最適化は、デバイスとしてマークされたメモリ領域に対しては実行できません。

さらに、デバイスとしてマークされたアドレスはキャッシュ不可です。デバイスメモリへの書き込みはバッファ可能ですが、書き込みを結合できるのはアクセスの回数、順序、サイズが正しく維持される場合のみです。同じアドレスに複数回のアクセスを行う場合、そのアドレスへのアクセス回数は変更できません。この場合、アクセスの結合は許容されません。

通常メモリ領域にマッピングされた、アクセスによりシステムに変更の発生する位置へのアクセスには、正しく実行されることを保証するためにメモリバリアが必要です。このような例として、メモリコントローラの構成レジスタにより、コントローラを制御するアクセスがあります。

デバイスとしてマークされているメモリへの明示的なアクセスは、P. B2-16*「メモリアクセスの順序に関する条件」*で説明されているアクセスの順序要件に準拠している必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

共有属性

共有属性は、MMU で各ページについて定義されます。これらの領域は、次のように呼ばれます。

- 共有デバイスとしてマークされたメモリ
- 非共有デバイスとしてマークされたメモリ

非共有デバイスとしてマークされたメモリは、単一のプロセッサからのみアクセス可能として定義されます。共有と非共有のデバイスメモリをサポートするシステムの例には、プライベートペリフェラルにローカルバスを、システムペリフェラルにはメイン（共有）システムバスを使用する実装があります。このようなシステムでは、ウオッチドッグタイマや割り込みコントローラなどのローカルペリフェラルのアクセス時間がより予測しやすくなります。

B2.4.3 ストロングオーダメモリ属性

ストロングオーダメモリ属性は、MMU で各ページについて定義されます。ストロングオーダとしてマークされたメモリについては、そのプロセッサからのすべての明示的なメモリアクセスについてストロングオーダメモリモデルが使用されます。ストロングオーダとしてマークされたメモリへのアクセスは、そのプロセッサからのアクセスの前後に DMB メモリバリアが挿入されたのと同様に動作する必要があります。詳細については、P. B2-18「*DataMemoryBarrier (DMB) CP15 レジスタ 7*」を参照して下さい。

ARMv5 との下位互換性を維持するため、CSPR の割り込みマスクを暗黙的または明示的に変更する ARMv5 命令で、プログラムにおいてストロングオーダアクセスの後に実行される命令は、ストロングオーダメモリアクセスの完了を待つ必要があります。このような命令には、制御フィールドマスクビットがセットされた `MSR` と、R15 をデスティネーションレジスタとして使用し、フラグを設定する算術演算命令と論理演算命令があります（これらの命令は SPSR を CSPR にコピーします)。この条件は以前のバージョンの ARM アーキテクチャとの下位互換性のみを目的としたもので、ARMv6 ではこの動作は推奨されません。ARMv6 準拠のプログラムはこの動作に依存せず、メモリアクセスと次の命令との間に明示的なメモリバリアを含める必要があります。同期が必要な場合の詳細については、P. B2-18「*DataSynchronizationBarrier (DSB) CP15 レジスタ 7*」を参照して下さい。

ストロングオーダとしてマークされたメモリへのプロセッサからの明示的なアクセスはプログラムのサイズで発生し、このようなロケーションに対して発生するアクセスの回数は、プログラムで指定された回数になります。プログラムにアクセスが 1 回しかない場合、このロケーションに対して実装がアクセスを繰り返すことはできません。言い換えれば、このアクセスは*再開可能*ではありません。

ストロングオーダとしてマークされたアドレスはキャッシュに保持されず、常に共有メモリロケーションとして扱われます。

ストロングオーダとしてマークされているメモリへの明示的なアクセスは、P. B2-16「*メモリアクセスの順序に関する条件*」で説明されているアクセスの順序要件に対応している必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.4.4 メモリアクセスの制限

メモリアクセスには次の制限が適用されます。

- 任意のアクセス X について、X によりアクセスされる複数のバイトはすべて同じメモリタイプ属性を持つ必要があります。そうでない場合、アクセスの動作は予測不能です。つまり、異なるメモリタイプの境界にまたがるアンアラインドアクセスは予測不能です。
- 任意の 2 つのアクセス X と Y について、X と Y が同じ命令から生成される場合、X と Y は同じメモリタイプ属性を持つ必要があります。そうでない場合、結果は予測不能です。たとえば、LDC、LDM、LDRD、STC、STM、STRD 命令のいずれかが通常とデバイスのメモリの境界にまたがる場合、結果は予測不能です。
- デバイスまたはストロングオーダのメモリに対してアンアラインドメモリアクセスを生成する命令は予測不能です。
- アクセス規則が維持されることを保証するため、デバイスまたはストロングオーダのメモリに対して複数のトランザクションを発生するメモリ操作は、4KB アドレス境界にまたがらないようにする必要があります。この理由から、揮発性のメモリデバイスへのアクセスは、単一の命令で 4KB アドレス境界にまたがらないようにする必要があります。この制約を満たすには、コンパイラがメモリアクセスのアライメントを確認するのではなく、システムのメモリマップにそのようなデバイスを配置することを避けます。
- デバイスまたはストロングオーダのメモリへのアクセスを生成する命令について、実装では命令の擬似コードで指定されたアクセスの順序を変更しません。これには、アクセスの回数、時間的な順序、および各アクセスのデータサイズや他のプロパティを変更しないことも含まれます。さらに、プロセッサコアの実装では、搭載されているメモリシステムがアクセスのメモリタイプにより種類を特定することができ、アクセスの回数、時間的な順序、データサイズ、その他のプロパティに関する同様の制約に従うことが想定されています。

  この規則の例外は次のとおりです。

  — プロセッサコアの実装は、元になるアクセスの回数、時間的な順序、その他の詳細を再現できる情報をメモリシステムに与えられる場合、この規則に従う必要はありません。さらに、実装はアクセスがデバイスまたはストロングオーダメモリへのものに行われる場合に、元のアクセスを再現するために、メモリシステムにこの条件を適用する必要があります。

    たとえば、LDM により生成される複数のワードロードは、64 ビットバスの実装により 64 ビットアクセスに結合できます。これは、アクセスがデバイスまたはストロングオーダメモリへのものである場合、2 つのワードロードをアンパックする条件がメモリシステムに適用されていれば、命令セマンティクスにより、64 ビットアクセスが常に低位アドレスのワードロードから高位アドレスのワードロードの順に行われることが保証されるためです。

  — 上に説明した制限に違反する動作をしない実装手法はすべて正当です。

- R15 のロードまたはストアを行う複数アクセス命令は、通常メモリのみにアクセスする必要があります。デバイスまたはストロングオーダのメモリに対してアクセスする場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- 命令フェッチは通常メモリにのみアクセスする必要があります。デバイスまたはストロングオーダのメモリに対してアクセスする場合、結果は予測不能です。たとえば、命令フェッチは読み出すことで何らかの動作を行うデバイスを含むメモリ領域に対しては実行できません。これは、命令フェッチと明示的なアクセスとの間に順序の必須条件がないのが理由です。
- 同じメモリロケーションが、たとえば仮想アドレスから物理アドレスへのマッピングのシノニムを使用して、MMU で共有通常と非共有通常にマークされている場合、結果は予測不能です。
- 同じメモリロケーションが、たとえば仮想アドレスから物理アドレスへのマッピングのシノニムを使用して、異なるメモリタイプ（通常、デバイス、ストロングオーダ）にマークされている場合、結果は予測不能です。
- 同じメモリロケーションが、たとえば仮想アドレスから物理アドレスへのマッピングのシノニムを使用して、異なるキャッシュ属性にマークされている場合、結果は予測不能です。
- 同じメモリロケーションが、たとえば仮想アドレスから物理アドレスへのマッピングのシノニムを使用して、MMU で共有デバイスと非共有デバイスにマークされている場合、結果は予測不能です。

—— 注 ——

また実装では、たとえば分岐予測器の結果として、非連続なアドレスからプリフェッチされる場合、読み出しによりシステムになんらかの動作を行うデバイスに予期しないアクセスが発生しないことを保証する必要があります。実装では、現在実行されている命令から実装定義の数だけ、プリフェッチを連続したアドレス上から行います。

ARMv6 以前は、低割り込みレイテンシモードをサポートするかどうかは実装定義でした。ARMv6 から、低割り込みレイテンシのサポートは、システム制御コプロセッサ（FI ビット）により制御されるようになりました。低割り込みレイテンシ構成で複数回のアクセスを行う可能性のある命令が正しく動作するかどうかは実装定義です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.4.5 下位互換性

ARMv6 のメモリ属性は、従来のアーキテクチャとは大きく異なっています。表 B2-3 は、従来のメモリタイプを新しい定義に当てはめて解釈したものです。

表 B2-3 下位互換性

以前のアーキテクチャ	ARMv6 での属性
NCNB（キャッシュ不可、バッファ不可）	ストロングオーダ[a]
NCB（キャッシュ不可、バッファ可能）	共有デバイス[a]
ライトスルーキャッシュ可能、バッファ可能	非共有通常（ライトスルーキャッシュ可能）
ライトバックキャッシュ可能、バッファ可能	非共有通常（ライトバックキャッシュ可能）

a. TCM 内に含まれるメモリロケーションはキャッシュ不可で、ストロングオーダや共有デバイスではないものとして扱われます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.5 メモリアクセスの順序に関する条件

ARMv6 では、アクセスのメモリ属性に基づいて、許容されるメモリオーダが制限されています。図 B2-1 は、2 つの明示的なアクセス A1 と A2 の間のメモリオーダを示したものです。ここで、A1 はプログラムの順序で A2 の前に発生します。

図 B2-1 に使用されている記号の意味は次のとおりです。

< アクセスがプログラム順序でグローバルに観測される必要がある、つまり A1 が必ずグローバルに A2 の前に観測される必要があることを示します。

(空白) 単一プロセッサセマンティクスの条件、たとえば単一プロセッサ内の命令間の依存性に関する条件が維持されている限り、アクセスがどのような順でグローバルに観測されてもよいことを示します。

A1 \ A2	通常読み出し	デバイス読み出し		ストロングオーダ読み出し	通常書き込み	デバイス書き込み		ストロングオーダ書き込み
		非共有	共有			非共有	共有	
通常読み出し				<				<
デバイス読み出し（非共有）		<		<		<		<
デバイス読み出し（共有）			<	<			<	<
ストロングオーダ読み出し	<	<	<	<	<	<	<	<
通常書き込み				<				<
デバイス書き込み（非共有）		<		<		<		<
デバイス書き込み（共有）			<	<			<	<
ストロングオーダ書き込み	<	<	<	<	<	<	<	<

図 B2-1 メモリオーダの制約

メモリのタイプにかかわらず、暗黙的なアクセスには順序の必要条件はありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.5.1 命令実行のプログラム順序

命令実行のプログラム順序は、制御フロートレースでの命令の順序です。

実行における明示的なメモリアクセスは次のいずれかです。

ストロングオーダ < で示されます。厳密に順序に従って発生する必要があります。

順序 <= で示されます。指定の順序、または同時に発生する必要があります。

複数転送ロード命令とストア命令、たとえば LDM、LDRD、STM、STRD などは、複数のワードアクセスを生成し、順序の判定ではそれぞれ独立したアクセスと見なされます。

2 つのアクセス A1 と A2 について、プログラム順序を判定する規則は次のとおりです。

A1 と A2 が 2 つの異なる命令から生成される場合

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

A1 と A2 が同じ命令から生成される場合

- A1 と A2 が SWP または SWPB 命令により生成されるロードとストアの場合
  - A1 がロードで A2 がストアの場合、A1 < A2
  - A2 がロードで A1 がストアの場合、A2 < A1
- A1 と A2 が、LDC、LDRD、LDM のいずれかの命令により生成される 2 ワードのロードの場合、または STC、STRD、STM のいずれかの命令により生成される 2 ワードのストアの場合（LDM または STM 命令でレジスタリストに PC が含まれている場合を除く）
  - A1 のアドレスが A2 のアドレスよりも低位の場合、A1 <= A2
  - A2 のアドレスが A1 のアドレスよりも低位の場合、A2 <= A1
- A1 と A2 が、レジスタリストに PC を含む LDM 命令により生成される 2 ワードのロードの場合、またはレジスタリストに PC を含む STM 命令により生成される 2 ワードのストアの場合、メモリ操作のプログラム順序は定義されません。
- A1 と A2 が、LDRD 命令により生成される 2 ワードのロードの場合、またはレジスタリストに PC を含む STRD 命令により生成される 2 ワードのストアの場合、Rd は R14 であり、命令の実行結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.6 メモリバリア

*メモリバリア*は、プロセッサコアでのロード / ストア命令の終了に関してプロセッサにより強制的な同期イベントを実行するために使用される命令、または一連の命令を指す一般的な用語です。メモリバリアは、プログラマモデルにおいて前のロード / ストア命令の完了を保証するため、イベントの前にプリフェッチされた命令をフラッシュするため、またはその両方の目的で使用されます。ARMv6 では、本章で説明されているメモリオーダモデルをサポートするため、システム制御コプロセッサに 3 つの明示的なバリア命令が必須になっています。これらの命令は、特権モードとユーザモードの両方で利用可能な必要があります。

- 「*DataMemoryBarrier (DMB) CP15 レジスタ 7*」で説明されている DataMemoryBarrier
- 「*DataSynchronizationBarrier (DSB) CP15 レジスタ 7*」で説明されている DataSynchronizationBarrier (DataWriteBarrier)
- P. B2-19「*PrefetchFlush CP15 レジスタ 7*」で説明されている PrefetchFlush

これらの命令はそれだけで十分な場合と、キャッシュやメモリ管理保守操作と組み合わせて使用することが必要な場合があります。これらの操作は、特権モードでのみ使用可能です。アーキテクチャの以前のバージョンでは、メモリバリアのサポートは実装定義でした。

明示的なメモリバリアは、CPU で実行されるロードおよびストア命令により生成される、メモリシステムへの読み出しと書き込みに影響します。L1 DMA トランザクションにより生成される読み出しと書き込み、および命令フェッチやハードウェアによるページテーブルにより発生するアクセスは、明示的なアクセスではありません。

B2.6.1 DataMemoryBarrier (DMB) CP15 レジスタ 7

DMB はデータメモリバリアとして機能し、次の動作を行います。

- プログラムでこの命令の前にある命令によるすべての明示的なメモリアクセスは、プログラム順序でこの命令の観測後に発生する命令によって発生するどの明示的なメモリアクセスよりも、グローバルに先行して観測されます。
- DataMemoryBarrier は、プロセッサで実行されている他の命令の順序には影響しません。

このため、DMB により命令の前後の明示的なメモリ操作の実行順序は保障されますが、それらの完了は保証されません。

DataMemoryBarrier のエンコードについては、P. B6-19「*レジスタ 7: キャッシュ管理機能*」を参照して下さい。

B2.6.2 DataSynchronizationBarrier (DSB) CP15 レジスタ 7

——— 注 ———

この操作は、従来は DrainWriteBuffer または DataWriteBarrier（DWB）と呼ばれていました。ARMv6 から、これらの名前（および DWB の使用）は新しい名前である DataSynchronizationBarrier、または DSB に置き換えられました。DSB は ARMv6 の提供する機能をより的確に反映しています。また、すべてのキャッシュ、TLB、分岐予測の保守操作と、明示的なメモリ操作を含むようにアーキテクチャ的に定義されています。

———————

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

DataSynchronizationBarrier 操作は、特殊なメモリバリアとして動作します。DSB 操作は、次の条件が満たされたときに完了します。

- この命令の前にある明示的なメモリアクセスがすべて完了した。
- この命令の前にあるキャッシュ、分岐予測器、TLB の保守操作がすべて完了した。

さらに、DSB が完了するまでは DSB 以後の命令は一切実行されません。

DataSynchronizationBarrier のエンコードについては、P. B6-19「*レジスタ 7: キャッシュ管理機能*」を参照して下さい。

B2.6.3 PrefetchFlush CP15 レジスタ 7

PrefetchFlush 命令は、プロセッサのパイプラインをフラッシュし、命令の完了後にパイプラインフラッシュ以後のすべての命令がキャッシュまたはメモリからフェッチされるようにします。これによってコンテキスト変更の操作、たとえば*アプリケーション空間識別子*（ASID）の変更、完了した TLB 保守操作、分岐予測器保守操作の実行終了が保証され、同時に PrefetchFlush 以前に実行された CP15 レジスタへの変更がすべて、PrefetchFlush 以後にフェッチされた命令から可視になります。

また、PrefetchFlush 操作により、この操作以後にプログラムに出現するすべての分岐が常に、PrefetchFlush 以後に可視となるコンテキストで分岐予測ロジックに書き込まれることが保証されます。これは、命令ストリームの正しい実行の保証に必要となります。

——— 注 ———

プログラムで PrefetchFlush より後にあるすべてのコンテキスト変更操作は、PrefetchFlush の実行後にのみ有効になります。これは、コンテキスト変更命令の動作によるものです。

———————

——— 注 ———

ARM の実装では、現在実行している命令から何命令先までプリフェッチを行うかは自由に選択できます。プリフェッチする命令数は固定でも、動的に変更してもかまいません。さらに何命令分をプリフェッチするのか自由に選択できるのと同様に、ARM の実装では以降に実行される可能性のある命令フローに関してどのようにフェッチするかも自由に選択できます。たとえば、分岐命令の後で分岐命令に続く命令と、分岐先の命令のどちらをプリフェッチしてもかまいません。これを分岐予測と呼びます。

———————

命令のプリフェッチに共通する潜在的な問題は、メモリにある命令がプリフェッチ後、実行前に変更される可能性があることです。この場合、メモリにある命令が変更されても、通常は既にプリフェッチされた命令の実行は停止されません。PrefetchFlush およびメモリバリア命令（DMB か DSB のどちらか適切な方）は必要に応じて命令の実行順序を強制するのに使われます。詳細については、P. B2-21「*メモリオーダモデル内でのキャッシュ保守操作の順序付け*」を参照して下さい。

PrefetchFlush のエンコードについては、P. B6-19「*レジスタ 7: キャッシュ管理機能*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.7 メモリのコヒーレンシとアクセスの考慮点

システム設計者とプログラマは、システム全体の正確性のため設計のすべての要素を考慮する必要があります。このセクションでは、いくつかの問題点や見落としやすい点と、システムの動作の予測可能性を保証するために行うべき手順について概説します。

――― 注 ―――

このセクションの説明では、例外からの復帰は以下のいずれか 1 つを指すものと定義されます。

- S ビットがセットされた状態でデータ処理命令を使用し、PC をデスティネーションとして使用する。
- CPSR の復元付き複数ロード命令を使用する。詳細については、P. A4-41「*LDM (3)*」を参照して下さい。
- `RFE` 命令を使用する。

―――――――――

B2.7.1 キャッシュコヒーレンシの概要

キャッシュ、書き込みバッファ、またはその両方を使用している場合、システムにはメモリの同じロケーションの値が複数保持されます。これらの値は、物理的にメインメモリ、ライトバッファ、キャッシュに存在している可能性があります。ハーバードキャッシュを使用している場合、命令キャッシュとデータキャッシュのどちらか、または両方にメモリロケーションの値が含まれる可能性があります。複数レベルのキャッシュでは、一部のレベルにのみキャッシュラインが存在し、上書きされているか、別の場所に追い出しされている可能性があります。

これらの物理的な場所すべてに、メモリに最後に書き込まれた値が保持されているとは限りません。メモリコヒーレンシの問題は、データ読み出しまたは命令フェッチでメモリロケーションが読み出されたとき、読み出される値が常に、そのロケーションに最後に書き込まれた値と等しいことを保証することです。

ARM のメモリシステムアーキテクチャでは、メモリシステムコヒーレンシの一部の要素をシステムが自動的に提供する必要があります。他の要素はメモリコヒーレンシ規則により対応されます。これは、メモリコヒーレンシを維持するためにプログラムが従う必要のある制約です。ARMv6 では P. B2-8「*ARMv6 のメモリ属性の概要*」で説明しているように、*共有*と*非共有*のメモリ属性が定義されています。これらの属性は、コヒーレンシの必要についての情報を提供し、実装でシステム全体の正確性を維持できるようにする、たとえば共有でキャッシュ可能としてマークされ、スヌーピングが提供されていないメモリ領域ではキャッシュ不可ポリシーを実装で強制できるようにするためのものです。メモリコヒーレンシ規則に従わないプログラムの動作は予測不能です。アドレスマッピングとキャッシュは、メモリコヒーレンシが常に維持されていることを保証するため注意深く管理する必要があります。キャッシュと書き込みバッファの管理には、一般に次の 1 つ以上を含む手順が必要です。

- ライトバックキャッシュの場合、データキャッシュのクリーニング
- データキャッシュの無効化
- 命令キャッシュの無効化
- 書き込みバッファのドレイン
- 命令パイプラインへのプリフェッチフラッシュの実行
- 分岐予測ロジック（分岐先バッファ）のフラッシュ

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ARMv6 以前は、操作とシーケンスは実装定義でした。ARMv6 では、メモリオーダモデル、キャッシュ、TLB、システム制御コプロセッサ（CP15）でサポートされるメモリバリア操作により、オペレーティングシステムによるレベル 1 メモリサポートの標準化が可能になりました。

—— 注 ——

実装で追加のキャッシュレベルが必要な場合は、将来の互換性に影響を与える可能性を最小にするため、ARM と密接な共同の上で設計を行うことを強くお勧めします。

B2.7.2 メモリオーダモデル内でのキャッシュ保守操作の順序付け

メモリオーダモデルに関して、キャッシュ保守操作には次の規則が適用されます。

- すべてのキャッシュと分岐予測器保守操作は、プログラム順序で実行されます。キャッシュまたは分岐予測器保守操作がプログラムでページテーブルの変更前にある場合、それらの操作はページテーブルへの変更が可視になる前に行われることが保証されます。
- プログラムで、ページテーブルの変更がキャッシュや分岐予測器保守操作の前にある場合、変更が可視になることを保証する前に、P. B2-22「*TLB 保守操作とメモリオーダモデル*」で概説している手順を実行する必要があります。
- DMB がある場合、プログラム順序でその DMB 操作の前にあるキャッシュ保守操作はすべて、DMB の後にある明示的なロードおよびストア操作から可視になります。また、DMB の前にあるキャッシュ保守操作はすべて、DMB の後にあるキャッシュ保守や明示的なメモリ操作が観測されるより前に、グローバルに観測可能であることが保証されます。DMB の完了は、関連する他のオブザーバにすべてのデータが可視であることを保証しません（ページテーブルウォークなど）。
- DSB があると、その DSB 操作の前にあるキャッシュ保守操作はすべて完了され、書き戻されたデータはすべての関連するオブザーバに可視であることが保証されます。
- PrefetchFlush または例外からの復帰があると、それ以前のすべての分岐予測器保守操作の影響は、以後のすべての命令から可視になります。
- 例外があると、例外が発生した命令より前の分岐予測器保守操作の影響は、例外のエントリ後に実行されるすべての命令から可視になります（それらの命令の命令フェッチを含みます）。
- MVA によるデータキャッシュ (または統合キャッシュ）の保守操作は、同じプロセッサによる MVA のキャッシュ操作の対象となるアドレス内への明示的なロードまたはストアに関して、プログラム内の順序で実行する必要があります。
- MVA によるデータキャッシュ（または統合キャッシュ）の保守操作は、同じプロセッサによる MVA のキャッシュ操作の対象となるアドレス外への明示的なロードやストアへの相対的な順序は制限されません。順序を制限する必要がある場合、順序を強制するため DMB 操作を挿入する必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- 同じプロセッサ上での、セット / ウェイによるデータキャッシュ（または統合キャッシュ）保守操作と明示的なロードやストアとの相対的な順序は制限されていません。順序が制限されている場合、順序を強制するため DMB 操作を挿入する必要があります。
- セット / ウェイによるデータキャッシュ（または統合キャッシュ）保守操作の実行は、DSB 操作が実行されるまではシステム内の他のオブザーバから可視であるとは限りません。
- 命令キャッシュ保守操作の実行完了が保証されるのは、DSB バリアの実行後です。
- 命令キャッシュ保守操作の実行完了が、命令フェッチに対して可視となることが保証されるのは、PrefetchFlush 操作の実行後か、例外または例外からの復帰後のみです。

最後の 2 つの点から、単一プロセッサシステムでの自己変更コードの行に対するキャッシュクリーニング操作の手順は次のようになります。

```
STR rx, [Instruction location]
Clean Data cache by MVA to point of unification [instruction location]
DSB            ; ensures visibility of the data cleaned from the D Cache
Invalidate Instruction cache by MVA [instruction location]
Invalidate BTB entry by MVA [instruction location]
DSB            ; ensures completion of the ICache invalidation
PrefetchFlush
```

B2.7.3 TLB 保守操作とメモリオーダモデル

メモリオーダモデルに関して、TLB 保守操作には次の規則が適用されます。

- TLB 保守操作の実行完了が保証されるのは、DSB 命令の実行後です。
- PrefetchFlush または例外からの復帰があると、プログラムでそれ以前にある、完了したすべての TLB 保守操作の影響は､以後のすべての命令から可視になります(それらの命令の命令フェッチも含みます)。
- 例外があると、例外の実行開始以前に完了した TLB 保守操作の影響は、以後のすべての命令から可視になります（それらの命令の命令フェッチを含みます)。
- すべての TLB 保守操作は、お互いに対しての相対的なプログラム順序で実行されます。
- データ（または統合）TLB 保守操作の実行は、プログラムでその操作よりも前にある命令による明示的なメモリトランザクションに影響しないことが、ハードウェアにより保証されます。このため、メモリバリアは必要ありません。
- データ（または統合）TLB 保守操作の実行は、TLB 操作の完了を保障するための DSB 操作実行後の明示的なロードまたはストア、PrefetchFlush 操作、例外の実行、例外からの復帰以降からのみ可視であることが保証されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- 命令（または統合）TLB 保守操作の実行は、TLB 操作の完了を保障するための DSB 操作実行後の命令フェッチ、PrefetchFlush 操作、例外の実行、例外からの復帰以降からのみ可視であることが保証されます。

ページテーブルエントリを書き込む場合、以後のトランザクション（キャッシュ保守操作を含む）からの可視性を保証するため次の規則が適用されます。

- TLB ページテーブルウォークは、TLB 保守の目的からは別のオブザーバと見なされます。
  - ページテーブルへの書き込み（必要ならキャッシュのクリーニングが行われた後）は、DSB 操作の実行後の明示的なロード / ストアにより発生したページテーブルウォークからのみ、可視となることが保証されます。ただし、ページテーブルへの書き込みはすべて、プログラムでその前に発生する明示的なメモリトランザクションからは可視でないことが保証されます。
  - ページテーブルがライトバックキャッシュ可能メモリに保持されている場合、ページテーブルのクリーニングは、ページテーブルへの書き込みと、ハードウェアのページテーブルウォークによりそれらが可視となるまでの間に実行する必要があります。
  - ページテーブルへの書き込み（必要ならキャッシュのクリーニングが行われた後）は、DSB 操作と PrefetchFlush 操作の実行後のページテーブルへの書き込みの後にある命令の命令フェッチにより発生するページテーブルウォークによってのみ、可視となることが保証されます。

単一プロセッサシステムで、ページテーブルエントリに書き込みを行う一般的なコード（命令またはデータのマッピングの変更をカバーするもの）は次のようになります。

```
STR rx, [Page table entry] ;
Clean line [Page table entry]
DSB            ; ensures visibility of the data cleaned from the D Cache
Invalidate TLB entry by MVA [page address]
Invalidate BTB
DSB            ; ensure completion of the Invalidate TLB
PrefetchFlush
```

B2.7.4 同期化基本命令とメモリオーダモデル

同期化基本命令である SWP/SWPB と LDREX/STREX は、これらの命令によりアクセスされるメモリタイプについて、次のメモリオーダモデルに従います。この理由から、次の規則が適用されます。

- スピンロックを要求するための移植可能コードは、スピンロックを要求してから、そのスピンロックを使用するアクセスを行うまでの間に DMB 命令を含む必要があります。
- スピンロックを解放するための移植可能コードは、スピンロックをクリアする書き込みの前に DMB 命令を含む必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.7.5 分岐予測器保守操作とメモリオーダモデル

メモリオーダモデルに関して、分岐予測器保守操作には次の規則が適用されます。

- 分岐予測器の無効化はすべて、PrefetchFlush 操作の実行、例外の取得、例外からの復帰のいずれかが行われた後にのみ効果が保証されます。

次のイベントのいずれかが発生した後では、分岐予測器のエントリを無効化するため分岐予測器保守操作を使用する必要があります。

- MMU の有効化または無効化
- 命令ロケーションへの新しいデータの書き込み
- ページテーブルへの新しいマッピングの書き込み
- TTBR0、TTBR1、TTBCR のいずれかの変更

エントリを無効化しない場合、古い分岐の実行により予測不能な結果が発生する可能性があります。

B2.7.6 CP15 のレジスタの変更とメモリオーダモデル

プログラムで、明示的なメモリ操作の後にある CP14 および CP15 レジスタへの変更はすべて、前のメモリ操作に影響を与えないことが保証されます。

CP14 および CP15 レジスタへの変更が、以後の命令に対して可視となることが保証されるのは、PrefetchFlush 操作の実行後か、例外の実行または例外からの復帰後のみです。

ただし、コプロセッサレジスタへのアクセスについては次の規則が適用されます。

- MCR 操作が書き込みに使用したのと同じレジスタ番号を使用して MRC 操作でレジスタを直接読み出す場合、MCR と MRC とのコンテキスト同期は必要なく、書き込まれた値が読み出されることが保証されます。
- 前の MCR 操作が書き込みに使用したのと同じレジスタ番号を使用して MCR 操作でレジスタに直接書き込む場合、2 つの MCR 命令間のコンテキスト同期は必要なく、2 番目の MCR の値が最終的に書き込まれます。

条件によっては、PrefetchFlush、例外、例外からの復帰の可視性を保証するため、それらの実行前に CP15 レジスタに対して追加の操作が必要な場合があります。このような条件は、各レジスタの定義に特に指定されています。

CP15 レジスタへの変更で、可視となることが保証されていないものは、例外処理に影響し、次の規則が適用されます。

- CP15 レジスタに保持されている状態の変更で、例外のトリガに関連しているものは可視となることが保証されませんが、例外自体の処理に関連する変更は（例外が受け入れられることが決定した後では）効果が適用されることが保証されます。

このため、次の例で（最初の値は A=1、V=0）LDR はアンアラインドトランザクションによりデータアボートを発生する可能性も、しない可能性もありますが、例外が発生すれば、使用されるベクタは V ビットの影響を受けます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

```
    MCR p15, r0, c1, c0, 0    ; clears the A bit and sets the V bit
    LDR r2, [R3]              ; unaligned load.
```

ASID と TTBR の変更の同期

TLB 管理の一般的な使用モデルでは、コンテキスト ID が異なるページテーブルと関連付けられるようにするため、コンテキスト ID と変換テーブルベースレジスタが同時に変更されることが必要です。ただし、実装定義のプリフェッチ段数と、分岐予測の使用により、コンテキスト ID と変換テーブルレジスタへの変更の同期を保証するうえで問題が発生する可能性があります。たとえば、TLB、分岐先キャッシュ、ASID と変換情報のキャッシュ、またはこれらの組み合わせが不正な変換により破壊される可能性があります。この同期は、次のいずれかの事態を回避するために必要です。

- *古い* ASID が、*新しい*ページテーブルからのページテーブルウォークに関連付けられる。
- *新しい* ASID が、*古い*ページテーブルからのページテーブルウォークに関連付けられる。

次の例に示すように、この問題を解決するにはさまざまな方法があります。

解決策の例

この方法では、ASID の値として 0 をオペレーティングシステムに予約し、ASID と変換テーブルベースレジスタの同期以外に使用しません。この場合、次の手順を使用します（グローバルとしてマークされたメモリから実行します)。

```
Change ASID to 0
PrefetchFlush
Change Translation Table Base Register
PrefetchFlush
Change ASID to new value
```

この手法により、古いページテーブルと新しいページテーブルのどちらがアクセスされるか不明な時点で（プリフェッチにより）アクセスされる非グローバルページは、使用されない ASID 値である 0 に関連付けられ、実行がエラーとならないことが保証されます。

同じ問題の別の症状は、ASID の変更とその同期との間に分岐が発生した場合、分岐予測器の値が誤った ASID に関連付けられる可能性があることです。この現象に対しても、ASID の 0 を使用する手法で対処できますが、そのような分岐を回避することでも対処できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B2.7.7 CPSR の変更とメモリオーダモデル

CPS、SETEND、MSR 命令による CPSR の変更（例外の発生または例外からの復帰を伴わない CPSR への操作）は、プログラムでその前にあるどの命令操作にも影響しないことが保証されます。

CPS、SETEND、MSR 命令による CPSR の変更（例外の発生または例外からの復帰を伴わない CPSR への操作）は、プログラムでその変更の後にあるすべての命令から、命令のアクセス許可チェックへの影響を除くすべての要素について可視であることが保証されます。CPSR への操作が実行の特権（またはセキュリティ）ステータスの変更である場合、この変更は PrefetchFlush 操作、例外の取得、例外からの復帰のいずれかが実行された後の命令アクセス許可チェックの目的でのみ可視となります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 B3 章
システム制御コプロセッサ

本章では、システム制御コプロセッサであるコプロセッサ 15 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。

- *システム制御コプロセッサについて* : P. B3-2
- *レジスタ* : P. B3-3
- *レジスタ 0: ID コード* : P. B3-7
- *レジスタ 1: 制御レジスタ* : P. B3-12
- *レジスタ 2 から 15 まで* : P. B3-18

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B3.1 システム制御コプロセッサについて

標準メモリとシステムの機能はすべて、システム制御コプロセッサと呼ばれるコプロセッサ 15（CP15）により制御されます。一部の機能は他の制御手法も使用します。これらについては、それぞれの機能に関する章で説明されています。たとえば、第 B4 章*「仮想メモリシステムアーキテクチャ」*で説明されているメモリ管理ユニットは、メモリのページテーブルによっても制御されます。

ARMv6 システムには、キャッシュ、密結合メモリ、コプロセッサ機構に関してセットアップのための情報を提供するシステム制御コプロセッサが必要です。これは、メモリ管理の制御機構も提供します（MMU と MPU のいずれかのサポート）。

ARMv6 以前は、CP15 が実装されていない場合は CP15 命令は未定義でした。しかし、ARMv4 以降、CP15 は実装におけるプロセッサ ID、キャッシュ制御、メモリ管理（MMU と MPU のサポート）のデファクトスタンダードになりました。CP15 サポートの正確な詳細を参照するには、このマニュアルと、関連するインプリメンテーションリファレンスマニュアルの両方を読む必要があります。

本章では、システム制御コプロセッサ全体の設計と、レジスタのアクセス方法について説明します。レジスタの一部については、詳細情報を解説します。他のレジスタはそれぞれの機能に割り当てられているため、対応する他の章で詳細に解説し、本章では概要のみ説明します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B3.2 レジスタ

システム制御コプロセッサには、16 個までの 1 次レジスタを含めることができ、各レジスタは 32 ビット長です。レジスタアクセス命令の追加フィールドは、CP15 の物理的な 32 ビットレジスタ数を増やし、そのレジスタを特定するために使用されます。4 ビットの 1 次レジスタ番号は、レジスタの機能を決定する一次要素であり、システム制御コプロセッサの種類を判断するために使用されます。

CP15 レジスタは読み出し専用、書き込み専用、読み出し / 書き込み用のいずれかです。レジスタの詳細な説明では、次の要素を指定します。

- 許容されるアクセスのタイプ
- 各タイプのアクセスにより実行される機能
- 1 次レジスタが複数の物理レジスタから構成されているかどうか、その場合複数のレジスタをどのように区別するか
- レジスタの使用に関する他の詳細

B3.2.1 レジスタアクセス命令

システム制御コプロセッサについて定義されている命令は次のとおりです。

- MCR 命令 : ARM® レジスタの値を CP15 レジスタに書き込みます。
- MRC 命令 : CP15 レジスタの値を ARM レジスタに読み出します。
- MCRR 命令 : ARMv6 で導入された範囲操作命令に使用し、以前のバージョンのアーキテクチャでもオプションとして使用します。
- MRRC 命令 : オプションとして実装定義の機能で使用します。

CP15 の CDP、LDC、STC 命令はすべて未定義です。

MCR/MRC 命令のフォーマットを次の図に示します。この図でビット [11:8] (*cp_num*) は CP15 を示し、CRn フィールドは 1 次レジスタ番号を示し、CRm と opcode2 は追加のレジスタデコードです。

31 28	27	26	25	24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
cond	1	1	1	0	opcode1	L	CRn	Rd	1 1 1 1	opcode2	1	CRm

MCR 命令と MRC 命令で CP15 レジスタにアクセスする場合、次の一般的な構文を使用します。

```
MCR{<cond>}  p15, 0, <Rd>, <CRn>, <CRm>{, <opcode2>}     (L = 0)
MRC{<cond>}  p15, 0, <Rd>, <CRn>, <CRm>{, <opcode2>}     (L = 1)
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL(常時)条件が使用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ビット [23:21] 命令のこれらのビットは、一般的なMRC命令とMCR命令では<opcode1>フィールドで、有効な CP15 命令では通常は 0b000 です。ただし、レベル 2 キャッシュサポートでは <opcode1> == 1 が使用され、他のいくつかの特別なタスクでもこれらのビットが使用されます。割り当てられていない値に対する動作は予測不能です。

`<Rd>` 転送に関係する ARM レジスタ(MCR のソースレジスタと、MRC のデスティネーションレジスタ)。このレジスタは R15 以外の必要があります。MRC 命令では通常は R15 も使用できますが、この場合には使用できません。CP15 の MRC 命令または MCR 命令の <Rd> について R15 が指定された場合、命令の動作は予測不能です。

`<CRn>` 転送に関係する 1 次 CP15 レジスタ(MCR のデスティネーションレジスタと、MRC のソースレジスタ)。標準の汎用コプロセッサレジスタ名は c0、c1、...、c15 です。

`<CRm>` レジスタのバージョン、アクセスのタイプ、または両方に関する追加情報を指定するいくつかの 1 次レジスタへのアクセスに使用される、追加のコプロセッサレジスタ名。

1 次レジスタの記述で <CRm> が指定されない場合、c0 を指定する必要があります。他のレジスタを指定した場合、命令の動作は予測不能です。

`<opcode2>` レジスタのバージョン、アクセスのタイプ、または両方に関する追加情報を指定するいくつかの1次レジスタへのアクセスに使用される、オプションの3ビットの数値。値が省略されている場合、0 が使用されます。

1 次レジスタの記述で <opcode2> が指定されない場合、省略するか 0 を指定する必要があります。他の値を指定した場合、命令の動作は予測不能です。

MCRR フォーマット(P. A4-65「*MCRR*」参照)はデコードの有効範囲がより小さくなっています。1 次レジスタは(CRn フィールドなしで)暗黙に指定され、CRm と opcode フィールドは機能のデコードに使用されます。

ARMv6 以前は、MCR 命令と MRC 命令はプロセッサが特権モードにあるときにのみ使用可能でした。プロセッサがユーザモードにあるときにこれらの命令が実行されると、未定義命令例外が発生します。

ARMv6 では、次のコマンドについてユーザアクセスが追加されています。

- プリフェッチフラッシュ
- データ同期バリア
- データメモリバリア
- クリーニングとプリフェッチ範囲操作

―――― 注 ――――

ユーザモードのプログラムから、特権モード専用のシステム制御コプロセッサ機能にアクセスする必要がある場合、通常はオペレーティングシステムが 1 つ以上の SWI を定義して、ユーザモードから使用できるようにします。利用可能なメモリとシステムの機能の正確なセットは、プロセッサによって大きく異なるため、これらの SWI は簡単に置き換え可能なモジュールとして実装し、このモジュールの SWI インターフェースは可能な限り、プロセッサの詳細とは独立したものとして定義することをお勧めします。

――――――――

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B3.2.2 1 次レジスタの割り当て

表 B3-1 は、システム制御コプロセッサの 1 次レジスタの割り当てを示したものです。

表 B3-1 1 次レジスタの割り当て

レジスタ	一般的な用途	特定の用途	詳細の記述があるページ
0	ID コード（読み出し専用）	プロセッサ ID、キャッシュ、密結合メモリ、TLB タイプ	P. B3-7 *「レジスタ 0: ID コード」*
1	制御ビット（読み出し / 書き込み）	システム構成ビット	P. B3-12 *「制御レジスタ」* と P. B4-40 *「レジスタ 1: 制御レジスタ」*
2	メモリ保護と制御	ページテーブル制御	P. B4-41 *「レジスタ 2: 変換テーブルのベース」*
3	メモリ保護と制御	ドメインアクセス制御	P. B4-42 *「レジスタ 3: ドメインアクセス制御」*
4	メモリ保護と制御	予約	なし。このレジスタは予約済みです。
5	メモリ保護と制御	フォルトステータス	P. B4-19 *「フォルトアドレスレジスタとフォルトステータスレジスタ」* と P. B4-43 *「レジスタ 5: フォルトステータス」*
6	メモリ保護と制御	フォルトアドレス	P. B4-19 *「フォルトアドレスレジスタとフォルトステータスレジスタ」* と P. B4-44 *「レジスタ 6: フォルトアドレスレジスタ」*
7	キャッシュとライトバッファ	キャッシュとライトバッファの制御	P. B6-19 *「レジスタ 7: キャッシュ管理機能」*
8	メモリ保護と制御	TLB 制御	P. B4-45 *「レジスタ 8: TLB 機能」*
9	キャッシュとライトバッファ	キャッシュロックダウン	P. B6-31 *「レジスタ 9: キャッシュのロックダウン機能」*
10	メモリ保護と制御	TLB ロックダウン	P. B4-47 *「レジスタ 10: TLB ロックダウン」*
11	密結合メモリ制御	DMA 制御	P. B7-9 *「CP15 のレジスタ 11 を使用した L1 DMA 制御」*
12	予約	予約	なし。このレジスタは予約済みです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B3-1 1 次レジスタの割り当て

レジスタ	一般的な用途	特定の用途	詳細の記述があるページ
13	プロセス ID	プロセス ID	*P. B4-52「レジスタ 13: プロセス ID」* と P. B8-7「*レジスタ 13: FCSE PID*」
14	予約	-	-
15	実装定義	実装定義	実装のドキュメント

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B3.3 レジスタ 0: ID コード

CP15 のレジスタ 0 には、ARM とシステム実装の 1 つ以上の識別コードが含まれています。このレジスタを読み出すとき、MRC 命令の opcode2 フィールドには表 B3-2 に示す、読み出す識別コードを、CRm フィールドには c0 を指定する必要があります。CRm フィールドにそれ以外の値を指定した場合、命令の動作は予測不能です。CP15 のレジスタ 0 に書き込みを行った場合、動作は予測不能です。

表 B3-2 システム制御コプロセッサ ID レジスタ

opcode2	レジスタ	詳細の記述があるページ
0b000	メイン ID レジスタ	*「メイン ID レジスタ」*
0b001	キャッシュタイプレジスタ	P. B3-10 *「キャッシュタイプレジスタ」*
0b010	*密結合メモリ*（TCM）タイプのレジスタ	P. B3-10 *「TCM タイプレジスタ」*
0b011	TLB タイプのレジスタ	
0b100	MPU タイプのレジスタ（PMSAv6）	
その他	予約（本文を参照して下さい）	-

実装されていない、または予約 ID のレジスタに対応する <opcode2> の値が検出された場合、システム制御コプロセッサはメイン ID レジスタの値を返します。

メイン ID レジスタ以外の ID レジスタは、実装されたときに値がメイン ID レジスタの値と等しくならないように定義されています。このため、ソフトウェアでメイン ID レジスタと目的のレジスタの両方を読み出し、値を比較することで、それらのレジスタが存在しているかどうかを判定できます。2 つの値が等しくない場合、目的のレジスタは存在しています。

B3.3.1 メイン ID レジスタ

CP15 レジスタ 0 が <opcode2> == 0 で読み出された場合、返される識別コードから ARM アーキテクチャのバージョン番号、Thumb® 命令セットが実装されているかどうか、その他の情報が判定できます。

——— 注 ———

CP15 レジスタ 0 のフィールドで、アーキテクチャにより定義されているのは一部のみです。他のフィールドは実装定義で、プロセッサの正確なバリエーションについての詳細情報を提供しています。各プロセッサで使用されている識別コードについては、個別のデータシートを参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

実装者コード

メイン ID レジスタのビット [31:24] には、実装者コードが含まれています。

次のコードが定義されています（アーキテクチャコードの他の値はすべて、ARM 社により予約されています）。

0x41	A (ARM Limited)
0x44	D (Digital Equipment Corporation)
0x4D	M (Motorola - Freescale Semiconductor Inc.)
0x56	V (Marvell Semiconductor Inc.)
0x69	I (Intel Corporation)

ARM プロセッサの実装 ID

歴史的な理由から、CP15 レジスタ 0 の ID コードはさまざまな方法で解釈する必要があります。ビット [19] が 0 の場合、ビット [15:12] を次のように解釈する必要があります。

- これらのビットが 0x0 の場合、OBSOLETE パーツ（ARMv4 以前のアーキテクチャ）であることを示します。
- これらのビットが 0x7 の場合、プロセッサが ARM7 ファミリであることを示します。
- > 0x7 の場合、ARM7 より新しいプロセッサファミリです。

ARM7 のプロセッサ ID は、次のように解釈されます。

31 24	23	22 16	15 4	3 0
実装者	A	バリエーション	1 次パーツナンバ	リビジョン

ビット [3:0] プロセッサの実装定義のリビジョン番号。

ビット [15:4] プロセッサの 1 次パーツナンバを示す実装定義の値。この数値の上位 4 ビットは 0x7 です。

ビット [22:16] 実装定義のバリエーション番号。

ビット [23] ARM7 ベースのプロセスに含まれる 2 つのアーキテクチャのうちどちらが使用されているかを示します。

0	アーキテクチャ 3（OBSOLETE パーツ）
1	アーキテクチャ 4T

ビット [31:24] 0x41 = A（ARM 社）実装コード

ARM7 以降のプロセッサ実装では、ARM からの実装とアーキテクチャライセンシすべてにわたって共通の、ビット [23:0] の一般的なフォーマットが使用されています。ビット [19] の値により、2 つの一般的なフォーマットが定義されています。これらについては、以下のセクションで解説します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ARM9 以降のプロセッサ

ID コードのビット [15:12] が 0x0 と 0x7 のいずれでもない場合、ID は次のように解釈されます。

31–24	23–20	19–16	15–4	3–0
実装者	バリエーション	アーキテクチャ	1 次パーツナンバ	リビジョン

ビット **[3:0]** プロセッサの実装定義のリビジョン番号。

ビット **[15:4]** プロセッサの 1 次パーツナンバを示す実装定義の値。この数値の上位 4 ビットは 0x0 と 0x7 以外の値です。

ビット **[19:16]** アーキテクチャコード。次のアーキテクチャコードが定義されています。

0x1 ARM アーキテクチャ v4
0x2 ARM アーキテクチャ v4T
0x3 ARM アーキテクチャ v5
0x4 ARM アーキテクチャ v5T
0x5 ARM アーキテクチャ v5TE
0x6 ARM アーキテクチャ v5TEJ
0x7 ARM アーキテクチャ v6
0xF 修正された CPUID フォーマット。詳細は ARM から入手できます。

アーキテクチャコードの他の値はすべて、ARM 社により予約されています。

ビット **[23:20]** 実装定義のバリエーション番号。同じ 1 次パーツに属する 2 つのバリエーション、たとえばキャッシュサイズが異なるバリエーションの区別に一般に使用されます。

ビット **[31:24]** 実装者コード。詳細については、P. B3-8 *「実装者コード」* を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B3.3.2 キャッシュタイプレジスタ

キャッシュタイプレジスタは、キャッシュに関する次の詳細を示します。

- キャッシュが統合キャッシュか、命令キャッシュとデータキャッシュが分離されているか
- キャッシュのサイズ、ライン長、アソシアティビティ
- ライトスルーキャッシュかライトバックキャッシュか
- キャッシュクリーニングとロックダウン機能

キャッシュタイプレジスタのフォーマットは次のとおりです。

31–29	28–25	24	23–12	11–0
0 0 0	ctype	S	Dsize	Isize

ctype S ビットと Dsize および Isize フィールドにより指定されない、キャッシュの詳細を指定します。表に指定されていない値はすべて、将来拡張用に予約されています。

S ビット キャッシュが統合キャッシュか（S == 0）、命令キャッシュとデータキャッシュが分離されているか（S == 1）を示します。S == 0 の場合、Isize フィールドと Dsize フィールドの両方が統合キャッシュのサイズを示し、この 2 つの値は同一の必要があります。

Dsize データキャッシュ（S == 0 の場合は統合キャッシュ）のサイズ、ライン長、アソシアティビティを示します。

Isize 命令キャッシュ（S == 0 の場合は統合キャッシュ）のサイズ、ライン長、アソシアティビティを示します。

キャッシュの詳細については、第 B6 章*「キャッシュとライトバッファ」*を参照して下さい。キャッシュタイプレジスタフィールドのエンコードについては、P. B6-14*「キャッシュタイプレジスタ」*を参照して下さい。

B3.3.3 TCM タイプレジスタ

密結合メモリ（TCM）タイプレジスタのフォーマットは次のとおりです。

31	30	29	28–19	18–16	15–3	2–0
0	0	0	SBZ/UNP	DTCM	SBZ/UNP	ITCM

ITCM（ビット [2:0]）

実装されている命令（または統合）密結合メモリ数を示します。この値は 0 ～ 4 で、それ以外の値はすべて予約済みです。命令 TCM はすべて、命令側とデータ側の両方からアクセス可能な必要があります。

DTCM（ビット [18:16]）

実装されているデータ密結合メモリ数を示します。この値は 0 ～ 4 で、それ以外の値はすべて予約済みです。

密結合メモリの詳細については、第 B7 章*「密結合メモリ」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B3.3.4 TLB タイプレジスタ

TLB タイプレジスタのフォーマットは次のとおりです。

31–24	23–16	15–8	7–1	0
SBZ/UNP	ILsize	DLsize	SBZ/UNP	S

S ビット TLB が統合 TLB か（S == 0)、命令 TLB とデータ TLB が分離されているか（S == 1）を示します。

DLsize データ TLB（S == 1）または統合 TLB（S == 0）のロック可能なエントリ数を示します。

ILsize S == 1 の場合は命令 TLB のロック可能なエントリ数を示し、それ以外の場合は常に 0 です。

仮想メモリシステムアーキテクチャの詳細については、第 B4 章*「仮想メモリシステムアーキテクチャ」*を参照して下さい。

B3.3.5 MPU タイプレジスタ

メモリ保護ユニット（MPU）タイプレジスタのフォーマットは次のとおりです。

31–24	23–16	15–8	7–1	0
SBZ/UNP	IRegion	DRegion	SBZ/UNP	S

S ビット MPU が統合 MPU か（S == 0)、命令 MPU とデータ MPU が分離されているか（S == 1）を示します。

DRegion データ MPU（S == 1）または統合 MPU（S == 0）の保護領域数を示します。

IRegion S == 1 の場合は命令 MPU の保護領域数、それ以外の場合は常に 0 です。

保護メモリシステムアーキテクチャの詳細については、第 B5 章*「保護メモリシステムアーキテクチャ」*を参照して下さい。

——— 注 ———

MPU タイプレジスタは、PMSAv6 で導入されたものです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B3.4　レジスタ 1: 制御レジスタ

CP15 のレジスタ 1 は、ARM プロセッサの構成制御ビットを含んでいます。これには、opcode_2 フィールドによって選択される 3 つのレジスタが含まれます。opcode_2 が 0 の場合、アーキテクチャにより指定される制御レジスタが選択されます。opcode_2 が 1 の場合、実装定義の制御レジスタが選択されます。

表 B3-3 システム制御コプロセッサ制御レジスタ

opcode2	レジスタ
0b000	制御レジスタ
0b001	補助制御レジスタ（フォーマットは実装定義です）。
0b010	コプロセッサアクセス制御レジスタ
その他	予約

B3.4.1　制御レジスタ

このレジスタには、次の情報が含まれています。

- 他の CP15 レジスタにより主に制御されるキャッシュ、MMU、その他のメモリシステムブロックの許可 / 禁止ビット。これにより、これらのメモリシステムブロックを許可する前に正しくプログラムできます。
- メモリシステムブロックと ARM プロセッサ自体の各種の構成ビット。

——— 注 ———

各種のビットが将来追加される可能性があります。このため、このレジスタは通常はリード・モディファイ・ライトの手法を使用して更新し、現在割り当てられていないビットが不必要に変更されないようにする必要があります。この規則に従わない場合、コードが将来のプロセッサで予期しない動作を引き起こす可能性があります。

31–27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNP/SBZP	L2	EE	VE	XP	U	FI		L4	RR	V	I	Z	F	R	S	B	L	D	P	W	C	A	M

CP15 のレジスタ 1 の制御ビットが特定の実装に適用されない場合､その実装を最も的確に反映する値が読み出され、書き込みは無視されます。この一般的な規則に関する例は、以下の個別のビットの説明で示します。この規則に従って 1 が読み出されるビットを除き､CP15 レジスタ 1 のビットはすべてリセット時に 0 にセットされます。

M（ビット **[0]**）　MMU または保護ユニットの許可 / 禁止を設定します。

0 = MMU または保護ユニットを禁止する。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

1 = MMU または保護ユニットを許可する。

MMU がないシステムでは、このビットから 0 が読み出され、書き込みは無視されます。

A（ビット **[1]**） ARM アーキテクチャ v6 では、このビットにより厳密な境界アラインメントが制御されます。

0 = 厳密な境界アラインメントを行わない。

1 = 厳密な境界アラインメントを行う。データアクセスが、アクセスされるデータアイテムの幅に合わせて境界アラインされていない場合、データアボート例外が生成されます。

v6 以前のアーキテクチャでは、データメモリアクセスのアライメントをチェックするオプションのあるメモリシステムについて、このビットによりアライメントフォルトのチェックを許可 / 禁止できます。

0 = アライメントフォルトチェックを禁止する。

1 = アライメントフォルトチェックを許可する。

他のメモリシステムについては、このビットへの書き込みは無視され、読み出しについてはメモリシステムがデータメモリアクセスのアライメントをチェックするかどうかによって 1 または 0 が読み出されます。

C（ビット **[2]**） L1 統合キャッシュが使用されている場合、統合キャッシュの許可 / 禁止ビットとなります。分離した L1 キャッシュが使用されている場合、データキャッシュの許可 / 禁止ビットとなります。どちらの場合も、値の意味は次のとおりです。

0 = L1 統合 / データキャッシュを禁止する。

1 = L1 統合 / データキャッシュを許可する。

L1 キャッシュが実装されていない場合、このビットから 0 が読み出され、書き込みは無視されます。L1 キャッシュを禁止できない場合、このビットから 1 が読み出され、書き込みは無視されます。

このビットの状態は、システムの他のレベルのキャッシュには影響しません。

W（ビット **[3]**） ライトバッファの許可 / 禁止を設定します。

0 = ライトバッファを禁止する。

1 = ライトバッファを許可する。

ライトバッファが実装されていない場合、このビットから 0 が読み出され、書き込みは無視されます。ライトバッファを禁止できない場合、このビットから 1 が読み出され、書き込みは無視されます。

SBO（ビット **[4:6]**）

これらのビットからは 1 が読み出され、書き込みは無視されます。

B（ビット **[7]**） このビットは、メモリシステムのエンディアン形式に合わせて ARM プロセッサを構成するために使用します。

リトルエンディアンとビッグエンディアン両方のワード不変型メモリシステムをサポートする ARM プロセッサでは、このビット使用して ARM プロセットを構成し、32 ビットワード内の 4 バイトアドレスの構成を変更します。

v6 では、この機構を使用して従来のビッグエンディアンオペレーティングシステムとアプリケーションをサポートしています。

0 = リトルエンディアンメモリシステム（LE）として構成されている。

1 = ビッグエンディアンワード不変メモリシステム（BE-32）として構成されている。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

CFGEND[1:0] の 2 つの構成ビットは、リセット時に P. A2-35 表 A2-11 に示すようにエンディアンモデルを定義します（以前のアーキテクチャでは、実装定義の構成オプションを使用して外部的にこのビットを、外部メモリサブシステムに依存してプリセットまたはリセットすることが可能でした）。

S（ビット **[8]**） システム保護ビットで、下位互換性のためにサポートされています。このビットの影響については、P. B4-8 *「アクセス許可」*を参照して下さい。この機能は、ARMv6 では推奨されません。

R（ビット **[9]**） ROM 保護ビットで、下位互換性のためにサポートされています。このビットの影響については、P. B4-8 *「アクセス許可」*を参照して下さい。この機能は、ARMv6 では推奨されません。

F（ビット **[10]**） このビットの意味は実装定義です。

Z（ビット **[11]**） 分岐予測をサポートする ARM プロセッサでは、このビットにより分岐予測が許可 / 禁止されます。

0 = プログラムフロー予測が禁止されている。

1 = プログラムフロー予測が許可されている。

プログラムフロー予測を禁止できない場合、このビットから 1 が読み出され、書き込みは無視されます。プログラムフロー予測には、命令ストリーム予測の可能な形態がすべて含まれます。例としては、静的予測、動的予測、復帰スタックがあります。

分岐予測をサポートしていない ARM プロセッサでは、このビットから 0 が読み出され、書き込みは無視されます。

I（ビット **[12]**） 分離した L1 キャッシュが使用されている場合、L1 命令キャッシュの許可 / 禁止ビットとなります。

0 = L1 命令キャッシュが禁止されている。

1 = L1 命令キャッシュが許可されている。

L1 統合キャッシュが使用されているか、L1 命令キャッシュが実装されていない場合、このビットから 0 が読み出され、書き込みは無視されます。L1 命令キャッシュを禁止できない場合、このビットから 1 が読み出され、書き込みは無視されます。

このビットの状態は、システムの他のレベルのキャッシュには影響しません。

V（ビット **[13]**） このビットは、例外ベクタのロケーション選択に使用されます。

0 = 通常例外ベクタが選択されている（アドレス範囲 `0x00000000` - `0x0000001C`）

1 = ハイ例外ベクタが選択されている（アドレス範囲 `0xFFFF0000` - `0xFFFF001C`）

実装により、リセット後のこのビットの状態を外部からの入力信号により決定することもできます。

RR（ビット **[14]**） キャッシュで、よりパフォーマンスが予測可能な別の置き換え手法を使用できる場合、このビットにより選択します。

0 = 通常の置き換え手法（ランダムな置き換えなど）

1 = 予測可能な置き換え手法（ラウンドロビン式の置き換えなど）

L4（ビット **[15]**） このビットをセットした場合、ARMv5T Thumb インターワーキング動作を示します。アドレスのビット [0] により CPSR T ビットが更新されなくなります。この禁止機能は、ARMv6 では推奨されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

この機能により影響を受ける命令は次のとおりです。

- *LDM (1)*：P. A4-36
- *LDR*：P. A4-44
- *POP*：P. A7-82

DT（ビット **[16]**） 常に 1。

SBZ（ビット **[17]**） このビットからは 0 が読み出され、書き込みは無視されます。

IT（ビット **[18]**） 常に 1。

SBZ（ビット **[19]**） このビットからは 0 が読み出され、書き込みは無視されます。

ST（ビット **[20]**） 常に 0/ 予測不能。

FI（ビット **[21]**） 高速割り込み機能に関する設定をします。実装定義のパフォーマンス改善機能を禁止し、実装での割り込みレイテンシを短縮するために使用できます。

0 = すべてのパフォーマンス改善機能を禁止する。

1 = 低割り込みレイテンシ構成を許可する。

U（ビット **[22]**） このビットは、アンアラインドデータアクセス操作を許可します。これには、リトルエンディアンとビッグエンディアンデータの混在のサポートも含まれます。

0 = アンアラインドロードは、ローテートされたアラインドデータアクセスとして処理されます（従来のコードの動作）。

1 = アンアラインドロードとストアが許容され、混在エンディアンデータのサポートが許可されます。

XP（ビット **[23]**） 拡張ページテーブルに関する設定をします。このビットは、ハードウェアページテーブルの変換機構を構成します。

0 = サブページ AP ビットを許可する。

1 = サブページ AP ビットを禁止する。この場合、ハードウェア変換テーブルにより追加の機能がサポートされます。

VE（ビット **[24]**） ベクタ割り込みに関する設定をします。実装定義のハードウェア機構を使用して、割り込みベクタを判定できるようにします。

0 = 割り込みベクタは固定。

- IRQ ベクタは V ビット == 0 の場合 `0x00000018`、V ビット == 1 の場合 `0xFFFF0018`。
- FIQ ベクタは V ビット == 0 の場合 `0x0000001C`、V ビット == 1 の場合 `0xFFFF001C`。

1 = 割り込みベクタは実装定義のハードウェア機構により定義されます。

EE ビット **[25]** 混在エンディアン例外のエントリ。EE ビットは、割り込みベクタへのエントリ時（リセットを含む）の CPSR E ビットの値を定義するために使用されます。この値は、ページテーブル参照に使用されるページテーブルデータのエンディアン形式を示すためにも使用されます。このビットは、システムリセット時に CFGEND[1:0] 端子によりプリセットされます。詳細については、P. A2-34 *「エンディアンの構成と制御」*を参照して下さい。

L2 ビット **[26]** L2 統合キャッシュを許可します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ビット [31:26] 予約。これらのビットは通常はリード・モディファイ・ライトの手法を使用して更新し、現在割り当てられていないビットが不必要に変更されないようにする必要があります。この規則に従わない場合、コードが将来のプロセッサで予期しない動作を引き起こす可能性があります。例外として、条件によってはこれらのビットに 0 を書き込んで、そのビットをリセット時の状態に復元できます。

B3.4.2 補助制御レジスタ

このレジスタの内容は実装定義です。このレジスタは、実装で制御ビットが存在しない場合でも、必ず特権読み出し / 書き込みでアクセス可能なことが保証されています。

B3.4.3 コプロセッサアクセスレジスタ

このレジスタは、CP15 と CP14 を除くすべてのコプロセッサへのアクセスを制御します。

このレジスタの一般的な使用方法は、オペレーティングシステムがコプロセッサリソースのアプリケーション間での共有を制御できるようにすることです。最初は、すべてのアプリケーションが共有リソースへのアクセスを拒否されます。アプリケーションがこのリソースの使用を試みると、未定義命令例外が発生します。この場合、未定義命令ハンドラがコプロセッサアクセスレジスタの対応するビットをセットし、そのリソースへのアクセスを許可できます。

アプリケーション間でリソースを共有するには、状態保存機構が必要です。このためには 2 つの方法があります。

- オペレーティングシステムは、コンテキスト切り替え時に、最後に実行されたプロセスがコプロセッサへのアクセス権を持っていた場合、コプロセッサの状態を保存します。
- また、オペレーティングシステムはコプロセッサのアクセス要求を処理した後に、最後にコプロセッサにアクセスしたプロセスの以前のコプロセッサ状態を復元します。

31 29	27	25	23	21	19	17	15	13	11	9	7	5	3	0
UNP/SBZP	cp13	cp12	cp11	cp10	cp9	cp8	cp7	cp6	cp5	cp4	cp3	cp2	cp1	cp0

コプロセッサアクセス権

ビットの各ペアは、各コプロセッサのアクセス権に対応しています。

00 アクセスは拒否されます。対応するコプロセッサにアクセスを試みた場合、未定義例外が発生します。

01 特権アクセスのみが許可されます。対応するコプロセッサにユーザモードでアクセスを試みた場合、未定義例外が発生します。

10 予約（動作は予測不能です）。

11 完全アクセス（関連するコプロセッサにより定義されます）。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

このレジスタを更新した後で、コプロセッサアクセスレジスタへの変更が可視になることを保証するため、PrefetchFlush 命令を実行する必要があります。このレジスタを変更してから、PrefetchFlush を実行するまでの間に、コプロセッサアクセス権限の変更により影響を受けるコプロセッサ命令を実行しないで下さい。

システムリセット後には、すべてのコプロセッサアクセス権限は*アクセス拒否*に設定されます。

実装されていないコプロセッサはすべて、関連するビットフィールドが 0 として読み出されます(RAZ)。このため、システムソフトウェアはコプロセッサアクセスレジスタに対してすべてのビットが 1 である値を書き込んでから結果を読み出すことで、自動構成手順の一部として、どのコプロセッサが存在しているかを知ることができます。

ある機能のセットに対して複数のコプロセッサが使用される場合（たとえば、VFP では CP10 と CP11 が使用されます）、それらのコプロセッサのコプロセッサアクセスレジスタのフィールドに対して異なった値が存在すると、動作は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B3.5 レジスタ 2 から 15 まで

レジスタ 0 と 1 を除くシステム制御コプロセッサレジスタは、次のように特定の領域に割り当てられます。

- CP15 のレジスタ 2 ～ 6、8、10、13 はメモリ保護システムに割り当てられます。これらのレジスタの詳細については、第 B4 章 *「仮想メモリシステムアーキテクチャ」*、第 B5 章 *「保護メモリシステムアーキテクチャ」*、第 B8 章 *「高速コンテキストスイッチ拡張機能」*を参照して下さい。
- CP15 のレジスタ 7 と 9 は、キャッシュとライトバッファの制御に割り当てられます。これらのレジスタの詳細については、第 B6 章 *「キャッシュとライトバッファ」* を参照して下さい。
- CP15 のレジスタ 11 は、レベル 1 メモリ DMA サポートに割り当てられます。詳細については、第 B7 章 *「密結合メモリ」* を参照して下さい。
- CP15 のレジスタ 15 は、実装定義の目的に予約されています。これらのレジスタで利用可能な機能の詳細については、実装のテクニカルリファレンスマニュアル、または他の実装固有のドキュメントを参照して下さい。
- CP15 のレジスタ 12 と 14 は、将来の拡張用に予約されています。これらのレジスタのいずれかにアクセスした結果は予測不能で、ARMv6 以降は未定義です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 B4 章
仮想メモリシステムアーキテクチャ

本章では、*メモリ管理ユニット*（MMU）をベースとした*仮想メモリシステムアーキテクチャ*（VMSA）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。

- *VMSA の概要* : P. B4-2
- *メモリアクセスのシーケンス* : P. B4-4
- *メモリアクセスの制御* : P. B4-8
- *メモリ領域属性* : P. B4-11
- *アボート* : P. B4-14
- *フォルトアドレスレジスタとフォルトステータスレジスタ* : P. B4-19
- *ハードウェアページテーブル変換* : P. B4-23
- *ファインページテーブルとタイニーページのサポート* : P. B4-35
- *CP15 のレジスタ* : P. B4-39

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.1 VMSA の概要

高度なオペレーティングシステムでは一般に、仮想メモリシステムを使用して、各プロセスに独立の保護されたアドレス空間を提供します。プロセスには、*メモリ管理ユニット*（MMU）の制御下でメモリや、メモリマップされた他のシステムリソースが動的に割り当てられます。MMU により、MMU 内の*変換ルックアサイドバッファ*（TLB）と呼ばれる 1 つ以上の構造に保持されている仮想アドレスから物理アドレスへのマッピングと、関連するメモリプロパティを使用して、非常に小さいサイズを基本としたメモリ管理をすることができます。TLB の内容は、メモリ内に保持されている一連の変換テーブルへのハードウェアによる変換ルックアップにより管理されます。

一連の変換テーブルルックアップを行う手順を*変換テーブルウォーク*と呼びます。この処理はハードウェアによって自動的に行われ、最低 1 メインメモリアクセス、もしくは 2 メインメモリアクセス分の長い実行時間が必要となります。TLB を使用すると、変換テーブルウォークの結果をキャッシュしてメモリアクセスの平均コストを減らすことができます。実装では、統合 TLB（フォンノイマンアーキテクチャ）と、独立した命令 TLB とデータ TLB（ハーバードアーキテクチャ）のどちらでも使用できます。

ARMv6 の VMSA は大幅に拡張されています。新しいアーキテクチャは VMSAv6 と呼ばれます。コンテキスト切り替え時に TLB の無効化が必要となることを避けるため、仮想アドレスから物理アドレスへのアドレスマッピングのそれぞれを、特定のアプリケーション空間に関連付けられたもの、またはすべてのアプリケーション空間についてグローバルなものとしてマークできます。グローバルマッピング、および現在のアプリケーション空間へのマッピングがその時点で有効になります。*アプリケーション空間識別子*（ASID）を変更することで、有効な仮想アドレスから物理アドレスへのマッピングを変更することができます。VMSAv6 には、各種のメモリタイプの定義（詳細については P. B2-8 *「ARMv6 のメモリ属性の概要」*を参照して下さい）と、他の属性（詳細については P. B4-8 *「メモリアクセスの制御」*を参照して下さい）が追加されています。下位互換性のため、システム制御コプロセッサの CP15 のレジスタ 1 には P. B4-40 *「レジスタ 1: 制御レジスタ」*に定義されている XP 制御ビットがあります。

各 TLB エントリに関連付けられたメモリプロパティには次のものが含まれます。

メモリのアクセス許可の制御

プログラムがメモリ領域に対して、アクセス許可がないか、読み出し専用のアクセス許可を持つか、読み出し / 書き込みのアクセス許可を持つかを制御します。アクセスが許可されていない場合、メモリアボート信号がプロセッサに送られます。

許可されるアクセスのレベルは、プログラムがユーザモードと特権モードのどちらで実行されているかと、ドメインの使用にも影響を受けます。

メモリ領域属性

メモリ領域のプロパティを示します。例としては、デバイス（VMSAv6）、キャッシュ不可、ライトスルー、ライトバックがあります。

仮想アドレスから物理アドレスへのマッピング

ARM® プロセッサによって生成されるアドレスは、*仮想アドレス*と呼ばれます。このアドレスは、MMU によって別の*物理アドレス*にマップされます。この物理アドレスは、アクセスされるメインメモリのロケーションを示します。

この操作を使用して、各種の方法で物理メモリの割り当てを管理できます。たとえば、複数のプロセスに対して場合により競合するアドレスマップを割り当てたり、物理メモリの連続した領域を使ってアプリケーションに空き空間のあるアドレスマップを割り当てたりすることができます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

—— 注 ——

高速コンテキストスイッチ拡張機能（FCSE。詳細については第 B8 章を参照して下さい）のため、本章で示すすべての仮想アドレスへの参照は、特に記載のある場合を除いて FCSE により生成される*修飾仮想アドレス*を示します。FCSE 機構が禁止されている場合（PID = 0）、仮想アドレスと修飾仮想アドレスは等しくなります。

ARMv6 では、FCSE は下位互換性の目的でのみ搭載されています。新規システムでの使用は推奨されません。

システム制御コプロセッサレジスタにより、変換テーブルのロケーションなどシステムの高度な制御が可能です。これらは、メモリアボートのステータス情報を ARM プロセッサに提供するためにも使用されます。

VMSA により、TLB 内で特定の TLB エントリをロックダウンすることが可能です。これにより、関連するメモリ領域へのアクセスに､変換テーブルウォークによる参照が必要ないことが保証されます。これによって、リアルタイムルーチンでのコードおよびデータへのワーストケースのアクセス時間が最小化され、確定的になります。

メモリ内の変換テーブルが変更された、または異なる変換テーブルが選択された（CP15 のレジスタ 2 への書き込みにより）場合、TLB にある、以前にキャッシュされた変換テーブルウォークの結果は無効になります。このため、VMSA では TLB のフラッシュ操作が用意されています。

B4.1.1 VMSAv6 で導入された主要な変更

VMSAv6 で導入された変更の概要は次のとおりです。

- エントリは、アプリケーション空間識別子と関連付けるか、グローバルマッピングとしてマークすることができます。これによって、ほとんどのコンテキスト切り替えにおいて TLB のフラッシュが必要なくなります。
- アクセス許可が拡張され、特権モードの読み取り専用と、特権モード / ユーザモードの読み取り専用モードが同時にサポート可能になりました。システム (S) および ROM (R) ビットを使用してアクセス許可の決定を制御する方法は、下位互換性の目的でのみサポートされています。
- メモリ領域の属性により、ページが複数のプロセッサで共有されることをマークできます。
- タイニーページと、ファインページテーブルの 2 次レベルフォーマットは使用されなくなりました。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.2 メモリアクセスのシーケンス

ARM CPU がメモリアクセスを行うとき、MMU は要求された修飾仮想アドレスのマッピングを TLB から探すルックアップを実行します。VMSAv6 から、現在の ASID を含んでルックアップを実行します。実装では、ハーバード TLB と統合 TLB のどちらでも使用できます。実装で独立した命令 TLB とデータ TLB を使用する場合、それぞれが次の用途に使用されます。

- 命令 TLB は命令フェッチに使用されます。
- データ TLB は他のすべてのアクセスに使用されます。

対応する TLB に、グローバルマッピング、または現在選択されている ASID に対応するマッピング（VMSAv6）での修飾仮想アドレスが見つからない場合、ハードウェアにより変換テーブルウォークが自動的に実行されます。

——— 注 ———

VMSAv6 以前は、すべての修飾仮想アドレス変換はグローバルマップされたものと見なすことができました。ARMv6 からは非グローバルアドレスにアクセスする場合は、32 ビットの修飾仮想アドレスに ASID ビットを加えた修正仮想アドレスと見なす必要があります。

第 B8 章*「高速コンテキストスイッチ拡張機能」*で説明されている FCSE 機構は、ARMv6 では推奨されません。また、FCSE と ASID を同時に使用した場合の動作は予測不能です。FCSE レジスタをクリアするか、すべてのメモリをグローバルとして宣言する必要があります。

———

一致する TLB エントリが見つかった場合、そこに含まれている情報は次のように使用されます。

1. アクセス許可ビットとドメインを使用して、アクセスが許可されているかどうかを判定します。アクセスが許可されていない場合、MMU がメモリアボート信号を発生します。それ以外の場合、アクセスが許可されます。
2. メモリ領域の属性により、次の制御が行われます。
   - キャッシュと書き込みバッファ
   - アクセスがキャッシュの対象となるかどうか
   - ターゲットのメモリタイプ
   - ターゲットのメモリが共有か非共有か
3. 外部または密結合メモリへのすべてのアクセスには物理アドレスが使用され、物理タグキャッシュ実装におけるキャッシュエントリのタグの一致を見るために使用することもできます。

P. B4-5 図 B4-1 は、キャッシュが使用されているシステムの構造を示したものです。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

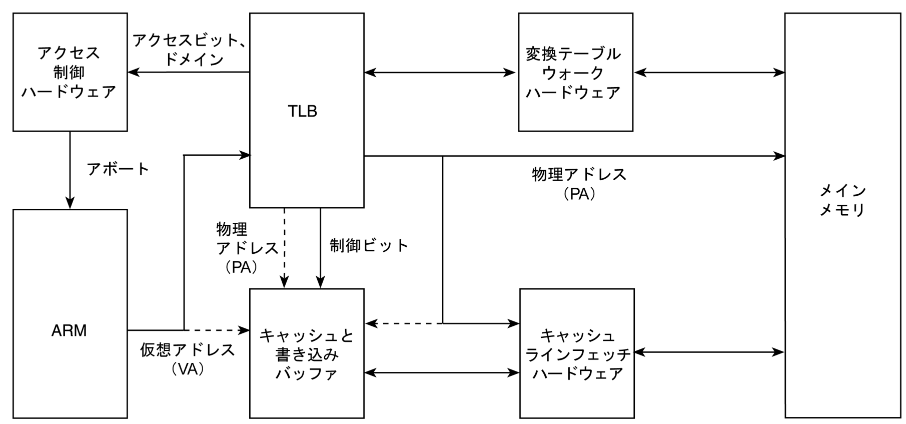


図 B4-1 キャッシュを使用している MMU メモリシステムの概要

B4.2.1 TLB 一致プロセス

各 TLB エントリには、修飾仮想アドレス、ページサイズ、物理アドレス、一連のメモリプロパティが含まれています。これらの情報は、特定のアプリケーション空間に関連付けられているか、すべてのアプリケーション空間についてグローバルです。ASID が使用されている場合、CP15 のレジスタ 13 によって、現在選択されているアプリケーション空間が判定されます。

TLB エントリは、修飾仮想アドレスのビット 31 - *N* が一致しており、かつグローバルとしてマークされているか、ASID が現在の ASID に一致している場合、一致すると見なされます。ここで、*N* は TLB エントリのページサイズの $\log_2$ です。

2 つ以上のエントリが同時に一致する場合（グローバルエントリと ASID 固有エントリを含みます）、TLB の動作は予測不能です。オペレーティングシステムは、同時に複数の TLB エントリが一致しないことを保証する必要があります。このために、通常はグローバルページのマッピングが変更されたときに TLB をフラッシュします。

TLB には、次のブロックサイズに基づいたエントリが格納できます。

スーパーセクション 16MB ブロックのメモリで構成されます。

セクション 1MB ブロックのメモリで構成されます。

ラージページ 64KB ブロックのメモリで構成されます。

スモールページ 4KB ブロックのメモリで構成されます。

注

タイニー（1 KB）ページの使用は、VMSAv6 ではサポートされていません。

スーパーセクション、セクション、ラージページは大きなメモリ領域によりマッピングされますが、1 つの TLB のエントリにより使用できます。

TLB 内にアドレスのマッピングが見つからない場合、ハードウェアにより自動的に変換テーブルが読み出され、TLB にマッピングが保存されます。詳細については、P. B4-23 *「ハードウェアページテーブル変換」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.2.2 仮想から物理への変換マッピング命令

VMSA は、仮想アドレスでインデクス付けされ、物理アドレスでタグ付けされるキャッシュと組み合わせて使用できます。仮想アドレスから物理アドレスへのマッピングページテーブルに関する制約については、P. B6-11 *「ページテーブルのマッピングの制限」*を参照して下さい。

B4.2.3 MMU の許可と禁止

MMU は、システム制御コプロセッサのレジスタ 1 の M ビット（ビット [0]）に書き込むことで許可または禁止できます。リセット時にはこのビットは 0 でクリアされ、MMU は禁止されます。

MMU が禁止されている場合、メモリアクセスは次のように処理されます。

- すべてのデータアクセスはキャッシュ不可かつストロングオーダとして扱われます。予期しないデータキャッシュヒットの動作は実装定義となります。
- ハーバードキャッシュ構成が使用されている場合、すべての命令アクセスは、CP15 のレジスタ 1 の I ビット（ビット [12]）がセット (1) されている場合はキャッシュ可能、共有不可、通常メモリとして扱われ、I ビットがクリア (0) されている場合はキャッシュ不可、共有不可、通常メモリとして扱われます。キャッシュに関連する他のメモリ属性（ライトスルーキャッシュ可能、ライトバックキャッシュ可能など）は実装定義です。

  統合キャッシュが使用されている場合、すべての命令アクセスは共有不可、通常メモリ、キャッシュ不可として扱われます。
- すべての明示的なアクセスはストロングオーダです。CP15 のレジスタ 1 の W ビット（ビット [3]、ライトバッファ許可）の値は無視されます。
- メモリアクセス許可のチェックは行われず、MMU によるアボートは発生しません。
- すべてのアクセスの物理アドレスは、修飾仮想アドレスに等しくなります。これをフラットアドレスマッピングと呼びます。
- MMU が無効の場合、FCSE PID（*P. B4-52「レジスタ 13: プロセス ID」*参照）は*常に 0*（SBZ）である必要があります。これは、FCSE PID のリセット時の値です。MMU を禁止するには、FCSE PID をクリアする必要があります。MMU が禁止されており、FCSE がクリアされていない場合、動作は予測不能です。
- CP15 のキャッシュ操作は、MMU が許可されているか、およびメモリ属性の値にかかわらず、対象となるキャッシュに対して動作します。ただし、MMU が禁止されている場合、アーキテクチャによるフラットマッピングが使用されます。

  CP15 TLB の無効化操作は、MMU が許可されているかどうかにかかわらず、対象となる TLB に対して動作します。
- 命令とデータのプリフェッチ操作は通常に動作します。

  ——— 注 ———

  MMU が禁止されている場合はデータキャッシュは許可できないため、データのプリフェッチ操作は無効です。命令キャッシュが禁止されている場合、命令プリフェッチ操作は無効です。命令キャッシュが許可されている場合、命令キャッシュにプリフェッチが実行されます。メモリアクセス許可は実行されず、アドレスはフラットマップです。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- TCM が許可されている場合、TCM へのアクセスは通常に動作します。

MMU を許可する前に、関連するすべての CP15 レジスタをプログラムする必要があります。これには、メモリへの適切な変換テーブルのセットアップが含まれます。MMU を許可する前に、命令キャッシュを禁止し、無効化する必要があります。その後で、MMU を許可すると同時に命令キャッシュを再度許可できます。

—— 注 ——

MMU を許可または禁止すると、実質的に仮想アドレスから物理アドレスへのマッピングが変更されます（変換テーブルがフラットアドレスマッピングを実装するようにセットアップされている場合を除きます）。たとえば、そのときに有効になっていた、仮想的にタグ付けされたキャッシュはすべてフラッシュする必要があります（P. B2-20*「メモリのコヒーレンシとアクセスの考慮点」*を参照して下さい）。

さらに、MMU を許可または禁止するコードの物理アドレスが修飾仮想アドレスと異なる場合、命令のプリフェッチによって問題が発生することがあります（P. B2-19*「PrefetchFlush CP15 レジスタ 7」*参照）。このため、MMU を許可または禁止するコードは、仮想アドレスと物理アドレスを同一にすることを強くお勧めします。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.3 メモリアクセスの制御

メモリ領域へのアクセスは、TLB エントリのアクセス許可およびドメインビットにより制御されます。APX ビットと XN（実行不可）ビットは、VMSAv6 で追加されたものです。これらは、P. B4-23「*ハードウェアページテーブル変換*」で説明されているように、ページテーブルエントリフォーマットの一部です。

B4.3.1 アクセス許可

アクセス許可ビットは、対応するメモリ領域へのアクセスを制御します。必要なアクセス許可なしにメモリ領域に対してアクセスが行われた場合、アクセス許可フォルトが発生します。アクセス許可は、ページテーブルの AP および APX ビットと、CP15 のレジスタ 1 の S および R ビットの組み合わせによって判定されます。ページテーブルのフォーマットが APX ビットをサポートしていない場合、値として 0 が使用されます。

——— 注 ———

S ビットと R ビットの使用は、VMSAv6 では推奨されません。S ビットと R ビットへの変更は、既に TLB にあるエントリのアクセス許可には影響しません。S ビットと R ビットの値を更新した後で、その効果を反映するには、TLB をフラッシュする必要があります。

————————

必要なアクセス許可なしにメモリ領域に対してアクセスが行われた場合、アクセス許可フォルトが発生します。詳細については、P. B4-14「*アボート*」を参照して下さい。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B4-1 は、アクセス許可のエンコードの一覧です。

表 B4-1 MMU のアクセス許可

S	R	APX[a]	AP[1:0]	特権アクセス許可	ユーザアクセス許可	説明
0	0	0	0b00	アクセス不可	アクセス不可	すべてのアクセスについて、アクセス許可フォルトが発生します。
x	x	0	0b01	読み出し / 書き込み	アクセス不可	特権アクセスのみ許可されます。
x	x	0	0b10	読み出し / 書き込み	読み出し専用	ユーザーモードで書き込みを行うと、アクセス許可フォルトが発生します。
x	x	0	0b11	読み出し / 書き込み	読み出し / 書き込み	完全アクセス。
0	0	1	0b00	-	-	予約
0	0	1	0b01	読み出し専用	アクセス不可	特権モードでの読み出しのみ許可されます。
0	0	1	0b10	読み出し専用	読み出し専用	特権モードとユーザモードでの読み出しのみ許可されます。
0	0	1	0b11	-	-	予約
S ビットと R ビットの使用は、VMSAv6 では推奨されません。以下のエントリは、従来のシステムのみに適用されます。						
0	1	0	0b00	読み出し専用	読み出し専用	特権モードとユーザモードでの読み出しのみ許可されます。
1	0	0	0b00	読み出し専用	アクセス不可	特権モードでの読み出しのみ許可されます。
1	1	0	0b00	-	-	予約
0	1	1	0bxx	-	-	予約
1	0	1	0bxx	-	-	予約
1	1	1	0bxx	-	-	予約

a. VMSAv6 以降のみ

各メモリ領域には、実行可能なコードを含まないことを示すタグを付けることができます。*実行不可*（XN）ビットが 1 にセットされている場合、その領域の命令を実行しようとするとアクセス許可フォルトが発生します。XN ビットが 0 にクリアされている場合、そのメモリ領域からコードを実行できます。

— 注 —

XN ビットは、追加のアクセス許可チェックとして動作します。また、アドレスには有効な読み出しアクセスも必要です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.3.2 ドメイン

ドメインは、メモリ領域の集合です。ARM アーキテクチャでは 16 個のドメインがサポートされています。各ページテーブルエントリと TLB エントリには、そのエントリの属するドメインを示すフィールドが含まれています。各ドメインへのアクセスは、ドメインアクセス制御レジスタの 2 ビットのフィールドにより制御されます。各フィールドにより、迅速にドメイン全体に対してアクセスを許可 / 禁止できるため、特定のメモリ領域をまとめて仮想メモリに効率的にスワップイン / スワップアウトできます。2 種類のドメインアクセスがサポートされています。

クライアント ドメインのユーザで（プログラムの実行とデータへのアクセスを行う）、そのドメインについての TLB エントリのアクセス許可により保護されます。

マネージャ ドメインの動作を制御し（ドメイン内のセクションとページ、およびドメインアクセス）、そのドメインについての TLB エントリのアクセス許可によって保護されません。

1 つのプログラムがあるドメインについてクライアントであり、別のドメインについてマネージャであり、他のドメインについてはアクセス許可がない場合もあります。これによって、各種のメモリリソースにアクセスするプログラムについて、非常に柔軟なメモリ保護が可能になります。表 B4-2 は、ドメインアクセス制御レジスタのビットのエンコードを示したものです。

表 B4-2 ドメインアクセスの値

値	アクセスタイプ	説明
0b00	アクセス不可	すべてのアクセスがドメインフォルトを生成します。
0b01	クライアント	TLB エントリのアクセス許可ビットに対してアクセスがチェックされます。
0b10	予約	この値を使用した場合、結果は予測不能です。
0b11	マネージャ	TLB エントリのアクセス許可ビットに対してアクセスチェックは行われず、アクセス許可フォルトは発生しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.4 メモリ領域属性

各 TLB エントリには、関連する一連のメモリ領域属性があります。これらの属性により、キャッシュへのアクセス、ライトバッファの使用方法、メモリ領域が共有可能かどうか、そのためにコヒーレンシを維持する必要があるかどうかが制御されます。

VMSAv6 以前は、C（キャッシュ可能）ビットと B（バッファ可能）ビットのみが存在していました。これらの正確な使用モデル(たとえば､ビットの設定がライトスルーキャッシュとライトバックキャッシュのポリシーにどのように影響するか) 、その他の制御は実装定義でした。VMSAv6 では、より多くの正式なメモリモデルが導入され（P. B2-8「*ARMv6 のメモリ属性の概要*」参照）、追加のビットフィールド（TEX）と、このセクションで説明する定義によりサポートされるようになりました。

B4.4.1 C、B、TEX のエンコード

ページテーブルのフォーマットでは、メモリ領域のタイプのエンコードに TEX[2:0]、C、B ビットの 5 ビットを使用します。P. B4-12 表 B4-3 は、*タイプ拡張子*フィールド（TEX）のマッピングと、キャッシュ可能およびバッファ可能ビット（C ビットと B ビット）からメモリ領域タイプへのマッピングを示したものです。ページテーブルのフォーマットに TEX フィールドがない場合、値として 0b000 が使用されます。

また、特定のページテーブルには共有ビット (S) が含まれています。このビットは通常メモリにのみ適用され、デバイスやストロングオーダメモリには適用されず、メモリ領域が共有か (1) 非共有か (0) を示します。このビットが存在しない場合、S ビットは 0（非共有）であると見なされます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B4-3 は、C、B、TEX のエンコードを示したものです。

表 B4-3 C、B、TEX のエンコード

TEX	C	B	説明	メモリタイプ	ページの共有
0b000	0	0	ストロングオーダ	ストロングオーダ	共有可能
0b000	0	1	共有デバイス	デバイス	共有可能
0b000	1	0	外部と内部のライトスルー、書き込み割り当てなし	通常	S
0b000	1	1	外部と内部のライトバック、書き込み割り当てなし	通常	S
0b001	0	0	外部と内部でキャッシュ不可	通常	S
0b001	0	1	予約	-	-
0b001	1	0	実装定義	実装定義	実装定義
0b001	1	1	外部と内部のライトバック、書き込み割り当てあり	通常	S
0b010	0	0	非共有デバイス	デバイス	非共有
0b010	0	1	予約	-	-
0b010	1	X	予約	-	-
0b011	X	X	予約	-	-
0b1BB	A	A	キャッシュされたメモリ BB = 外部ポリシー、AA = 内部ポリシー	通常	S

「S」はページテーブルが存在し、その S ビットがセットされている場合は共有可能を、それ以外の場合は共有不可を示します。

共有可能属性と、通常、ストロングオーダ、デバイスの各メモリタイプの説明については、P. B2-8 *「ARMv6 のメモリ属性の概要」*を参照して下さい。

内部と外部という用語は、システムに搭載されているキャッシュのレベルを指します。内部とは、レベル 1 キャッシュを含む最も内側のキャッシュを指します。外部とは、最も外側のキャッシュを指します。内部キャッシュと外部キャッシュの境界は、キャッシュを持つシステムの実装により定義されます。内部キャッシュには、常にレベル 1 キャッシュが含まれます。たとえば、3 レベルのキャッシュを持つシステムの場合、レベル 1 とレベル 2 に内部属性が、レベル 3 に外部属性が適用されることがあります。キャッシュが 2 レベルのシステムの場合、レベル 1 が内部、レベル 2 が外部と見なされます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B4-4 は、内部と外部のキャッシュポリシーのエンコードです。

表 B4-4 内部と外部のキャッシュポリシー

エンコード		説明
0	0	キャッシュ不可
0	1	ライトバック、書き込み割り当てあり
1	0	ライトスルー、書き込み割り当てなし
1	1	ライトバック、書き込み割り当てなし

どの書き込み割り当てポリシーをサポートするかは、実装で決定できます。*書き込み時割り当て*と*非書き込み時割り当て*キャッシュポリシーは、メモリ領域について優先される割り当てポリシーを示しますが、メモリシステムがそのポリシーを実装していることを前提にはできません。

すべての内部と外部のキャッシュポリシーが必須ではありません。表 B4-5 に実装オプションを示します。

表 B4-5 キャッシュポリシーの実装オプション

キャッシュポリシー	実装オプション
内部キャッシュ不可	必須。
内部ライトスルー	必須。
内部ライトバック	オプション。サポートされていない場合、メモリシステムに内部ライトスルーを実装する必要があります。
外部キャッシュ不可	必須。
外部ライトスルー	オプション。サポートされていない場合、メモリシステムは外部キャッシュ不可として実装する必要があります。
外部ライトバック	オプション。サポートされていない場合、メモリシステムに外部ライトスルーを実装する必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.5 アボート

メモリアクセスにより、ARM プロセッサに例外が発生する機構は次のとおりです。

MMU フォルト MMU が制限を検出し、プロセッサに信号を送ります。

デバッグアボート モニタデバッグモードが有効になり、ブレークポイントまたはウォッチポイントが検出されます。

外部アボート 外部メモリシステムから、不正な、またはフォルトの発生したメモリアクセスが報告されます。

これらを*アボート*と総称します。アボートを発生したアクセスはアボートされたと呼び、関連するコンテキスト情報を記録するため*フォルトアドレス*および*フォルトステータスレジスタ*が使用されます。FAR レジスタと FSR レジスタについては、P. B4-19 *「フォルトアドレスレジスタとフォルトステータスレジスタ」*を参照して下さい。

B4.5.1 MMU フォルト

MMU は 4 種類のフォルトを生成します。

- アライメントフォルト
- 変換フォルト
- ドメインフォルト
- アクセス許可フォルト

MMU により検出されたアボートは、そのアボートが検出されたアドレスへの外部アクセスを発生しません。

アボートされたメモリ要求が命令フェッチの場合、そのアボートされたアクセスに対応する命令の実行がプロセッサにより試みられたときにプリフェッチアボート例外が発生します。アボートされたアクセスがデータアクセスかキャッシュ保守操作の場合、データアボート例外が発生します。プリフェッチアボートとデータアボートの詳細については、P. A2-16 *「例外」*を参照して下さい。

フォルトチェック手順

MMU がアクセスフォルトをチェックするために使用する手順は、セクションの場合とページの場合で多少異なります。P. B4-15 図 B4-2 は、両方のタイプのアクセスについての手順です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

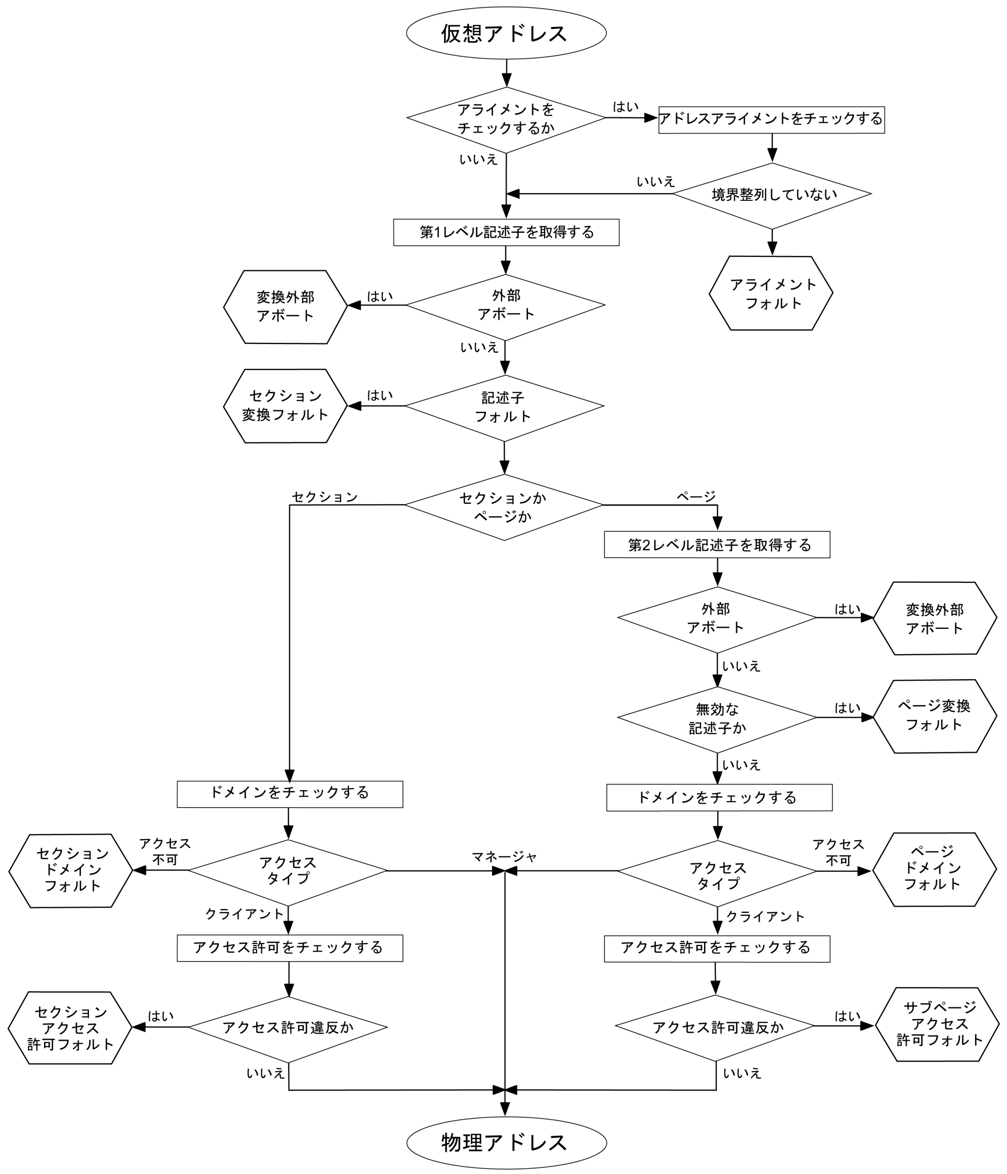


図 B4-2 フォルトチェックの手順

アライメントフォルト

アライメントフォルトが発生する条件の詳細については、P. A2-40 表 A2-14 を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

変換フォルト

変換フォルトには 2 種類あります。

セクション 第 1 レベル記述子が無効にマークされている場合に発生します。記述子のビット [1:0] が両方とも 0 の場合と、VMSAv6 フォーマットで値が予約済みの 0b11 の場合に、このフォルトが発生します。

ページ 第 2 レベル記述子が無効にマークされている場合に発生します。記述子のビット [1:0] が両方とも 0 の場合に発生します。

結果として変換フォルトが発生したフェッチされたページテーブルエントリはキャッシュされない（TLB が更新されない）ことが保障されます。TLB 保守操作により、変換フォルトを発生させたエントリをフラッシュする必要はありません。

ドメインフォルト

ドメインフォルトには 2 種類あります。

セクションドメインフォルト
ドメインは、第 1 レベル記述子が返されるときにチェックされます。

ページドメインフォルト
ドメインは、有効な第 2 レベル記述子が返されるときに（第 1 レベル記述子のドメインフィールドに基づいて）チェックされます。

ドメインフォルトの結果、関連するページテーブルが更新される場合、対応する TLB エントリの正確性を保証するためにそのエントリをフラッシュする必要があります。詳細については、P. B2-22「*TLB 保守操作とメモリオーダダモデル*」にあるページテーブルエントリ更新の例を参照して下さい。

ドメインアクセス制御レジスタの変更は、PrefetchFlush 操作の実行（または、例外か例外からの復帰の結果）により有効になります。詳細については、P. B2-24「*CP15 のレジスタの変更とメモリオーダモデル*」を参照して下さい。

アクセス許可フォルト

2 ビットのドメインフィールドが返す値が*クライアント*（01）の場合、TLB エントリのアクセス許可フィールドに対してアクセス許可がチェックされます。

アクセス許可フォルトの結果、関連するページテーブルが更新される場合、対応する TLB エントリの正確性を保証するためにそのエントリをフラッシュする必要があります。詳細については、P. B2-22「*TLB 保守操作とメモリオーダダモデル*」にあるページテーブルエントリ更新の例を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.5.2 デバッグイベント

モニタデバッグモードが有効な場合、命令アクセスのブレークポイント、またはデータアクセスのウォッチポイントによりアボートが発生する可能性があります。

モニタデバッグモードのためアボートが発生した場合、デバッグアボートを示すために、対応する FSR（命令またはデータ）が更新されます。これは、プリフェッチアボート（ブレークポイント）デバッグイベントで保存される唯一の情報です。これは正確なアボートです。フォルトの発生した命令のアドレスを判定するには、R14_abt が使用されます。

ウォッチポイントは、次の命令が複数ロードおよびストアと同時に実行されている可能性があることから、正確に取得されません。デバッグイベントを発生した命令を判定するには、デバッガで *ウォッチポイントフォルトアドレスレジスタ*（WFAR）を読み出す必要があります。

B4.5.3 外部アボート

外部メモリエラーは、MMU で検出される以外のメモリシステムで発生したエラーです。外部メモリエラーは発生する頻度が低いと想定されており、実行中のプロセスに対して多くの場合致命的エラーとなります。外部メモリエラーを引き起こすイベントの例としては、レベル 2 メモリ構造での修正不能なパリティまたは ECC エラーがあります。

外部アボートをサポートするかどうか、どの外部アボートをサポートするかは実装定義です。

正確な外部アボートの存在は、DFSR または IFSR により通知されます。不正確なアボートモデルの詳細については、P. A2-23 *「不正確なデータアボート」* を参照して下さい。

命令フェッチ時の外部アボート

命令プリフェッチ時に外部で生成されたエラーは本質的に正確で、エラーを発生したロケーションからフェッチされた命令を CPU が実行しようとしたときにのみ、CPU によって認識されます。

命令フェッチで外部アボートが発生した場合、フォルトアドレスレジスタは更新されません。

データの読み出し / 書き込み時の外部アボート

データの読み出しまたは書き込み時に外部で生成されたエラーは、不正確なことがあります。つまり、このようなアボートが発生した場合、アボートハンドラが開始された時点で、R14_abt がその例外を発生した命令に関連するアドレスを保持しているとは限りません。このため、外部アボートは回復不能なことがあります。

アボート状態がリエントラントでないときに、不正確な外部アボートによってアボート状態に入った場合、R14 と SPSR の値が破壊されているため、プロセッサは回復不能な状態になります。この理由から、不正確な外部アボートの存在は、アボート状態がリエントラントな場合のみ、プロセッサから認識されることが必要です。これは、CSPR にある、不正確なアボートを示すマスクにより管理されます。これは A ビットと呼ばれます。

不正確な外部アボートによりアボート状態に入った場合、DFSR には不正確な外部アボートの存在が示されています。データアクセスで不正確な外部アボートが発生した場合、FAR は更新されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ハードウェアページテーブルウォークでの外部アボート

ハードウェアページテーブルのアクセスで発生した外部アボートは、ページテーブルデータと共に返される必要があります。このアボートは正確です。FAR は、データアクセスのハードウェアページテーブルウォークで発生した外部アボートのときは更新されますが、命令アクセスで発生した外部アボートでは更新されません。対応する FSR（命令またはデータ）により、このアボートが発生したかどうかが示されます。

パリティエラーの報告

パリティエラーは、正確なアボート（レベル 1 キャッシュヒットの読み出しなど）か、不正確なアボート（キャッシュラインフィルなど）として発生します。パリティエラーを報告するためにフォルトステータスコードが定義されています。パリティエラーに対してどのようなサポートを行うか、そのエラーを報告するために割り当てられたフォルトステータスコードや他のエンコードを使用するかは実装定義です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.6 フォルトアドレスレジスタとフォルトステータスレジスタ

VMSAv6 以前のアーキテクチャは、単一の*フォルトアドレスレジスタ*（FAR）と*フォルトステータスレジスタ*（FSR）をサポートしていました。

VMSAv6 では 4 つのレジスタが必須です。

- *命令フォルトステータスレジスタ*（IFSF）: プリフェッチアボート時に更新されます。
- *データフォルトステータスレジスタ*（DFSR）: データアボート時に更新されます。
- *フォルトアドレスレジスタ*(FAR): 正確な例外の発生時に、フォルトアドレスが書き込まれます。
- *ウォッチポイントフォルトアドレスレジスタ*（WFAR）: ウォッチポイントアクセス時に、データアボートを発生した命令のアドレスが書き込まれます。

——— 注 ———

IFSR と DRSR は、命令キャッシュ保守操作のため、データアボート時に更新されます。

正確な例外と不正確な例外の説明については、P. A2-16*「例外」*を参照して下さい。

VMSAv6 では、IFSR と DFSR の両方に 5 番目のフォルトステータスビット（ビット [10]）が追加されました。VMSAv6 以前のアーキテクチャでは、このビットのエンコードは実装定義でした。また、書き込みフラグ（DFSR のビット [11]）も導入されています。

データアクセスにより発生した正確なアボート（正確なデータアボート）は、CPU により直ちに対応されます。DFSR には 5 ビットのフォルトステータス（FS[10,3:0]）と、アクセスのドメイン番号が書き込まれます。また、データアボートを発生した修飾仮想アドレスも FAR に書き込まれます。データアクセスが複数のタイプのデータアボートを同時に発生した場合、P. B4-20 表 B4-6 に示す順序で優先度が判定されます。最も優先度の高いアボートが報告されます。

命令フェッチから発生したアボートは、その命令が命令パイプラインに入るときにフラグ付けされます。その命令が実行された場合のみ、プリフェッチアボート例外が発生します。命令フェッチの結果として発生したアボートは、その命令が使用されない場合は処理されません（分岐により別の命令が実行された場合など）。

プリフェッチアボート例外に関連するフォルトアドレスは、プリフェッチアボート例外ベクタに入ったときに R14_abt に保存されている値によって判定されます。*命令フォルトアドレスレジスタ*（IFAR）が実装されている場合、アボートを発生した修飾仮想アドレスもこのレジスタに保存されます。

命令フェッチにより発生したアボートで DFSR と FAR が更新されるか、される場合にフォルトに関するどのような情報を含むかは実装定義です。ただし、データアクセスで発生したアボートにより DFSR と FAR が更新されてから、対応するデータアボート例外ベクタに入るまでは、命令フェッチで発生したアボートによりこれらのレジスタが更新されることはありません。つまり、命令フェッチで発生したアボートにプロセッサが対応しなくても FAR と DFSR の値は破壊されないことを、データアボートハンドラは前提とすることができます。VMSAv6 以降では、IFSR のみがプリフェッチアボートによって更新されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B4-6 フォルトステータスレジスタのエンコード

アーキテクチャ	優先度	ソース		FS [10,3:0]	ドメイン[a]	FAR
すべて	最高	アライメント		0b00001	無効	有効
VMSAv6		PMSA - TLB ミス（MPU）		0b00000	無効	有効
		アライメント（非推奨）		0b00011		
VMSAv6		命令キャッシュ保守操作フォルト		0b00100	無効	有効
すべて		変換時の外部アボート	第 1 レベル 第 2 レベル	0b01100 0b01110	無効 有効	有効 有効
すべて		変換	セクション ページ	0b00101 0b00111	無効 有効	有効 有効
すべて		ドメイン	セクション ページ	0b01001 0b01011	有効 有効	有効 有効
すべて		アクセス許可	セクション ページ	0b01101 0b01111	有効 有効	有効 有効
VMSAv6		正確な外部アボート		0b01000	無効	有効
		正確な外部アボート（非推奨）		0b01010		
VMSAv6		TLB ロック[b]		0b10100	無効	無効
VMSAv6		コプロセッサデータアボート（実装定義）		0b11010	無効	無効
VMSAv6		不正確な外部アボート		0b10110	無効	無効
VMSAv6		パリティエラー例外		0b11000	無効	実装定義
VMSAv6	最低	デバッグイベント		0b00010	有効	予測不能

a. ドメインは DFSR でのみ有効です。

b. 詳細については、P. B4-51 *「変換とロックモデルによる TLB ロック解除プロシージャ」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.6.1 フォルトステータスレジスタのエンコードテーブルに関する注意事項

VMSAv6 以前は、ARMv6 としてマークされた項目に関連する FS[3:0] の使用方法は実装定義でした。これは、FS[10] のどの値についても当てはまります。

他の FS エンコードはすべて予約されています。

ドメインとフォルトアドレスの情報は、データアクセスについてのみ存在します。命令アボートの場合は、R14 を使用してフォルトアドレスを判定する必要があります。ドメイン情報は、フォルトアドレスについて TLB ルックアップを実行し、ドメインフィールドを抽出して判定できます。

VMSAv6 以降については、次の条件が適用されます。

- すべてのデータアボートはデータフォルトステータスレジスタ（DFSR）を更新するため、アボートの原因を判定できます。すべての命令アボートは命令フォルトステータスレジスタ（IFSR）を更新するため、アボートの原因を判定できます。
- 外部アボート（変換時に発生するもの以外）を除くすべてのデータアボートについて、フォルトアドレスレジスタ（FAR）にはアボートが発生したアドレスが書き込まれます。変換時以外の外部データアボートはすべて不正確な可能性があるため、FAR の内容はアボートの発生したアドレスではありません。不正確なアボートの詳細については、P. A2-23 *「不正確なデータアボート」*を参照して下さい。
- 修飾仮想アドレスによるデータキャッシュ保守操作時に変換アボートが発生した場合、データアボートが発生し、DFSR にアボートの原因が書き込まれます。FAR にはフォルトアドレスが書き込まれます。
- 命令キャッシュ保守操作時に正確なアボートが発生した場合、データアボートが発生し、DFSR は命令キャッシュ保守操作フォルトを示します。IFSR にはアボートの原因が書き込まれます。FAR にはフォルトの発生した修飾仮想アドレスが書き込まれます。
- WFAR には PC のコピーが書き込まれます。この値は ARM ステートでの実行時にはアドレス＋8、Thumb® ステートでの実行時にはアドレス＋4 です。この値は、アボートを発生した命令の仮想アドレスとの相対値で、修飾仮想アドレスではありません。
- WFAR は、ウォッチポイントアクセスを発生した命令のアドレスを保持するために使用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.6.2 アボート時の FSR/FAR の更新の概要

VMSAv6 で、どのアボートベクタが使用されるか、および各アボートタイプについてどのフォルトステータスレジスタとフォルトアドレスレジスタが更新されるかの概要を表 B4-7 に示します。IFAR はオプションです。

表 B4-7 アボート時の FSR/FAR の更新の概要

アボートタイプ	ベクタ	正確	IFSR	DFSR	FAR	WFAR	IFAR
命令 MMU フォルト	PABORT	はい	Y	N	N	N	Y
命令デバッグアボート	PABORT	はい	Y	N	N	N	UNP
変換時の命令外部アボート	PABORT	はい	Y	N	N	N	Y
命令外部アボート	PABORT	はい	Y	N	N	N	Y
命令キャッシュパリティエラー	PABORT	はい	Y	N	N	N	Y
命令キャッシュ保守操作	DABORT	はい	Y	Y	Y	N	N
データ MMU フォルト	DABORT	はい	N	Y	Y	N	N
データデバッグアボート	DABORT	いいえ	N	Y	N	Y	N
変換時のデータ外部アボート	DABORT	はい	N	Y	Y	N	N
データ外部アボート	DABORT	いいえ	N	Y	N	N	N
データキャッシュパリティエラー	DABORT	いいえ	N	Y	N	N	N
データキャッシュ保守操作	DABORT	はい	N	Y	Y	N	N

表中の記号は次の意味です。

Y　　このアボートタイプではレジスタが更新されます。

N　　このアボートタイプではレジスタが更新されません。

UNP　　予測不能

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.7 ハードウェアページテーブル変換

MMU は、次のセクションまたはページをベースとしたメモリアクセスをサポートしています。

スーパーセクション（オプション）
16MB ブロックのメモリで構成されます。

セクション　1MB ブロックのメモリで構成されます。

次のページサイズがサポートされています。

タイニーページ（**VMSAv6** ではサポートされていません）
1KB ブロックのメモリで構成されます。

スモールページ　4KB ブロックのメモリで構成されます。

ラージページ　64KB ブロックのメモリで構成されます。

セクションとラージページのサポートにより、1 つの TLB エントリを使用するだけで大きなメモリ領域のマッピングが可能です。追加のアクセス制御機構により、スモールページ内の 1KB のサブページや、ラージページ内の 16KB のサブページを作成できます。サブページ AP ビットの使用は、VMSAv6 では推奨されません。

メインメモリに保持されている変換テーブルには 2 つのレベルがあります。

第 1 レベルテーブル　セクションとスーパーセクションの変換と、第 2 レベルテーブルへのポインタを保持しています。

第 2 レベルテーブル　ラージページとスモールページの変換を保持しています。ページテーブルがコアースページテーブルではないときは、ファインページテーブルで、タイニーページをサポートしています。

MMU は、CPU により生成される修飾仮想アドレスを、外部メモリにアクセスするための物理アドレスに変換し、同時にアクセス許可の派生とチェックを行います。変換は TLB ミスの結果として発生し、第 1 レベルのフェッチから始まります。セクションマップされたアクセスでは、第 1 レベルのフェッチのみが必要です。これに対して、ページマップされたアクセスでは第 2 レベルのフェッチも必要になります。

ページテーブルルックアップのエンディアン形式を判定するには、システム制御コプロセッサの EE ビットの値が使用されます。詳細については、P. A2-34 *「エンディアンの構成と制御」*を参照して下さい。

注

ファインページテーブル形式とタイニーページのサポートは現在は使用されない形式で、これらの機能の定義は P. B4-35 *「ファインページテーブルとタイニーページのサポート」*のセクションで説明されています。

B4.7.1 変換テーブルのベース

変換処理は、要求された修飾仮想アドレスのエントリがオンチップ TLB に含まれていないときに開始されます。*変換テーブルベースレジスタ*（CP15 のレジスタ 2 の TTBR）には、第 1 レベルテーブルのベースとなる物理アドレスが保持されています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

VMSAv6 以前は、TTBR は 1 つだけ存在していました。変換テーブルベースレジスタのビット [31:14] のみが有意で、ビット [13:0] は 0 の必要がありました。このため、第 1 レベルページテーブルは 16KB 境界に存在している必要がありました。

VMSAv6 では、追加の変換テーブルベースレジスタと、変換テーブルベース制御レジスタとして、TTBR0、TTBR1、TTBCR が導入されています。TLB ミスが発生した場合、修飾仮想アドレスの最上位ビットにより、第 1 レベルと第 2 レベルのどちらの変換テーブルベースを使用するかが判定されます。使用モデルの詳細については、P. B4-25 「*VMSAv6 でのページテーブル変換*」を参照して下さい。

TTBR1 はオペレーティングシステムと I/O アドレスでの使用を想定したもので、コンテキストスイッチにより変更されません。TTBR0 はプロセス固有のアドレスでの使用を想定したものです。TTBCR が 0 にプログラムされている場合、すべての変換は TTBR0 を使用し、アーキテクチャの従来のバージョンと互換となります。TTBR1 テーブルのサイズは常に 16KB ですが、TTBR0 テーブルのサイズは TTBCR の値 (N) により、128 バイトから 16KB まで変化します。ここで、N = 0 〜 7 です。すべての変換テーブルは、そのテーブルサイズの境界でアラインしている必要があります。

VMSAv6 では、TTBR の最下位ビットとして制御ビットフィールドも導入されています。詳細については、P. B4-25 「*VMSAv6 でのページテーブル変換*」を参照して下さい。

B4.7.2 第 1 レベルフェッチ

変換テーブルベースレジスタのビット [31:14] は、修飾仮想アドレスのビット [31:20]、および 2 つの 0 ビットと連結され、図 B4-3 に示す 32 ビットの物理アドレスとなります。このアドレスは、4 バイトの変換テーブルエントリで、セクションの第 1 レベル記述子か、第 2 レベルページテーブルへのポインタです。

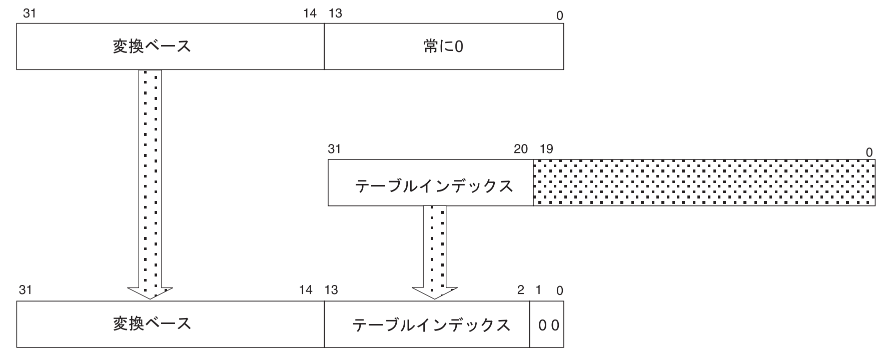


図 B4-3 変換テーブルの第 1 レベル記述子へのアクセス

——— 注 ———

VMSAv6 では、TTBR1 が選択されている場合の*変換*ベースは常にアドレス [31:14] です。ただし、TTBR0 で使用される値は、TTBCR の値が N＝0 から N＝7 まで変化するのに対応して、アドレス [31:14] からアドレス [31:7] まで変化します。図 B4-3 から P. B4-34 図 B4-7 までに示す X の値は、TTBR1 が使用される場合は 0、TTBR0 が使用される値は TTBCR の値 N です。

VMSAv6 以前は TTBR0 のみが存在し、これらの図に示す X の値は常に 0 でした。

———————

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.7.3 VMSAv6 でのページテーブル変換

VMSAv6 では、2 つのページテーブルフォーマットがサポートされています。

- サブページアクセス許可をサポートする、下位互換のフォーマット。このフォーマットは、特定のページテーブルエントリが拡張領域タイプをサポートするように拡張されています。
- サブページアクセス許可をサポートせず、VMSAv6 の機能をサポートする新しいフォーマット。次のような機能がサポートされています。
  - 拡張領域タイプ
  - グローバルとプロセス固有のページ
  - アクセス許可の追加
  - 共有領域と非共有領域
  - 実行不可領域

サブページについては、P. B4-31 *「第 2 レベル記述子 - コアースページテーブルのフォーマット」*を参照して下さい。

ハードウェアページテーブルウォークによってレベル 1 統合 / データキャッシュからの読み出しが発生するかどうかは実装定義です。ハードウェアページテーブルウォークで、TCM からの読み出しが発生することはありません。変換テーブルベースレジスタの RGN、P、S、C ビットは、ページテーブルウォークのメモリ領域属性を決定します。レベル1キャッシュからのページテーブルアクセスをサポートしない実装では、コヒーレンシを保証するためにページテーブルはライトスルーキャッシュを使用して格納するか、ライトバックキャッシュが使用されている場合は、変更後に対応するキャッシュエントリをクリーニングする必要があります。ページテーブルウォークは、P. B4-38 *「ファインページテーブルのページ参照の変換」*で説明されている TTBR 領域（RGN）ビットの定義に従って、外部キャッシュ可能でアクセス可能な場合があります。

ページテーブルフォーマットは、CP15 のレジスタ 1 の XP ビットによって選択されます。サブページ AP ビットが有効な場合（CP15 のレジスタ 1 の XP = 0）、ページテーブルのフォーマットは ARMv4/v5 と互換であり、以下の条件が適用されます。

- すべてのマッピングはグローバルで、実行可能とみなされます（XN = 0）。
- すべての通常メモリは非共有です。
- デバイスメモリは、TEX + CB ビットにより共有または非共有に決定されます。
- AP3、AP2、AP1、AP0 が異なる値を含むサブページ AP ビットの使用は推奨されません。

サブページ AP ビットが無効な場合（CP15 のレジスタ 1 の XP = 1）、ページテーブルは ARMv6 の MMU 機能をサポートしています。これらの機能をサポートするため、ページテーブルに新しいビットが追加されています。

- 非グローバル（nG）ビット : 変換を TLB でグローバル (0)、プロセス固有 (1) のどちらにマーク付けするかを示します。プロセス固有の変換の場合、コンテキスト ID レジスタから現在の ASID を取得して TLB へのルックアップが行われます。
- 共有 (S) ビット : 変換が非共有 (0) メモリと共有 (1) メモリのどちらに対してのものかを示します。このビットは、通常メモリ領域にのみ適用されます。デバイスメモリは、TEX + CB ビットにより共有または非共有に決定されます。ストロングオーダメモリは、常に共有として扱われます。
- 実行不可（XN）ビット : 領域が実行可能 (0) か実行不可 (1) かを示します。
- 3 つのアクセス許可ビット : アクセス許可拡張（APX）ビットにより、アクセス許可を追加指定できます。
- すべてのページテーブルマッピングは TEX フィールドをサポートしています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

—— 注 ——

VMSAv6 では、無効なエントリ（ビット [1:0] = 0b00）または予約エントリ（ビット [1:0] = 0b11）は変換フォルトを発生します。

以下のセクションでは、第 1 レベルと第 2 レベルアクセス機構について説明し、VMSAv6 とそれ以前のバージョンのアーキテクチャについて各種のテーブルフォーマットを定義します。

B4.7.4 第 1 レベル記述子

第 1 レベルテーブルの各エントリは、それに関連する 1MB の修飾仮想アドレス範囲がどのようにマップされるかを示す記述子です。第 1 レベルページテーブルエントリのビット [1:0] は、第 1 レベル記述子のタイプを次のように示しています。

- ビット [1:0] == 0b00 の場合、関連する修飾仮想アドレスはマップされておらず、そのアドレスにアクセスを試みた場合は変換フォルトが発生します（P. B4-14 *「アボート」* 参照）。これらの記述子のビット [31:2] はハードウェアで無視されるため、ソフトウェアで独自の目的に使用できます。必要に応じて、ビット [31:2] には常に記述子の有効なアクセス許可を保持することをお勧めします。
- ビット [1:0] == 0b10 の場合、そのエントリは関連する修飾仮想アドレスのセクション記述子です。この場合の解釈の詳細については、P. B4-28 *「セクションとスーパーセクション」* を参照して下さい。
- ビット [1:0] == 0b01 の場合、そのエントリはコアース第 2 レベルテーブルの物理アドレスで、関連する 1MB の修飾仮想アドレス範囲のマップ方法を示しています。コアーステーブルはテーブルごとに 1KB を必要とし、ラージページとスモールページの両方をマップ可能です（P. B4-30 *「コアースページテーブル記述子」* 参照）。
- ビット [1:0] == 0b11 の場合、そのエントリは VMSAv6 以前のファイン第 2 レベルテーブルの物理アドレスで、VMSAv6 では予約されています。詳細については、P. B4-35 *「ファインページテーブルとタイニーページのサポート」* を参照して下さい。

第 1 レベル記述テーブルの形式には 2 種類あります。

- P. B4-27 表 B4-8 に示す、VMSAv6 でサブページが有効な形式
- P. B4-27 表 B4-9 に示す、VMSAv6 でサブページが無効な形式

AP、APX、ドメインフィールドの説明については、P. B4-8 *「メモリアクセスの制御」* を参照して下さい。C、B、TEX フィールドの説明については、P. B4-11 *「メモリ領域属性」* を参照して下さい。

実際定義（IMP）ビット [9] は、実装定義の機能が有効な場合のみ 1 に、それ以外の場合は 0 にセットします。このビットが 0 の場合、ビットは無視されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B4-8 第 1 レベル記述子の形式（VMSAv6、サブページ有効）

	31–20	19–15	14–12	11–10	9	8–5	4	3	2	1	0
フォルト	IGN									0	0
コアースページテーブル	コアースページテーブルのベースアドレス				IMP	ドメイン	SBZ			0	1
セクション	セクションのベースアドレス	SBZ	TEX	AP	IMP	ドメイン	SBZ	C	B	1	0
	予約									1	1

表 B4-9 第 1 レベル記述子の形式（VMSAv6、サブページ無効）

	31–24	23–20	19	18	17	16	15	14–12	11–10	9	8–5	4	3	2	1	0
フォルト	IGN														0	0
コアースページテーブル	コアースページテーブルのベースアドレス									IMP	ドメイン	SBZ			0	1
セクション	セクションのベースアドレス		SBZ	0	nG	S	APX	TEX	AP	IMP	ドメイン	XN	C	B	1	0
スーパーセクション	スーパーセクションのベースアドレス	ベースアドレス [35:32]	SBZ	1	nG	S	APX	TEX	AP	IMP	ベースアドレス [39:36]	XN	C	B	1	0
	予約														1	1

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.7.5 セクションとスーパーセクション

ビット [1:0] == 0b10 の場合、その第 1 レベル記述子は 1MB セクションか 16MB スーパーセクションの記述子です。

スーパーセクションはオプションです。使用する場合、32 ビットの修飾仮想アドレスをより大きな物理アドレス空間（アドレスを最大 8 ビットまで拡張できます）に変換し、次のように定義されます。

- ビットフィールドは、VMSAv6 で修正された形式で記述されています。詳細については、P. B4-27 *「第 1 レベル記述子の形式（VMSAv6、サブページ無効）」*を参照して下さい。

ビット **[1:0]**	= 0b10
ビット **[18]**	= 0 の場合、1MB セクションを定義します。
	= 1 の場合、16MB スーパーセクションを定義します。
ビット **[8:5]**	オプションの拡張物理アドレスビット、PA[39:36]
ビット **[23:20]**	オプションの拡張物理アドレスビット、PA[35:32]

- いくつの追加アドレスビットをサポートするかは実装定義です。
- スーパーセクションのデフォルトはドメイン 0 です。
- ARMv6　以前のセクション記述子フォーマットでスーパーセクションが提供されるかどうかは実装定義です。

P. B4-29 図 B4-4 は、セクションについて仮想アドレスから物理アドレスを生成する方法です。記述子の影付きの部分は、アクセス制御データフィールドを示しています。

——— 注 ———

第 1 レベル記述子のアクセス許可は、物理アドレスの生成前にチェックする必要があります。アクセス許可をチェックする手順については、P. B4-8 *「アクセス許可」*を参照して下さい。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

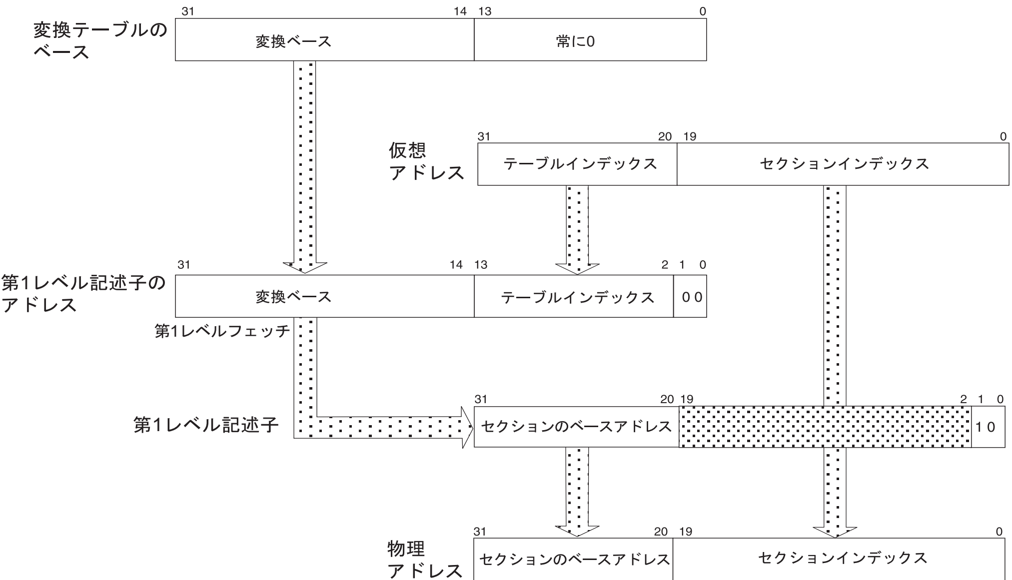


図 B4-4 セクションの変換

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.7.6 コアースページテーブル記述子

第 1 レベル記述子がコアースページテーブル記述子の場合、フィールドの意味は次のとおりです。

ビット [1:0] 記述子のタイプを識別します（0b01 はコアースページテーブル記述子を示します）。

ビット [4:2] これらのビットの意味は実装定義です。VMSAv6 以降は、これらのビットは常に 0 です。

ビット [8:5] ドメインフィールドで、この記述子により制御されるすべてのページについて、可能な 16 個のドメインのいずれかを指定します。

ビット [9] 実装定義。

ビット [31:10] ページテーブルベースアドレスはコアース第 2 レベルページテーブルへのポインタで、実行する第 2 レベルフェッチのベースアドレスを指定します。コアース第 2 レベルページテーブルは 1KB 境界にアラインしている必要があります。

第 1 レベルフェッチによりコアースページテーブル記述子が返された場合、図 B4-5 に示すように、第 2 レベル記述子を取得するために第 2 レベルフェッチが開始されます。記述子の影付きの部分は、アクセス制御データフィールドを示しています。

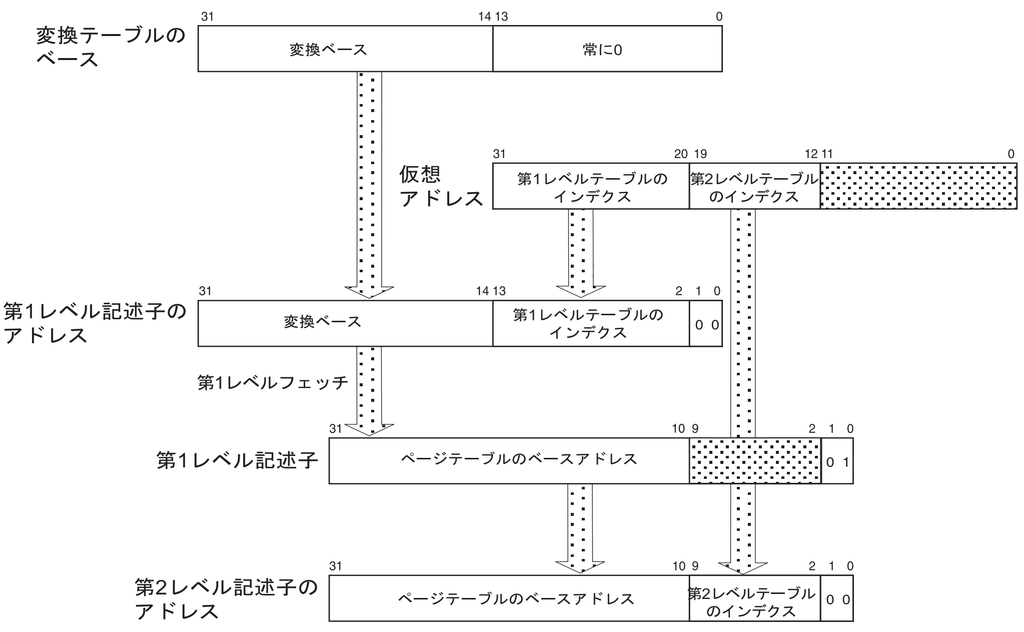


図 B4-5 コアースページテーブルの第 2 レベル記述子へのアクセス

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.7.7 第 2 レベル記述子 - コアースページテーブルのフォーマット

コアーステーブルのサイズは 1KB です。各 32 ビットエントリは、4KB のメモリについての変換情報を提供します。

VMSAv6 では、2 つのページサイズがサポートされています。

- ラージページのサイズは 64KB です。
- スモールページのサイズは 4KB です。

第 2 レベルテーブルは、各エントリでマップできるページサイズより大きいページサイズをサポートできます。このためには、ページテーブルエントリを適切な数だけテーブル内で複製する必要があります。また、以下の条件が適用されます。

- コアーステーブルの場合、ラージページには 16 個の複製エントリが必要です。

第 2 レベル記述子テーブルのフォーマットには 2 種類あります（コアースページフォーマット）。

- 表 B4-10 に示す、サブページが有効なフォーマット
- 表 B4-11 に示す、サブページが無効なフォーマット

表 B4-10 第 2 レベル記述子のフォーマット（サブページ有効）

	31–16	15	14–12	11	10	9	8	7	6	5	4	3	2	1	0
フォルト	IGN													0	0
ラージページ	ラージページのベースアドレス	SBZ	TEX	AP3		AP2		AP1		AP0		C	B	0	1
スモールページ	スモールページのベースアドレス			AP3		AP2		AP1		AP0		C	B	1	0
拡張スモールページ	拡張スモールページのベースアドレス			SBZ			TEX			AP		C	B	1	1

表 B4-11 第 2 レベル記述子のフォーマット（サブページ無効）

	31–16	15	14–12	11	10	9	8	7	6	5	4	3	2	1	0
フォルト	IGN													0	0
ラージページ	ラージページのベースアドレス	XN	TEX	nG	S	APX	SBZ			AP		C	B	0	1
拡張スモールページ	拡張スモールページのベースアドレス			nG	S	APX	TEX			AP		C	B	1	XN

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 2 レベルページテーブルの記述子フィールド

第 2 レベルページテーブルのフィールドの意味は次のとおりです。

ビット [1:0] 記述子のタイプを識別します（修正された VMSAv6 フォーマットの XN ビットも含みます）。

ビット [3:2] キャッシュ可能ビットとバッファ可能ビット。

ビット [5:4] ページ全体へのアクセス許可ビット、または AP0 サブページ。

次のビットは、対応する物理アドレスビットに使用されます。フィールドサイズはページサイズに依存します。

ビット [31:16] ラージ（64 KB）ページ

ビット [31:12] ラージ（4 KB）ページ

次のビットは、追加のアクセス制御機能に使用されます。

ビット [15:6] フォーマットにより、次の意味を持ちます。

- ラージページ制御
- サブページのアクセス許可
- TEX
- APX
- S
- nG
- XN

ビット [11:6] フォーマットにより、次の意味を持ちます。

- スモールページ制御
- サブページのアクセス許可
- TEX
- APX
- S
- nG

ビット [9:6] タイニーページ制御、常に 0。

これらのフィールドの詳細については、以下のセクションを参照して下さい。

AP と APX　P. B4-8「*アクセス許可*」を参照して下さい。

C、B、TEX　P. B4-11「*C、B、TEX のエンコード*」を参照して下さい。

XN、nG、S　P. B4-25「*VMSAv6 でのページテーブル変換*」を参照して下さい。

サブページがサポートされている場合、ページはそれぞれ同じサイズの 4 つのブロックに分割されます。AP0 はブロックで最も下位のブロックベースアドレスにあるブロックを指し、AP1、AP2、AP3 は順に上位のブロックベースアドレスのブロックを指します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.7.8 コアースページテーブルのページ参照の変換

図 B4-6 は、コアース第 2 レベルテーブルの 64KB ラージページの完全な変換手順です。

――― 注 ―――

ページインデックスの上位 4 ビットと、第 2 レベルテーブルインデックスの下位 4 ビットはオーバラップしているため、ラージページの各ページテーブルエントリは、コアースページテーブル内で 16 回繰り返し（連続したメモリロケーションに）存在している必要があります。

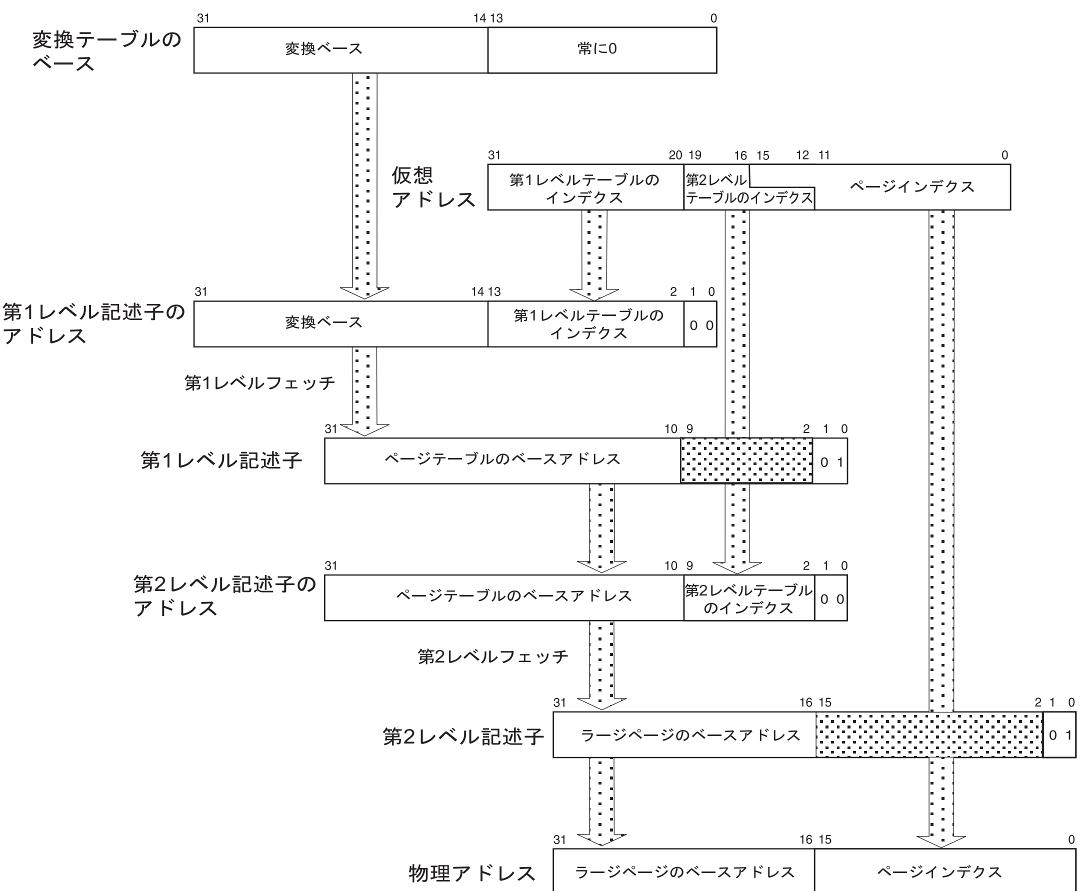


図 B4-6 コアース第 2 レベルテーブルでのラージページ変換

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

図 B4-7 は、コアース第 2 レベルテーブルの 4KB スモールページ（標準または拡張）の完全な変換手順です。

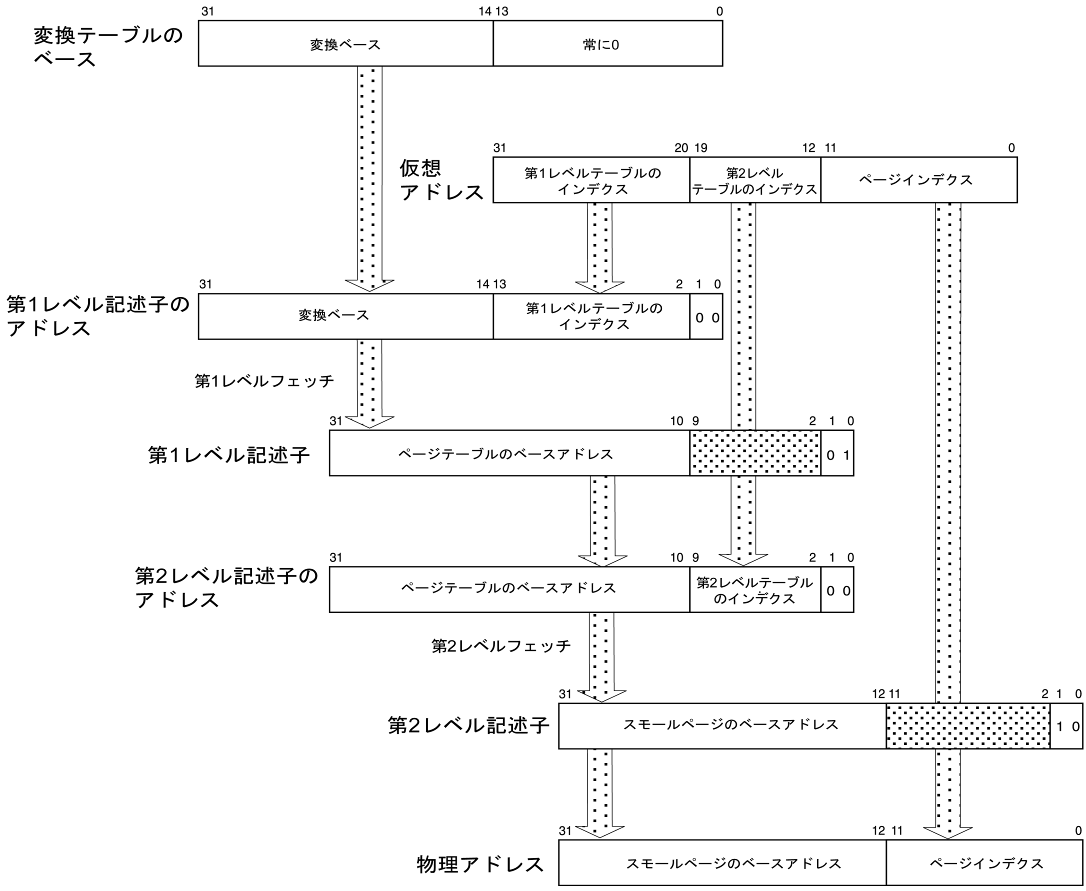


図 B4-7 コアース第 2 レベルテーブルでのスモールページ変換

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.8 ファインページテーブルとタイニーページのサポート

タイニーページとファインページのテーブルフォーマットは、VMSAv6 では使用されていません。このため、タイニーページのサポートと、関連する第 1 および第 2 レベル記述子の定義は、P. B4-23 *「ハードウェアページテーブル変換」*にあるコアースページテーブルフォーマットとは別に記載されています。

B4.8.1 第 1 レベル記述子

第 1 レベルテーブルの各エントリは､それに関連する 1MB の修飾仮想アドレス範囲がどのようにマップされるかを示す記述子です。第 1 レベルページテーブルエントリのビット [1:0] は、第 1 レベル記述子のタイプを次のように示しています。

- ビット [1:0] == 0b00 の場合、関連する修飾仮想アドレスはマップされておらず、そのアドレスにアクセスを試みた場合は変換フォルトが発生します（P. B4-14 *「アボート」*参照）。これらの記述子のビット [31:2] はハードウェアで無視されるため、ソフトウェアで独自の目的に使用できます。必要に応じて、ビット [31:2] には常に記述子の有効なアクセス許可を保持することをお勧めします。
- ビット [1:0] == 0b10 の場合、そのエントリは関連する修飾仮想アドレスのセクション記述子です。この場合の解釈の詳細については、P. B4-28 *「セクションとスーパーセクション」*を参照して下さい。
- ビット [1:0] == 0b01 の場合、そのエントリはコアース第 2 レベルテーブルの物理アドレスで、関連する 1MB の修飾仮想アドレス範囲のマップ方法を示しています。
- ビット [1:0] == 0b11 の場合､そのエントリはファイン第 2 レベルテーブルの物理アドレスです。ファイン第 2 レベルページテーブルは、関連する 1MB の修飾仮想アドレス範囲がどのようにマップされるかを示します。このページテーブルはテーブルごとに 4KB を必要とし、ラージページ、スモールページ、タイニーページをマップ可能です。詳細については P. B4-35 *「ファインページテーブルとタイニーページのサポート」*を参照して下さい。

ファインページテーブルをサポートする第 1 レベル記述子フォーマットを表 B4-12 に示します。

AP とドメインフィールドの説明については、P. B4-8 *「メモリアクセスの制御」*を参照して下さい。C フィールドと B フィールドの説明については、P. B4-11 *「メモリ領域属性」*を参照して下さい。

表 B4-12 第 1 レベル記述子のフォーマット

<table>
<tr><th></th><th>31 … 20</th><th>19 … 14 … 12</th><th>11 10</th><th>9</th><th>8 … 5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>フォルト</td><td colspan="8">IGN</td><td>0</td><td>0</td></tr>
<tr><td>コアースページテーブル</td><td colspan="3">コアースページテーブルのベースアドレス</td><td>SBZ</td><td>ドメイン</td><td colspan="3">IMP</td><td>0</td><td>1</td></tr>
<tr><td>セクション</td><td>セクションのベースアドレス</td><td>SBZ</td><td>AP</td><td>SBZ</td><td>ドメイン</td><td>IMP</td><td>C</td><td>B</td><td>1</td><td>0</td></tr>
<tr><td>ファインページテーブル</td><td colspan="2">ファインページテーブルのベースアドレス</td><td colspan="2">SBZ</td><td>ドメイン</td><td colspan="3">IMP</td><td>1</td><td>1</td></tr>
</table>

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.8.2 第 2 レベル記述子

ファインテーブルのサイズは 4KB です。各 32 ビットエントリは、1KB のメモリについての変換情報を提供します。

VMSA では、3 つのページサイズがサポートされています。

- ラージページのサイズは 64KB です。
- スモールページのサイズは 4KB です。
- タイニーページのサイズは 1KB です。

第 2 レベルテーブルは、各エントリでマップできるページサイズより大きいページサイズをサポートできます。このため、タイニーページはファインページテーブルでのみサポートされます。エントリサイズより大きなページをマップするには、ページテーブルエントリをスモールページの場合は 4 回、ラージページの場合は 64 回複製する必要があります。

ファインページテーブルをサポートする第 2 レベル記述子のフォーマットを表 B4-13 に示します。

表 B4-13 第 2 レベル記述子のフォーマット

	31 … 16	15 … 12	11 10	9 8	7 6	5 4	3	2	1	0
フォルト	IGN								0	0
ラージページ	ラージページのベースアドレス	SBZ	AP3	AP2	AP1	AP0	C	B	0	1
スモールページ	スモールページのベースアドレス		AP3	AP2	AP1	AP0	C	B	1	0
タイニーページ	ラージページのベースアドレス			SBZ		AP	C	B	1	1

第 1 レベル記述子がファインページテーブル記述子の場合、フィールドの意味は次のとおりです。

ビット [1:0] 記述子のタイプを識別します（0b11 はファインページテーブル記述子を示します）。

ビット [4:2] これらのビットの意味は実装定義です。

ビット [8:5] ドメインフィールドで、この記述子により制御されるすべてのページについて、可能な 16 個のドメインのいずれかを指定します。

ビット [11:9] これらのビットは現在使用されていません。常に 0 にして下さい。

ビット [31:12] ページテーブルベースアドレスはファイン第 2 レベルページテーブルへのポインタで、実行する第 2 レベルフェッチのベースアドレスを指定します。ファイン第 2 レベルページテーブルは 4KB 境界にアラインしている必要があります。

第 1 レベルフェッチによりファインページテーブル記述子が返された場合、P. B4-37 図 B4-8 に示すように、第 2 レベル記述子を取得するために第 2 レベルフェッチが開始されます。記述子の影付きの部分は、アクセス制御データフィールドを示しています。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

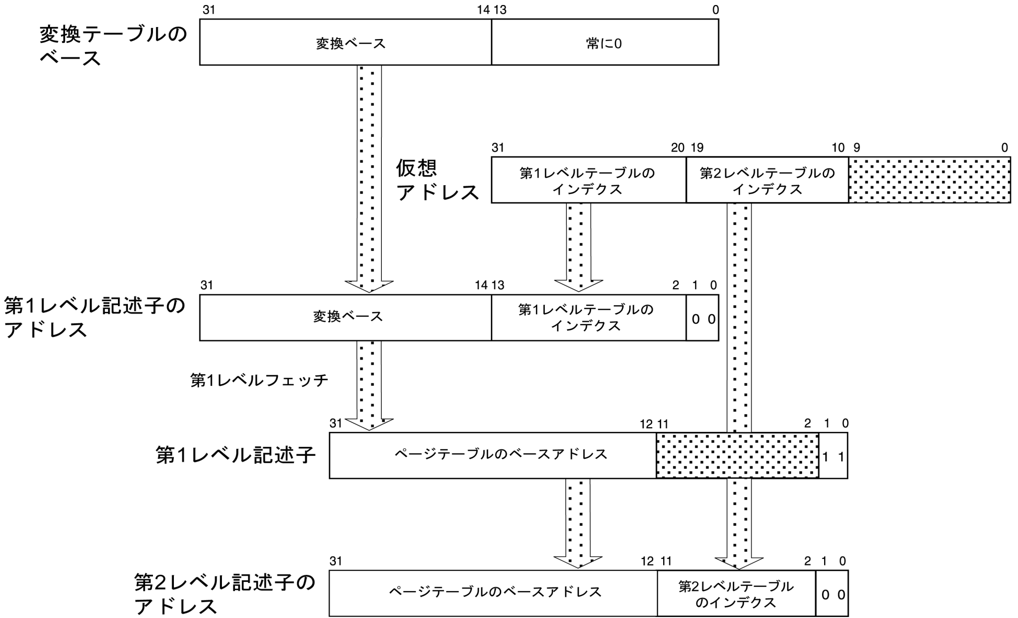


図 B4-8 ファインページテーブルの第 2 レベル記述子へのアクセス

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.8.3 ファインページテーブルのページ参照の変換

ファイン第 2 レベルテーブルでのラージまたはスモールページの変換手順はコアースページの場合とほぼ同じですが、第 2 レベル記述子のアドレスは図 B4-9 に示すように決定されます。

スモールページがファイン第 2 レベルテーブルに存在する場合、ページインデクスの上位 2 ビットと第 2 レベルテーブルインデックスの下位 2 ビット（ビット [11:10]）はオーバラップします。スモールページの各ページテーブルエントリは、ファインページテーブル内の連続したメモリ場所に 4 個存在する必要があります。ラージページの場合、ビット [15:10] の 6 ビットがオーバラップし、各ページエントリは 64 個存在している必要があります。

タイニーページにはオーバラップはなく、1KB ページごとにエントリがあります。図 B4-9 は、ファイン第 2 レベルテーブルの 1KB タイニーページの完全な変換シーケンスです。

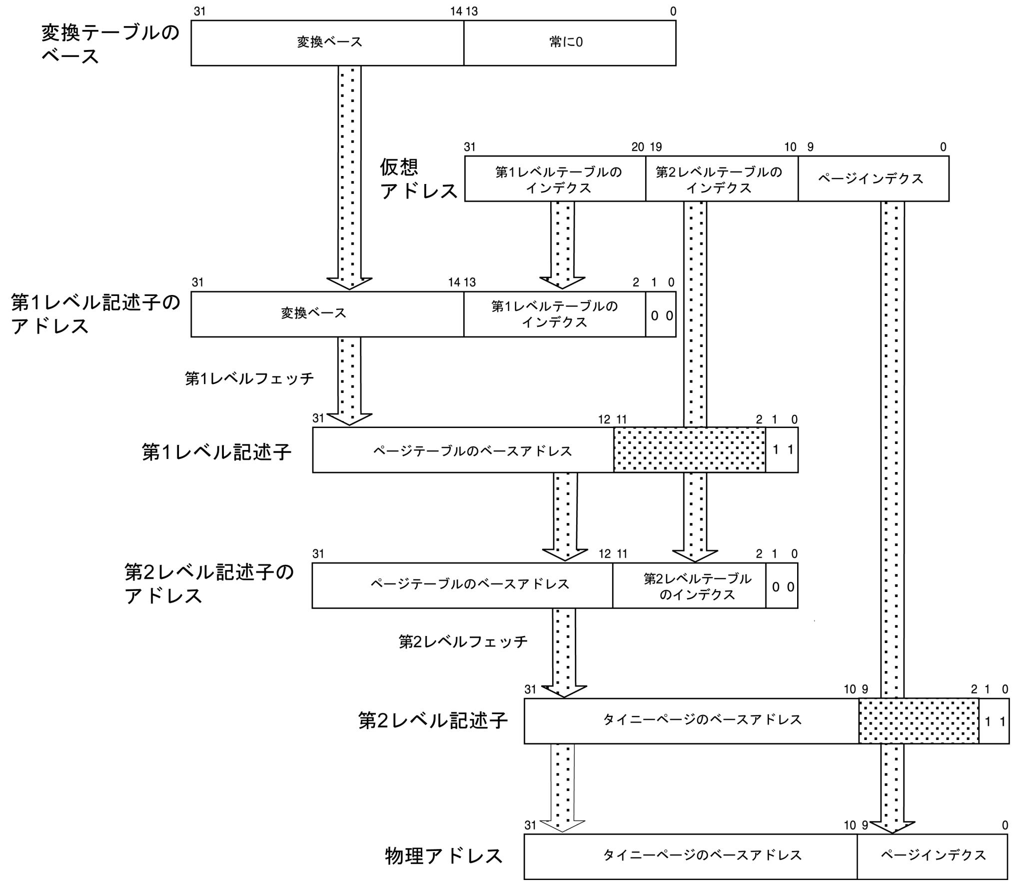


図 B4-9 ファイン第 2 レベルテーブルでのタイニーページ変換

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.9 CP15 のレジスタ

MMU は、システム制御コプロセッサのレジスタにより制御されます。VMSAv6 からは、いくつかの新しいレジスタとレジスタフィールドが追加されています。

- レジスタ 0 の TLB タイプレジスタ
- レジスタ 1 の追加制御ビット
- 第 2 変換テーブルベースレジスタと、レジスタ 2 の新しい制御フィールド
- レジスタ 5 へのフォルトステータスレジスタの追加
- レジスタ 6 へのフォルトアドレスレジスタの追加
- レジスタ 8 での ASID による TLB 無効化のサポート
- レジスタ 13 での ASID 制御

ドメインサポート（レジスタ 3）と TLB ロックダウンのサポート（レジスタ 10）は、以前のバージョンのアーキテクチャと同じです。

VMSA に関連するレジスタはすべて、次の形式の命令でアクセスされます。

```
MRC p15, 0, Rd, CRn, CRm, opcode_2
MCR p15, 0, Rd, CRn, CRm, opcode_2
```

ここで、CRn はシステム制御コプロセッサのレジスタです。特に指定されていない限り、CRm と opcode_2 は常に 0 です。

B4.9.1 レジスタ 0: TLB タイプレジスタ（VMSAv6）

TLB のサイズと構成は実装定義です。このレジスタは読み出し専用で、ロック可能な TLB エントリの数と、個別の命令用とデータ用の TLB と、統合 TLB のどちらを持っているかを示しています。これによって、オペレーティングシステムは TLB の管理方法を決定できます。TLB タイプレジスタには、opcode_2 フィールドを 0b011 に設定して CP15 のレジスタ 0 を読み出すことでアクセスできます。次に例を示します。

```
MRC p15, 0, Rd, c0, c0, 3 ; returns TLB Type register
```

ビット **[0]** 0 = 統合 TLB
1 = 命令 TLB とデータ TLB が独立に存在

ビット **[7:1]** 常に 0

ビット **[15:8]** ロック可能な統合 / データ TLB エントリ数 0 <= N <= 255

ビット **[23:16]** ロック可能な命令 TLB エントリ数 0 <= N <= 255。統合 TLB の場合、ビット [23:16] は常に 0

ビット **[31:24]** 常に 0

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.9.2 レジスタ 1: 制御レジスタ

システム制御コプロセッサのレジスタ 1 にある以下のビットにより、MMU を制御します。

M（ビット **[0]**） MMU の許可 / 禁止ビット。

0 = MMU は禁止されています。

1 = MMU は許可されています。

MMU やメモリ保護ユニット（MPU）がないシステムでは、このビットから 0 が読み出され、書き込みは無視されます。

A（ビット **[1]**） アライメントフォルトチェックの許可 / 禁止ビット。詳細については P. B4-15 *「アライメントフォルト」*を参照して下さい。

0 = アライメントフォルトチェックは禁止されています。

1 = アライメントフォルトチェックは許可されています。

W（ビット **[3]**） ライトバッファの許可 / 禁止ビット。

0 = ライトバッファは禁止されています。

1 = ライトバッファは許可されています。

実装で W ビットを省くこともできます。その場合、このビットからは 1 が読み出され、書き込みは無視されます。

S（ビット **[8]**） システム保護ビット。この機能は VMSAv6 からは非推奨です。このビットの影響については、P. B4-8 *「アクセス許可」*を参照して下さい。

R（ビット **[9]**） ROM 保護ビット。この機能は VMSAv6 からは非推奨です。このビットの影響については、P. B4-8 *「アクセス許可」*を参照して下さい。

XP（ビット **[23]**）

拡張ページテーブルの構成。このビットは、ハードウェアページテーブルの変換機構を構成します。

0 = VMSAv4/v5 と VMSAv6、サブページ有効

1 = VMSAv6、サブページ無効

EE（ビット **[25]**）

例外エンディアンビットで、VMSAv6 にのみ存在します。EE ビットは、割り込みベクタへのエントリ時（リセットを含む）の CPSR E ビットの値を定義するために使用されます。この値は、ページテーブルルックアップに使用されるページテーブルデータのエンディアン形式を示すためにも使用されます。詳細については、P. A2-34 *「エンディアンの構成と制御」*を参照して下さい。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.9.3 レジスタ 2: 変換テーブルのベース

表 B4-14 に示すように、変換テーブルベースレジスタが 2 つ、制御レジスタが 1 つ用意されています。

表 B4-14 変換テーブルレジスタ

レジスタ名	Opcode2
変換テーブルベース 0（TTBR0）	0
変換テーブルベース 1（TTBR1）	1
変換テーブルのベース制御	2

変換テーブルのベース制御レジスタは、特定の修飾仮想アドレスにページテーブルミスが発生した場合、変換テーブルベースレジスタの 0 と 1 のどちらを使用するかを指定します。形式は次のとおりです。

31　　　　　　　　　　3	2　　0
予測不能 / 常に 0	N

ページテーブルのベースレジスタは、次のように選択されます。

N = 0 の場合、常に TTBR0 が使用されます。N = 0（リセット時の状態）の場合、変換テーブルのベースはアーキテクチャの以前のバージョンに下位互換です。

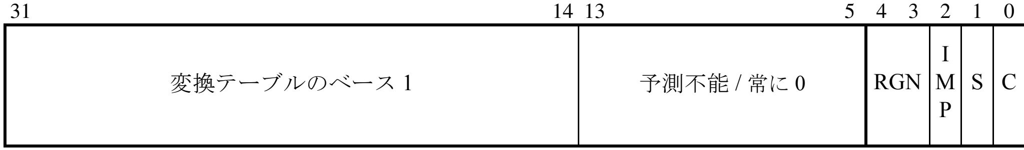
N > 0 の場合、修飾仮想アドレスのビット [31:32-N] がすべて 0 であれば TTBR0 が、それ以外の場合は TTBR1 が使用されます。N の範囲は 0 <= N <= 7 です。このため、N = 1 ならば、VA[31] == 0 の場合は TTBR0 が、それ以外の場合は TTBR1 が使用されます。N = 2 ならば、VA[31:30] == 0b00 の場合は TTBR0 が、それ以外の場合は TTBR1 が使用されます。

TTBR0 の形式は次のとおりです。

31　　　　14 - n	13 - n　　　5	4　3	2	1	0
変換テーブルのベース 0	予測不能 / 常に 0	RGN	IMP	S	C

変換テーブルベース 0 レジスタのビット [31:14-N] のみが有意です。このため、たとえば N = 0 の場合はページは 16KB 境界に、N = 1 の場合は 8KB 境界に存在している必要があります。

TTBR1 の形式は次のとおりです。



変換テーブルベース 1 レジスタのビット [31:14] のみが有意です。このため、TTBR1 は 16KB 境界に存在している必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

これら 2 つのページテーブルベースレジスタの予測される使用方法では、TTBR1 がオペレーティングシステムと I/O アドレスに使用されます。これらは、コンテキストスイッチにより変更されません。TTBR0 は、プロセス固有のアドレスに使用され、各プロセスは独立の第 1 レベルページテーブルを持ちます。コンテキストスイッチ時には、TTBR0 とコンテキスト ID レジスタが変更されます。

RGN、IMP、S、C ビットは、ページテーブルウォークのメモリ属性を制御します。

RGN ページテーブルメモリが、レベル2以降のメモリでキャッシュ可能かどうかを示します。

- **00** VMSAv5: 外部キャッシュ不可
  VMSAv6: 通常メモリキャッシュ不可
- **01** 予測不能
- **10** 外部でライトスルーキャッシュ可能
- **11** 外部でライトバックキャッシュ可能

IMP 実装定義、使用しない場合は常に 0。

S ページテーブルウォークが共有 (1) と非共有 (0) のどちらのメモリに対してのものかを示します。

C ページテーブルウォークが内部キャッシュ可能 (1) か内部キャッシュ不可 (0) かを示します。

——— 注 ———

ページテーブルウォークによってレベル1キャッシュからの読み出しが可能かどうかは実装定義です。このため、コヒーレンシを確保するために、各ページテーブルを内部ライトスルーメモリに格納するか、内部ライトバックの場合は変更後に対応するキャッシュエントリをクリーニングし、ハードウェアページテーブルウォーク機構から可視にする必要があります。

B4.9.4 レジスタ 3: ドメインアクセス制御

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

ドメインアクセス制御レジスタは 16 個の 2 ビットフィールドで構成され、各フィールドは 16 個のドメインのそれぞれについてアクセス許可を定義しています。ドメインの値は P. B4-10「*ドメイン*」で定義されています。

B4.9.5 レジスタ 4: 予約

CP15 のレジスタ 4 に読み出しまたは書き込みを行った場合、動作は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.9.6 レジスタ 5: フォルトステータス

このレジスタにより、表 B4-15 に示すように Opcode2 フィールドの値に従って、データおよび命令フォルトステータスレジスタへアクセスできます。VMSAv6 以前は、統合 FSR のみが定義されていました。

表 B4-15 フォルトステータスレジスタ

レジスタ名	Opcode2
統合 / データ FSR	0
命令 FSR	1

CP15 のレジスタ 5 を読み出すと、データまたは命令のフォルトステータスレジスタ（DFSR/IFSR）の値が返されます。フォルトステータスレジスタには、最後のアボートのソースが含まれています。この値は、アボートが発生したときに行われていたアクセスのドメイン（存在する場合）とタイプを示しています。

ビット **[11]** VMSAv6 で追加されたものです。アボートされたデータアクセスが読み出し (0) と書き込み (1) どちらのアクセスだったかを示しています。CP15 キャッシュ保守操作フォルトの場合、読み出される値は 1 です。IFSR の場合、このビットは常に 0 です。

ビット **[7:4]** DFSR 専用で、メモリシステムアボートが発生したときにアクセスされていたドメインを示しています。

ビット **[10, 3:0]** アボートの原因。詳細については、P. B4-20 表 B4-6 を参照して下さい。

IFSR と DFSR は、読み出し / 書き込みレジスタです。デバッガでコンテキストの保存と復元に使用できます。

データフォルトステータスレジスタ

DFSR のフォーマットは次のとおりです。

<table>
<tr><th>31 … 12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7 … 4</th><th>3 … 0</th></tr>
<tr><td>予測不能 / 常に 0</td><td>W
R</td><td>FS
[4]</td><td>UNP/
SBZ</td><td>0</td><td>ドメイン</td><td>ステータス</td></tr>
</table>

命令フォルトステータスレジスタ

IFSR のフォーマットは次のとおりです。

<table>
<tr><th>31 … 11</th><th>10</th><th>9 … 4</th><th>3 … 0</th></tr>
<tr><td>予測不能 / 常に 0</td><td>FS
[4]</td><td>予測不能 / 常に 0</td><td>ステータス</td></tr>
</table>

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.9.7 レジスタ 6: フォルトアドレスレジスタ

このレジスタにより、表 B4-16 に示すように Opcode2 フィールドの値に従って、データおよびウォッチポイントフォルトステータスレジスタへアクセスできます。VMSAv6 以前は、統合 FAR のみが定義されていました。

表 B4-16 フォルトアドレスレジスタ

レジスタ名	Opcode2
統合 / データ FAR	0
ウォッチポイント FAR（WFAR）	1
命令 FAR（IFAR）: オプション	2

FAR、WFAR、IFAR はアボート時に、P. B4-22 表 B4-7 に従って更新されます。

—— 注 ——

WFAR の内容は*修飾仮想アドレス*（MVA）ではなく、仮想アドレスです。FAR と IFAR には、FCSE 機構が使用されている場合は、MVA で保存されています。詳細については P. B8-3 *「修飾仮想アドレス」*を参照して下さい。

WFAR 機能は、CP15 から CP14 のデバッグアーキテクチャに移行しており、CP15 による WFAR のデコードは ARMv6 では推奨されません。移行先については P. D3-2 *「コプロセッサ 14 のデバッグレジスタ」*を参照して下さい。

IFAR は VMSAv6 ではオプションで、PMSAv6 では必須です。このレジスタは、プリフェッチアボート時にのみ更新されます。

CP15 のレジスタ 6 を使用して、FAR、IFAR、WFAR へ値を書き込むことができます。これは、デバッガが値を復元するときに便利です。FAR が MCR 命令により書き込まれた場合、その値はデータとして扱われ、FCSE によるアドレスの修正は行われません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.9.8 レジスタ 8: TLB 機能

CP15 のレジスタ 8 は、TLB の制御に使用される書き込み専用レジスタです。表 B4-17 は、定義されている TLB 操作と、それらについて MCR 命令で使用される <CRm> と <opcode2> の値を示したものです。この表に指定されていない <CRm> と <opcode2> の値の組み合わせを使用した場合、結果は予測不能です。

更新された TLB の内容と前後の命令との同期に関する機能については、P. B2-22「*TLB 保守操作とメモリオーダモデル*」を参照して下さい。

修飾仮想アドレス（MVA）は、変換が行われる前に、非グローバルページの ASID と結合されます。P. B8-2「*FCSE の概要*」で説明されているように、FCSE を非グローバルページとともに使用した場合の動作は予測不能です。

MRC 命令で CP15 のレジスタ 8 の読み出しを試みた場合の結果は予測不能です。

表 B4-17 TLB 機能

機能	データ	命令
統合 TLB、または命令 / データ TLB 全体を無効化する	SBZ	`MCR p15, 0, Rd, c8, c7, 0`
統合された単一のエントリを無効化する	MVA	`MCR p15, 0, Rd, c8, c7, 1`
統合 TLB 上の ASID が一致するエントリを無効化する	ASID	`MCR p15, 0, Rd, c8, c7, 2`
命令 TLB 全体を無効化する	SBZ	`MCR p15, 0, Rd, c8, c5, 0`
単一の命令エントリを無効化する	MVA	`MCR p15, 0, Rd, c8, c5, 1`
命令 TLB 上の ASID が一致するエントリを無効化する	ASID	`MCR p15, 0, Rd, c8, c5, 2`
データ TLB 全体を無効化する	SBZ	`MCR p15, 0, Rd, c8, c6, 0`
単一のデータエントリを無効化する	MVA	`MCR p15, 0, Rd, c8, c6, 1`
データ TLB 上の ASID が一致するエントリを無効化する	ASID	`MCR p15, 0, Rd, c8, c6, 2`

統合 TLB を持つ実装で、命令またはデータ TLB 操作が使用された場合、統合 TLB に対してその操作が実行されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

——— 注 ———

ある時点でロック解除されたエントリが TLB に保持されている保証はないため、TLB 全体を無効化する操作は、実装内において無効化しようとする動作自体を表すものと考えることができます。単一エントリや ASID による操作についても同様です。

TLB の無効化

TLB にある、ロック解除されているエントリがすべて無効化されます。TLB 保守操作の同期については、P. B2-22「*TLB 保守操作とメモリオーダモデル*」を参照して下さい。

単一のエントリの無効化

単一エントリの無効化は、メモリを再マッピングする前に、メモリ領域の一部を無効化するために使用できます。再マッピングされる各メモリ領域（セクション、VMSAv6 以前のタイニーページ、スモールページ、ラージページ）について、その領域に含まれる修飾仮想アドレスに対応する単一のエントリを無効化する必要があります。

この機能によって、指定された MVA と ASID に一致する TLB エントリ、または指定された MVA に一致するグローバル TLB エントリが無効化されます。この機能では、グローバル TLB エントリに対して ASID はチェックされません。

ASID 一致による無効化

これは単一の割り込み可能な操作で、指定の ASID に一致する非グローバルページの TLB エントリをすべて無効化します。

実装でセットアソシエイティブ TLB が使用されている場合、この操作を完了するには多くのサイクル数を必要とすることがあり、命令が割り込まれることがあります。割り込みが発生した場合、R14 のステートは `MCR` 命令が実行されていないことを示します。このため、R14 は `MCR` + 4 のアドレスを指しており、割り込まれたルーチンは `MCR` 命令から自動的に再開されます。

統合 TLB を持つ実装で、命令またはデータ TLB 操作が使用された場合、統合 TLB に対してその操作が実行されます。

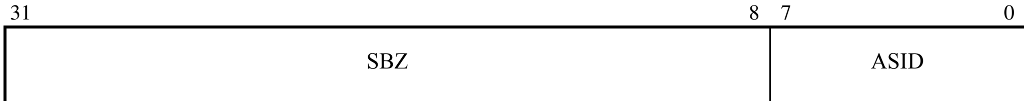
この操作中に割り込みが発生して、後で再開された場合、割り込みルーチン内で TLB へフェッチされたエントリで、かつ指定された ASID を持つエントリが、再開された無効化操作により無効化されるかどうかは予測不能です。

単一エントリの無効化操作には、引数として修飾仮想アドレスが必要です。渡される修飾仮想アドレスのフォーマットは、XP 制御ビットがセットされているかどうかにより異なります。XP 制御ビットが 0 の場合、修飾仮想アドレスのフォーマットは単純な 32 ビット MVA で、ビット [11:0] は無視されます。XP 制御ビットが 1 の場合、修飾仮想アドレスのフォーマットは次のようになります。

31 - 12	11 - 8	7 - 0
MVA	IGN	ASID

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ASID 一致による無効化には、引数として ASID が必要です。このフォーマットは次のとおりです。



B4.9.9 レジスタ 10: TLB ロックダウン

一部の ARM メモリシステムにある TLB ロックダウンは、指定の変換テーブルウォークの結果を、以降の変換テーブルウォークの結果により上書きされないような形で TLB にロードする機能です。これは、CP15 のレジスタ 10 によってプログラムされます。

変換テーブルウォークには多くの時間が必要な場合があります。特に、低速な可能性があるメインメモリへのアクセスが関与する場合は、その可能性が大きくなります。リアルタイム割り込みハンドラでは、ハンドラ、そのアクセスするデータ、またはその両方のエントリが TLB 上にないために発生した変換テーブルウォークにより、割り込みのレイテンシが大きく増大する可能性があります。

ARM アーキテクチャでは 2 つの基本的なロックダウンモデルがサポートされています。

- エントリによる TLB ロックモデル
- 変換とロックモデル

VMSAv6 以降は、TLB タイププレジスタを使用して、統合 TLB とハーバード TLB のどちらが実装されているか、および利用可能なロック可能 TLB エントリ数を確認できます。詳細については、P. B3-11「*TLB タイプレジスタ*」を参照して下さい。ARMv6 では、エントリによるロックモデルのみがサポートされています。VMSAv6 以前では、使用される TLB ロック機構は実装定義です。

各種の機構をサポートするために使用される TLB 操作を表 B4-18 に示します。

表 B4-18

機能	**CRm**	**Opc_2**	**命令**	**ロックモデル**
データ（または統合）ロックダウンレジスタ	c0	0	`MCR p15,0,Rd,c10,c0,0` `MRC p15,0,Rd,c10,c0,0`	明示的
命令ロックダウンレジスタ	c0	1	`MCR p15,0,Rd,c10, c0,1` `MRC p15,0,Rd,c10,c0,1`	明示的
命令 TLB エントリの変換とロック	c4	0	`MCR p15,0,Rd,c10,c4,0`	変換とロック
命令 TLB のロック解除	c4	1	`MCR p15,0,Rd,c10,c4,1`	変換とロック
データ TLB エントリの変換とロック	c8	0	`MCR p15,0,Rd,c10,c4,0`	変換とロック
データ TLB のロック解除	c8	1	`MCR p15,0,Rd,c10,c4,1`	変換とロック

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

W が TLB エントリ数の 2 を底とする対数で、必要に応じて整数に切り上げられる場合、CP15 のレジスタ 10 の形式は次のようになります。

31 … 32-W	31-W … 32-2W	31-2W … 1	0
base	victim	UNP/SBZP	P

実装で命令とデータに独立した TLB を使用している場合、このレジスタには 2 つのバリエーションがあり、レジスタ 10 へのアクセスに使用される MCR 命令か MRC 命令の <opcode2> フィールドにより選択されます。

<opcode2> == 0　データ TLB ロックダウンレジスタを選択します。

<opcode2> == 1　命令 TLB ロックダウンレジスタを選択します。

実装で統合 TLB を使用している場合、このレジスタのバリエーションは 1 つだけで、<opcode2> は常に 0 です。

レジスタ 10 にアクセスする MCR 命令と MRC 命令では、<CRm> は常に c0 です。

レジスタ 10 への書き込みは次のような効果があります。

- victim フィールドは、次の TLB ミスにより生成される変換テーブルウォークの結果として、どの TLB エントリが置き換えられるかを指定します。
- ベースフィールドにより、TLB の置換が（base）から（TLB エントリ数）- 1 までの番号を持つ TLB エントリだけを使用して行われるように制限することができます。victim フィールドはこの範囲の中に含まれます。
- P == 1 の間、変換テーブルウォークの結果として TLB に書き込まれたエントリは、レジスタ 8 により TLB 全体を無効化されることから保護されます。P == 0 のときに書き込まれたデータはこれらの操作により通常に無効化されます。

—— 注 ——

TLB エントリ数が 2 のべき乗でない場合、base または victim フィールドに TLB エントリ数以上の値を書き込んだ場合の結果は予測不能です。

——————

レジスタ 10 を読み出すと、base フィールドと P ビットに最後に書き込まれた値、および victim フィールドで次に置き換えられる TLB エントリ数が返されます。

エントリによる TLB ロックモデル

インクリメントされた victim フィールドは、base フィールドの値にラップアラウンドされます。

アーキテクチャでは、この修飾形式として、base フィールドが 0 に固定された形式が許容されます。この形式は、実装で汎用 TLB 機構と独立したリソースとして、専用のロック可能なエントリ（統合またはハーバード形式）を提供する場合に特に適切です。どの形式のロックモデルが使用されているかを判定するには、すべてのビットが 1 である値を base フィールドに書き込み、再度読み出して、その値が 1 であるかどうかをチェックします。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

変換とロックモデル

この機構は、明示的な TLB 操作を使用して、指定のアドレスを TLB に変換し、ロックします。エントリは、ロック解除操作を使用して、グローバルベースでロック解除されます。アドレスは、それぞれの*修飾仮想アドレス*（MVA）を使用してロードされます。詳細については、P. B8-3「*修飾仮想アドレス*」を参照して下さい。以下の操作は予測不能です。

- これらの機能に読み出し（MRC）コマンドでアクセスする
- MMU が無効の状態で機能を使用する
- 既に TLB に存在しているアドレスに対して変換とロックを試みる

変換時のデータアボートはすべて、ロックアボートとして報告されます。詳細については P. B4-20 表 B4-6 を参照して下さい。検出されるのは、外部アボートまたは変換アボートのみです。これらの機能に関するアクセス許可、ドメイン、アライメントチェックはすべて実装定義です。アボートを生成する操作は、対象となる TLB には影響しません。

このモデルを統合 TLB に適用する場合、データ側の操作を使用する必要があります。

Invalidate_all（I、D、または I と D）操作は、ロックされたエントリには効果がありません。このモデルでは、ASID またはエントリによる無効化は実装定義です。

エントリモデルによる TLB ロックモデルプロシージャ

ベースフィールドが変更可能な場合、N 個の TLB エントリをロックダウンする通常のプロシージャは次のようになります。

1. 割り込み禁止などの操作を行い、このプロシージャの実行中にプロセッサ例外が発生しないことを保証します。
2. 命令 TLB または統合 TLB をロックダウンする場合、base == N、victim == N、P == 0 をレジスタ 10 の適切なバージョンに書き込みます。必要に応じて、分岐予測など命令プリフェッチ動作を予測しにくくする機構も停止します。
3. ロックダウンの対象となる TLB 全体を無効化します。
4. 命令 TLB をロックダウンする場合、以降のロックダウンプロシージャによりプリフェッチされる可能性のある命令に関連する TLB エントリがすべてロードされていることを確認します。ロックダウンプロシージャの開始する場所について注意が払われていれば、通常は 1 つの TLB エントリでその全体をカバーできます。その場合、その TLB が無効化された後で最初の命令プリフェッチでこの処理を実行できます。

   データ TLB をロックダウンする場合、以降のロックダウンプロシージャによりアクセスされる可能性のあるデータに関連する TLB エントリが、コードで使用されるインラインリテラルも含めてすべてロードされていることを確認します。このためには通常、ロックダウンプロシージャでインラインリテラルの使用を避け、使用される他のデータをすべて単一の TLB エントリでカバーされる領域に置き、領域内の適当なデータをロードするのが最良の方法です。

   統合 TLB をロックダウンする場合、上の操作の両方を実行します。
5. 0 ～ N-1 の各 i について、
   a. レジスタ 10 に base == i、victim == i、P == 1 を書き込みます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

   b. 次の操作を行い、変換テーブルウォークの結果が TLB エントリ i にロックされるメモリ領域について、強制的に変換テーブルウォークを発生させます。
      - データ TLB または統合 TLB をロックダウンする場合、メモリの領域から適当なデータをロードします。
      - 命令 TLB をロックダウンする場合、P. B6-19「*レジスタ 7: キャッシュ管理機能*」で定義されているレジスタ 7 の*プリフェッチ命令*キャッシュ*ライン*操作を使用して、命令がメモリ領域からプリフェッチされるようにします。

6. レジスタ 10 に base == N、victim == N、P == 0 を書き込みます。

——— 注 ———

高速コンテキストスイッチ拡張機能（FCSE）（第 B8 章参照）を使用している場合、次の理由から手順 5b に注意が必要です。

- データ TLB または統合 TLB をロックダウンする場合、ロード命令で使用するアドレスは FCSE による変更の対象となります。
- 命令 TLB をロックダウンする場合、レジスタ 7 操作で使用するアドレスはデータと見なされ、FCSE による変更の対象となりません。

これによって発生する混乱の可能性を最小にするため、ロックダウンプロシージャは次のように行うことをお勧めします。

- 最初に FCSE を禁止します（PID を 0 に設定します）。
- 場合によって、適切な PID の値と、その使用する仮想アドレスの上位 7 ビットの論理和を取り、修飾仮想アドレス自体を生成します。

———————————

ベースフィールドが 0 に固定されている場合、アルゴリズムを次のように簡略化できます。

1. 割り込み禁止などの操作を行い、このプロシージャの実行中にプロセッサ例外が発生しないことを保証します。

2. 現在ロックされているエントリを削除する必要がある場合、対応する単一エントリ無効化、または ASID による無効化操作を行う必要があります。

3. 分岐予測を停止します。

4. 命令 TLB をロックダウンする場合、以降のロックダウンプロシージャによりプリフェッチされる可能性のある命令に関連する TLB エントリがすべてロードされていることを確認します。ロックダウンプロシージャの開始する場所について注意が払われていれば、通常は 1 つの TLB エントリでその全体をカバーできます。その場合、その TLB が無効化された後で最初の命令プリフェッチでこの処理を実行できます。

   データ TLB をロックダウンする場合、以降のロックダウンプロシージャによりアクセスされる可能性のあるデータに関連する TLB エントリが、コードで使用されるインラインリテラルも含めてすべてロードされていることを確認します。このためには通常、ロックダウンプロシージャでインラインリテラルの使用を避け、使用される他のデータをすべて単一の TLB エントリでカバーされる領域に置き、1 つのデータ項目をロードするのが最良の方法です。

   統合 TLB をロックダウンする場合、上の操作の両方を実行します。

5. 0 ～ N - 1 の各 i について
   a. レジスタ 10 に base == 0、victim == i、P == 1 を書き込みます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

b. 次の操作を行い、変換テーブルウォークの結果が TLB エントリ i にロックされるメモリ領域について、強制的に変換テーブルウォークを発生させます。
   - データ TLB または統合 TLB をロックダウンする場合、メモリの領域から適当なデータをロードします。
   - 命令 TLB をロックダウンする場合、P. B6-19 *「レジスタ 7: キャッシュ管理機能」*で定義されているレジスタ 7 の*プリフェッチ命令キャッシュライン*操作を使用して、命令がメモリ領域からプリフェッチされるようにします。

6. 適切なロックダウンレジスタをクリアします。

変換とロックモデルによる TLB ロックダウンプロシージャ

以前にロックされたエントリはすべて、適切なロック解除操作を命令サイド、またはデータサイドに発行することで解除できます。その後で、レジスタ Rd に対象となる MVA をセットして、明示的なロックダウン操作を発行することができます。

エントリモデルによる TLB ロック解除プロシージャ

上のプロシージャを使用して TLB がロックダウンされた後で､ロックダウンされた TLB の一部をロック解除するには、次のような操作を行います。

1. レジスタ 8 操作を使用して、ロックダウンされた各エントリを 1 つづつ無効化します。
2. レジスタ 10 に base == 0、victim == i、P == 0 を書き込みます。

——— 注 ———

手順 1 により､P == 1 のエントリが TLB に残っていないことを保証します。これらのエントリが TLB に残っている場合、以降の TLB ロックダウンプロシージャで行われる TLB 全体の無効化手順（手順 3）は期待している結果が得られない場合があります。

変換とロックモデルによる TLB ロック解除プロシージャ

適切な（I または D）TLB ロック解除操作を発行すると、ロックされたエントリがすべて解除されます。単一のエントリの無効化または ASID による無効化によりロック状態が解除されるかどうかは実装定義です。

——— 注 ———

エントリモデルによるロックでは、単一または ASID による無効化動作が保証されている点が異なります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B4.9.10 レジスタ 13: プロセス ID

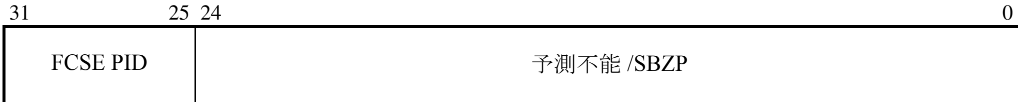
このレジスタは、現在実行されているプロセスを示しています。opcode2 フィールドの値によって、2 つのレジスタにアクセスできます。詳細については、表 B4-19 を参照して下さい。ASID の更新時には、現在の命令とデータストリームは ASID に依存したメモリ領域ではなく、グローバルの必要があります。リセット時には、FCSE PID レジスタは*常に 0* で、コンテキスト ID レジスタの値は未定義です。

表 B4-19 プロセス ID レジスタ

レジスタ名	Opcode2
FCSE PID	0
コンテキスト ID	1

FCSE PID

高速コンテキストスイッチ拡張機能（FCSE）を制御します。FCSE の使用は推奨されません。



コンテキスト ID

このレジスタの下位 8 ビットは、現在実行されている ASID です。上位ビットは、ASID を汎用プロセス ID に拡張します。

実装により、システムでこの値を利用することができます。すべてのアクセスが現在のコンテキスト ID に関連していることを保証するため、このレジスタを変更する前にはソフトウェアでデータ同期バリア操作を実行する必要があります。

このレジスタ全体は、*組み込みトレースマクロセル*（ETM）とデバッグロジックの両方により使用されます。この値は ETM によりトレースユニットに伝えられ、現在実行されているプロセスを示すために使用できます。このためには、この値は各プロセスで固有の値がプログラムから設定される必要があります。その結果、ETM は ASID をプロセスを識別するために使用することができます。この値は ETM により、仮想メモリから物理メモリへのマッピングを決定するため使用されます。

また、プロセスに依存したブレークポイントと命令を有効にするためにも使用可能です。

コンテキスト ID レジスタの変更の同期については、P. B2-24「*CP15 のレジスタの変更とメモリオーダモデル*」を参照して下さい。

31 ... 8	7 ... 0
PROCID	ASID

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 B5 章
保護メモリシステムアーキテクチャ

本章では、*メモリ保護ユニット*（MPU）をベースとした*保護メモリシステムアーキテクチャ*（PMSA）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。

- *PMSA の概要* : P. B5-2
- *メモリアクセスのシーケンス* : P. B5-4
- *メモリアクセスの制御* : P. B5-8
- *メモリアクセスの属性* : P. B5-10
- *メモリアボート（PMSAv6）* : P. B5-13
- *フォルトステータスレジスタとフォルトアドレスレジスタのサポート* : P. B5-16
- *CP15 のレジスタ* : P. B5-18

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.1 PMSA の概要

MPU ベースの*保護メモリシステムアーキテクチャ*（PMSA）は、第 B4 章*「仮想メモリシステムアーキテクチャ」*で説明されている MMU ベースのモデルと比較して、大幅に単純化されたメモリ保護スキーマです。ハードウェアとソフトウェアの両方が単純化されます。

主要な要素は、MPU は変換テーブルを使用しないことです。代わりに、システム制御プロセッサ（CP15）レジスタを使用して*保護領域*が完全に定義されるため、ハードウェアによる変換テーブルウォークと、ソフトウェアによる変換テーブルのセットアップと保守の必要がなくなります。これには、メモリチェックにかかる時間が完全に予測可能になるという利点があります。ただし、制御のレベルはページベースではなく領域ベースとなるため、小さな領域での制御は行われなくなります。

もう一つの単純化は、仮想アドレスから物理アドレスへのマッピングがサポートされていないことです。物理メモリアドレスは常に、ARM® プロセッサで生成される仮想アドレスと同一です。以下の機能はすべての PMSA 設計に共通です。

- メモリは領域に分割されます。システムコプロセッサレジスタは、領域サイズ、ベースアドレス、および領域に対するキャッシュ可能、バッファ可能、アクセス許可などのメモリ属性の定義に使用されます。
- メモリ領域制御（読み出しと書き込みのアクセス）は特権モードからのみ許可されます。
- アドレスが複数の領域で定義されている場合、固定優先度スキーマ（最も高い領域番号）を使用して、アクセスされるアドレスのプロパティが定義されます。
- どの領域にも定義されていないアドレスにアクセスした場合、メモリアボートが発生します。
- すべてのアドレスは物理アドレスで、アドレス変換はサポートされていません。
- 命令とデータのアドレス空間が統合されたフォンノイマン形式と、独立したハーバード形式の両方がサポートされています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.1.1 VMSAv6 で導入された主要な変更

PMSA は、ARMv6 で定義された新しいメモリ属性をサポートするため、レビューと更新が行われています。更新されたバージョンは PMSAv6 と呼ばれ、VMSAv6 と対になっています。VMSAv6 の詳細については、第 B4 章*「仮想メモリシステムアーキテクチャ」*を参照して下さい。PMSAv6 で導入された変更は次のとおりです。

- サポートされる領域の数は 8 つに固定されるという制限がなくなりました。サポートされる領域の数と、関連する CP15 レジスタにアクセスする方法は、*密結合メモリ*（TCM）のサポート方法とほぼ同じです。詳細については、第 B7 章*「密結合メモリ」*を参照して下さい。

  —— 注 ——

  PMSAv6 のプログラミングモデルは、以前のバージョンと下位互換ではありません。

  ————

- メモリ属性とアクセス許可は、ARMv6 で定義された新しい機能をサポートするため拡張されています。アクセス許可が拡張され、特権モードの読み出し専用と、特権モード / ユーザモードの読み出し専用モードが同時にサポート可能になりました。
- アボート機構は、CP15 で定義されているフォルトステータスレジスタとフォルトアドレスレジスタを使用して、アボートの原因と、フォルトの発生したアドレスを報告します。ARMv6 以前は、PMSA で発生したアボートは破壊的と見なされ、アーキテクチャには復元機構は存在していませんでした。

—— 注 ——

詳細について他章への参照が含まれていますが、PMSA の説明に関しては仮想アドレスまたは修飾仮想アドレスに関する記載はすべて無視し、物理アドレスとして扱う必要があります。

————

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.2 メモリアクセスのシーケンス

ARM CPU がメモリアクセスを生成するとき、MPU はメモリアドレスをプログラムされたメモリ領域と比較します。

- 一致するメモリ領域が見つからない場合、メモリアボート信号がプロセッサに送られます。
- 一致するメモリ領域が見つかった場合、領域の情報は次のように使用されます。
    1. アクセス許可ビットを使用して、アクセスが許可されているかどうかを判定します。アクセスが許可されていない場合、MPU がメモリアボート信号を発生します。それ以外の場合、アクセスが許可されます。アクセス許可ビットの説明については、P. B5-8 *「メモリアクセスの制御」*を参照して下さい。
    2. メモリ領域属性は、アクセス属性、たとえばキャッシュ可能 / キャッシュ不可の判定に使用されます。詳細については、P. B5-10 *「メモリアクセスの属性」*を参照して下さい。

図 B5-1 は、キャッシュが使用されているシステムのメモリアクセスシーケンスです。

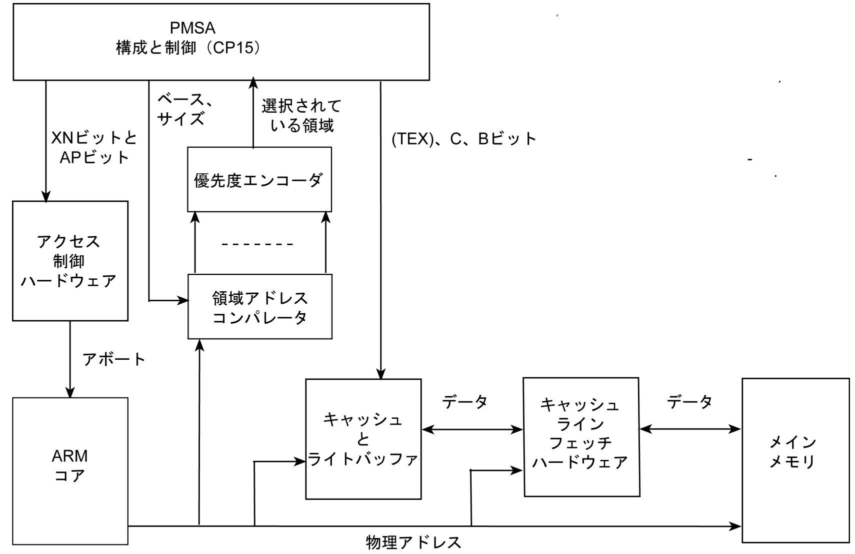


図 B5-1 キャッシュを使用している保護ユニットメモリシステムの概要

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.2.1 オーバラップしている領域

保護ユニットは、2 つ以上のオーバラップする領域を持つようにプログラムすることも可能です。オーバラップする領域がプログラムされている場合、メモリアクセスにどの領域の属性が適用されるかは、固定優先度スキーマによって判定されます。

領域 7 の属性が最優先され、領域 0 の属性は最も優先度が低くなります。次に例を示します。

- データ領域 2 がアドレス `0x3000` から始まり、サイズが 4KB で、AP == 0b010 にプログラムされている場合（特権モードで完全アクセス、ユーザモードで読み出し専用）
- データ領域 1 がアドレス `0x0` から始まり、サイズが 16KB で、AP == 0b001 にプログラムされている場合（特権モードアクセス専用）

プロセッサがユーザモードでアドレス `0x3010` からデータロードを実行すると、図 B5-2 に示すように、このアドレスは領域 1 と領域 2 の両方に属することになります。衝突が発生するため、領域 2 に設定された属性が適用されます。この場合、ロードでアボートは発生しません。

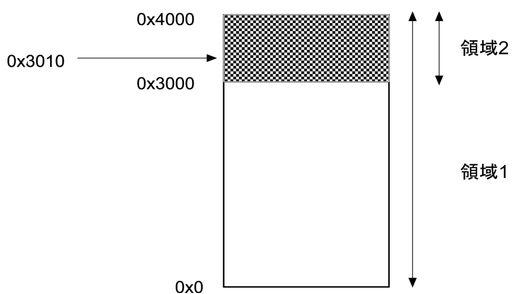


図 B5-2 オーバラップしているメモリ領域

B5.2.2 バックグラウンド領域

領域のオーバラップにより、領域をシステムの物理メモリデバイスにマップする方法がより柔軟になります。オーバラップのプロパティは、バックグラウンド領域の指定にも使用できます。たとえば、多数の物理メモリ領域が、4GB のアドレス空間にわたってまばらに分散されている事例を考えます。これらの領域のみを構成した場合、定義されたまばらなアドレス空間の外部にアクセスを行うとアボートが発生します。領域 0 を 4GB のバックグラウンド領域としてプログラムすると、この動作をオーバライドできます。この事例では、アドレスが他の領域のいずれにも含まれない場合、アクセスは領域 0 に指定された属性により制御されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.2.3 MPU の許可と禁止

MMU は、CP15 のレジスタ 1 の M ビット（ビット [0]）に書き込むことで許可または禁止できます。リセット時にはこのビットは 0 でクリアされ、MPU は禁止されます。

MPU を許可する前に、関連するすべての CP15 レジスタをプログラムする必要があります。これには、最低 1 つのメモリ領域のセットアップが含まれます。MPU を有効にする前に、次の操作を行う必要があります。

- 命令キャッシュを禁止し、無効化します。
- データキャッシュを禁止し、クリーニングし、無効化します。

CP15 レジスタの変更を同期する方法については、P. B2-24「*CP15 のレジスタの変更とメモリオーダモデル*」を参照して下さい。この方法は、MPU、キャッシュ、または両方を許可 / 禁止する変更に適用されます。

MPU が禁止されている場合の動作

ARMv6 以前は、MPU が禁止されている場合、すべてのメモリ領域はキャッシュ不可、バッファ不可、アボート不可として扱われていました。

PMSAv6 では、MPU が禁止されている場合、メモリアクセスは次のように処理されます。

- メモリアクセス許可のチェックは行われず、MPU でアボートは発生しません。
- データアクセスは、P. B5-7 表 B5-1 に示すデフォルトのメモリマップを使用します。下位 2GB メモリへのデータアクセスは、CP15 のレジスタ 1 の C ビット（ビット 2）がセットされ、データ（または統合）キャッシュが許可されている場合、キャッシュ可能として扱われます。上位 2GB へのデータアクセスは、キャッシュ不可として扱われます。
- ハーバードキャッシュ構成が使用されている場合、メモリの下位 2GB への命令アクセスはすべて通常、共有不可、キャッシュ可能メモリとして扱われます。命令アクセスは、CP15 のレジスタ 1 の I ビット（ビット [12]）がセット (1) されている場合はキャッシュ可能、I ビットがクリア (0) されている場合はキャッシュ不可です。上位 2GB メモリへの命令アクセスは、キャッシュ不可として扱われます。

  統合キャッシュが使用されている場合、下位 2GB メモリへの命令アクセスは、CP15 のレジスタ 1 の C ビット（ビット [2]）がセットされていれば、すべてキャッシュ可能として扱われます。上位 2GB へのアクセスは、キャッシュ不可として扱われます。
- プログラムフロー予測は通常に機能し、CP15 のレジスタ 1 の Z ビット（ビット [11]）のステートにより制御されます。
- MPU とキャッシュの CP15 操作はすべて、MPU が禁止されている場合も通常に動作します。
- 命令とデータのプリフェッチ操作は通常に動作します。データキャッシュが禁止されている場合、データプリフェッチ操作は無効です。命令キャッシュが禁止されている場合、命令プリフェッチ操作は無効です。
- TCM が許可されている場合、TCM へのアクセスは通常に動作します。
- 外部（またはレベル 2）メモリの属性は、レベル 1 メモリシステムの属性と同じです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B5-1 デフォルトメモリマップ

アドレス範囲	ICache 許可の場合の命令メモリタイプ	ICache 禁止の場合の命令メモリタイプ	DCache 許可の場合のデータメモリタイプ	DCache 禁止の場合のデータメモリタイプ
0xFFFFFFFF 0xC0000000	通常、 キャッシュ不可	通常、 キャッシュ不可	ストロングオーダ	ストロングオーダ
0xBFFFFFFF 0xA0000000	通常、 キャッシュ不可	通常、 キャッシュ不可	共有デバイス	共有デバイス
0x9FFFFFFF 0x80000000	通常、 キャッシュ不可	通常、 キャッシュ不可	非共有デバイス	非共有デバイス
0x7FFFFFFF 0x60000000	通常、 ライトスルーキャッシュ可能、非共有	通常、 キャッシュ不可、 非共有	通常、 キャッシュ不可、 共有	通常、 キャッシュ不可、 共有
0x5FFFFFFF 0x40000000	通常、 ライトスルーキャッシュ可能、非共有	通常、 キャッシュ不可、 非共有	通常、 ライトスルーキャッシュ可能、非共有	通常、 キャッシュ不可、 共有
0x3FFFFFFF 0x00000000	通常、 ライトスルーキャッシュ可能、非共有	通常、 キャッシュ不可、 非共有	通常、 WBWA キャッシュ可能、非共有	通常、 キャッシュ不可、 共有

——— 注 ———

密結合メモリ（TCM）が実装されている場合、これらのメモリはアドレス空間内で許可されているかどうかにかかわらず、キャッシュ不可、共有不可、通常メモリとしてマップされます。

MPU を含まない実装の動作

実装に MPU が含まれていない場合、前のセクションで概説されているデフォルトのメモリマップをオプションで採用できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.3 メモリアクセスの制御

メモリ領域へのアクセスは、MPU にプログラムされているアクセス許可ビットにより制御されます。ARMv6 以前では、サポートされている 8 つの領域のアクセス許可ビットは、CP15 のレジスタ 5 のみによってプログラムされます。詳細については、P. B5-23*「レジスタ 5: アクセス許可ビット(PMSAv6 以前)」*を参照して下さい。PMSAv6 では、アクセス許可とメモリ属性は、CP15 のレジスタ 6 で決定され、領域により指定されます。詳細については、P. B5-25*「レジスタ 6: メモリ領域プログラミング(PMSAv6 以前)」*を参照して下さい。

B5.3.1 データと命令のアクセス許可(PMSAv6 以前)

アクセス許可は、各領域について 2 ビットのフィールドで定義されます。AP ビットの組み合わせの解釈を表 B5-2 に示します。要求されたアクセスタイプが許可されていない場合、アボート信号が ARM プロセッサに送られます。

表 B5-2 MPU アクセス許可(PMSAv6 以前)

AP	特権アクセス許可	ユーザアクセス許可
0b00	アクセス不可	アクセス不可
0b01	読み出し / 書き込み	アクセス不可
0b10	読み出し / 書き込み	読み出し専用
0b11	読み出し / 書き込み	読み出し / 書き込み

—— 注 ——

AP ビットの解釈は、CP15 のレジスタ 1 のシステム (S) および ROM (R) ビットによって変更されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.3.2 データアクセス許可（PMSAv6）

PMSAv6 では、アクセス許可ビットは 3 ビットフィールドに拡張されており、対応するメモリ領域へのアクセスを制御します。必要なアクセス許可なしにメモリ領域に対してアクセスが行われた場合、アクセス許可フォルトが発生します。アクセス許可は、データアクセス許可レジスタの AP ビットにより決定されます。

表 B5-3 データアクセス許可（PMSAv6）

AP[2:0]	特権アクセス許可	ユーザアクセス許可	説明
000	アクセス不可	アクセス不可	すべてのアクセスについて、アクセス許可フォルトが発生します。
001	読み出し / 書き込み	アクセス不可	特権アクセスのみ許可されます。
010	読み出し / 書き込み	読み出し専用	ユーザモードで書き込みを行うと、アクセス許可フォルトが発生します。
011	読み出し / 書き込み	読み出し / 書き込み	完全アクセス。
100	予測不能	予測不能	予約
101	読み出し専用	アクセス不可	特権読み出しのみ許可されます。
110	読み出し専用	読み出し専用	特権アクセスと、ユーザモードでの読み出しのみ許可されます。
111	予測不能	予測不能	予約

B5.3.3 命令アクセス許可（PMSAv6）

命令アクセスのために、独立したアクセス許可機構がサポートされています。これによって、データアクセスに影響することなくメモリ領域を実行不可、つまりデータ専用にマークできます。命令を実行するメモリ領域は、データ読み出しアクセスが許可され（AP[2:0] で示されます）、XN ビットが 0 の必要があります。

表 B5-4 命令アクセス許可

XN	説明
0	すべての命令フェッチが許可されます。
1	すべての命令フェッチは禁止されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.4 メモリアクセスの属性

ARMv6 以前は、メモリ属性はキャッシュ可能性とバッファ可能性のビットのみでした。キャッシュビットは、許可される 8 つの領域ごとに 1 つ存在し、CP15 のレジスタ 2 によって定義されます。詳細については P. B5-22 *「レジスタ 2: キャッシュ可能ビット（PMSAv6 以前）」*を参照して下さい。対応するバッファビットは CP15 のレジスタ 3 で定義されています。詳細については、P. B5-22 *「レジスタ 3: バッファ可能ビット（PMSAv6 以前）」*を参照して下さい。

各メモリ領域には、関連する一連のメモリ領域属性があります。これらの属性により、キャッシュへのアクセス、ライトバッファの使用方法、メモリ領域が共有可能かどうか、そのためにコヒーレンシを維持する必要があるかどうかが制御されます。

B5.4.1 CB + TEX のエンコード（ARMv6 以降）

メモリ属性レジスタは、メモリ領域のタイプのエンコードに 5 つのビットを使用します。TEX[2:0]、C、B ビットです。P. B5-11 表 B5-5 は、タイプ拡張子フィールド（TEX）のマッピングと、キャッシュ可能およびバッファ可能ビット（C ビットと B ビット）からメモリ領域タイプへのマッピングを示したものです。

また、メモリ属性レジスタには共有ビット (S) が含まれています。このビットは、メモリが複数のプロセッサ間で共有可能かどうかを示します。このビットは通常メモリにのみ適用され、デバイスやストロングオーダメモリには適用されず、メモリ領域が共有か (1) 非共有か (0) を示します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

共有可能属性と、通常、ストロングオーダ、デバイスの各メモリタイプの説明については、P. B2-8 *「ARMv6 のメモリ属性の概要」*を参照して下さい。

表 B5-5 CB + TEX のエンコード（ARMv6 以降）

TEX	C	B	説明	メモリタイプ	領域が共有可能か
000	0	0	ストロングオーダ	ストロングオーダ	共有可能
000	0	1	共有デバイス	デバイス	共有可能
000	1	0	外部と内部のライトスルー、書き込み割り当てなし	通常	s
000	1	1	外部と内部のライトバック、書き込み割り当てなし	通常	s
001	0	0	外部と内部でキャッシュ不可	通常	s
001	0	1	予約	予約	予約
001	1	0	実装定義	実装定義	実装定義
001	1	1	外部と内部のライトバック、書き込み割り当てあり	通常	s
010	0	0	非共有デバイス	デバイス	非共有
010	0	1	予約	予約	予約
010	1	X	予約	予約	予約
1BB	A	A	キャッシュされたメモリ BB = 外部ポリシー、AA = 内部ポリシー	通常	s

ここで、s はメモリ属性レジスタの S ビットの値です。

内部と外部という用語は、システムに搭載されているキャッシュのレベルを指します。内部とは、レベル 1 キャッシュを含む最も内側のキャッシュを指します。外部とは、最も外側のキャッシュを指します。内部キャッシュと外部キャッシュの境界は、キャッシュを持つシステムの実装により定義されます。内部キャッシュには、常にレベル 1 キャッシュが含まれます。たとえば、3 レベルのキャッシュを持つシステムの場合、レベル 1 とレベル 2 に内部属性が、レベル 3 に外部属性が適用されることがあります。キャッシュが 2 レベルのシステムの場合、レベル 1 が内部、レベル 2 が外部と見なされます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B5-6 キャッシュポリシーのエンコード

メモリ属性のエンコード	キャッシュポリシー
00	キャッシュ不可
01	ライトバック、書き込み割り当てあり
10	ライトスルー、書き込み割り当てなし
11	ライトバック、書き込み割り当てなし

――― 注 ―――

どの書き込み割り当てポリシーをサポートするかは、実装で決定できます。*書き込み時割り当て*と*非書き込み時割り当て*キャッシュポリシーは、メモリ領域について優先される割り当てポリシーを示しますが、メモリシステムがそのポリシーを実装していることを前提にはできません。すべての内部と外部のキャッシュポリシーが必須ではありません。

表 B5-7 実装オプション

キャッシュポリシー	実装オプション
内部キャッシュ不可	必須
内部ライトスルー	必須
内部ライトバック	オプション。サポートされていない場合、メモリシステムに内部ライトスルーを実装する必要があります。
外部キャッシュ不可	必須
外部ライトスルー	オプション。サポートされていない場合、メモリシステムは外部キャッシュ不可として実装する必要があります。
外部ライトバック	オプション。サポートされていない場合、メモリシステムに外部ライトスルーを実装する必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.5 メモリアボート（PMSAv6）

メモリアクセスにより、ARM プロセッサに例外が発生する機構は次のとおりです。

MPU フォルト MPU が制限を検出し、プロセッサに信号を送ります。

デバッグアボート モニタデバッグモードが有効になり、ブレークポイントまたはウォッチポイントが検出されます。

外部アボート 外部メモリシステムから、無効な、またはフォルトの発生したメモリアクセスが報告されます。

これらを*アボート*と総称します。アボートを発生するアクセスは、*アボートされた*と呼びます。

アボートされたメモリ要求がアボートフェッチの場合、そのアボートされたアクセスに対応する命令の実行がプロセッサにより試みられたときにプリフェッチアボート例外が発生します。アボートされたアクセスがデータアクセスかキャッシュ保守操作の場合、データアボート例外が発生します。

すべてのデータアボートは*データフォルトステータスレジスタ*（DFSR）を更新するため、アボートの原因を判定できます。すべての命令アボートは*命令フォルトステータスレジスタ*（IFSR）を更新するため、アボートの原因を判定できます。

外部アボートを除くすべてのデータアボートについて、*フォルトアドレスレジスタ*（FAR）にはアボートが発生したアドレスが書き込まれます。外部データアボートはすべて不正確な可能性があるため、FAR の内容はアボートの発生したアドレスではありません。

命令アボートの場合、*命令フォルトアドレスレジスタ*（IFAR）にはアボートが発生したアドレスが書き込まれます。アボートハンドラはこのレジスタを使用し、アボートの発生したアドレスを判定できます。IFAR に記憶される正確な値については、P. B5-16*「フォルトステータスレジスタとフォルトアドレスレジスタのサポート」*を参照して下さい。

ウォッチポイントフォルトアドレスレジスタ（WFAR）には、ウォッチポイントがアクセスされたときの命令のアドレスが書き込まれます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.5.1 MPU フォルト

MPU は 3 種類のフォルトを生成します。

- アライメントフォルト
- バックグラウンドフォルト
- アクセス許可フォルト

MPU により検出されたアボートは、そのアボートが検出されたアドレスへの外部アクセスを発生しません。

アライメントフォルト

厳密アライメントチェックのサポートは、CP15 のレジスタ 1 の A ビットと U ビットにより制御されます。詳細については、P. B5-21 *「レジスタ 1: 制御レジスタ」*を参照して下さい。ARMv6 メモリアーキテクチャでは、この機能は必須です。この機能により、オペレーティングシステムがアンアラインドデータアクセスをトラップできることが保証されます。ARMv6 以前はこのサポートはオプションでした。このフォルトは、低位アドレスがデータアクセスの幅に境界アラインしていないとデータアボート例外に入ります。

ダブルワードのロードとストア（LDRD、STRD）に関するアライメントフォルトが強化されています。

- U == 0 の場合、偶数ワードアドレスに境界アラインされていないアクセスがトラップされます（アドレスビット [2:0] != 0）。
- U == 1 の場合、ワード境界にアラインされていないアクセスがトラップされます（アドレスビット [1:0] != 0）。

バックグラウンドフォルト

メモリアクセスのアドレスが、プログラムされているメモリ領域の 1 つに一致しない場合、バックグラウンドフォルトが生成されます。

アクセス許可フォルト

アクセス許可は、P. B5-8 *「メモリアクセスの制御」*で定義されているように、プロセッサのメモリアクセスに対してチェックされます。アクセスが許可されていない場合、アボート信号がプロセッサに送られます。

B5.5.2 デバッグアボート

モニタデバッグモードが有効な場合、命令アクセスのブレークポイント、またはデータアクセスのウォッチポイントによりアボートが発生する可能性があります。どちらの場合も、アボートの発生前にメモリシステムによりアクセスが完了されます。モニタデバッグモードのためアボートが発生した場合、デバッグアボートを示すために、対応する FSR（命令またはデータ）が更新されます。

ウォッチポイントでアボートが発生した場合、WFAR にはウォッチポイントを発生したアドレスが設定されます。ウォッチポイントは、次の命令がロードおよびストア命令と同時に実行されている可能性があることから、正確に取得されません。デバッグイベントを発生した命令を判定するには、デバッガで WFAR を読み出す必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.5.3 外部アボート

外部メモリエラーは、MPU で検出される以外のメモリシステムで発生したエラーです。外部メモリエラーは発生する頻度が極めて低いと想定されており、実行中のプロセスに対して多くの場合致命的エラーとなります。外部メモリエラーを引き起こすイベントの例としては、レベル 2 メモリ構造での修正不能なパリティまたは ECC エラーがあります。

外部アボートの存在は、データまたは命令フォルトステータスレジスタにより示されます。データアボートは、正確な場合と不正確な場合があり、タイプは DFSR のエンコードされた値により識別されます。詳細については、P. B5-16 *「フォルトステータスレジスタとフォルトアドレスレジスタのサポート」*を参照して下さい。

命令フェッチ時の外部アボート

命令プリフェッチ時に外部で生成されたエラーは本質的に正確で、エラーを発生したロケーションからフェッチされた命令を CPU が実行しようとしたときにのみ、CPU によって認識されます。結果として発生する障害は、より優先度の高いアボート（データアボートを含む）が発生していなければ、命令フォルトステータスレジスタにより報告されます。

データの読み出し / 書き込み時の外部アボート

データの読み出しまたは書き込み時に外部で生成されたエラーは、正確な場合と不正確な場合があります。つまり、外部で不正確なアボートが発生した場合、アボートハンドラが開始された時点で、R14_ABORT がその例外を発生した命令に関連するアドレスを保持しているとは限りません。このため、外部アボートは回復不能なことがあります。

アボート状態がリエントラントでないときに、不正確な外部アボートによってアボート状態に入った場合、R14 と SPSR の値が破壊されているため、プロセッサは回復不能な状態になります。この理由から、不正確な外部アボートの存在は、アボート状態がリエントラントな場合のみ、プロセッサから認識されることが必要です。これは、CSPR にある、不正確な外部アボートを示すマスクにより管理されます。このマスクは A ビットと呼ばれます。

不正確な外部アボートによりアボート状態に入った場合、DFSR には不正確な外部アボートの存在が示されています。データアクセスで不正確な外部アボートが発生した場合、FAR は更新されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.6 フォルトステータスレジスタとフォルトアドレスレジスタのサポート

統合型と、命令とデータが独立したハーバード型のどちらのアドレス空間メモリモデルを使用しているかにより、フォルトの通知には 1 つまたは 2 つの FSR が使用されます。アドレスに関する異なる情報を通知するために、3 つの FAR が使用されます。FSR のエンコードを表 B5-8 に示します。

表 B5-8 フォルトステータスレジスタのエンコード

優先度	ソース	FS[10,3:0]	FAR	IFAR[a]
最高	アライメント	0b00001	有効	有効
	バックグラウンド	0b00000	有効	有効
	アクセス許可	0b01101	有効	有効
	正確な外部アボート	0b01000	有効	有効
	不正確な外部アボート	0b10110	予測不能	予測不能
	正確なパリティエラー例外	0b00110	[b]	有効
	不正確なパリティエラー例外	0b11000	予測不能	予測不能
最低	デバッグイベント	0b00010	[c]	予測不能

a. IFAR は、プリフェッチアボート時にのみ更新されます。
b. *データフォルトアドレスレジスタ*（DFAR）がパリティエラーにより更新されるかどうかは実装定義です。
c. FAR は、ウオッチポイントデバッグイベントの場合を除き更新されません。更新される場合の値は予測不能です。

他の FSR エンコードはすべて予約されています。

どのアボートベクタが取得されるか、および各アボートタイプについてどの FSR と FAR が更新されるかの概要を P. B5-17 表 B5-9 に示します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B5-9 アボート時の FSR/FAR の更新の概要

アボートタイプ	ベクタ	正確	IFSR	DFSR	FAR	WFAR	IFAR
命令 MPU フォルト	PABORT	はい	Y	N	N	N	Y
命令デバッグアボート	PABORT	はい	Y	N	N	N	UNP
命令バックグラウンドフォルト	PABORT	はい	Y	N	N	N	Y
命令外部アボート	PABORT	はい	Y	N	N	N	Y
命令キャッシュパリティエラー	PABORT	はい	Y	N	N	N	Y
データ MPU フォルト	DABORT	はい	N	Y	Y	N	N
データデバッグアボート	DABORT	いいえ	N	Y	Y	Y	N
データバックグラウンドフォルト	DABORT	はい	N	Y	Y	N	N
データ外部アボート	DABORT	いいえ	N	Y	N	N	N
データキャッシュパリティエラー	DABORT	いいえ	N	Y	注 1	N	N

各項目の説明については以下を参照して下さい。

Y = このアボートタイプではレジスタが更新されます。

N = このアボートタイプではレジスタが更新されません。

UNP = 予測不能。

——— 注 ———

データキャッシュパリティエラーの FAR は、そのパリティエラーがプロセッサによるキャッシュメモリの読み出し時に発生した場合に更新されます。キャッシュ保守とキャッシュクリーニング操作で生成されたエラーでは、FAR を更新する必要はありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.7 CP15 のレジスタ

ARMv6 以前は、MPU はシステム制御コプロセッサのレジスタ 1、2、3、5 により制御されていました。レジスタ 0 において構成情報は提供されず、CPU ID が既知でない限りソフトウェアにより内容を判定することはできませんでした。詳細については、P. B3-7*「メイン ID レジスタ」*を参照して下さい。システム制御コプロセッサのレジスタ 1 で使用できるのは、MPU を許可する M ビットのみで（P. B3-12*「制御レジスタ」*参照）、オプションとして A ビットによりアライメントチェックが可能でした。

PMSAv6 からは、MPU はシステム制御コプロセッサのレジスタ 0、5、6、13 により制御されるようになり、レジスタ 1 で使用可能なビットの数も増えました。

すべてのレジスタアクセスは特権モードに制限され、次の形式の命令でのみアクセス可能です。

```
MCR/MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

PMSA で定義されている CP15 レジスタのいずれかがユーザモードでアクセスされた場合、未定義命令トラップが生成されます。PMSAv6 以前に使用されていたレジスタの概要を表 B5-10 に示します。PMSAv6 で使用されているレジスタの概要を P. B5-19 表 B5-11 に示します。

表 B5-10 PMSAv6 以前のレジスタの概要

機能	命令
システム制御	MRC/MCR p15, 0, Rd, c1, c0, 0
データ（または統合）キャッシュ制御	MRC/MCR p15, 0, Rd, c2, c0, 0
命令キャッシュ制御	MRC/MCR p15, 0, Rd, c2, c0, 1
書き込みバッファ制御	MRC/MCR p15, 0, Rd, c3, c0, 0
データ（または統合）アクセス許可制御 （拡張レジスタ）	MRC/MCR p15, 0, Rd, c5, c0, 2
データ（または統合）アクセス許可制御 （標準レジスタ）	MRC/MCR p15, 0, Rd, c5, c0, 0
命令アクセス許可制御 （拡張レジスタ）	MRC/MCR p15, 0, Rd, c5, c0, 3
命令アクセス許可制御 （標準レジスタ）	MRC/MCR p15, 0, Rd, c5, c0, 1
データ（または統合）領域構成（× 8）	MRC/MCR p15, 0, Rd, c6, c0-7, 0
命令領域構成（× 8）	MRC/MCR p15, 0, Rd, c6, c0-7, 1

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B5-11 CP15 の PMSAv6 でのレジスタの概要

機能	命令
MPU タイプ	`MRC/MCR p15, 0, Rd, c0, c0, 4`
システム制御	`MRC/MCR p15, 0, Rd, c1, c0, 0`
データフォルトステータス	`MRC/MCR p15, 0, Rd, c5, c0, 0`
命令フォルトステータス	`MRC/MCR p15, 0, Rd, c5, c0, 1`
フォルトアドレス	`MRC/MCR p15, 0, Rd, c6, c0, 0`
ウォッチポイントフォルトアドレス	`MRC/MCR p15, 0, Rd, c6, c0, 1`
命令フォルトアドレス	`MRC/MCR p15, 0, Rd, c6, c0, 2`
データ（または統合）領域ベースアドレス	`MRC/MCR p15, 0, Rd, c6, c1, 0`
命令領域ベースアドレス	`MRC/MCR p15, 0, Rd, c6, c1, 1`
データ（または統合）領域のサイズと許可	`MRC/MCR p15, 0, Rd, c6, c1, 2`
命令領域のサイズと許可	`MRC/MCR p15, 0, Rd, c6, c1, 3`
データ（または統合）領域のアクセス制御	`MRC/MCR p15, 0, Rd, c6, c1, 4`
命令領域のアクセス制御	`MRC/MCR p15, 0, Rd, c6, c1, 5`
MPU 領域番号	`MRC/MCR p15, 0, Rd, c6, c2, 0`
プロセス ID	`MRC/MCR p15, 0, Rd, c13, c0, 1`

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.7.1 レジスタ 0: MPU タイプレジスタ（PMSAv6）

このレジスタは読み出し専用で、Opcode_2 フィールドを 4 に設定してアクセスします。

31 — 24	23 — 16	15 — 8	7 — 1	0
SBZ/UNP	IRegion	DRegion	SBZ/UNP	S

表 B5-12 レジスタ 0

ビット	フィールド	説明
[31:24]	SBZ/UNP	
[23:16]	IRegion	命令領域の数を指定します。統合 MPU を持つ実装の場合、この値は 0 です。
[15:8]	DRegion	データまたは統合メモリ領域の数を指定します。この値が 0 になることは想定外です。
[7:1]	SBZ/ 予測不能	
[0]	S	MPU が統合 MPU か (0)、命令 MPU とデータ MPU が分離されているか (1) を示します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.7.2 レジスタ 1: 制御レジスタ

システム制御コプロセッサのレジスタ 1 にある以下のビットは、MPU の制御に使用されます。

M（ビット **[0]**） MPU の許可 / 禁止ビット。

0 = MPU は禁止されています。

1 = MPU は許可されています。

A（ビット **[1]**） アライメントフォルトチェックの許可 / 禁止ビット。

0 = アライメントフォルトチェックは禁止されています。

1 = アライメントフォルトチェックは許可されています。

W（ビット **[3]**） ライトバッファの許可 / 禁止ビット。

0 = ライトバッファは禁止されています。

1 = ライトバッファは許可されています。

実装で W ビットを省くこともできます。その場合、このビットからは 1 が読み出され、書き込みは無視されます。

U（ビット **[22]**） このビットは、アンアラインドデータアクセス操作を許可します。これには、リトルエンディアンとビッグエンディアンのデータの混在のサポートも含まれます。

EE（ビット **[25]**） このビットは、例外発生時の CPSR E ビットの設定を決定します。

——— 注 ———

1. ここに示したものは、PMSA に直接関係する制御ビットだけです。ハイベクタサポートなど他のビットは、アーキテクチャ全体の準拠性のために必要なものです。
2. PMSAv6 以前では A ビットはオプションです。
3. U ビットと EE ビットは PMSAv6 にのみ適用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.7.3　レジスタ 2: キャッシュ可能ビット（PMSAv6 以前）

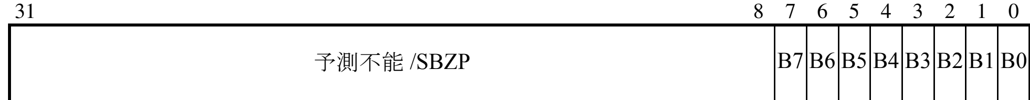
31–8	7	6	5	4	3	2	1	0
予測不能 /SBZP	C7	C6	C5	C4	C3	C2	C1	C0

CP15 のレジスタ 2 を読み出すと、8 つの保護領域それぞれについてのキャッシュ可能 (C) ビットが、ビット [7:0] として返されます。ここで、ビット [n] は領域 n に対応します。ビット [31:8] の値は予測不能です。

CP15 のレジスタ 2 に書き込むと、8 つの保護領域のキャッシュ可能 (C) ビットが更新されます。領域 n の C ビットは、ビット [n] に書き込まれた値に設定されます。ビット [31:8] には 0、またはこのレジスタのビット [31:8] から以前に読み出した値を書き込む必要があります。

どちらの場合も、MRC または MCR 命令の <CRm> フィールドは使用されず、c0 にする必要があります。実装に 8 つの保護領域の組が 1 つしか存在しない場合、Opcode_2 フィールドは 0 にします。命令とデータのアクセスに保護領域の別の組を使用している場合、Opcode_2 はデータ保護領域を選択する場合は 0、命令保護領域を選択する場合は 1 にします。

B5.7.4　レジスタ 3: バッファ可能ビット（PMSAv6 以前）



CP15 のレジスタ 3 を読み出すと、8 つの保護領域それぞれについてのバッファ可能 (B) ビットが、ビット [7:0] として返されます。ここで、ビット [n] は領域 n に対応します。ビット [31:8] の値は予測不能です。

CP15 のレジスタ 3 に書き込むと、8 つの保護領域のバッファ可能 (B) ビットが更新されます。領域 n の B ビットは、ビット [n] に書き込まれた値に設定されます。ビット [31:8] には 0、またはこのレジスタのビット [31:8] から以前に読み出した値を書き込む必要があります。

どちらの場合も、MRC または MCR 命令の <CRm> フィールドと Opcode_2 フィールドは使用されず、それぞれ c0 と 0 にする必要があります。

B5.7.5　レジスタ 4、8、10、11、12、14: 予約

これらのレジスタのいずれかにアクセス（読み出しまたは書き込み）した結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.7.6 レジスタ 5: アクセス許可ビット（PMSAv6 以前）

31–16	15–14	13–12	11–10	9–8	7–6	5–4	3–2	1–0
予測不能 /SBZP	AP7	AP6	AP5	AP4	AP3	AP2	AP1	AP0

CP15 のレジスタ 3 を読み出すと、8 つの保護領域それぞれについての AP ビットが、ビット [15:0] として返されます。ここで、ビット [2n+1:2n] は領域 n に対応します。ビット [31:16] の値は予測不能です。

CP15 のレジスタ 3 に書き込むと、8 つの保護領域の AP ビットが更新されます。領域 n の AP ビットは、ビット [2n + 1:2n] に書き込まれた値に設定されます。ビット [31:16] には 0、またはこのレジスタのビット [31:16] から以前に読み出した値を書き込む必要があります。

どちらの場合も、MRC または MCR 命令の <CRm> フィールドは使用されず、c0 にする必要があります。実装に 8 つの保護領域の組が 1 つしか存在しない場合、Opcode_2 フィールドは 0 にします。命令とデータのアクセスに保護領域の別の組を使用している場合、Opcode_2 はデータ保護領域を選択する場合は 0、命令保護領域を選択する場合は 1 にします。

AP ビットの組み合わせの解釈を P. B5-24 表 B5-13 に示します。要求されたアクセスタイプが許可されていない場合、アボート信号が ARM プロセッサに送られます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.7.7 レジスタ 5: フォルトステータス（PMSAv6）

このレジスタは、Opcode_2 フィールドの値に従って、データおよび命令フォルトステータスレジスタへのアクセスを有効にします。

表 B5-13 レジスタ 5

名前	Opcode_2
データ FSR	0
命令 FSR	1

——— 注 ———

データフォルトステータスレジスタと、P. B5-23 *「レジスタ 5: アクセス許可ビット（PMSAv6 以前）」*で定義されているアクセス許可レジスタの間には、エンコードの競合があります。PMSAv6 以前のシステムで統合アクセス許可レジスタとフォルトステータスレジスタの両方を提供する場合、サポート方法は実装定義です。

———————

フォルトステータスレジスタには、最後のアボートのソースが含まれています。この値は、アボートが発生したときに行われていたアクセスのドメイン（存在する場合）とタイプを示しています。

フォルトステータスの原因の説明とエンコードについては、P. B5-16 表 B5-8 を参照して下さい。

データフォルトステータスレジスタ

命令フォルトステータスレジスタ

31 ... 12	11	10	9	8	7 ... 4	3 ...
予約	0	S	0	0	UNP/SBZ	ステータス

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.7.8 レジスタ 6: メモリ領域プログラミング（PMSAv6 以前）

31 – 12	11 – 6	5 – 1	0
ベースアドレス	UNP/SBZP	サイズ	E

CP15 のレジスタ 6 を読み出すと、保護領域の現在のベースアドレス、サイズ、許可 / 禁止のステータスが、上の図に示すフォーマットで返されます。ビット[11:6]について読み出される値は予測不能です。

CP15 のレジスタ 6 に書き込むと、保護領域の現在のベースアドレス、サイズ、許可 / 禁止のステータスが、上の図に示すフォーマットで設定されます。ビット [11:6] には 0、または CP15 レジスタ 6 のビット [11:6] から以前に読み出した値を書き込む必要があります。

レジスタ 6 には、保護ユニットの各保護領域ごとに 1 つのレジスタが存在します。使用されるレジスタ（すなわち、影響を受ける保護領域）は、レジスタへのアクセスに使用される MCR または MRC 命令の <CRm> フィールドと Opcode_2 フィールドにより選択されます。

- <CRm> は、保護領域の番号の指定に使用されます。c0 を指定すると保護領域 0、c1 を指定すると保護領域 1、以下同様に、c7 を指定すると保護領域 7 が選択されます。
- 実装に 8 つの保護領域の組が 1 つしか存在しない場合、Opcode_2 フィールドは 0 にします。
- 命令とデータのアクセスに保護領域の別の組を使用している場合、Opcode_2 はデータ保護領域を選択する場合は 0、命令保護領域を選択する場合は 1 にします。

レジスタ 6 に読み出し / 書き込みされる値のフィールドの意味は次のとおりです。

- En ビットは、関連する保護領域を許可 / 禁止します。

  0 = 保護領域は禁止されています。

  1 = 保護領域は許可されています。

  禁止されている保護領域はどのアドレスとも一致せず、どのような形でもメモリアクセスシーケンスに影響しません。リセット時には、すべての保護領域が禁止されます。

- サイズフィールドは、関連する保護領域のサイズを選択します。このサイズは 4KB から 4GB までです。このエンコードを P. B5-26 表 B5-14 に示します。

- ベースアドレスフィールドは、関連する保護領域の最下位アドレスのビット [31:12] を指定します。

  最下位アドレスは、領域サイズの複数倍であることが必要です。サポートされる領域の最小サイズの関係で、ビット [11:0] は常に 0 です。P. B5-26 表 B5-14 に従い、ベースアドレスの他のビットは 0 の必要があります。この関係が維持されていない場合、保護領域が正しく境界アラインされず、動作は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B5-14 領域サイズのエンコード

サイズフィールド	エリアのサイズ	ベースエリアの制約
0b00000-0b01010	予測不能	-
0b01011	4KB	なし
0b01100	8KB	ビット [12] は 0 であること
0b01101	16KB	ビット [13:12] は 0 であること
0b01110	32KB	ビット [14:12] は 0 であること
0b01111	64KB	ビット [15:12] は 0 であること
0b10000	128KB	ビット [16:12] は 0 であること
0b10001	256KB	ビット [17:12] は 0 であること
0b10010	512KB	ビット [18:12] は 0 であること
0b10011	1MB	ビット [19:12] は 0 であること
0b10100	2MB	ビット [20:12] は 0 であること
0b10101	4MB	ビット [21:12] は 0 であること
0b10110	8MB	ビット [22:12] は 0 であること
0b10111	16MB	ビット [23:12] は 0 であること
0b11000	32MB	ビット [24:12] は 0 であること
0b11001	64MB	ビット [25:12] は 0 であること
0b11010	128MB	ビット [26:12] は 0 であること
0b11011	256MB	ビット [27:12] は 0 であること
0b11100	512MB	ビット [28:12] は 0 であること
0b11101	1GB	ビット [29:12] は 0 であること
0b11110	2GB	ビット [30:12] は 0 であること
0b11111	4GB	ビット [31:12] は 0 であること

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.7.9 フォルトアドレス（PMSAv6）

CRm フィールドが c0 の場合、レジスタ 6 は Opcode_2 フィールドの値に従って、データおよび命令フォルトアドレスレジスタへのアクセスを許可します。

表 B5-15 フォルトアドレスレジスタのデコード（PMSAv6）

名前	Opcode_2
FAR	0
WFAR	1
IFAR	2

——— 注 ———

WFAR 機能は、CP15 から CP14 のデバッグアーキテクチャに移行しており、CP15 による WFAR のデコードは ARMv6 では推奨されません。移行先については、P. D3-2*「コプロセッサ 14 のデバッグレジスタ」*を参照して下さい。

———————

CRm の値の c1 と c2 は、P. B5-28*「レジスタ 6: メモリ領域プログラム (PMSAv6)」*で定義されているように、メモリ領域属性の構成に使用されます。

FAR と IFAR はアボート時に、P. B5-28 表 B5-16 に従って更新されます。

CP15 のレジスタ 6 に書き込むと、FAR と IFAR の値を書き込むことができます。これは、デバッガが値を復元するときに便利です。

WFAR は、モニタモードでのデバッグイベント時に更新されます。ARM ステートでウォッチポイントが取得された場合、WFAR にはイベントが発生したときの命令アドレス + 0x8 が含まれています。Thumb® ステートでウォッチポイントが取得された場合、WFAR にはイベントが発生したときの命令アドレス + 0x4 が含まれています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

——— 注 ———

1. ARMv6 以前は、フォルト通知は一般に致命的と見なされ、標準の例外処理機構によってのみ通知されていました（例外モードの SPSR で、R14 をリンクレジスタとして使用）。

2. FAR レジスタと領域構成レジスタのデコードの競合があるため、この場合に FAR レジスタをサポートするかどうかは実装定義です。ARM では、CRm の値を c0 に保持し、Opcode_2 フィールドを次のように使用することを推奨します。

表 B5-16 フォルトアドレスレジスタの推奨のデコード（PMSAv6 以前）

名前	Opcode_2
FAR	4
WFAR	5
IFAR	6

B5.7.10 レジスタ 6: メモリ領域プログラム（PMSAv6）

レジスタ 6 は、MPU 領域のプログラムと、前のセクションで説明したフォルトアドレス情報に使用されます。アクセスされるレジスタは、表 B5-17 に示すように、CRm と Opcode_2 フィールドの値に依存します。

MPU 領域番号により、特定の領域をサポートするレジスタの組が選択されます。

表 B5-17 MPU 領域のプログラムレジスタ

CRm	Opcode_2	説明
C1	0	データ（または統合）領域ベースアドレス
C1	1	命令領域ベースアドレス
C1	2	データ（または統合）領域のサイズと許可
C1	3	命令領域のサイズと許可
C1	4	データ（または統合）領域のアクセス制御
C1	5	命令領域のアクセス制御
C2	0	MPU 領域番号

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

領域のベースアドレス

ベースアドレスは、領域サイズに境界アラインしている必要があります。

サポートされる物理アドレス空間のサイズは、すべてのビットが 1 の値をベースアドレスレジスタに書き込み、再度読み出すことで判定できます。

サポートされる物理アドレス空間は、1 にセットされている最上位のビットで示されます（サポートされるアドレス空間 = 2^{N+1}）。

領域の解像度は、1 にセットされている最下位のビットで示されます（解像度 = 2^N バイト）。

領域サイズ

領域サイズは、領域サイズレジスタにエンコードされます。

メモリ領域は、使用前に必ず許可する必要があります。領域を許可するには、En（ビット [0]）を 1 にセットします。リセット時には、メモリ領域が禁止されます。

領域の最小サイズと最大サイズは実装定義です。

キャッシュを含む実装では、領域の最小サイズはキャッシュライン長の整数倍の必要があります。これによって、キャッシュラインの途中でキャッシュ属性が変更されることを回避できます。

実装でサポートされている範囲外の領域サイズを書き込んだ場合、動作は予測不能です。

表 B5-18 領域サイズのエンコード

領域サイズフィールド（サイズ [4:0]）	領域サイズ（バイト）
0b00000	予約（UNP）
N (N は 0 以外)	2^{N+1}

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

領域アクセスの制御

このレジスタは、メモリ領域のメモリ領域属性と、アクセス許可を定義します。メモリ属性は P. B5-10 *「メモリアクセスの属性」*で定義されています。

実装で、命令アクセス用の独立したメモリ領域属性をサポートすることもできます。命令メモリ属性にアクセスするには、Opcode_2 を 5 に設定します。

31 13	12	11	10 8	7 6	5 3	2	1	0
予約	XN	予約	AP	予約	TEX	S	C	B

各項目の説明については以下を参照して下さい。

TEX、S、C、B ビットの説明については、P. B5-10 *「メモリアクセスの属性」*を参照して下さい。

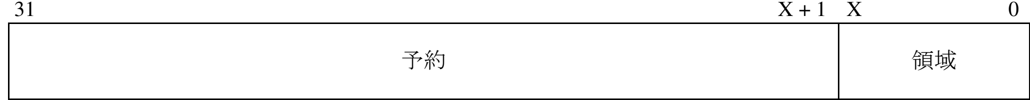
AP[2:0] は、P. B5-9 表 B5-3 の AP ビットを示しています。

ビット [12] は、P. B5-9 表 B5-4 の XN ビットを示しています。

命令を実行するメモリ領域は、AP ビットで読み出しアクセスが許可され（ユーザ、特権、または両方のモード）、XN ビットが 0 の必要があります。

メモリ領域番号

メモリ属性、アクセス許可、メモリ領域の各レジスタは、実装されているメモリ領域ごとに 1 つ存在します。どのレジスタにアクセスするかは、領域番号レジスタに含まれている値により判定されます。



領域 [X:0] は、どのグループのレジスタがアクセスされるかを定義します。実装によりサポートされている領域の数 (N) は、MPU タイプレジスタから取得できます。詳細については、P. B5-20 *「レジスタ 0: MPU タイプレジスタ（PMSAv6）」*を参照して下さい。

X の値は、サポートされている領域数の 2 を底とする対数で、整数値に切り上げられます。領域番号は、領域 0 から領域（N - 1）までです。このレジスタに N 以上の値を書き込んだ場合、関連するレジスタバンクアクセスとともに、結果は予測不能です。

B5.7.11 レジスタ 7 から 9: キャッシュとライトバッファ制御

これらのレジスタは、第 B6 章 *「キャッシュとライトバッファ」*で定義されているキャッシュとライトバッファ機能に関連しています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B5.7.12 レジスタ 13: プロセス ID（PMSAv6）

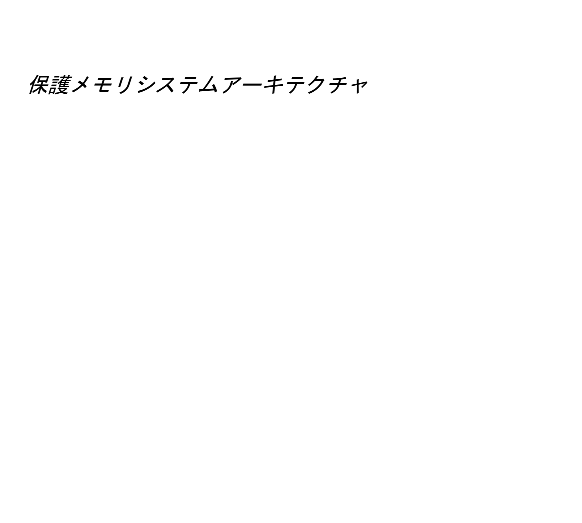
このレジスタは、現在実行されているプロセスを判定します。このレジスタにアクセスするには、Opcode_2 を 1 に指定します。

リセット時にはプロセス ID レジスタの値は未定義です。

このレジスタは、*組み込みトレースマクロセル*（ETM）とデバッグロジックにより使用されます。この値は ETM によりトレースユニットに伝えられ、現在実行されているプロセスを示すことができます。このためには、この値は各プロセスで固有の値がプログラムから設定される必要があります。

この値は、プロセス固有のブレークポイントや命令を有効にするためにも使用できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.



 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 B6 章
キャッシュとライトバッファ

本章では、MMU ベースのメモリシステムと保護ユニットベースのメモリシステムに共通な、キャッシュとライトバッファの制御機能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。

- *キャッシュとライトバッファの概要* : P. B6-2
- *キャッシュの構成* : P. B6-4
- *キャッシュのタイプ* : P. B6-7
- *L1 キャッシュ* : P. B6-10
- *キャッシュのレベルの追加に関する考慮事項* : P. B6-12
- *CP15 のレジスタ* : P. B6-13

ARMv6 以前は、アーキテクチャのガイドラインは構成と管理機能のためのシステム制御コプロセッサ（CP15）を使用して提供されていました。ARMv6 ではシステム制御コプロセッサの使用が必須で、以前のアーキテクチャのバリエーションの機構が拡張されています。以下の説明で、ARMv6 で導入された新機能には注記がつけられています。これらの新機能を、以前のバージョンのアーキテクチャ（ARMv4 や ARMv5）に準拠した実装で採用するかどうかは実装定義です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.1 キャッシュとライトバッファの概要

現在の ARM® メモリシステムでは、平均システム性能を向上させるキャッシュとライトバッファは一般的なものとなっています。近年、コアのクロック速度は、メモリのアクセス速度よりも急速に増大しています。そのほかプロセスの微細化、オンチップメモリのコスト、システムの消費電力の制限などの要素から、システムに要求される条件の増大に対応するためキャッシュを使うことがよくあります。しかし、階層（P. B1-4 *「メモリ階層」*参照）の別の場所にあるメモリと比較して密結合メモリのコストは依然として高価です。このため、密結合メモリはキャッシュとライトバッファの提供する共有メモリモデルに適しています。

キャッシュは高速なメモリのブロックで、アドレス情報（通常は TAG ビットと呼ばれます）と関連データの両方を含んでいます。キャッシュは、メモリアクセスの平均速度の向上を目的としたものです。キャッシュは局所性に関する次の 2 つの原則に基づいて動作します。

空間的局所性 ある位置にアクセスが行われた後で、次のアクセスは隣接した位置に対して行われる可能性が高いこと。例としては、シーケンシャルな命令の実行やデータ構造の使用が挙げられます。

時間的局所性 あるメモリ領域へのアクセスは、短い時間に繰り返される可能性が高いこと。例としては、ループのあるコードの実行が挙げられます。

記憶される制御情報の量を最小にするため、空間的な局所性の特徴を使用して、いくつかのロケーションを同じ TAG によりグループ化します。メモリロケーションの論理ブロックは一般にキャッシュラインと呼ばれ、通常は 32 バイト長です。データがキャッシュにロードされていると、以降のロードとストアのアクセス時間が劇的に減少し、総合的なパフォーマンスが向上します。既にキャッシュにある情報へのアクセスをキャッシュヒットと呼び、他のアクセスをキャッシュミスと呼びます。

通常、キャッシュは自己管理が行われ、自動的に更新されます。プロセッサがキャッシュ可能な位置にアクセスすると、常にキャッシュがチェックされます。そのアクセスがキャッシュヒットの場合、アクセスは直ちに発生します。それ以外の場合、ロケーションのひとつが割り当てられ、キャッシュラインがメモリからロードされます。キャッシュのトポロジとアクセスポリシーにはいろいろな種類のものが存在します。すべてのキャッシュトポロジとアクセスポリシーは、メモリコヒーレンシモデルに準拠する必要があります。メモリの順序付けの詳細については、第 B2 章*「メモリオーダモデル」*を参照して下さい。

ライトバッファは、メインメモリへのストアを最適化することを目的とする高速メモリのブロックです。ストアが発生すると、データ、アドレス、データサイズなどその他の詳細が、高速でライトバッファに書き込まれます。次に、ライトバッファはメインメモリの速度でストアを完了します。これは通常、ARM プロセッサの速度よりもはるかに遅くなります。その間に、ARM プロセッサは全速で以後の命令を実行できます。

ライトバッファとキャッシュを使用すると、次の理由から多くの問題が起きる可能性があります。

- メモリへのアクセスが、プログラマが通常予測するのとは異なる時期に発生する。
- データアイテムが、物理的に複数の場所に保持される。

本章では、キャッシュの特徴、関連する問題、キャッシュとライトバッファを管理するために使用できる制御機構について説明します。これらは、第 B4 章*「仮想メモリシステムアーキテクチャ」*で説明されている MMU システムアーキテクチャと、第 B5 章*「保護メモリシステムアーキテクチャ」*で説明されている PMU システムアーキテクチャの両方に共通です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ARMv6 以前は、ARM コアに関連するキャッシュは仮想アドレシングを使用していました。これは、仮想アドレスから物理アドレスへのマッピングが変化した、または P. B2-20「*キャッシュコヒーレンシの概要*」で説明されている他の特定の条件が発生したときはキャッシュの無効化、クリーニング、または両方が必要なことを意味します。ARMv6 では、物理キャッシュに対応したキャッシュ動作が導入され、コンテキストスイッチ時にエントリをフラッシュする必要性が低減するように設計されています。

第 B8 章で説明されている*高速コンテキストスイッチ拡張機能*（FCSE）が使用されている場合、本章で示す仮想アドレスへの参照はすべて、それによって生成される修飾仮想アドレスを意味します。

——— 注 ———

FCSE の使用は、ARMv6 では推奨されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.2 キャッシュの構成

キャッシュの記憶域の基本単位はキャッシュラインです。キャッシュラインにキャッシュされたデータや命令が含まれている場合、そのキャッシュラインは*有効*、そうでない場合は*無効*と呼びます。リセット時には、すべてのキャッシュラインが無効化されます。キャッシュラインにメモリからデータや命令がロードされると、そのキャッシュラインは有効になります。

キャッシュラインが有効な場合、連続したメインメモリロケーションのブロックの最新の値が含まれています。キャッシュラインの長さは必ず 2 のべき乗で、一般に 16 ～ 64 バイトです。キャッシュラインの長さが 2^L バイトの場合、メインメモリロケーションのブロックは必ず 2^L バイト境界にアラインされています。アライメントの条件から、仮想アドレスのビット [31:L] はキャッシュラインのすべてのバイトについて同じです。

キャッシュヒットは、ARM プロセッサにより指定されたアドレスのビット [31:L] が、有効なキャッシュラインに対応したアドレスのビットと一致する場合に発生します。従来は、これは仮想アドレスに対する一致でした。しかし、ARMv6 からは、規定の動作は物理アドレスのキャッシュに合わせられています。

キャッシュは通常、いくつかのキャッシュセットに分割され、各セットには固定数のキャッシュラインが割り当てられます。キャッシュセットの数（NSETS）は常に 2 のべき乗です。キャッシュライン長が 2^L バイトで、キャッシュセットの数が 2^S の場合、ARM プロセッサにより指定される仮想アドレスのビット [L + S - 1:L] がキャッシュセットの選択に使用されます。そのセットに含まれるキャッシュラインのみが、そのアドレスのデータや命令を保持できます。これらは一般に、パフォーマンス上の理由から並列にチェックされます。

仮想アドレスの残りのビット（ビット [31:L + S]）は、*タグ*ビットと呼ばれます。キャッシュヒットは、ARM プロセッサにより指定されたアドレスのタグビットが、選択されたキャッシュセットの有効なラインに関連付けられたタグビットと一致する場合に発生します。

P. B6-5 図 B6-1 は、仮想アドレスを使用してキャッシュのデータや命令を参照する方法です。この形式のキャッシュは、多くの場合*仮想インデクス仮想タグ*（VIVT）キャッシュ、または仮想キャッシュと呼ばれます。キャッシュアクセスの前にアドレス変換の必要がある場合、そのキャッシュは*物理インデクス物理タグ*（PIPT）キャッシュ、または物理キャッシュと呼ばれます。実装でハイブリッド手法を（VIPT）選択する場合もありますが、この場合はさらにいくつかの記述が追加されます。詳細については、P. B6-11 *「ページテーブルのマッピングの制限」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

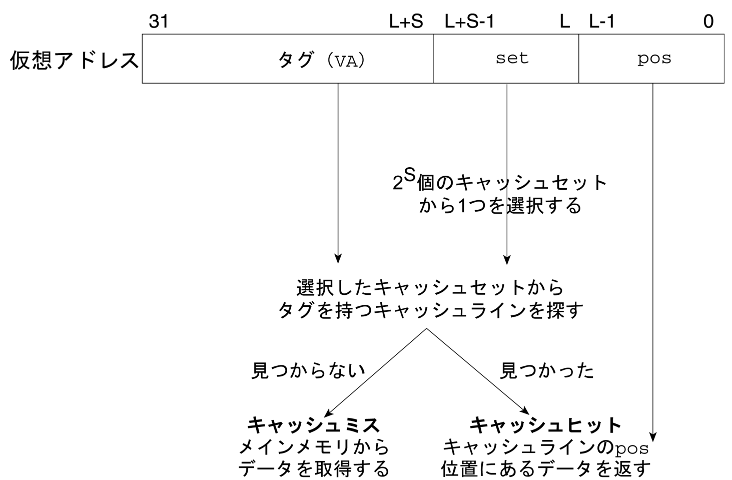


図 B6-1 仮想キャッシュのルックアップ

B6.2.1 セットアソシアティビティ

キャッシュの*セットアソシアティビティ*は、各キャッシュセットに含まれるキャッシュライン数で、ASSOCIATIVITY またはキャッシュのウェイ数（NWAYS）と呼ばれます。この数は ≥1 の任意の数で、2 のべき乗には制限されません。

セットアソシアティビティが低いと、一般にキャッシュのルックアップが簡素化され、関連する電力消費が最小になります。ただし、頻繁に使用されるメモリキャッシュラインで、特定のキャッシュセットを使用するものの数がセットアソシアティビティを超えた場合、メインメモリの動作が増え、パフォーマンスが低下します。これは、*キャッシュの競合*と呼ばれる現象で、セットアソシアティビティが少ないほど起きやすくなります。

2 つの極端なケースとして、フルアソシアティブなキャッシュと、ダイレクトマップキャッシュが考えられます。

- *フルアソシアティブキャッシュ*は、キャッシュ全体が 1 つのキャッシュセットで構成されているものです。これは N ウェイセットアソシアティブで、N はキャッシュのラインの総数を指しますフルアソシアティブキャッシュでは、キャッシュのルックアップのたびにすべてのキャッシュラインをチェックする必要があります。
- *ダイレクトマップキャッシュ*は、1 ウェイセットアソシアティブのキャッシュです。各キャッシュセットは単一のキャッシュラインで構成されるため、キャッシュのルックアップ時は 1 つのキャッシュラインのみを選択してチェックする必要があります。ただし、ダイレクトマップキャッシュでは、キャッシュの競合が非常に起きやすくなります。

各キャッシュセット内のキャッシュラインには、0 から（セットアソシアティビティ）- 1 までの番号が付けられます。各キャッシュラインに関連付けられた番号を、*ウェイ番号*と呼びます。ウェイ番号は、セットアドレスフィールドと組み合わせて、キャッシュメモリの特定のキャッシュラインブロックを示します。

*キャッシュウェイ*は、ウェイ番号の特定の値に関連付けられたすべての（2^S）キャッシュラインとして定義されます。一部のキャッシュ操作では、ソフトウェアループでキャッシュの各ウェイに対してシステマティックに操作を行うため、たとえば P. B6-31「*レジスタ 9: キャッシュのロックダウン機能*」で説明されているキャッシュロックダウン機構などの場合、パラメータとしてキャッシュウェイ番号が必要です（A、B、C の形式）。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

―――― 注 ――――

ARM アーキテクチャの以前のバージョンでは、キャッシュウェイに関連付けられた値は*インデクス*と呼ばれていました。ただし、コンピュータアーキテクチャの用語についての慣習では、インデクスという用語はキャッシュセットのアドレスフィールドを指します。このため混用を避けるため、ここではセットとウェイという用語を使用します。このため、特定のキャッシュラインに対するキャッシュ無効化操作は、*セット / インデクスによる無効化*ではなく、*セット / ウェイによる無効化*と呼びます。キャッシュに関する文で*インデクス*という用語を使用する場合、セット規則に関連するもの、つまり仮想インデクスキャッシュを指します。実装のドキュメントでは、この点で混乱が生じる可能性があります。アーキテクチャのマニュアルと実装データ、たとえばテクニカルリファレンスマニュアルなどの情報を関連付けるときには注意が必要です。

――――――――

B6.2.2 キャッシュサイズ

一般に、キャッシュのサイズが増大するほど、メモリアクセスがキャッシュにヒットする可能性は高くなります。これによって、メモリアクセスあたりの平均時間が短くなり、パフォーマンスが向上します。ただし、大容量のキャッシュは一般にコアの大きな面積を占め、大きな電力を消費します。したがって、ARM メモリシステムではパフォーマンス、コアの面積、消費電力の相対的な重要性により、各種のサイズのキャッシュを使用します。

キャッシュのサイズは、3 つの要素の積に分解できます。

- LINELEN: キャッシュラインの長さで、バイト単位で表されます。
- ASSOCIATIVITY: セットアソシアティビティを示します。キャッシュセットは、ASSOCIATIVITY 個のキャッシュラインで構成されるため、キャッシュセットのサイズは ASSOCIATIVITY × LINELEN で示されます。
- NSETS: キャッシュを構成するキャッシュセットの数。

サイズとアドレスビットの定義については、P. B6-4*「キャッシュの構成」*と P. B6-5*「セットアソシアティビティ」*を参照して下さい。

```
Cache size = ASSOCIATIVITY × NSETS × LINELEN
           = NWAYS × NSETS × LINELEN
           = NWAYS × 2^S × 2^L bytes
```

データと命令に別のキャッシュが使用される場合、それぞれについてこれらのパラメータの値が異なり、結果としてキャッシュサイズが異なる場合があります。

ARMv6 以降は、システム制御コプロセッサのキャッシュタイプレジスタは、L1 キャッシュを定義するための必須の方式です。詳細については、P. B6-14*「キャッシュタイプレジスタ」*を参照して下さい。以前のバージョンのアーキテクチャでも、この方式を使用することをお勧めします。また、レベル 2 キャッシュのサポートのガイドラインについては P. B6-12*「キャッシュのレベルの追加に関する考慮事項」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.3 キャッシュのタイプ

キャッシュには多くのタイプがあり、次のような実装上の選択により区別されます。

- キャッシュのサイズとアソシアティビティ
- 命令フェッチの処理方法
- データ書き込みの処理方法

以下のサブセクションでは、実装オプションのいくつかについて詳細を解説します。

B6.3.1 統合キャッシュと分離キャッシュ

メモリシステムでは、命令フェッチの処理とデータのロードやストアの処理に同じキャッシュを使用することもできます。このようなキャッシュを*統合*キャッシュと呼びます。統合されたキャッシュとメモリモデルは、多くの場合*フォンノイマンアーキテクチャ*と呼ばれます。

または、メモリシステムで、命令フェッチの処理とデータのロードやストアの処理に別のキャッシュを使用することもできます。この場合、2 つのキャッシュを合わせて*分離*キャッシュと呼び、それぞれを*命令*キャッシュと*データ*キャッシュと呼びます。別の命令キャッシュとデータキャッシュを使用した場合、メインメモリが統合であっても、多くの場合*ハーバード*アーキテクチャと呼ばれます。

分離キャッシュを使用すると、キャッシュメモリがマルチポートでなくても、メモリシステムが多くの場合命令フェッチとデータのロード/ストアを同じクロックサイクルに実行できる利点があります。主要な欠点は、命令キャッシュがデータキャッシュ、メインメモリ、または両方と比べて古くならないようにするための注意が必要なことです。詳細については、P. B2-20*「メモリのコヒーレンシとアクセスの考慮点」*を参照して下さい。

また、メモリシステムに命令キャッシュだけがありデータキャッシュがない、または逆にデータキャッシュだけがあり命令キャッシュがない構成も可能です。メモリシステムアーキテクチャの観点からは、このようなシステムは分離キャッシュで、片方のキャッシュが存在しない、またはサイズが 0 であると見なされます。

B6.3.2 ライトスルーキャッシュとライトバックキャッシュ

データストアアクセスでキャッシュヒットが発生すると、データを含むキャッシュラインが新しい値に更新されます。このキャッシュラインは後で別のアドレスに再割り当てされるため、新しい値はメインメモリのロケーションにも書き込む必要があります。この処理を行う手法は一般に 2 つあります。

- *ライトスルー*キャッシュでは、新しいデータはメモリ階層の次のレベルに書き込まれます。データがメモリ階層の次のレベルから可視であることを保証するため、DSB 同期バリア（P. B2-18 *「DataSynchronizationBarrier (DSB) CP15 レジスタ 7」*参照）が必要です。これは通常、プロセッサの速度低下を避けるためライトバッファにより行われます。
- *ライトバック*キャッシュでは、キャッシュラインは*ダーティ*としてマークされます。これは、そのキャッシュラインにメインメモリよりも新しいデータが保持されていることを示しています。ダーティなキャッシュラインが選択され、別のアドレスに再割り当てされるとき、現在キャッシュラインにあるデータはメインメモリに書き戻されます。この方法でキャッシュラインの内容を書き戻すことを、キャッシュラインの*クリーニング*、または*犠牲書き込み*と呼びます。ライトバックキャッシュは、*コピーバック*キャッシュとも呼ばれます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ライトスルーキャッシュは、プロセッサがライトバッファの処理よりも速くデータストアを生成可能な場合、プロセッサの停止を引き起こすことがあります。この場合、システムのパフォーマンスが低下します。

ライトバックキャッシュは、キャッシュラインが再割り当てされるときにのみメインメモリにストアを行います。それ以外は、キャッシュラインに対して多数のストアが発生しても、メインメモリへのストアは行われません。このため、ライトバックキャッシュではライトスルーキャッシュと比較して、メインメモリへのストアは通常は少なくなります。これによってキャッシュからメインメモリへの帯域幅要件が減少し、ライトスルーキャッシュについて前に説明した問題が軽減されます。ただし、ライトバックキャッシュには次のような多くの欠点があります。

- キャッシュとメインメモリの内容が長い時間相違したままになります。詳細についてはP. B2-20 *「メモリのコヒーレンシとアクセスの考慮点」* を参照して下さい。
- データロードが完了する前のメインメモリ操作のワーストケースシーケンスが長くなり、システムのワーストケース割り込みレイテンシが増大する可能性があります。
- 正当性を保証するため、キャッシュとメモリの管理ポリシーの一部としてキャッシュクリーニングが必要になることがあります。
- 実装がより複雑になります。

一部のライトバックキャッシュでは、ライトバックとライトスルーの動作を選択できます。

B6.3.3 読み出し割り当てキャッシュと書き込み割り当てキャッシュ

データストアアクセスでキャッシュミスが発生した場合の対処方法には、一般に二通りあります。

- *読み出し割り当て*キャッシュでは、データは単にメインメモリに記憶されます。キャッシュラインは、データが読み出し / ロードされるときのにみメモリロケーションに割り当てられ、書き込み / ストアされるときには割り当てられません。
- *書き込み割り当て*キャッシュでは、キャッシュラインがデータに割り当てられ、現在のメインメモリの内容が読み込まれてからデータがキャッシュラインに書き込まれます。キャッシュがライトスルーとライトバックのどちらであるかにより、メインメモリにも書き込まれる可能性があります。

これらの手法の主要な利点と欠点はパフォーマンスに関するものです。読み出し割り当てキャッシュと比較して、書き込み割り当てキャッシュでは余分なメインメモリへの読み出しアクセスが発生し、またデータがキャッシュに置かれるため、以降のストアでメインメモリへの保存アクセスが発生することがあります。利点と欠点のどちらが大きいかは、対象のデータへのロード / ストアアクセスの回数と種類、およびキャッシュがライトスルーとライトバックのどちらかにより決定されます。

ARMv6 以前は、ARM メモリシステムで書き込み割り当てと読み出し割り当てのどちらのキャッシュが使用されるかは実装定義でした。

VMSAv6 では、P. B4-11 *「C、B、TEX のエンコード」* で説明されているようにキャッシュ割り当てポリシーが定義されています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.3.4 置換方式

キャッシュがダイレクトマップでない場合、メモリアドレスでキャッシュミスが発生したときは、そのアドレスに関連付けられたキャッシュセットのキャッシュラインの 1 つを再割り当てする必要があります。このキャッシュラインを選択する方法を、キャッシュの*置換方式*と呼びます。

一般的な置換方式には次のものがあります。

ランダム置換

キャッシュ制御ロジックに擬似乱数発生器が含まれており、その出力を使用して再割り当てされるキャッシュラインが選択されます。

ラウンドロビン置換

キャッシュ制御ロジックに、再割り当てされるキャッシュラインを選択するためのカウンタが含まれています。再割り当てが行われるたびにカウンタがインクリメントされ、次の再割り当て時には別のキャッシュラインが選択されます。

ARM 実装では、システム制御コプロセッサにある制御ビットを使用して、2 つの置換方式のいずれかを選択できます。詳細については、P. B6-18*「レジスタ 1: キャッシュとライトバッファの制御ビット」*を参照して下さい。一般に、単純で容易に予測可能なラウンドロビン置換のような手法が 1 つの選択で、ランダム置換アルゴリズムが別の選択となります。

ラウンドロビン置換方式はより予測可能ですが、データセットによりパフォーマンスが大きく変化します。たとえば、プログラムが D1、D2、...、Dn というデータアイテムに周期的にアクセスし、これらのデータアイテムすべてが同じキャッシュセットを使用している場合を考えます。m ウェイセットアソシアティブキャッシュでラウンドロビン置換を使用した場合、プログラムは次のような結果になります。

- $n \leq m$ の場合、これらのデータアイテムについて 100% 近いキャッシュヒットが得られます。
- n が $m + 1$ 以上の場合、キャッシュヒットは 0% になります。

つまり、処理されるデータが少し増加しただけで、キャッシュの有効性が劇的に低下します。

ランダム置換は動作がより予測しにくくなります。このため、ワーストケースの動作が判定しにくくなりますが、キャッシュの平均パフォーマンスが、ワーキングセットのサイズなどの影響を受けにくくなります。

アーキテクチャ的には、置換手法の選択は必須ではありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.4 L1 キャッシュ

L1 キャッシュは、CPU に最も近いレベルのキャッシュです。ARMv6 で完全に仕様化されているのは、このレベルのキャッシュのみです。

L1 キャッシュは、命令とデータに別のキャッシュを使用するハーバード構成と、命令とデータを含むすべてのキャッシュ対象のアイテムが統合した構造に保持されるフォンノイマン構成の、どちらの方式でも実装できます。ハーバード構成では、命令キャッシュとデータキャッシュのコヒーレンシをサポートするためのハードウェアを実装に含める必要はありません。このようなサポートが必要になる場合、たとえば自己変更型のコードでは、問題を回避するためにソフトウェアでキャッシュクリーニング命令を使用する必要があります。

L1 キャッシュは、ソフトウェアから見て次のように動作している必要があります。

- 仮想メモリから物理メモリへのマッピングが異なるせいで、キャッシュ可のエントリを、ソフトウェアでクリーニングや無効化をする必要はない。
- 同じ物理アドレスへの複数のエイリアスが、ページテーブルでキャッシュに指定されているメモリ領域に存在する可能性がある。このエイリアスは、P. B6-11 *「ページテーブルのマッピングの制限」*に説明されている 4KB のスモールページの制約に従う。

これらの動作条件に適合すれば、キャッシュは仮想アドレスまたは物理アドレス（インデクスを含む）で実装可能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.4.1 ページテーブルのマッピングの制限

仮想インデクス物理アドレスのキャッシュの実装と、そのエイリアスの処理を容易にするため、仮想アドレスのビット [13:12] を再マップするページのマッピングには制限を設ける必要があります。この場合、制限の必要は、キャッシュタイプレジスタの命令キャッシュとデータキャッシュのキャッシュサイズフィールドのビット 11 をセットすることで示されます。詳細については P. B6-15「*キャッシュサイズフィールド*」を参照して下さい。

この制限により、仮想アドレスのこれらのビットを使用して、エイリアスの問題を避けるためのハードウェアサポートを必要とせずにキャッシュにインデクス付けが可能になります。この制限は、キャッシュウェイのサイズが最大16KBまでのキャッシュへの仮想インデクスをサポートしています。サポートされるウェイの数には制限はありません。4KB 以下のキャッシュウェイでは、任意のアドレス（仮想または物理アドレス）が 1 つのキャッシュセットにしか割り当てられないため、本質的にこの制限に影響されません。P. B6-4「*キャッシュの構成*」にある定義を使用すると、仮想インデクス付けに関連する ARMv6 キャッシュポリシーは次のように説明できます。

```
Log2(NSETS × LINELEN) =< 12        ; no VI restriction
12 < Log2(NSETS × LINELEN) =< 14   ; VI restrictions apply
Log2(NSETS × LINELEN) > 14         ; PI only, VI not supported
```

非共有としてマークされているページの場合、キャッシュサイズフィールドのビット 11 がセットされている、つまり仮想アドレスのビット [13:12] を再マップするページに制限が適用される場合、4KB ページを使用したときのエイリアスの問題を回避するため、次の 2 つの制限のセットのうちいずれかが適用されます。

- 複数の仮想アドレスが同じ物理アドレスにマップされる場合、すべてのマッピングについて仮想アドレスのビット [13:12] が等しく、物理アドレスのビット [13:12] とも等しい必要があります。同じ物理アドレスは、4KB、64KB、セクションという異なったページサイズの TLB エントリによってマップできます。
- 一つのアドレスへのマッピングがすべて 4KB のページサイズを使用している場合、仮想アドレスのビット [13:12] が物理アドレスのビット [13:12] に等しいという条件は必要ありません。この場合も、すべての仮想アドレスエイリアスのビット [13:12] は等しい必要があります。

仮想アドレス内でこれより上位のビットが、対応する物理アドレスのビットに等しい必要はありません。

——— 注 ———

ARMv6 では、1KB（タイニー）ページは使用されません。詳細については、P. B4-3「*VMSAv6 で導入された主要な変更*」を参照して下さい。

———————

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.5 キャッシュのレベルの追加に関する考慮事項

P. B1-4 *「メモリ階層」*で説明しているように、システムに追加のレベルのキャッシュを実装することができます。ARMv6 では、レベル 1 メモリシステムについて詳細に定義していますが、他のレベルのキャッシュのアーキテクチャは同様に詳細に定義されているわけではありません。ただし、レベル 2 キャッシュをサポートする ARM システムの数は増加し続けているため、レベル 2 キャッシュの採用を促進し、これらのシステム間の移植性を確保するため、標準的な手法を推奨します。すべてのレベルのキャッシュには、最低でも以下の機能を持つ制御機能を含めることを推奨します。

- キャッシュクリーニングのサポート
- キャッシュ無効化のサポート

レベル 2 キャッシュはコアに密結合することも、メモリマップされたペリフェラルとして扱うこともできます。メモリマップの場合、キャッシュの制御機能でアドレスがパラメータとして必要な場合、たとえばアドレスを指定してエントリをクリーニングする場合、そのアドレスは本質的に物理アドレス（PA）の必要があります。レベル 2 キャッシュがコアにより密接に結合されている場合、仮想アドレスと物理アドレスのどちらでも使用できます。この場合、VA と PA パラメータのどちらでも使用できます。PA パラメータを使用する場合、実装ではシステム制御コプロセッサ用に定義されている VA => PA アドレス変換をサポートする必要があります。

ARMv6 では、たとえばレベル 1 キャッシュでライトスルー、レベル 2 キャッシュでライトバックをサポートするような、メモリ階層にわたって 2 つのキャッシュポリシーをサポートする、内部と外部の属性の概念がメモリ管理に導入されています。

MMU での内部と外部の属性については、P. B4-11 *「C、B、TEX のエンコード」*を参照して下さい。これらの属性は、各ページについて定義されます。これらの属性は、メモリの異なる領域について、異なるキャッシュレベルのキャッシュポリシーを制御するために使用されます。内部属性は、レベル 1 のキャッシュポリシーの定義に使用されます。実装では、内部属性と外部属性を使用して、実装定義の手法で他のレベルのキャッシュポリシーを記述できます。

レベル 2 キャッシュのシステム制御コプロセッサ機構については、P. B6-29 *「キャッシュの追加レベル」*を参照して下さい。

レベル 2（またはそれ以上）のキャッシュの実装を検討する場合、この分野は開発が継続中なため、ARM と密接な協力のうえでシステム設計を行うことをお勧めします。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.6 CP15 のレジスタ

ARMv6 では、キャッシュのタイプはシステムコプロセッサのレジスタ 0 で決定され、レジスタ 1、7、9 により制御されます。以前のアーキテクチャバリエーションでは、これらの機能のどれを搭載するかは実装定義です。すべてのレジスタは、特に記載のない限り特権モードでのみアクセス可能です。実装されていないレジスタは未定義です。

——— 注 ———

ARMv6 以前は、システムコプロセッサは必須ではありませんでした。以前のアーキテクチャバリエーションでは、機能のサブセットを実装している場合や、実装定義の方法で使用していることがあります。新しい機能が追加されている部分や、既存の機能の定義が変更されている部分では、明確にそのことが説明されています。

B6.6.1 レジスタ 0: キャッシュタイプ

キャッシュのサイズと構成は実装固有です。このレジスタは読み出し専用で、キャッシュの構成とサイズを示しています。オペレーティングシステムはこの情報を使用して、キャッシュのクリーニングやロックダウンなどの操作を実行する方法を判定できます。opcode_2 フィールドを 1 に設定して CP15 のレジスタ 0 を読み出すと、キャッシュタイプレジスタにアクセスできます。

```
MRC p15, 0, Rd, c0, c0, 1 ; returns Cache Type register
```

キャッシュタイプレジスタは、キャッシュに関する次の詳細を示しています。

- キャッシュが統合キャッシュか、分離キャッシュか
- サイズ、ライン長、アソシアティビティ
- ページマッピングの条件（ARMv6 のみ）
- ライトスルーキャッシュかライトバックキャッシュか
- 効率的にクリーニングする方法（ライトバックキャッシュの場合）
- キャッシュのロックダウンがサポートされているかどうか

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.6.2 キャッシュタイプレジスタ

キャッシュタイプレジスタは、キャッシュに関する次の詳細を示しています。

- キャッシュが統合キャッシュか、命令キャッシュとデータキャッシュが分離されているか
- キャッシュのサイズ、ライン長、アソシアティビティ
- ライトスルーキャッシュかライトバックキャッシュか
- キャッシュのクリーニングとロックダウン機能

キャッシュタイプレジスタのフォーマットは次のとおりです。

31 29	28 25	24	23 12	11 0
0 0 0	ctype	S	Dsize	Isize

ctype S ビットと Dsize および Isize フィールドでは指定されないキャッシュの詳細を指定します。エンコードの詳細については、表 B6-1 を参照して下さい。表に指定されていない値はすべて、将来拡張用に予約されています。

S ビット キャッシュが統合キャッシュか（S == 0）、命令キャッシュとデータキャッシュが分離されているか（S == 1）を示します。S == 0 の場合、Isize フィールドと Dsize フィールドの両方が統合キャッシュのサイズを示し、この 2 つの値は同一の必要があります。

Dsize データキャッシュ（S == 0 の場合は統合キャッシュ）のサイズ、ライン長、アソシアティビティを示します。エンコードの詳細については、P. B6-15 *「キャッシュサイズフィールド」*を参照して下さい。

Isize 命令キャッシュ（S == 0 の場合は統合キャッシュ）のサイズ、ライン長、アソシアティビティを示します。エンコードの詳細については、P. B6-15 *「キャッシュサイズフィールド」*を参照して下さい。

表 B6-1 キャッシュタイプの値

ctype フィールド	方式	キャッシュクリーニング	キャッシュロックダウン
0b0000	ライトスルー	不要	未サポート
0b0001	ライトバック	データブロックの読み出し	未サポート（ARMv6 では推奨されません）
0b0010	ライトバック	レジスタ 7 の操作	未サポート（ARMv6 では推奨されません）
0b0110	ライトバック	レジスタ 7 の操作	フォーマット A
0b0111	ライトバック	レジスタ 7 の操作	フォーマット B（ARMv6 では推奨されません）
0b1110	ライトバック	レジスタ 7 の操作	フォーマット C
0b0101	ライトバック	レジスタ 7 の操作	フォーマット D

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ctype == 0b0001 でエンコードされる、ライトバックキャッシュをクリーニングする*データブロック読み出し*メソッドは、キャッシュと等しいサイズで、キャッシュに存在していないことが既知のシーケンシャルなデータブロックをロードする操作です。このメソッドが適切なのは、この操作でキャッシュ全体の再ロードが行われることが保証されているキャッシュ構成の場合のみです。たとえば、ダイレクトマップされたキャッシュや、ある種のラウンドロビン置換方式を採用しているキャッシュは、通常この性質を備えています。

——— 注 ———

この方式でキャッシュのクリーニングを行うのは、キャッシュタイプレジスタの ctype == 0b0001 の場合か、実装ドキュメントでこの実装についてこの方式が有効であることが明記されている場合*のみ*使用する必要があります。

他のライトバックキャッシュをクリーニングするために使用されるレジスタ 7 の操作の詳細については、P. B6-19 *「レジスタ 7: キャッシュ管理機能」*を参照して下さい。

P. B6-14 表 B6-1 で示されているキャッシュのロックダウンとフォーマットの説明については、P. B6-31 *「レジスタ 9: キャッシュのロックダウン機能」*を参照して下さい。

B6.6.3 キャッシュサイズフィールド

キャッシュタイプレジスタの Dsize と Isize フィールドは、次のような同じフォーマットを使用しています。

11	10	9–6	5–3	2	1–0
P	0	size	assoc	M	len

ビット [11]（P ビット）は、仮想アドレスのビット [13:12] に関するページ割り当てに制限が存在するかどうかを示しています。

0 制限なし。

1 制限が適用されます。詳細については、P. B6-11 *「ページテーブルのマッピングの制限」*を参照して下さい。

ビット [10] は将来の拡張用に予約されています。

キャッシュのサイズは、P. B6-16 表 B6-2 に示すように、size フィールドと M ビットにより判定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B6-2 キャッシュサイズ

サイズ フィールド	M == 0 の場合の サイズ	M == 1 の場合の サイズ
0b0000	0.5KB	0.75KB
0b0001	1KB	1.5KB
0b0010	2KB	3KB
0b0011	4KB	6KB
0b0100	8KB	12KB
0b0101	16KB	24KB
0b0110	32KB	48KB
0b0111	64KB	96KB
0b1000	128KB	192KB

キャッシュのライン長は、表 B6-3 に示すように、len フィールドにより判定されます。

表 B6-3 キャッシュのライン長

len フィールド	キャッシュのライン長
0b00	2 ワード（8 バイト）
0b01	4 ワード（16 バイト）
0b10	8 ワード（32 バイト）
0b11	16 ワード（64 バイト）

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

キャッシュのアソシアティビティは、表 B6-4 に示すように、assoc フィールドとアソシアティビティ修飾子（M ビット）により判定されます。

表 B6-4 キャッシュのアソシアティビティ

assoc フィールド	**M == 0 の場合のアソシアティビティ**	**M == 1 の場合のアソシアティビティ**
0b000	1 ウェイ（直接マップ）	キャッシュは存在しない
0b001	2 ウェイ	3 ウェイ
0b010	4 ウェイ	6 ウェイ
0b011	8 ウェイ	12 ウェイ
0b100	16 ウェイ	24 ウェイ
0b101	32 ウェイ	48 ウェイ
0b110	64 ウェイ	96 ウェイ
0b111	128 ウェイ	192 ウェイ

*キャッシャ不在*のエンコードは、キャッシュサイズフィールドの他のデータすべてに優先します。

キャッシュ不在（assoc == 0b000、M == 1）以外の場合、P. B6-6「*キャッシュサイズ*」で定義されているように、次の式を使用して LINELEN、ASSOCIATIVITY、NSETS の値を判定できます。

```
LINELEN         = 1 << (len+3)                /* In bytes */

MULTIPLIER      = 2 + M
ASSOCIATIVITY   = MULTIPLIER << (assoc-1)

NSETS           = 1 << (size + 6 - assoc - len)
```

これらの値を掛け合わせると、次のようにキャッシュ全体のサイズを算出できます。

```
CACHE_SIZE      = MULTIPLIER << (size+8)      /* In bytes */
```

注

キャッシュ長フィールドで (size + 6 - assoc - len) < 0 の値は、ライン長、アソシアティビティ、キャッシュ全体のサイズの不可能な組み合わせに対応しているため、無効です。このため、NSET の式が負のシフト量となることはありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.6.4 レジスタ 1: キャッシュとライトバッファの制御ビット

システム制御コプロセッサのレジスタ 1 にある以下のビットは、キャッシュとライトバッファの制御に使用されます。

C（ビット **[2]**） 統合キャッシュが使用されている場合、統合キャッシュの許可 / 禁止ビットとなります。分離キャッシュが使用されている場合、データキャッシュの許可 / 禁止ビットとなります。どちらの場合も、値の意味は次のとおりです。

0 = キャッシュは禁止されています。

1 = キャッシュは許可されています。

キャッシュが実装されていない場合、このビットからは *0 が読み出され* (RAZ)、書き込みは無視されます。キャッシュが実装されている場合、このビットに 0 を書き込むことでキャッシュを禁止できる必要があります。

リセット時の C ビットの値は 0 の必要があります。ARMv6 以前では、動作が異なる可能性があります。

W（ビット **[3]**） ライトバッファの許可 / 禁止ビット。

0 = ライトバッファは禁止されています。

1 = ライトバッファは許可されています。

統合キャッシュが使用されているか、命令キャッシュが実装されていない場合、このビットからは 0 が読み出され、書き込みは無視されます。書き込みバッファを禁止できない場合、このビットから 1 が読み出され、書き込みは無視されます。

I（ビット **[12]**） 分離キャッシュが使用されている場合、命令キャッシュの許可 / 禁止ビットとなります。

0 = キャッシュは禁止されています。

1 = キャッシュは許可されています。

統合キャッシュが使用されているか、命令キャッシュが実装されていない場合、このビットからは 0 が読み出され、書き込みは無視されます。命令キャッシュが実装されている場合、このビットに 0 を書き込むことでキャッシュを禁止できる必要があります。

リセット時の I ビットの値は 0 の必要があります。ARMv6 以前では、動作が異なる可能性があります。

RR（ビット **[14]**） キャッシュで、ワーストケースパフォーマンスがより予測しやすい別の置換方式を使用できる場合、このビットにより選択します。

0 = 通常の置換方式（ランダム置換など）

1 = 予測可能な置換方式（ラウンドロビン置換など）

リセット時の RR ビットの値は 0 の必要があります。ARMv6 以前では、動作が異なる可能性があります。

RR ビットの値と置換手法の関連は実装定義です。

システム制御コプロセッサのレジスタ 1 の完全な説明については、P. B3-12 *「制御レジスタ」* を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.6.5 レジスタ 7: キャッシュ管理機能

システム制御コプロセッサのレジスタ 7 は書き込み専用レジスタで、L1 キャッシュとライトバッファの制御に使用されます。また、プリフェッチバッファや分岐先キャッシュが存在する場合、それらに同様のいくつかの機能を実装するために使用され、*割り込み待ち*クロック制御機能の実装にも使用されます。P. B6-21 表 B6-6 は、利用可能な機能の一覧です。

P. B6-21 表 B6-6 に示すレベル 1 キャッシュの保守操作は、次の命令フォーマットを使用して起動されます。

```
MCR p15,0,<Rd>,c7,<CRm>,<opcode2>
```

P. B6-21 表 B6-6 に示されていない `<CRm>` と `<opcode2>` の値の組み合わせをレジスタ 7 に書き込んだ場合、結果は予測不能です。

CP15 のレジスタ 7 の操作のほとんどは、特権モードでのみ実行可能です。P. B6-21 表 B6-6 で c とマークされている一部の命令は、ユーザモードでも実行できます。特権操作をユーザモードで実行しようとすると、未定義命令例外が発生します。

P. B6-21 表 B6-6 で使用されている用語の意味は次のとおりです。

クリーニング
: ライトバックのデータキャッシュに適用され、ストアされたデータがキャッシュラインに含まれており、メインメモリに書き出されていない場合、メインメモリに書き出し、ラインをクリーンとしてマークすることを意味します。

無効化
: キャッシュライン（またはキャッシュのすべてのライン）を無効としてマークすることを意味します。このラインについて、アドレスに再割り当てが行われるまではキャッシュヒットは発生しません。

  ライトバックのデータキャッシュの場合、特に示されていない限り、この操作にはキャッシュラインのクリーニングは含まれません。

プリフェッチ
: 指定の仮想アドレスのロケーションがアボートを発生しなければ、そのアドレスの内容をメモリキャッシュラインにロードし、キャッシュ可としてマークします。プリフェッチでアボートが発生した場合（MMU または MPU によって）、この操作はメモリにアクセスしないことが保証されます。

  ARMv6 では、仮想アドレスについてのアライメントの条件はありません。ARMv6 以前では、アドレスがキャッシュラインでアラインされている必要があります。この操作は、フォーマット C のロックダウンを使用するキャッシュでサポートされている必要があります。詳細については P. B6-14 表 B6-1 を参照して下さい。それ以外の場合、この操作は実装定義です。

データ同期バリア
: 以前はデータ書き込みバリア、*DataWriteBarrier*（*DWB*）と呼ばれていたものです。

  DSB。ARMv6 の新しい定義については、P. B2-18「*DataSynchronizationBarrier (DSB) CP15 レジスタ 7*」を参照して下さい。

  データ同期バリアは、特権モードとユーザモードのどちらからでも実行できます。

データメモリバリア
: DMB。ARMv6 で導入された機能です。詳細については、P. B2-18「*DataMemoryBarrier (DMB) CP15 レジスタ 7*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

DMB は、特権モードとユーザモードのどちらからでも実行できます。

割り込み待ち

ARM を低電力ステートに置き、割り込みまたはデバッグ要求が発生するまで以後の実行を停止します。割り込みとデバッグイベントが発生すると、割り込みがマスクされているかどうかにかかわらず ARM プロセッサは常に動作を再開します。デバッグイベントには、デバッグが有効になっている必要があります。

割り込みが発生した場合、`MCR` 命令が完了し、次の命令が実行されるか（割り込みイベントで割り込みがマスクされている場合）、IRQ か FIQ ハンドラが通常に開始されます。R14_irq または R14_fiq の復帰リンクには `MCR` 命令のアドレス + 8 が含まれているため、割り込みから復帰する通常の命令（`SUBS PC,R14,#4`）を使用して、`MCR` の次の命令に復帰できます。

プリフェッチフラッシュ

命令プリフェッチバッファをフラッシュすると、その命令の実行後に、プログラムでその命令の後にあるすべての命令が、L1 キャッシュや TCM を含むメモリシステムからフェッチされます。この操作は、自己変更型のコードが正しく実行されることを保証するために便利です。

プリフェッチフラッシュは、特権モードとユーザモードのどちらからでも実行できます。

データ　　レジスタ 7 に書き込まれる値。これは、`MCR` 命令のレジスタ `<Rd>` で指定された値です。

ARMv6 以降では、データが仮想アドレスで指定される場合、キャッシュラインでアラインされている必要はありません。この操作のために、キャッシュでアドレスの参照が行われます。キャッシュミスの場合、無効化とクリーニングの操作は無効です。対応するエントリが TLB にない場合、これらの命令によりハードウェアのページテーブルウォークが発生することがあります。

第 B8 章*「高速コンテキストスイッチ拡張機能」*で説明されている*高速コンテキストスイッチ拡張機能*（FCSE）が使用されている場合、このセクションにある MVA への参照はすべて修飾仮想アドレス、つまり FCSE 変換の結果として生成されるアドレスを意味し、それ以上の変換は実行されません。修飾仮想アドレスは、変換が行われる前に、非グローバルページの ASID と結合されます。P. B8-2 *「FCSE の概要」*で説明されているように、FCSE を非グローバルページとともに使用した場合の動作は予測不能です。

単一ラインのキャッシュ制御操作をループで使用し、特定の範囲のアドレスに関連するすべてのキャッシュラインをクリーニング、無効化、または両方を行うことができます。

データがセット / ウェイで指定される場合、そのデータはその操作が適用されるキャッシュラインの属するキャッシュセットと、セット内のウェイ番号を指定します。

この種のループ操作を使用して、キャッシュ全体をクリーニングや無効化することもできます。

セット / ウェイデータのフォーマットを P. B6-21 表 B6-5 に示します。ここで、L、A、S はキャッシュサイズパラメータである LINELEN、ASSOCIATIVITY、NSETS の 2 を底とする対数で、A については整数に切り上げられた値です。これらのパラメータは、キャッシュタイプレジスタにあります。NSETS は、他の 2 つのパラメータを使用して、サイズ情報から派生します。データの TC フィールドは、P. B7-6 *「スマートキャッシュの動作」*に示すように、そのデータがキャッシュ、またはスマートキャッシュとして構成されているいずれかの TCM に適用されるかどうかを示しています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B6-5 セット / ウェイデータレジスタの値

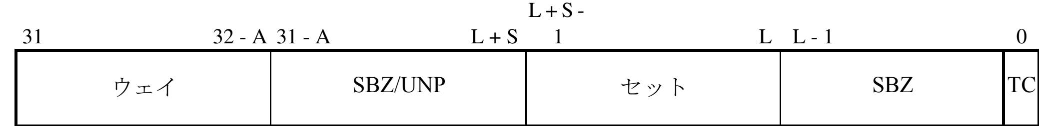


表 B6-6 レジスタ 7: キャッシュ制御と類似機能

<CRm>	<opcode2>	機能	データ	
c0	4	割り込み待ち	SBZ	
c5	0	命令キャッシュ全体を無効化する（適用される場合、分岐先キャッシュをフラッシュする）	SBZ	a
c5	1	命令キャッシュラインを無効化する	MVA[b]	a
c5	2	命令キャッシュラインを無効化する	セット / ウェイ	a
c5	4	プリフェッチバッファをフラッシュする（PrefetchFlush）	SBZ	c
c5	6	分岐先キャッシュ全体をフラッシュする（適用される場合）	SBZ	
c5	7	分岐先キャッシュのエントリをフラッシュする（適用される場合）	MVA[b]	
c6	0	データキャッシュ全体を無効化する	SBZ	d
c6	1	データキャッシュラインを無効化する	MVA[b]	d
c6	2	データキャッシュラインを無効化する	セット / ウェイ	d
c7	0	命令キャッシュとデータキャッシュの両方、または統合キャッシュを無効化する（適用される場合、分岐先キャッシュをフラッシュする）	SBZ	e
c7	1	統合キャッシュラインを無効化する	MVA[b]	f
c7	2	統合キャッシュラインを無効化する	セット / ウェイ	f
c10	0	データキャッシュ全体をクリーニングする	SBZ	d
c10	1	データキャッシュラインをクリーニングする	MVA[b]	d
c10	2	データキャッシュラインをクリーニングする	セット / ウェイ	d
c10	3	テストとクリーニング（オプション）	-	

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B6-6 レジスタ 7: キャッシュ制御と類似機能（続き）

<CRm>	<opcode2>	機能	データ	
c10	4	データ同期バリア（従来はライトバッファドレイン）	SBZ	c
c10	5	データメモリバリア（ARMv6 から導入。以前のアーキテクチャバリエーションにも適用される可能性あり）		c
c11	0	統合キャッシュ全体をクリーニングする	SBZ	f
c11	1	統合キャッシュラインをクリーニングする	MVA[b]	f
c11	2	統合キャッシュラインをクリーニングする	セット / ウェイ	f
c13	1	命令キャッシュラインをプリフェッチする（オプション）[g]	MVA[b]	
c14	0	データキャッシュ全体をクリーニングし無効化する	SBZ	d
c14	1	データキャッシュラインをクリーニングし無効化する	MVA[b]	d
c14	2	データキャッシュラインをクリーニングし無効化する	セット / ウェイ	d
c14	3	テスト、クリーニング、無効化（オプション）	-	
c15	0	統合キャッシュ全体をクリーニングし無効化する	SBZ	f
c15	1	統合キャッシュラインをクリーニングし無効化する	MVA[b]	f
c15	2	統合キャッシュラインをクリーニングし無効化する	セット / ウェイ	f

a. 分離命令キャッシュにのみ適用
b. *修飾仮想アドレス*（MVA）については、P. B8-3「*修飾仮想アドレス*」の説明を参照して下さい。
c. ユーザモードで利用可能
d. 分離データキャッシュにのみ適用
e. 統合キャッシュと分離キャッシュに適用される
f. 統合キャッシュにのみ適用
g. フォーマット C ロックダウンでは必須、それ以外の場合は実装定義

特記する場合を除き、あるキャッシュ構成用の、P. B6-21 表 B6-6 にあるすべての機能は、その構成を使用する実装により搭載されている必要があります。他の機能は無効の必要があります。

キャッシュの無効化操作は、キャッシュ内でロックされたものも含め、すべてのキャッシュロケーションに適用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

データ（または統合）キャッシュ全体のクリーニングと無効化操作

CP15 のレジスタ 7 は、データ（または統合）キャッシュ全体のクリーニング操作と、データ（または統合）キャッシュ全体のクリーニングと無効化を指定します。これらの操作の途中で割り込みが発生した場合、割り込み時にキャプチャされる R14 の値はキャッシュのクリーニング操作を開始した命令のアドレス + 4 を指しています。これにより、割り込みからの標準の復帰機構で操作を再開できます。

特定の操作についてキャッシュがクリーンである（またはクリーンかつ無効である）ことが重要な場合、その操作のためにキャッシュのクリーニング（またはクリーニングと無効化）を実行する命令のシーケンスでは、禁止になっていない割り込みについて常に考慮しておく必要があります。これは、以前にクリーニングされたキャッシュブロックに、割り込みによって書き込みが行われる可能性があるためです。この理由から、キャッシュダーティステータスレジスタは、キャッシュに対して最後にクリーニング操作が開始された後キャッシュに書き込みが行われたかどうかを示しています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

キャッシュダーティステータスレジスタは読み出し専用です。このレジスタにアクセスするには、次の命令を使用します。

```
MRC p15, 0, Rd, c7, c10, 6
```

キャッシュダーティステータスレジスタのフォーマットは表 B6-7 のとおりです。

表 B6-7 キャッシュダーティステータスレジスタ

C（ビット [0]） キャッシュダーティステータス

0 最後のキャッシュクリーニングまたはリセットによりキャッシュが正常にクリーニングされた後で、書き込みがキャッシュにヒットしていないことを示します。

1 キャッシュにダーティデータが含まれている可能性があります。

キャッシュがクリーンであるかどうかはキャッシュダーティステータスレジスタを調べることで行え、この操作が割り込みの禁止中に行われた場合、以後の操作ではキャッシュがクリーンであることを前提とできます。次のシーケンスは、この手法を示したものです。

```
        ; interrupts are assumed to be enabled at this point

Loop1   MOV R1, #0
        MCR CP15, 0, R1, C7, C10, 0  ; Clean (for Clean & Invalidate
                                     ; use "C7, C14, 0")
        MRS R2, CPSR                 ; Cache
        CPSID iaf                    ; Disable interrupts
        MRC CP15, 0, R1, C7, C10, 6  ; Read Cache Dirty Status Register
        ANDS R1, R1, #01             ; Check if it is clean
        BEQ UseClean
        MSR CPSR, R2                 ; Re-enable interrupts
        B Loop1                      ; Clean the cache again
UseClean    Do_Clean_Operations      ; Perform whatever operation relies on
                                     ; the cache being clean/invalid.
                                     ; To reduce impact on interrupt latency,
                                     ; this sequence should be short
        MCR CP15, 0, R1, C7, C6, 0   ; can use this "invalidate all" command to
                                     ; optionally invalidate a "clean" loop.
        MSR CPSR, R2                 ; Re-enable interrupts
```

——— 注 ———

このルーチン全体では、長いキャッシュクリーニング操作が割り込みを許可した状態で実行されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

テストとクリーニング操作

別のクリーニング（および、クリーニングと無効化）手法は、ARMv5 ではオプションです。この方式では、プログラムカウンタをデスティネーションとして MRC 命令を実行することで、データキャッシュ全体を効率的にクリーニング、またはクリーニングして無効化できます。グローバルキャッシュダーティステータスビットが Z フラグに書き込まれます。デスティネーションに R15 を指定した MRC 命令は、条件フラグを更新します。詳細については、P. A4-71「*MRC*」を参照して下さい。命令の繰り返しごとにいくつのラインをテストするかは実装定義です。

この方式でデータキャッシュ全体をクリーニングするには、次のコードループが使用できます。

```
tc_loop  MRC p15, 0, r15, c7, c10, 3       ; test and clean
         BNE tc_loop
```

この方式でデータキャッシュ全体をクリーニングして無効化するには、次のコードループが使用できます。

```
tci_loop  MRC p15, 0, r15, c7, c14, 3       ; test, clean and invalidate
          BNE tci_loop
```

B6.6.6 CP15 のレジスタ 7 を使用するブロック転送操作

表 B6-8 に示すブロック転送操作は、CP15 のレジスタ 7 を使用してオプションでサポート可能です。ブロック操作は、ARMv6 でアーキテクチャに導入されたものです。この操作が実装されていない場合、未定義命令例外が発生する必要があります。ブロック操作の許可される組み合わせは次のとおりです。

- すべて（4 つ）の操作
- クリーニング、クリーニングと無効化、無効化の各操作
- なし

スマートキャッシュ（P. B7-6「*スマートキャッシュの動作*」参照）動作をサポートする実装では、範囲のクリーニングと無効化の操作を実装する必要があります。

表 B6-8 ブロック転送操作

操作	ブロッキング[a] または ノンブロッキング	命令またはデータ	ユーザまたは特権	例外の動作
範囲のプリフェッチ	ノンブロッキング	命令またはデータ	ユーザ / 特権	なし
範囲のクリーニング	ブロッキング	データのみ	ユーザ / 特権	データアボート
範囲のクリーニングと無効化	ブロッキング	データのみ	特権	データアボート
範囲の無効化	ブロッキング	命令またはデータ	特権	データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

a. キャッシュのある範囲のアドレスについてクリーニング、無効化、または両方を行うキャッシュのブロック転送操作がブロッキング操作です。以後の命令は、この操作が完了するまで実行できません。ノンブロッキング操作では、操作の完了前に以後の命令の実行が許可されます。ノンブロッキング操作で例外が発生した場合、コアに例外は通知されません。これによって、実装は正確なプロセッサステートを保持する必要なしに、ノンブロッキング操作の実行中に以後の命令を終了できます。

それぞれの範囲操作は、MCRR 命令を使用して開始されます。ブロック開始アドレスとブロック終了アドレスを指定するため、2 つのレジスタのデータが使用されます。すべてのブロック操作は、ブロック開始アドレスとブロック終了アドレスを含むアドレス範囲を含むキャッシュ（またはスマートキャッシュ）ライン上で実行されます。ブロック開始アドレスがブロック終了アドレスよりも大きい場合、結果は予測不能です。

同時にサポートされるブロック転送は 1 つだけです。最初のブロック転送が進行中に 2 番目のブロック転送を開始しようとすると、最初のブロック転送が破棄され、2 番目のブロック転送が開始されます。ブロック転送ステータスレジスタは、ブロック転送が進行中かどうかを示しています。このレジスタを使用して、望ましくない待ち状態を無くせます。コンテキストスイッチ時にはブロック転送を停止することが前提です。

すべてのブロック転送は割り込み可能です。ブロック転送に割り込みが発生した場合、キャプチャされる R14 の値はブロック操作を開始した命令のアドレス + 4 を指しています。これにより、割り込みからの標準の復帰機構で操作を再開できます。

パフォーマンス上の理由から、ノンブロックの範囲プリフェッチ命令の実行中に、実装が以後の命令の実行を許可することが期待されます。このような実装では、割り込み時にキャプチャされる R14 の値は、以後の命令ストリームで割り込みに対して提示される実行ステートにより決定されます。ただし、範囲プリフェッチ命令をブロッキング操作として扱う実装では、前の段落で説明したように R14 をキャプチャする必要があります。

範囲プリフェッチ操作の実行中に FCSE PID（P. B8-7「*CP15 のレジスタ*」参照）が変更された場合、その変更がどの時点で範囲プリフェッチから可視になるかは予測不能です。

例外の動作

ブロックキングなブロック転送では、有効なページテーブルエントリがフェッチできない場合に変換フォルトのデータアボートが発生します。CP15 の FAR はフォルトの発生したアドレスを、CP15 の FSR はフォルトの原因を示しています。

範囲プリフェッチ操作でどのようなフォルトが発生した場合も、操作はエラーを通知せずに失敗します。

レジスタのエンコード

ブロック操作は、P. B6-27 表 B6-9 に示すように、CP15 のレジスタ 7 命令を使用してサポートされます。これらの操作は、MCRR 命令でのみ実行可能です。これらのレジスタに対して、他の操作はすべて無視されます。

命令のフォーマットは次のとおりです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

```
MCRR p15, Opcode, Rd, Rn, CRm
```

表 B6-9 MCRR を使用した拡張キャッシュ制御操作

<CRm>	オペコード	機能	Rn データ（VA [a]）	Rd データ（VA [a]）
c5	0	命令キャッシュの範囲を無効化する[b]	開始アドレス	終了アドレス
c6	0	データキャッシュの範囲を無効化する [b]	開始アドレス	終了アドレス
c12	0	データキャッシュの範囲をクリーニングする[c]	開始アドレス	終了アドレス
c12	1	範囲の命令をプリフェッチする [c]	開始アドレス	終了アドレス
c12	2	範囲のデータをプリフェッチする [c]	開始アドレス	終了アドレス
c14	0	データキャッシュの範囲をクリーニングして無効化する [b]	開始アドレス	終了アドレス

a. 高速コンテキストスイッチ拡張機能（第 B8 章*「高速コンテキストスイッチ拡張機能」*参照）による修飾が行われる前の真の仮想アドレス。このアドレスは、FCSE ロジックにより変換されます。
b. 特権モードでのみアクセス可能。この操作をユーザモードで実行しようとすると、未定義命令例外が発生します。
c. ユーザモードと特権モードの両方でアクセス可能。

それぞれの範囲操作は、開始アドレスと終了アドレスの間（それら自体を含む）のキャッシュ（またはスマートキャッシュ）ラインの間で動作します。

MCRR命令で渡される開始アドレスと終了アドレスの値のフォーマットを表B6-10に示します。ここで、L はキャッシュサイズパラメータ LINELEN の 2 を底とする対数です。最下位ビットは無視されるため、転送はプログラムされたアドレスにわたるライン長の複数倍に自動的に調整されます。

表 B6-10 ブロックアドレスレジスタ

31 … L	L-1 … 0
仮想アドレス	IGN

開始アドレス　**仮想アドレス（ビット [31:L]）**

ブロック転送の最初の仮想アドレス

終了アドレス　**仮想アドレス（ビット [31:L]）**

ブロック転送が停止する仮想アドレス（このアドレスは、ブロック転送により処理される最後のアドレスを含むラインの先頭です）。

注

これらのブロック操作のみが、真の仮想アドレスを使用します。他のアドレスベースのキャッシュ操作はすべて、MVA を使用します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ブロック転送の管理をサポートするため、CP15 のレジスタ 7 に 2 つの追加操作が用意されています。

StopPrefetchRange `MCR p15, 0, Rd, c7, c12, 5 ; Rd SBZ`

PrefetchStatus `MRC p15, 0, Rd, c7, c12, 4 ; Rd returns the status`

どちらの操作も、ユーザモードと特権モードの両方でアクセス可能です。すべてのブロック操作は排他的で、同時にアクティブとなれる操作は 1 つだけなため、PrefetchStatus 操作は最後に発行された Prefetch 要求（命令またはデータ）のステータスを返します。

ブロック転送ステータスレジスタのフォーマットを表 B6-11 に示します。

表 B6-11 ブロック転送ステータスレジスタ

R（ビット **[0]**） ブロックプリフェッチが実行中かどうかを示します。

0 プリフェッチが実行されていません。

1 プリフェッチが実行されています。

B6.6.7 スマートキャッシュとして構成された TCM のキャッシュクリーニングと無効化の操作

すべてのキャッシュラインとブロックのクリーニングおよび無効化操作は、CP15 のレジスタ 7 に定義されているように、スマートキャッシュとして構成された TCM 領域を含む仮想アドレスを基礎としています。詳細については P. B7-6 *「スマートキャッシュの動作」*を参照して下さい。

セット / ウェイ操作は、スマートキャッシュとして動作する TCM についてもサポートされています。この場合、ウェイ番号が TCM 番号となり、セット番号の意味は変更されません。これらの操作がキャッシュに適用される場合と TCM に適用される場合を区別するために、P. B6-21 表 B6-5 に示すようにセット / ウェイデータレジスタの最下位ビットが使用されます。

TC（ビット **[0]**）TCM ビット。このレジスタがキャッシュではなく TCM を指していることを示します。

0 レジスタはキャッシュを指しています。

1 レジスタは TCM を指しています。

スマートキャッシュとして動作している TCM のライン長は、キャッシュタイプレジスタで定義されているキャッシュライン長と同じである必要があります。

キャッシュ全体を無効化、クリーニング、クリーニングと無効化する操作は、スマートキャッシュとして動作している TCM に対しては無効です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.6.8 キャッシュの追加レベル

すべてのシステム制御コプロセッサ操作（MCRR フォーマットを使用する範囲操作を除く）は、次の命令を使用してアクセスされます。

```
MCR p15,<opcode1>,<Rd>,<CRn>,<CRm>,<opcode2>
MRC p15,<opcode1>,<Rd>,<CRn>,<CRm>,<opcode2>
```

すべての一般およびレベル 1 操作は、<opcode1 >== 0 の値を使用するものとして定義されます。レジスタ 7 と関連付けられる一般的なキャッシュ操作では <CRn> == 7、レジスタ 9 と関連付けられるロック操作では <CRn> == 9 です。

<opcode1> == 1 は、レベル 2 キャッシュ操作用に予約されています。一般に、サポートされる操作は、レベル 1 で <CRm> と <opcode2> の組み合わせで示される形式を踏襲します。

したがって、レベル 2 の一般的なキャッシュ操作は次の形式の命令でアクセスされます。

```
MCR p15,1,<Rd>,c7,<CRm>,<opcode2>
MRC p15,1,<Rd>,c7,<CRm>,<opcode2>
```

レベル 2 のキャッシュロック操作は次の形式の命令でアクセスされます。

```
MCR p15,1,<Rd>,c9,<CRm>,<opcode2>
MRC p15,1,<Rd>,c9,<CRm>,<opcode2>
```

VA => PA の変換サポートが必須な場合、レベル 1 のキャッシュ操作と同様に、対応する操作がレジスタ 7 に定義されています。

```
MCR p15,0,<Rd>,c7,<CRm>,<opcode2>
MRC p15,0,<Rd>,c7,<CRm>,<opcode2>
```

将来の互換性のため、複数レベルのキャッシュを実装する場合は ARM との密接な協力のうえで設計することをお勧めします。

現在予約されている定義を P. B6-30 表 B6-12 に示します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 B6-12 レベル 2 キャッシュ操作用に予約されている定義

<opcode1>	<CRn>	<CRm>	<opcode2>	機能
1	7	5	x	L2 命令キャッシュの無効化操作
1	7	6	x	L2 データキャッシュの無効化操作
1	7	7	x	L2 統合キャッシュの無効化操作
1	7	10	x	L2 データキャッシュのクリーニング操作
1	7	11	x	L2 統合キャッシュのクリーニング操作
1	7	14	x	L2 データキャッシュのクリーニングと無効化操作
1	7	15	x	L2 統合キャッシュのクリーニングと無効化操作
1	9	5	x	L2 命令キャッシュのロック操作
1	9	6	x	L2 データキャッシュのロック操作
1	9	7	x	L2 統合キャッシュのロック操作
0	7	8	x	PA ルックアップ操作（実行、MCR のみ）
0	7	4	0	PA 値へのアクセス - 読み出し / 書き込み（書き込みはデバッグ用）

L2 キャッシュでは、最低でも次の操作セットを搭載することをお勧めします。

- アドレスによるキャッシュラインの無効化
- アドレスによるキャッシュラインのクリーニング
- セット / ウェイによるキャッシュラインのクリーニング
- セット / ウェイによるキャッシュラインのクリーニングと無効化

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B6.6.9 レジスタ 9: キャッシュのロックダウン機能

キャッシュは通常、データと命令への平均アクセス時間を短縮しますが、キャッシュの問題の 1 つは、ワーストケースアクセス時間は逆に増大するということです。これは、次のようないくつかの理由に起因します。

- キャッシュミスが発生したことをシステムが判定し、メインメモリへのアクセスを開始するまでに遅延がある。
- ライトバックキャッシュが使用されている場合、再割り当てされるキャッシュラインの内容をストアする必要があるため、さらに遅延が発生することがある。
- ARM プロセッサが要求するデータだけでなく、キャッシュライン全体がメインメモリからロードされる。

リアルタイムアプリケーションの場合、このワーストケースのアクセス時間の増大が大きな問題になります。

*キャッシュのロックダウン*は、この問題を軽減するためにほとんどの ARM メモリシステムに装備されている機能です。これは、重要なコードやデータ（優先度の高い割り込みルーチンや、そのアクセスするデータなど）をキャッシュにロードし、それらのデータを含むキャッシュラインが以降に再割り当てされないようにする方法です。これにより、以降にそれらのコードやデータに対して行われるアクセスは必ずキャッシュヒットとなり、迅速に完了することが保証されます。

ARM アーキテクチャは、フォーマット A、フォーマット B、フォーマット C、フォーマット D という 4 つのキャッシュのロックダウン機構のフォーマットをサポートしています。システム制御コプロセッサのキャッシュタイプレジスタ（CP15 のレジスタ 0）には、採用されているロックダウン機構についての情報が含まれています。詳細については、P. B6-14「*キャッシュタイプレジスタ*」を参照して下さい。

注

フォーマット B は ARMv6 では推奨されません。

フォーマット A、B、C はすべてキャッシュウェイに対して動作します（P. B6-5「*セットアソシアティビティ*」参照）。フォーマット D は、キャッシュエントリのロック機構です。

フォーマット A、B、C のロックダウンに適用される一般的な条件

CP15 のレジスタ 9 のロックダウンレジスタにアクセスするために使用される命令は次のとおりです。

```
MCR p15, 0, Rd, c9, c0, 0      ; write unified/data lockdown register
MRC p15, 0, Rd, c9, c0, 0      ; read unified/data lockdown register
MCR p15, 0, Rd, c9, c0, 1      ; write instruction lockdown register
MRC p15, 0, Rd, c9, c0, 1      ; read instruction lockdown register
```

LINELEN、ASSOCIATIVITY、NSETS はキャッシュサイズパラメータです。詳細については、P. B6-6「*キャッシュサイズ*」を参照して下さい。キャッシュウェイは、各キャッシュセットからそれぞれ 1 つのキャッシュラインで構成され、0 から ASSOCIATIVITY - 1 までのラベルが付けられます。フォーマット A、B、C はすべてロックダウンの単位（ロックダウンブロック）としてキャッシュウェイを使用します。キャッシュのロック方式では、1 から ASSOCIATIVITY - 1 までの任意の数のロックダウンブロックを使用できます。N 個のロックダウンブロックをロックダウンする場合、そのインデクスは 0 から N - 1 までで、ロックダウンブロック N から ASSOCIATIVITY - 1 までは通常のキャッシュ動作に使用できます。

キャッシュウェイベースのロックダウンの実装では、キャッシュ全体をロックダウンすることはできません。最低でも 1 つのキャッシュウェイブロックを、通常のキャッシュ動作用に残す必要があります。この制限に準拠しない場合、動作は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ロックダウンにおいては、ひとつのキャッシュウェイは一つの*ロックダウンブロック*として定義され、各ロックダウンブロックは各キャッシュセットからの 1 ラインで構成されます。ロックダウンブロックには、0 から ASSOCIATIVITY - 1 までのインデクスが付けられます。ロックダウンブロックのキャッシュラインは、ロックダウンブロックと同じ WAY 番号を持つように選択されます（P. B6-5 *「セットアソシアティビティ」*参照）。したがって、ロックダウンブロック ***n*** は、各キャッシュセットに含まれるインデクス *n* のキャッシュラインで構成されます。ここで、*n* は 0 から ASSOCIATIVITY - 1 までです。

各ロックダウンブロックは、各メモリキャッシュラインが異なるキャッシュセットと関連付けられていれば、NSETS 個のメモリキャッシュラインを保持できます。システムは、各ロックダウンブロックに NSETS 個の連続するメモリキャッシュラインのセットが含まれるように設計することをお勧めします。これは NSETS × LINELEN の連続するメモリロケーションで、キャッシュラインの境界から開始されます（このようなセットは、特定が容易で、各キャッシュセットに関連付けられたそれぞれ 1 つのキャッシュラインで構成されていることが保証されます）。

フォーマット A とフォーマット B のロックダウン

フォーマット A とフォーマット B では、任意のロックダウンブロックのウェイ数を保持できる幅を持つ WAY フィールドを使用します。この幅 W は ASSOCIATIVITY の 2 を底とする対数で、必要に応じて整数に切り上げられます。

フォーマット A のロックダウンレジスタの形式を表 B6-13 に示します。

表 B6-13 フォーマット A のロックダウンレジスタ

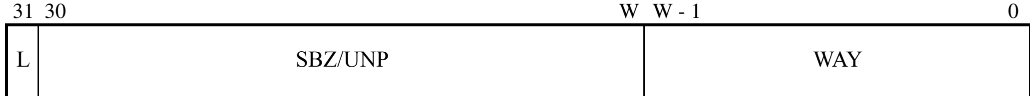
フォーマット A レジスタを読み出すと、最後に書き込まれた値が返されます。

フォーマット A レジスタへの書き込みは次のような効果があります。

- 各キャッシュセットにおける次のキャッシュミスは、そのキャッシュラインを、そのキャッシュセット中の指定された WAY で置換します。
- レジスタに再度書き込みが行われるまで、キャッシュの置換手法は、指定の、またはより高い番号の WAY を持つキャッシュラインのみを選択できるように制限されます。

フォーマット B のロックダウンレジスタの形式を表 B6-14 に示します。

表 B6-14 フォーマット B のロックダウンレジスタ



フォーマット B レジスタを読み出すと、最後に書き込まれた値が返されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

フォーマット B レジスタへの書き込みは次のような効果があります。

- L == 1 の場合、レジスタに再度書き込みが行われるまで、すべてのキャッシュミスは、そのキャッシュラインを、対応するキャッシュセット中の指定された WAY で置換します。
- L == 0 の場合、次の動作が実行されます。
  - 以前の L の値が 0 であり、以前の WAY の値が新しい値よりも小さい場合、動作は予測不能です。
  - 以前の L の値が 0 でない場合、レジスタに再度書き込みが行われるまで、キャッシュの置換手法は、指定の、またはより高い番号の WAY を持つキャッシュラインのみを選択できるように制限されます。

フォーマット A とフォーマット B のキャッシュのロックダウン手順

N 個のロックダウンブロックをロックダウンする手順は次のとおりです。

1. 割り込み禁止などの操作を行い、この手順の実行中にプロセッサ例外が発生しないことを保証します。なんらかの理由でこの操作を実行できない場合、呼び出される可能性がある例外ハンドラで使用されるすべてのコードとデータは、このプロシージャの手順 2 と手順 3 のために使用されるコードとデータとして扱う必要があります。
2. 命令キャッシュか統合キャッシュをロックダウンする場合、この手順により実行されるすべてのコードが、キャッシュ不可のメモリ領域にあることを確認します。
3. データキャッシュか統合キャッシュをロックダウンする場合、以下のコードで使用されるすべてのデータが、ロックダウンされるデータを除いて、キャッシュ不可のメモリ領域にあることを確認します。
4. ロックダウンされるデータや命令が、キャッシュ可なメモリ領域にあることを確認します。
5. 必要ならばキャッシュのクリーニング、キャッシュの無効化、または両方を使用し、ロックダウンされるデータや命令が、既にキャッシュに存在していないことを確認します。
6. i = 0 から N - 1 までの各 i について、次の処理を実行します。
   a. レジスタ 9 に WAY === i（フォーマット A と B の場合）、L == 1（フォーマット B の場合）を書き込みます。
   b. ロックダウンブロック i でロックダウンされる各キャッシュラインについて、次の処理を実行します。

      データキャッシュまたは統合キャッシュをロックダウンする場合、`LDR` 命令を使用してメモリキャッシュラインからワードをロードします。これにより、メモリキャッシュラインがキャッシュにロードされていることが保証されます。

      命令キャッシュをロックダウンする場合、レジスタ 7 の*プリフェッチ命令キャッシュライン*操作（`<CRm>` == c13、`<opcode2>` == 1）を使用して、メモリキャッシュラインをキャッシュにフェッチします。
7. レジスタ 9 に WAY == N（フォーマット A と B の場合）、L == 0（フォーマット B の場合）を書き込みます。

——— 注 ———

高速コンテキストスイッチ拡張機能（FCSE）（第 B8 章参照）を使用している場合、次の理由から手順 6b に注意が必要です。

- データキャッシュまたは統合キャッシュをロックダウンする場合、`LDR` 命令で使用するアドレスは FCSE による変更の対象となります。
- 命令キャッシュをロックダウンする場合、レジスタ 7 操作で使用するアドレスはデータと見なされ、FCSE による変更の対象となりません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

これによって発生する混乱の可能性を最小にするため、ロックダウン手順は次のように実行することをお勧めします。

- 最初に FCSE を禁止します（PID を 0 に設定します）。
- 必要であれば、適切な PID の値と、その使用する仮想アドレスの上位 7 ビットの論理和を取り、修飾仮想アドレス自体を生成します。

フォーマット A とフォーマット B のキャッシュのロック解除手順

キャッシュのロックダウンされた部分をロック解除するには、レジスタ 9 に WAY === 0（フォーマット A と B の場合）、L == 0（フォーマット B の場合）を書き込みます。

注

フォーマット B は ARMv6 では推奨されません。

フォーマット C のロックダウン

フォーマット C のキャッシュのロックダウンは、キャッシュウェイをベースとしたロックとは異なっています。この方式では、各キャッシュウェイについて許可 / 禁止を割り当てることができます。これによって、特定のアプリケーションにより発生するキャッシュの汚染や、重要な領域をキャッシュにロックする従来のロックダウン機能と比較して、より的確な制御が可能になります。

各キャッシュウェイのロッキングビットにより、通常のキャッシュ割り当て機構を使用してキャッシュウェイにアクセスできるかどうかが判定されます。

アソシアティビティの大きいキャッシュについては、キャッシュウェイ 0 から 31 までのみがロック可能です。

N ウェイキャッシュについて、最大 N - 1 ウェイをロックできます。これによって、通常のキャッシュライン置換が実行可能なことが保証されます。L == 0 であるキャッシュウェイが存在しない場合、キャッシュミス時の処理の動作は予測不能になります。

ロックダウンレジスタ（opcode2 の値により命令またはデータ）の 32 のビットにより、キャッシュウェイに関連付けられた L ビットが判定されます。

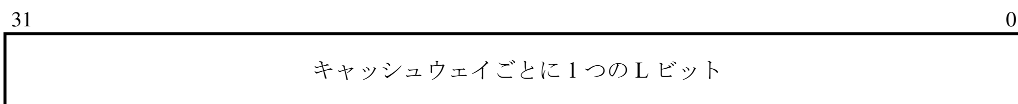
キャッシュロックダウンレジスタは、通常はリードモディファイライトのシーケンスで変更されます。たとえば、次のシーケンスは命令キャッシュのウェイ 0 の L ビットを 1 にセットします。

```
MRC p15, 0, Rn, c9, c0, 1
ORR Rn, Rn, 0x01
MCR p15, 0, Rn, c9, c0, 1    ; set way 0 L-bit for the Icache
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

キャッシュロックダウンレジスタのフォーマットを表 B6-15 に示します。

表 B6-15 フォーマット C のロックダウンレジスタ



ビット [31:0] キャッシュウェイごとの L ビット。あるキャッシュウェイが実装されていない場合、そのウェイの L ビットからは 1 が読み出され、書き込みは無視されます。各ビットは、対応するキャッシュウェイと関連しています。つまり、ビット N はウェイ N に関連しています。

0 キャッシュウェイへの割り当ては、標準の置換アルゴリズムにより決定されます（リセット時の状態）。

1 このキャッシュウェイには割り当ては実行されません。

フォーマット C のロックダウンレジスタは、キャッシュラインフィルを引き起こす可能性があるすべての未完了アクセスが完了したことが確実な場合のみ変更される必要があります。この理由から、ロックダウンレジスタの変更前にはデータ同期バリア命令を実行する必要があります。

フォーマット C キャッシュのロック手順

フォーマット C を使用して、*N* キャッシュウェイでキャッシュウェイ *i* をロックダウンする手順には、対象のキャッシュウェイ *i* 以外のキャッシュウェイに割り当てを不可能にする操作が含まれます。これは、データをキャッシュにロックするためのアーキテクチャ定義の方式です。

1. 割り込み禁止などの操作を行い、この手順の実行中にプロセッサ例外が発生しないことを保証します。なんらかの理由でこの操作を実行できない場合、呼び出される可能性があるすべての例外ハンドラで使用されるコードとデータすべては、この手順で手順 2 と手順 3 の目的に使用されるコードとデータとして扱う必要があります。

2. 命令キャッシュか統合キャッシュをロックダウンする場合、この手順により実行されるすべてのコードが、キャッシュ不可のメモリ領域（密結合メモリを含む）、または既にロックされているキャッシュウェイにあることを確認します。

3. データキャッシュか統合キャッシュをロックダウンする場合、以下のコードで使用されるすべてのデータが、ロックダウンされるデータを除いて、キャッシュ不可のメモリ領域（密結合メモリを含む）、または既にロックされているキャッシュウェイにあることを確認します。

4. ロックダウンされるデータや命令が、キャッシュ可なメモリ領域にあることを確認します。

5. 必要ならばキャッシュのクリーニング、キャッシュの無効化、または両方を使用し、ロックダウンされるデータや命令が、既にキャッシュに存在していないことを確認します。

6. レジスタ 9 に <CRm> == 0 を書き込み、同時にビット *i* を L == 0 に、他のすべてのウェイを L == 1 にセットします。これにより、対象のキャッシュウェイへの割り当てが可能になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

7. キャッシュウェイ *i* でロックダウンされる各キャッシュラインについて、次の操作を行います。
   - データキャッシュまたは統合キャッシュをロックダウンする場合、LDR 命令を使用してメモリキャッシュラインからワードをロードします。これにより、メモリキャッシュラインがキャッシュにロードされていることが保証されます。
   - 命令キャッシュをロックダウンする場合、レジスタ 7 のプリフェッチ命令キャッシュライン操作（<CRm> == c13、<opcode2> == 1）を使用して、メモリキャッシュラインをキャッシュにフェッチします。
8. レジスタ 9 に <CRm> == 0 を書き込み、ビット *i* を L == 1 に設定し、他のビットすべてはこのルーチンの開始前の値に戻します。

フォーマット C のキャッシュのロック解除プロシージャ

キャッシュのロックダウンされた部分をロック解除するには、レジスタ 9 の各ビットに L == 0 を書き込みます。

フォーマット D のロックダウン

このフォーマットでは、キャッシュウェイ方式を使用せず、個別の L1 キャッシュラインエントリをロックします。命令キャッシュとデータキャッシュの場合で方法は異なります。

CP15 のレジスタ 9 のフォーマット D ロックダウンレジスタにアクセスするために使用される命令は次のとおりです。

```
MCR p15, 0, Rd, c9, c5, 0    ; fetch and lock instruction cache line,
                             ; Rd = MVA
MCR p15, 0, Rd, c9, c5, 1    ; unlock instruction cache,
                             ; Rd ignored
MCR p15, 0, Rd, c9, c6, 0    ; write data cache lock register,
                             ; Rd = set/clear lock mode
MRC p15, 0, Rd, c9, c6, 0    ; read data cache lock register,
                             ; Rd = lock mode status
MCR p15, 0, Rd, c9, c6, 1    ; unlock data cache,
                             ; Rd ignored
```

——— 注 ———

ARMv6 以前は、一部のフォーマット D 実装では c5 と c6 ではなく、c1 と c2 を使用していました。ARMv6 以前のアーキテクチャバリエーションでは、CP15 の機能が必須ではなく、ガイドラインのみであったため、実装のテクニカルリファレンスマニュアルを必ず確認する必要があります。

キャッシュセット内でロック可能なエントリ数については、3 つの規則があります。

- キャッシュセットごとに最低でも 1 つのエントリを、通常のキャッシュ動作用に残す必要があります。この制限に準拠しない場合、動作は予測不能です。
- 各キャッシュセット内でいくつのウェイをロック可能かは実装定義です。
  MAX_CACHESET_ENTRIES_LOCKED < NWAYS
- フォーマット D で追加のエントリをロックする試みがロック解除されたエントリとして割り当てられるか、または無視されるかは、実装依存です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

命令キャッシュの場合、個別のキャッシュラインをフェッチしてロックするためにフェッチとロックの操作が使用されます。各キャッシュラインは、修飾仮想アドレスで指定されます。詳細についてはP. B8-3 *「修飾仮想アドレス」*を参照して下さい。コードを命令キャッシュにロックする場合、次の規則が適用されます。

- ラインを命令キャッシュにロックするために使用するルーチンは、キャッシュ不可メモリから実行する必要があります。
- 命令キャッシュにロックされるコードはキャッシュ可な必要があります。
- キャッシュラインをロックダウンする前に、命令キャッシュを許可し、無効化する必要があります。

これらの制限に準拠しない場合、動作は予測不能です。エントリは、グローバル命令キャッシュロック解除コマンドを使用してロック解除する必要があります。

キャッシュラインをデータキャッシュにロックするには、最初にグローバルロック制御ビットをセットする必要があります。グローバルロック制御ビットがセットされている間に発生したデータキャッシュのラインフィルは、データキャッシュにロックされます。データをデータキャッシュにロックする場合、次の規則が適用されます。

- ロックされるデータはキャッシュに存在していない必要があります。この条件を満たすため、キャッシュのクリーニングと無効化の操作が必要な場合があります。
- ロックされるデータはキャッシュ可な必要があります。
- データキャッシュが有効になっている必要があります。

データキャッシュロックレジスタのフォーマットを表 B6-16 に示します。

表 B6-16 データキャッシュロックレジスタ

L（ビット **[0]**） ロックビット。

0	ロックは発生しません。
1	このビットがセットされている間はすべてのデータフィルがロックされます。

レジスタ 7 の操作とのインタラクション

キャッシュロックダウンでは、キャッシュミス時に使用される通常の置換手法が、ロックダウンされている領域からキャッシュラインを再割り当てしないようにするだけです。キャッシュの内容を無効化、クリーニング、クリーニングと無効化するレジスタ 7 操作は、ロックダウンされているキャッシュラインについても通常に機能します。無効化操作を使用する場合、ロックダウンされているキャッシュラインに影響する仮想アドレスやキャッシュセット / ウェイの組み合わせを使用しないように注意する必要があります。または、ロックダウンされているキャッシュラインへの影響を避けることが困難な場合、その後でキャッシュのロックダウン手順を繰り返し実行します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 B7 章
密結合メモリ

本章では、*密結合メモリ*（TCM）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。

- *TCM について* : P. B7-2
- *TCM の構成と制御* : P. B7-3
- *TCM とキャッシュへのアクセス* : P. B7-7
- *レベル 1 （L1）DMA モデル* : P. B7-8
- *CP15 のレジスタ 11 を使用した L1 DMA 制御* : P. B7-9

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B7.1 TCM について

TCM は、キャッシュの特徴である予測不能性のないレイテンシの低いメモリをプロセッサが使用できるように設計されています。このようなメモリは、割り込み処理ルーチンやリアルタイムタスクなど、キャッシュの不確定性が望ましくない重要なルーチンの保持に使用できます。さらに、スクラッチパッドデータ、キャッシュに適していない局所性を持つデータタイプ、割り込みスタックなど重要なデータ構造の保持にも使用できます。

アーキテクチャでは、最大 4 バンクまでのデータ TCM と、4 バンクまでの命令 TCM がサポートされています。各バンクは、物理メモリマップで別々のロケーションにプログラムする必要があります。

TCM はシステムの物理メモリマップの一部として使用することを目的としており、同じ物理アドレスにある別レベルの外部メモリによって補完されるものではありません。このため、TCM の動作は、ライトスルーキャッシュ可にマークされているメモリ領域に対するキャッシュの動作とは異なっています。このような領域では、TCM に格納されているメモリロケーションへの書き込みに対して、外部書き込みは発生しません。P. B7-6「*スマートキャッシュの動作*」で定義されているように、規定された（ベースアドレス、サイズ）メモリ領域に対して TCM がキャッシュのような動作をするオプションのスマートキャッシュ動作モードも存在します。

データと命令は、P. B7-9「*CP15 のレジスタ 11 を使用した L1 DMA 制御*」で説明されている L1 DMA により、TCM から外に、または TCM 内に転送できます。

特定のメモリロケーションは、TCM またはキャッシュのいずれかに含まれている必要があります。これらのロケーションは、両方に同時に存在できません。特に、TCM とキャッシュの間ではコヒーレンシ機構は一切サポートされていません。このため、TCM のベースアドレスを割り当てる場合、TCM と同じアドレス範囲がキャッシュに含まれていないことを確認する必要があります。

B7.1.1 ページテーブルのマッピングの制限

TCM は物理アドレスでインデクス付けされ、物理アドレスでアドレス指定されるように実装されているように見えなければならず、次のように動作する必要があります。

- TCM のエントリは、異なる仮想メモリから物理メモリへのマッピングのために、ソフトウェアでクリーニングや無効化を行う必要はありません。
- TCM に保持されているメモリ領域中に、同じ物理アドレスへの複数のエイリアスが存在可能です。結果として、TCM に対するページマッピングの制限は、キャッシュについてのものよりも緩くなっています。

B7.1.2 ページテーブル属性の制限

TCM によって扱われるメモリ領域を記述するページテーブルエントリは、キャッシュ可とキャッシュ不可のどちらにもマークできますが、共有にマークすることはできません。これらのエントリがデバイスまたはストロングオーダとしてマークされている場合、あるいは共有属性が設定されている場合、TCM に含まれているロケーションは非共有でキャッシュ不可として扱われます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B7.2 TCM の構成と制御

システム制御コプロセッサ（CP15 のレジスタ 0、1、9）は、システムの TCM の構成と制御に使用されます。ARMv6 以前は、TCM のサポート方法は実装定義ですが、通常はシステム制御コプロセッサインターフェース経由で行われます。

B7.2.1 TCM ステータスレジスタ CP15 のレジスタ 0

実装される TCM の数は実装固有で、TCM ステータスレジスタにより示されます。このレジスタは読み出し専用で、次のように opcode_2 フィールドを 2 に設定して CP15 のレジスタ 0 を読み出すことでアクセスできます。

```
MRC p15, 0, Rd, C0, C0, 2 ; returns the TCM Status register
```

TCM ステータスレジスタのフォーマットは次のとおりです。

31	30	29	28 - 19	18 - 16	15 - 3	2 - 0
0	0	0	SBZ/UNP	DTCM	SBZ/UNP	ITCM

ITCM（ビット **[2:0]**）

実装されている命令（または統合）TCM 数を示します。この値は 0 ～ 4 で、それ以外の値はすべて予約済みです。命令 TCM はすべて、命令側とデータ側の両方からアクセス可能な必要があります。

DTCM（ビット **[18:16]**）

実装されているデータ TCM 数を示します。この値は

0 ～ 4 で、それ以外の値はすべて予約済みです。

B7.2.2 CP15 のレジスタ 1 の TCM 制御ビット

システム制御コプロセッサのレジスタ 1 にある以下のビットは、以前は TCM の制御に使用されていました。

DT（ビット **[16]**） このビットは現在では*常に 1* です（SBO）。このビットは、ARM946 と ARM966 コアで、データ TCM を有効にするために使用されていました。ARMv6 では、各 TCM ブロックについて個別の許可機能があります。このため、グローバルビットは必要ありません。

IT（ビット **[18]**） このビットは現在では常に 1 です。このビットは、ARM946 と ARM966 コアで、命令 TCM を許可するために使用されていました。ARMv6 では、各 TCM ブロックについて個別の許可機能があります。このため、グローバルビットは必要ありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B7.2.3 CP15 のレジスタ 9 を使用する TCM 領域レジスタ

各 TCM バンクには専用の領域レジスタがあります。このレジスタは、その TCM の物理ベースアドレスとサイズを示し、許可 / 禁止と動作モードを制御します。範囲プリフェッチまたは DMA 操作の実行中に TCM 領域レジスタを変更した場合、効果は予測不能です。

各 TCM 領域レジスタにアクセスするには、TCM 選択レジスタを目的の TCM に設定します。これらのレジスタは、特権モードでの動作中のみアクセス可能で、表 B7-1 に示すように CP15 のレジスタ 9 を使用してアクセスします。

表 B7-1 TCM レジスタ

ARM 命令	TLB 領域レジスタ
`MRC/MCR P15, 0, Rd, C9, C1, 0`	データ TCM 領域レジスタ
`MRC/MCR P15, 0, Rd, C9, C1, 1`	命令 / 統合 TCM 領域レジスタ
`MRC/MCR P15, 0, Rd, C9, C2, 0`	TCM 選択レジスタ

TCM 選択レジスタ

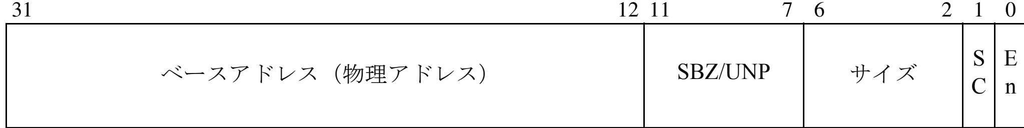
TCM 選択レジスタのフォーマットは次のとおりです。

TCM 番号（ビット **[1:0]**）

領域レジスタが適用される TCM 番号を示します。リセット時の値は 0 です。TCM 選択レジスタに対して、実装されていないメモリを指す値が書き込まれた場合、書き込みは無視されます。

TCM 領域レジスタ

TCM 領域レジスタのフォーマットは次のとおりです。



ベースアドレス（ビット **[31:12]**）

TCM の物理ベースアドレスを示します。ベースアドレスは、TCM のサイズでアラインされていると見なされます。[(log2(RAM サイズ) - 1):12] の範囲のビットは無視されます。リセット時のベースアドレスは 0 です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

サイズ（ビット [6:2]）

読み出し時には TCM のサイズを示し、書き込みは無視されます。サイズフィールドのエンコードを表 B7-2 に示します。

SC（ビット **[1]**） この TCM はスマートキャッシュとして許可されています（TCM がスマートキャッシュをサポートしている場合）。

0 = ローカル RAM（リセット時の状態）

1 = スマートキャッシュ

RAM がスマートキャッシュをサポートしていない場合、このビットから 0 が読み出され、書き込みは無視されます。このビットを使用して、TCM がスマートキャッシュをサポートしているかどうかを判定できます。

En（ビット **[0]**） 0 = 禁止（リセット時の状態）

1 = 許可

TCM が実装されていない場合、その TCM の TCM 領域レジスタは常に 0/ 予測不能です。

実装固有のサイズの表現を表 B7-2 に示します。

表 B7-2 TCM のサイズ

サイズフィールド	メモリサイズ	サイズフィールド	メモリサイズ
0b00000	0KB	0b01101	4MB
0b00011	4KB	0b01110	8MB
0b00100	8KB	0b01111	16MB
0b00101	16KB	0b10000	32MB
0b00110	32KB	0b10001	64MB
0b00111	64KB	0b10010	128MB
0b01000	128KB	0b10011	256MB
0b01001	256KB	0b10100	512MB
0b01010	512KB	0b10101	1GB
0b01011	1MB	0b10110	2GB
0b01100	2MB	0b10111	4GB

各 TCM のベースアドレスは異なっていなければならず、複数の TCM に含まれるメモリロケーションが存在しないようにしておく必要があります。特定のメモリロケーションが複数の TCM に含まれている場合、そのメモリから返される命令やデータは予測不能です。実装では、このような状態で TCM に物理的な損傷が起きないことを保証する必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B7.2.4 スマートキャッシュの動作

TCM がスマートキャッシュとして構成されている場合、その TCM は連続したキャッシュ領域を形成し、メモリの内容は外部メモリに書き出されます。TCM の各ラインはキャッシュラインと同じ長さ（対応するキャッシュのキャッシュタイプレジスタで示されます）で、個別に有効または無効に設定できます。RAM 領域レジスタに書き込みを行うと、各ラインの有効な情報がクリアされます（無効としてマークされます）。無効なラインに対して読み出しアクセスが行われた場合、そのラインはキャッシュミスの場合と全く同じ方法で L2 メモリシステムからフェッチされ、フェッチされたラインは有効としてマークされます。

スマートキャッシュをサポートする TCM の数は実装定義です。

TCM がスマートキャッシュの動作を行うには、スマートキャッシュとして動作している TCM でカバーされるメモリ領域はキャッシュ可としてマークされている必要があります。キャッシュ不可としてマークされているメモリロケーションが TCM によってカバーされている領域に存在する場合、対応するスマートキャッシュラインが無効としてマークされていれば、外部メモリからフェッチされ、有効にマークされているロケーションからメモリアクセスによりフェッチを行うことはできません。対応するスマートキャッシュラインが有効としてマークされている場合、アクセスが TCM と外部メモリのどちらに対して行われるかは予測不能です。

共有としてマークされているメモリ領域は、スマートキャッシュをメモリに対して透過的にする機構が実装でサポートされている場合のみ、スマートキャッシュでカバーできます。このため、スマートキャッシュでカバーされるメモリ領域を共有としてマークできるかどうかは実装定義です。

B7.2.5 ローカル RAM の動作

TCM がローカル RAM として構成されている場合、その TCM は連続したメモリ領域を形成し、TCM が許可されている場合は常に有効です。このため、各ラインのスマートキャッシュで使用される有効ビットは使用されません。ローカル RAM として構成される TCM はシステムの物理メモリマップの一部として使用することを目的としており、同じ物理アドレスにある別レベルの外部メモリによって補完されるものではありません。このため、TCM の動作は、ライトスルーキャッシュ可にマークされているメモリ領域のキャッシュの動作とは異なっています。このような領域では、TCM に格納されているメモリロケーションへの書き込みに対して、外部書き込みは発生しません。

DMA は、ローカル RAM として構成されている TCM 領域に対してのみ動作します。これによって、キャッシュリフィルと DMA 動作とのインタラクションが必要でなくなります。スマートキャッシュとして構成されている TCM 領域に DMA を実行すると、P. B7-9 *「CP15 のレジスタ 11 を使用した L1 DMA 制御」*で説明されているように、内部 DMA エラー（密結合 DMA 範囲外）が発生します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B7.3 TCM とキャッシュへのアクセス

要求されたアドレスが TCM とキャッシュの両方に含まれている場合、命令やデータがどのメモリから返されるかは予測不能です。実装では、このような状態でキャッシュや TCM に物理的な損傷が起きないことを保証する必要があります。このような状態は、TCM のベースレジスタが変更されたときにキャッシュの無効化を行わないことでのみ発生します。これはプログラムのエラーです。

ハーバード構成のキャッシュと TCM の場合、データの読み出しと書き込みで、ローカルメモリとして構成された任意の命令 TCM に対して、読み出しと書き込み両方のアクセスが可能な必要があります。これによって、リテラルプール、未定義命令、SWI 番号へのアクセスが可能になり、デバッグが容易になります。このような実装では、ポート間のアービトレーションは実装定義です。この理由から、ローカルメモリとして構成されている命令 TCM は統合 TCM として動作する必要がありますが、命令フェッチに最適化することはかまいません。この条件は、TCM をローカル RAM として構成したときにのみ適用されます。

命令 TCM への書き込みと、その書き込みに依存する命令との間には、命令メモリバリアを必ず挿入する必要があります。また、命令 TCM 内の分岐を上書きする場合、分岐予測機構はすべて無効化または禁止する必要があります。

逆に、命令ポートがデータ TCM にアクセスできる必要はありません。命令ポートから、データ TCM でカバーされる範囲のアドレスにアクセスを試みた場合、データ TCM へのアクセスは行われません。この場合、命令はメインメモリからフェッチされる必要があります。このようなアクセスは、そのアドレス範囲がメインメモリでサポートされていない可能性があるため、一部のシステムでは外部アボートになります。

命令 TCM は、データ TCM と同じベースアドレスにプログラムできません。また、2 つの RAM のサイズが異なっている場合、2 つの RAM の物理メモリ領域がオーバラップすることはできません。ただし、両方の TCM がスマートキャッシュとして動作するように構成されている場合は除きます。データ TCM と命令 TCM がオーバラップし、いずれかがスマートキャッシュとして構成されていない場合、命令やデータがどのメモリから返されるかは予測不能です。実装では、このような状態で TCM に物理的な損傷が起きないことを保証する必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B7.4 レベル 1（L1）DMA モデル

L1 DMA の目的は、データのブロックを TCM との間で転送するバックグラウンドルートの提供です。これはワードや小さな構造体ではなく、比較的大きなデータのブロックを移動することを主な目的としています。実装可能な DMA チャネルの数は実装定義で、0 でもかまいません。アーキテクチャで定義された DMA モデルが望ましいですが、必ずこの動作を行う必要があるわけではありません。コアの外部にあるエージェントにより TCM 用の DMA をサポートすることも許容されます。他のモデルは本質的に実装定義です。

L1 DMA は適切なシステム制御コプロセッサ（CP15）レジスタと命令へのアクセスにより、開始、制御できます。この手順は DMA の内部的な開始と終了のアドレス、外部的な開始アドレス、DMA の方向を指定します。指定されるアドレスは仮想アドレスで、L1 DMA ハードウェアには仮想アドレスから物理アドレスへの変換と、保護属性のチェックを含める必要があります。実装では、DMA のページテーブルエントリを保持するため、P. B4-2「*VMSA の概要*」で説明しているように TLB を使用できますが、DMA により使用される TLB はページテーブルと整合している必要があります。保護チェックから発生するエラーは、割り込みを使用して CPU に信号を送るように構成できます。

DMA の完了も、エラーで使用するのと同じ割り込みを使用して、CPU に信号を送るように構成できます。

DMA のステータスは、その DMA に関連する CP15 レジスタから読み出すことができます。

利用可能な DMA チャネルの数は実装定義で、それぞれに独自の制御レジスタとステータスレジスタがあります。定義可能な DMA チャネルの最大数はアーキテクチャにより 2 に制限されています。同時にアクティブな DMA チャネルは 1 つだけです。他の DMA チャネルが開始された場合、新しく開始されたチャネルはキューに保存され、現在アクティブなチャネルの動作が完了した後で、メモリに対する動作が開始されます。

DMA 転送は、外部メモリと TCM の間で、指定の方向で行われる必要があります。命令 TCM とデータ TCM の間の転送や、L1 の外部にある 2 つのメモリロケーション間の転送はサポートされていません。このような転送を試みた場合、DMA チャネルによりエラーが通知されます。

L1 DMA は CPU の他の部分とは独立したマスタとして動作し、L1 DMA システムによりアクセスされる外部アドレスが CPU の他の部分からもアクセスされる場合、共有メモリ領域を扱うのと同じ機構を使用する必要があります。ユーザモードの DMA 転送が、共有としてマークされていない外部アドレスを使用して行われた場合、DMA チャネルによりエラーが通知されます。

チャネルの実行中に、L1 DMA によって発生するメモリアクセスと、CPU による読み出しや書き込みによるメモリアクセスとの間には、順序の条件はありません。チャネルの実行が完了したとき、チャネルのトランザクションはシステムにある他のすべてのオブザーバから可視になります。DMA により発生するすべてのメモリアクセスは、メモリタイプにかかわらず、DMA チャネルで指定された順序で発生します。

DMA がストロングオーダメモリに対して実行された場合、DMA によって発生したトランザクションによるアクセスが完了するまで、その DMA による以降のトランザクションは生成されません。トランザクションは、ターゲットロケーションの状態が変更された、またはデータが DMA に返された時点で完了します。

DMA チャネルが実行中か待機中の状態の間に FCSE PID、ドメインアクセス制御レジスタ、ページテーブルのマッピングのいずれかが変更された、または TLB がフラッシュされた場合、これらの変更の影響が DMA から可視かどうかは予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B7.5 CP15 のレジスタ 11 を使用した L1 DMA 制御

L1 DMA は、CP15 のレジスタ 11 を使用して制御されます。CP15 のレジスタ 11 に関連するレジスタはいくつかあり、DMA の制御と監視に使用されます。これらは表 B7-3 で定義されています。命令の形式は次のとおりです。

```
MCR p15, 0, Rd, c11, CRm, Opcode2
MRC p15, 0, Rd, c11, CRm, Opcode2
```

ここで、CRm は関連するレジスタ、Rd は ARM® のソースまたはデスティネーションレジスタです。これらと Opcode2 について表 B7-3 に示します。詳細については、以下のサブセクションで解説します。

表 B7-3 L1 DMA 制御レジスタ

レジスタ	CRm	Opcode 2	機能
識別 / ステータス	0	存在 / 待機中 / 実行中 / 割り込み中	特権モードのみ : 読み出し専用
ユーザアクセス許可	1	0	特権モードのみ : 読み出し / 書き込み
チャネル番号	2	0	読み出し / 書き込み
許可	3 [a]	停止 / 開始 / クリア	書き込み専用
制御	4 [a]	0	読み出し / 書き込み
内部開始アドレス	5 [a]	0	読み出し / 書き込み
外部開始アドレス	6 [a]	0	読み出し / 書き込み
内部終了アドレス	7 [a]	0	読み出し / 書き込み
チャネルステータス	8 [a]	0	読み出し専用
予約 SBZ/UNP	9-14	0	読み出し / 書き込み
コンテキスト ID	15 [a]	0	特権モードのみ : 読み出し / 書き込み

a. チャネルあたり 1 レジスタ

有効、制御、内部的な開始アドレス、外部的な開始アドレス、内部的な終了アドレス、チャネルステータス、コンテキスト ID のレジスタは複数のレジスタで、実装されている各チャネルごとに 1 つ存在します。アクセスされるレジスタは、P. B7-4「*CP15 のレジスタ 9 を使用する TCM 領域レジスタ*」で説明されているように、チャネル番号レジスタにより決定されます。

B7.5.1 CP15 のレジスタ 11 操作へのユーザからのアクセス

CP15 のレジスタ 11 の操作のいくつかは、ユーザモードのコードからも実行可能です。

CP15 のレジスタ 11 の特権操作をユーザモードで実行しようとすると、未定義命令例外が発生します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B7.5.2 識別とステータスレジスタ

これらのレジスタは、特定のデバイスに物理的に実装されている DMA チャネルと、その現在のステータスを定義します。DMA を扱うプロセスはこれらの情報を使用して、実装されている物理リソースと、それらが利用可能かどうかを判断できます。各レジスタの下位 2 ビットはチャネルビットで、アーキテクチャで定義されている 2 つのチャネルに対応しています（ビット 0 はチャネル 0、ビット 1 はチャネル 1 に対応しています)。

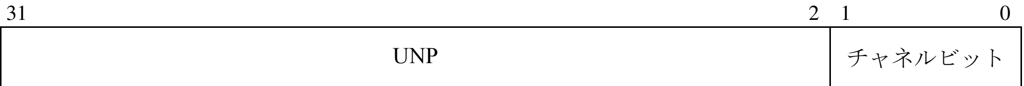
Opcode2 の値は、実装されているレジスタを表 B7-4 に示すように識別します。

表 B7-4 L1 DMA の識別とステータスレジスタ

Opcode2	機能
0	存在 : 存在しているチャネルについては 1、存在しないチャネルについては 0。
1	待機中 : チャネルが待機中の場合 1、それ以外の場合 0。実装されていないチャネルについては 0 が返されます。
2	実行中 : チャネルが実行中の場合 1、それ以外の場合 0。実装されていないチャネルについては 0 が返されます。
3	割り込み中 : 割り込みを発生している（完了またはエラー）チャネルについては 1、それ以外の場合 0。実装されていないチャネルについては 0 が返されます。
4-7	予約、予測不能。

これらのレジスタは、特権プロセスによってのみ読み出し可能です。ユーザプロセスからアクセスを試みると、未定義命令例外が発生します。

レジスタ 0 ～ 3 の形式は次のとおりです。



これらのレジスタにアクセスする命令は次のとおりです。

```
MRC p15, 0, Rd, c11, c0, n ; where n is 0, 1, 2, or 3
```

B7.5.3 ユーザアクセス許可レジスタ

このレジスタには、各チャネルのビットが含まれており、そのチャネルの U ビットと呼ばれます。これは、そのチャネルのレジスタがユーザモードプロセスからアクセス可能かどうかを示しています。チャネルの U ビットが 1 である場合、次のレジスタにアクセス可能です。

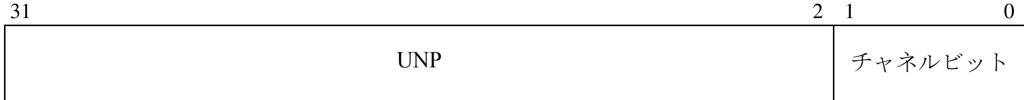
- 許可
- 制御
- 内部開始アドレス
- 外部開始アドレス
- 内部終了アドレス
- チャネルステータス

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

これらのレジスタがユーザからアクセス可能な場合、レジスタの内容はタスクスイッチ時に保持される必要があります。

チャネルの U ビットが 0 に設定されている場合、ユーザプロセスからアクセスを試みると、未定義命令例外が発生します。

ユーザアクセス許可レジスタのフォーマットは次のとおりです。



これらのレジスタにアクセスする命令は次のとおりです。

```
MCR p15, 0, Rd, c11, c1, 0
MRC p15, 0, Rd, c11, c1, 0
```

B7.5.4 チャネル番号レジスタ

許可、制御、内部開始アドレス、外部開始アドレス、内部終了アドレス、チャネルステータス、コンテキスト ID のレジスタは多重レジスタで、実装されている各チャネルごとに 1 つ存在します。これらのレジスタのいずれかが指定された場合、多重レジスタのどれがアクセスされるかは、チャネル番号レジスタに含まれている値により判定されます。

いずれかのチャネルの U ビットが 1 に設定されている場合、このレジスタはユーザプロセスからアクセス可能です。どのチャネルの U ビットも 1 に設定されていない場合は、ユーザプロセスからアクセスを試みると、未定義命令例外が発生します。

DMA チャネル番号レジスタのフォーマットは次のとおりです。

これらのレジスタにアクセスする命令は次のとおりです。

```
MCR p15, 0, Rd, c11, c2, 0
MRC p15, 0, Rd, c11, c2, 0
```

B7.5.5 許可レジスタ

実装されている各 DMA チャネルには、そのチャネルの開始、停止、クリアを行うために書き込み可能な固有のレジスタロケーションがあります。MCR 命令の Opcode2 の値により、P. B7-12 表 B7-5 に示すように実行される操作が決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

これらのレジスタにアクセスする命令は次のとおりです。

```
MCR p15, 0, Rd, c11, c3, n ; where n is 0, 1, or 2
```

表 B7-5 DMA チャネル許可操作

Opcode2	操作
0	停止
1	開始
2	クリア
3-7	予約

チャネルの U ビットが 1 に設定されている場合、ユーザプロセスがそのチャネルに対してこれらの操作を実行できます。チャネルの U ビットが 0 に設定されている場合、ユーザプロセスがこれらの操作のいずれかの実行を試みると、未定義命令例外が発生します。

チャネルステータスについては、P. B7-16 *「チャネルステータスレジスタ」*を参照して下さい。

開始

開始コマンドは、チャネルの DMA 転送を開始します。開始コマンドを実行すると、その時点で他の DMA チャネルが動作中でなければ、チャネルステータスが実行中に変更されます。そうでない場合、ステータスは待機中に設定されます。チャネルのステータスが待機中または実行中の場合、またはチャネルがエラーを示している場合、そのチャネルは動作中です。チャネルがエラーを示している場合、ステータスはエラー / 完了で、内部または外部エラーの値が 0b01000 以上です。

停止

停止コマンドは、チャネルステータスが実行中の場合に発行されます。この命令が発行されると、DMA チャネルによるメモリアクセスは可能な限り早く停止されます。アクセスが再起動可能な外部メモリへのものである場合、外部メモリから読み出されているが、まだ TCM に書き込まれていないアクセスを破棄することで、この過程を高速化できます。この高速化手法は、デバイスとしてマークされているメモリ領域との間の DMA 転送には使用できません。

停止命令が発行されてから、DMA チャネルが停止するまでに、多くのサイクルが必要となることがあります。チャネルが停止するまで、チャネルステータスは実行中に保持されます。すべての未解決のメモリアクセスが完了した時点で、チャネルステータスはアイドルに設定されます。チャネルの停止時に、開始アドレスレジスタには、動作の再開に必要なアドレスが含まれています。

チャネルステータスが待機中のときに停止コマンドが発行されると、チャネルステータスはアイドルに変更されます。

チャネルステータスが実行中と待機中のいずれでもない場合、停止コマンドは無効です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

クリア

クリアコマンドは、チャネルステータスをエラー / 完了からアイドルに変更します。同時に、エラーまたは完了の結果としてチャネルに設定されている割り込み（定義については*「制御レジスタ」*を参照して下さい）をクリアします。内部または外部の開始アドレスレジスタの内容は、このコマンドで変更されません。

チャネルステータスが実行中または待機中の場合、クリアコマンドは無効です。

DMA のデバッグに関する考慮点

L1 DMA はプロセッサコアと独立したエンジンとして動作し、開始時には自律的に動作します。その結果、L1 DMA にステータスが実行中または待機中のチャネルが含まれている場合、プロセッサがデバッグ機構により停止しても、それらのチャネルは実行を継続、または実行を開始できます。その結果、プロセッサがデバッグで停止している間に TCM の内容が変化することがあります。この状態を回避するには、停止操作により DMA チャネルを停止する必要があります。

B7.5.6 制御レジスタ

実装されている各 DMA チャネルには、DMA 操作を制御する専用のレジスタがあります。これらのレジスタの形式は次のとおりです。

31	30	29	28	27	26	25 - 20	19 - 8	7 - 2	1 - 0
TR	DT	IC	IE	FT	UM	UNP/SBZ	ST	UNP/SBZ	TS

これらのレジスタにアクセスする命令は次のとおりです。

```
MCR p15, 0, Rd, c11, c4, 0
MRC p15, 0, Rd, c11, c4, 0
```

TS（ビット **[1:0]**） トランザクションサイズ。トランザクションサイズは、DMA チャネルにより実行されるトランザクションのサイズを示します。デバイスやストロングオーダのメモリロケーションでは、これらのメモリへのアクセスがプログラムされたサイズで発生することを保証するため、この情報は特に重要です。

00 = バイト

01 = ハーフワード

10 = ワード

11 = ダブルワード（8 バイト）

ST（ビット **[19:8]**） ストライド（バイト）。ストライドは、DMA の連続したアクセスごとに外部アドレスに加算される値を示します。ストライドが 0 の場合、外部アドレスを加算しないことを示します。これは、FIFO などの場所へのアクセスを容易にするように設計されています。

ストライドの値は、トランザクションサイズでアラインされている必要があります。そうでない場合、動作は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ストライドは、正または 0 の数値として解釈されます。

内部的なアドレス増分はストライドの影響を受けず、トランザクションサイズに固定されます。

UM（ビット **[26]**） ユーザモード。アクセス許可チェックが、ユーザモードにいる DMA と特権モードにいる DMA のどちらに基づいているかを示します。

0 = 転送は特権転送です。

1 = 転送はユーザモード転送です。

チャネルの U ビットがセットされている場合、UM ビットには 1 のみが書き込めます。この場合、このビットに 0 を書き込む操作は無効です。

FT（ビット **[27]**） フル転送。DMA が、TCM から外部メモリへのデータ転送の一部として、データのすべてのワードを転送することを示します。

0 = FT ビットが 0 に設定されていた以前の DMA によって読み出し / 書き込みされて以来、またはリセット以来のどちらかより最近の動作でのストア操作によって変更された TCM 中の DMA アドレス範囲のメモリロケーションは少なくとも転送する。実装では、このビットの値の結果 TCM から転送されるデータの量を最小化するようにします。

1 = ロケーションがストアにより変更されているかどうかにかかわらず、DMA のアドレス範囲全体を転送します。FT ビットが 1 にセットされている TCM への、DMA によるアクセスは、どのロケーションが書き込みにより変更されたかの記録に影響しません。

IE（ビット **[28]**） エラー時の割り込み。このビットの動作は、U ビットの設定により異なります。詳細については、P. B7-10 *「ユーザアクセス許可レジスタ」*を参照して下さい。

U = 0 かつ IE[28] = 0 の場合、DMA チャネルはエラーによる割り込みをアサートしません。

U = 1 と IE[28] = 1 のいずれかが成立する場合、DMA チャネルはエラーによる割り込みをアサートします。

割り込みは、チャネルが開始操作により実行中に設定されるか（P. B7-11 *「許可レジスタ」*参照）、クリア操作によりアイドルに設定されると、（そのソースから）アサート解除されます。チャネルの DMA トランザクションのうち、U ビットが 1 にセットされている（P. B7-10 *「ユーザアクセス許可レジスタ」*参照）ものはすべて、このビットに書き込まれた値にかかわらず、エラー時に割り込みをアサートします。

IC（ビット **[29]**） 完了時の割り込み。完了時の割り込みビットは、DMA 転送の完了時に DMA チャネルが割り込みをアサートするかどうかを示します。割り込みは、それを発生したチャネルに対してクリア操作が実行されると（そのソースから）アサート解除されます。詳細については、P. B7-11 *「許可レジスタ」*を参照して下さい。完了時に割り込みが生成されるかどうかは、U ビットの影響を受けません。

0 = 完了時に割り込みは生成されません。

1 = 完了時に割り込みが生成されます。

DT（ビット **[30]**） 転送の方向。

0 = L2 メモリから TCM へ。

1 = TCM から L2 メモリへ。

TR（ビット **[31]**） ターゲット TCM。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

0 = データ（または統合）

1 = 命令

チャネルの U ビットが 0 に設定されている場合、ユーザプロセスからレジスタにアクセスを試みると、未定義命令例外が発生します。チャネルのステータスが実行中または待機中のときに制御レジスタに書き込みを試みた場合、結果は予測不能です。

実装上の注意

FT ビットが 0 の場合の機能を実装する機構は実装定義です。TCM の各ライン（または複数のライン）を、そのライン内のロケーションにストアが行われたことを記録する（つまり、そのラインがダーティである）ことを示す*ダーティビット*でマークする方法は、一部のロケーションが誤ってダーティにマークされる可能性があるとしても、許容される実装です。このような実装では、DMA 転送の一部でない TCM ライン内のロケーションを、DMA 転送の結果としてクリーンにマークすることはできません。TCM からの DMA 書き込みの場合（DT == 1）、ラインのダーティビットはライン全体が DMA 転送により書き込まれた場合にクリアされますが、DMA 転送がラインの一部だけに書き込みを行った場合には変更されません。

B7.5.7 内部開始アドレスレジスタ

これらのレジスタは、各 DMA チャネルについて TCM の最初のアドレス、つまりデータが転送される最初のアドレスを示します。内部開始アドレスは仮想アドレスで、物理的なマッピングはチャネル開始時のページテーブル中に記述されます。その仮想アドレスのメモリ属性が転送に使用されます。メモリのアクセス許可フォルトが生成される可能性があります。内部開始アドレスは TCM 内の必要があります。そうでない場合、チャネルステータスレジスタにエラーが通知されます。TCM 内のメモリロケーションをデバイスとしてマークした場合、結果は予測不能です。

DMA チャネルが実行中のこのレジスタの内容は予測不能です。停止コマンドかエラーによりチャネルが停止したときは、このレジスタにはトランザクション再起動に必要なアドレスが含まれています。完了時は、このレジスタの値は内部終了アドレスと同じです。

内部開始アドレスは、制御レジスタに設定されているトランザクションサイズにアラインされている必要があります。そうでない場合、結果は予測不能です。

チャネルの U ビットが 0 にセットされている場合、ユーザプロセスからレジスタにアクセスを試みると、未定義命令例外が発生します。DMA チャネルが実行中か待機中のときは、このレジスタへの書き込みは無効です（エラーを発行せずに失敗します）。

このレジスタに読み出しと書き込みを行うには、次の命令を使用します。

```
MCR p15, 0, Rd, c11, c5, 0
MRC p15, 0, Rd, c11, c5, 0
```

B7.5.8 外部開始アドレスレジスタ

これらのレジスタは、各 DMA チャネルについて外部メモリの最初のアドレス、つまりデータが転送される最初のアドレスを示します。外部開始アドレスは仮想アドレスで、物理的なマッピングはチャネル開始時のページテーブル中に記述されます。その仮想アドレスのメモリ属性が転送に使用されます。メモリのアクセス許可フォルトが生成される可能性があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

外部開始アドレスは、L1 メモリシステム外の外部メモリの必要があります。そうでない場合、結果は予測不能です。

DMA チャネルが実行中のこのレジスタの内容は予測不能です。停止コマンドかエラーによりチャネルが停止したときは、このレジスタにはトランザクション再開に必要なアドレスが含まれています。完了時は、このレジスタの値はアクセスされた最後のアドレスにストライドを加算したアドレスを示しています。

外部開始アドレスは、制御レジスタに設定されているトランザクションサイズでアラインされている必要があります。そうでない場合、結果は予測不能です。

チャネルの U ビットが 0 に設定されている場合､ユーザプロセスからレジスタにアクセスを試みると、未定義命令例外が発生します。DMA チャネルが実行中か待機中のときは、このレジスタへの書き込みは無効です。

このレジスタに読み出しと書き込みを行うには、次の命令を使用します。

```
MCR p15, 0, Rd, c11, c6, 0
MRC p15, 0, Rd, c11, c6, 0
```

B7.5.9 内部終了アドレスレジスタ

これらのレジスタは、内部終了アドレスを定義しています。このレジスタの値は、内部開始アドレスよりも大きい必要があります。内部終了アドレスは、DMA がアクセスする最後の内部アドレス（トランザクションサイズによるモジュロ）にトランザクションサイズを加えた値です。内部終了アドレスは、DMA がアクセスしない最初の（インクリメントされた）アドレスです。最後の内部アドレスで規定されるトランザクションが完了すると、DMA 転送全体が完了します。

内部終了アドレスは、制御レジスタに設定されているトランザクションサイズでアラインされている必要があります。そうでない場合、結果は予測不能です。

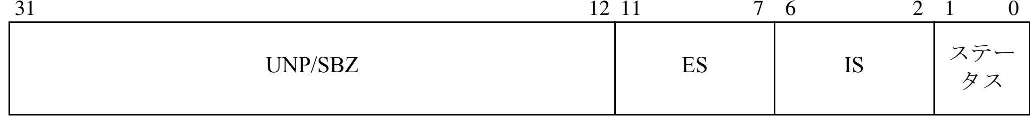
チャネルの U ビットが 0 に設定されている場合､ユーザプロセスからレジスタにアクセスを試みると、未定義命令例外が発生します。DMA チャネルが実行中か待機中のときは、このレジスタへの書き込みは無効です。

このレジスタに読み出しと書き込みを行うには、次の命令を使用します。

```
MCR p15, 0, Rd, c11, c7, 0
MRC p15, 0, Rd, c11, c7, 0
```

B7.5.10 チャネルステータスレジスタ

これらのレジスタは、各チャネルについて、そのチャネルで最後に開始された DMA 動作のステータスを定義しています。このレジスタは読み出し専用です。チャネルステータスレジスタのフォーマットは次のとおりです。



このレジスタから読み出しを行うには、次の命令を使用します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

```
MRC p15, 0, Rd, c11, c8, 0
```

ステータス（ビット **[1:0]**）

00 = アイドル

01 = 待機中

10 = 実行中

11 = 完了 / エラー

IS（ビット **[6:2]**）

内部アドレスエラーステータスで、次のようなエンコードを使用します。

0b00xxx = エラーなし（リセット時の値）

0b01000 = TCM が範囲外

0b11100 = 第 1 レベルページテーブル変換時の外部アボート

0b11110 = 第 2 レベルページテーブル変換時の外部アボート

0b10101 = 変換フォルト（セクション）

0b10111 = 変換フォルト（ページ）

0b11001 = ドメインフォルト（セクション）

0b11011 = ドメインフォルト（ページ）

0b11101 = アクセス許可フォルト（セクション）

0b11111 = アクセス許可フォルト（ページ）

他のエンコードはすべて予約されています。

ES（ビット **[11:7]**）

外部アドレスエラーステータスで、次のようなエンコードを使用します。

0b00xxx = エラーなし（リセット時の値）

0b01001 = 非共有データエラー

0b11010 = 外部アボート（不正確の可能性あり）

0b11100 = 第 1 レベルページテーブル変換時の外部アボート

0b11110 = 第 2 レベルページテーブル変換時の外部アボート

0b10101 = 変換フォルト（セクション）

0b10111 = 変換フォルト（ページ）

0b11001 = ドメインフォルト（セクション）

0b11011 = ドメインフォルト（ページ）

0b11101 = アクセス許可フォルト（セクション）

0b11111 = アクセス許可フォルト（ページ）

他のエンコードはすべて予約されています。

エラーが発生した場合、外部エラー（ES == 0b11010）の場合を除き、フォルトアドレスは対応する開始アドレスレジスタに含まれています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ステートが待機中のチャネルは、他のチャネル（実装されている場合）がアイドルまたはエラーなしで完了 / エラーに変化した場合、自動的に実行中に変化します。

チャネルが DMA のすべての転送を完了し、それらの転送により発生したメモリロケーションへの変更がすべて他のオブザーバから可視になると、チャネルのステータスは実行中から完了 / エラーに変化します。この変化は、転送による外部アクセスが完了するまでは起こりません。

チャネルの U ビットが 0 に設定されている場合、ユーザプロセスによってレジスタの読み出しを試みると、未定義命令例外が発生します。

制御レジスタの UM ビットがセットされている DMA 転送が外部メモリロケーションにアクセスを試み、そのメモリロケーションが共有としてマークされていない場合、非共有データ外部アドレスエラーが通知されます。UM ビットがクリアされている場合、このエラーは発生しません。

B7.5.11 コンテキスト ID レジスタ

このレジスタには、実装されている各 DMA チャネルについて、そのチャネルを使用しているプロセスのコンテキスト ID レジスタが含まれています。このレジスタには、そのチャネルの初期化の一部として、チャネルを使用するプロセスの CPU コンテキスト ID を書き込む必要があります。チャネルがユーザアクセス可能なチャネルとして割り当てられている場合、そのチャネルをユーザが使用できるように初期化する特権プロセスにより、チャネルの U ビットが書き込まれるのとほぼ同時に、コンテキスト ID が書き込まれます。コンテキスト ID レジスタの下位 8 ビットは、異なる仮想アドレスマップが共存できるようにするため、仮想アドレスから物理アドレスへのアドレス変換に使用されます。DMA チャネルが実行中か待機中のときは、このレジスタへの書き込みは無効です。

ユーザプロセスから DMA チャネルの使用を隠匿できるようにするため、このレジスタは特権プロセスからのみ読み出し可能です。セキュリティの理由から、書き込みは特権プロセスからのみ可能です。コンテキストスイッチ時に、DMA のステートがスタックされて復元されるときは、保存されるステートにこのレジスタを含める必要があります。

これらのレジスタのフォーマットは次のとおりです。

31 〜 0
コンテキスト ID

ユーザプロセスからこの特権レジスタにアクセスを試みると、未定義命令例外が発生します。

このレジスタに読み出しと書き込みを行うには、次の命令を使用します。

```
MCR p15, 0, Rd, c11, c15, 0
MRC p15, 0, Rd, c11, c15, 0
```

注

DMA チャネルと、関連するコンテキスト ID レジスタは、現在アクティブなページテーブルを使用します。ソフトウェアでは、ページテーブルの更新により、実行中か待機中かにかかわらず、アクティブな DMA が影響を受けないことを保証する必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 B8 章
高速コンテキストスイッチ拡張機能

本章では、*高速コンテキストスイッチ拡張機能*（FCSE）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。

- *FCSE の概要* : P. B8-2
- *修飾仮想アドレス* : P. B8-3
- *FCSE の許可* : P. B8-5
- *デバッグとトレース* : P. B8-6
- *CP15 のレジスタ* : P. B8-7

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B8.1 FCSE の概要

高速コンテキストスイッチ拡張機能（FCSE）は、ARM® メモリシステムの動作を変更します。この変更により、ARM プロセッサで実行されている複数のプログラムが同じアドレス範囲を使用し、同時にそれらがメモリシステムの他の部分に対して示すアドレスが異なることを保証できます。

通常は、アドレス範囲がオーバラップする 2 つのソフトウェアプロセスを切り替える場合、MMU ページテーブルで定義される、仮想アドレスから物理アドレスへのマッピング（第 B4 章*「仮想メモリシステムアーキテクチャ」*参照）を変更する必要があります。また、この場合通常はキャッシュと TLB の内容が無効になるため（古い仮想アドレスから物理アドレスへのマッピングに対応していることから）、キャッシュと TLB をフラッシュする必要があります。結果として、プロセスの切り替えごとに、直接的にはページテーブルの変更コストから、間接的には以後にキャッシュと TLB の再ロードが必要なことから、大きなオーバヘッドが発生します。

FCSE では、異なるソフトウェアが同じアドレスを使用していても、メモリシステムの他の部分には別のアドレスとして示されるようにすることで、このオーバヘッドを回避できます。また、これによって複数のソフトウェアプロセスが、メモリシステムの他の部分が仮想アドレスから物理アドレスへのマッピングをサポートしていない場合も、同じアドレス範囲を使用できます。

——— 注 ———

FCSE 機構は、ARMv6 では推奨されません。FCSE と、VMSAv6 で導入された非グローバル /ASID ベースのメモリ属性の両方を使用した場合、動作は予測不能です。FCSE をクリアするか、すべてのメモリをグローバルとして宣言する必要があります。

———————

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B8.2 修飾仮想アドレス

4GB の仮想アドレス空間は、それぞれ 32MB の 128 個のプロセスブロックに分割されます。各プロセスブロックは、0x00000000 から 0x01FFFFFF までのアドレス範囲を使用するようにコンパイルされたプログラムを含むことができます。0 から 127 までのそれぞれの i について、プロセスブロック I はアドレス（i × 0x02000000）からアドレス（i × 0x02000000 + 0x01FFFFFF）までの範囲で実行されます。

FCSE は、ARM プロセッサにより生成された各メモリアクセスの仮想アドレスを処理し、*修飾仮想アドレス*を生成します。この修飾仮想ドレスはメモリシステムの他の部分に送信され、通常の仮想アドレスの代わりに使用されます。MMU ベースのメモリシステムの場合、この処理は図 B8-1 に示すとおりです。

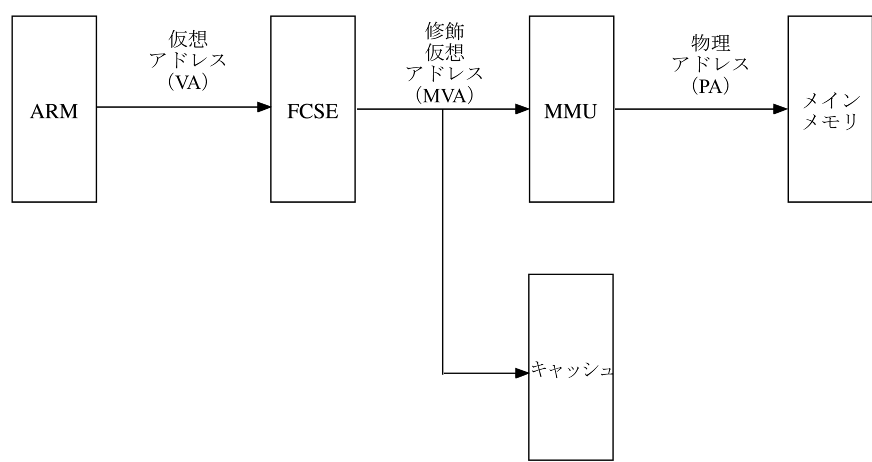


図 B8-1 FCSE を持つ MMU メモリシステム内でのアドレスフロー

ARM プロセッサがメモリアクセスを生成するときの*仮想アドレス*（VA）と*修飾仮想アドレス*（MVA）の関係は次のとおりです。

```
if (VA[31:25] == 0b0000000) then
    MVA = VA | (PID << 25)
else
    MVA = VA
```

ここで、PID は現在のプロセスがロードされているプロセスブロックを識別する 7 ビットの番号です。これは、現在のプロセスの（*FCSE）プロセス ID* とも呼ばれます。

いずれかの VMSAv6 テーブルエントリで代替コンテキスト ID が有効になっている場合、FCSE PID を 0 以外の値に設定した場合の ASID ベースのサポート（nG ビット == 1）は予測不能です。ASID の詳細については、P. B4-2「*VMSA の概要*」を参照して下さい。

——— 注 ———

仮想アドレスは、メモリシステムにデータとして渡される場合もあります。例としては、P. B6-19「*レジスタ 7: キャッシュ管理機能*」で説明されているいくつかのキャッシュ制御操作を参照して下さい。これらの操作の場合、アドレス修飾は行われず、MVA = VA です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

各プロセスは、`0x00000000` から `0x01FFFFFF` までのアドレス範囲を使用するようにコンパイルされます。したがって、固有の命令とデータを参照するときは、プログラムの生成する VA の上位 7 ビットはすべて 0 です。結果として生成される MVA では、上位 7 ビットが PID に置き換えられるため、現在のプロセスのプロセスブロックを指すようになります。

プログラムで、上位 7 ビットがすべて 0 ではない VA を生成することもできます。この場合、MVA は VA に等しくなります。これにより、他のプロセスのプロセス ID が 0 でなければ、プログラムで他のプロセスのプロセスブロックをアドレス指定することが可能になります。アクセス許可が正しく設定されていれば、この機能を使用してプロセス間の通信を行えます。

——— 注 ———

プロセス ID が 0 のプロセスはこの方法で通信できないため、一般的な目的のプロセスにはプロセス ID として 1 以上のみを使用することをお勧めします。

———

このため、FCSE の使用によりプロセススワップのコストは次のものだけになります。

- PID に書き込むコスト
- 新しいプロセスでアクセス許可を変更する必要がある場合、その変更コスト。MMU ベースのシステムでは、この場合にページテーブルエントリの個別の変更や、TTBR を変更して新しいページテーブルを指すようにする動作が行われることがあります。ページテーブルに変更を加えた場合、影響を受ける TLB エントリの無効化を伴う可能性が高くなります。ただし、これらの動作は FCSE がない場合に必要なキャッシュのフラッシュよりも、一般にコストが大幅に低くなります。また、ドメインを使用することで、ページテーブルへの変更と、関連付けられた明示的な TLB 管理も避けることが可能な場合があります。これによって、ドメインアクセス制御レジスタ（P. B4-10「*ドメイン*」参照）への書き込みのコストが低減されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B8.3 FCSE の許可

PID == 0b0000000 の場合、FCSE が存在しない場合と同様に、VA の処理規則の結果は常に MVA == VA となります。

このため、特別な FCSE イネーブルビットは存在しません。その代わりに、PID はリセット時に 0b0000000 に初期化され、FCSE は事実上禁止されます。

FCSE を許可するには、PID に 0 以外の値を書き込みます。無効にするには、PID に 0b0000000 を書き込みます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B8.4 デバッグとトレース

ブレークポイントやウォッチポイントの機構で VA や MVA を使用するかどうかは実装定義です。ただし、トリガのエイリアスを回避するため、将来の実装ではすべて MVA を使用することを強く推奨します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B8.5 CP15 のレジスタ

FCSE はコプロセッサ 15 のレジスタのうち、レジスタ 13 のみを使用します。

B8.5.1 レジスタ 13: FCSE PID

レジスタ 13 を読み出すと、ビット [31:25] に PID が返されます。読み出される値のビット [24:0] は予測不能です。

レジスタ 13 に書き込むと、書き込まれた値のビット [31:25] が PID に設定されます。ビット [24:0] には 0、またはレジスタ 13 のビット [24:0] から以前に読み出した値を書き込む必要があります。ビット [24:0] にそれ以外の値を書き込んだ場合の結果は予測不能です。

レジスタ 13 の読み出しと書き込みに `MCR` 命令と `MRC` 命令を使用する場合、`<CRm>` は c0 に、`<opcode2>` は 0（または省略）にする必要があります。他の値を指定した場合、命令の動作は予測不能です。

―――― 注 ――――

PID に書き込みを行うと、仮想アドレスから物理アドレスへのマッピング全体が変更されます。このため、既にプリフェッチされている可能性がある命令がアドレスマッピングの変更により影響を受けないよう注意が必要です。

――――――――

B8.5.2 レジスタ 13: コンテキスト ID

ARMv6 では、代替のコンテキスト ID 機構が導入されています。このレジスタについては、*P. B4-52「レジスタ 13: プロセス ID」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

パート C

ベクタ浮動小数点アーキテクチャ

第 C1 章
ベクタ浮動小数点アーキテクチャの概要

本章では、*ベクタ浮動小数点（VFP）アーキテクチャ*と、IEEE754 標準への準拠の概要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。

- *ベクタ浮動小数点アーキテクチャの概要* : P. C1-2
- *VFP アーキテクチャの概要* : P. C1-4
- *IEEE754 標準への準拠* : P. C1-9
- *IEEE 754 実装上の選択* : P. C1-10

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C1.1 ベクタ浮動小数点アーキテクチャの概要

ベクタ浮動小数点（VFP）アーキテクチャは、ARM® アーキテクチャのコプロセッサ拡張機能です。これは、*ANSI/IEEE 標準 754-1985: IEEE standard for Binary Floating-Point Arithmetic で定義されている単精度と倍精度の浮動小数点演算を行う機能です。*このドキュメントでは ANSI/IEEE 標準 754-1985 について以降は *IEEE754 標準*と呼びます。

最大 8 つの単精度、もしくは 4 つの倍精度のデータで構成されるショートベクタは、VFP アーキテクチャにより特に効率的に処理されます。ほとんどの演算命令はこれらのベクタデータに対して実行可能で、SIMD（*Single-Instruction, Multiple-Data*）命令による並列処理も可能になります。さらに、浮動小数点のロードとストア命令には*複数レジスタ*形式があり、ベクタデータをメモリとレジスタとの間で効率的に転送できます。

倍精度のサポートはオプションで、このこのオプションの有無はバリエーションの文字 D で示されます。したがって、VFPv1D バリエーションは単精度と倍精度の両方を持ち、VFPv1xD は単精度のみをサポートしています。デフォルトでは、倍精度オプションをサポートしています。

現在までに、VFP アーキテクチャには 2 つの主要なバージョンが定義されています。

- VFP アーキテクチャは、*ARM アーキテクチャリファレンスマニュアル*の第 2 版で導入されたものです。このバージョンは VFPv1 と呼ばれ、ARM10 rev0 で実装されたものです。
- VFPv2 は VFPv1 に置き換わるもので、P. C1-3 *「VFPv1 から VFPv2 への変更点」*で説明されているようなアーキテクチャの拡張が行われています。

VFP アーキテクチャの完全な実装には、*サポートコード*と呼ばれるソフトウェアコンポーネントが含まれている必要があります。サポートコードは、ハードウェアで実装されていない IEEE 754 準拠機能を提供します。詳細についてはP. C1-5 *「サポートコード」*を参照して下さい。

VFP ハードウェアと VFP サポートコードの間のインターフェース定義を、*サブアーキテクチャ*と呼びます。これは、オペレーティングシステムをサポートするための一貫したインターフェースを提供するのが目的です。

実装では、VFP 命令空間として CP10 と CP11 を使用します。一般に、CP10 は単精度演算のエンコード、CP11 は倍精度演算のエンコードに使用されます。使用されていないコードはすべて予約されています。

C1.1.1 浮動小数点モデルのサポート

アーキテクチャでは、以下に示すような各種のレベルで IEEE754 標準に準拠しています。

- 完全準拠モードでは、FPSCR の例外ビット（P. C2-23 *「FPSCR」*参照）によるユーザモードトラップ処理のサポートを含む完全な準拠が行われます。すべての実装では、トラップされた例外の処理が許可されている場合のサポートコードが存在している必要があります。
- 多くの実装では、IEEE 754 への完全な準拠は必要ではなく、VFPv2 アーキテクチャでは浮動小数点演算の総合的なパフォーマンスを向上する 2 つの非準拠モードが用意されています。これらのモードは、P. C2-14 *「Flush-to-Zero モード」*で説明されている *Flush-to-Zero* モードと、P. C2-16 *「デフォルト NaN モード」*で説明されている*デフォルト NaN* モードです。実装では、P. C1-6 *「ハードウェアとソフトウェアの実装」*で説明されているように、トラップが禁止されている場合に、これらのモードをすべてハードウェアでサポートすることもできます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C1.1.2 VFPv1 から VFPv2 への変更点

- VFPv2 では、FPSID 浮動小数点識別レジスタのビット [19:16] は 0b0001 です。VFPv1 では、これらのビットは 0b0000 です。FPSID レジスタの詳細については、P. C2-22 *「FPSID」*を参照して下さい。
- 新しいアーキテクチャには、VFP コプロセッサとの間で ARM レジスタの組をロード / ストアするための、`MRRC` および `MCRR` に代わる新しいコプロセッサ命令があります。

  FMDRR 2 つの 32 ビット ARM レジスタの内容を、倍精度 VFP レジスタに転送します。詳細については、P. C4-54 *「FMDRR」*を参照して下さい。

  FMRRD 倍精度 VFP レジスタの内容を、2 つの 32 ビット ARM レジスタに転送します。詳細については、P. C4-57 *「FMRRD」*を参照して下さい。

  FMSRR 2 つの 32 ビット ARM レジスタの内容を、1 組の単精度 VFP レジスタに転送します。詳細については、P. C4-70 *「FMSRR」*を参照して下さい。

  FMRRS 1 組の単精度 VFP レジスタの内容を、2 つの 32 ビット ARM レジスタに転送します。詳細については、P. C4-58 *「FMRRS」*を参照して下さい。

- FPSCR レジスタには、3 つの新しいビットがあります。

  DN ビット このビットをセットすると、デフォルト NaN モードが有効になります。詳細については、P. C2-16 *「デフォルト NaN モード」*を参照して下さい。

  IDE ビット 入力非正規トラップが許可されます。詳細については、P. C2-10 *「浮動小数点例外」*を参照して下さい。

  IDC ビット 入力非正規が検出されます。*Flush-to-Zero* モードでは、このビットは入力が 0 にフラッシュされたことを示します。これはスティッキービットです。つまり、このビットがセットされると、FPSCR レジスタへの明示的な書き込みによってクリアされるまでセットされた状態が続きます。詳細については、P. C2-10 *「浮動小数点例外」*を参照して下さい。

- Flush-to-Zero モードに変更が加えられています。詳細については、P. C2-10 *「浮動小数点例外」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C1.2 VFP アーキテクチャの概要

このセクションでは、VFP アーキテクチャの概要について説明します。アーキテクチャのより詳細な説明については、第 C2 章 *「VFP プログラマモデル」* を参照して下さい。

C1.2.1 レジスタ

VFP には 32 個の汎用レジスタがあり、それぞれが単精度浮動小数点数値か、32 ビット整数を保持できます。アーキテクチャの D バリエーションでは、これらのレジスタをペアとして使用し、16 個までの倍精度浮動小数点数値を保持することもできます。また、3 つかそれ以上のシステムレジスタが存在します。

FPSID 読み出し専用。読み出すことで、VFP アーキテクチャのどの実装が使用されているかを判定できます。

FPSCR ユーザレベルのすべてのステータスと制御を提供します。ステータスビットには、比較結果と浮動小数点例外の累積フラグが保持されます。制御ビットにより、丸めオプションとベクタの長さ / ストライドを選択し、浮動小数点例外トラップを許可できます。

FPEXC システムレベルのステータスと制御に使用するいくつかのビットが含まれています。

FPEXC レジスタの他のビット、および他のシステムレジスタはサブアーキテクチャ定義で、通常は VFP 実装のハードウェアとソフトウェアコンポーネントの間の内部的な通信に使用されます。詳細については P. C1-6 *「ハードウェアとソフトウェアの実装」* を参照して下さい。

ここに挙げたレジスタのほか、VFP アクセスはシステム制御コプロセッサのコプロセッサアクセスレジスタによっても制御されます。詳細については、P. B3-16 *「コプロセッサアクセスレジスタ」* を参照して下さい。VFP コプロセッサにアクセスするには、cp10 と cp11 フィールドを同時に更新し、両方のレジスタフィールドを必ず同じモード（ユーザアクセス、もしくは特権アクセス）にする必要があります。また、アクセス許可も両方同時にセットする必要があります。両方のフィールドの値が異なっている場合、動作は予測不能です。

*特権アクセス専用*としてマークされたレジスタや操作はすべて、特権アクセス許可が必要です。それ以外の場合、これらは未定義です。

C1.2.2 命令

以下の動作を行う命令が提供されています。

- 浮動小数点数値をメモリからレジスタにロードし、レジスタからメモリにストアします。これらの命令の一部は、複数のレジスタの値を転送することができ、浮動小数点について ARM の `LDM` と `STM` に対応する命令となります。これらの命令は、他の目的のほか、浮動小数点数値のショートベクタをロードまたはストアするためにも使用できます。
- 32 ビットの値を VFP と ARM の汎用レジスタとの間で直接転送します。
- 32 ビットの値を、VFP システムレジスタと ARM の汎用レジスタとの間で直接転送します。
- 浮動小数点レジスタの値の加算、減算、乗算、除算、平方根演算を行います。これらの命令は、個別の浮動小数点値とショートベクタの両方に使用できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- レジスタ間の浮動小数点数値のコピー。単純なコピーの他、その転送過程で符号ビットの反転またはクリア等、符号反転や絶対値変換をすることもできます。これらの命令はすべて、ショートベクタにも使用できます。
- 浮動小数点数値とショートベクタに対して結合積和演算を実行します。この演算は一般的な乗算、符号反転、加算、減算のシーケンスと比較して、効率的に領域を使用できます。
- 単精度数値、倍精度数値、符号なし 32 ビット整数、2 の補数形式の符号付き 32 ビット整数の間の変換を実行します。
- レジスタの浮動小数点数値を互いに、または 0 と比較します。

C1.2.3 浮動小数点例外

VFP アーキテクチャでは、IEEE754 標準で定義されている 5 つの浮動小数点例外がすべてサポートされています。

- 無効操作
- 0 による除算
- オーバフロー
- アンダーフロー
- 不正確

VFPv2 アーキテクチャでは、*「浮動小数点例外」*で説明されている、入力非正規浮動小数点例外のサポートが追加されています。

これらの例外は、非トラップとトラップの両方の形式でサポートされています。

例外の非トラップ処理

FPSCR の対応する累積フラグが 1 にセットされ、例外を生成した命令の結果レジスタは標準により指定されている結果の値に設定されます。その後で、例外を生成した命令を含むプログラムの実行が継続されます。

例外のトラップ処理

FPSCR の対応する制御ビットをセットすると、この動作が選択されます。例外が発生すると、トラップハンドラソフトウェアルーチンが呼び出されます。これは、アプリケーションソフトウェアにオーバフロー / アンダーフローなどの条件や、NaN と非正規の取り扱いに関する特別な必要条件がある場合に有用です。

トラップハンドラルーチンの呼び出し方法の詳細はサブアーキテクチャ定義です。

C1.2.4 サポートコード

トラップ浮動小数点例外が存在するため、VFP アーキテクチャの完全な実装には、*サポートコード*と呼ばれるソフトウェアコンポーネントが含まれている必要があります。VFP サポートコードの詳細については、*ARM アプリケーションノート 98* を参照して下さい。

VFP ハードウェアが VFP 命令に応答しない場合、通常は ARM 未定義命令ベクタ経由でサポートコードが実行されます。このソフトウェアエントリを*差し戻し*と呼びます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

差し戻し機構は、トラップ浮動小数点例外のサポートに使用されます。トラップ浮動小数点例外は *トラップ*と呼ばれ、解決のために実装によりアプリケーションソフトウェアに戻される必要がある浮動小数点例外です。詳細については、P. C1-5 *「浮動小数点例外」*を参照して下さい。サポートコードは、トラップ例外をキャッチし、トラップハンドラコールに変換する作業を行います。

サポートコードは、トラップハンドラコール以外の、実装により定義されているタスクを実行することもできます。この機能は、ハードウェアで実装が困難なまれな状況や操作に対処することや、ハードウェアではゲートを大量に使用するような操作を行うために使用できます。これにより、ハードウェアサポートの度合が異なっても、同じ機能が実現できます。

VFP の実装において、作業をハードウェアとソフトウェアのコンポーネントでどのように分割するかは実装定義です。

サポートコードとハードウェアの間のインターフェースの詳細はサブアーキテクチャ定義です。

C1.2.5 ハードウェアとソフトウェアの実装

VFP の実装は、ハードウェアコンポーネントを含むかどうかで分類されます。

ソフトウェア実装

この実装はソフトウェアのみで構成され、すべての浮動小数点演算は ARM ルーチンによりエミュレートされます。ソフトウェア実装は、*VFP エミュレータ*と呼ばれることもあります。

ソフトウェアのみの実装は、ソフトウェアの浮動小数点ライブラリを直接使用した場合よりもパフォーマンスがかなり低下するため、推奨されません。

ハードウェア実装

この実装には、ハードウェアとソフトウェア両方のコンポーネントが含まれます。一般に、パフォーマンス最適化のため、通常のケースはすべてハードウェアで処理するように設計が行われます。ハードウェアだけで処理できないようなケースが発生した場合、ソフトウェアコンポーネント（ハードウェアの *サポートコード*とも呼ばれます）が読み出され、処理を実行します。ハードウェアとそのサポートコードがどのようにインタラクションを行うかの詳細はサブアーキテクチャにて定義されます。

トラップ浮動小数点例外が禁止されている場合、VFP ハードウェア実装は、VFP アーキテクチャの実装定義サブセットをすべてハードウェアで実装していることを前提とできます。このサブセットのみを使用するアプリケーションは、サポートコードを必要としません。

通常の実装では、次の一般的なサブセットについて完全なハードウェアサポートを保証する必要があります。

- トラップ浮動小数点例外が禁止されている、完全な VFP 命令セット。
- トラップ浮動小数点例外が禁止されている、完全な VFP 命令セット。ただし、*近似値への丸め*（RN）モードが選択されている場合のみ。
- RN、*Flush-to-Zero*、*デフォルト NaN* モードが許可されている構成での、トラップ浮動小数点例外が禁止されている、完全な VFP 命令セット。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

このタイプの実装では、これらの構成を *RunFast* モードと呼びます。RunFast モードは、既存の VFP 実装の大部分に搭載されている機能で、それらの実装でパフォーマンスの向上を実現します。

- 不完全構成

  ハードウエアによる最小限の機能の実装には、完全な VFP レジスタバンクと、そのレジスタバンクで動作するロード、ストア、コピー命令のすべてが含まれます。

トラップ浮動小数点例外が許可されている場合、必ずソフトウェアコンポーネントが必要です。また、いかなるハードウェアコンポーネントによる実装も、すべてのモードにおいて一部の命令を完了するため、ソフトウェアコンポーネントを必要とすることがあります。

C1.2.6 ARM アーキテクチャとのインタラクション

VFP アーキテクチャは、ARM コプロセッサアーキテクチャに完全に準拠するように設計されています。すべての VFP 命令は、コプロセッサ番号 10 と 11 を使用する ARM 汎用コプロセッサ命令（`CDP`、`LDC`、`MCR`、`MRC`、`STC`）により発行されます。一般的な規則として、単精度命令にはコプロセッサ 10 が、倍精度命令にはコプロセッサ 11 が使用されます。

VFP 命令としての意味を割り当てられていないコプロセッサ 10 と 11 の命令はすべて、VFP アーキテクチャの将来の拡張用に予約されており、未定義として扱う必要があります。VFP アーキテクチャのハードウェアコプロセッサ実装はこれらの命令には応答せず、未定義命令例外が発生します。詳細については、P. A2-19 *「未定義命令例外」*を参照して下さい。

VFP コプロセッサからサポートコードを起動する推奨の方法は、同じ機構を使用します。

1. VFP ハードウェアを許可する前に、未定義命令ベクタにサポートコードをインストールします。
2. ハードウェアでサポートコードの補助が必要な場合、そのハードウェアは VFP 命令に応答しません。
3. その結果、未定義命令例外が発生し、サポートコードが実行されます。

このようなシステムでは、サポートコードがこれらの未定義命令例外を、予約命令によって引き起こされた例外と区別し、それぞれ異なるアクションを実行する必要があります。

ARM は、コプロセッサ命令が条件（P. A3-3 *「条件フィールド」*で説明されているもの）を満たしているかどうかを CPSR フラグを使用してテストし、条件を満たしていない場合は `NOP` として扱います。この場合、ARM プロセッサは命令を実行しないようにコプロセッサに対して通知するため、コプロセッサも命令を `NOP` として扱います。これは、すべての VFP 命令は、条件チェックが失敗した場合は `NOP` として扱われることを意味します。

条件コードチェックは、ARM プロセッサの CPSR フラグによるもので、VFP の FPSCR レジスタにある似た名前のフラグを使用するものではありません。条件付き実行に FPSCR フラグを使用するには、まず `FMSTAT` 命令によって CPSR に変換する必要があります。

VFP のロードとストア命令はデータアボートを生成できるため、VFP 実装はそのような命令で発生した任意のメモリアクセスによるデータアボートと共存可能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

割り込み

上に述べたように、ハードウェアによる VFP 実装は一般に、未定義命令例外を使用してハードウェアとソフトウェアのコンポーネント間で通信を行います。ソフトウェアによる VFP 実装でも、未定義命令例外を使用できます。これは、ハードウェアコプロセッサにより処理されないコプロセッサ命令はすべて、未定義命令として扱われるためです。

未定義命令例外が開始されると、IRQ は禁止され（P. A2-19 *「未定義命令例外」* 参照)、通常は命令ハンドラからの復帰までは再度許可されません。したがって、単純にシステムで VFP を使用すると、ワーストケースの IRQ レイテンシが大幅に増大します。

未定義命令ハンドラの開始直後に割り込みを明示的に再度許可すると、この IRQ レイテンシのペナルティを大幅に低減できます。これには、未定義命令ハンドラがオペレーティングシステムの他の部分に十分に統合されている必要があります。この実装を行う方法の詳細はシステムに大きく依存し、このマニュアルの範囲外です。

ハードウェア実装では、IRQ ハンドラ自体が VFP コプロセッサを使用する場合、IRQ レイテンシが増大する第 2 の原因となる可能性があります。これは、割り込まれたプログラムによってレイテンシの長い VFP 操作が開始されている場合、多くのサイクルの間 IRQ ハンドラが VFP ハードウェアの使用を禁止されるためです。

したがって、システムに割り込みレイテンシが短いことと、VFP 命令を使用することの両方が必要な IRQ ハンドラがある場合、レイテンシの最も長い VFP 命令の使用を避けることをお勧めします。特に、ベクタ除算命令とベクタ平方根命令は一般に非常にレイテンシが長いため、このようなシステムでは推奨されません。

注

FIQ は未定義命令ハンドラへのエントリ時に禁止されないため、FIQ のレイテンシは VFP 実装が未定義命令例外を使用する方法に影響されません。

ただし、これは同時に、VFP 実装のソフトウェアコンポーネントの実行中に、未定義命令ハンドラのエントリおよびイグジット（終了）シーケンスを含む、どの時点でも FIQ が発生する可能性があることを意味します。FIQ ハンドラが VFP 実装のステートを一切変更しない場合を除いて、すべてのケースを正しく処理できるように十分な注意が必要です。これは通常、FIQ が高速な割り込み処理を提供するという意図と相容れないため、FIQ ハンドラでは VFP を使用しないことを推奨します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C1.3 IEEE754 標準への準拠

VFP アーキテクチャは、IEEE 754 機能のサブセットを提供しています。次の操作は、標準では必須ですが、VFP アーキテクチャでは提供されていません。

- 剰余演算
- 2 進⇔ 10 進変換
- *浮動小数点数から整数値への丸め演算*
- VFP アーキテクチャの D バリエーションの場合、単精度値を倍精度に変換せず、単精度と倍精度の値を直接比較する操作

標準に完全に準拠した実装を実現するには、VFP アーキテクチャにこれらの操作を追加する必要があります（通常はソフトウェアライブラリルーチンの形で追加されます）。

——— 注 ———

一部の環境では、これらの演算すべてが必須ではありません。たとえば、C 言語では `float` と `double` を比較する場合、比較を実行する*前*に通常のバイナリ変換を使用して最初の引数を `double` に変換する必要があります。このため、C コードでは単精度値と倍精度値の直接比較は行われません。

———————

また、FPSCR の *Flush-to-Zero*（FZ) ビットが 1 にセットされている場合、VFP アーキテクチャで非正規化数とアンダーフロー例外を処理する方法は、標準に準拠していません。VFP アーキテクチャによって完全に準拠の動作を行うには、FZ ビットは 0 にセットする必要があります。詳細については P. C2-14「*Flush-to-Zero モード*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C1.4 IEEE 754 実装上の選択

IEEE 754 では浮動小数点システム設計上の多くの選択肢が実装オプションとして残されています。VFP アーキテクチャはこれらのうち選択可能なものを定義しています。このセクションの残りでは、これらの実装上の選択について簡単に説明します。

C1.4.1 サポートされているフォーマット

VFP アーキテクチャは、IEEE 754 で定義されたフォーマットの中で*基本単精度*浮動小数点フォーマットをサポートし、D バリエーションでは*基本倍精度*浮動小数点フォーマットもサポートしています。このマニュアルでは、これらを*単精度*と*倍精度*と呼びます。

標準の拡張形式はサポートされていません。

サポートされている整数形式は、符号なし32ビット整数と、2の補数形式の符号付き32ビット整数です。

C1.4.2 NaN

IEEE754 標準では、シグナル NaN とクワイエット NaN が最低 1 つづつあることと、NaN の表現をどうするかの部分的な定義（任意の NaN について、指数フィールドは最大値、仮数フィールドは非 0 であること）が指定されているだけです。VFP アーキテクチャでは、NaN がより完全に指定されています。

- それぞれのフォーマットで、指数フィールドが最大値、仮数フィールドが非 0 の値はすべて有効な NaN です。符号ビット、仮数フィールド、または両方が異なっている場合、別の NaN として扱います。
- シグナル NaN をフォーマットを変更してコピーした場合、無効演算例外は生成されません。
- シグナル NaN は、仮数の最上位ビットによりクワイエット NaN と区別されます。このビットが 0 の場合はシグナル NaN、1 の場合はクワイエット NaN です。
- VFP アーキテクチャには、各演算の結果が NaN である場合、どの NaN が生成されるかの正確な規則があります。これらの規則については、P. C2-5 「*NaN*」を参照して下さい。

——— 注 ———

符号または仮数ビットが異なる NaN は VFP アーキテクチャで別の NaN として扱われるという事実は、浮動小数点比較命令を使用してこれらを互いに区別できることを意味しません。IEEE754 標準では、すべての NaN が自身を含むあらゆる値との間で*順序付けなし*として比較されることが要求されています。

つまり、異なる NaN はその正確なビットパターンを調べる ARM コードを使用して区別でき、NaN の処理規則は標準により要求される場合を除いてNaN値のビットを変更しないように設計されています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C1.4.3　比較結果

比較命令の結果は、条件コードとして示されます。具体的には、これらはフラグの組み合わせ（N、Z、C、V）で、ARM プログラムのステータスレジスタで使用されているものと互換です。

命題をテストする別の手法に役立てるため､それぞれの比較命令は 2 つのバリエーションで提供され、NaN に関する処理はそれぞれ異なっています。4 つの比較結果のフラグ（N、Z、C、V）は、単一の ARM 条件チェックによりテストできる命題を最も多くするように選択されています。詳細については、P. C3-8「*IEEE 754 命題のテスト*」を参照して下さい。

C1.4.4　アンダーフロー例外

アンダーフローは IEEE754 の定義に基づき、*小ささ*に関しては*丸め前*検出にて、*不正確さ*に関しては*精度消失*の検出にて発生します。

C1.4.5　例外のトラップ

FPSCR には、例外のトラップが有効になっているかどうかを指定するビットが含まれており、VFP の実装は IEEE754 標準で定義されているトラップ例外が実際に発生するかどうかを判定します。トラップ例外の処理に関するそれ以上の詳細はすべてサブアーキテクチャ定義です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 C2 章
VFP プログラマモデル

本章では、VFP プログラマモデルの詳細を解説します。本章は以下のセクションから構成されています。

- *浮動小数点形式* : P. C2-2
- *丸め* : P. C2-9
- *浮動小数点例外* : P. C2-10
- *Flush-to-Zero モード* : P. C2-14
- *デフォルト NaN モード* : P. C2-16
- *浮動小数点汎用レジスタ* : P. C2-17
- *システムレジスタ* : P. C2-21
- *リセット時の動作と初期化* : P. C2-29

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.1 浮動小数点形式

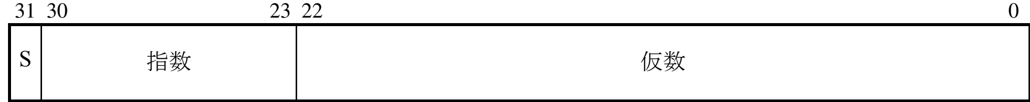
このセクションでは、IEEE754 標準により定義され、VFP アーキテクチャで使用されている基本的な単精度と倍精度の浮動小数点形式の概要について説明します。また、これらの形式のうち、標準では定義されていない VFP 固有の詳細についても解説します。

VFP アーキテクチャのすべてのバージョンとバリエーションは、単精度形式をサポートしています。D バリエーションは、倍精度形式もサポートしています。VFP アーキテクチャでは、IEEE754 標準で定義されている拡張形式はサポートしていません。

このセクションは、これらの形式と、それに含まれる各種の値について概要を説明することのみを目的としており、包括的な参照資料ではありません。詳細な情報、特に無限大、NaN、符号付き 0 の処理については、IEEE754 標準を参照して下さい。

C2.1.1 単精度形式

単精度値は 32 ビットワードで、メモリ上ではワード境界にアラインしている必要があります。単精度値の形式は次のとおりです。



表現される値は、主に指数フィールドにより決定されます。

- 0 < 指数 < `0xFF` の場合、値は*正規化数*で、次の値に等しくなります。

  $-1^{S} \times 2^{指数 - 127} \times (1.仮数)$

  値の*仮数部*は 1. 仮数で、次のように構成されます。

  — 1

  — 小数点

  — 23 個の仮数ビット

  したがって、仮数部の範囲は $1 \leq$ 仮数部 < 2 で、2^{-23} の整数倍です。

  値の*非バイアス指数*は、この式で計算される 2 のべき乗です。この場合、(指数 –127) です。

  最小の正の正規化数は、2^{-126}、または約 1.175×10^{-38} です。最大の正の正規化数は、$(2 - 2^{-23}) \times 2^{127}$、または約 3.403×10^{38} です。

- 指数 == 0 の場合、値は 0 か*非正規化数*で、仮数ビットにより決定されます。

  — 仮数 == 0 の場合、値は 0 です。

  0 には 2 種類の異なる 0 があります。

+0	S == 0 の場合
–0	S == 1 の場合

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

これら 2 種類の 0 は、ほとんどの場合同一に扱われます。+0 と –0 が浮動小数点数として比較された場合、*等しい*という結果が返されます。ただし、一部の例外的な状況では、これら 2 種類の 0 により異なる結果が返されます。たとえば、0 による除算の例外で得られるデフォルトの結果として生成される無限大の符号は異なります。また、2 つのワードを整数として比較した場合、互いに異なる数値として区別されます。

— 仮数 != 0 の場合、値は非正規化数で、次の値に等しくなります。

$-1^S \times 2^{-126} \times$（0. 仮数)

この場合、正規化された数値とは異なり、値の仮数部のうち小数点の前の部分は 0 です。仮数部の範囲は 0 < 仮数部 < 1 で、2^{-23} の整数倍です。値の非バイアス指数は –126 です。

最小の正の非正規化数は、2^{-149}、または約 1.401×10^{-45} です。

- 指数 == `0xFF` の場合、値は*無限大*か*非数*（NaN）で、仮数ビットにより決定されます。

  仮数 == 0 の場合、値は無限大です。これには 2 種類の異なる無限大があります。

  +∞ S == 0 の場合で、大きすぎるために正規化数として正確に表現できない正の数値すべてを表します。

  –∞ S == 1 の場合で、絶対値が大きすぎるために正規化数として正確に表現できない負の数値すべてを表します。

  仮数 != 0 の場合、値は NaN で、*クワイエット NaN* または*シグナル NaN* です（NaN の種類の詳細については P. C2-5 「*NaN*」を参照して下さい）。

  VFP アーキテクチャでは、2 種類の NaN は仮数の最上位ビット（ビット[22]）により区別されます。

  — ビット [22] == 0 の場合、その NaN はシグナル NaN です。符号ビットはどのような値も取りえます。また、仮数の他のビットはすべて 0 以外のどのような値も取りえます。したがって、シグナル NaN の可能な値は $2 \times (2^{22}-1) = 8388606$ 種類です。

  — ビット [22] == 1 の場合、その NaN はクワイエット NaN です。符号ビットと仮数の他のビットはどのような値もとり得ます。したがって、クワイエット NaN の可能な値は $2 \times 2^{22} = 8388608$ 種類です。

  VFP アーキテクチャでは、2 つの NaN は符号ビットと仮数ビットのいずれか、または両方が異なる場合、別の値として扱われます。これは、ワードで可能な 2^{32} 種類の値のすべてが、VFP アーキテクチャにより互いに異なる値として扱われることを意味します。

注

符号または仮数ビットが異なる NaN は別の NaN として扱われるという事実は、浮動小数点比較命令を使用してこれらを互いに区別できることを意味しません。これは、IEEE754 標準では、すべての NaN が自身を含むあらゆる値との間で*順序付けなし*として比較されることが要求されているためです。

ただし、整数比較を使用した場合、異なる NaN は区別されます。また、NaN を処理する規則は、ある NaN を別の NaN に変更しないように設計されています。詳細については、P. C2-5「*NaN*」を参照して下さい。

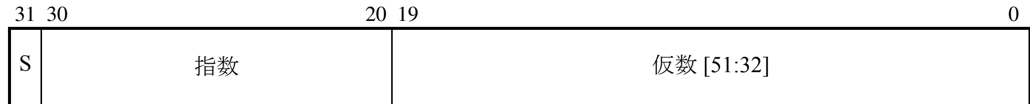
NaN に関する規則では、単精度レジスタを使用して、破壊する危険なしに整数値を保持できることも保証されています。詳細については、P. C2-20「*単精度レジスタへの整数の保持*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

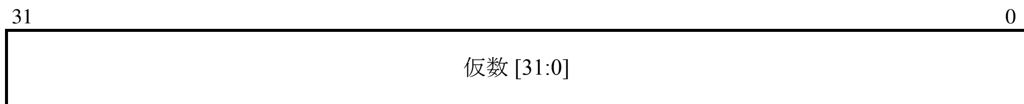
C2.1.2 倍精度形式

倍精度値は 2 つの 32 ビットワードで構成され、次の形式に従います。

最上位ワード



最下位ワード



メモリ上に置く場合、これら 2 つのワードは必ず連続し、両方のワードがワード境界にアラインしている必要があります。2 つのワードの順序は、メモリシステムのエンディアン方式に依存しています。

- リトルエンディアンのメモリシステムでは、最下位ワードが下位のメモリアドレスに、最上位ワードが上位のメモリアドレスに置かれます。
- ビッグエンディアンのメモリシステムでは、最上位ワードが下位のメモリアドレスに、最下位ワードが上位のメモリアドレスに置かれます。

VFP実装では、その付属しているARM® 実装と同じエンディアン形式を使用する必要があります。ARM 実装でエンディアン形式が構成可能な場合、メモリシステムに合わせて ARM プロセッサのエンディアン形式が設定されるまで、倍精度値のロードやストアは実行できません。詳細については、P. A2-30「*エンディアンのサポート*」を参照して下さい。

——— 注 ———

ここで VFP アーキテクチャについて定義されるワード順序は、以前の FPA 浮動小数点アーキテクチャのものとは異なっています。FPA アーキテクチャでは、メモリシステムのエンディアン形式にかかわらず、常に最上位ワードが下位のメモリアドレスに、最下位ワードが上位のメモリアドレスに置かれます。

倍精度値は、単精度値と同様に数値、無限大、NaN を表現できます。

- 0 < 指数 < `0x7FF` の場合、値は正規化数で、次の値に等しくなります。

  $-1^{S} \times 2^{指数 - 1023} \times$（1. 仮数)

  値の仮数部は 1. 仮数の数値で、1、小数点、52 個の仮数ビットの順に構成されます。したがって、仮数部の範囲は $1 \leq$ 仮数部 < 2 で、2^{-52} の整数倍です。

  値の非バイアス指数は（指数 – 1023）です。

  最小の正の正規化数は、2^{-1022}、または約 2.225×10^{-308} です。最大の正の正規化数は、$(2-2^{-52}) \times 2^{1023}$、または約 1.798×10^{308} です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- 指数 == 0 の場合、値は 0 か非正規化数で、仮数ビットにより決定されます。

  仮数 == 0 の場合、値は 0 です。単精度の 0 と同様に、2 つの異なる 0 があり、単精度の場合と同様に動作します。

  +0　　　S == 0 の場合

  –0　　　S == 1 の場合

  仮数 != 0 の場合、値は非正規化数で、次の値に等しくなります。

  $-1^S \times 2^{-1022} \times$（0. 仮数)

  この場合、正規化された数値とは異なり、値の仮数部のうち小数点の前の部分は 0 です。仮数部の範囲は 0 < 仮数部 < 1 で、2^{-52} の整数倍です。値の非バイアス指数は –1022 です。

  最小の正の非正規化数は、2^{-1074}、または約 4.941×10^{-324} です。

- 指数 == `0x7FF` の場合、値は無限大か NaN で、仮数ビットにより決定されます。

  仮数 == 0 の場合、値は無限大です。単精度の場合と同様に、2 種類の異なる無限大があります。

  +∞　　　S == 0 の場合で、プラス無限大を表します。

  +∞　　　S == 1 の場合で、マイナス無限大を表します。

  仮数 != 0 の場合、値は NaN で、*クワイエット NaN* または*シグナル NaN* です（NaN の種類の詳細については「*NaN*」を参照して下さい）。

  VFP アーキテクチャでは、2 種類の NaN は仮数の最上位ビット（最上位ワードのビット [19]）により区別されます。

  — ビット [19] == 0 の場合、その NaN はシグナル NaN です。符号ビットはどのような値も取りえます。残りの仮数ビットは、すべて 0 以外のどのような値も取りえます。

  — ビット [19] == 1 の場合、その NaN はクワイエット NaN です。符号ビットと、残りの仮数ビットは、どのような値もとり得ます。

  2 つの NaN の符号ビット、仮数、または両方が異なる場合、その 2 つは異なる NaN です。

C2.1.3　NaN

NaN は特別な浮動小数点値で、数値も無限大も適切でない場合に使用されます。NaN には 2 種類あり、それぞれが各種の目的に使用できます。

クワイエット NaN　これらの NaN は、ほとんどの浮動小数点演算で変更されず維持されます。これらは一部のまれな状況で、他の意味のある結果が存在しない場合、浮動小数点算術演算により生成可能です。それ以後の計算は、結果としてクワイエット NaN を生成した演算の結果に依存します。クワイエット NaN は、関連する無効演算例外がトラップされない場合のみ、この方法で生成されます。トラップされる場合、代わりにトラップハンドラが呼び出されます。クワイエット NaN の他の一般的な用途は、存在しない、または利用不能なデータの値を示すことです。この場合、存在しない値に依存する演算の結果も、すべてクワイエット NaN になります。

シグナル NaN　浮動小数点演算のオペランドにシグナル NaN が指定された場合、常に無効演算例外が発生します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

シグナル NaN の 1 つの使用法は、デバッグ時に一部の初期化されていない変数の使用を追跡することです。このためには、メモリにシグナル NaN のコピーを事前にロードしておき、無効演算トラップが有効な状態でプログラムをロードし、実行します。浮動小数点演算に、初期化されていないメモリからロードされたオペランドが指定された場合、無効演算トラップハンドラが呼び出されます。

IEEE754 標準では、2 種類の NaN を区別する方法や、浮動小数点システムに存在できる異なる NaN の種類の数が定義されていません。これらの詳細は VFP アーキテクチャにより定義されています。詳細については P. C2-2「*単精度形式*」と P. C2-4「*倍精度形式*」を参照して下さい。

以下のサブセクションでは、NaN に関する浮動小数点演算の動作についての IEEE754 標準による主要な必要条件と、VFP アーキテクチャでそのような演算に対して要求される追加の必要条件について説明します。

非浮動小数点の結果を持つ命令

VFP アーキテクチャには、浮動小数点値を整数に変換する命令が含まれています。IEEE754 標準に従い、これらの命令はオペランドが NaN の場合は、それがシグナル NaN かクワイエット NaN かにかかわらず、常に無効命令例外を生成します。この例外がトラップされない場合、VFP アーキテクチャでは整数の結果を 0 にする必要があります。

VFP アーキテクチャには比較命令も含まれており、これらは結果を条件コードとして生成します。これらの命令は、オペランドがシグナル NaN の場合に無効命令例外を生成します。オペランドがクワイエット NaN の場合、一部の命令は同様に無効演算例外を生成し、他の命令は条件コードに*順序付けなし*の結果を生成します。無効演算例外が生成され、その例外がトラップされない場合も、条件コードの結果は常に*順序付けなし*になります。詳細については、P. C3-6「*比較命令*」を参照して下さい。

NaN を処理する他の VFP 命令はすべて、結果が浮動小数点数値となります。

浮動小数点の結果を持つ命令

浮動小数点の結果を持つ演算で、1 つ以上のオペランドが NaN の場合、IEEE754 標準では次のような動作が要求されます。

- NaN オペランドのいずれかがシグナル NaN の場合、無効演算例外を生成する必要があります。この例外がトラップされない場合、結果はクワイエット NaN の必要があります。
- NaN オペランドがすべてクワイエット NaN の場合、結果はクワイエット NaN で、NaN オペランドのいずれかに等しい必要があります。

——— 注 ———

この目的から、標準では浮動小数点数値に対する一部のコピー操作を非浮動小数点演算として扱い、この形式で NaN が処理されないようにすることが許容されています。VFP アーキテクチャでは、これらのコピー操作を非浮動小数点演算として扱うことが必須となっています。

これによって影響を受ける命令については、P. C3-13「*コピー、符号反転、絶対値命令*」、P. C3-14「*ロードとストア命令*」、P. C3-18「*単一レジスタ転送命令*」を参照して下さい。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

VFP アーキテクチャの浮動小数点命令のほとんどは、オペランドと結果に同じフォーマットを使用します。これらに関して VFP アーキテクチャでは、上の各ケースについて、正しいクワイエット NaN 結果を次のように定義しています。

1. ベクタオペランドに対して命令を実行する場合、ベクタのエレメントに対する個別の演算について独立に、次の規則 2 から 4 までが適用されます。

2. FMAC、FMSC、FNMAC、FNMSC 命令はそれぞれ 2 つの浮動小数点演算を指定し、各演算に 2 つのオペランドがあります。最初の演算のオペランドのいずれかが NaN の場合、演算の結果は次の規則 3 と規則 4 に従って決定されます。次に、3 番目のオペランドと最初の演算の結果が（FNMAC と FNMSC では符号ビットが反転されます)、次の演算のオペランドになります。これらのいずれかが NaN の場合、最終結果は次の規則 3 と規則 4 に従って決定されます。

3. オペランドがシグナル NaN の場合、結果はクワイエット NaN で、そのオペランドをコピーし、仮数の最上位ビットを 0 から 1 に変更した値になります。2 つのオペランドがある演算で、両方のオペランドがシグナル NaN の場合、最初のオペランドからこの方式で結果が生成されます。

4. どちらのオペランドもシグナル NaN でないが、片方のオペランドがクワイエット NaN の場合、結果はクワイエット NaN オペランドのコピーになります。2 つのオペランドがある演算で、両方のオペランドがクワイエット NaN の場合、最初のオペランドがコピーされ、結果が生成されます。

IEEE754 標準では、オペランドが NaN でない特定の演算について、無効演算例外を生成することも指定されています。浮動小数点の結果を生成する次の演算は、この例外を生成する可能性があります。

- 加算で、2 つのオペランドが符号の異なる無限大の場合。この影響を受ける VFP 命令は、FADD、FMAC、FNMAC です。
- 減算で、2 つのオペランドが符号の同じ無限大の場合。この影響を受ける VFP 命令は、FMSC、FNMSC、FSUB です。
- 乗算で、片方のオペランドが 0、もう片方が無限大の場合。この影響を受ける VFP 命令は、FMAC、FMSC、FMUL、FNMAC、FNMSC、FNMUL です。
- 除算で、両方のオペランドが 0、または両方のオペランドが無限大の場合。この影響を受ける VFP 命令は FDIV です。
- 平方根で、オペランドが負の値の場合。これには、–∞（マイナス無限大）を含みますが、–0 は除きます。この影響を受ける VFP 命令は FSQRT です。

どの場合も、この例外がトラップされない場合、結果はクワイエット NaN の必要があります。VFP アーキテクチャでは、これらの場合に生成されるクワイエット NaN の符号ビットは 0、仮数の最上位ビットは 1、他の仮数ビットはすべて 0 の必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

特別なケース

オペランドと結果の浮動小数点形式が異なる命令が 2 つあります。これらの命令には、NaN の処理について次のような特別の規則があります。

- FCVTDS 命令は、単精度値を倍精度に変換します。オペランドが単精度のクワイエット NaN の場合、結果は次のような倍精度クワイエット NaN です。

  ```
  S               = S bit of operand
  fraction[51:29] = fraction[22:0] of operand
  fraction[28:0]  = 0
  ```

  オペランドが単精度シグナル NaN の場合、無効演算例外が生成されます。この例外がトラップされない場合、結果は次のような倍精度のクワイエット NaN になります。

  ```
  S               = S bit of operand
  fraction[51]    = 1
  fraction[50:29] = fraction[21:0] of operand
  fraction[28:0]  = 0
  ```

- FCVTSD 命令は、倍精度値を単精度に変換します。オペランドが倍精度のクワイエット NaN の場合、結果は次のような単精度クワイエット NaN です。

  ```
  S               = S bit of operand
  fraction[22:0]  = fraction[51:29] of operand
  ```

  オペランドが倍精度シグナル NaN の場合、無効演算例外が生成されます。この例外がトラップされない場合、結果は次のような単精度のクワイエット NaN になります。

  ```
  S               = S bit of operand
  fraction[22]    = 1
  fraction[21:0]  = fraction[50:29] of operand
  ```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.2 丸め

浮動小数点演算は本質的に精度が制限されています。数値演算の正確な結果には、デスティネーションに定義された形式で表現できる値よりも高い精度の値が多くの場合含まれているためです。この問題を解決するために、結果はデスティネーションの形式に合わせて*丸め*られ、その形式において正確な演算結果を近似する数値が選択されます。

IEEE754 標準では、4 つの*丸めモード*が定義されており、それぞれについて演算の正確な丸め結果が指定されています。丸めモードに関する以下の説明で、*丸め誤差*は次の値と定義されます。

（丸め結果）-（正確な結果)

丸めモードは次のとおりです。

近似値への丸め（**RN**）モード

このモードでは、丸め結果は丸め前の結果に最も近い表現可能な数値、つまり丸め誤差の絶対値が最も小さくなる数値になります。丸め前の結果が、表現可能な 2 つの数値の正確に中間の場合、最下位ビットが 0 の方の数値が使用されます。

これはデフォルトの丸めモードで、一般に最も正確な結果が得られます。他の丸めモードは主に、区間演算など特化された目的に使用されます。

プラス無限大への丸め（**RP**）モード

このモードでは、丸め結果は正確な結果に等しいか、正確な結果より大きく、正確な結果に最も近い表現可能な数値、つまり丸め誤差の絶対値が最も小さくなる数値になります。ただし、ここで（丸め誤差）≥ 0 という条件があります。正確な結果が、デスティネーションの形式で表現可能な最大の正の正規化数よりも大きい場合、丸め結果は $+\infty$（プラス無限大）になります。

マイナス無限大への丸め（**RM**）モード

このモードでは、丸め結果は正確な結果に等しいか、正確な結果より小さく、正確な結果に最も近い表現可能な数値、つまり丸め誤差の絶対値が最も小さくなる数値になります。ただし、ここで（丸め誤差）≤ 0 という条件があります。正確な結果が、デスティネーションの形式で表現可能な（絶対値が）最大の負の正規化数よりも小さい場合、丸め結果は $-\infty$（マイナス無限大）になります。

ゼロへの丸め（**RZ**）モード

このモードでは、丸め結果は正確な結果よりも絶対値が大きくない、正確な結果に最も近い表現可能な数値、つまり、丸め結果の絶対値 $\leq$ 正確な結果の絶対値という条件を満たした上で、丸め誤差の絶対値が最小になる数値が選ばれます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.3 浮動小数点例外

IEEE754 標準では、5 種類の浮動小数点例外が指定されています。

無効演算例外

この例外は、浮動小数点演算の結果として、数値もしくは無限大で表すには適当でない各種のケースと、浮動小数点演算のオペランドがシグナル NaN の場合に発生します。無効演算例外の詳細については、P. C2-5「*NaN*」を参照して下さい。

0 による除算例外

この例外は、正規化数または非正規化数が 0 により除算されたときに発生します。

オーバフロー例外

この例外は、2 つの浮動小数点数値に対する数値演算の結果の絶対値が大きすぎるため、デスティネーションの形式では、使用されている丸めモードでの大きな丸め誤差なしには表現不能である場合に発生します。

より正確には、浮動小数点演算の*理想的な丸め結果*は、仮にデスティネーションの形式で非バイアス指数の範囲に制限がなかったならば、その丸めモードの結果として生成されるであろう数値として定義されています。理想的な丸め結果の非バイアス指数が、デスティネーションの形式には大きすぎる場合（つまり、単精度では > 127、倍精度では > 1023 の場合)、実際の丸め結果とは異なることになり、オーバフロー例外が発生します。

アンダーフロー例外

この例外が発生する条件は、*Flush-to-Zero* モードが使用されているかどうかと、*アンダーフロー例外イネーブル*（UFE）ビット（FPSCR のビット [11]）の値によって変化します。

Flush-to-Zero モードが使用されておらず、UFE ビットが 0 の場合、アンダーフローは浮動小数点演算の丸め前の結果が 0 < 結果の絶対値 < MinNorm の条件を満たし、最終結果が不正確（つまり、丸め前の結果とは異なる値）な場合に発生します。ここで、MinNorm は単精度の場合 2^{-126}、倍精度の場合 2^{-1022} です。

Flush-to-Zero モードが使用されておらず、UFE ビットが 1 の場合、アンダーフローは浮動小数点演算の丸め前の結果が 0 < 結果の絶対値 < MinNorm の条件を満たす場合に発生します。この場合、最終結果が不正確かそうでないかは関係ありません。

Flush-to-Zero モードで発生するアンダーフロー例外は、UFE ビットの実際の値にかかわらず、常に非トラップとして扱われます。この詳細と、*Flush-to-Zero* モードの他の要素については、P. C2-14「*Flush-to-Zero モード*」を参照して下さい。

——— 注 ———

IEEE754 標準では、アンダーフロー例外の定義において、2 つの判定方法が許されます。VFP アーキテクチャにおける上記の説明は以下の規格上の言葉で表されます。

- *丸め前*検出による*小ささ*の判定
- *精度の低下*による*不正確さ*の判定

Flush-to-Zero モードでは、丸め前検出により小ささの判定が用いられます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

不正確例外

2 つの浮動小数点数値に対して行われる数値演算の結果には、デスティネーションレジスタに保持できるより多い有意ビットが含まれていることがあります。この場合、結果はデスティネーションレジスタに保持できる値に丸められ、この結果は*不正確*と呼ばれます。

不正確例外は、次の条件が起きた場合に常に発生します。

- 結果が、丸め前の計算結果と等しくない。
- 非トラップオーバフロー例外が発生した。
- *Flush-to-Zero* モードでないとき、非トラップアンダーフロー例外が発生した。

——— 注 ———

不正確例外は、通常の浮動小数点計算でも時々発生し、一部の特化されたアプリケーション以外では、重要な数値エラーを示しているわけではありません。不正確例外を有効にすると、コプロセッサのパフォーマンスが大きく低下する場合があります。

VFP アーキテクチャでは、追加の例外が 1 つ定義されています。

入力非正規例外

この例外は *Flush-to-Zero* モードで、数値演算への入力が非正規化数の場合のみ発生し、この入力は 0 と見なされます。

この例外は非数値演算、FABS、FCPY、FNEG では発生しません。これらの命令の詳細については、P. C3-13 *「コピー、符号反転、絶対値命令」*を参照して下さい。

これらの例外はそれぞれ 2 つの方法のいずれかで処理され、どちらの方法が選択されるかはその例外に関連付けられた*トラップイネーブルビット*で決定されます。

トラップイネーブルビットが **0** の場合

例外の非トラップ処理が選択されます。

この場合、演算の結果は IEEE754 標準で定義されているデフォルト値となり、その例外に関連付けられた*累積例外ビット*が 1 になります。それぞれの例外について、結果の値が決定される方法を P. C2-12 表 C2-1 に示します。

累積例外ビットが 0 になるのは、FMXR 命令を使用して FPSCR に明示的な書き込みが行われた場合のみです。他の浮動小数点命令はこのビットを変更しないか（非トラップ例外が発生しなかった場合)、非トラップ例外が発生した場合にビットの 1 つ以上を 1 にセットするか、いずれかの動作のみを行います。このため、プログラムで一連の計算の前にこれらのビットを 0 にセットし、計算の終了後にテストすることで、計算の間に非トラップ例外が発生したかどうかを知ることができます。

トラップイネーブルビットが **1** の場合

例外のトラップ処理が選択されます。

この場合、例外のトラップハンドラルーチンが呼び出されます。トラップハンドラの選択方法と、トラップハンドラを呼び出すためのインターフェースは実装定義です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

トラップハンドラルーチンの呼び出しは、*不正確*であることが許容されます。つまり、例外の発生した浮動小数点命令の実行よりも後の時点で呼び出しが発生してもかまいません。ただし、この呼び出しは、その命令の結果に依存する以後の命令や、*直列化命令*の実行前に必ず発生します（直列化命令の詳細については、P. C4-62「*FMRX*」とP. C4-77「*FMXR*」を参照して下さい)。

例外のトラップ処理では、累積例外ビットはセットされません。この動作が必要な場合、トラップハンドラルーチンで FMRX/ORR/FMXR のシーケンスを FPSCR に対して使用し、ビットをセットできます。

表 C2-1 例外のデフォルトの結果

例外タイプ	正の符号についてのデフォルトの結果	負の符号についてのデフォルトの結果
無効演算	クワイエット NaN	クワイエット NaN
0 による除算	+∞（プラス無限大)	–∞（マイナス無限大)
オーバフロー	RN、RP: +∞（プラス無限大) RM、RZ: +MaxNorm	RN、RM: –∞（マイナス無限大) RP、RZ: –MaxNorm
アンダーフロー	通常の丸められた結果	通常の丸められた結果
不正確	通常の丸められた結果	通常の丸められた結果
入力非正規	通常の丸められた結果	通常の丸められた結果

表 C2-1 には、次の注記が適用されます。

- 無効演算例外については、デフォルトの結果として生成されるクワイエット NaN の詳細についてP. C2-5「*NaN*」を参照して下さい。
- 0 による除算例外については、デフォルトの結果は除算に関する通常の判定と同様に、符号ビットに依存します。つまり、2 つのオペランドの符号ビットの排他的論理和により決定されます。
- オーバフロー例外については、デフォルトの結果はオーバフローする演算に関する通常の判定と同様に、符号ビットに依存し、同時に使用されている丸めモードにも依存します。MaxNorm は、デスティネーションの精度で表現可能な最大の正規化数を示します。

C2.3.1 例外の組み合わせ

同じ演算で、複数の例外が同時に発生することがあります。同時に発生する可能性がある例外の組み合わせは、オーバフロー / 不正確とアンダーフロー / 不正確のみです。これらの場合、次に示すように不正確例外の優先度が低くなります。

- オーバフローまたはアンダーフロー例外がトラップされた場合、トラップハンドラが呼び出されます。トラップハンドラのパラメータに不正確例外の情報が含まれるかどうかは実装定義です。それ以外の点では、このケースでは不正確例外は無視されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- オーバフローまたはアンダーフロー例外がトラップされない場合、累積ビットが 1 にセットされ、デフォルトの結果が評価されます。次に、不正確例外が通常に処理され、このデフォルトの結果は演算の通常の丸められた結果として扱われます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.4 Flush-to-Zero モード

一部の VFP 実装は、非正規化数とアンダーフロー例外が関与する計算を実行するときに、通常よりもパフォーマンスが大きく低下します。特に、正規化数と 0 のみをハードウェアで扱い、他の種類の値についてはサポートコードを呼び出すハードウェア実装ではこの現象が発生します。

アルゴリズムで、オペランドと中間結果の多くが非正規化数の場合、これによってパフォーマンスが大きく低下することがあります。このようなアルゴリズムの一部では、非正規化オペランドと中間結果を 0 で置き換えると、最終結果の精度に大きな影響を与えることなく、パフォーマンスを改善できます。ただし、この方法はすべてのアルゴリズムに適用可能ではありません。この最適化を可能にするため、VFP 実装には *Flush-to-Zero* モードという特殊な処理モードがあります。

Flush-to-Zero モードの動作は、以下の点で通常の IEEE 754 数値演算とは異なっています。

- 浮動小数点演算の入力のうち、非正規化数であるものはすべて 0 として扱われます。これによって、入力非正規例外が発生します。この例外は、*Flush-to-Zero* モードでのみ発生します。関連するトラップが禁止されている場合、FPSCR の IDC ビットがセットされるだけです。
- 浮動小数点演算の丸め前の結果が 0 < 結果の絶対値 < MinNorm の条件を満たす場合、結果は 0 にフラッシュされます。ここで、MinNorm は単精度の場合は 2^{-126}、倍精度の場合は 2^{-1022} です。これによって、FPSCR の UFC ビットがセットされます。
- アンダーフロー例外は、Flush-to-Zero モードで結果が 0 にフラッシュされた場合のみ発生します。これらの例外は常に非トラップとして扱われ、FPSCR のアンダーフロートラップイネーブル（UFE）ビットは無視されます。
- Flush-to-Zero モードでは、入力または結果が 0 にフラッシュされた場合に不正確例外は発生しません。これらは、IEEE 754 の規則に従い、結果が通常に丸められた場合に発生します。

入力または結果が 0 にフラッシュされた場合、その 0 の符号ビットの値は VFPv2 では実装定義です。実装では、常に符号ビットを変更せず残すことを選択できます。将来のバージョンのアーキテクチャではこれが必須となる予定です。VFPv2 では、実装で常に正の 0 にフラッシュすることも選択できます。

Flush-to-Zero モードの目的から、コピー操作は浮動小数点演算としては扱われません。Flush-to-Zero モードにより影響を受ける演算は、オペランドがシグナル NaN の場合に無効演算例外を生成しない演算とまったく同じです。詳細については、P. C3-13 *「コピー、符号反転、絶対値命令」*、P. C3-14 *「ロードとストア命令」*、P. C3-18 *「単一レジスタ転送命令」*を参照して下さい。

注

Flush-to-Zero モードは IEEE754 標準とは非互換で、IEEE 754 との互換性が必要な場合は使用できません。Flush-to-Zero モードの扱いには十分な注意が必要です。上に述べたように、一部のアルゴリズムでは大きなパフォーマンス向上が実現できることがありますが、このモードを使用した場合に注意すべき点も多く存在します。これはアプリケーションに依存します。

- 多くのアルゴリズムでは、通常は非正規化数を使用しないため、有意な効果はありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- 他の多くのアルゴリズムでは、例外が発生する、またはアルゴリズムの結果の精度が大きく低下することがあります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.5 デフォルト NaN モード

VFPv2 では、*デフォルト NaN モード*が導入されています。

FPSCR の DN ビット（ビット [25]）をセットすると、デフォルト NaN モードが選択されます。デフォルトは 0（禁止）です。このビットをセットすると、IEEE 754 との整合性はありますが、現在の汎用または組み込みシステムとは異なる動作が指定されます。

IEEE 754 では、NaN が関与する演算はクワイエット NaN を返すことが指定されています。現在のほとんどの浮動小数点実装では、返される仮数ビットは入力 NaN の仮数ビットで、複数の入力 NaN がある場合はそのいずれかが返されます。どの入力 NaN が返されるかはアーキテクチャで指定されています。これは、VFPv2 でデフォルト NaN モードが有効になっていない場合の動作です。

デフォルト NaN モードでは、1 つまたは複数の入力 NaN（クワイエット NaN かシグナル NaN かに関係なく）が関与するすべての数値演算、または無効な結果として NaN が返される演算では、常にデフォルト NaN が返されます。デフォルト NaN のフォーマットを表 C2-2 に示します。

非数値演算の FCPY、FABS、FNEG は、NaN を仮数ビットを変えずに処理します。これらの命令で NaN を処理する場合、例外ステータスビットはセットされません。

C2.5.1 無効演算例外

無効演算例外の機能は、デフォルト NaN モードの影響を受けません。これは、IEEE 754 仕様の、NaN 入力に関する無効演算例外の仕様により規定されています。

ハードウェア実装では、デフォルト NaN モードの場合は NaN 値をハードウェアで処理し、それ以外の NaN 値はサポートコードを呼び出して処理することを選択できます。このような実装では、デフォルト NaN モードで動作している場合、入力に NaN が存在すると、入力 NaN の 1 つ以上がシグナル NaN であり、FPSCR の IOE トラップイネーブルビットがセットされている場合のみ、サポートコードが起動されます。

C2.5.2 デフォルト NaN のフォーマット

ARM 浮動小数点プロセッサのデフォルト NaN は表 C2-2 に示すとおりです。

表 C2-2 デフォルト NaN のエンコード

	単精度	倍精度
符号	0	0
指数	0xFF	0x7FF
仮数	ビット [22] == 1	ビット [51] == 1
	ビット [21:0] == 0	ビット [50:0] == 0

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.6 浮動小数点汎用レジスタ

VFP 実装には 32 個の汎用レジスタがあり、それぞれが単精度浮動小数点数値か、32 ビット整数を保持できます。これらは S0 ～ S31 と呼ばれます。

VFP アーキテクチャの D バリエーションでは、これらのレジスタを 16 個の倍精度レジスタとして扱うこともできます。この場合、これらのレジスタは D0 ～ D15 と呼ばれます。倍精度レジスタ D0 は、単精度レジスタ S0 および S1 とオーバラップしています。倍精度レジスタ D1 は単精度レジスタ S2 および S3 とオーバラップしています。以下同様に、図 C2-1 に示すようにレジスタがオーバラップしています。

S0	S1		D0
S2	S3		D1
S4	S5		D2
S6	S7		D3
S8	S9		D4
S10	S11		D5
S12	S13		D6
S14	S15		D7
S16	S17	相互にオーバラップ	D8
S18	S19		D9
S20	S21		D10
S22	S23		D11
S24	S25		D12
S26	S27		D13
S28	S29		D14
S30	S31		D15

図 C2-1 VFP 汎用レジスタ

倍精度レジスタと、対応する単精度レジスタの組とのマッピングは次のとおりです。

- S<2n> は、D<n> の下位半分となります。
- S<2n+1> は、D<n> の上位半分となります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.6.1 精度が不明な値のストアと再ロード

——— 注 ———

FLDMX 命令と FSTMX 命令は ARMv6 では推奨されません。データの精度が不明な場合、値の保存と復元には FLDMD と FSTMD を使用して下さい。

プログラムで、レジスタの値が単精度値か倍精度値かを判定せず、レジスタの値をメモリにストアし、後で再ロードすることが必要な場合があります。このような状況の一般的な 2 つの例を示します。

- プロシージャ呼び出しでは、レジスタが*呼び出し先保存レジスタ*として指定されることがあります (呼び出されたプロシージャがそれらのレジスタを保存する必要がある)。呼び出されたプロシージャで呼び出し先保存レジスタを使用する必要がある場合、エントリシーケンスでレジスタの値をスタックにストアする必要があります。後で、プロシージャの復帰シーケンスではそれらの値をスタックから再ロードし、レジスタの元の内容を復元する必要があります。

  ただし、スタックにストアされるレジスタの内容は、その内容が呼び出し元でどのように使用されるかによって異なり、呼び出し元によってレジスタの使い方は異なります。このため、呼び出されたプロシージャのエントリシーケンスは呼び出し先保存レジスタを、精度が不明な値を含むレジスタとして扱う必要があります。

- プロセススワップコードは、プロセスをスワップアウトするときにレジスタの内容をストアし、そのプロセスがスワップインされるときに再ロードする必要があります。プロセスによってレジスタの使用法は異なるため、プロセススワップコードは VFP レジスタの値を精度が不明なものとして扱う必要があります。

このような場合、2 つの VFP 命令（FLDMX 命令と FSTMX 命令）を使用します。これらの命令は、ソースの精度が命令の精度と一致する必要があるという通常の規則の例外になります。

FSTMX　1 つ以上の倍精度レジスタをストアします。

FLDMX　対応する FSTMX でストアされたレジスタを再ロードします。

対応する FLDMX は、レジスタに含まれていた元の値が単精度値と倍精度値のどちらであったかにかかわらず、レジスタの元の内容を正しく再ロードします。この目的では、*対応する FLDMX* とは、FSTMX でストアされたのと正確に同じレジスタのセットに対してロードを実行し、その FSTMX と同じメモリアドレスを生成するものを指します。

FSTMX でストアされたデータに対して通常実行される操作は、対応する FLDMX による再ロードのみです。ただし、デバッグソフトウェアではスタックフレームまたはプロセス制御ブロックの内容を解析 / 変更することが必要な場合があるため、FSTMX/FLDMX のメモリ形式を知ることが必要な場合があります。

FSTMX はレジスタの内容を、同じレジスタに対する FSTMD がストアするのと正確に同じ方法でストアします。最初の倍精度レジスタの値は 2 つのワードとして、プロセッサで構成されているエンディアン形式に従った正しい順序でストアされます。次に、2 番目の倍精度レジスタの値が同様にストアされ、以下同様に N 番目までのレジスタは 2N メモリワードにストアされます。(2N + 1) 番目のメモリワードは使用されません。

対応する FLDMX 命令は、同じレジスタに対する FLDMD と正確に同じ方法で、データを倍精度値として再ロードします。実装では、この方法で倍精度レジスタを再ロードする場合、仮にそれらのレジスタに単精度値が含まれていても内容が正しく再ロードされることを保証する必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例

たとえば、図 C2-2 は次の命令の動作を示しています。

```
FSTMIAX Rn,{D4-D6}
```

この例では、D4 と D6 に倍精度値が含まれており、S10 と S11（D5 とオーバラップしています）には単精度値が含まれていると想定しています。

アドレス	
Rn+24	未使用
Rn+20	D6、後半ワード
Rn+16	D6、前半ワード
Rn+12	D5、後半ワード
Rn+8	D5、前半ワード
Rn+4	D4、後半ワード
Rn	D4、前半ワード

図 C2-2 STMX/FLDMX のメモリフォーマット

C2.6.2 ショートベクタ

単精度レジスタは、8 つまでの単精度値のショートベクタを保持するために使用できます。このようなベクタの要素すべてに対する数値演算は、1 つの単精度演算命令で指定できます。この方法の詳細については、P. C5-2*「アドレシングモード 1 - 単精度ベクタ（非単項）」*と P. C5-14*「アドレシングモード 3 - 単精度ベクタ（単項）」*を参照して下さい。

同様に、倍精度レジスタは、4 つまでの倍精度値のショートベクタを保持するために使用でき、これらのベクタに対して倍精度演算命令で演算を指定できます。詳細については、P. C5-8*「アドレシングモード 2 - 倍精度ベクタ（非単項）」*と P. C5-18*「アドレシングモード 4 - 倍精度ベクタ（単項）」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.6.3 単精度レジスタへの整数の保持

各単精度レジスタは、単精度浮動小数点数値の代わりに 32 ビット整数を保持することもできます。レジスタの内容は、32 ビット整数の場合も、同じワードで表現される単精度数値の場合と同じです。つまり、FMRS、FMSR、単精度ロード / ストア命令を使用して、整数と単精度数値のどちらも転送できます。

32 ビット整数と同じワードで表現される単精度浮動小数点数値は、通常は整数と同じ値を意味しません。たとえば、整数の 2 と –1 は、ワード 0x00000002 と 0xFFFFFFFF で表現されます。単精度浮動小数点数値では、同じワードはそれぞれ非正規化数の 2^{-148} とクワイエット NaN を表します。

したがって、浮動小数点レジスタに保持されている整数を直接単精度数値として使用することや、単精度数値を直接整数として使用することはできません。整数と浮動小数点数値の間の変換が必要な場合、次の 2 つのサブセクションで説明するように、明示的な変換命令を使用する必要があります。

浮動小数点から整数へ

浮動小数点数値から整数への変換には、2 つの命令が使用されます。

1. 最初の命令は、FTOSID、FTOSIS、FTOUID、FTOUIS で、浮動小数点オペランドが倍精度と単精度のどちらであるか、および結果として要求される整数が符号付きと符号なしのどちらであるかによって使い分けられます。この命令の実行後、要求される結果が単精度レジスタに整数として保持されています。

   FTOSIZD、FTOSIZS、FTOUIZD、FTOUIZS 命令の特別な形式により、FPSCR で指定されている丸めモードを変更せず、ゼロへの丸め（RZ）モードを使用した変換が実行できます。これは、C、C++、それに関連する言語で要求される浮動小数点から整数への変換形式です。

2. 2 番目の命令は通常 FMRS 命令で、整数の結果を ARM レジスタに転送しますが、多くの他の命令も使用できます。詳細については、P. C3-11 「*変換命令*」を参照して下さい。

整数から浮動小数点へ

同様に、整数から浮動小数点数値への変換には、2 つの命令が使用されます。

1. 最初の命令は通常 FMSR 命令で、整数のオペランドを単精度レジスタに転送しますが、多くの他の命令も使用できます。詳細については、P. C3-11 「*変換命令*」を参照して下さい。

2. 2 番目の命令は、FSITOD、FSITOS、FUITOD、FUITOS で、整数オペランドを符号付きと符号なしのどちらとして扱うか、および結果として要求されるのが倍精度と単精度どちらの浮動小数点数値であるかによって使い分けられます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.7 システムレジスタ

VFP 実装には、3 つ以上の特別な目的を持つ*システムレジスタ*が含まれています。

- *浮動小数点システム ID* レジスタ（FPSID）は、読み出し専用レジスタで、どの VFP 実装が使用されているかを示します。詳細については、P. C2-22「*FPSID*」を参照して下さい。
- *浮動小数点ステータスと制御レジスタ*（FPSCR）は読み出し / 書き込みレジスタで、浮動小数点システムに関するすべてのユーザレベルのステータスと制御に使用されます。FPSCR の詳細については、P. C2-23「*FPSCR*」を参照して下さい。
- *浮動小数点例外*レジスタ（FPEXC）は読み出し / 書き込みレジスタで、このレジスタにある 2 つのビットがシステムレベルのステータスと制御に使用されます。このレジスタの他のビットは、実装のハードウェアとソフトウェアのコンポーネントの間で例外の情報を通知するために使用可能です。この方法はサブアーキテクチャ定義です。FPEXC の詳細については、P. C2-27「*FPEXC*」を参照して下さい。
- 個別の VFP 実装で、他のシステムレジスタを定義し、実装のハードウェアとソフトウェアのコンポーネント間の通信や、実装定義の VFP 実装の制御目的に使用できます。これらのレジスタはすべて、サブアーキテクチャ定義です。サブアーキテクチャ固有のドキュメントに記載されている方法を除き、これらのレジスタを実装し使用することはできません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.7.1 FPSID

FPSID の形式は次のとおりです。

31-24	23	22	21	20	19-16	15-8	7-4	3-0
実装者コード	SW	0	0	SNG	アーキテクチャ	部品番号	バリエーション	リビジョン

ビット **[31:24]** 実装者コード。次のコードが定義されています。

```
0x41     = A (ARM Ltd)
```

実装者コードの他の値はすべて、ARM 社により予約されています。

ビット **[23]** 実装にハードウェアコプロセッサが含まれている場合は 0、純粋にソフトウェアの実装の場合は 1 です。

ビット **[22:21]** 常に 0（他の値は予約されています)。

ビット **[20]** 実装で単精度と倍精度の両方をサポートしている場合は 0（アーキテクチャの D バリエーション)、単精度のみをサポートしている場合は 1 です (D 以外のバリエーション)。

ビット **[19:16]** アーキテクチャのバージョン番号で、次のようにエンコードされています。

0b0000 VFPv1 を表しています。

0b0001 VFPv2 を表しています。

アーキテクチャバージョンコードの他の値はすべて、ARM 社により予約されています。

ビット **[15:8]** VFP 実装の 1 次部品番号を表す実装定義の値。

ビット **[7:4]** 実装定義のバリエーション番号。この値は一般に、同じ 1 次部品のバリエーションを区別するために使用されます。たとえば、同じ VFP 実装について 2 つのバリエーションがあり、それぞれ異なる ARM プロセッサとともに動作するようにハードウェアのコプロセッサインターフェースが設計されている場合があります。

ビット **[3:0]** 部品の実装定義のリビジョン番号。

FPSID レジスタは読み出し専用で、特権モードと非特権モードの両方でアクセス可能です。FPSID レジスタへの書き込みは無視されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.7.2 FPSCR

FPSCR のフォーマットは次のとおりです。

31	30	29	28	27 26	25	24	23 22	21 20	19	18 17 16	15	14 13	12	11	10	9	8	7	6 5	4	3	2	1	0
N	Z	C	V	DNM	DN	FZ	RMODE	STRIDE	DNM	LEN	IDE	DNM	IXE	UFE	OFE	DZE	IOE	IDC	DNM	IXC	UFC	OFC	DZC	IOC

これらのビットはすべて読み出しと書き込みが可能で、特権モードと非特権モードの両方でアクセス可能です。

——— 注 ———

図で DNM（変更不可）と記載されているビットはすべて、将来拡張用に予約されています。これらのビットは 0 に初期化されます。初期化以外のコードでは、これらのビットが変更されないよう、リード・モディファイ・ライトの手法を使用して FPSCR を扱う必要があります。この規則に従わない場合、コードが将来のシステムで予期しない動作を引き起こす可能性があります。

———————

FPSCR のビットについては、以下のサブセクションで解説します。

条件フラグ

FPSCR のビット [31:28] には、最後に行われた浮動小数点比較の結果が保持されています。

N 比較の結果が *“より小さい”* の場合、1 にセットされます。

Z 比較の結果が *“等しい”* の場合、1 にセットされます。

C 比較の結果が *“等しい”*、*“より大きい”*、*“順序付けなし”* の場合、1 にセットされます。

V 比較の結果が *“順序付けなし”* の場合、1 にセットされます。

これらの条件フラグは、ARM 命令と VFP 命令のどちらでも、条件付き実行に直接影響しません。比較命令の後には、通常は `FMSTAT` 命令が置かれます。この命令は FPSCR 条件フラグを ARM CPSR フラグに転送し、これによってフラグが条件付き実行に影響するようになります。

比較を実行する方法の詳細については、P. C3-6 *「比較命令」* を参照して下さい。

デフォルト NaN モードの制御

ビット [25] はデフォルト NaN モードの制御ビットです。詳細については、P. C2-16 *「デフォルト NaN モード」* を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

Flush-to-Zero モードの制御

FPSCR のビット [24] は FZ ビットで、*Flush-to-Zero* モードを制御します。この処理モードの詳細については、P. C2-14「*Flush-to-Zero モード*」を参照して下さい。

FZ == 0 Flush-to-Zero モードは禁止され、浮動小数点システムの動作は IEEE754 標準に完全に準拠しています。

FZ == 1 Flush-to-Zero モードが許可されています。

丸めモードの制御

FPSCR のビット [23:22] は現在の丸めモードを選択します。この丸めモードは、ほとんどの浮動小数点命令に使用されます。このモードを使用しない浮動小数点命令は `FTOSIZD`、`FTOSIZS`、`FTOUIZD`、`FTOUIZS` のみで、これらの命令は常に RZ モードを使用します。

丸めモードは次のようにエンコードされます。

0b00 *近似値への丸め*（RN）モードを表します。

0b01 *プラス無限大への丸め*（RP）モードを表します。

0b10 *マイナス無限大への丸め*（RM）モードを表します。

0b11 *ゼロへの丸め*（RZ）モードを表します。

丸めモードの詳細については、P. C2-9「*丸め*」を参照して下さい。

ベクタの長さ / ストライドの制御

FPSCR の LEN フィールド（ビット [18:16]）は、ショートベクタに対して演算を行う VFP 命令のベクタ長、つまりベクタオペランドにあるレジスタ数を表します。同様に、STRIDE フィールド（ビット [21:20]）はベクタのストライド、つまりベクタのレジスタがレジスタバンク内でいくつ離れているかを表します。LEN と STRIDE について許容される組み合わせを P. C2-25 表 C2-3 に示します。

LEN と STRIDE の表にない他の組み合わせはすべて、結果は予測不能です。

LEN == 0b000、STRIDE == 0b00 の組み合わせを*スカラモード*と呼ぶこともあります。このモードが有効な場合、すべての演算命令は単純なスカラ演算となります。それ以外の場合、一部の演算命令を除いてデスティネーションが S0 ～ S7（単精度の場合）または D0 ～ D3（倍精度の場合）の範囲内にある場合、スカラ演算となります。どのオペランドがベクタであるかを判定するために使用される完全な規則と、ベクタオペランドを指定する方法の完全な詳細については、第 C5 章「*VFP のアドレシングモード*」と、個別の命令の説明を参照して下さい。

ベクタオペランドの規則では、ベクタ内に同じレジスタを 2 回以上使用できません。P. C2-25 表 C2-3 に記載されている LEN/STRIDE の許容される組み合わせでは、単精度命令についてはこの状態が決して発生しないため、単精度のスカラとベクタ命令ではこれらの LEN/STRIDE の組み合わせをすべて使用できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

倍精度のベクタ命令では、許容される LEN/STRIDE の組み合わせの一部で、ベクタ内に同じレジスタが 2 回出現することがあります。このような LEN/STRIDE の組み合わせが指定されている状態で倍精度ベクタ命令が実行された場合、命令の動作は予測不能です。表 C2-3 の最後の列は、これが適用される LEN/STRIDE の組み合わせを表しています。倍精度スカラ命令は、許容される LEN/STRIDE の組み合わせすべてで正しく動作します。

表 C2-3 ベクタの長さ / ストライドの組み合わせ

LEN	STRIDE	ベクタの長さ	ベクタのストライド	倍精度ベクタ命令
0b000	0b00	1	-	すべての命令はスカラである
0b001	0b00	2	1	正常に動作する
0b001	0b11	2	2	正常に動作する
0b010	0b00	3	1	正常に動作する
0b010	0b11	3	2	予測不能
0b011	0b00	4	1	正常に動作する
0b011	0b11	4	2	予測不能
0b100	0b00	5	1	予測不能
0b101	0b00	6	1	予測不能
0b110	0b00	7	1	予測不能
0b111	0b00	8	1	予測不能

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外のステータスと制御

FPSCR には、各種の例外についてのトラップイネーブルビットと累積例外ビットが含まれています。これらの機能の詳細については、P. C2-10*「浮動小数点例外」*を参照して下さい。

各ビットと例外との関連付けを表 C2-4 に示します。

表 C2-4 例外のステータスと制御ビット

例外タイプ	トラップイネーブルビット	累積例外ビット
無効演算	IOE（ビット [8])	IOC（ビット [0])
0 による除算	DZE（ビット [9])	DZC（ビット [1])
オーバフロー	OFE（ビット [10])	OFC（ビット [2])
アンダーフロー	UFE（ビット [11])	UFC（ビット [3])
不正確	IXE（ビット [12])	IXC（ビット [4])
入力非正規	IDE（ビット [15])	IDC（ビット [7])

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.7.3 FPEXC

FPEXC レジスタのフォーマットは次のとおりです。

31	30	29 ... 0
EX	EN	サブアーキテクチャ定義

このレジスタは、特権モードでのみアクセス可能です。

EX ビット

EX ビット（ビット [31]）はステータスビットで、浮動小数点システムのステートを記録するために保存の必要がある情報の量を表します。このビットはすべての VFP 実装で読み出すことができ、主にプロセススワップコードによって使用されます。

EX == 0 この場合、浮動小数点システムの重要なステートはアーキテクチャで定義されている書き込み可能なレジスタ、つまり汎用レジスタ、FPSCR、FPEXC の内容のみです。プロセスがスワップアウトされたときに EX == 0 の場合、これらのレジスタのみ保存が必要で、そのプロセスが再度スワップインされるときも、これらのレジスタのみ再ロードが必要です。また、レジスタの保存と再ロードの間に、予期しない ARM 例外（ハードウェアで保留の例外を処理するための未定義命令例外など）が発生することはできません。

EX == 1 この場合、浮動小数点システム内にサブアーキテクチャ定義の追加の重要なステートがあり、プロセススワップコードが処理する必要があります。これは一般に、VFP ハードウェアが発生し得る例外を処理するためにサポートコードの補助を必要とし、1 つまたは複数の追加ハードウェアシステムレジスタに発生しうる例外の詳細が含まれている場合に当てはまります。一部の実装では、これをハードウェアが*例外状態*にあると説明しています。EX == 1 の場合にプロセスをスワップアウトするために必要な動作と、そのプロセスを再度スワップインするために必要な動作はサブアーキテクチャ定義です。

FPEXC が書き込まれるときの EX ビットの動作はサブアーキテクチャ定義ですが、EX ビットに 0 を書き込むのは正当な動作で、その直後に読み出された場合は 0 を返す必要があるという制約が適用されます。この制約に従わない場合、EX == 0 の場合について上で説明したプロセススワップ手法は成立しません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

EN ビット

EN ビット（ビット [30]）はグローバルイネーブルビットで、読み出しと書き込みの両方が可能です。

EN == 1 この場合、浮動小数点システムは許可され、通常に動作します。

EN == 0 この場合、浮動小数点システムは禁止されています。この状態では、すべての VFP 命令は、非特権 ARM プロセッサモードで実行された場合に未定義命令として扱われ、以下のものを除くすべての命令は特権ARMプロセッサモードでも未定義命令として扱われます。

- デスティネーションが FPEXC または FPSID レジスタの `FMXR` 命令
- ソースが FPEXC または FPSID レジスタの `FMRX` 命令

——— 注 ———

FPSCR への `FMXR` または FPSCR からの `FMRX` は、EN == 0 の場合に未定義命令として扱われます。VFP 実装に FPSID、FPSCR、FPEXC 以外の追加システムレジスタが含まれている場合、それらのレジスタへの `FMXR` 命令とそれらのレジスタからの `FMRX` 命令の動作はサブアーキテクチャ定義です。

———

他のビット

EX ビットと EN ビットを除く FPSCR のすべてのビットは、読み出し可能 / 書き込み可能かどうかも含めてサブアーキテクチャ定義です。これらは一般にハードウェア実装で、VFP ハードウェアとサポートコードの間で例外の情報を通知するために使用されます。

これらのビットには、EX ビットが 0 の場合、VFP の汎用レジスタ、FPSCR、FPEXC のみを保存し再ロードすることで、浮動小数点システムの重要なステートすべての保存と再ロードが可能な必要があるという制約があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C2.8 リセット時の動作と初期化

ハードウェア VFP 実装がリセットされると、FPEXC の EN ビットは 0 にリセットされます。他の VFP レジスタと、FPEXC の他のビットについては、ハードウェアリセット時の動作はすべて実装定義です。

VFP 実装のソフトウェアコンポーネントが初期化を完了すると、以下の状態になります。

- FPEXC の EN ビットが 1 にセットされる。
- FPEXC の EX ビットが 0 にセットされる。
- FPSCR の他のビットがすべて 0 にセットされる。ただし、一部のケースでは条件コードフラグが例外となることがあります。これによって、次の設定が選択されます。
  - — Flush-to-Zero モードではない、通常の IEEE 754 モード
  - — *近似値への丸め*モード
  - — スカラモード（ベクタ長 1)
  - — すべての例外は非トラップで、累積ステータスビットはその種類の例外がまだ検出されていないことを示す

VFP 汎用レジスタと FPSCR 条件フラグが初期化されるか、初期化される場合どの値に初期化されるかは実装定義です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 C3 章
VFP 命令セットの概要

本章では、VFP 命令セットの概要を解説します。本章は以下のセクションから構成されています。

- *データ処理命令* : P. C3-2
- *ロードとストア命令* : P. C3-14
- *単一レジスタ転送命令* : P. C3-18
- *2 レジスタ転送命令* : P. C3-22

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C3.1 データ処理命令

VFP のデータ処理命令はすべてコプロセッサ 10 または 11 の CDP 命令で、次のフォーマットです。

31 30 29 28	27 24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	1 1 1 0	P	D	q	r	Fn	Fd	cp_num	N	s	M	0	Fm

p、q、r、s

これらのビットは全体として、命令の 1 次オペコードを構成します。これらのオペコードの割り当てを P. C3-3 表 C3-1 に示します。p、q、r、s のすべてが 1 の場合、その命令は 2 オペランドの*拡張命令*で、Fn と N ビットに拡張オペコードが指定されます。

Fd と D

これらのビットは一般に、命令のデスティネーションレジスタを指定します。

- 単精度命令の場合、Fd にはレジスタ番号の上位 4 ビット、D には最下位ビットが保持されます。
- 倍精度命令の場合、Fd にレジスタ番号が保持され、D は常に 0 です。

倍精度命令で D に 1 を指定した場合、命令の動作は予測不能です。

積和命令の場合、このレジスタは累算オペランドレジスタとしても使用されます。比較命令の場合、このレジスタはデスティネーションレジスタではなく、最初のオペランドレジスタになります。

Fn と N

これらのビットは一般に、命令の最初のオペランドレジスタを指定します。

- 単精度命令の場合、Fn にはレジスタ番号の上位 4 ビット、N には最下位ビットが保持されます。
- 倍精度命令の場合、Fn にレジスタ番号が保持され、N は常に 0 です。

ただし、p、q、r、s がすべて 1 の場合、その命令は拡張命令で、Fn と N フィールドはレジスタの指定ではなく、拡張オペコードとなります。これらの拡張オペコードの割り当てを P. C3-4 表 C3-2 に示します。

倍精度の非拡張命令で N に 1 を指定した場合、その命令は未定義です。

Fm と M

これらのビットは、命令の 2 番目のオペランドレジスタを指定します。または一部の拡張命令で、唯一のオペランドレジスタを指定します。

- 単精度命令の場合、Fm にはレジスタ番号の上位 4 ビット、M には最下位ビットが保持されます。
- 倍精度命令の場合、Fm にレジスタ番号が保持され、M は常に 0 です。

倍精度命令で M に 1 を指定した場合、命令の動作は予測不能です。

cp_num

cp_num が 0b1010（コプロセッサ番号 10）の場合、その命令は単精度命令です。cp_num が 0b1011（コプロセッサ番号 11）の場合、その命令は倍精度命令です。

単精度と倍精度の間の変換を行う命令の場合（FCVTDS と FCVTSD)、cp_num はソースの精度に一致します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

VFP データ処理オペコードの割り当てを表 C3-1 と P. C3-4 表 C3-2 に示します。これらの表では、Fd は対応する精度のデスティネーションレジスタを意味します。つまり、単精度命令では Sd、倍精度命令では Dd です。Fn と Fm も同様に使用されます。

表 C3-1 VFP データ処理 1 次オペコード

p	q	r	s	命令名 cp_num = 10	命令名 cp_num = 11	命令の機能
0	0	0	0	FMACS	FMACD	Fd = Fd + (Fn * Fm)
0	0	0	1	FNMACS	FNMACD	Fd = Fd - (Fn * Fm)
0	0	1	0	FMSCS	FMSCD	Fd = -Fd + (Fn * Fm)
0	0	1	1	FNMSCS	FNMSCD	Fd = -Fd - (Fn * Fm)
0	1	0	0	FMULS	FMULD	Fd = Fn * Fm
0	1	0	1	FNMULS	FNMULD	Fd = -(Fn * Fm)
0	1	1	0	FADDS	FADDD	Fd = Fn + Fm
0	1	1	1	FSUBS	FSUBD	Fd = Fn - Fm
1	0	0	0	FDIVS	FDIVD	Fd = Fn / Fm
1	0	0	1	-	-	未定義
1	0	1	0	-	-	未定義
1	0	1	1	-	-	未定義
1	1	0	0	-	-	未定義
1	1	0	1	-	-	未定義
1	1	1	0	-	-	未定義
1	1	1	1	P. C3-4 表 C3-2 参照	P. C3-4 表 C3-2 参照	拡張命令

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 C3-2 VFP データ処理拡張オペコード

拡張命令		命令名		
Fn	**N**	**cp_num = 10**	**cp_num = 11**	**命令の機能**
0000	0	FCPYS	FCPYD	Fd = Fm
0000	1	FABSS	FABSD	Fd = abs(Fm)
0001	0	FNEGS	FNEGD	Fd = -Fm
0001	1	FSQRTS	FSQRTD	Fd = sqrt(Fm)
001x	x	-	-	未定義
0100	0	FCMPS	FCMPD	Fd と Fm を比較します。クワイエット NaN について例外は発生しません。
0100	1	FCMPES	FCMPED	Fd と Fm を比較します。クワイエット NaN について例外が発生します。
0101	0	FCMPZS	FCMPZD	Fd と 0 を比較します。クワイエット NaN について例外は発生しません。
0101	1	FCMPEZS	FCMPEZD	Fd と 0 を比較します。クワイエット NaN について例外が発生します。
0110	x	-	-	未定義
0111	0	-	-	未定義
0111	1	FCVTDS	FCVTSD	単精度⇔倍精度の変換
1000	0	FUITOS	FUITOD	符号なし整数 → 浮動小数点の変換
1000	1	FSITOS	FSITOD	符号付き整数 → 浮動小数点の変換
1001	x	-	-	未定義
101x	x	-	-	未定義
1100	0	FTOUIS	FTOUID	浮動小数点 → 符号なし整数の変換
1100	1	FTOUIZS	FTOUIZD	浮動小数点 → 符号なし整数の変換、RZ モード
1101	0	FTOSIS	FTOSID	浮動小数点 → 符号付き整数の変換
1101	1	FTOSIZS	FTOSIZD	浮動小数点 → 符号付き整数の変換、RZ モード
111x	x	-	-	未定義

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C3.1.1 基本算術命令と平方根

FADDS、FSUBS、FMULS、FDIVS、FSQRTS 命令は、単精度値に対して 4 つの基本算術演算と平方根演算を実行します。同様に、FADDD、FSUBD、FMULD、FDIVD、FSQRTD 命令は、倍精度値に対してこれらの演算を実行します。また、FNMULS と FNMULD 命令はそれぞれ単精度値と倍精度値に対して符号反転乗算を実行します。これらの演算の結果は、FMULS や FMULD 命令の後で FNEGS や FNEGD 命令（結果の符号を反転する）を実行した結果と完全に同じです。

これらの命令はすべて、FPSCR の LEN と STRIDE フィールドを適切に設定することで、ショートベクタに対して演算を実行できます。詳細については、第 C5 章「*VFP のアドレシングモード*」を参照して下さい。

また、これらの命令によって実行される加算、減算、乗算、除算、平方根の演算はすべて、NaN 処理と Flush-to-Zero モードの両方で浮動小数点演算として扱われます。特に、シグナル NaN は無効演算例外を発生させ、Flush-to-Zero モードでは非正規化数オペランドは 0 と見なされ、値はすべて 0 に変換されます。

これらの命令の中で実行される符号反転処理は、浮動小数点演算として扱われません。オペランドが NaN の場合も含めて、必ず符号ビットが反転されます。

C3.1.2 積和命令

FMACS、FMACD、FNMACS、FNMACD、FMSCS、FMSCD、FNMSCS、FNMSCD は*積和命令*です。これらの命令は 2 つの主オペランドの乗算を実行し、指定に応じて積の符号ビットを反転し、デスティネーションレジスタの値を加算または減算してから、結果をデスティネーションレジスタに書き戻します。これらはすべての面において、次の基本算術演算命令と符号反転命令のシーケンスと等価です。

```
FMACS  Sd,Sn,Sm:  FMULS   St,Sn,Sm
                  FADDS   Sd,Sd,St

FMACD  Dd,Dn,Dm:  FMULD   Dt,Dn,Dm
                  FADDD   Dd,Dd,Dt

FNMACS Sd,Sn,Sm:  FMULS   St,Sn,Sm
                  FNEGS   St,St
                  FADDS   Sd,Sd,St

FNMACD Dd,Dn,Dm:  FMULD   Dt,Dn,Dm
                  FNEGD   Dt,Dt
                  FADDD   Dd,Dd,Dt

FMSCS  Sd,Sn,Sm:  FMULS   St,Sn,Sm
                  FNEGS   Sd,Sd
                  FADDS   Sd,Sd,St

FMSCD  Dd,Dn,Dm:  FMULD   Dt,Dn,Dm
                  FNEGD   Dd,Dd
                  FADDD   Dd,Dd,Dt

FNMSCS Sd,Sn,Sm:  FMULS   St,Sn,Sm
                  FNEGS   St,St
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

```
                        FNEGS   Sd,Sd
                        FADDS   Sd,Sd,St

FNMSCD Dd,Dn,Dm:        FMULD   Dt,Dn,Dm
                        FNEGD   Dt,Dt
                        FNEGD   Dd,Dd
                        FADDD   Dd,Dd,Dt
```

ここで、St または Dt は中間結果を保持する仮想レジスタで、Sd または Dd がスカラの場合はスカラ、Sd または Dd がベクタの場合はベクタとして扱われます。

——— 注 ———

これは、どの積和演算でも 2 回の丸めが行われることを意味します。

- 乗算の結果に対する丸め
- 最終的な加算または減算の結果に対する丸め

これらの丸めはどちらも完全に、IEEE754 標準の定義に従って実行されます。特に、これらの命令では他のアーキテクチャの多くで使用される*融合積和*が指定されていません。

これらの命令はすべて、FPSCR の LEN と STRIDE フィールドを適切に設定することで、ショートベクタに対して演算を実行できます。詳細については、第 C5 章「*VFP のアドレシングモード*」を参照して下さい。これらの命令によって実行される乗算と加算はすべて、NaN 処理と Flush-to-Zero モードの両方で浮動小数点演算として扱われます。特に、シグナル NaN は無効演算例外を発生させ、Flush-to-Zero モードでは非正規化数オペランドは 0 と見なされ、値はすべて 0 に変換されます。

これらの命令の中で実行される符号反転処理は、浮動小数点演算として扱われません。オペランドが NaN の場合も含めて、必ず符号ビットが反転されます。

C3.1.3 比較命令

FCMPS、FCMPD、FCMPES、FCMPED 命令は、2 つのレジスタの値を比較します。FCMPZS、FCMPZD、FCMPEZS、FCMPEZD 命令は、レジスタの値を定数 +0 と比較します。

IEEE754 標準では、比較される任意の 2 つの値について、4 つの関係のうち 1 つだけが成り立つことが指定されています。この関係は次のとおりです。

- 2 つの値は、次の条件のいずれかが成り立つときに等しいと見なされます。
  - — 両方の値が数値であり、数値としての値が同じである。これは通常、両方の表現が完全に同じであることを意味しますが、片方が +0、片方が -0 の場合も含まれます。
  - — 両方の値が +∞（プラス無限大）である。
  - — 両方の値が –∞（マイナス無限大）である。
- 最初の値は、次の条件のいずれかが成り立つときに 2 番目の値より小さいと見なされます。
  - — 両方の値が数値であり、最初の数値の値が 2 番目の値よりも小さい。
  - — 最初の値が –∞（マイナス無限大）であり、2 番目の値が数値である。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

— 最初の値が数値であり、2 番目の値が +∞（プラス無限大）である。

— 最初の値が –∞（マイナス無限大）であり、2 番目の値が +∞（プラス無限大）である。

- 最初の値は、次の条件のいずれかが成り立つときに 2 番目の値より大きいと見なされます。

— 両方の値が数値であり、最初の数値の値が 2 番目の値よりも大きい。

— 最初の値が +∞（プラス無限大）であり、2 番目の値が数値である。

— 最初の値が数値であり、2 番目の値が –∞（マイナス無限大）である。

— 最初の値が +∞（プラス無限大）であり、2 番目の値が –∞（マイナス無限大）である。

- 2 つの値は、片方または両方が NaN の場合、*順序付けなし*と見なされます。

——— 注 ———

両方の値が同じ NaN の場合、比較結果は*順序付けなし*であり、*等しい*ではありません。この理由と、+0 と -0 が等しいと見なされることから、ビット単位の正確な比較が必要な場合は VFP 比較命令ではなく ARM® 比較命令を使用する必要があります。

———————

すべての比較命令について、比較結果は表 C3-3 に示すように FPSCR フラグに設定されます。

表 C3-3 VFP 比較フラグの値

比較結果	N	Z	C	V
等しい	0	1	1	0
より小さい	1	0	0	0
より大きい	0	0	1	0
順序付けなし	0	0	1	1

FPSCR フラグの値をベースにして ARM の条件付き実行を行うには、値を ARM CPSR フラグにコピーする必要があります。この目的には、特別な形式の FMRX 命令（FMSTAT と呼ばれます）を使用します。この命令については、P. C3-21「*システムレジスタ転送命令*」を参照して下さい。

比較結果が*順序付けなし*の場合、オペランドが NaN であるために比較によって無効演算例外も生成されている可能性があります。これらの命令には、無効演算例外を生成する2つの異なる形式があります。

- FCMPS、FCMPD、FCMPZS、FCMPZD 命令は、オペランドの片方または両方がシグナル NaN の場合、通常に無効演算例外を生成します。どちらのオペランドもシグナル NaN ではなく、片方または両方がクワイエット NaN の場合、無効演算例外を生成せず、結果は*順序付けなし*になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- FCMPES、FCMPED、FCMPEZS、FCMPEZD 命令は、オペランドの片方または両方が NaN の場合、それがシグナル NaN かクワイエット NaN かにかかわらず、無効演算例外を生成します。これらの命令では、結果が*順序付けなし*の場合は必ず無効演算例外も生成されます。

VFP 比較命令のオペランドは、FPSCR の LEN と STRIDE フィールドの設定にかかわらず、常にスカラと見なされます。

これらの命令によって実行される演算はすべて、NaN 処理と Flush-to-Zero モードの両方で浮動小数点演算として扱われます。特に、オペランドがシグナルNaNの場合は無効演算例外が発生し、Flush-to-Zero モードでは非正規化数オペランドは 0 と見なされます。

IEEE 754 命題のテスト

IEEE754 標準では、浮動小数点比較の結果を算出する方法が 2 つ指定されています。

- *条件コード*結果として、4 つの関係の一つを示します。
  - — 等しい
  - — より小さい
  - — より大きい
  - — 順序付けなし
- 26 の*命題*の 1 つについて真偽の結果を示します。これらの命題はそれぞれ、値についての特定のテストを示します。このうち 6 つは標準の ==、!=、<、<=、>、>= 比較で、C、C++、それらに関連する言語など一般的な言語で使用されます。

VFP アーキテクチャは最初の手法を使用します。ただし、条件コードの結果は、VFP 比較命令と FMSTAT 命令の使用後に、ARM 条件付き実行で可能な限り多くの命題をテストできるように考慮して選択されています。これには、一般に使用される 6 つの命題がすべて含まれます。

IEEE754 標準に従った正しい結果を得るように各命題をテストする方法を表 C3-4 に示します。

表 C3-4 VFP 命題のテスト

一般的なプログラミング言語の条件記号	IEEE 命題	命令タイプ	ARM 条件
==	=	FCMP	EQ
!=	?<>	FCMP	NE
>	>	FCMPE	GT
>=	>=	FCMPE	GE
<	<	FCMPE	MI または CC
<=	<=	FCMPE	LS

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 C3-4 VFP 命題のテスト (continued)

一般的なプログラミング言語の条件記号	IEEE 命題	命令タイプ	ARM 条件
	?	FCMP	VS
	<>	FCMPE	2 つの条件
	<=>	FCMPE	VC
	?>	FCMP	HI
	?>=	FCMP	PL または CS
	?<	FCMP	LT
	?<=	FCMP	LE
	?=	FCMP	2 つの条件
	NOT(>)	FCMPE	LE
	NOT(>=)	FCMPE	LT
	NOT(<)	FCMPE	PL または CS
	NOT(<=)	FCMPE	HI
	NOT(?)	FCMP	VC
	NOT(<>)	FCMPE	2 つの条件
	NOT(<=>)	FCMPE	VS
	NOT(?>)	FCMP	LS
	NOT(?>=)	FCMP	MI または CC
	NOT(?<)	FCMP	GE
	NOT(?<=)	FCMP	GT
	NOT(?=)	FCMP	2 つの条件

どの場合も、主要な 2 つの選択は次のものです。

- FCMP タイプの命令（FCMPS、FCMPD、FCMPZS、FCMPZD のうちいずれか適切なもの）と FCMPE タイプの命令（FCMPES、FCMPED、FCMPEZS、FCMPEZD のうちいずれか適切なもの）を適切に選択することにより無効演算例外を正しく生成させることができます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- どの ARM 条件を使用するか。これは、常に明確ではありません。たとえば、浮動小数点数値の標準的な < 比較では、浮動小数点比較は常に符号付きですが、ARM 条件の MI もしくは LO/CC を使用する必要があります。

  この列に *2 つの条件*が含まれている場合、単一の ARM 条件を使用して命題をテストすることはできません。これらの命題のそれぞれは、いくつかの異なる方法で適切な 2 つの ARM 条件を組み合わせてテストできます。たとえば、命題 <> は NE と VC が両方とも真であること、または GT と MI のいずれかが真であることをチェックしてテストできます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C3.1.4 変換命令

VFP 変換命令のオペランドは、FPSCR の LEN と STRIDE フィールドの設定にかかわらず、常にスカラと見なされます。

単精度と倍精度の間の変換

FCVTDS 命令と FCTVSD 命令は、単精度値と倍精度値の間の変換を実行します。FCVTDS は単精度を倍精度に変換するコプロセッサ 10 命令で、FCVTSD は倍精度を単精度に変換するコプロセッサ 11 命令です。

FCVTDS と FCVTSD による変換はすべて、NaN 処理と Flush-to-Zero モードの両方で浮動小数点演算として扱われます。特に、オペランドがシグナル NaN の場合は無効演算例外が発生し、Flush-to-Zero モードでは非正規化数オペランドは 0 と見なされます。

FCVTDS で発生する可能性のある例外は、シグナル NaN オペランドによる無効演算例外のみです。これは、単精度数値は必ず倍精度で正確に表現されるためです。FCVTSD はこれに加えて、オーバフロー、アンダーフロー、不正確、またはこれらの複数の例外を発生することがあります。

浮動小数点から整数への変換

FTOSIS命令とFTOSID命令は浮動小数点数値を符号付き整数に、FTOUIS命令とFTOUID命令は浮動小数点数値を符号なし整数に、FPSCR で指定されている丸めモードを使用して変換します。

これらの命令のバリエーションは FTOSIZS、FTOSIZD、FTOUIZS、FTOUIZD と呼ばれ、同様の変換をゼロへの丸め（RZ）モードを使用して行います。C やそれに関連する言語では、浮動小数点 → 整数の変換をこのモードで行うため、これらの命令が便利です。他のほとんどの演算では、通常は近似値への丸め（RN）モードが使用されます。これらの命令を使用すると、浮動小数点 → 整数の変換が必要になるたびに FPSCR の丸めモードを変更する必要を省くことができます。

すべての浮動小数点 → 整数変換命令は、整数の結果を単精度レジスタに置きます。よってこれらの結果は、次のいずれかの方法で使用できるようになります。

- FSTS または FSTMS を使用してメモリにストアする。
- FMRS を使用して ARM レジスタに転送する。
- FSITOS、FSITOD、FUITOS、FUITOD のいずれかを使用して、浮動小数点数に変換する。

これらの命令によって実行される演算はすべて、NaN 処理と Flush-to-Zero モードの両方で浮動小数点演算として扱われます。特に、オペランドがシグナルNaNの場合は無効演算例外が発生し、Flush-to-Zero モードでは非正規化数オペランドは 0 と見なされます。

これらの命令の実行中に発生するほとんどの例外的な状況は、無効演算命令として通知されます。これらの命令のデスティネーションは整数なため、結果として通常のクワイエット NaN 値が生成されることはありません。その代わりに、次の一覧に示す値は無効演算命令を生成すると同時に、それぞれの場合について整数のデフォルトの結果が指定されています。

- オペランドが数値で、適切な丸めモードで整数への変換を行うと、デスティネーションの整数で表現可能な最大値よりも大きな整数となる場合、デフォルトの結果はデスティネーションで表現可能な最大の整数になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- オペランドが数値で、適切な丸めモードで整数への変換を行うと、デスティネーションの整数で表現可能な最小値よりも小さな整数となる場合、デフォルトの結果はデスティネーションで表現可能な最小の整数になります。
- オペランドが +∞（プラス無限大）の場合、デフォルトの結果はデスティネーションで表現可能な最大の整数になります。
- オペランドが –∞（マイナス無限大）の場合、デフォルトの結果はデスティネーションで表現可能な最小の整数になります。
- オペランドが NaN（シグナル NaN とクワイエット NaN のいずれか）の場合、デフォルトの結果は 0 になります。

これらの無効演算例外のほか、浮動小数点 → 整数の変換で生成される可能性がある例外は不正確例外のみです。

整数から浮動小数点への変換

`FSITOS`命令と`FSITOD`命令は符号付き整数を浮動小数点数値に、`FUITOS`命令と`FUITOD`命令は符号なし整数を浮動小数点数値に変換します。これらの命令はすべて、単精度レジスタの内容を整数オペランドとして使用します。このオペランドは、つぎのいずれかの方法でレジスタに置くことができます。

- `FLDS` または `FLDMS` を使用してメモリからロードする。
- `FMRS` を使用して ARM レジスタから転送する。
- `FTOSIS`、`FTOSID`、`FTOSIZS`、`FTOSIZD`、`FTOUIS`、`FTOUID`、`FTOUIZS`、`FTOUIZD` のいずれかを使用して、浮動小数点数値を整数に変換する。

整数 0 が浮動小数点に変換された場合、結果は +0 です。`FSITOS` 命令と `FUITOS` 命令では、絶対値が 2^{24} を超える一部の整数オペランドは正しく変換されません。これらのオペランドを変換した場合、FPSCR で指定されている丸めモードに従って丸められ、不正確例外が生成されます。それ以外、整数 → 浮動小数点の変換では一切の例外は生成されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C3.1.5 コピー、符号反転、絶対値命令

FCPYS命令とFCPYD命令は、あるレジスタから別のレジスタに、浮動小数点数値を正確にコピーします。

FNEGS 命令と FNEGD 命令は、FCPYS および FCPYD と同じ動作を実行しますが、コピー時に符号ビットを反転します。これによって数値と無限大が、IEEE 754 標準の付録に記述されているように反転されます。

FABSS 命令と FABSD 命令は、FCPYS および FCPYD と同じ動作を実行しますが、コピー時に符号ビットを 0 に変更します。これによって数値と無限大が、IEEE754 標準の付録に記述されているように絶対値に変換されます。

これらの命令はすべて、FPSCR の LEN と STRIDE のフィールドを適切に設定することで、ショートベクタに対して演算を実行できます。詳細については、第 C5 章「*VFP のアドレシングモード*」を参照して下さい。

IEEE754 標準とその付録では、NaN の処理に関して、これらの演算をすべて非浮動小数点演算として扱うことが認められています。VFP アーキテクチャではこれを行う必要があります。特に、これは次のことを意味します。

- VFP アーキテクチャでは、これらの命令のオペランドがシグナル NaN の場合に無効演算例外を生成*しない*ことが必要です。
- これらの命令の結果は、オペランドが NaN の場合も含め、オペランドをコピーすることで生成されます（必要な符号ビットの調整は行われます）。これは、命令に 1 つ以上の NaN オペランドが存在する場合に結果を生成するための通常の規則よりも優先されます（P. C2-5「*NaN*」参照）。

また、VFP アーキテクチャでは、Flush-to-Zero モードに関して、これらの命令を非浮動小数点演算として扱うことが必要です。Flush-to-Zero モードでは、非正規化オペランドは通常のモードと同じ方法でコピーされ、0 としては扱われません。

―――― 注 ――――

FNEGS または FNEGD を使用した -x の値の計算は、FSUBS または FSUBD を使用して（+0 - x）や（-0 - x）を使用して計算した結果と正確に同じにはなりません。これらの相違点は次のとおりです。

- FSUBS または FSUBD は、x がシグナル NaN の場合に無効演算例外を生成しますが、FNEGS と FNEGD は符号ビットが反転した x を生成し、例外は生成されません。
- FSUBS または FSUBD は、x がクワイエット NaN の場合に x の正確なコピーを生成しますが、FNEGS と FNEGD は符号ビットが反転した x を生成します。
- 0 に対して FNEGS または FNEGD を実行した場合、常に符号が逆の 0 が生成されます。FSUBS または FSUBD を使用して（+0 - x）の値を計算すると、丸めモードが RM の場合はこの計算が実行されますが、丸めモードが RN、RP、RZ の場合は常に結果が +0 になります。（-0 - x）を計算すると、丸めモードが RM の場合は常に結果が -0 になり、丸めモードが RN、RP、RZ の場合は符号が逆の 0 が得られます。
- Flush-to-Zero モードでは、FSUBS または FSUBD を使用した計算では非正規化されたオペランドが 0 として扱われるため、x が非正規化数の場合の結果は 0 になります。FNEGS または FNEGD では Flush-to-Zero モードが無視され、結果は符号が反転された x になります。

――――――――

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C3.2 ロードとストア命令

VFP におけるロードおよびストア命令はすべてコプロセッサ 10 および 11 に対して発行される LDC および STC 命令で、以下の形式で表されます。

31-28	27-25	24	23	22	21	20	19-16	15-12	11-8	7-0
cond	1 1 0	P	U	D	W	L	Rn	Fd	cp_num	offset

P、U、W これらのビットは、*ARM*P. A5-49 *「アドレシングモード 5 - コプロセッサのロード / ストア」*で説明されているように、LDC または STC 命令のアドレシングモードを指定します。また、P. C3-15 表 C3-5 で示されているように、VFP 実装はこれらのビットを使用して、どのロード / ストア操作が必要かを判定します。

Fd と D これらのビットは、ロード命令のデスティネーション浮動小数点レジスタ、またはストア命令のソース浮動小数点レジスタを指定します。

- 単精度命令の場合、Fd にはレジスタ番号の上位 4 ビット、D には最下位ビットが保持されます。
- 倍精度命令の場合、Fd にレジスタ番号が保持され、D は常に 0 です。

倍精度命令で D に 1 を指定した場合、命令の動作は予測不能です。

複数ロード命令と複数ストア命令の場合、これらのフィールドで指定されるレジスタは転送対象のうちで最も番号の小さいレジスタです。以後のレジスタはレジスタ番号順に、offset フィールドで指定された番号まで転送されます。これによって、S31 または D15 より後のレジスタが転送されることになる場合、結果は予測不能です。

L ビット このビットは、命令がロード（L == 1）とストア（L == 0）のどちらかを示します。

Rn *ARM* P. A5-49 *「アドレシングモード 5 - コプロセッサのロード / ストア」*で説明されているように、アドレス計算のベースレジスタとして使用される ARM レジスタを指定します。

cp_num cp_num が 0b1010（コプロセッサ番号 10）の場合、その命令は単精度命令です。cp_num が 0b1011（コプロセッサ番号 11）の場合、その命令は倍精度命令であるか、精度が不明な値を処理するために使用される命令の 1 つです。詳細については、P. C2-18 *「精度が不明な値のストアと再ロード」*を参照して下さい。

offset これらのビットは、ワードオフセットとしてベースレジスタの値に適用され、転送の開始メモリアドレスを生成します。詳細については、*ARM* P. A5-49 *「アドレシングモード 5 - コプロセッサのロード / ストア」*を参照して下さい。

また、P. C3-15 表 C3-5 で示されているように、このオフセットの最下位ビットはどのロード / ストア操作が必要かの判定にも使用されます。さらに、複数ロード命令と複数ストア命令では、転送するレジスタの数の判定にもこのオフセットが使用されます。

P、U、W、L ビットと、cp_num および offset フィールドにより判定される、命令の名前と他の詳細を P. C3-15 表 C3-5 に示します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

表 C3-5 VFP のロード命令とストア命令

PUW	cp_num	offset [0]	命令 L == 0	命令 L == 1	Addr モード	転送対象のレジスタ
0 0 0	x	x	2 つのレジスタの転送	-	-	P. C3-22「*2 レジスタ転送命令*」参照
0 0 1	x	x	未定義	-	-	-
0 1 0	0b1010	x	FSTMS	FLDMS	インデクスなし	(オフセット) 単精度レジスタ
0 1 0	0b1011	0	FSTMD	FLDMD	インデクスなし	(オフセット)/2 倍精度レジスタ
0 1 0	0b1011	1	FSTMX	FLDMX	インデクスなし	(オフセット - 1)/2 倍精度レジスタ
0 1 1	0b1010	x	FSTMS	FLDMS	インクリメント	(オフセット) 単精度レジスタ
0 1 1	0b1011	0	FSTMD	FLDMD	インクリメント	(オフセット)/2 倍精度レジスタ
0 1 1	0b1011	1	FSTMX	FLDMX	インクリメント	(オフセット - 1)/2 倍精度レジスタ
1 0 0	0b1010	x	FSTS	FLDS	負のオフセット	1 つの単精度レジスタ
1 0 0	0b1011	x	FSTD	FLDD	負のオフセット	1 つの倍精度レジスタ
1 0 1	0b1010	x	FSTMS	FLDMS	デクリメント	(オフセット) 単精度レジスタ
1 0 1	0b1011	0	FSTMD	FLDMD	デクリメント	(オフセット)/2 倍精度レジスタ
1 0 1	0b1011	1	FSTMX	FLDMX	デクリメント	(オフセット - 1)/2 倍精度レジスタ
1 1 0	0b1010	x	FSTS	FLDS	正のオフセット	1 つの単精度レジスタ
1 1 0	0b1011	x	FSTD	FLDD	正のオフセット	1 つの倍精度レジスタ
1 1 1	x	x	未定義	-	-	-

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

すべてのロード命令は、ロードされる値をメモリからコピーし、すべてのストア命令はストアされる値をメモリにコピーします。例外は一切発生せず、転送される値は変更されません。ただし実装された内部レジスタの形式に合わせた可逆変換が発生する可能性はあります。NaN の処理と Flush-to-Zero モードの目的から、コピー操作は非浮動小数点演算として扱われます。特に、VFP アーキテクチャでは以下の条件が要求されます。

- シグナル NaN のロードまたはストアでは、無効演算例外は発生せず、シグナル NaN がクワイエット NaN に変更されることもありません。
- Flush-to-Zero モードで非正規化数をロードまたはストアした場合、0 に変更されることはありません。

C3.2.1 1 つの値のロード / ストア

`FLDS` 命令と `FSTS` 命令は、単精度数値と 32 ビット整数のロードとストアに使用できます。`FLDD` 命令と `FSTD` 命令は倍精度数値のロードとストアに使用できます。これらの命令はどれも、対象のタイプのレジスタ 1 つだけを転送します。

これらの命令では、*ARM* P. A5-49 *「アドレシングモード 5 - コプロセッサのロード / ストア」*で説明されているアドレシングモードのうち、イミディエートオフセットモード（P. A5-51 *「コプロセッサのロード / ストア - イミディエートオフセット」*参照）のみが許可されます。このアドレシングモードでは、ベースアドレスの値 Rn に対して、範囲 0 ～ 1020 のイミディエートオフセットに 4 を掛けた値を加算または減算して、アドレスを指定します。ベースレジスタライトバックは使用できません。

C3.2.2 複数の値のロード / ストア

`FLDMS` 命令と `FSTMS` 命令は、複数の単精度数値と整数、または両方のロードとストアに使用できます。`FLDMD` 命令と `FSTMD` 命令は、複数の倍精度数値のロードとストアに使用できます。

これらの命令はそれぞれ、命令の offset フィールドにより指定される複数のレジスタを転送します。offset フィールドはこれらの命令すべてについて、転送される合計ワード数に等しくなります。つまり、`FLDMS` と `FSTMS` ではレジスタ数、`FLDMD` と `FSTMD` ではレジスタ数の 2 倍の値を指定します。

さらに、`FSTMX` 命令は、倍精度レジスタに単精度と倍精度のどちらの数値が含まれているか不明な場合のストアに使用できます。この場合、値は対応する `FLDMX` 命令で正しく再ロードできる形式でストアされます（P. C2-18 *「精度が不明な値のストアと再ロード」*参照）。これらの命令では、offset フィールドには転送される倍精度レジスタ数 + 1 を指定します。これは、これらの命令で転送可能な最大のワード数です。一部の実装では、この最大数よりも 1 つ少ないワードを転送し、メモリワード 1 つを使用されずに残します。

`FSTMX` 命令と `FLDMX` 命令は、`FSTMD` および `FLDMD` と同様に、コプロセッサ 11 命令としてエンコードされます。これらは、offset フィールドによって区別されます。`FSTMX` 命令と `FLDMX` 命令では offset フィールドが奇数、`FSTMD` 命令と `FLDMD` 命令では偶数です。

`FSTMX` 命令と `FLDMX` 命令は、VFP アーキテクチャで唯一、単精度のみのバリエーション（D バリエーション以外）に存在するコプロセッサ 11 命令です。ソフトウェアを移植しやすくするため、これらのバリエーション用に作成するプログラムでは、たとえ非 D バリエーションでレジスタに単精度値が保

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

持されていることが明白でも、D バリエーション用に作成されるプログラムと同じ状況でこれらの命令を使用する必要があります。この影響を受ける主な状況は、呼び出し先保存レジスタのストアと再ロード、およびプロセススワップコードです。

これらの命令では、3 つのアドレシングモードが利用可能です。

- インデクスなしモードは、LDC/STC のインデクスなしアドレシングモードと同じです。詳細については、P. A5-54 *「コプロセッサのロード / ストア - インデクスなし」*を参照して下さい。ベースレジスタ Rn により、転送の開始アドレスが決定され、Rn は変更されません。

  offset フィールドは転送するレジスタの数を決定しますが、アドレス計算には影響しません。

- インクリメントモードは、LDC/STC のイミディエートポストインデクスアドレシングモードで、オフセットが正の場合と同じです。詳細については、P. A5-53 *「コプロセッサのロード / ストア - イミディエートポストインデクス」*を参照して下さい。ベースレジスタ Rn により、転送の開始アドレスが決定されます。offset フィールドは、転送するレジスタの数を決定し、この値に 4 を掛けて Rn の値を足し、Rn に書き戻されます。

  このため、転送後の Rn は転送された最後のワード（または、FSTMX と FLDMX の場合、転送が*可能*であった最後のワード）の直後のメモリワードを指しています。つまり、これらの命令は値を空上昇スタックにプッシュしたり、フル下降スタックからポップしたりするのに適しています。

- デクリメントモードは、LDC/STC のイミディエートポストインデクスアドレシングモードで、オフセットが負の場合と同じです。詳細については、P. A5-52 *「コプロセッサのロード / ストア - イミディエートプリインデクス」*を参照して下さい。オフセットに 4 を掛け、ベースレジスタ Rn の値を足して、転送の開始アドレスが決定され、この開始アドレスが Rn に書き戻されます。offset フィールドは、転送するレジスタの数の決定にも使用されます。

  したがって、転送前の Rn は転送される最後のワード（または、FSTMX と FLDMX の場合、転送が*可能*であった最後のワード）の直後のメモリワードを指しています。このため、これらの命令は値をフル下降スタックにプッシュしたり、空上昇スタックからポップしたりするのに適しています。

——— 注 ———

このように、ロード命令とストア命令にはショートベクタ形式は存在しませんが、FLDMS、FLDMD、FSTMS、FSTMD 命令を使用してショートベクタの多くをロード / ストアすることができます。ただし、ショートベクタは第 C5 章 *「VFP のアドレシングモード」* で説明されているようにバンク内でラップアラウンドするのに対して、複数ロードおよび複数ストア命令は単に S0 ～ S31 または D0 ～ D15 内で直線的に進むだけであることに注意して下さい。ラップアラウンドするショートベクタのロードやストアを行う場合、2 つ以上の命令を使用する必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C3.3 単一レジスタ転送命令

VFP における単一レジスタ転送命令はすべてコプロセッサ 10 および 11 に対して発行される MCR および MRC 命令で、以下の形式で表されます。

31 30 29 28	27 24	23 21	20	19 16	15 12	11 8	7	6 5 4	3 0
cond	1 1 1 0	opcode	L	Fn	Rd	cp_num	N	0 0 1	SBZ

opcode P. C3-19 表 C3-6 に示すように、要求されるレジスタ転送操作を決定します。

L ビット 転送の方向を決定します。

L == 0 ARM レジスタから VFP レジスタに転送を行います（MCR 命令)。

L == 1 VFP レジスタから ARM レジスタに転送を行います（MRC 命令)。

Fn と N ビット これらのビットは、転送に関与する VFP レジスタを指定します。

- 単精度レジスタの場合、Fn にはレジスタ番号の上位 4 ビット、N には最下位ビットが保持されます。
- 倍精度レジスタの場合、Fn にレジスタ番号が保持され、N は常に 0 です。
- システムレジスタの場合、Fn と N により P. C3-19 表 C3-7 に示すようにレジスタが指定されます。

倍精度レジスタについて N に 1 を指定した場合、命令の動作は予測不能です。

Rd 転送に関与する ARM レジスタを指定します。Rd が R15 の場合、命令の動作は一般的な ARM 命令について指定されたものと同じです。

- MCR 命令の場合（L == 0)、命令の動作は予測不能です。
- MRC 命令の場合（L == 1)、転送される値の上位 4 ビットが ARM 条件コードフラグに置かれ、残りの 28 ビットは破棄されます。FMSTAT 命令はこの動作を使用する唯一の VFP 命令で、比較結果を ARM に転送するために使用されます。他の MRC 命令はすべて、Rd が R15 の場合の動作は予測不能です。

cp_num cp_num が 0b1010（コプロセッサ番号 10）の場合、その命令は単精度またはシステムレジスタ転送です。

cp_num が 0b1011（コプロセッサ番号 11）の場合、その命令は倍精度レジスタ転送です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

レジスタ転送オペコードの割り当てと、命令に関する他の詳細を表 C3-6 に示します。

表 C3-6 VFP 単一レジスタ転送命令

opcode	cp_num	L	命令名	命令の機能
000	0b1010	0	FMSR	Sn = Rd
000	0b1010	1	FMRS	Rd = Sn
000	0b1011	0	FMDLR	Dn[31:0] = Rd
000	0b1011	1	FMRDL	Rd = Dn[31:0]
001	0b1010	x	-	未定義
001	0b1011	0	FMDHR	Dn[63:32] = Rd
001	0b1011	1	FMRDH	Rd = Dn[63:32]
01x	0b101x	x	-	未定義
10x	0b101x	x	-	未定義
110	0b101x	x	-	未定義
111	0b1010	0	FMXR	SystemReg(Fn,N) = Rd
111	0b1010	1	FMRX	Rd = SystemReg(Fn,N)
111	0b1011	x	-	未定義

表 C3-7 は、FMXR 命令と FMRX 命令におけるシステムレジスタの指定です。

表 C3-7 VFP システムレジスタのエンコード

Fn	N	システムレジスタ
0b0000	0	FPSID
0b0001	0	FPSCR
0b1000	0	FPEXC

この表に示されていないエンコードは次のとおりです。

- Fn の最上位ビットが 0 の場合、将来の拡張用に予約されています。これらを使用した FMXR 命令と FMRX 命令は未定義です。
- Fn の最上位ビットが 1 の場合、追加のサブアーキテクチャ定義システムレジスタ用に予約されています。これらを使用した FMXR 命令と FMRX 命令はサブアーキテクチャ定義です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C3.3.1 汎用単一レジスタ転送命令

FMRS命令は単精度レジスタにある単精度数値または32ビット整数をARMレジスタに転送するために使用され、FMSR 命令は ARM レジスタから単精度レジスタに同様の転送を行います。

FMRDH 命令と FMRDL 命令は、倍精度レジスタにある倍精度数値を 1 組の ARM レジスタに転送します。FMRDH 命令は、倍精度数値の最上位ワードを転送します。このワードには、符号、指数、仮数の上位 20 ビットが含まれています。FMRDL 命令は、残りの仮数ビットを含む最下位ワードを転送します。

同様に、FMDHR 命令と FMDLR 命令は、1 組の ARM レジスタにある倍精度数値を倍精度レジスタに転送します。FMDHR は最上位ワードを、FMDLR は最下位ワードを転送します。

——— 注 ———

FMDHR 命令と FMDLR 命令は組で使用し、同じ倍精度レジスタに書き込む必要があります。これらの命令は続けて実行する必要はありませんが、片方だけが実行されてもう片方が実行されていない場合、デスティネーションの倍精度レジスタの有効な使用法は次のもののみです。

- 組の 2 番目の命令のデスティネーションレジスタとして。
- FSTMX 命令でストアし、FLDMX で再ロードし、ストアと再ロードの間に別の目的に使用する。

これらの命令の浮動小数点オペランドはすべて、FPSCR の LEN と STRIDE のフィールドの設定にかかわらず、常にスカラと見なされます。

実行されるレジスタ転送は常に、単純なコピーです。例外は一切発生せず、転送される値は変更されません。ただし実装された内部レジスタの形式に合わせた可逆変換が発生する可能性はあります。

NaN の処理と Flush-to-Zero モードの目的から、コピー操作は非浮動小数点演算として扱われます。特に、VFP アーキテクチャでは以下の条件が要求されます。

- シグナル NaN のレジスタ転送では、無効演算例外は発生せず、シグナル NaN がクワイエット NaN に変更されることもありません。
- Flush-to-Zero モードで非正規化数をレジスタ転送した場合、0 に変更されることはありません。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C3.3.2 システムレジスタ転送命令

FMRX 命令は、システムレジスタの値を ARM レジスタに転送します。FMXR 命令は、ARM レジスタの値をシステムレジスタに転送します。正確な影響は、関与するシステムレジスタの定義によって異なります。アーキテクチャ定義のシステムレジスタの詳細については P. C2-21「*システムレジスタ*」を、サブアーキテクチャ定義のシステムレジスタについては各サブアーキテクチャのドキュメントを参照して下さい。

これらの命令は*直列化*命令です。

FMXR 命令または FMRX 命令が FPEXC/FPSID へのアクセスのため実行されるときは、レジスタ転送は次の条件が整うまで遅延します。

- 進行中のすべての浮動小数点演算で、例外が生成されるかどうかが決定した。
- 進行中の浮動小数点演算について、それらの浮動小数点演算のソフトウェア処理を有効にするために必要な、サブアーキテクチャ定義レジスタの内容への影響がすべて発生した。
- 進行中の浮動小数点演算がすべて、システムレジスタの内容の変更による影響を受けなくなった（たとえば、丸めモードや Flush-to-Zero モードの変更)。

FMXR 命令または FMRX 命令が FPSCR へのアクセスのため実行されるときは、レジスタ転送は次の条件が整うまで遅延します。

- 進行中のすべての浮動小数点演算で、例外が生成されるかどうかが決定した。
- 進行中の浮動小数点演算について、トラップ例外処理または他のソフトウェア処理がすべて完了した。
- 進行中の浮動小数点演算について、システムレジスタの内容への影響（非トラップ例外について累積例外フラグをセットするなど）がすべて発生した。
- 進行中の浮動小数点演算がすべて、システムレジスタの内容の変更による影響を受けなくなった（たとえば、丸めモードや Flush-to-Zero モードの変更)。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C3.4 2 レジスタ転送命令

VFP における 2 レジスタ転送命令はすべてコプロセッサ 10 および 11 に対して発行される MCRR および MRRC 命令で、以下の形式で表されます。

31 30 29 28	27 24	23 21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	1 1 0 0	0 1 0	L	Rn	Rd	cp_num	0	0	M	1	Fm

L ビット 転送の方向を決定します。

L == 0 2 つの ARM レジスタから VFP レジスタに転送を行います（MCRR 命令)。

L == 1 VFP レジスタから 2 つの ARM レジスタに転送を行います（MRRC 命令)。

Fm と M ビット

これらのビットは、転送に関与する VFP レジスタまたはレジスタの組を指定します。

- 単精度レジスタの組の場合、Fm にはレジスタ番号の上位 4 ビット、M には最下位ビットが保持されます。
- 倍精度レジスタの場合、Fm にレジスタ番号が保持され、M は常に 0 です。

倍精度レジスタを転送する命令について M に 1 を指定した場合、命令の動作は予測不能です。

Rn 倍精度レジスタの上位ワードもしくはFmで指定された単精度レジスタの転送先もしくは転送元になる ARM レジスタを指定します。

Rn が R15 の場合、動作は予測不能です。

Rd 倍精度レジスタの下位ワードもしくはFmに1を加えた番号で指定された単精度レジスタの転送先もしくは転送元になる ARM レジスタを指定します。

Rn が R15 の場合、動作は予測不能です。

cp_num cp_num が 0b1010（コプロセッサ番号 10）の場合、その命令は 2 つの単精度レジスタの転送です。

cp_num が 0b1011（コプロセッサ番号 11）の場合、その命令は倍精度レジスタ転送です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

命令の詳細を表 C3-8 に示します。

表 C3-8 VFP の 2 つのレジスタの転送命令

cp_num	L	命令名	命令の機能
0b1010	0	FMSRR	Fm = Rn, (Fm + 1) = Rd
0b1010	1	FMRRS	Rn = Fm, Rd = (Fm + 1)
0b1011	0	FMDRR	Fm[31:0] = Rd, Fm[63:32] = Rn
0b1011	1	FMRRD	Rd = Fm[31:0], Rn = Fm[63:32]

FMRRS命令は、2つの連続した番号の単精度レジスタにある2つの単精度数値または32ビット整数を、2つの ARM レジスタに転送します。FMSRR 命令は、2 つの ARM レジスタから、2 つの連続した番号の単精度レジスタに、同様の転送を行います。ARM レジスタは連続している必要はありません。

FMRRD 命令は、倍精度レジスタにある倍精度数値を 2 つの ARM レジスタに転送します。ARM レジスタは連続している必要はありません。

同様に、FMDRR 命令は、2 つの ARM レジスタにある倍精度値を VFP の倍精度レジスタに転送します。ARM レジスタは連続している必要はありません。

これらの命令の浮動小数点オペランドはすべて、FPSCR の LEN と STRIDE のフィールドの設定にかかわらず、常にスカラと見なされます。

実行されるレジスタ転送は常に、単純なコピーです。例外は一切発生せず、転送される値は変更されません。ただし実装された内部レジスタの形式に合わせた可逆変換が発生する可能性はあります。

NaN の処理と Flush-to-Zero モードの目的から、コピー操作は非浮動小数点演算として扱われます。特に、VFP アーキテクチャでは以下の条件が要求されます。

- シグナル NaN のレジスタ転送では、無効演算例外は発生せず、シグナル NaN がクワイエット NaN に変更されることもありません。
- Flush-to-Zero モードで非正規化数をレジスタ転送した場合、0 に変更されることはありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 C4 章
VFP 命令

本章では、VFP の各命令の構文と用法を解説します。本章は以下のセクションから構成されています。

• *VFP 命令のアルファベット順一覧* : P. C4-2

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1 VFP 命令のアルファベット順一覧

各 VFP 命令について詳しく説明します。

C4.1.1 FABSD

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	0	1	1	0	0	0	0	Dd	1	0	1	1	1	1	0	0	Dm

FABSD（浮動小数点絶対値変換、倍精度）命令は、倍精度レジスタの値を絶対値に変換して他の倍精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FABSD{<cond>}  <Dd>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Dm> ソースレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = abs(Dm[i])
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

絶対値関数 関数 `abs(x)` は、IEEE754-1985 標準の付録で定義されているように、`x` の符号ビットを強制的に 0 としてコピーする操作を意味します。

Flush-to-Zero モード

FPSCR の FZ ビットは、この命令のオペランドや結果に影響しません。

ベクタ FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、`FABSD` は 1 つの絶対値演算を実行し、`vec_len = 1`、`Dd[0] = Dd`、`Dm[0] = Dm` です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、`FABSD` は複数の絶対値演算を実行する可能性があります。`FABSD` で使用されるレジスタのエンコードと、`vec_len`、`Dd[i]`、`Dm[i]` の決定方法については、P. C5-18 *「アドレシングモード 4 - 倍精度ベクタ（単項）」*を参照して下さい。

シグナル NaN

VFP アーキテクチャに準拠するため、`FABSD` はソースレジスタの値がシグナル NaN の場合も例外を生成できません。これは、IEEE754-1985 標準の付録に定義されているよりも厳格な条件です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.2 FABSS

31 28	27 26 25 24 23	22	21 20	19 18 17 16	15 12	11 10 9 8	7	6	5	4	3 0
cond	1 1 1 0 1	D	1 1	0 0 0 0	Fd	1 0 1 0	1	1	M	0	Fm

FABSS（浮動小数点絶対値変換、単精度）命令は、単精度レジスタの値を絶対値に変換し他の単精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FABSS{<cond>}  <Sd>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sm> ソースレジスタ。番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = abs(Sm[i])
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

絶対値関数 関数 `abs(x)` は、IEEE754-1985 標準の付録で定義されているように、`x` の符号ビットを強制的に 0 としてコピーする操作を意味します。

Flush-to-Zero モード

FPSCR の FZ ビットは、この命令のオペランドや結果に影響しません。

ベクタ FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、`FABSS` は 1 つの絶対値演算を実行し、`vec_len = 1`、`Sd[0] = Sd`、`Sm[0] = Sm` です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、`FABSS` は複数の絶対値演算を実行する可能性があります。`FABSS` で使用されるレジスタのエンコードと、`vec_len`、`Sd[i]`、`Sm[i]` の決定方法については、P. C5-14 *「アドレシングモード 3 - 単精度ベクタ（単項）」*を参照して下さい。

シグナル **NaN**

VFP アーキテクチャに準拠するため、`FABSS` はソースレジスタの値がシグナル NaN の場合も例外を生成できません。これは、IEEE754-1985 標準の付録に定義されているよりも厳格な条件です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.3 FADDD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	0	1	1	Dn	Dd	1	0	1	1	0	0	0	0	Dm

FADDD（浮動小数点加算、倍精度）命令は、2 つの倍精度レジスタの値を加算し、結果を 3 番目の倍精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FADDD{<cond>}  <Dd>, <Dn>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Dn> 加算の最初のオペランドレジスタ。

<Dm> 加算の 2 番目のオペランドレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dn[i] + Dm[i]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、FADDD は 1 つの加算を実行し、vec_len = 1、Dd[0] = Dd、Dn[0] = Dn、Dm[0] = Dm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FADDD は複数の加算を実行する可能性があります。FADDD で使用されるレジスタのエンコードと、vec_len、Dd[i]、Dn[i]、Dm[i] の決定方法については、P. C5-8*「アドレシングモード 2 - 倍精度ベクタ（非単項）」*を参照して下さい。

丸め　この演算は完全な丸めが行われる加算です。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.4 FADDS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	D	1	1	Fn	Fd	1	0	1	0	N	0	M	0	Fm

FADDS（浮動小数点加算、単精度）命令は、2 つの単精度レジスタの値を加算し、結果を 3 番目の単精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FADDS{<cond>}  <Sd>, <Sn>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sn> 加算の最初のオペランドレジスタ。番号は、Fn（上位 4 ビット）と N（最下位ビット）としてエンコードされます。

<Sm> 加算の 2 番目のオペランドレジスタ。番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sn[i] + Sm[i]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、FADDS は 1 つの加算を実行し、vec_len = 1、Sd[0] = Sd、Sn[0] = Sn、Sm[0] = Sm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FADDS は複数の加算を実行する可能性があります。FADDS で使用されるレジスタのエンコードと、vec_len、Sd[i]、Sn[i]、Sm[i] の決定方法を、P. C5-2*「アドレシングモード 1 - 単精度ベクタ（非単項）」*に示します。

丸め　この演算は完全な丸めが行われる加算です。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.5 FCMPD

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	0	1	1	0	1	0	0	Dd	1	0	1	1	0	1	0	0	Dm

FCMPD（浮動小数点比較、倍精度）命令は、2 つの倍精度レジスタの内容を比較し、結果を FPSCR フラグに書き込みます。このフラグは通常、次に FMSTAT 命令を使用して ARM® フラグに転送されます。

構文

```
FCMPD{<cond>}  <Dd>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> 比較の最初のオペランドレジスタ。

<Dm> 比較の 2 番目のオペランドレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、入力非正規

動作

```
if ConditionPassed(cond) then
    if (Dd is a signaling NaN) or (Dm is a signaling NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Dd < Dm) then 1 else 0
    FPSCR Z flag = if (Dd == Dm) then 1 else 0
    FPSCR C flag = if (Dd < Dm) then 0 else 1
    FPSCR V flag = if (Dd and Dm compare as unordered) then 1 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　　FCMPD は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

NaN　　IEEE754 標準では、比較結果が <、==、>、*順序付けなし*の 1 つだけであることが指定されています。Dd と Dm の片方または両方が NaN の場合、これらは順序付けなしで、（Dd < Dm）、（Dd == Dm）、（Dd > Dm）のいずれも偽になります。この結果として、FPSCR フラグは N = 0、Z = 0、C = 1、V = 1 に設定されます。

FCMPD で無効演算例外が発生するのは、オペランドにシグナル NaN が含まれている場合のみです。よって比較結果においてシグナル NaN 以外の要因による例外を発生させたくない命題のテストに有効です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.6 FCMPED

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	0	1	1	0	1	0	0	Dd	1	0	1	1	1	1	0	0	Dm

FCMPED（浮動小数点比較（NaN 例外）、倍精度）命令は、2 つの倍精度レジスタの内容を比較し、結果を FPSCR フラグに書き込みます。このフラグは通常、次に FMSTAT 命令を使用して ARM フラグに転送されます。

構文

```
FCMPED{<cond>}  <Dd>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> 比較の最初のオペランドレジスタ。

<Dm> 比較の 2 番目のオペランドレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、入力非正規

動作

```
if ConditionPassed(cond) then
    if (Dd is a NaN) or (Dm is a NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Dd < Dm) then 1 else 0
    FPSCR Z flag = if (Dd == Dm) then 1 else 0
    FPSCR C flag = if (Dd < Dm) then 0 else 1
    FPSCR V flag = if (Dd and Dm compare as unordered) then 1 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FCMPED は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

NaN　IEEE754 標準では、比較結果が <、==、>、*順序付けなし*の 1 つだけであることが指定されています。Dd と Dm の片方または両方が NaN の場合、これらは順序付けなしで、（Dd < Dm）、（Dd == Dm）、（Dd > Dm）のいずれも偽になります。この結果として、FPSCR フラグは N = 0、Z = 0、C = 1、V = 1 に設定されます。

FCMPED で無効演算例外が発生するのは、オペランドにシグナル NaN が含まれている場合のみです。よって比較結果においてシグナル NaN 以外の要因による例外を発生させたくない命題のテストに有効です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.7 FCMPES

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	D	1	1	0	1	0	0	Fd	1	0	1	0	1	1	M	0	Fm

FCMPES（浮動小数点比較（NaN 例外）、単精度）命令は、2 つの単精度レジスタの内容を比較し、結果を FPSCR フラグに書き込みます。このフラグは通常、次に FMSTAT 命令を使用して ARM フラグに転送されます。

構文

```
FCMPES{<cond>}  <Sd>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> 比較の最初のオペランドレジスタ。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sm> 比較の 2 番目のオペランドレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、入力非正規

動作

```
if ConditionPassed(cond) then
    if (Sd is a NaN) or (Sm is a NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Sd < Sm) then 1 else 0
    FPSCR Z flag = if (Sd == Sm) then 1 else 0
    FPSCR C flag = if (Sd < Sm) then 0 else 1
    FPSCR V flag = if (Sd and Sm compare as unordered) then 1 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　　`FCMPES` は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

NaN　　IEEE754 標準では、比較結果が `<`、`==`、`>`、*順序付けなし*の 1 つだけであることが指定されています。Dd と Dm の片方または両方が NaN の場合、これらは順序付けなしで、（Dd < Dm）、（Dd == Dm）、（Dd > Dm）のいずれも偽になります。この結果として、FPSCR フラグは N = 0、Z = 0、C = 1、V = 1 に設定されます。

`FCMPES` で無効演算例外が発生するのは、オペランドにシグナル NaN が含まれている場合のみです。よって比較結果においてシグナル NaN 以外の要因による例外を発生させたくない命題のテストに有効です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.8 FCMPEZD

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	0	1	1	0	1	0	1	Dd	1	0	1	1	1	1	0	0	SBZ

FCMPEZD（浮動小数点 0 との比較（NaN 例外）、倍精度）命令は、倍精度レジスタの内容を 0 と比較し、結果を FPSCR フラグに書き込みます。このフラグは通常、次に FMSTAT 命令を使用して ARM フラグに転送されます。

構文

```
FCMPEZD{<cond>}  <Dd>
```

各項目の説明については以下を参照して下さい。

<cond>　この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd>　比較の最初のオペランドレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、入力非正規

動作

```
if ConditionPassed(cond) then
    if (Dd is a NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Dd < 0.0) then 1 else 0
    FPSCR Z flag = if (Dd == 0.0) then 1 else 0
    FPSCR C flag = if (Dd < 0.0) then 0 else 1
    FPSCR V flag = if (Dd is a NaN) then 1 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　　`FCMPEZD` は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

NaN　　IEEE754 標準では、比較結果が `<`、`==`、`>`、*順序付けなし*の 1 つだけであることが指定されています。Dd が NaN の場合、0 との比較は順序付けなしで、（Dd < 0.0）、（Dd == 0.0）、（Dd > 0.0）のいずれも偽になります。この結果として、FPSCR フラグは N = 0、Z = 0、C = 1、V = 1 に設定されます。

`FCMPEZD` で無効演算例外が発生するのは、オペランドにいずれかのタイプの NaN が含まれている場合のみです。よって比較結果において NaN 以外の原因による例外を発生させたくない命題のテストに有効です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.9 FCMPEZS

31 28	27 26 25 24 23	22	21 20	19 18 17 16	15 12	11 10 9 8	7	6	5	4	3 2 1 0
cond	1 1 1 0 1	D	1 1	0 1 0 1	Fd	1 0 1 0	1	1	0	0	SBZ

FCMPEZS（浮動小数点 0 との比較（NaN 例外）、単精度）命令は、単精度レジスタの内容を 0 と比較し、結果を FPSCR フラグに書き込みます。このフラグは通常、次に FMSTAT 命令を使用して ARM フラグに転送されます。

構文

```
FCMPEZS{<cond>}  <Sd>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> 比較の最初のオペランドレジスタ。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、入力非正規

動作

```
if ConditionPassed(cond) then
    if (Sd is a NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Sd < 0.0) then 1 else 0
    FPSCR Z flag = if (Sd == 0.0) then 1 else 0
    FPSCR C flag = if (Sd < 0.0) then 0 else 1
    FPSCR V flag = if (Sd is a NaN) then 1 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　　FCMPEZS は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

NaN　　IEEE754 標準では、比較結果が `<`、`==`、`>`、*順序付けなし*の 1 つだけであることが指定されています。Dd が NaN の場合、0 との比較は順序付けなしで、（Dd < 0.0）、（Dd == 0.0）、（Dd > 0.0）のいずれも偽になります。この結果として、FPSCR フラグは N = 0、Z = 0、C = 1、V = 1 に設定されます。

FCMPEZS で無効演算例外が発生するのは、オペランドにいずれかのタイプの NaN が含まれている場合のみです。よって比較結果において NaN 以外の原因による例外を発生させたくない命題のテストに有効です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.10 FCMPS

31 28	27 26 25 24 23	22	21 20	19 18 17 16	15 12	11 10 9 8	7	6	5	4	3 0
cond	1 1 1 0 1	D	1 1	0 1 0 0	Fd	1 0 1 0	0	1	M	0	Fm

FCMPS（浮動小数点比較、単精度）命令は、2 つの単精度レジスタの内容を比較し、結果を FPSCR フラグに書き込みます。このフラグは通常、次に FMSTAT 命令を使用して ARM フラグに転送されます。

構文

```
FCMPS{<cond>}  <Sd>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> 比較の最初のオペランドレジスタ。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sm> 比較の 2 番目のオペランドレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、入力非正規

動作

```
if ConditionPassed(cond) then
    if (Sd is a signaling NaN) or (Sm is a signaling NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Sd < Sm) then 1 else 0
    FPSCR Z flag = if (Sd == Sm) then 1 else 0
    FPSCR C flag = if (Sd < Sm) then 0 else 1
    FPSCR V flag = if (Sd and Sm compare as unordered) then 1 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　　FCMPS は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

NaN　　IEEE754 標準では、比較結果が <、==、>、*順序付けなし*の 1 つだけであることが指定されています。Dd と Dm の片方または両方が NaN の場合、これらは順序付けなしで、（Dd < Dm）、（Dd == Dm）、（Dd > Dm）のいずれも偽になります。この結果として、FPSCR フラグは N = 0、Z = 0、C = 1、V = 1 に設定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.11 FCMPZD

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	0	1	1	0	1	0	1	Dd	1	0	1	1	0	1	0	0	SBZ

FCMPZD（浮動小数点 0 との比較、倍精度）命令は、倍精度レジスタの内容を 0 と比較し、結果を FPSCR フラグに書き込みます。このフラグは通常、次に FMSTAT 命令を使用して ARM フラグに転送されます。

構文

```
FCMPZD{<cond>}  <Dd>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> 比較の最初のオペランドレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、入力非正規

動作

```
if ConditionPassed(cond) then
    if (Dd is a signaling NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Dd < 0.0) then 1 else 0
    FPSCR Z flag = if (Dd == 0.0) then 1 else 0
    FPSCR C flag = if (Dd < 0.0) then 0 else 1
    FPSCR V flag = if (Dd is a NaN) then 1 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　　`FCMPZD` は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

NaN　　IEEE754 標準では、比較結果が `<`、`==`、`>`、*順序付けなし*の 1 つだけであることが指定されています。Dd が NaN の場合、0 との比較は順序付けなしで、（Dd < 0.0）、（Dd == 0.0）、（Dd > 0.0）のいずれも偽になります。この結果として、FPSCR フラグは N = 0、Z = 0、C = 1、V = 1 に設定されます。

`FCMPZD` で無効演算例外が発生するのは、オペランドにいずれかのタイプの NaN が含まれている場合のみです。よって比較結果において NaN 以外の原因による例外を発生させたくない命題のテストに有効です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.12 FCMPZS

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	D	1	1	0	1	0	1	Fd	1	0	1	0	0	1	0	0	SBZ

FCMPZS（浮動小数点 0 との比較、単精度）命令は、単精度レジスタの内容を 0 と比較し、結果を FPSCR フラグに書き込みます。このフラグは通常、次に FMSTAT 命令を使用して ARM フラグに転送されます。

構文

```
FCMPZS{<cond>}  <Sd>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> 比較の最初のオペランドレジスタ。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、入力非正規

動作

```
if ConditionPassed(cond) then
    if (Sd is a signaling NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Sd < 0.0) then 1 else 0
    FPSCR Z flag = if (Sd == 0.0) then 1 else 0
    FPSCR C flag = if (Sd < 0.0) then 0 else 1
    FPSCR V flag = if (Sd is a NaN) then 1 else 0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　`FCMPZS` は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

NaN　IEEE754 標準では、比較結果が `<`、`==`、`>`、*順序付けなし*の 1 つだけであることが指定されています。Dd が NaN の場合、0 との比較は順序付けなしで、（Dd < 0.0）、（Dd == 0.0）、（Dd > 0.0）のいずれも偽になります。この結果として、FPSCR フラグは N = 0、Z = 0、C = 1、V = 1 に設定されます。

`FCMPZS` で無効演算例外が発生するのは、オペランドにいずれかのタイプの NaN が含まれている場合のみです。よって比較結果において NaN 以外の原因による例外を発生させたくない命題のテストに有効です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.13 FCPYD

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	0	1	1	0	0	0	0	Dd	1	0	1	1	0	1	0	0	Dm

FCPYD（浮動小数点コピー、倍精度）命令は、1 つの倍精度レジスタの内容を他の倍精度レジスタにコピーします。ベクタに対しても同じ演算を実行できます。

構文

```
FCPYD{<cond>}  <Dd>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Dm> ソースレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dm[i]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、FCPYD は 1 つのコピーを実行し、vec_len = 1、Dd[0] = Dd、Dm[0] = Dm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FCPYD は複数のコピーを実行する可能性があります。FCPYD で使用されるレジスタのエンコードと、vec_len、Dd[i]、Dm[i] の決定方法については、P. C5-18 *「アドレシングモード 4 - 倍精度ベクタ（単項）」*を参照して下さい。

Flush-to-Zero モード

FPSCR の FZ ビットは、この命令のオペランドや結果に影響しません。

シグナル NaN

VFP アーキテクチャに準拠するため、FCPYD はソースレジスタの値がシグナル NaN の場合も例外を生成できません。これは、IEEE754-1985 標準の付録に定義されているよりも厳格な条件です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.14 FCPYS

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	D	1	1	0	0	0	0	Fd	1	0	1	0	0	1	M	0	Fm

FCPYS（浮動小数点コピー、単精度）命令は、1 つの単精度レジスタの内容を他の単精度レジスタにコピーします。ベクタに対しても同じ演算を実行できます。

構文

```
FCPYS{<cond>}  <Sd>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sm> ソースレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sm[i]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、FCPYS は 1 つのコピーを実行し、vec_len = 1、Sd[0] = Sd、Sm[0] = Sm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FCPYS は複数のコピーを実行する可能性があります。FCPYS で使用されるレジスタのエンコードと、vec_len、Sd[i]、Sm[i] の決定方法については、P. C5-14 *「アドレシングモード 3 - 単精度ベクタ（単項）」*を参照して下さい。

Flush-to-Zero モード

FPSCR の FZ ビットは、この命令のオペランドや結果に影響しません。

シグナル NaN

VFP アーキテクチャに準拠するため、FCPYS はソースレジスタの値がシグナル NaN の場合も例外を生成できません。これは、IEEE754-1985 標準の付録に定義されているよりも厳格な条件です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.15 FCVTDS

31–28	27–23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
cond	1 1 1 0 1	0	1 1	0 1 1 1	Dd	1 0 1 0	1	1	M	0	Fm

FCVTDS（浮動小数点の単精度から倍精度への変換）命令は、単精度レジスタの値を倍精度に変換し、結果を倍精度レジスタに書き込みます。

構文

```
FCVTDS{<cond>}  <Dd>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Sm> ソースレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、入力非正規

動作

```
if ConditionPassed(cond) then
    Dd = ConvertSingleToDouble(Sm)
```

注

ベクタ FCVTDS は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.16 FCVTSD

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11	10	9	8	7	6	5	4	3-0
cond	1	1	1	0	1	D	1	1	0	1	1	1	Fd	1	0	1	1	1	1	0	0	Dm

FCVTSD（浮動小数点の倍精度から単精度への変換）命令は、倍精度レジスタの値を単精度に変換し、結果を単精度レジスタに書き込みます。

構文

```
FCVTSD{<cond>}  <Sd>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Dm> ソースレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    Sd = ConvertDoubleToSingle(Dm)
```

注

ベクタ FCVTSD は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

丸め FCVTSD は、完全な丸め付き変換を実行します。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.17 FDIVD

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
cond	1	1	1	0	1	0	0	0	Dn	Dd	1	0	1	1	0	0	0	0	Dm

FDIVD（浮動小数点除算、倍精度）命令は、倍精度レジスタの値を別の倍精度レジスタの値で除算し、結果を 3 番目の倍精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FDIVD{<cond>}  <Dd>, <Dn>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Dn> 除算の最初のオペランドレジスタを指定します。

<Dm> 除算の 2 番目のオペランドレジスタを指定します。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、0 による除算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dn[i] / Dm[i]
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

除算には、ほとんどの実装で多くのサイクルが必要であり、ベクタ除算は比例的に必要サイクル数が多くなります。これにより、パフォーマンスに大きな影響が出ることがあります。

同じ数による除算を多く行う場合、1 回の除算で逆数を計算し、その逆数を使用して多くの乗算を行うことで一般にパフォーマンスを改善できます。これにより、丸め誤差が 1 回ではなく 2 回発生するため、計算の精度が多少低下しますが、通常は許容範囲なトレードオフです。

割り込みレイテンシの一部の考慮点について、P. C1-8 *「割り込み」*も参照して下さい。

注

ベクタ FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、`FDIVD` は 1 つの除算を実行し、`vec_len = 1`、`Dd[0] = Dd`、`Dn[0] = Dn`、`Dm[0] = Dm` です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、`FDIVD` は複数の除算を実行する可能性があります。`FDIVD` で使用されるレジスタのエンコードと、`vec_len`、`Dd[i]`、`Dn[i]`、`Dm[i]` の決定方法については、P. C5-8 *「アドレシングモード 2 - 倍精度ベクタ（非単項）」*を参照して下さい。

丸め この演算は完全な丸めが行われる除算です。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.18 FDIVS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	D	0	0	Fn	Fd	1	0	1	0	N	0	M	0	Fm

FDIVS（浮動小数点除算、単精度）命令は、単精度レジスタの値を別の単精度レジスタの値で除算し、結果を 3 番目の単精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FDIVS{<cond>}  <Sd>, <Sn>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sn> 除算の最初のオペランドレジスタを指定します。レジスタ番号は、Fn（上位 4 ビット）と N（最下位ビット）としてエンコードされます。

<Sm> 除算の 2 番目のオペランドレジスタを指定します。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、0 による除算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sn[i] / Sm[i]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用法

除算には、ほとんどの実装で多くのサイクルが必要であり、ベクタ除算は比例的に必要サイクル数が多くなります。これにより、パフォーマンスに大きな影響が出ることがあります。

同じ数による除算を多く行う場合、1 回の除算で逆数を計算し、その逆数を使用して多くの乗算を行うことで一般にパフォーマンスを改善できます。これにより、丸め誤差が 1 回ではなく 2 回発生するため、計算の精度が多少低下しますが、通常は許容範囲なトレードオフです。

割り込みレイテンシの一部の考慮点について、P. C1-8 *「割り込み」*も参照して下さい。

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、`FDIVS` は 1 つの除算を実行し、`vec_len = 1`、`Sd[0] = Sd`、`Sn[0] = Sn`、`Sm[0] = Sm` です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、`FDIVS` は複数の除算を実行する可能性があります。`FDIVS` で使用されるレジスタのエンコードと、`vec_len`、`Sd[i]`、`Sn[i]`、`Sm[i]` の決定方法については、P. C5-2 *「アドレシングモード 1 - 単精度ベクタ（非単項）」*を参照して下さい。

丸め　この演算は完全な丸めが行われる除算です。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.19 FLDD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
cond	1	1	0	1	U	0	0	1	Rn	Dd	1	0	1	1	offset

FLDD（浮動小数点ロード、倍精度）命令は、倍精度レジスタにメモリから値をロードします。

構文

```
FLDD{<cond>}  <Dd>, [<Rn>{, #+/-(<offset>*4)}]
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Rn> 転送のベースアドレスを保持している ARM レジスタを示します。

<offset> offset に 4 を掛け、ベースアドレスに加算（U == 1 の場合）または減算（U == 0 の場合）することで、転送の実際のアドレスが計算されます。このオフセットが省略されている場合、デフォルトは +0 です。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (U == 1)
        address = Rn + offset * 4
    else
        address = Rn - offset * 4
    if (big-endian)
        Dd = (Memory[address,4] << 32) OR Memory[address+4,4]
    else
        Dd = (Memory[address+4,4] << 32) OR Memory[address,4]
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

アドレシングモード

これは、P. C5-22「*アドレシングモード5 - VFP 複数ロード/ストア*」の特別なケースです。

変換

プログラマモデルでは、FLDD は転送される値に対して変換を実行しません。実装では、必要に応じて値を正しい倍精度値に回復できる限り、転送される値を自由に内部形式に変換できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.20 FLDMD

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7 0
cond	1	1	0	P	U	0	W	1	Rn	Dd	1	0	1	1	offset

FLDMD（浮動小数点複数ロード、倍精度）命令は、連続した複数の倍精度レジスタにメモリから値をロードします。

構文

```
FLDM<addressing_mode>D{<cond>}  <Rn>{!}, <registers>
```

各ビットの説明については以下を参照して下さい。

<addressing_mode>
: 命令によって使用される開始アドレスと終了アドレスの値を決定するアドレシングモードを指定します。詳細については、P. C5-22 *「アドレシングモード 5 - VFP 複数ロード / ストア」*を参照して下さい。

<cond>
: この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rn>
: <addressing_mode> で使用するベースレジスタを指定します。

!
: 命令の W ビットを 1 に設定し、ベースレジスタ <Rn> が命令によって更新されることを示します。この値が省略されている場合、命令の W ビットは 0 に設定され、ベースレジスタ <Rn> は変更されません。<addressing_mode> と ! の存在 / 非存在の組み合わせの一部は許容されません。詳細については、P. C5-22 *「アドレシングモード 5 - VFP 複数ロード / ストア」*を参照して下さい。

<registers>
: ロードされるレジスタを、カンマで区切られ角かっこで囲まれた、連続した番号を持つ倍精度レジスタの一覧として指定します。レジスタ番号は、Dd を一覧の最初のレジスタの番号、offset を一覧に含まれるレジスタ番号の 2 倍に設定することで、命令にエンコードされます。一覧には、最低 1 つのレジスタを指定する必要があります。

  たとえば、<registers> が {D2,D3,D4} の場合、命令の Dd フィールドは 2、offset フィールドは 6 になります。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to (offset-2)/2
        /* d is the number of register Dd;                  */
        /* D(n) is the double-precision register numbered n */
        if (big-endian)
            D(d+i) = (Memory[address,4] << 32) OR Memory[address+4,4]
        else
            D(d+i) = (Memory[address+4,4] << 32) OR Memory[address,4]
        address = address + 8
    assert end_address = address - 4
```

注

エンコード P = 1 かつ W = 0 の場合、命令は FLDD となります。それ以外の場合、オフセットが奇数であれば命令は FLDMX となります。

ベクタ FLDMD命令はFPSCRのLENおよびSTRIDEフィールドの影響を受けず､データ処理命令のベクタオペランドのようにバンク境界でラップアラウンドすることはありません。レジスタは単純に、レジスタ番号の昇順にロードされます。

無効なレジスタ指定

Dd と offset が有効なレジスタ一覧を指定していない場合、命令の動作は予測不能です。これは次の 2 つの条件で発生することがあります。

- offset == 0 の場合、つまりどのレジスタも転送しない命令が指定された場合
- d + offset / 2 > 16 の場合、つまり D15 よりも後のレジスタを転送しようとした場合

変換 プログラマモデルでは、FLDMD は転送される値に対して変換を実行しません。実装では、必要に応じて値を正しい倍精度値に回復できる限り、転送される値を自由に内部形式に変換できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.21 FLDMS

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 10 9 8	7 0
cond	1 1 0	P	U	D	W	1	Rn	Fd	1 0 1 0	offset

`FLDMS`（浮動小数点複数ロード、単精度）命令は、連続した複数の単精度レジスタにメモリから値をロードします。

構文

```
FLDM<addressing_mode>S{<cond>}  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

`<addressing_mode>`
: 命令によって使用される開始アドレスと終了アドレスの値を決定するアドレシングモードを指定します。詳細については、P. C5-22 *「アドレシングモード 5 - VFP 複数ロード / ストア」*を参照して下さい。

`<cond>`
: この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。`<cond>` が省略されている場合、`AL`（常時）条件が使用されます。

`<Rn>`
: `<addressing_mode>` で使用するベースレジスタを指定します。

`!`
: 命令の W ビットを 1 に設定し、ベースレジスタ `<Rn>` が命令によって更新されることを示します。この値が省略されている場合、命令の W ビットは 0 に設定され、ベースレジスタ `<Rn>` は変更されません。`<addressing_mode>` と `!` の存在 / 非存在の組み合わせの一部は許容されません。詳細については、P. C5-22 *「アドレシングモード 5 - VFP 複数ロード / ストア」*を参照して下さい。

`<registers>`
: ロードされるレジスタを、カンマで区切られ角かっこで囲まれた、連続した番号を持つ単精度レジスタの一覧として指定します。`d` が一覧の最初のレジスタの場合、一覧は `Fd` と `D` をそれぞれ `d` の上位 4 ビットと最下位ビットに設定し、`offset` を一覧のレジスタ数として、命令にエンコードされます。一覧には、最低 1 つのレジスタを指定する必要があります。

  たとえば、`<registers>` が `{S5,S6,S7}` の場合、命令の Fd フィールドは 0b0010、D ビットは 1、offset フィールドは 3 になります。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to offset-1
        /* d is as defined for <registers> above;           */
        /* S(n) is the single-precision register numbered n */
        S(d+i) = Memory[address,4]
        address = address + 4
    assert end_address = address - 4
```

注

エンコード P = 1 かつ W = 0 の場合、命令は FLDS となります。

ベクタ FLDMD命令はFPSCRのLENおよびSTRIDEフィールドの影響を受けず、データ処理命令のベクタオペランドのようにバンク境界でラップアラウンドすることはありません。レジスタは単純に、レジスタ番号の昇順にロードされます。

無効なレジスタ指定

Fd、D、offset が有効なレジスタ一覧を指定していない場合、命令の動作は予測不能です。これは次の 2 つの条件で発生することがあります。

- offset == 0 の場合、つまりどのレジスタも転送しない命令が指定された場合
- d + offset > 32 の場合、つまり S31 よりも後のレジスタを転送しようとした場合

変換 プログラマモデルでは、FLDMS は転送される値に対して変換を実行しません。メモリワードには、整数と単精度浮動小数点数値のどちらも保持可能です。ほとんどの VFP 演算命令は、ロードされた値を単精度浮動小数点数値として扱います。それらの値が整数の場合、そのような演算命令で正しい結果を得るには、整数から浮動小数点への変換命令を使用して変換する必要があります。実装では、必要に応じて値を正しい単精度値または正しい整数値に回復できる（その後でレジスタが使用される方法によって異なります）限り、転送される値を自由に内部形式に変換できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.22 FLDMX

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
cond	1	1	0	P	U	0	W	1	Rn	Dd	1	0	1	1	offset

FLDMX（浮動小数点複数ロード、精度不明）命令は、連続した複数の倍精度レジスタにメモリから値をロードします。これによって、レジスタに整数、単精度数値、倍精度数値を正しく再ロードできます。

——— 注 ———

FLDMX 命令は、ARMv6 では推奨されません。データの精度が不明な場合、値の保存と復元には FLDMD を使用して下さい。

構文

```
FLDM<addressing_mode>X{<cond>}  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<addressing_mode>
: 命令によって使用される開始アドレスと終了アドレスの値を決定するアドレシングモードを指定します。詳細については、P. C5-22「*アドレシングモード 5 - VFP 複数ロード / ストア*」を参照して下さい。

<cond>
: この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rn>
: <addressing_mode> で使用するベースレジスタを指定します。

!
: 命令の W ビットを 1 に設定し、ベースレジスタ <Rn> が命令によって更新されることを示します。この値が省略されている場合、命令の W ビットは 0 に設定され、ベースレジスタ <Rn> は変更されません。<addressing_mode> と ! の存在 / 非存在の組み合わせの一部は許容されません。詳細については、P. C5-22「*アドレシングモード 5 - VFP 複数ロード / ストア*」を参照して下さい。

<registers>
: ロードされるレジスタを、カンマで区切られ角かっこで囲まれた、連続した番号を持つ倍精度レジスタの一覧として指定します。レジスタ番号は、Dd を一覧の最初のレジスタの番号、offset を一覧に含まれるレジスタ番号の 2 倍に 1 を加えた値に設定することで、命令にエンコードされます。一覧には、最低 1 つのレジスタを指定する必要があります。

たとえば、<registers> が {D2,D3,D4} の場合、命令の Dd フィールドは 2、offset フィールドは 7 になります。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to (offset-3)/2
        /* d is the number of register Dd;                  */
        /* D(n) is the double-precision register numbered n */
        if (big-endian)
            D(d+i) = (Memory[address,4] << 32) OR Memory[address+4,4]
        else
            D(d+i) = (Memory[address+4,4] << 32) OR Memory[address,4]
        address = address + 8
    assert end_address = address - 4
```

用法

FLDMX は、以前に FSTMX で VFP レジスタの値をメモリにストアした場合、その値を VFP レジスタに再ロードするために使用されます。この命令は、次のような場合に使用されます。

- 標準のプロシージャコールにて保存されたデータのプロシージャ終了時における保存先からの復帰
- プロセススワップコード

注

エンコード　P = 1 かつ W = 0 の場合、命令は FLDD となります。それ以外の場合、offset が偶数であれば、命令は FLDMD となります。

ベクタ　FLDMX命令はFPSCRのLENおよびSTRIDEフィールドの影響を受けず、データ処理命令のベクタオペランドのようにバンク境界でラップアラウンドすることはありません。レジスタは単純に、レジスタ番号の昇順にロードされます。

無効なレジスタ指定

Dd と offset が有効なレジスタ一覧を指定していない場合、命令の動作は予測不能です。これは次の 2 つの条件で発生することがあります。

- offset == 0 の場合、つまりどのレジスタも転送しない命令が指定された場合
- d + (offset - 1) / 2 > 16 の場合、つまり D15 よりも後のレジスタを転送しようとした場合

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.23 FLDS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
cond	1	1	0	1	U	D	0	1	Rn	Fd	1	0	1	0	offset

FLDS（浮動小数点ロード、単精度）命令は、単精度レジスタにメモリから値をロードします。

構文

```
FLDS{<cond>}  <Sd>, [<Rn>{, #+/-(<offset>*4)}]
```

各項目の説明については以下を参照して下さい。

`<cond>` この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。`<cond>` が省略されている場合、AL（常時）条件が使用されます。

`<Sd>` デスティネーションレジスタ。番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

`<Rn>` 転送のベースアドレスを保持している ARM レジスタ。

`<offset>` offset に 4 を掛け、ベースアドレスに加算（U == 1 の場合）または減算（U == 0 の場合）することで、転送の実際のアドレスが計算されます。このオフセットが省略されている場合、デフォルトは +0 です。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (U == 1)
        address = Rn + offset * 4
    else
        address = Rn - offset * 4
    Sd = Memory[address,4]
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

アドレシングモード

これは、P. C5-22「*アドレシングモード5 - VFP 複数ロード/ストア*」の特別なケースです。

変換 プログラマモデルでは、`FLDS` は転送される値に対して変換を実行しません。メモリワードには、整数と単精度浮動小数点数のどちらも保持可能です。ほとんどの VFP 演算命令は、`Sd` の値を単精度浮動小数点数として扱います。それらの値が整数の場合、そのような演算命令で正しい結果を得るには、整数から浮動小数点への変換命令を使用して変換する必要があります。実装では、必要に応じて値を正しい単精度値または正しい整数値に回復できる（その後で `Sd` が使用される方法によって異なります）限り、転送される値を自由に内部形式に変換できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.24 FMACD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	0	0	0	Dn	Dd	1	0	1	1	0	0	0	0	Dm

FMACD（浮動小数点積和、倍精度）命令は、2 つの倍精度レジスタの値を乗算し、その積に 3 番目の倍精度レジスタの値を加算して、結果を 3 番目のレジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FMACD{<cond>} <Dd>, <Dn>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。このレジスタは、加算の最初のオペランドとしても使用されます。

<Dn> 乗算の最初のオペランドレジスタ。

<Dm> 乗算の 2 番目のオペランドレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dd[i] + (Dn[i] * Dm[i])
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、FMACD は 1 つの積和演算を実行し、vec_len = 1、Dd[0] = Dd、Dn[0] = Dn、Dm[0] = Dm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FMACD は複数の積和演算を実行する可能性があります。FMACD で使用されるレジスタのエンコードと、vec_len、Dd[i]、Dn[i]、Dm[i] の決定方法については、P. C5-8 *「アドレシングモード 2 - 倍精度ベクタ（非単項）」*を参照して下さい。

丸め　この演算では、完全な丸めが行われる乗算に続いて、完全な丸めが行われる加算が実行されます。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.25 FMACS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	D	0	0	Fn	Fd	1	0	1	0	N	0	M	0	Fm

FMACS（浮動小数点積和、単精度）命令は、2 つの単精度レジスタの値を乗算し、その積に 3 番目の単精度レジスタの値を加算して、結果を 3 番目のレジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FMACS{<cond>} <Sd>, <Sn>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond>　この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd>　デスティネーションレジスタ。このレジスタは、加算の最初のオペランドとしても使用されます。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sn>　乗算の最初のオペランドレジスタ。レジスタ番号は、Fn（上位 4 ビット）と N（最下位ビット）としてエンコードされます。

<Sm>　乗算の 2 番目のオペランドレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sd[i] + (Sn[i] * Sm[i])
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ

FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、`FMACS` は 1 つの積和演算を実行し、`vec_len = 1`、`Sd[0] = Sd`、`Sn[0] = Sn`、`Sm[0] = Sm` です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、`FMACS` は複数の積和演算を実行する可能性があります。`FMACS` で使用されるレジスタのエンコードと、`vec_len`、`Sd[i]`、`Sn[i]`、`Sm[i]` の決定方法については、P. C5-2 *「アドレシングモード 1 - 単精度ベクタ（非単項）」*を参照して下さい。

丸め

この演算では、完全な丸めが行われる乗算に続いて、完全な丸めが行われる加算が実行されます。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.26 FMDHR

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6–5	4	3–0
cond	1	1	1	0	0	0	1	0	Dn	Rd	1	0	1	1	0	SBZ	1	SBZ

FMDHR（ARM レジスタから浮動小数点倍精度レジスタ上位ワードへの転送）命令は、ARM レジスタ Rd の内容を倍精度レジスタ Dn の上位半分に転送します。この命令は、FMDLR と組み合わせて、ARM レジスタと浮動小数点レジスタの間で倍精度値を転送するために使用されます。

構文

```
FMDHR{<cond>}  <Dn>, <Rd>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dn> デスティネーションレジスタ。

<Rd> ソースとなる ARM レジスタ。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Dn[63:32] = Rd
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

FMDLR と組み合わせての使用

FMDHR は、同じデスティネーションレジスタを指定している FMDLR 命令と組み合わせて使用する必要があります。片方のみが実行された状態ではDnの内容は予測不能です。

- 2 番目の命令の実行は、2 つの命令の実行で転送される倍精度数を含む <Dn> に対して転送を行う必要があります。
- Dn が FSTMX 命令によってメモリに保存され、後で正確に一致する FLDMX 命令によって再ロードされる場合、<Dn> の最終的な値は元の値と機能的に等価な必要があります。

変換　プログラマモデルでは、FMDHR と FMDLR の組み合わせは転送される値に対して変換を実行しません。実装では、FMDHR と FMDLR 命令の両方が実行されたときに値を正しい倍精度値に回復できる限り、転送される値を自由に内部形式に変換できます。

R15 の使用　レジスタ <Rd> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.27 FMDLR

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6–5	4	3–0
cond	1	1	1	0	0	0	0	0	Dn	Rd	1	0	1	1	0	SBZ	1	SBZ

FMDLR（ARM レジスタから浮動小数点倍精度レジスタの下位ワードへの転送）命令は、ARM レジスタ Rd の内容を倍精度レジスタ Dn の下位半分に転送します。この命令は、FMDHR と組み合わせて、ARM レジスタと浮動小数点レジスタの間で倍精度値を転送するために使用されます。

構文

```
FMDLR{<cond>}  <Dn>, <Rd>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dn> デスティネーションレジスタ。

<Rd> ソースとなる ARM レジスタ。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Dn[31:0] = Rd
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

FMDHR と組み合わせての使用

FMDLR は、同じデスティネーションレジスタを指定している FMDHR 命令と組み合わせて使用する必要があります。片方のみが実行された状態では Dn の内容は予測不能です。

- 2 番目の命令の実行は、2 つの命令の実行で転送される倍精度数を含む <Dn> に対して転送を行う必要があります。
- Dn が FSTMX 命令によってメモリに保存され、後で正確に一致する FLDMX 命令によって再ロードされる場合、<Dn> の最終的な値は元の値と機能的に等価な必要があります。

変換 プログラマモデルでは、FMDHR と FMDLR の組み合わせは転送される値に対して変換を実行しません。実装では、FMDHR と FMDLR 命令の両方が実行されたときに値を正しい倍精度値に回復できる限り、転送される値を自由に内部形式に変換できます。

R15 の使用 レジスタ <Rd> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.28 FMDRR

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
cond	1	1	0	0	0	1	0	0	Rn	Rd	1	0	1	1	0	0	0	1	Dm

FMDRR（2 つの ARM レジスタから倍精度レジスタへ浮動小数点数値の転送）命令は、2 つの ARM レジスタの内容を倍精度 VFP レジスタに転送します。

構文

```
FMDRR{<cond>}  <Dm>, <Rd>, <Rn>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dm> デスティネーションとなる倍精度 VFP レジスタ。

<Rd> 64 ビットオペランドの下位半分を含むソース ARM レジスタ。

<Rn> 64 ビットオペランドの上位半分を含むソース ARM レジスタ。

アーキテクチャのバージョン

VFPv2 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Dm[63:32] = Rn
    Dm[31:0]  = Rd
```

注

変換 プログラマモデルでは、FMDRR は転送される値に対して変換を実行しません。いずれの ARM レジスタに対する演算命令でも、内容は整数として扱われます。ほとんどの VFP 命令は、Dm の値を倍精度浮動小数点数値として扱います。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.29 FMRDH

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6–5	4	3–0
cond	1	1	1	0	0	0	1	1	Dn	Rd	1	0	1	1	0	SBZ	1	SBZ

FMRDH（浮動小数点倍精度レジスタの上位ワードを ARM レジスタに転送）命令は、倍精度レジスタ Dn の内容の上位半分を ARM レジスタ Rd に転送します。この命令は、FMRDL と組み合わせて、ARM レジスタと浮動小数点レジスタの間で倍精度値を転送するために使用されます。

構文

```
FMRDH{<cond>}  <Rd>, <Dn>
```

各項目の説明については以下を参照して下さい。

<cond>　この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd>　デスティネーションとなる ARM レジスタ。

<Dn>　ソースレジスタ。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = Dn[63:32]
```

注

変換　実装で倍精度値について内部形式を使用している場合、変換により外部の倍精度形式に戻す必要があります。それ以外の場合、変換は必要ありません。

R15 の使用　レジスタ <Rd> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.30 FMRDL

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6–5	4	3–0
cond	1	1	1	0	0	0	0	1	Dn	Rd	1	0	1	1	0	SBZ	1	SBZ

FMRDL（浮動小数点倍精度レジスタの下位ワードを ARM レジスタに転送）命令は、倍精度レジスタ Dn の内容の下位半分を ARM レジスタ Rd に転送します。この命令は、FMRDH と組み合わせて、ARM レジスタと浮動小数点レジスタの間で倍精度値を転送するために使用されます。

構文

```
FMRDL{<cond>}  <Rd>, <Dn>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションとなる ARM レジスタ。

<Dn> ソースレジスタ。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = Dn[31:0]
```

注

変換 実装で倍精度値について内部形式を使用している場合、変換により外部の倍精度形式に戻す必要があります。それ以外の場合、変換は必要ありません。

R15 の使用 レジスタ <Rd> として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.31 FMRRD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	0	0	0	1	0	1	Rn	Rd	1	0	1	1	0	0	0	1	Dm

FMRRD（浮動小数点倍精度レジスタから 2 つの ARM レジスタへの転送）命令は、倍精度 VFP レジスタの内容を 2 つの ARM レジスタに転送します。

構文

```
FMRRD{<cond>}  <Rd>, <Rn>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> 64 ビットオペランドの下位半分が転送されるデスティネーション ARM レジスタ。

<Rn> 64 ビットオペランドの上位半分が転送されるデスティネーション ARM レジスタ。

<Dm> ソースとなる倍精度 VFP レジスタ。

アーキテクチャのバージョン

VFPv2 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rn = Dm[63:32]
    Rd = Dm[31:0]
```

注

R15 の使用 レジスタ <Rd> または <Rn> として R15 を指定した場合、結果は予測不能です。

変換 プログラマモデルでは、FMRRD は転送される値に対して変換を実行しません。いずれの ARM レジスタに対する演算命令でも、内容は整数として扱われます。ほとんどの VFP 命令は、Dm の値を倍精度浮動小数点数値として扱います。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.32 FMRRS

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
cond	1	1	0	0	0	1	0	1	Rn	Rd	1	0	1	0	0	0	M	1	Sm

FMRRS（2 つの浮動小数点単精度レジスタから 2 つの ARM レジスタへの転送）命令は、連続した番号を持つ 2 つの単精度 VFP レジスタの内容を 2 つの ARM レジスタに転送します。ARM レジスタは連続している必要はありません。

構文

```
FMRRS{<cond>}  <Rd>, <Rn>, {<Sm>, <Sm1>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> 単精度値 Sm が転送されるデスティネーション ARM レジスタ。

<Rn> 単精度値 Sm^1 が転送されるデスティネーション ARM レジスタ。

<Sm> ソースの最初の単精度 VFP レジスタ。これは、m の上位 4 ビットを Sm に、最下位ビットを M にセットすることで命令にエンコードされます。

<Sm^1> ソースの 2 番目の単精度 VFP レジスタ。これは、<Sm> の次の単精度 VFP レジスタです。

アーキテクチャのバージョン

VFPv2 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rn = Sm1
    Rd = Sm
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

R15 の使用 <Rd> または <Rn> として R15 を指定した場合、結果は予測不能です。

変換 プログラマモデルでは、FMRRS は転送される値に対して変換を実行しません。いずれの ARM レジスタに対する演算命令でも、内容は整数として扱われます。ほとんどの VFP 命令は、Sm と Sn の値を単精度浮動小数点数として扱います。

無効なレジスタ指定

<Sm> として S31 を指定した場合、結果は予測不能です。レジスタの組が連続していない場合、アセンブラによってエラーが報告されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.33 FMRS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6–5	4	3–0
cond	1	1	1	0	0	0	0	1	Fn	Rd	1	0	1	0	N	SBZ	1	SBZ

FMRS（単精度浮動小数点レジスタから ARM レジスタへの転送）命令は、単精度レジスタ Fn の内容を ARM レジスタ Rd に転送します。転送される値は、整数（通常は FTOSID、FTOSIS、FTOUID、FTOUIS 命令により生成されます）でも単精度浮動小数点数（通常は他の演算命令により生成されます）でもかまいません。

構文

```
FMRS{<cond>}  <Rd>, <Sn>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションとなる ARM レジスタ。

<Sn> ソースレジスタ。番号は、Fn（上位 4 ビット）と N（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = Sn
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

変換 プログラマモデルでは、FMRS は転送される値に対して変換を実行しません。ソースレジスタ Sn とデスティネーションレジスタ Rd はいずれも、整数と単精度浮動小数点数のどちらも保持可能です。ARM の演算命令では Rd の値は整数として扱われ、VFP コプロセッサのほとんどの演算命令では Sn の値は単精度浮動小数点数として扱われます。これらで表現される数値を一致させるには、FMRS 命令の前に浮動小数点から整数への変換命令の 1 つを実行する必要があります。

実装では、FMRS によって内部形式を外部形式に変換でき、レジスタに含まれている値が単精度浮動小数点数か整数かにかかわらず正しいデータを回復できる限り、Sn の値を自由に内部形式で保持できます。

R15 の使用 Rd として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.34 FMRX

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6–5	4	3–0
cond	1	1	1	0	1	1	1	1	reg	Rd	1	0	1	0	0	SBZ	1	SBZ

FMRX（浮動小数点システムレジスタから ARM レジスタへの転送）命令は、VFP システムレジスタの 1 つの内容を ARM レジスタ Rd に転送します。

構文

```
FMRX{<cond>}  <Rd>, <reg>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rd> デスティネーションとなる ARM レジスタ。

<reg> ソースとなるシステムレジスタを次のように指定します。

```
<reg> = 0b0000: FPSID
<reg> = 0b0001: FPSCR
<reg> = 0b1000: FPEXC
```

個別の VFP 実装で、実装定義の目的に、他の <reg> の値を使用することもできます。一般に、これらはハードウェアコプロセッサから、コプロセッサのサポートコードにデータを転送するために使用されます。

他のすべてのコードでは、このような <reg> の値を予測不能として扱う必要があります。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Rd = reg
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

直列化 FMRX は直列化命令です。この意味の詳細については、P. C3-21 *「システムレジスタ転送命令」*を参照して下さい。

例外処理 VFP システムにハードウエアコプロセッサがある場合、直列化の後、そのコプロセッサに保留中の例外が発生する可能性があります。FMRX 命令がそのような例外の処理を起動するかどうかは、以下に示すように、転送されるシステムレジスタに依存します。

FPSID の読み出し

ソースに FPSID を指定した FMRX 命令は、どの ARM プロセッサモードからも実行できます。直列化の後で、FPSID の値が Rd に書き込まれ、例外処理は起動されません。

FPSCR の読み出し

ソースに FPSCR を指定した FMRX 命令は、どの ARM プロセッサモードからも実行できます。直列化の後で、必要に応じて例外処理が起動されます。それ以外の場合、FPSCR の値が Rd に書き込まれます。

FPEXC の読み出し

ソースに FPEXC を指定した FMRX 命令は、特権 ARM プロセッサモードからのみ実行できます。ユーザモードでこの命令を実行しようとすると、ARM の未定義命令例外が発生します。

直列化の後で、FPEXC の値が Rd に書き込まれ、例外処理は起動されません。FPEXC のビット [31:30] 以外のビットはすべて実装定義なため、実装固有でないコードは Rd に書き込まれた値のビット [31:0] のみに依存する必要があります。

R15 の使用 ソースシステムレジスタが FPSCR 以外で、レジスタ Rd として R15 を指定した場合、結果は予測不能です。ソースシステムレジスタが FPSCR の場合、この命令は FMSTAT 命令と同じです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.35 FMSCD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	0	0	1	Dn	Dd	1	0	1	1	0	0	0	0	Dm

FMSCD（浮動小数点積減算、倍精度）命令は、2 つの倍精度レジスタの値を乗算し、その積に 3 番目の倍精度レジスタの値を符号反転して加算し、結果を 3 番目のレジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FMSCD{<cond>}  <Dd>, <Dn>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。このレジスタの値は、符号反転して加算する最初のオペランドとしても使用されます。

<Dn> 乗算の最初のオペランドレジスタ。

<Dm> 乗算の 2 番目のオペランドレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = neg(Dd[i]) + (Dn[i] * Dm[i])
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、`FMSCD` は 1 つの積減算演算を実行し、`vec_len = 1`、`Dd[0] = Dd`、`Dn[0] = Dn`、`Dm[0] = Dm` です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、`FMSCD` は複数の積減算演算を実行する可能性があります。`FMSCD` で使用されるレジスタのエンコードと、`vec_len`、`Dd[i]`、`Dn[i]`、`Dm[i]` の決定方法については、P. C5-8 *「アドレシングモード 2 - 倍精度ベクタ（非単項）」*を参照して下さい。

丸め　この演算では完全な丸めが行われる乗算に続いて、完全な丸めが行われる減算が実行されます。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.36 FMSCS

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
cond	1	1	1	0	0	D	0	1	Fn	Fd	1	0	1	0	N	0	M	0	Fm

FMSCS（浮動小数点積減算、単精度）命令は、2 つの単精度レジスタの値を乗算し、その積に 3 番目の単精度レジスタの値を符号反転して加算し、結果を 3 番目のレジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FMSCS{<cond>}  <Sd>, <Sn>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。このレジスタの値は、符号反転して加算する最初のオペランドとしても使用されます。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sn> 乗算の最初のオペランドレジスタ。レジスタ番号は、Fn（上位 4 ビット）と N（最下位ビット）としてエンコードされます。

<Sm> 乗算の 2 番目のオペランドレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = neg(Sd[i]) + (Sn[i] * Sm[i])
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、FMSCS は 1 つの積減算演算を実行し、`vec_len = 1`、`Sd[0] = Sd`、`Sn[0] = Sn`、`Sm[0] = Sm` です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FMSCS は複数の積減算演算を実行する可能性があります。FMSCS で使用されるレジスタのエンコードと、`vec_len`、`Sd[i]`、`Sn[i]`、`Sm[i]` の決定方法については、P. C5-2 *「アドレシングモード 1 - 単精度ベクタ（非単項）」*を参照して下さい。

丸め　この演算では完全な丸めが行われる乗算に続いて、完全な丸めが行われる減算が実行されます。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.37 FMSR

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6–5	4	3–0
cond	1	1	1	0	0	0	0	0	Fn	Rd	1	0	1	0	N	SBZ	1	SBZ

FMSR（ARM レジスタから浮動小数点単精度レジスタへの転送）命令は、ARM レジスタ Rd の内容を単精度レジスタ Fn に転送します。転送される値は、その後で整数（FSITOD、FSITOS、FUITOD、FUITOS 命令のソースレジスタとして使用される場合）と単精度浮動小数点数値（他の演算命令で使用される場合）のどちらとして扱われてもかまいません。

構文

```
FMSR{<cond>}  <Sn>, <Rd>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sn> デスティネーションレジスタ。番号は、Fn（上位 4 ビット）と N（最下位ビット）としてエンコードされます。

<Rd> ソースとなる ARM レジスタ。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Sn = Rd
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

変換 プログラマモデルでは、FMSR は転送される値に対して変換を実行しません。ソースレジスタ Rd とデスティネーションレジスタ Sn はいずれも、整数と単精度浮動小数点数値のどちらも保持可能です。ARM の演算命令では Rd の値は整数として扱われ、VFP コプロセッサのほとんどの演算命令では Fn の値は単精度浮動小数点数値として扱われます。転送されるのが整数の場合、以降の VFP 命令で正しい結果を得るには、FMSR 命令の後で整数から浮動小数点への変換命令を使用する必要があります。

実装では、必要に応じて値を正しい単精度数値または正しい整数値に回復できる（その後で Sn が使用される方法によって異なります）限り、転送される値を自由に内部形式に変換できます。

R15 の使用 レジスタ Rd として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.38 FMSRR

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	0	0	0	1	0	0	Rn	Rd	1	0	1	0	0	0	M	1	Sm

FMSRR（2 つの ARM レジスタから 2 つの浮動小数点単精度レジスタへの転送）命令は、2 つの ARM レジスタの内容を 2 つの単精度 VFP レジスタに転送します。

構文

```
FMSRR{<cond>}   {<Sm>, <Sm1>}, <Rd>, <Rn>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sm> 最初のデスティネーションとなる単精度 VFP レジスタ。これは、m の上位 4 ビットを Sm に、最下位ビットを M にセットすることで命令にエンコードされます。

<Sm^1> 2 番目のデスティネーションとなる単精度 VFP レジスタ。

<Rd> VFP 単精度レジスタ Sm のソースとなる ARM レジスタ。これは、<Sm> の次の単精度 VFP レジスタです。

<Rn> VFP 単精度レジスタ Sm^1 のソースとなる ARM レジスタ。

アーキテクチャのバージョン

VFPv2 以降に存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Sm = Rd
    Sm1 = Rn
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

変換　プログラマモデルでは、FMSRR は転送される値に対して変換を実行しません。いずれの ARM レジスタに対する演算命令でも、内容は整数として扱われます。ほとんどの VFP 命令は、Sm と Sn の値を単精度浮動小数点数として扱います。

無効なレジスタ指定

Sm として S31 を指定した場合、結果は予測不能です。レジスタの組が連続していない場合、アセンブラによってエラーが報告されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.39 FMSTAT

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7–5	4	3–0
cond	1	1	1	0	1	1	1	1	0	0	0	1	1	1	1	1	1	0	1	0	SBZ	1	SBZ

FMSTAT（浮動小数点ステータス転送）命令は、FPSCR の N、Z、C、V フラグを ARM の CPSR の対応するフラグに転送します。この命令は一般に、VFP 比較命令で FPSCR フラグが設定された後で使用されます。

構文

```
FMSTAT{<cond>}
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    CPSR N Flag = FPSCR N Flag
    CPSR Z Flag = FPSCR Z Flag
    CPSR C Flag = FPSCR C Flag
    CPSR V Flag = FPSCR V Flag
```

注

エンコード FMSTAT{<cond>} 命令は次のようにエンコードされます。

```
FMRX{<cond>}  r15, FPSCR
```

詳細については、P. C4-62 *「FMRX」* を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.40 FMULD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	0	1	0	Dn	Dd	1	0	1	1	0	0	0	0	Dm

FMULD（浮動小数点乗算、倍精度）命令は、2 つの倍精度レジスタの値を乗算し、結果を 3 番目の倍精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FMULD{<cond>}  <Dd>, <Dn>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Dn> 乗算の最初のオペランドレジスタ。

<Dm> 乗算の 2 番目のオペランドレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dn[i] * Dm[i]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ

FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、FMULD は 1 つの乗算を実行し、vec_len = 1、Dd[0] = Dd、Dn[0] = Dn、Dm[0] = Dm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FMULD は複数の乗算を実行する可能性があります。FMULD で使用されるレジスタのエンコードと、vec_len、Dd[i]、Dn[i]、Dm[i] の決定方法については、P. C5-8 *「アドレシングモード 2 - 倍精度ベクタ（非単項）」*を参照して下さい。

丸め

この演算は完全な丸めが行われる乗算です。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.41 FMULS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	D	1	0	Fn	Fd	1	0	1	0	N	0	M	0	Fm

FMULS（浮動小数点乗算、単精度）命令は、2 つの単精度レジスタの値を乗算し、結果を 3 番目の単精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FMULS{<cond>}  <Sd>, <Sn>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sn> 乗算の最初のオペランドレジスタ。番号は、Fn（上位 4 ビット）と N（最下位ビット）としてエンコードされます。

<Sm> 乗算の 2 番目のオペランドレジスタ。番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sn[i] * Sm[i]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、FMULS は 1 つの乗算を実行し、vec_len = 1、Sd[0] = Sd、Sn[0] = Sn、Sm[0] = Sm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FMULS は複数の乗算を実行する可能性があります。FMULS で使用されるレジスタのエンコードと、vec_len、Sd[i]、Sn[i]、Sm[i] の決定方法については、P. C5-2 *「アドレシングモード 1 - 単精度ベクタ（非単項）」*を参照して下さい。

丸め　この演算は完全な丸めが行われる乗算です。丸めモードは FPSCR によって決定されます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.42 FMXR

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6–5	4	3–0
cond	1	1	1	0	1	1	1	0	reg	Rd	1	0	1	0	0	SBZ	1	SBZ

FMXR（ARM レジスタから浮動小数点システムレジスタへの転送）命令は、ARM レジスタ Rd の内容を VFP システムレジスタの 1 つに転送します。

構文

```
FMXR{<cond>}  <reg>, <Rd>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<reg> デスティネーションシステムレジスタを次のように指定します。

```
<reg> = 0b0000: FPSID
<reg> = 0b0001: FPSCR
<reg> = 0b1000: FPEXC
```

個別の VFP 実装で、実装定義の目的に、他の <reg> の値を使用することもできます。一般に、これらはハードウェアコプロセッサのサポートコードから、コプロセッサにデータを転送するために使用されます。

他のすべてのコードでは、このような <reg> の値を予測不能として扱い、その値に依存しないようにする必要があります。

<Rd> ソースとなる ARM レジスタ。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

未定義命令

動作

```
if ConditionPassed(cond) then
    reg = Rd
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

直列化 FMXR は直列化命令です。この意味の詳細については、P. C3-21「*システムレジスタ転送命令*」を参照して下さい。

例外処理 VFP システムにハードウエアコプロセッサがある場合、直列化の後、そのコプロセッサに保留中の例外が発生する可能性があります。FMXR 命令がそのような例外の処理を起動するかどうかは、以下に示すように、転送されるシステムレジスタに依存します。

FPSID への書き込み

デスティネーションに FPSID を指定した FMXR 命令は、どの ARM プロセッサモードからも実行できます。FPSID は読み出し専用レジスタなため、これは直列化 NOP 命令で、例外処理を起動しません。

FPSCR への書き込み

デスティネーションに FPSCR を指定した FMXR 命令は、どの ARM プロセッサモードからも実行できます。直列化の後で、必要に応じて例外処理が起動されます。それ以外の場合、Rd の値が FPSCR に書き込まれます。

FPEXC への書き込み

デスティネーションに FPEXC を指定した FMXR 命令は、特権 ARM プロセッサモードからのみ実行できます。ユーザモードでこの命令を実行しようとすると、ARM の未定義命令例外が発生します。

直列化の後で、Rd の値が FPEXC に書き込まれ、例外処理は起動されません。FPEXC のビット [31:30] 以外のビットはすべて実装定義なため、実装固有でないコードは読み出し - ビット [31:0] の修正 - 書き込みの一部としてのみ、このような命令を使用する必要があります。

R15 の使用 レジスタ Rd として R15 を指定した場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.43 FNEGD

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
cond	1	1	1	0	1	0	1	1	0 0 0 1	Dd	1	0	1	1	0	1	0	0	Dm

FNEGD（浮動小数点符号反転、倍精度）命令は、倍精度レジスタの値を符号反転し、他の倍精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FNEGD{<cond>}  <Dd>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Dm> ソースレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = neg(Dm[i])
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

符号反転　　関数 neg(x) は、IEEE754-1985 標準の付録で関数 -x として定義されているように、x の符号ビットを反転してコピーする操作を意味します。

Flush-to-Zero モード

FPSCR の FZ ビットは、この命令のオペランドや結果に影響しません。

ベクタ　　FPSCR の LEN フィールドでスカラモードが指定されている場合 (ベクタ長 1)、FNEGD は 1 つのマイナス値演算を実行し、vec_len = 1、Dd[0] = Dd、Dm[0] = Dm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1)、FNEGD は複数のマイナス値演算を実行する可能性があります。FNEGD で使用されるレジスタのエンコードと、vec_len、Dd[i]、Dm[i] の決定方法については、P. C5-18 *「アドレシングモード 4 - 倍精度ベクタ（単項）」*を参照して下さい。

シグナル **NaN**

VFP アーキテクチャに準拠するため、FNEGD はソースレジスタの値がシグナル NaN の場合も例外を生成できません。これは、IEEE754-1985 標準の付録に定義されているよりも厳格な条件です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.44 FNEGS

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	D	1	1	0	0	0	1	Fd	1	0	1	0	0	1	M	0	Fm

FNEGS（浮動小数点符号反転、単精度）命令は、単精度レジスタの値を符号反転し、他の単精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FNEGS{<cond>}  <Sd>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sm> ソースレジスタ。番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = neg(Sm[i])
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

符号反転 関数 neg(x) は、IEEE754-1985 標準の付録で関数 -x として定義されているように、x の符号ビットを反転してコピーする操作を意味します。

Flush-to-Zero モード

FPSCR の FZ ビットは、この命令のオペランドや結果に影響しません。

ベクタ FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、FNEGS は 1 つのマイナス値演算を実行し、vec_len = 1、Sd[0] = Sd、Sm[0] = Sm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FNEGS は複数のマイナス値演算を実行する可能性があります。FNEGS で使用されるレジスタのエンコードと、vec_len、Sd[i]、Sm[i] の決定方法については、P. C5-14 *「アドレシングモード 3 - 単精度ベクタ（単項）」*を参照して下さい。

シグナル NaN

VFP アーキテクチャに準拠するため、FNEGS はソースレジスタの値がシグナル NaN の場合も例外を生成できません。これは、IEEE754-1985 標準の付録に定義されているよりも厳格な条件です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.45 FNMACD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	0	0	0	Dn	Dd	1	0	1	1	0	1	0	0	Dm

FNMACD（浮動小数点符号反転積和、倍精度）命令は、2 つの倍精度レジスタの値を乗算し、その積を符号反転して 3 番目の倍精度レジスタに加算し、結果を 3 番目のレジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FNMACD{<cond>}  <Dd>, <Dn>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。このレジスタは、加算の最初のオペランドとしても使用されます。

<Dn> 乗算の最初のオペランドレジスタ。

<Dm> 乗算の 2 番目のオペランドレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dd[i] + (neg(Dn[i] * Dm[i]))
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合(ベクタ長 1)、FNMACD は 1 つの符号反転積和演算を実行し、vec_len = 1、Dd[0] = Dd、Dn[0] = Dn、Dm[0] = Dm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FNMACD は複数の符号反転積和演算を実行する可能性があります。FNMACD で使用されるレジスタのエンコードと、vec_len、Dd[i]、Dn[i]、Dm[i] の決定方法については、P. C5-18 *「アドレシングモード 4 - 倍精度ベクタ（単項）」*を参照して下さい。

丸め　この演算では、完全な丸めが行われる乗算に続いて、符号ビットの反転と、完全な丸めが行われる加算が実行されます。丸めモードは FPSCR により決定されます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.46 FNMACS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	D	0	0	Fn	Fd	1	0	1	0	N	1	M	0	Fm

FNMACS（浮動小数点符号反転積和、単精度）命令は、2 つの単精度レジスタの値を乗算し、その積の符号反転に 3 番目の単精度レジスタの値を加算して、結果を 3 番目のレジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FNMACS{<cond>}  <Sd>, <Sn>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。このレジスタは、加算の最初のオペランドとしても使用されます。番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sn> 乗算の最初のオペランドレジスタ。番号は、Fn（上位 4 ビット）と N（最下位ビット）としてエンコードされます。

<Sm> 乗算の 2 番目のオペランドレジスタ。番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sd[i] + (neg(Sn[i] * Sm[i]))
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合(ベクタ長 1)、FNMACS は 1 つの符号反転積和演算を実行し、vec_len = 1、Sd[0] = Sd、Sn[0] = Sn、Sm[0] = Sm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FNMACS は複数の符号反転積和演算を実行する可能性があります。FNMACS で使用されるレジスタのエンコードと、vec_len、Sd[i]、Sn[i]、Sm[i] の決定方法については、P. C5-2 *「アドレシングモード 1 - 単精度ベクタ（非単項）」*を参照して下さい。

丸め　この演算では、完全な丸めが行われる乗算に続いて、符号ビットの反転と、完全な丸めが行われる加算が実行されます。丸めモードは FPSCR により決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.47 FNMSCD

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
cond	1	1	1	0	0	0	0	1	Dn	Dd	1	0	1	1	0	1	0	0	Dm

FNMSCD（浮動小数点符号反転積減算、倍精度）命令は、2 つの倍精度レジスタの値を乗算し、その積の符号反転に 3 番目の倍精度レジスタの値の符号反転を加算して、結果を 3 番目のレジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FNMSCD{<cond>}  <Dd>, <Dn>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。このレジスタの値は、符号反転して加算の最初のオペランドとしても使用されます。

<Dn> 乗算の最初のオペランドレジスタ。

<Dm> 乗算の 2 番目のオペランドレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = neg(Dd[i]) + (neg(Dn[i] * Dm[i]))
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合(ベクタ長 1)、FNMSCD は 1 つの符号反転積減算演算を実行し、vec_len = 1、Dd[0] = Dd、Dn[0] = Dn、Dm[0] = Dm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1)、FNMSCD は複数の符号反転積減算演算を実行する可能性があります。FNMSCD で使用されるレジスタのエンコードと、vec_len、Dd[i]、Dn[i]、Dm[i] の決定方法については、P. C5-8 *「アドレシングモード 2 - 倍精度ベクタ（非単項)」*を参照して下さい。

丸め　丸めに関しては、この演算では、完全な丸めが行われる乗算に続いて、符号ビットの反転と、完全な丸めが行われる減算が実行されるのと等価です。丸めモードは FPSCR により決定されます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.48 FNMSCS

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
cond	1	1	1	0	0	D	0	1	Fn	Fd	1	0	1	0	N	1	M	0	Fm

FNMSCS（浮動小数点符号反転積減算、単精度）命令は、2 つの単精度レジスタの値を乗算し、その積の符号反転に 3 番目の倍精度レジスタの値の符号反転を加算して、結果を 3 番目のレジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FNMSCS{<cond>}  <Sd>, <Sn>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。このレジスタの値は、符号反転して加算の最初のオペランドとしても使用されます。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sn> 乗算の最初のオペランドレジスタ。レジスタ番号は、Fn（上位 4 ビット）と N（最下位ビット）としてエンコードされます。

<Sm> 乗算の 2 番目のオペランドレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = neg(Sd[i]) + (neg(Sn[i] * Sm[i]))
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、`FNMSCS` は 1 つの符号反転積減算演算を実行し、`vec_len = 1`、`Sd[0] = Sd`、`Sn[0] = Sn`、`Sm[0] = Sm` です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、`FNMSCS` は複数の符号反転積減算演算を実行する可能性があります。`FNMSCS` で使用されるレジスタのエンコードと、`vec_len`、`Sd[i]`、`Sn[i]`、`Sm[i]` の決定方法については、P. C5-2 *「アドレシングモード 1 - 単精度ベクタ（非単項）」*を参照して下さい。

丸め　丸めに関しては、この演算では、完全な丸めが行われる乗算に続いて、符号ビットの反転と、完全な丸めが行われる減算が実行されるのと等価です。丸めモードは FPSCR により決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.49 FNMULD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	0	1	0	Dn	Dd	1	0	1	1	0	1	0	0	Dm

FNMULD（浮動小数点符号反転乗算、倍精度）命令は、2 つの倍精度レジスタの値を乗算し、結果の符号反転を 3 番目の倍精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FNMULD{<cond>}  <Dd>, <Dn>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Dn> 乗算の最初のオペランドレジスタ。

<Dm> 乗算の 2 番目のオペランドレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = neg(Dn[i] * Dm[i])
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

符号反転 関数 `neg(x)` は、IEEE754-1985 標準の付録で関数 `-x` として定義されているように、`x` の符号ビットを反転してコピーする操作を意味します。

乗算でクワイエット NnN が返される場合、NaN の符号が反転されます。これはデフォルト NaN モードでも同じです。

ベクタ FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、`FNMULD` は 1 つの符号反転乗算を実行し、`vec_len = 1`、`Dd[0] = Dd`、`Dn[0] = Dn`、`Dm[0] = Dm` です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、`FNMULD` は複数の符号反転乗算を実行する可能性があります。`FNMULD` で使用されるレジスタのエンコードと、`vec_len`、`Dd[i]`、`Dn[i]`、`Dm[i]` の決定方法については、P. C5-8 *「アドレシングモード 2 - 倍精度ベクタ（非単項）」*を参照して下さい。

丸め この演算では完全な丸めが行われる乗算に続いて、結果の符合ビットが反転されます。丸めモードは FPSCR によって決定されます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.50 FNMULS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	D	1	0	Fn	Fd	1	0	1	0	N	1	M	0	Fm

FNMULS（浮動小数点符号反転乗算、単精度）命令は、2つの単精度レジスタの値を乗算し、結果の符号反転を3番目の単精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FNMULS{<cond>}  <Sd>, <Sn>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond>が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。レジスタ番号は、Fd（上位4ビット）とD（最下位ビット）としてエンコードされます。

<Sn> 乗算の最初のオペランドレジスタ。レジスタ番号は、Fn（上位4ビット）とN（最下位ビット）としてエンコードされます。

<Sm> 乗算の2番目のオペランドレジスタ。レジスタ番号は、Fm（上位4ビット）とM（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、アンダーフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = neg(Sn[i] * Sm[i])
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

符号反転 関数 neg(x) は、IEEE754-1985 標準の付録で関数 -x として定義されているように、x の符号ビットを反転してコピーする操作を意味します。

乗算でクワイエット NnN が返される場合、NaN の符号が反転されます。これはデフォルト NaN モードでも同じです。

ベクタ FPSCR の LEN フィールドでスカラモードが指定されている場合(ベクタ長 1)、FNMULS は 1 つの符号反転乗算を実行し、vec_len = 1、Sd[0] = Sd、Sn[0] = Sn、Sm[0] = Sm です。

LEN フィールドでベクタモードが指定されている場合(ベクタ長 > 1)、FNMULS は複数の符号反転乗算を実行する可能性があります。FNMULS で使用されるレジスタのエンコードと、vec_len、Sd[i]、Sn[i]、Sm[i] の決定方法については、P. C5-2 *「アドレシングモード 1 - 単精度ベクタ(非単項)」*を参照して下さい。

丸め この演算では完全な丸めが行われる乗算に続いて、結果の符合ビットが反転されます。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.51 FSITOD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	0	1	1	1 0 0 0	Dd	1	0	1	1	1	1	M	0	Fm

FSITOD（符号付き整数から浮動小数点媒性倍精度への変換）命令は、単精度レジスタにある符号付き整数値を倍精度に変換し、結果を倍精度レジスタに書き込みます。整数値は、通常はメモリから単精度ロード命令で、または ARM レジスタから FMSR 命令で単精度レジスタに転送されます。

構文

```
FSITOD{<cond>}  <Dd>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Sm> ソースレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Dd = ConvertSignedIntegerToDouble(Sm)
```

注

ベクタ FSITOD は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

0 Smの値が整数値の0の場合、結果は倍精度の+0.0になります。倍精度の-0.0にはなりません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.52 FSITOS

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	D	1	1	1	0	0	0	Fd	1	0	1	0	1	1	M	0	Fm

FSITOS（符号付き整数から浮動小数点単精度への変換）命令は、単精度レジスタにある符号付き整数値を単精度に変換し、結果を単精度レジスタに書き込みます。整数値は、通常はメモリから単精度ロード命令で、または ARM レジスタから FMSR 命令で単精度レジスタに転送されます。

構文

```
FSITOS{<cond>}  <Sd>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sm> ソースレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 不正確

動作

```
if ConditionPassed(cond) then
    Sd = ConvertSignedIntegerToSingle(Sm)
```

注

ベクタ FSITOS は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

0 Smの値が整数値の0の場合、結果は単精度の +0.0 になります。単精度の -0.0 にはなりません。

丸め 一部の大きなオペランド値について、丸めが必要になります。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.53 FSQRTD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	0	1	1	0 0 0 1	Dd	1	0	1	1	1	1	0	0	Dm

FSQRTD（浮動小数点平方根、倍精度）命令は、倍精度レジスタの値の平方根を計算し、結果を他の倍精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FSQRTD{<cond>}  <Dd>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Dm> ソースレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = sqrt(Dm[i])
```

用法

平方根演算には、ほとんどの実装で多くのサイクルが必要であり、ベクタ平方根演算は比例的に必要サイクル数が多くなります。これによってパフォーマンスが大きく低下する可能性があるため、できる限り大量の平方根演算は避ける必要があります。

割り込みレイテンシの一部の考慮点について、P. C1-8 *「割り込み」* も参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合(ベクタ長 1)、FSQRTD は 1 つの平方根演算を実行し、vec_len = 1、Dd[0] = Dd、Dm[0] = Dm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FSQRTD は複数の平方根演算を実行する可能性があります。FSQRTD で使用されるレジスタのエンコードと、vec_len、Dd[i]、Dm[i] の決定方法については、P. C5-18 *「アドレシングモード 4 - 倍精度ベクタ（単項）」*を参照して下さい。

丸め　この演算は完全な丸めが行われる平方根演算です。丸めモードは FPSCR によって決定されます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.54 FSQRTS

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	D	1	1	0	0	0	1	Fd	1	0	1	0	1	1	M	0	Fm

FSQRTS（浮動小数点平方根、単精度）命令は、単精度レジスタの値の平方根を計算し、結果を他の単精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FSQRTS{<cond>}  <Sd>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sm> ソースレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = sqrt(Sm[i])
```

用法

平方根演算には、ほとんどの実装で多くのサイクルが必要であり、ベクタ平方根演算は比例的に必要サイクル数が多くなります。これによってパフォーマンスが大きく低下する可能性があるため、できる限り大量の平方根演算は避ける必要があります。

割り込みレイテンシの一部の考慮点について、P. C1-8 *「割り込み」* も参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合(ベクタ長 1)、`FSQRTS` は 1 つの平方根演算を実行し、`vec_len = 1`、`Sd[0] = Sd`、`Sm[0] = Sm` です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1)、`FSQRTS` は複数の平方根演算を実行する可能性があります。`FSQRTS` で使用されるレジスタのエンコードと、`vec_len`、`Sd[i]`、`Sm[i]` の決定方法については、P. C5-14 *「アドレシングモード 3 - 単精度ベクタ（単項)」*を参照して下さい。

丸め　この演算は完全な丸めが行われる平方根演算です。丸めモードは FPSCR により決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.55 FSTD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
cond	1	1	0	1	U	0	0	0	Rn	Dd	1	0	1	1	offset

FSTD（浮動小数点ストア、倍精度）命令は、倍精度レジスタの値をメモリにストアします。

構文

```
FSTD{<cond>}  <Dd>, [<Rn>{, #+/-(<offset>*4)}]
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> ソースレジスタ。

<Rn> 転送のベースアドレスを保持している ARM レジスタ。

<offset> offset に 4 を掛け、ベースアドレスを加算（U == 1 の場合）または減算（U == 0 の場合）することで、転送の実際のアドレスが計算されます。offset が省略されている場合、デフォルトは +0 です。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (U == 1)
        address = Rn + offset * 4
    else
        address = Rn - offset * 4
    if (big-endian)
        Memory[address,4] = Dd[63:32]
        Memory[address+4,4] = Dd[31:0]
    else
        Memory[address,4] = Dd[31:0]
        Memory[address+4,4] = Dd[63:32]
    if (Shared(address))
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,4)
        /* See P. A2-49「操作の概要」*/
```

注

アドレシングモード

これは、P. C5-22「*アドレシングモード5 - VFP 複数ロード/ストア*」の特別なケースです。

変換 実装で倍精度値について内部形式を使用している場合、変換により外部の倍精度形式に戻す必要があります。それ以外の場合、変換は必要ありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.56 FSTMD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
cond	1	1	0	P	U	0	W	0	Rn	Dd	1	0	1	1	offset

FSTMD（浮動小数点複数ストア、倍精度）命令は、連続した複数の倍精度レジスタの値をメモリにストアします。

構文

```
FSTM<addressing_mode>D{<cond>}  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<addressing_mode>
命令によって使用される start_address と end_address の値を決定するアドレシングモードを指定します。詳細については、P. C5-22 *「アドレシングモード 5 - VFP 複数ロード / ストア」* を参照して下さい。

<Rn> <addressing_mode> で使用するベースレジスタを指定します。

! 命令の W ビットを 1 に設定し、ベースレジスタ <Rn> が命令によって更新されることを示します。この値が省略されている場合、命令の W ビットは 0 に設定され、ベースレジスタ <Rn> は変更されません。<addressing_mode> と ! の存在 / 非存在の組み合わせの一部は許容されません。詳細については、*アドレシングモード 5 - VFP 複数ロード / ストア* : P. C5-22 を参照して下さい。

<registers>
ストアされるレジスタを、カンマで区切られ角かっこで囲まれた、連続した番号を持つ倍精度レジスタの一覧として指定します。レジスタ番号は、Dd を一覧の最初のレジスタの番号、offset を一覧に含まれるレジスタ番号の 2 倍に設定することで、命令にエンコードされます。一覧には、最低 1 つのレジスタを指定する必要があります。

たとえば、<registers> が {D2,D3,D4} の場合、命令の Dd フィールドは 2、offset フィールドは 6 になります。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to (offset-2)/2
        /* d is the number of register Dd;                  */
        /* D(n) is the double-precision register numbered n */
        if (big-endian)
            Memory[address,4] = D(d+i)[63:32]
            Memory[address+4,4] = D(d+i)[31:0]
        else
            Memory[address,4] = D(d+i)[31:0]
            Memory[address+4,4] = D(d+i)[63:32]
        if (Shared(address))
            physical_address = TLB(address)
            ClearExclusiveByAddress(physical_address,processor_id,size)
        if (Shared(address+4))
            physical_address = TLB(address+4)
            ClearExclusiveByAddress(physical_address,processor_id,size)
                /* See P. A2-49「操作の概要」*/
        address = address + 8
    assert end_address = address - 4
```

注

エンコード　P = 1 かつ W = 0 の場合、命令は FSTD となります。それ以外の場合、offset が奇数であれば命令は FSTMX となります。

ベクタ　FSTMD命令はFPSCRのLENおよびSTRIDEフィールドの影響を受けず、データ処理命令のベクタオペランドのようにバンク境界でラップアラウンドすることはありません。レジスタは単純に、レジスタ番号の昇順にストアされます。

無効なレジスタ指定

Dd と offset が有効なレジスタ一覧を指定していない場合、命令の動作は予測不能です。これは次の 2 つの条件で発生することがあります。

- offset == 0 の場合、つまりどのレジスタも転送しない命令が指定された場合
- d + offset / 2 > 16 の場合、つまり D15 よりも後のレジスタを転送しようとした場合

変換　実装で倍精度値について内部形式を使用している場合、変換により外部の倍精度形式に戻す必要があります。それ以外の場合、変換は必要ありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.57 FSTMS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
cond	1	1	0	P	U	D	W	0	Rn	Fd	1	0	1	0	offset

FSTMS（浮動小数点多重ストア、単精度）命令は、連続した複数の単精度レジスタの値をメモリにストアします。

構文

```
FSTM<addressing_mode>S{<cond>}  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<addressing_mode>

命令によって使用される start_address と end_address の値を決定するアドレシングモードを指定します。詳細については、P. C5-22 *「アドレシングモード 5 - VFP 複数ロード / ストア」* を参照して下さい。

<Rn> <addressing_mode> で使用するベースレジスタを指定します。

! 命令の W ビットを 1 に設定し、ベースレジスタ <Rn> が命令によって更新されることを示します。この値が省略されている場合、命令の W ビットは 0 に設定され、ベースレジスタ <Rn> は変更されません。<addressing_mode> と ! の存在 / 非存在の組み合わせの一部は許容されません。詳細については、P. C5-22 *「アドレシングモード 5 - VFP 複数ロード / ストア」* を参照して下さい。

<registers>

ストアされるレジスタを、カンマで区切られ角かっこで囲まれた、連続した番号を持つ単精度レジスタの一覧として指定します。d が一覧の最初のレジスタの場合、一覧は Fd と D をそれぞれ d の上位 4 ビットと最下位ビットに設定し、offset を一覧のレジスタ数として、命令にエンコードされます。一覧には、最低 1 つのレジスタを指定する必要があります。

たとえば、<registers> が {S5,S6,S7} の場合、命令の Fd フィールドは 0b0010、D ビットは 1、offset フィールドは 3 になります。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to offset-1
        /* d is as defined for <registers> above;           */
        /* S(n) is the single-precision register numbered n */
        Memory[address,4] = S(d+i)
        address = address + 4
        if (Shared(address))
            physical_address = TLB(address)
            ClearExclusiveByAddress(physical_address,processor_id,4)
            /* See P. A2-49「操作の概要」*/
    assert end_address = address - 4
```

注

エンコード　P = 1 かつ W = 0 の場合、命令は FSTS となります。

ベクタ　FSTMS命令はFPSCRのLENおよびSTRIDEフィールドの影響を受けず、データ処理命令のベクタオペランドのようにバンク境界でラップアラウンドすることはありません。レジスタは単純に、レジスタ番号の昇順にストアされます。

無効なレジスタ指定

Fd、Dd、offset が有効なレジスタ一覧を指定していない場合、命令の動作は予測不能です。これは次の 2 つの条件で発生することがあります。

- offset == 0 の場合、つまりどのレジスタも転送しない命令が指定された場合
- d + offset > 32 の場合、つまり S31 よりも後のレジスタを転送しようとした場合

変換　プログラマモデルでは、FSTMS は転送される値に対して変換を実行しません。ソースレジスタには、整数と単精度浮動小数点数のどちらも保持可能です。整数値は、一般に浮動小数点から整数への変換命令の結果として得られます。

実装では、FSTMS によって内部形式を外部形式に変換でき、レジスタに含まれている値が単精度浮動小数点数か整数かにかかわらず正しいデータを回復できる限り、ソースレジスタの値を自由に内部形式で保持できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.58 FSTMX

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
cond	1	1	0	P	U	0	W	0	Rn	Dd	1	0	1	1	offset

FSTMX（浮動小数点複数ストア、精度不明）命令は、連続した複数の倍精度レジスタの値をメモリにストアします。これによって、レジスタに整数、単精度値、倍精度値のいずれが含まれているかにかかわらず、値を正しく再ロードできます。

――― 注 ―――

FSTMX 命令は、ARMv6 では推奨されません。データの精度が不明な場合、値の保存と復元には FSTMD を使用して下さい。

―――――――

構文

```
FSTM<addressing_mode>X{<cond>}  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<addressing_mode>

命令によって使用される start_address と end_address の値を決定するアドレシングモードを指定します。詳細については、P. C5-22「*アドレシングモード 5 - VFP 複数ロード / ストア*」を参照して下さい。

<Rn> <addressing_mode> で使用するベースレジスタを指定します。

! 命令の W ビットを 1 に設定し、ベースレジスタ <Rn> が命令によって更新されることを示します。この値が省略されている場合、命令の W ビットは 0 に設定され、ベースレジスタ <Rn> は変更されません。<addressing_mode> と ! の存在 / 非存在の組み合わせの一部は許容されません。詳細については、P. C5-22「*アドレシングモード 5 - VFP 複数ロード / ストア*」を参照して下さい。

<registers>

ストアされるレジスタを、カンマで区切られ角かっこで囲まれた、連続した番号を持つ倍精度レジスタの一覧として指定します。レジスタ番号は、Dd を一覧の最初のレジスタの番号、offset を一覧に含まれるレジスタ番号の 2 倍 + 1 に設定することで、命令にエンコードされます。一覧には、最低 1 つのレジスタを指定する必要があります。

たとえば、<registers> が {D2,D3,D4} の場合、命令の Dd フィールドは 2、offset フィールドは 7 になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to (offset-3)/2
        /* d is the number of register Dd;                  */
        /* D(n) is the double-precision register numbered n */
        if (big-endian)
            Memory[address,4] = D(d+i)[63:32]
            Memory[address+4,4] = D(d+i)[31:0]
        else
            Memory[address,4] = D(d+i)[31:0]
            Memory[address+4,4] = D(d+i)[63:32]
        if (Shared(address))
            physical_address = TLB(address)
            ClearExclusiveByAddress(physical_address,processor_id,4)
        if (Shared(address+4))
            physical_address = TLB(address+4)
            ClearExclusiveByAddress(physical_address,processor_id,4)
                /* See P. A2-49「操作の概要」*/
        address = address + 8
    assert end_address = address - 4
```

用法

FSTMX は、VFP レジスタに含まれているデータのタイプが不明な場合、そのレジスタの値をメモリに保存するために使用されます。一般に次のような場合に使用されます。

- 標準のプロシージャコールでのプロシージャ起動時のデータ保存シーケンス
- プロセススワップコード

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

エンコード　P = 1 かつ W = 0 の場合、命令は FSTD となります。それ以外の場合、offset が偶数であれば命令は FSTMD となります。

ベクタ　FSTMX命令はFPSCRのLENおよびSTRIDEフィールドの影響を受けず、データ処理命令のベクタオペランドのようにバンク境界でラップアラウンドすることはありません。レジスタは単純に、レジスタ番号の昇順にストアされます。

無効なレジスタ指定

Dd と offset が有効なレジスタ一覧を指定していない場合、命令の動作は予測不能です。これは次の 2 つの条件で発生することがあります。

- offset == 0 の場合、つまりどのレジスタも転送しない命令が指定された場合
- d + (offset - 1) / 2 > 16 の場合、つまり D15 よりも後のレジスタを転送しようとした場合

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.59 FSTS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
cond	1	1	0	1	U	D	0	0	Rn	Fd	1	0	1	0	offset

FSTS（浮動小数点ストア、単精度）命令は、単精度レジスタの値をメモリにストアします。

構文

```
FSTS{<cond>}  <Sd>, [<Rn>{, #+/-(<offset>*4)}]
```

各項目の説明については以下を参照して下さい。

`<cond>` この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。`<cond>` が省略されている場合、AL（常時）条件が使用されます。

`<Sd>` ソースレジスタ。番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

`<Rn>` 転送のベースアドレスを保持している ARM レジスタ。

`<offset>` offset に 4 を掛け、ベースアドレスを加算（U == 1 の場合）または減算（U == 0 の場合）することで、転送の実際のアドレスが計算されます。このオフセットが省略されている場合、デフォルトは +0 です。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

データアボート

動作

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (U == 1)
        address = Rn + offset * 4
    else
        address = Rn - offset * 4
    Memory[address,4] = Sd
    if (Shared(address))
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,size)
        /* See P. A2-49「操作の概要」*/
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

アドレシングモード

これは、P. C5-22「*アドレシングモード5 - VFP 複数ロード/ストア*」の特別なケースです。

変換

プログラマモデルでは、`FSTS` は転送される値に対して変換を実行しません。ソースレジスタ `Sd` には、整数と単精度浮動小数点数値のどちらも保持可能です。整数値は、一般に浮動小数点から整数への変換命令の結果として得られます。

実装では、`FSTS` によって内部形式を外部形式に変換でき、`Sd` に含まれている値が単精度浮動小数点数値か整数かにかかわらず正しいデータを回復できる限り、`Sd` の値を自由に内部形式で保持できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.60 FSUBD

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	0	1	1	Dn	Dd	1	0	1	1	0	1	0	0	Dm

FSUBD（浮動小数点減算、倍精度）命令は、倍精度レジスタの値を別の倍精度レジスタの値から減算し、結果を 3 番目の倍精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FSUBD{<cond>}  <Dd>, <Dn>, <Dm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Dn> 減算の最初のオペランドレジスタ。

<Dm> 減算の 2 番目のオペランドレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dn[i] - Dm[i]
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、FSUBD は 1 つの減算を実行し、vec_len = 1、Dd[0] = Dd、Dn[0] = Dn、Dm[0] = Dm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FSUBD は複数の減算を実行する可能性があります。FSUBD で使用されるレジスタのエンコードと、vec_len、Dd[i]、Dn[i]、Dm[i] の決定方法については、P. C5-8 *「アドレシングモード 2 - 倍精度ベクタ（非単項）」*を参照して下さい。

丸め　この演算は完全な丸めが行われる減算です。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.61 FSUBS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	0	D	1	1	Fn	Fd	1	0	1	0	N	1	M	0	Fm

FSUBS（浮動小数点減算、単精度）命令は、単精度レジスタの値を別の単精度レジスタの値から減算し、結果を 3 番目の単精度レジスタに書き込みます。ベクタに対しても同じ演算を実行できます。

構文

```
FSUBS{<cond>}  <Sd>, <Sn>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sn> 減算の最初のオペランドレジスタ。レジスタ番号は、Fn（上位 4 ビット）と N（最下位ビット）としてエンコードされます。

<Sm> 減算の 2 番目のオペランドレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、オーバフロー、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sn[i] - Sm[i]
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FPSCR の LEN フィールドでスカラモードが指定されている場合（ベクタ長 1）、FSUBS は 1 つの減算を実行し、vec_len = 1、Sd[0] = Sd、Sn[0] = Sn、Sm[0] = Sm です。

LEN フィールドでベクタモードが指定されている場合（ベクタ長 > 1）、FSUBS は複数の減算を実行する可能性があります。FSUBS で使用されるレジスタのエンコードと、vec_len、Sd[i]、Sn[i]、Sm[i] の決定方法については、P. C5-2 *「アドレシングモード 1 - 単精度ベクタ（非単項）」*を参照して下さい。

丸め　この演算は完全な丸めが行われる減算です。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.62 FTOSID

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	D	1	1	1	1	0	1	Fd	1	0	1	1	Z	1	0	0	Dm

FTOSID（浮動小数点の倍精度から符号付き整数への変換）命令は、倍精度レジスタにある値を符号付き整数に変換し、結果を単精度レジスタに書き込みます。整数値は、通常はその後で単精度ストア命令でメモリへ、または FMRS 命令で ARM レジスタへ転送されます。

構文

```
FTOSI{Z}D{<cond>}  <Sd>, <Dm>
```

各項目の説明については以下を参照して下さい。

Z　　命令の Z ビットを 1 に設定し、動作で*ゼロへの*丸めモードを使用することを示します。Zが指定されていない場合、命令のZビットは0になり、動作ではFPSCRで指定されている丸めモードが使用されます。

<cond>　　この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd>　　デスティネーションレジスタ。番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Dm>　　ソースレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    Sd = ConvertDoubleToSignedInteger(Dm)
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　　`FTOSID` は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

範囲外の値　　オペランドが $-\infty$（マイナス無限大）の場合、または丸め後の結果が -2^{31} より小さい場合、無効演算例外が発生します。この例外がトラップされない場合、結果は `0x80000000` になります。

オペランドが $+\infty$（プラス無限大）の場合、または丸め後の結果が $2^{31}-1$ より大きい場合、無効演算例外が発生します。この例外がトラップされない場合、結果は `0x7FFFFFFF` になります。

オペランドが NaN の場合、無効演算例外が発生します。この例外がトラップされない場合、結果は `0x00000000` になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.63 FTOSIS

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	D	1	1	1 1 0 1	Fd	1	0	1	0	Z	1	M	0	Fm

FTOSIS（浮動小数点の単精度から符号付き整数への変換）命令は、単精度レジスタにある値を符号付き整数に変換し、結果を単精度レジスタに書き込みます。整数値は、通常はその後で単精度ストア命令でメモリへ、または FMRS 命令で ARM レジスタへ転送されます。

構文

```
FTOSI{Z}S{<cond>}  <Sd>, <Sm>
```

各項目の説明については以下を参照して下さい。

Z　命令の Z ビットを 1 に設定し、動作で*ゼロへの*丸めモードを使用することを示します。Zが指定されていない場合、命令のZビットは0になり、動作ではFPSCRで指定されている丸めモードが使用されます。

<cond>　この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd>　デスティネーションレジスタ。番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sm>　ソースレジスタ。番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    Sd = ConvertSingleToSignedInteger(Sm)
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FTOSIS は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

範囲外の値　オペランドが –∞（マイナス無限大）の場合、または丸め後の結果が -2^{31} より小さい場合、無効演算例外が発生します。この例外がトラップされない場合、結果は 0x80000000 になります。

オペランドが +∞（プラス無限大）の場合、または丸め後の結果が $2^{31}-1$ より大きい場合、無効演算例外が発生します。この例外がトラップされない場合、結果は 0x7FFFFFFF になります。

オペランドが NaN の場合、無効演算例外が発生します。この例外がトラップされない場合、結果は 0x00000000 になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.64 FTOUID

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	D	1	1	1	1	0	0	Fd	1	0	1	1	Z	1	0	0	Dm

FTOUID（浮動小数点の倍精度から符号なし整数への変換）命令は、倍精度レジスタにある値を符号なし整数に変換し、結果を単精度レジスタに書き込みます。整数値は、通常はその後で単精度ストア命令でメモリへ、または FMRS 命令で ARM レジスタへ転送されます。

構文

```
FTOUI{Z}D{<cond>}  <Sd>, <Dm>
```

各項目の説明については以下を参照して下さい。

Z　命令の Z ビットを 1 に設定し、動作で*ゼロへの*丸めモードを使用することを示します。Zが指定されていない場合、命令のZビットは0になり、動作ではFPSCRで指定されている丸めモードが使用されます。

<cond>　この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd>　デスティネーションレジスタ。番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Dm>　ソースレジスタ。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

浮動小数点例外 : 無効演算、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    Sd = ConvertDoubleToUnsignedInteger(Dm)
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FTOUID は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

範囲外の値　オペランドが –∞（マイナス無限大）の場合、または丸め後の結果が 0 より小さい場合、無効演算例外が発生します。この例外がトラップされない場合、結果は 0x00000000 になります。

オペランドが +∞（プラス無限大）の場合、または丸め後の結果が 2^{32}–1 より大きい場合、無効演算例外が発生します。この例外がトラップされない場合、結果は 0xFFFFFFFF になります。

オペランドが NaN の場合、無効演算例外が発生します。この例外がトラップされない場合、結果は 0x00000000 になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.65 FTOUIS

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	D	1	1	1	1	0	0	Fd	1	0	1	0	Z	1	M	0	Fm

FTOUIS（浮動小数点の単精度から符号なし整数への変換）命令は、単精度レジスタにある値を符号なし整数に変換し、結果を単精度レジスタに書き込みます。整数値は、通常はその後で単精度ストア命令でメモリへ、または FMRS 命令で ARM レジスタへ転送されます。

構文

```
FTOUI{Z}S{<cond>}  <Sd>, <Sm>
```

各項目の説明については以下を参照して下さい。

Z　　命令の Z ビットを 1 に設定し、動作で*ゼロへの*丸めモードを使用することを示します。Zが指定されていない場合、命令のZビットは0になり、動作ではFPSCRで指定されている丸めモードが使用されます。

<cond>　　この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd>　　デスティネーションレジスタ。番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sm>　　ソースレジスタ。番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 無効演算、不正確、入力非正規

動作

```
if ConditionPassed(cond) then
    Sd = ConvertSingleToUnsignedInteger(Sm)
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

注

ベクタ　FTOUIS は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

範囲外の値　オペランドが –∞（マイナス無限大）の場合、または丸め後の結果が 0 より小さい場合、無効演算例外が発生します。この例外がトラップされない場合、結果は 0x00000000 になります。

オペランドが +∞（プラス無限大）の場合、または丸め後の結果が 2^{32}–1 より大きい場合、無効演算例外が発生します。この例外がトラップされない場合、結果は 0xFFFFFFFF になります。

オペランドが NaN の場合、無効演算例外が発生します。この例外がトラップされない場合、結果は 0x00000000 になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.66 FUITOD

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	0	1	1	1	0	0	0	Dd	1	0	1	1	0	1	M	0	Fm

FUITOD（符号なし整数から浮動小数点倍精度への変換）命令は、単精度レジスタにある符号なし整数値を倍精度に変換し、結果を倍精度レジスタに書き込みます。整数値は、通常はメモリから単精度ロード命令で、または ARM レジスタから FMSR 命令で単精度レジスタに転送されます。

構文

```
FUITOD{<cond>}  <Dd>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3*「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Dd> デスティネーションレジスタ。

<Sm> ソースレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

D バリエーションにのみ存在します。

例外

なし

動作

```
if ConditionPassed(cond) then
    Dd = ConvertUnsignedIntegerToDouble(Sm)
```

注

ベクタ FUITOD は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

0 Smの値が整数値の0の場合、結果は倍精度の +0.0 になります。倍精度の -0.0 にはなりません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C4.1.67 FUITOS

31–28	27	26	25	24	23	22	21	20	19–16				15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	D	1	1	1	0	0	0	Fd	1	0	1	0	0	1	M	0	Fm

FUITOS（符号なし整数から浮動小数点単精度への変換）命令は、単精度レジスタにある符号なし整数値を単精度に変換し、結果を単精度レジスタに書き込みます。整数値は、通常はメモリから単精度ロード命令で、または ARM レジスタから FMSR 命令で単精度レジスタに転送されます。

構文

```
FUITOS{<cond>}  <Sd>, <Sm>
```

各項目の説明については以下を参照して下さい。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」*で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Sd> デスティネーションレジスタ。レジスタ番号は、Fd（上位 4 ビット）と D（最下位ビット）としてエンコードされます。

<Sm> ソースレジスタ。レジスタ番号は、Fm（上位 4 ビット）と M（最下位ビット）としてエンコードされます。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

例外

浮動小数点例外 : 不正確

動作

```
if ConditionPassed(cond) then
    Sd = ConvertUnsignedIntegerToSingle(Sm)
```

注

ベクタ FUITOS は FPSCR の LEN フィールドに関係なく、常にスカラとして動作します。

0 Smの値が整数値の0の場合、結果は単精度の +0.0 になります。単精度の -0.0 にはなりません。

丸め 一部の大きなオペランド値について、丸めが必要になります。丸めモードは FPSCR によって決定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 C5 章
VFP のアドレシングモード

本章では、VFP の 5 つのアドレシングモードについて、それぞれの構文と使用法を解説します。本章は以下のセクションから構成されています。

- *アドレシングモード 1 - 単精度ベクタ（非単項）*: P. C5-2
- *アドレシングモード 2 - 倍精度ベクタ（非単項）*: P. C5-8
- *アドレシングモード 3 - 単精度ベクタ（単項）*: P. C5-14
- *アドレシングモード 4 - 倍精度ベクタ（単項）*: P. C5-18
- *アドレシングモード 5 - VFP 複数ロード / ストア*: P. C5-22

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.1 アドレシングモード 1 - 単精度ベクタ（非単項）

31-28	27	26	25	24	23	22	21-20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
cond	1	1	1	0	Op	D	Op	Fn	Fd	1	0	1	0	N	Op	M	0	Fm

FPSCR で示されているベクタ長が 1 より大きい場合、単精度の 2 オペランド命令（FADDS、FDIVS、FMULS、FNMULS、FSUBS）では 3 つの異なる動作を指定できます。

- 2 つのスカラ値に対して 1 つの演算を実行し、結果はスカラになる。

  ScalarA op ScalarB → ScalarD

  この方式が選択されている場合（P. C5-5「*スカラ演算*」参照）、FPSCR で指定されているベクタ長は無視され、実行される演算は 1 つだけです。これによって、FPSCR を再プログラムする必要なしに、スカラ演算とベクタ演算の混在が可能です。

- N 個の数値演算の組。N は FPSCR により指定されているベクタ長で、最初のオペランドはベクタ内でスキャンされ、2 番目のオペランドは一定で、デスティネーションはベクタ内でスキャンされます。

  ```
  VectorA[0] op ScalarB → VectorD[0]
  VectorA[1] op ScalarB → VectorD[1]
  ...
  VectorA[N-1] op ScalarB → VectorD[N-1]
  ```

  これは、次のように短縮できます。

  VectorA op ScalarB → VectorD

- N 個の数値演算の組。N は FPSCR により指定されているベクタ長で、両方のオペランドとデスティネーションがベクタ内でスキャンされます。

  ```
  VectorA[0] op VectorB[0] → VectorD[0]
  VectorA[1] op VectorB[1] → VectorD[1]
  ...
  VectorA[N-1] op VectorB[N-1] → VectorD[N-1]
  ```

  これは、次のように短縮できます。

  VectorA op VectorB → VectorD

単精度の 3 オペランド命令（FMACS、FMSCS、FNMACS、FNMSCS）では、加算 / 減算オペランドとデスティネーションに同じレジスタを使用できます。このため、上の 3 つに対応する 3 つの形式が存在します。

- 純粋なスカラ形式

  ± (ScalarA * ScalarB) ± ScalarD → ScalarD

- 乗算の 2 番目のオペランドがスカラで、他のすべてがベクタ内でスキャンされる形式

  ```
  ± (VectorA[0]   * ScalarB) ± VectorD[0] → VectorD[0]
  ± (VectorA[1]   * ScalarB) ± VectorD[1] → VectorD[1]
  ...
  ± (VectorA[N-1] * ScalarB) ± VectorD[N-1] → VectorD[N-1]
  ```

  これは、次のように短縮できます。

  ± (VectorA * ScalarB) ± VectorD → VectorD

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- すべてのオペランドがベクタ内でスキャンされる形式

  ```
  ±（VectorA[0] * VectorB[0]）± VectorD[0] → VectorD[0]
  ±（VectorA[1] * VectorB[1]）± VectorD[1] → VectorD[1]
  ...
  ±（VectorA[N-1] * VectorB[N-1]）± VectorD[N-1] → VectorD[N-1]
  ```

  これは、次のように短縮できます。

  ```
  ±（VectorA * VectorB）± VectorD → VectorD
  ```

C5.1.1 レジスタバンク

これら各種の形式を指定できるようにするため、32 個の単精度レジスタの組は、それぞれ 8 つのレジスタで構成される 4 つのバンクに分割されます。命令によって使用される形式は、最初のバンクにあるオペランドに依存します。規則の根底となる一般原則は、スカラオペランドの保持には最初のバンクを使用する必要があり、他のバンクはベクタオペランドの保持に使用されることです。すべてのデスティネーションレジスタへの書き込みと、多くのソースレジスタの読み出しはこの原則に従っていますが、一部のソースレジスタ読み出しはベクタエレメントへのスカラアクセスや、スカラのグループへのベクタアクセスを行うことがあります。

ベクタオペランドは、単一のバンクに含まれる 2 ～ 8 個のレジスタで構成され、レジスタの数は FPSCR のベクタ長フィールドによって指定されます。詳細については、P. C2-24 *「ベクタの長さ / ストライドの制御」*を参照して下さい。命令に含まれるレジスタ番号は、ベクタの最初のエレメントを含むレジスタを指定します。ベクタ内の以降の各エレメントは、FPSCR のベクタストライドフィールドで指定された値だけレジスタ番号をインクリメントして得られます。これによってレジスタ番号がレジスタバンクの最上位からオーバフローする場合、図 C5-1 に示すように、レジスタ番号はバンクの最下位にラップアラウンドします。

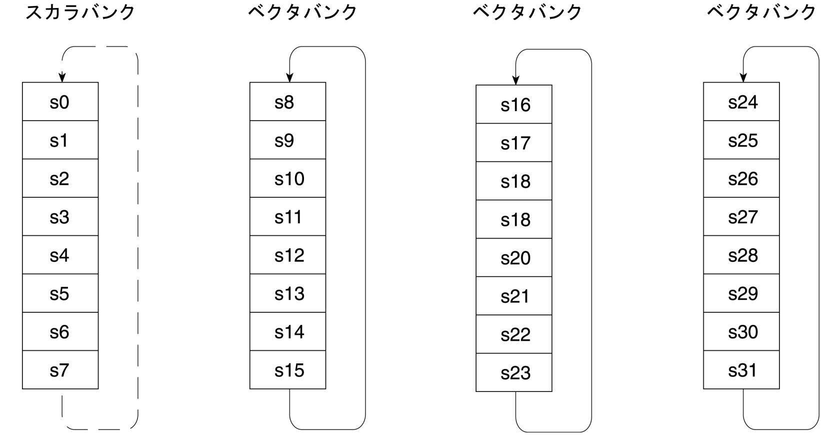


図 C5-1 単精度レジスタのバンク

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.1.2 動作

以下のセクションでは、可能な 3 つのアドレシングモードの各形式について解説します。

- *スカラ演算* : P. C5-5
- *ベクタとスカラの混在した演算* : P. C5-6
- *ベクタ操作* : P. C5-7

どの場合も、以下の値が生成されます。

`vec_len` 命令によって指定される個別の演算の数。

`Sd[0] ... Sd[vec_len-1]`
個別の演算のデスティネーションレジスタ。

`Sn[0] ... Sn[vec_len-1]`
個別の演算の最初のソースレジスタ。

`Sm[0] ... Sm[vec_len-1]`
個別の演算の 2 番目のソースレジスタ。

どの場合も、命令によって指定されるレジスタは命令の Fd、Fn、Fm フィールドを、それぞれ D、N、M ビットと連結することで決定されます。

```
d_num = (Fd << 1) | D
n_num = (Fn << 1) | N
m_num = (Fm << 1) | M
```

次に、これらのレジスタ番号は次のようにバンク番号と、バンク内のインデクスに分解されます。

```
d_bank  = d_num[4:3]
d_index = d_num[2:0]

n_bank  = n_num[4:3]
n_index = n_num[2:0]

m_bank  = m_num[4:3]
m_index = m_num[2:0]
```

——— 注 ———

FPSCR でベクタ長に 1 が指定されるのは特別なケースではありません。アドレシングモードの 3 つの形式すべての規則は、ベクタ長が 1 の場合には次のように簡素化されます。

```
vec_len = 1
Sd[0] = d_num
Sn[0] = n_num
Sm[0] = m_num
```


Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.1.3 スカラ演算

デスティネーションレジスタが最初のバンクに含まれる 8 つのレジスタ内にある場合、その命令についてスカラ演算が指定されます。

```
if d_bank == 0 then
    vec_len = 1
    Sd[0] = d_num
    Sn[0] = n_num
    Sm[0] = m_num
```

注

ソースオペランド ソースオペランドは、どのバンクにあるかにかかわらず常にスカラです。これによって、ベクタの個別のエレメントをスカラとして使用できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.1.4 ベクタとスカラの混在した演算

命令で指定されたデスティネーションレジスタが最初のバンクに含まれる8つのレジスタ内ではなく、2 番目のソースレジスタが最初のバンクに含まれている場合、デスティネーションレジスタと最初のソースレジスタはベクタを、2 番目のソースレジスタはスカラを指定します。

```
if d_bank != 0 and m_bank == 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Sd[i] = (d_bank << 3) | d_index
        Sn[i] = (n_bank << 3) | n_index
        Sm[i] = m_num
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 7 then
            d_index = d_index - 8
        n_index = n_index + (vector stride specified by FPSCR)
        if n_index > 7 then
            n_index = n_index - 8
```

注

最初のソースオペランド

最初のオペランドは、どのバンクにあるかにかかわらず常にベクタです。これによって、最初のバンクにある連続したレジスタをベクタとして扱うことができます。

ベクタのラップアラウンド

ベクタ演算は、ラップアラウンドしてすでに使用したエレメントを再使用することはできません。これを行った場合、命令の結果は予測不能です。ベクタ長は最大 8 なため、FPSCR でベクタストライドとして 1 が指定されている場合、これは制約とはなりません。FPSCR でベクタストライドとして 2 が指定されている場合、ベクタ長を最大 4 にする必要があります。

オペランドのオーバラップ

2 つのオペランドがオーバラップしている場合、アクセスされるレジスタとそのアクセス順序の両方が同一の必要があります。そうでない場合、命令の結果は予測不能です。このため、次のような制約があります。

- `Sd[i]` で生成されたレジスタ番号の組が、`Sn[i]` で生成されたレジスタ番号の組とオーバラップしている場合、`d_num` と `n_num` は同一の必要があります。
- `Sn[i]` で生成されるレジスタ番号の組に `m_num` が含まれている場合、ベクタ長は 1 の必要があります。

バンクが異なるため、`Sd[i]` で生成されるレジスタ番号の組に `m_num` が含まれることはありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.1.5 ベクタ操作

デスティネーションレジスタと2番目のソースレジスタのいずれも、最初のバンクに含まれる8つのレジスタ内にない場合、すべてのレジスタオペランドがベクタとして扱われます。

```
if d_bank != 0 and m_bank != 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Sd[i] = (d_bank << 3) | d_index
        Sn[i] = (n_bank << 3) | n_index
        Sm[i] = (m_bank << 3) | m_index
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 7 then
            d_index = d_index - 8
        n_index = n_index + (vector stride specified by FPSCR)
        if n_index > 7 then
            n_index = n_index - 8
        m_index = m_index + (vector stride specified by FPSCR)
        if m_index > 7 then
             m_index = m_index - 8
```

注

ベクタのラップアラウンド

ベクタ演算は、ラップアラウンドしてすでに使用したエレメントを再使用することはできません。これを行った場合、命令の結果は予測不能です。ベクタ長は最大 8 なため、FPSCR でベクタストライドとして 1 が指定されている場合、これは制約とはなりません。FPSCR でベクタストライドとして 2 が指定されている場合、ベクタ長を最大 4 にする必要があります。

オペランドのオーバラップ

2つのオペランドがオーバラップしている場合、アクセスされるレジスタとそのアクセス順序の両方が同一の必要があります。そうでない場合、命令の結果は予測不能です。このため、次のような制約があります。

- `Sd[i]` で生成されたレジスタ番号の組が、`Sn[i]` で生成されたレジスタ番号の組とオーバラップしている場合、`d_num` と `n_num` は同一の必要があります。
- `Sd[i]` で生成されたレジスタ番号の組が、`Sm[i]` で生成されたレジスタ番号の組とオーバラップしている場合、`d_num` と `m_num` は同一の必要があります。
- `Sn[i]` で生成されたレジスタ番号の組が、`Sm[i]` で生成されたレジスタ番号の組とオーバラップしている場合、`n_num` と `m_num` は同一の必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.2 アドレシングモード 2 - 倍精度ベクタ（非単項）

31–28	27	26	25	24	23	22	21–20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	Op	0	Op	Dn	Dd	1	0	1	1	0	Op	0	0	Dm

FPSCR で示されているベクタ長が 1 より大きい場合、倍精度の 2 オペランド命令（FADDD、FDIVD、FMULD、FNMULD、FSUBD）では 3 つの異なる動作を指定できます。

- 2 つのスカラ値に対して 1 つの演算を実行し、結果はスカラになる。

  ScalarA op ScalarB → ScalarD

  この方式が選択されている場合（P. C5-11「*スカラ演算*」参照）、FPSCR で指定されているベクタ長は無視され、実行される演算は 1 つだけです。これによって、FPSCR を再プログラムする必要なしに、スカラ演算とベクタ演算の混在が可能です。

- N 個の数値演算の組。N は FPSCR により指定されているベクタ長で、最初のオペランドはベクタ内でスキャンされ、2 番目のオペランドは一定で、デスティネーションはベクタ内でスキャンされます。

  VectorA[0] op ScalarB → VectorD[0]
  VectorA[1] op ScalarB → VectorD[1]
  ...
  VectorA[N-1] op ScalarB → VectorD[N-1]

  これは、次のように短縮できます。

  VectorA op ScalarB → VectorD

- N 個の数値演算の組。N は FPSCR により指定されているベクタ長で、両方のオペランドとデスティネーションがベクタ内でスキャンされます。

  VectorA[0] op VectorB[0] → VectorD[0]
  VectorA[1] op VectorB[1] → VectorD[1]
  ...
  VectorA[N-1] op VectorB[N-1] → VectorD[N-1]

  これは、次のように短縮できます。

  VectorA op VectorB → VectorD

倍精度の 3 オペランド命令（FMACD、FMSCD、FNMACD、FNMSCD）では、加算 / 減算オペランドとデスティネーションに同じレジスタを使用します。このため、上の 3 つに対応する 3 つの形式が存在します。

- 純粋なスカラ形式

  ±（ScalarA * ScalarB）± ScalarD → ScalarD

- 乗算の 2 番目のオペランドがスカラで、他のすべてがベクタ内でスキャンされる形式

  ±（VectorA[0] * ScalarB）± VectorD[0] → VectorD[0]
  ±（VectorA[1] * ScalarB）± VectorD[1] → VectorD[1]
  ...
  ±（VectorA[N-1] * ScalarB）± VectorD[N-1] → VectorD[N-1]

  これは、次のように短縮できます。

  ±（VectorA * ScalarB）± VectorD → VectorD

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- すべてのオペランドがベクタ内でスキャンされる形式

  ```
  ±（VectorA[0] * VectorB[0]）± VectorD[0] → VectorD[0]
  ±（VectorA[1] * VectorB[1]）± VectorD[1] → VectorD[1]
  ...
  ±（VectorA[N-1] * VectorB[N-1]）± VectorD[N-1] → VectorD[N-1]
  ```

  これは、次のように短縮できます。

  ```
  ±（VectorA * VectorB）± VectorD → VectorD
  ```

C5.2.1 レジスタバンク

これら各種の形式を指定できるようにするため、16 個の倍精度レジスタの組は、それぞれ 4 つのレジスタで構成される 4 つのバンクに分割されます。命令によって使用される形式は、最初のバンクにあるオペランドに依存します。規則の根底となる一般原則は、スカラオペランドの保持には最初のバンクを使用する必要があり、他のバンクはベクタオペランドの保持に使用されることです。すべてのデスティネーションレジスタへの書き込みと、多くのソースレジスタの読み出しはこの原則に従っていますが、一部のソースレジスタ読み出しはベクタエレメントへのスカラアクセスや、スカラのグループへのベクタアクセスを行うことがあります。

ベクタオペランドは、単一のバンクに含まれる 2 〜 4 個のレジスタで構成され、レジスタの数は FPSCR のベクタ長フィールドによって指定されます。詳細については、P. C2-24 *「ベクタの長さ / ストライドの制御」*を参照して下さい。命令に含まれるレジスタ番号は、ベクタの最初のエレメントを含むレジスタを指定します。ベクタ内の以降の各エレメントは、FPSCR のベクタストライドフィールドで指定された値だけレジスタ番号をインクリメントして得られます。これによってレジスタ番号がレジスタバンクの最上位からオーバフローする場合、図 C5-2 に示すように、レジスタ番号はバンクの最下位にラップアラウンドします。

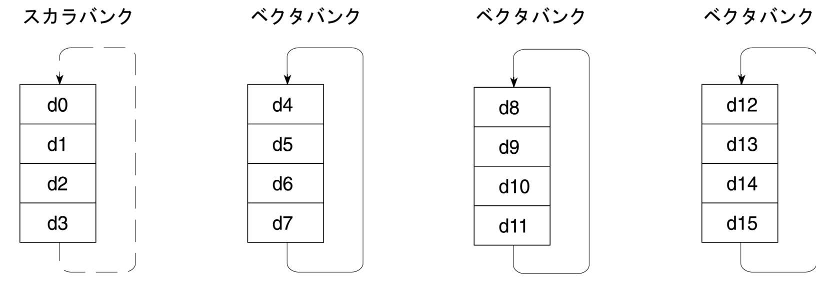


図 C5-2 倍精度レジスタのバンク

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.2.2 動作

以下のページでは、可能な 3 つのアドレシングモードの各形式について解説します。

- *スカラ演算*：P. C5-11
- *ベクタとスカラの混在した演算*：P. C5-12
- *ベクタ操作*：P. C5-13

どの場合も、以下の値が生成されます。

`vec_len` 命令によって指定される個別の演算の数。

`Dd[0] ... Dd[vec_len-1]`
個別の演算のデスティネーションレジスタ。

`Dn[0] ... Dn[vec_len-1]`
個別の演算の最初のソースレジスタ。

`Dm[0] ... Dm[vec_len-1]`
個別の演算の 2 番目のソースレジスタ。

命令に指定されているレジスタ番号は次のようにバンク番号と、バンク内のインデクスに分解されます。

```
d_bank  = Dd[3:2]
d_index = Dd[1:0]

n_bank  = Dn[3:2]
n_index = Dn[1:0]

m_bank  = Dm[3:2]
m_index = Dm[1:0]
```

——— 注 ———

FPSCR でベクタ長に 1 が指定されるのは特別なケースではありません。アドレシングモードの 3 つの形式すべての規則は、ベクタ長が 1 の場合には次のように簡素化されます。

```
vec_len = 1
Dd[0] = Dd
Dn[0] = Dn
Dm[0] = Dm
```

———————

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.2.3 スカラ演算

デスティネーションレジスタが最初のバンクに含まれる 4 つのレジスタ内にある場合、その命令についてスカラ演算が指定されます。

```
if d_bank == 0 then
    vec_len = 1
    Dd[0] = Dd
    Dn[0] = Dn
    Dm[0] = Dm
```

注

ソースオペランド　ソースオペランドは、どのバンクにあるかにかかわらず常にスカラです。これによって、ベクタの個別のエレメントをスカラとして使用できます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.2.4 ベクタとスカラの混在した演算

命令で指定されたデスティネーションレジスタが最初のバンクに含まれる4つのレジスタ内ではなく、2 番目のソースレジスタが最初のバンクに含まれている場合、デスティネーションレジスタと最初のソースレジスタはベクタを、2 番目のソースレジスタはスカラを指定します。

```
if d_bank != 0 and m_bank == 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Dd[i] = (d_bank << 2) | d_index
        Dn[i] = (n_bank << 2) | n_index
        Dm[i] = Dm
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 3 then
            d_index = d_index - 4
        n_index = n_index + (vector stride specified by FPSCR)
        if n_index > 3 then
            n_index = n_index - 4
```

注

最初のソースオペランド

最初のオペランドは、どのバンクにあるかにかかわらず常にベクタです。これによって、最初のバンクにある連続したレジスタをベクタとして扱うことができます。

ベクタのラップアラウンド

ベクタ演算は、ラップアラウンドしてすでに使用したエレメントを再使用することはできません。これを行った場合、命令の結果は予測不能です。FPSCR でベクタストライドとして 1 が指定されている場合、ベクタ長を最大 4 にする必要があります。FPSCR でベクタストライドとして 2 が指定されている場合、ベクタ長を最大 2 にする必要があります。

オペランドのオーバラップ

2 つのオペランドがオーバラップしている場合、アクセスされるレジスタとそのアクセス順序の両方が同一の必要があります。そうでない場合、命令の結果は予測不能です。このため、次のような制約があります。

- Dd[i] で生成されたレジスタ番号の組が、Dn[i] で生成されたレジスタ番号の組とオーバラップしている場合、Dd と Dn は同一の必要があります。
- Dn[i] で生成されるレジスタ番号の組に Dm が含まれている場合、ベクタ長は 1 の必要があります。

バンクが異なるため、Dd[i] で生成されるレジスタ番号の組に Dm が含まれることはありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.2.5 ベクタ操作

デスティネーションレジスタと 2 番目のソースレジスタのいずれも、最初のバンクに含まれる 4 つのレジスタ内にない場合、すべてのレジスタオペランドがベクタとして扱われます。

```
if d_bank != 0 and m_bank != 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Dd[i] = (d_bank << 2) | d_index
        Dn[i] = (n_bank << 2) | n_index
        Dm[i] = (m_bank << 2) | m_index
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 3 then
            d_index = d_index - 4
        n_index = n_index + (vector stride specified by FPSCR)
        if n_index > 3 then
            n_index = n_index - 4
        m_index = m_index + (vector stride specified by FPSCR)
        if m_index > 3 then
            m_index = m_index - 4
```

注

ベクタのラップアラウンド

ベクタ演算は、ラップアラウンドしてすでに使用したエレメントを再使用することはできません。そうでない場合、命令の結果は予測不能です。FPSCR でベクタストライドとして 1 が指定されている場合、ベクタ長を最大 4 にする必要があります。FPSCR でベクタストライドとして 2 が指定されている場合、ベクタ長を最大 2 にする必要があります。

オペランドのオーバラップ

2 つのオペランドがオーバラップしている場合、アクセスされるレジスタとそのアクセス順序の両方が同一の必要があります。そうでない場合、命令の結果は予測不能です。このため、次のような制約があります。

- Dd[i] で生成されたレジスタ番号の組が、Dn[i] で生成されたレジスタ番号の組とオーバラップしている場合、Dd と Dn は同一の必要があります。
- Dd[i] で生成されたレジスタ番号の組が、Dm[i] で生成されたレジスタ番号の組とオーバラップしている場合、Dd と Dm は同一の必要があります。
- Dn[i] で生成されたレジスタ番号の組が、Dm[i] で生成されたレジスタ番号の組とオーバラップしている場合、Dn と Dm は同一の必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.3 アドレシングモード 3 - 単精度ベクタ（単項）

31–28	27–23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
cond	1 1 1 0 1	D	1 1	Op	Fd	1 0 1 0	Op	1	M	0	Fm

FPSCR で示されているベクタ長が 1 より大きい場合、単精度の 1 オペランド命令（`FABSS`、`FCPYS`、`FNEGS`、`FSQRTS`）では 3 つの異なる動作を指定できます。

- スカラ値の演算として、スカラが生成される。

  `Op(ScalarB) → ScalarD`

  この方式が選択されている場合（P. C5-16「*スカラからスカラへの演算*」参照）、FPSCR で指定されているベクタ長は無視され、実行される演算は 1 つだけです。これによって、FPSCR を再プログラムする必要なしに、スカラ演算とベクタ演算の混在が可能です。

- スカラ値への演算として、結果がベクタの `N` 個のエレメントそれぞれに書き込まれる。ここで、`N` は FPSCR で指定されているベクタ長です。

  ```
  Op(ScalarB) → VectorD[0]
  Op(ScalarB) → VectorD[1]
  ...
  Op(ScalarB) → VectorD[N-1]
  ```

  これは、次のように短縮できます。

  `Op(ScalarB) → VectorD`

- `N` 個の演算の組。`N` は FPSCR により指定されているベクタ長で、オペランドとデスティネーションの両方がベクタ内でスキャンされます。

  ```
  Op(VectorB[0]) → VectorD[0]
  Op(VectorB[1]) → VectorD[1]
  ...
  Op(VectorB[N-1]) → VectorD[N-1]
  ```

  これは、次のように短縮できます。

  `Op(VectorB) → VectorD`

これら各種の形式を指定できるようにするため、32 個の単精度レジスタの組は、それぞれ 8 つのレジスタで構成される 4 つのバンクに分割されます。詳細については、P. C5-3「*レジスタバンク*」を参照して下さい。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.3.1　動作

以下のページでは、可能な 3 つのアドレシングモードの各形式について解説します。

- *スカラからスカラへの演算*：P. C5-16
- *スカラからベクタへの演算*：P. C5-16
- *ベクタからベクタへの演算*：P. C5-17

どの場合も、以下の値が生成されます。

`vec_len`　　命令によって指定される個別の演算の数。

`Sd[0] ... Sd[vec_len-1]`

個別の演算のデスティネーションレジスタ。

`Sm[0] ... Sm[vec_len-1]`

個別の演算のソースレジスタ。

どの場合も、命令によって指定されるレジスタは命令の Fd と Fm フィールドを、それぞれ D および M ビットと連結することで決定されます。

```
d_num = (Fd << 1) | D
m_num = (Fm << 1) | M
```

次に、これらのレジスタ番号は次のようにバンク番号と、バンク内のインデクスに分解されます。

```
d_bank  = d_num[4:3]
d_index = d_num[2:0]

m_bank  = m_num[4:3]
m_index = m_num[2:0]
```

——— 注 ———

FPSCR でベクタ長に 1 が指定されるのは特別なケースではありません。アドレシングモードの 3 つの形式すべての規則は、ベクタ長が 1 の場合には次のように簡素化されます。

```
vec_len = 1
Sd[0] = d_num
Sm[0] = m_num
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.3.2 スカラからスカラへの演算

デスティネーションレジスタが最初のバンクに含まれる 8 つのレジスタ内にある場合、その命令についてスカラ演算が指定されます。

```
if d_bank == 0 then
    vec_len = 1
    Sd[0] = d_num
    Sm[0] = m_num
```

注

ソースオペランド　ソースオペランドは、どのバンクにあるかにかかわらず常にスカラです。これによって、ベクタの個別のエレメントをスカラとして使用できます。

C5.3.3 スカラからベクタへの演算

命令で指定されたデスティネーションレジスタが最初のバンクに含まれる8つのレジスタ内ではなく、ソースレジスタが最初のバンクに含まれている場合、デスティネーションレジスタはベクタを、ソースレジスタはスカラを指定します。

```
if d_bank != 0 and m_bank == 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Sd[i] = (d_bank << 3) | d_index
        Sm[i] = m_num
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 7 then
            d_index = d_index - 8
```

注

ベクタのラップアラウンド

ベクタ演算は、ラップアラウンドしてすでに使用したエレメントを再使用することはできません。これを行った場合、命令の結果は予測不能です。ベクタ長は最大 8 なため、FPSCR でベクタストライドとして 1 が指定されている場合、これは制約とはなりません。FPSCR でベクタストライドとして 2 が指定されている場合、ベクタ長を最大 4 にする必要があります。

オペランドのオーバラップ

ソースとデスティネーションがオーバラップしている場合、アクセスされるレジスタとそのアクセス順序の両方が同一の必要があります。`Sn[i]` で生成されるレジスタ番号の組に `m_num` が含まれている場合、ベクタ長は 1 の必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.3.4 ベクタからベクタへの演算

デスティネーションレジスタとソースレジスタのいずれも、最初のバンクに含まれる 8 つのレジスタ内にない場合、両方のレジスタオペランドがベクタとして扱われます。

```
if d_bank != 0 and m_bank != 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Sd[i] = (d_bank << 3) | d_index
        Sm[i] = (m_bank << 3) | m_index
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 7 then
            d_index = d_index - 8
        m_index = m_index + (vector stride specified by FPSCR)
        if m_index > 7 then
            m_index = m_index - 8
```

注

ベクタのラップアラウンド

ベクタ演算は、ラップアラウンドしてすでに使用したエレメントを再使用することはできません。そうでない場合、命令の結果は予測不能です。ベクタ長は最大 8 なため、FPSCR でベクタストライドとして 1 が指定されている場合、これは制約とはなりません。FPSCR でベクタストライドとして 2 が指定されている場合、ベクタ長を最大 4 にする必要があります。

オペランドのオーバラップ

ソースとデスティネーションがオーバラップしている場合、アクセスされるレジスタとそのアクセス順序の両方が同一の必要があります。そうでない場合、命令の結果は予測不能です。`Sd[i]` で生成されたレジスタ番号の組が、`Sm[i]` で生成されたレジスタ番号の組とオーバラップしている場合、`d_num` と `m_num` は同一の必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.4 アドレシングモード 4 - 倍精度ベクタ（単項）

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	1	1	1	0	1	0	1	1	Op	Dd	1	0	1	1	Op	1	0	0	Dm

FPSCR で示されているベクタ長が 1 より大きい場合、倍精度の 1 オペランド命令（FABSD、FCPYD、FNEGD、FSQRTD）では 3 つの異なる動作を指定できます。

- スカラ値の演算として、スカラが生成される。

  ```
  Op(ScalarB)  -->  ScalarD
  ```

  この方式が選択されている場合（P. C5-20 *「スカラからスカラへの演算」*参照）、FPSCR で指定されているベクタ長は無視され、実行される演算は 1 つだけです。これによって、FPSCR を再プログラムする必要なしに、スカラ演算とベクタ演算の混在が可能です。

- スカラ値への演算として、結果がベクタの N 個のエレメントそれぞれに書き込まれる。ここで、N は FPSCR で指定されているベクタ長です。

  ```
  Op(ScalarB)  -->  VectorD[0]
  Op(ScalarB)  -->  VectorD[1]
  ...
  Op(ScalarB)  -->  VectorD[N-1]
  ```

  これは、次のように短縮できます。

  ```
  Op(ScalarB)  -->  VectorD
  ```

- N 個の演算の組。N は FPSCR により指定されているベクタ長で、オペランドとデスティネーションの両方がベクタ内でスキャンされます。

  ```
  Op(VectorB[0])     -->  VectorD[0]
  Op(VectorB[1])     -->  VectorD[1]
  ...
  Op(VectorB[N-1])  -->  VectorD[N-1]
  ```

  これは、次のように短縮できます。

  ```
  Op(VectorB)  -->  VectorD
  ```

これら各種の形式を指定できるようにするため、16 個の倍精度レジスタの組は、それぞれ 4 つのレジスタで構成される 4 つのバンクに分割されます。詳細については、P. C5-9 *「レジスタバンク」*を参照して下さい。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.4.1 動作

以下のページでは、可能な 3 つのアドレシングモードの各形式について解説します。

- *スカラからスカラへの演算* : P. C5-20
- *スカラからベクタへの演算* : P. C5-20
- *ベクタからベクタへの演算* : P. C5-21

どの場合も、以下の値が生成されます。

`vec_len` 命令によって指定される個別の演算の数。

`Dd[0] ... Dd[vec_len-1]`
個別の演算のデスティネーションレジスタ。

`Dm[0] ... Dm[vec_len-1]`
個別の演算のソースレジスタ。

命令に指定されているレジスタ番号は次のようにバンク番号と、バンク内のインデクスに分解されます。

```
d_bank  = Dd[3:2]
d_index = Dd[1:0]

m_bank  = Dm[3:2]
m_index = Dm[1:0]
```

——— 注 ———

FPSCR でベクタ長に 1 が指定されるのは特別なケースではありません。アドレシングモードの 3 つの形式すべての規則は、ベクタ長が 1 の場合には次のように簡素化されます。

```
vec_len = 1
Dd[0] = Dd
Dm[0] = Dm
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.4.2 スカラからスカラへの演算

デスティネーションレジスタが最初のバンクに含まれる 4 つのレジスタ内にある場合、その命令についてスカラ演算が指定されます。

```
if d_bank == 0 then
    vec_len = 1
    Dd[0] = Dd
    Dm[0] = Dm
```

注

ソースオペランド　ソースオペランドは、どのバンクにあるかにかかわらず常にスカラです。これによって、ベクタの個別のエレメントをスカラとして使用できます。

C5.4.3 スカラからベクタへの演算

命令で指定されたデスティネーションレジスタが最初のバンクに含まれる4つのレジスタ内ではなく、ソースレジスタが最初のバンクに含まれている場合、デスティネーションレジスタはベクタを、ソースレジスタはスカラを指定します。

```
if d_bank != 0 and m_bank == 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Dd[i] = (d_bank << 2) | d_index
        Dm[i] = m_num
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 3 then
            d_index = d_index - 4
```

注

ベクタのラップアラウンド

ベクタ演算は、ラップアラウンドしてすでに使用したエレメントを再使用することはできません。そうでない場合、命令の結果は予測不能です。FPSCR でベクタストライドとして 1 が指定されている場合、ベクタ長を最大 4 にする必要があります。FPSCR でベクタストライドとして 2 が指定されている場合、ベクタ長を最大 2 にする必要があります。

オペランドのオーバラップ

ソースとデスティネーションがオーバラップしている場合、アクセスされるレジスタとそのアクセス順序の両方が同一の必要があります。`Dn[i]` で生成されるレジスタ番号の組に`Dm`が含まれている場合、ベクタ長は1の必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.4.4 ベクタからベクタへの演算

デスティネーションレジスタとソースレジスタのいずれも、最初のバンクに含まれる 4 つのレジスタ内にない場合、両方のレジスタオペランドがベクタとして扱われます。

```
if d_bank != 0 and m_bank != 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Dd[i] = (d_bank << 2) | d_index
        Dm[i] = (m_bank << 2) | m_index
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 3 then
            d_index = d_index - 4
        m_index = m_index + (vector stride specified by FPSCR)
        if m_index > 3 then
            m_index = m_index - 4
```

注

ベクタのラップアラウンド

ベクタ演算は、ラップアラウンドしてすでに使用したエレメントを再使用することはできません。これを行った場合、命令の結果は予測不能です。FPSCR でベクタストライドとして 1 が指定されている場合、ベクタ長を最大 4 にする必要があります。FPSCR でベクタストライドとして 2 が指定されている場合、ベクタ長を最大 2 にする必要があります。

オペランドのオーバラップ

ソースとデスティネーションがオーバラップしている場合、アクセスされるレジスタとそのアクセス順序の両方が同一の必要があります。そうでない場合、命令の結果は予測不能です。Dd[i] で生成されたレジスタ番号の組が、Dm[i] で生成されたレジスタ番号の組とオーバラップしている場合、Dd と Dm は同一の必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.5 アドレシングモード 5 - VFP 複数ロード / ストア

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7–0
cond	1	1	0	P	U	D	W	L	Rn	Fd	cp_num	offset

VFP 複数ロード命令（FLDMD、FLDMS、FLDMX）は、ARM® LDC 命令の一種です。これらのアドレシングモードの詳細については、P. A5-49 *「アドレシングモード 5 - コプロセッサのロード / ストア」*を参照して下さい。同様に、VFP 複数ストア命令（FSTMD、FSTMS、FSTMX）は、ARM STC 命令の一種で、同じアドレシングモードを持っています。ただし、VFP の複数ロードおよび複数ストア命令では、LDC/STC のアドレシングモードすべてが利用可能ではありません。これは、FLDD、FLDS、FSTD、FSTS 命令が一部専用のオプションを使用することと、LDC/STC 命令の 8_bit_offset フィールドが VFP 命令では別の目的に使用されることが理由です。

このセクションでは、VFP の複数ロード命令と複数ストア命令で使用できる LDC/STC アドレシングモードの詳細と、それぞれのオプションのアセンブラ構文について説明します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.5.1 概要

LDC/STC アドレシングモードが VFP の複数ロード / ストア命令で使用できるかどうかは、命令の P、U、W ビットにより判定可能です。この詳細を表 C5-1 に示します。

表 C5-1 VFP ロード / ストアのアドレシングモード

P	U	W	命令	モード
0	0	0	2 レジスタ転送命令	
0	0	1	未定義	
0	1	0	FLDMD、FLDMS、FLDMX、FSTMD、FSTMS、FSTMX	インデクスなし
0	1	1	FLDMD、FLDMS、FLDMX、FSTMD、FSTMS、FSTMX	インクリメント
1	0	0	FLDD、FLDS、FSTD、FSTS	(負のオフセット)
1	0	1	FLDMD、FLDMS、FLDMX、FSTMD、FSTMS、FSTMX	デクリメント
1	1	0	FLDD、FLDS、FSTD、FSTS	(正のオフセット)
1	1	1	未定義	注を参照して下さい

——— 注 ———

VFP 命令セットの実行においてハードウェアコプロセッサを実装している場合、表 C5-1 の未定義エントリは、コプロセッサがその命令に応答しないことを示します。この場合、ARM 未定義命令例外が発生します。詳細については、P. A2-19 *「未定義命令例外」*を参照して下さい。

ソフトウェア実装の場合、未定義エントリはその命令を、コプロセッサ以外の未定義命令を処理するシステムの通常の機構に渡す必要があることを示します。この正確な詳細はシステムによって異なります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.5.2 VFP 複数ロード / ストア - インデクスなし

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7–0
cond	1	1	0	0	1	D	0	L	Rn	Fd	cp_num	offset

このアドレシングモードは、VFP の複数ロードおよび複数ストア命令用のもので、広範なアドレスを構成します。構成される最初のアドレスは start_address で、ベースレジスタ Rn の値です。以降のアドレスは、前のアドレスに 4 を加算して生成されます。

- FLDMS 命令と FSTMS 命令の場合、命令の offset は転送される単精度レジスタの数と同じです。各レジスタについて 1 つのアドレスが生成されるため、end_address は（ベースレジスタ Rn の値 + offset * 4 - 4）になります。
- FLDMD 命令と FSTMD 命令の場合、命令の offset は転送される倍精度レジスタの数の 2 倍です。各レジスタについて 2 つのアドレスが生成されるため、end_address は（ベースレジスタ Rn の値 + offset * 4 - 4）になります。
- FLDMX 命令と FSTMX 命令の場合、命令の offset は転送される倍精度レジスタの数の 2 倍より 1 多くなります。各レジスタについて 2 つのアドレスが生成されるため、end_address は（ベースレジスタ Rn の値 + offset * 4 - 8）になります。

命令の構文

```
<opcode>IA<precision>{<cond>}  <Rn>, <registers>
```

各項目の説明については以下を参照して下さい。

<opcode> FLDM または FSTM で、L ビットの値を決定します。

<precision> D、S、X のいずれかで、cp_num と offset[0] の値を決定します。

<cond> この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rn> ベースレジスタを指定します。<Rn> として R15 が指定されている場合、命令のアドレス + 8 が値として使用されます。

<registers> 命令によってロード / ストアされるレジスタの一覧を指定します。命令で指定されるレジスタと、Fd、D、offset の設定方法の詳細については、各命令の解説を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if (offset[0] == 1) and (cp_num == 0b1011) then          /* FLDMX or FSTMX */
    word_count = offset - 1
else                                                     /* Others */
    word_count = offset
start_address = Rn
end_address = start_address + 4 * word_count - 4
```

用法

FLDMD、FLDMS、FSTMD、FSTMS の場合、このアドレシングモードは一般にショートベクタのロード / ストアに使用されます。たとえば、4 つの単精度座標で構成されるグラフィックのポイントを s8 〜 s11 にロードするには、次のようなコードが使用できます。

```
ADR      Rn, Point
FLDMIAS  Rn, {s8-s11}
```

FLDMX と FSTMX の場合、このアドレシングモードは一般に、プロセススワップコードで VFP のステートをロードまたは保存する動作の一部として、次のようなシーケンスで使用されます。

```
; Assume Rp points to the process block
ADD         Rn, Rp, #Offset_to_VFP_register_dump
FSTMIAX         Rn, {d0-d15}
```

注

オフセットの制限 オフセットの値は最低で 1、最大で 33 であることが必要です。オフセットが 0、または 33 より大きい場合、命令の動作は予測不能です。各命令で、Fd と D の値によっては、オフセットについて他の制限があります。これらの制限の詳細については、個別の命令の解説を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.5.3 VFP 複数ロード / ストア - インクリメント

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1	1	0	0	1	D	1	L	Rn	Fd	cp_num	offset

このアドレシングモードは、VFP の複数ロードおよび複数ストア命令用のもので、広範なアドレスを構成します。構成される最初のアドレスは start_address で、ベースレジスタ Rn の値です。以降のアドレスは、前のアドレスに 4 を加算して生成されます。

- FLDMS 命令と FSTMS 命令の場合、命令の offset は転送される単精度レジスタの数と同じです。各レジスタについて 1 つのアドレスが生成されるため、end_address は（ベースレジスタ Rn の値 + offset * 4 - 4）になります。
- FLDMD 命令と FSTMD 命令の場合、命令の offset は転送される倍精度レジスタの数の 2 倍です。各レジスタについて 2 つのアドレスが生成されるため、end_address は（ベースレジスタ Rn の値 + offset * 4 - 4）になります。
- FLDMX 命令と FSTMX 命令の場合、命令の offset は転送される倍精度レジスタの数の 2 倍より 1 多くなります。各レジスタについて 2 つのアドレスが生成されるため、end_address は（ベースレジスタ Rn の値 + offset * 4 - 8）になります。

すべての命令について、命令で指定されている条件が条件コードのステータスと一致する場合（P. A3-3「*条件フィールド*」参照）、Rn は命令で指定されている offset の 4 倍だけインクリメントされます。

命令の構文

```
<opcode>IA<precision>{<cond>}  <Rn>!, <registers>
```

各項目の説明については以下を参照して下さい。

<opcode>
: FLDM または FSTM で、L ビットの値を決定します。

<precision>
: D、S、X のいずれかで、cp_num と offset[0] の値を決定します。

<cond>
: この命令が実行される条件。条件は P. A3-3「*条件フィールド*」で定義されています。<cond> が省略されている場合、AL（常時）条件が使用されます。

<Rn>
: ベースレジスタを指定します。<Rn> として R15 を指定した場合、命令の動作は予測不能です。

!
: このアドレシングモードで発生するベースレジスタライトバックを示します。省略されている場合、インデクスなしのアドレシングモード（P. C5-24「*VFP 複数ロード / ストア - インデクスなし*」参照）となります。

<registers>
: 命令によってロード / ストアされるレジスタの一覧を指定します。指定されるレジスタと、Fd、D、offset の設定方法の詳細については、各命令の解説を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

動作

```
if (offset[0] == 1) and (cp_num == 0b1011) then  /* FLDMX or FSTMX */
    word_count = offset - 1
else                                             /* Others */
    word_count = offset
start_address = Rn
end_address = start_address + 4 * word_count - 4
if ConditionPassed(cond) then
    Rn = Rn + 4 * offset
```

用法

FLDMD、FLDMS、FSTMD、FSTMS の場合、このアドレシングモードは一般にショートベクタ配列をロード / ストアし、ポインタを次のエレメントに進めるために使用されます。たとえば、Rn がグラフィックポイントの配列のエレメントを指しており、それぞれが 4 つの単精度座標で構成されている場合、次のような命令を使用します。

```
FSTMIAS  Rn!, {s16-s19}
```

この命令は、単精度レジスタ s16、s17、s18、s19 の値を配列の現在のエレメントにストアし、Rn が次のエレメントを指すように進めます。

関連する使用法は、浮動小数点データのロングベクタについて発生します。Rn が単精度値のロングベクタを指している場合、同じ命令で s16、s17、s18、s19 をベクタの次の 4 つのエレメントにストアし、Rn が次のエレメントを指すように進めることができます。

FSTMD、FSTMS、FSTMX の場合、このアドレシングモードは空上昇スタックにレジスタの値をプッシュするために便利です。レジスタに倍精度データ、または単精度データのみが含まれていることが既知の場合、それぞれ FSTMD と FSTMS を使用します。レジスタに保持されているデータの精度が不明で、ストアされたデータに対して、後で対応する FLDMX 命令で再ロードする以外に行う操作がない場合は、FSTMX を使用します。これはたとえば、プロシージャのエントリシーケンスで、呼び出し先保存レジスタについて使用します。

複数のレジスタがあり、保持されている値の精度は判明しているがレジスタによって異なっている場合、ストアされたデータに対して、対応する FLDMX 命令で再ロードする以外に行う操作がなければ、FSTMX を使用してそれらのデータをスタックにプッシュできます。それ以外の場合は、FSTMD 命令と FSTMS 命令のシーケンスを使用する必要があります。

FLDMD、FLDMS、FLDMX の場合、このアドレシングモードはフル下降スタックからデータをポップするために便利です。どの命令を使用するかの選択基準は、前に説明したものと同じです。

注

オフセットの制限 オフセットの値は最低で 1、最大で 33 であることが必要です。オフセットが 0、または 33 より大きい場合、命令の動作は予測不能です。各命令で、Fd と D の値によっては、オフセットについて他の制限があります。これらの制限の詳細については、個別の命令の解説を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.5.4 VFP 複数ロード / ストア - デクリメント

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1 1 0	1	0	D	1	L	Rn	Fd	cp_num	offset

このアドレシングモードは、VFP の複数ロードおよび複数ストア命令用のもので、広範なアドレスを構成します。構成される最初のアドレスは start_address で、（ベースレジスタ Rn の値 - offset * 4）です。以降のアドレスは、前のアドレスに 4 を加算して生成されます。

- `FLDMS` 命令と `FSTMS` 命令の場合、命令の offset は転送される単精度レジスタの数と同じです。各レジスタについて 1 つのアドレスが生成されるため、end_address は（ベースレジスタ Rn の値 - 4）になります。
- `FLDMD` 命令と `FSTMD` 命令の場合、命令の offset は転送される倍精度レジスタの数の 2 倍です。各レジスタについて 2 つのアドレスが生成されるため、end_address は（ベースレジスタ Rn の値 - 4）になります。
- `FLDMX` 命令と `FSTMX` 命令の場合、命令の offset は転送される倍精度レジスタの数の 2 倍より 1 多くなります。各レジスタについて 2 つのアドレスが生成されるため、end_address は（ベースレジスタ Rn の値 - 8）になります。

すべての命令について、命令で指定されている条件が条件コードのステータスと一致する場合、Rn は命令で指定されている offset の 4 倍だけデクリメントされます。条件は P. A3-3 *「条件フィールド」* で定義されています。

命令の構文

```
<opcode>DB<precision>{<cond>}  <Rn>!, <registers>
```

各項目の説明については以下を参照して下さい。

<opcode>　`FLDM` または `FSTM` で、L ビットの値を決定します。

<precision>
　D、S、X のいずれかで、cp# と offset[0] の値を決定します。

<cond>　この命令が実行される条件。条件は P. A3-3 *「条件フィールド」* で定義されています。<cond> が省略されている場合、`AL`（常時）条件が使用されます。

<Rn>　ベースレジスタを指定します。<Rn> として R15 を指定した場合、命令の動作は予測不能です。

!　このアドレシングモードで発生するベースレジスタライトバックを示します。このアドレシングモードにはライトバックのないバリエーションが存在しないため、この項目は省略できません。

<registers>
　命令によってロード / ストアされるレジスタの一覧を指定します。命令で指定されるレジスタと、Fd、D、offset の設定方法の詳細については、各命令の解説を参照して下さい。

アーキテクチャのバージョン

すべてのアーキテクチャバージョンに存在します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

動作

```
if (offset[0] == 1) and (cp_num == 0b1011) then  /* FLDMX or FSTMX */
    word_count = offset - 1
else                                             /* Others */
    word_count = offset
start_address = Rn - 4 * offset
end_address = start_address +4 * word_count - 4
if ConditionPassed(cond) then
    Rn = Rn - 4 * offset
```

用法

FSTMD、FSTMS、FSTMX の場合、このアドレシングモードはフル下降スタックにレジスタの値をプッシュするために便利です。レジスタに倍精度データ、または単精度データのみが含まれていることが既知の場合、それぞれ FSTMD と FSTMS を使用します。レジスタに保持されているデータの精度が不明で、ストアされたデータに対して、後で対応する FLDMX 命令で再ロードする以外に行う操作がない場合は、FSTMX を使用します。これはたとえば、プロシージャのエントリシーケンスで、呼び出し先保存レジスタについて使用します。

複数のレジスタがあり、保持されている値の精度は判明しているがレジスタによって異なっている場合、ストアされたデータに対して、対応する FLDMX 命令で再ロードする以外に行う操作がなければ、FSTMX を使用してそれらのデータをスタックにプッシュできます。それ以外の場合は、FSTMD 命令と FSTMS 命令のシーケンスを使用する必要があります。

FLDMD、FLDMS、FLDMX の場合、このアドレシングモードは空上昇スタックからデータをポップするために便利です。どの命令を使用するかの選択基準は、前に説明したものと同じです。

FLDMD、FLDMS、FSTMD、FSTMS の場合、このアドレシングモードはロングベクタやショートベクタの配列を逆方向にスキャンするコードでも使用できます。どちらの場合も、エレメントへのポインタは一連の値から逆方向に移動され、その一連の値がレジスタにロードされます。

注

オフセットの制限　オフセットの値は最低で 1、最大で 33 であることが必要です。オフセットが 0、または 33 より大きい場合、命令の動作は予測不能です。各命令で、Fd と D の値によっては、オフセットについて他の制限があります。これらの制限の詳細については、個別の命令の解説を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

C5.5.5 VFP 複数ロード / ストアのアドレシングモード（別名）

ARM の複数ロード / ストアのアドレシングモードと同様に、これらのアドレシングモードはスタックへのアクセスに便利ですが、ロード（ポップ）命令とストア（プッシュ）命令では別のアドレシングモードを使用する必要があります。詳細については、P. A5-47*「複数ロード / ストアアドレシングモード（別名）」*を参照して下さい。

ARM 命令と同様に、スタック操作により適切な別のアドレシングモード名が用意されています。フル下降スタックと空上昇スタックに適した命令は、それぞれ FD および EA で示されます。

命令の非スタック名とスタック名の関係を表 C5-2 に示します。

表 C5-2 VFP 複数ロード / ストアのアドレシングモード

非スタック用のニーモニック	スタック用のニーモニック
FLDMIAD	FLDMFDD
FLDMIAS	FLDMFDS
FLDMIAX	FLDMFDX
FLDMDBD	FLDMEAD
FLDMDBS	FLDMEAS
FLDMDBX	FLDMEAX
FSTMIAD	FSTMEAD
FSTMIAS	FSTMEAS
FSTMIAX	FSTMEAX
FSTMDBD	FSTMFDD
FSTMDBS	FSTMFDS
FSTMDBX	FSTMFDX

—— 注 ——

フル上昇スタックや空下降スタックは VFP の複数ロード / ストアアドレシングモードで効率的にサポートされないため、これらのタイプのスタックに対応するニーモニックは用意されていません。これは、LDC や STC アドレシングモードがこれらのモードを効率的にサポートしていない結果です。詳細については、P. A5-49*「アドレシングモード 5 - コプロセッサのロード / ストア」*を参照して下さい。

このため、VFP アーキテクチャを使用するシステムではこれらのタイプのスタックを使用しないことをお勧めします。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

パート D

デバッグアーキテクチャ

第 D1 章
デバッグアーキテクチャの概要

本章では、デバッグアーキテクチャの概要を解説します。本章は以下のセクションから構成されています。

- *概要* : P. D1-2
- *トレース* : P. D1-4
- *デバッグと ARMv6* : P. D1-5

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D1.1 概要

ARMv6 は、デバッグ機構を組み込んだ最初のバージョンのアーキテクチャです。以前のバージョンではデバッグは付加機能であり、大部分の実装においては EmbeddedICE マクロセルの機構によりいくらかのデファクト標準が確立されていました。

従来、EmbeddedICE の完全な機能セットは、コプロセッサ 14 ベースの、*デバッグ通信チャネル*（DCC）へのソフトウェアインターフェースを例外として、外部デバッグインターフェース経由でのみ利用可能でした。DCC は*デバッグコムチャネル*や*通信チャネル*と呼ばれることもあります。DCC はデバッグモニタやアプリケーションに対して専用のアウトオブバンド情報チャネルを提供しており、このチャネルはセミホスティング機能などのサポートに使用できます。ARMv6 以前は、これらの機能はすべて実装定義でした。

ARMv6 では、コプロセッサ 14 のサポートが拡張され、次の機能が追加されています。

- *デバッグ識別レジスタ*（DIDR）
- *デバッグステータス / 制御レジスタ*（DSCR）
- ハードウェアによるブレークポイントとウォッチポイントのサポート
- DCC

このソフトウェアインターフェースに加えて、最低限の必要条件（デバッグイネーブル、要求、応答信号）をサポートする*外部デバッグインターフェース*が必須となっています。このインターフェースは、*デバッグイベント*の管理と制御に使用できます。

これを使用するには、DSCR を使用してコアを 2 つのデバッグモードのいずれかに構成する必要があります。

ホールトデバッグモード

このモードでは、デバッグイベントが発生するとシステムはデバッグステートに入ります。システムがデバッグステートにあるとき、プロセッサコアは停止し、外部デバッグインターフェースがプロセッサのコンテキストを調べ、以降の命令実行をすべて制御できます。プロセッサが停止しているため、外部システムは無視され、サービス割り込みには応答できません。

モニタデバッグモード

このモードでは、デバッグイベントの結果としてデバッグ例外が発生します。デバッグ例外は、命令の実行とデータアクセスのどちらに関連して発生したかによって、プリフェッチアボートまたはデータアボートと同じ例外ベクタによりサービスされます。

デバッグソリューションでは両方の方式を混在できます。端的な例として、*実行中システムデバッグ*（RSD）ではモニタデバッグモードを使用し、システム障害時やブート時のデバッグには代替手法としてホールトデバッグモードをサポートするような OS（または RTOS）をサポートする目的が挙げられます。これら 2 つのモードを切り替える機能は、アーキテクチャで完全にサポートされています。

コプロセッサ 14 の*ベクタキャッチレジスタ*（VCR）をプログラムして、多くの例外についてトラップ（またはキャッチ）を行い、デバッグイベントを発生できます。これを行わない場合、実行フローで通常の例外が発生します。

これらのデバッグモードが両方とも無効な場合、デバッグは単純なモニタソリューション（通常は ROM またはフラッシュベース）に制限されます。このようなモニタは、デバッグホストとの通信に UART やイーサネット接続などの標準のシステム機能を使用できます。または、DCC をホストへのアウトオブバンド通信チャネルとして使用し、システムリソースへの条件を最小にすることもできます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

これによって、ARM® の*デバッグプログラマモデル*（DPM）の基礎が形成されます。システムがますます複雑になり、高度に集積されるようになるにつれ、デバッグの重要性も増大しています。デバッグはコアのモニタから、複数のコアとシステムリソースの監視とプロファイリングに拡張されています。デバッグアーキテクチャは、ARM アーキテクチャの将来のバージョンでさらに拡張される予定です。

―――― 注 ――――

ARM で推奨される外部デバッグインターフェースは、IEEE 1149.1 *テストアクセスポート*（TAP）とバウンダリスキャンアーキテクチャ標準をベースとしたものです。ARM 仕様はこの標準のサブセットで、ARMv6 のデバッグリソースへのアクセスのみを目的としています。このため、ARM インターフェースのドキュメントではデバッグという用語が使用されています。したがって、たとえば TAP の代わりにデバッグテストアクセスポート（デバッグ TAP）への参照が使用されます。

論理デバッグ TAP ステートマシン（デバッグ TAPSM）アーキテクチャと、関連してサポートされる命令およびスキャンチェインのみが指定されています。デバッグ TAPSM 状態遷移を実行するための正確な物理インターフェースと機構は記述されておらず、必須でもありません。

――――――――

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D1.2 トレース

トレースサポートはアーキテクチャの拡張で、一般に組み込みトレースマクロセル（ETM）を使用して実装されます。ETM は、プロセッサの動作に対応するリアルタイムのトレースストリームを構築します。このトレースストリームが独立したダウンロードと解析のため組み込みトレースバッファ（ETB）にローカルに記憶されるか、トレースポート経由でトレースポートアナライザ（TPA）および関連付けられたホストベースのトレースデバッグツールに直接エクスポートされるかは実装定義です。

ETM の使用は侵蝕性ではありません。開発ツールが ETM に接続し、構成を行い、トレースのキャプチャとダウンロードを行っても、プロセッサの動作には一切影響を与えません。トレースアーキテクチャの拡張により、ランタイムシステムのより詳細な監視とデバッグが可能になります。これは、特に次のような場合に有用です。

- コアを停止すると、システムの動作に影響がある。
- 問題が検出された時点でシステムの可視なステートが不十分なため、原因を判定できない。トレースの機構により、システムのログ記録と、フォルトのバックトレースが可能になります。

トレースは、プロセッサで実行されているコードの分析、たとえばパフォーマンス解析やコードの範囲を実行するためにも使用できます。

ETM アーキテクチャについては、別のドキュメントで解説されています。ライセンス取得者とサードパーティツールベンダは、最新のバージョンについて ARM にお問合せ下さい。ETM アーキテクチャでは、次の内容が指定されています。

- ETM プログラマモデル
- 許容されるトレースプロトコルフォーマット
- 物理トレースポートコネクタ

ETM アーキテクチャのバージョンは、メジャーバージョンとマイナーバージョンにより ETMvX.Y の形式で定義されます。ここで X はメジャーバージョン番号、Y はマイナーバージョン番号です。現在のメジャーバージョン（ARMv6 に対応しています）は ETMv3 です。以前のバージョンと比較した改良点・拡張点には次のものがあります。

- より高度な圧縮を実現したトレースプロトコル形式
- FIFO が満杯に近くなったときに自動的にデータトレースを抑制し、命令トレースを継続しながらオーバフローを防止する機能
- データトレースを継続しながら、命令トレースを無効にする機能
- トレースされるプロセッサからの ETM の制御
- プロセス依存のフィルタリングとトリガ
- トレースポートとコアクロック周波数の分離

一部の機能は ETMv3 ではオプションです。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D1.3 デバッグと ARMv6

ARMv6 のデバッグアーキテクチャ定義と使用モデルは、以下の 2 つの章で定義されています。

- 第 2 章では、デバッグイベント、デバッグステート、外部デバッグインターフェース、デバッグ例外、デバッグがシステム制御コプロセッサ（CP15）に与える影響について解説します。
- 第 3 章では、コプロセッサ 14 のデバッグ機構について解説します。

D1.3.1 デバッグと仮想アドレス

特記されていない限り、以下の章で示すすべてのアドレスは、P. B4-2「*VMSA の概要*」で説明されている*仮想アドレス*（VA）です。

命令仮想アドレス（IVA）および*データ仮想アドレス*（DVA）という用語は、それぞれ命令フェッチとデータアクセスに対応する VA を意味します。IVA または DVA が仮想アドレスや修飾仮想アドレスを指す場合、本文に記載されています。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 D2 章
デバッグイベントと例外

本章では、デバッグイベントと、それに関するプロセッサの動作の概要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。

- *概要* : P. D2-2
- *モニタデバッグモード* : P. D2-5
- *ホールトデバッグモード* : P. D2-8
- *外部デバッグインターフェース* : P. D2-13

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D2.1 概要

デバッグイベントは、次のいずれかです。

- ソフトウェアデバッグイベント。詳細については、P. D2-3「*ソフトウェアデバッグイベント*」を参照して下さい。
- 外部デバッグインターフェースによりイベントが生成され、プロセッサがデバッグステートに入る。これは、次のような原因で発生します。
  - — 外部デバッグ要求信号がアクティブになる。詳細については、P. D2-13「*外部デバッグ要求信号*」を参照して下さい。
  - — デバッグステートエントリ要求コマンド。詳細については、P. D2-13「*デバッグステートエントリ要求コマンド*」を参照して下さい。

プロセッサは、デバッグイベントに対して次のいずれかの方法で応答します。

- デバッグイベントを無視する。
- デバッグステートに入る。
- デバッグ例外を取得する。

どの応答を行うかは、表 D2-1 に示すように構成によって異なります。

表 D2-1 デバッグイベント時のプロセッサの動作

DBGEN	DSCR [15:14]	デバッグモード（選択されて許可されている）	ソフトウェアデバッグイベント時の動作	EDBGRQ 時の動作	デバッグステートエントリ要求コマンド時の動作
0	0bxx	デバッグが禁止される	無視 /PAbort[a]	無視	無視
1	0b00	なし	無視 /PAbort[a]	実装定義[b]	実装定義[b]
1	0bx1	ホールト	デバッグステートエントリ	デバッグステートエントリ	デバッグステートエントリ
1	0b10	モニタ	デバッグ例外 / 無視[c]	実装定義[b]	実装定義[b]

a. デバッグが禁止されている場合、BKPT 命令は無視されず、プリフェッチアボート例外を生成します。

b. プロセッサはこれらの場合にデバッグイベントを無視するか、デバッグステートに入り、どの場合も同じ動作を行います。外部デバッグインターフェースによって **DBGEN** が使用され、デバッグが禁止されている（P. D2-13「*外部デバッグインターフェース*」参照）場合、これらのデバッグイベントは無視されます。

c. プリフェッチアボートとデータアボートベクタキャッチデバッグイベントは、モニタデバッグモードでは無視されます。リンクされていないコンテキスト ID ブレークポイントデバッグイベントは、プロセッサが特権モードで実行されており、モニタデバッグモードが選択され許可されている場合は無視されます。BVR が BCR[22] == 0b1 で IVA 比較用に設定されており、プロセッサが特権モードで実行されていて、モニタデバッグモードが選択され許可されている場合、そのリソースからのブレークポイントデバッグイベントは無視されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D2.1.1 ソフトウェアデバッグイベント

ソフトウェアデバッグイベントは、次のいずれかです。

- ウォッチポイントデバッグイベント。これは、次の場合に発生します。
  - — DVA がウォッチポイントの値と一致する。比較されるアドレスが命令の仮想アドレスか修飾仮想アドレスのどちらにするかは実装定義です。
  - — WCR のすべての条件が一致する。
  - — ウォッチポイントが許可される。
  - — リンクされている、コンテキスト ID を保持している BRP（存在する場合）が許可されており、その値が CP15 のレジスタ 13 にあるコンテキスト ID と一致する。
  - — メモリアクセスを開始した命令が、実行にコミットされている。命令が条件コードを満たす場合のみ、ウォッチポイントデバッグイベントが生成されます。
- ブレークポイントデバッグイベント。これは、次の場合に発生します。
  - — 命令がプリフェッチされ、IVA がブレークポイントの値と一致する。比較されるアドレスが命令の仮想アドレスか修飾仮想アドレスのどちらにするかは実装定義です。
  - — 命令がプリフェッチされると同時に、BCR のすべての条件が一致する。
  - — ブレークポイントが有効になっている。
  - — 命令がプリフェッチされると同時に、リンクされている、コンテキスト ID を保持している BRP（存在する場合）が許可され、その値が CP15 のレジスタ 13 にあるコンテキスト ID と一致する。
  - — 命令が実行にコミットされている。
  - — これらのデバッグイベントは、命令が条件コードを満たしても満たさなくても生成されます。
- ブレークポイントデバッグイベントは、次の場合にも発生します。
  - — 命令がプリフェッチされ、CP15 のコンテキスト ID（レジスタ 13）がブレークポイントの値と一致する。
  - — 命令がプリフェッチされると同時に、BCR のすべての条件が一致する。
  - — ブレークポイントが有効になっている。
  - — 命令が実行にコミットされている。
  - — これらのデバッグイベントは、命令が条件コードを満たしても満たさなくても生成されます。
- ソフトウェアブレークポイントデバッグイベント。これは、BKPT 命令が実行にコミットされているときに発生します。BKPTX は無条件命令です。
- ベクタキャッチデバッグイベント。これは、次の場合に発生します。
  - — 命令がプリフェッチされ、IVA がベクタロケーションアドレスと一致する。これには、例外エントリによるものだけでなく、すべての種類のプリフェッチが含まれます。比較されるアドレスは常に仮想アドレスで、修飾仮想アドレスは使用されません。
  - — 命令がプリフェッチされると同時に、VCR の対応するビットがセットされる（ベクタキャッチが許可される）。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

— 命令が実行にコミットされている。
— これらのデバッグイベントは、命令が条件コードを満たしても満たさなくても生成されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D2.2 モニタデバッグモード

デバッグ例外は、モニタデバッグモードが許可されている場合に取得され、次の場合を除いてソフトウェアデバッグイベントが発生します。

- プリフェッチアボートとデータアボートベクタについてのベクタキャッチデバッグイベント
- プロセッサが特権モードで実行されている場合の、リンクなしコンテキスト ID ブレークポイントデバッグイベント
- プロセッサが特権モードで実行されている場合の、BCR[22:21] == 0b10 でのブレークポイントデバッグイベント

これらのデバッグイベントは無視されます。これにより、プロセッサが回復不能なステートになることが回避されます。

デバッグ例外の原因がウォッチポイントデバッグイベントの場合、プロセッサは以下の動作を実行します。

- DSCR[5:2]（デバッグエントリの方式）ビットが、発生したウォッチポイントに設定されます。
- CP15 の DFSR および WFAR レジスタが、P. D2-7*「デバッグ例外によるコプロセッサ 15 のレジスタへの影響」*に説明されているように設定されます。
- 正確なデータアボート例外と同じ動作シーケンスが実行されます。これには、次の動作が含まれます。
  - — SPSR_abt が、保存された CPSR の値で更新されます。
  - — CPSR が更新され、アボートモードと ARM® ステートへの変更が行われ、通常の割り込みと不正確なアボートは禁止されます。
  - — 再開アドレスが R14_abt - 0x8 になるように、R14_abt が設定されます。
  - — PC が適切なアボートベクタに設定されます。

  詳細については、P. A2-21*「データアボート（データアクセス中に発生するメモリアボート）」*を参照して下さい。

データアボートハンドラは、DFSR または DSCR[5:2] ビットをチェックし、ルーチンへのエントリがデバッグ例外とデータアボート例外のどちらで発生したかを判断する必要があります。原因がデバッグ例外の場合、デバッグモニタに分岐する必要があります。ウォッチポイントデバッグイベントを発生した命令のアドレスは、WFAR により判定できます。R14_abt には、（再開先の命令のアドレス + 0x08）が設定されます。これは標準のデータアボートの動作です。

デバッグ例外の原因がブレークポイント、ソフトウェアブレークポイント、ベクタキャッチデバッグイベントのいずれかの場合、プロセッサは以下の動作を実行します。

- DSCR[5:2]（デバッグエントリの方式）ビットが、原因に応じた値に設定されます。
- CP15 の IFSR レジスタが、P. D2-7*「デバッグ例外によるコプロセッサ 15 のレジスタへの影響」*に説明されているように設定されます。
- プリフェッチアボート例外と同じ動作シーケンスが実行されます。これには、次の動作が含まれます。
  - — SPSR_abt が、保存された CPSR の値で更新されます。
  - — CPSR が更新され、アボートモードと ARM ステートへの変更が行われ、通常の割り込みと不正確なアボートは禁止されます。
  - — 通常のプリフェッチアボート例外に従って、R14_abt が設定されます。
  - — PC が適切なアボートベクタに設定されます。

  詳細については、P. A2-20*「プリフェッチアボート（命令フェッチ中に発生するメモリアボート）」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

プリフェッチアボートハンドラは、IFSR または DSCR[5:2] ビットをチェックし、ルーチンへのエントリがデバッグ例外とプリフェッチアボート例外のどちらで発生したかを判断する必要があります。原因がデバッグ例外の場合、デバッグモニタに分岐する必要があります。R14_abt には、(ソフトウェアデバッグイベントを発生した命令のアドレス + 0x04) が設定されます。これは標準のプリフェッチアボートの動作です。

プリフェッチアボートまたはデータアボートハンドラ内にブレークポイントまたはソフトウェアブレークポイントデバッグイベントを設定する場合、またはこれらのハンドラのいずれかによってアクセスされる可能性があるデータアドレスにウォッチポイントデバッグイベントを設定する場合は注意が必要です。デバッグイベントは、ハンドラが SPSR_abt と R14_abt を保存する前には発生できません。発生した場合、値が上書きされ、ソフトウェアの動作は予測不能です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D2.2.1 デバッグ例外によるコプロセッサ 15 のレジスタへの影響

アボート情報の記録に使用される CP15 レジスタは次の 5 つです。

FAR フォルトアドレスレジスタ
IFAR 命令フォルトアドレスレジスタ
WFAR ウォッチポイントフォルトアドレスレジスタ
IFSR 命令フォルトステータスレジスタ
DFSR データフォルトステータスレジスタ

通常の動作におけるこれらの使用モデルについては、以下を参照して下さい。

- 仮想メモリシステム（VMSA）で使用する場合、P. B4-22 表 B4-7
- 保護メモリシステム（PMSA）で使用する場合、P. B5-17 表 B5-9

モニタデバッグモードでは、ブレークポイント（CP14 によって制御される）、ソフトウェアブレークポイント（BKPT 命令）、VCR が許可されたイベントの動作は次のとおりです。

- IFSR に、プリフェッチアボートの原因が書き込まれます。
- IFAR に、プリフェッチアボートを取得する命令のアドレスが書き込まれます。
- DFSR、FAR、WFAR は変更されません。

ウォッチポイントデバッグイベントでは、動作は次のとおりです。

- IFSR は変更されません。
- DFSR に、デバッグイベントのエンコードが書き込まれます。
- FAR の値は予測不能です。
- WFAR に、ウォッチポイントの対象となるアドレスにアクセスした命令のアドレスが書き込まれます。
  - ARM ステートの場合、命令のアドレス + 8
  - Thumb® ステートの場合、命令のアドレス + 4
  - Jazelle® ステートの場合、命令のアドレス + 実装定義のオフセット

——— 注 ———

ARMv6 では、CP14 での WFAR のサポートはオプションです。

IFAR のサポートは PMSAv6 では必須で、VMSAv6 ではオプションです。

———————

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D2.3 ホールトデバッグモード

ホールトデバッグモードは、DSCR[14] の設定により構成されます。デバッグイベントが発生すると、プロセッサは*デバッグステート*と呼ばれる特別なステートに切り替わります。また、ホールトデバッグモードが構成されておらず､外部デバッグインターフェースによりデバッグが許可されている場合、プロセッサは外部デバッグ要求信号とデバッグステートエントリ要求コマンドがアクティブになることでデバッグステートに入ることもあります。詳細については、P. D2-2 *「概要」*を参照して下さい。デバッグステートでは、プロセッサは次のように動作する必要があります。

- DSCR[0]（コアがホールト中）ビットがセットされます。
- **DBGACK** 信号（P. D2-13 *「外部デバッグインターフェース」*参照）がアサートされます。
- DSCR[5:2]（デバッグエントリの方式）ビットが、P. D3-11 表 D3-6 に従ってセットされます。
- プロセッサはホールトします。パイプラインがフラッシュされ、命令がプリフェッチされなくなります。
- CPSR は変更されません。
- 割り込みは無視されます。
- DMA エンジンの実行は継続されます。外部デバッガは CP15 の操作を使用して、エンジンの停止と再開を実行できます。詳細については、P. B7-9 *「CP15 のレジスタ 11 を使用した L1 DMA 制御」*を参照して下さい。
- 例外は、P. D2-11 *「デバッグステートでの例外」*で説明されているように扱われます。
- 以降のデバッグイベントは無視されます。
  - ソフトウェアデバッグイベント
  - 外部デバッグ要求信号
  - デバッグステートエントリ要求コマンド
- 外部デバッグインターフェース経由で、プロセッサが ARM ステート命令を実行するように強制できる機構が存在する必要があります。この機構は、DSCR[13]（ARM 命令の実行イネーブル）ビットにより有効になります。

  プロセッサは、CPSR の T ビットと J ビットの実際の値にかかわらず、ARM ステートにあるものとして命令を実行します。
- CPSR を変更する命令と、分岐命令全般を除き、プロセッサはデバッグステートでどの ARM ステート命令も実行できます。分岐命令 `B`、`BL`、`BLX(1)`、`BLX(2)` は、デバッグステートでの動作は予測不能です。
- 外部デバッガは CPSR を更新するために、`MSR`、`BX`、`BXJ`、データ処理命令のみを使用する必要があります。通常ステートで CPSR を更新する他の命令はすべて、デバッグステートでは予測不能です。

  `MSR` 命令で J ビットと T ビットを直接変更した場合､結果は予測不能です｡J ビットと T ビットを変更するには､`BX` または `BXJ` 命令と､例外復帰用に設計されたデータ処理命令の暗黙的な `SPSR` から `CPSR` への移動を使用する必要があります。

  T ビットまたは J ビットがセットされている場合、`BXJ` 命令の動作は予測不能です。J ビットがセットされている場合、`BX` 命令の動作は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

外部デバッガは、T ビットの値を変更するには `BX` 命令を使用し、J ビットがクリアな場合にセットするには `BXJ` 命令を使用する必要があります。T ビットと J ビットへの他の変更を行う場合は常に、デバッガは次のようなシーケンスを実行する必要があります。

1. `CurrentModeHasSPSR()` が True であることを確認します。
2. r0、lr、SPSR を保存します。
3. CPSR に設定する値を r0 に書き込みます。
4. < 復帰アドレス > を lr に書き込みます。
5. 次のシーケンスを実行します。
```
MSR SPSR, r0
MOVS pc, lr
```
6. r0、lr、SPSR を復元します。

- 命令は、特権モードと同様に実行されます。たとえば、プロセッサがユーザモードの場合、`MSR` 命令で PSR を更新でき、すべての CP14 のデバッグ命令が実行できます。
- プロセッサは、CPSR のモードビットで示されているように、レジスタバンク、メモリ、外部コプロセッサにアクセスします。たとえば、プロセッサがユーザモードの場合、ユーザモードレジスタバンクを参照し、特権なしにメモリにアクセスします。
- PC は、*「デバッグステートでの PC の動作」*で説明されているように動作します。

外部デバッガがプロセッサをデバッグステートから復元させるために使用可能な、機構と再開コマンドが存在している必要があります。この再開コマンドは、DSCR[1]（コアが再起動された）フラグをクリアする必要があります。プロセッサが実際にデバッグステートからイグジット（終了）したとき、DSCR[1]（コアが再起動された）ビットがセットされ、DSCR[0]（コアがホールトしている）ビットと **DBGACK** 信号がクリアされる必要があります。

D2.3.1 デバッグステートでの PC の動作

デバッグステートでの PC と CPSR レジスタの動作は次のとおりです。

- PC はデバッグステートへのエントリに固定されます。つまり、ARM 命令を実行してもインクリメントされません。ただし、PC を直接変更する命令を実行した場合は内容が更新されます。
- プロセッサがデバッグステートに入った後で PC が読み出された場合、以前のステートとデバッグイベントの種類により、P. D2-10 表 D2-2 に記載されている値が返されます。
- デバッグステートの間に、PC に特定の値を書き込むシーケンスが実行され、その後でプロセッサが強制的に再起動された場合、書き込まれた値に対応するアドレスから実行が開始されます。ただし、PC に書き込みを行う前に、CPSR に復帰状態（ARM/Thumb/Jazelle）を設定する必要があります。これを行わない場合、プロセッサの動作は予測不能です。
- PC への書き込みを行わず、プロセッサが強制的に再起動された場合、再開アドレスは予測不能です。
- デバッグステートの間に PC または CPSR に書き込みが行われ、その後で PC が読み出された場合、返される値は予測不能です。CPSR は正しく読み出し可能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- PC に書き込みを行う命令で条件コードが失敗した場合、PC に書き込まれる値は予測不能です。つまり、その後でプロセッサが強制的に再起動されるか、PC が読み出された場合、結果は予測不能です。

——— 注 ———

分岐予測を使用する実装では、分岐が予測され、取得され、少し後で次のシーケンスアドレスに回復するときに、PC の変更を停止することが困難な場合があります。

———————

D2.3.2 非侵蝕性デバッグの動作

トレースやパフォーマンス監視ユニットなど、非侵蝕性のデバッグ機能が実装されている場合、プロセッサがデバッグステートにある間はこれらの機能を禁止しておく必要があります。

D2.3.3 デバッグイベントのレジスタへの影響

表 D2-2 デバッグステートへのエントリ後に読み出される PC の値

デバッグイベント	ARM	Thumb	Jazelle[a]	復帰アドレスの意味[b]
ブレークポイント	RA + 8	RA + 4	RA + オフセット	ブレークポイントの対象となる命令のアドレス
ウォッチポイント	RA + 8	RA + 4	RA + オフセット	実行を再開する命令のアドレス[c]
BKPT 命令	RA + 8	RA + 4	RA + オフセット	BKPT 命令のアドレス
ベクタキャッチ	RA + 8	RA + 4	RA + オフセット	ベクタアドレス
EDBGRQ 信号	RA + 8	RA + 4	RA + オフセット	実行を再開する命令のアドレス
デバッグステートエントリ要求コマンド	RA + 8	RA + 4	RA + オフセット	実行を再開する命令のアドレス

a. オフセットは実装定義の定数と、記載されている値です。
b. 復帰アドレスは、デバッグステートからのイグジット（終了）後にプロセッサが最初に実行する命令のアドレスです。
c. ウォッチポイントは不正確な場合があります。つまり、復帰アドレスがウォッチポイントをヒットした命令のアドレスではなく、プロセッサは何命令も後で停止している場合があります。ウォッチポイントをヒットした命令の仮想アドレスは、CP15 の WFAR にあります。

他のデータ処理およびプログラムステータスレジスタはすべて、SPSR_abt と R14_abt も含み、デバッグステートへのエントリ時に変更されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D2.3.4 デバッグイベントによるコプロセッサ 15 のレジスタへの影響

アボート情報の記録に使用される CP15 レジスタは次の 5 つです。

FAR フォルトアドレスレジスタ

IFAR 命令フォルトアドレスレジスタ

WFAR ウォッチポイントフォルトアドレスレジスタ

IFSR 命令フォルトステータスレジスタ

DFSR データフォルトステータスレジスタ

通常の動作の使用モデルは、仮想メモリシステム（VMSA）の場合は P. B4-22 表 B4-7、保護メモリシステム（PMSA）の場合は P. B5-17 表 B5-9 に示すとおりです。

ホールトデバッグモードでは、ウォッチポイントデバッグイベントにより WFAR が次のように更新されます。

- ARM ステートの場合、ウォッチポイントの対象となるアドレスにアクセスした命令の仮想アドレス + 8
- Thumb ステートの場合、ウォッチポイントの対象となるアドレスにアクセスした命令の仮想アドレス + 4
- Jazelle ステートの場合、ウォッチポイントの対象となるアドレスにアクセスした命令の仮想アドレス + 実装定義のオフセット

IFSR、DFSR、FAR、IFAR、SPSR_abt、R14_abt はすべて、デバッグステートへのエントリ時に変更されません。

——— 注 ———

ARMv6 では、CP14 での WFAR のサポートはオプションです。

IFAR のサポートは PMSAv6 では必須で、VMSAv6 ではオプションです。

———————

D2.3.5 デバッグステートでの割り込み

デバッグステートでの割り込みは、CPSR の I ビットと F ビットの値にかかわらず無視されます。I ビットと F ビットは、デバッグステートへのエントリにより変更されません。

D2.3.6 デバッグステートでの例外

リセット、プリフェッチ、デバッグ、SWI、未定義の各例外は、デバッグステートでは次のように処理されます。

リセット 通常のプロセッサステート（ARM/Thumb/Jazelle）と同様に取得されます。システムリセットの結果として、プロセッサはデバッグステートからイグジット（を終了）します。

プリフェッチアボート
デバッグステートでは命令はプリフェッチされないため、プリフェッチアボートは発生しません。

デバッグ デバッグステートではソフトウェアデバッグイベントは無視されるため、デバッグは発生しません。

SWI デバッグステートで `SWI` を実行した場合、動作は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

未定義 Jazelle 状態とデバッグステートで未定義命令を実行した場合、動作は予測不能です。プロセッサが ARM または Thumb 状態の場合、*「デバッグステートでの未定義またはデータアボート例外の取得」*で定義されているように例外が取得されます。

データアボート

デバッグステートで、メモリシステムによりメモリアボートが通知された場合、*「デバッグステートでの未定義またはデータアボート例外の取得」*に定義されているようにデータアボート例外が取得されます。

デバッグステートでの未定義またはデータアボート例外の取得

デバッグステートで例外が取得された場合、プロセッサは次のように動作する必要があります。

- PC、CPSR、SPSR_<例外モード>は、通常のプロセッサステートでの例外エントリと同様の方法で設定されます。例外が不正確なデータアボートで、PC に書き込みが行われていない場合、R14_abt は通常のプロセッサステートの例外エントリと同様に設定されます。他のすべての場合、R14_<例外モード>に設定される値は予測不能です。
- プロセッサはデバッグステートのままで、例外ベクタをプリフェッチしません。

また、例外がデータアボートの場合は次の動作が行われます。

- DFSR と FAR は、通常のプロセッサステートでの例外エントリと同様の方法で設定されます。WFAR に設定される値は予測不能です。IFSR は変更されません。
- DSCR[6]（スティッキーな正確アボートビット）または DSCR[7]（スティッキーな不正確アボートビット）がセットされます。
- DSCR[5:2]（デバッグエントリの方式）ビットが、発生した D- サイドアボート（b0110）に設定されます。

一部のデータアボートは不正確なため、外部デバッガの動作の結果ではなく、デバッグ対象であるソフトウェアにより発生したデバッグステートへのエントリ後にメモリエラーが発生する可能性があります。このため、外部デバッガはプロセッサのステートを検査する前に、データ同期バリア命令を発行する必要があります。このようなメモリエラーがデバッグステートで発生した場合、不正確なアボートがトリガされ、その後でデバッガにより読み出されるプロセッサステートに反映されます。デバッグステートからのイグジット（終了）時に、ソフトウェアは適切なデータアボートベクタから実行を再開します。

プロセッサが例外ハンドラを実行しているときに発生したデバッグイベントの処理には、注意が必要です。デバッガは、デバッグステートで未定義例外が取得される可能性がある操作を実行する前に、SPSR_und と R14_und の値を保存する必要があります。また、デバッガはデバッグステートでデータアボートを発生する可能性がある操作を実行する前に、SPSR_abt と R14_abt、および DFSR、FAR、WFAR レジスタの値を保存する必要があります。これを行わない場合、値が上書きされる可能性があり、ソフトウェアの動作が予測不能になります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D2.4 外部デバッグインターフェース

外部デバッグインターフェースは、プロセッサのデバッグリソースにアクセスするために外部デバッガが使用する、任意の外部信号のセットです。ARM プロセッサの従来の外部デバッグインターフェースは、IEEE1149.1 標準に基づいていました。しかし、ARMv6 の実装では、以下の条件が満たされていれば、他の任意のインターフェースを使用できます。

選択されているインターフェースにかかわらず、次の信号が必須です。

DBGACK デバッグ応答信号。プロセッサは、システムがデバッグステートに入ったことを示すため、この出力信号をアサートします。デバッグステートの定義については、P. D2-8 *「ホールトデバッグモード」*を参照して下さい。

DBGEN デバッグイネーブル信号。この入力信号が LOW（デバッグ禁止）の場合、プロセッサは DSCR[15:14] が 0b00 である場合と同様に動作します。詳細については、P. D3-10 *「レジスタ 1: デバッグステータス / 制御レジスタ（DSCR）」*を参照して下さい。この信号が LOW の場合、外部デバッグ要求信号とデバッグステートエントリ要求コマンドは無視されます。

EDBGRQ 外部デバッグ要求信号。P. D2-9 *「デバッグステートでの PC の動作」*で説明されているように、この入力信号はデバッグロジックがホールトデバッグモードである場合、プロセッサを強制的にデバッグステートに移行します。

ARM の外部デバッグインターフェースの推奨仕様である ARM デバッグインターフェースは、このマニュアルとは別のドキュメントに解説されています。

D2.4.1 外部デバッグ要求信号

ARMv6 準拠のプロセッサは、外部デバッグ要求入力信号を搭載している必要があります。このタイプの要求が発生した場合、プロセッサはデバッグステートに入ります。この場合、DSCR[5:2]（デバッグエントリの方式）ビットは 0b0100 に設定されます。

この信号は ETM によって駆動し、プロセッサにトリガを通知できます。たとえば、プロセッサがホールトデバッグモードにあり、メモリアクセス許可フォルトが発生した場合、外部のトレースアナライザで、プロセッサが停止すると同時にこのトリガイベントに関するトレース情報を収集できます。

D2.4.2 デバッグステートエントリ要求コマンド

外部デバッグインターフェースには、プロセッサを強制的にデバッグステートに移行できる機構が存在する必要があります。この場合、DSCR[5:2]（デバッグエントリの方式）ビットは 0b0000 に設定されます。

外部デバッグインターフェースが ARM デバッグインターフェースに準拠している場合、この機構は IR 命令です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

第 D3 章
コプロセッサ 14: デバッグコプロセッサ

本章では、コプロセッサ 14: デバッグコプロセッサ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。

- *コプロセッサ 14 のデバッグレジスタ* : P. D3-2
- *コプロセッサ 14 のデバッグ命令* : P. D3-5
- *デバッグレジスタリファレンス* : P. D3-8
- *CP14 のデバッグレジスタのリセット時の値* : P. D3-24
- *外部デバッグインターフェースから CP14 のデバッグレジスタへのアクセス* : P. D3-25

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.1 コプロセッサ 14 のデバッグレジスタ

CP14 のデバッグレジスタの説明を表 D3-1 に示します。

CP14 のデバッグレジスタにアクセスするには、opcode_1 と CRn に 0 を設定します。コプロセッサ命令の opcode_2 と CRm フィールドは、CP14 デバッグレジスタ番号のエンコードに使用されます。ここでレジスタ番号は {opcode2, CRm} です。

表 D3-1 CP14 デバッグレジスタのマップ

バイナリアドレス		レジスタ番号	省略形[a]	CP14 デバッグレジスタ名
Opcode_2	CRm			
000	0000	0	DIDR	デバッグ ID レジスタ
000	0001	1	DSCR	デバッグステータス / 制御レジスタ
000	0010-0100	2-4	-	予約
000	0101	5	DTR	データ転送レジスタ
000	0110	6	WFAR	ウォッチポイントフォルトアドレスレジスタ[b]
000	0111	7	VCR	ベクタキャッチレジスタ
000	1000-1111	8-15	-	予約
001-011	0000-1111	16-63	-	予約
100	0000-1111	64-79	BVRy	ブレークポイント値レジスタ / 予約
101	0000-1111	80-95	BCRy	ブレークポイント制御レジスタ / 予約
110	0000-1111	96-111	WVRy	ウォッチポイント値レジスタ / 予約
111	0000-1111	112-127	WCRy	ウォッチポイント制御レジスタ / 予約

a. y はバイナリ番号 CRm の 10 進表現です。
b. WFAR は CP15 レジスタで、ARMv6 では推奨されません。ツールの将来の互換性のため、WFAR もこの CP14 ロケーションにデコードすることをお勧めします。

ブレークポイントデバッグイベントを設定するには、*ブレークポイント値レジスタ*（BVR）と*ブレークポイント制御レジスタ*（BCR）の 2 つのレジスタが必要です。BCRy は、BVRy に対応する制御レジスタです。一組のブレークポイントレジスタ BVRy/BCRy は、*ブレークポイントレジスタペア*（BRP）と呼ばれます。BVR と BRP は命令アドレスとともにロードされ、その後でその内容がプロセッサの IVA と比較されます。比較されるアドレスが命令の仮想アドレスと修飾仮想アドレスのどちらかは実装定義です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

同様に、一組のウォッチポイントレジスタ WVRy/WCRy は、*ウォッチポイントレジスタペア*（WRP）と呼ばれます。WVR と WRP はデータアドレスとともにロードされ（P. D1-5 *「デバッグと仮想アドレス」* 参照）、その後でその内容がプロセッサの DVA と比較されます。比較されるアドレスがメモリアクセスの仮想アドレスと修飾仮想アドレスのどちらかは実装定義です。

アドレスマップには、最大 16 個までの BRP と 16 個までの WRP のスペースが予約されています。

ARMv6 に準拠したプロセッサには、スレッド対応のブレークポイントとウォッチポイントのサポートが搭載されています。コンテキスト ID は BVR にロードでき、BCR は BVR の値を IVA バスではなく CP15 コンテキスト ID（レジスタ 13）と比較するように構成可能です。その後で、命令アドレスやデータアドレスがロードされている別のレジスタペアを、コンテキスト ID を保持している BRP とリンクできます。ブレークポイント / ウォッチポイントのデバッグイベントは、アドレスとコンテキスト ID の両方が同時に一致する場合のみ生成されます。この方法で、タスク内の特定のスレッドをデバッグするときに不要なヒットを回避できます。

コンテキスト ID 一致のみで生成されるブレークポイントデバッグイベントもサポートされています。ただし、プロセッサが特権モードで実行中にこの一致が発生し、プロセッサがモニタデバッグモードに構成されている場合、一致は無視されます。これにより、プロセッサが回復不能な状況になることが回避されます。

すべての BRP がコンテキスト ID 比較機能を持つことは必須ではありません。ARMv6 に準拠したプロセッサでは、次のものを実装可能です。

- 2 個から 16 個までの任意の数の BRP。この場合、DIDR[27:24] = n かつ $n >= 1$ で、サポートされる BRP の数は $n + 1$ になります。これは、コンテキスト ID 機能を持つものと持たないものの両方を含む BRP の総数です。
- 1 個から 16 個までの任意の数の WRP。この場合、DIDR[31:28] = n で、サポートされる WRP の数は $n + 1$ になります。
- 1 つから BRP の実装数まで任意の数の、コンテキスト ID 比較機能を持つ BRP。この場合、DIDR[23:20] = n で、コンテキスト ID 機能付き BRP の数は $n + 1$ になります。

実装されていないレジスタは予約されているため、他の目的には使用できません。

実装されているレジスタのペアの番号は、次に示すとおりになる必要があります。

- 実装されている BRP の番号は 0 で始まります。たとえば、6 つの BRP が実装されている場合（DIDR[27:24] = 0b0101）、番号は BRP0 から BRP5 までになります。
- 実装されている WRP の番号も 0 で始まります。たとえば、2 つの WRP が実装されている場合（DIDR[31:28] = 0b0001）、番号は WRP0 から WRP1 までになります。
- コンテキスト ID 比較機能を持つ BRP は、上位の BRP 番号を占めます。たとえば、6 つの BRP が実装されており、そのうち 2 つがコンテキスト ID 比較機能を持つ場合、番号は BRP4 と BRP5 です（DIDR[27:24] = 0b0101、DIDR[23:20] = 0b0001）。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.1.1 BRP と WRP の最少数

実装には、最低 2 つの BRP と 1 つの WRP が含まれている必要があります。最低 1 つの BRP は、コンテキスト ID 比較機能を持っている必要があります。

これによって、最低でも以下の動作を実行可能なことが保証されます。

- IVA にリンクなしブレークポイントを設定する
- コンテキスト ID 値にリンクなしブレークポイントを設定する
- リンク付きブレークポイントを設定する
- リンクなしウォッチポイントを設定する
- リンク付きウォッチポイントを設定する

ただし、ARM® では最低 3 つの BRP と 1 つの WRP を実装し、最低 1 つの BRP にコンテキスト ID 比較機能を持たせることを推奨します。追加の BRP はシングルステップ専用とする（アプリケーションが停止した次の命令を指す）こともでき、この場合他のリソースはこれらのデバッグイベントのうち任意のものを自由にプログラムできます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.2 コプロセッサ 14 のデバッグ命令

このセクションでは、実装が必要なすべての CP14 デバッグ命令について説明します。

正当な CP14 デバッグ命令を表 D3-2 に示します。

表 D3-2 正当なコプロセッサ 14 のデバッグ命令

バイナリアドレス		レジスタ番号	省略形	正当な命令[a]
Opcode_2	CRm			
000	0000	0	DIDR	MRC p14,0,Rd,c0,c0,0
000	0001	1	DSCR	MRC p14,0,Rd,c0,c1,0 MRC p14,0,R15,c0,c1,0 MCR p14,0,Rd,c0,c1,0
000	0101	5	DTR	MRC p14,0,Rd,c0,c5,0 MCR p14,0,Rd,c0,c5,0 STC p14,c5,<addressing_mode> LDC p14,c5,<addressing_mode>
000	0110	6	WFAR	MRC p14,0,Rd,c0,c6,0 MCR p14,0,Rd,c0,c6,0
000	0111	7	VCR	MRC p14,0,Rd,c0,c7,0 MCR p14,0,Rd,c0,c7,0
100	0000-1111	64-79	BVR	MRC p14,0,Rd,c0,CRm,4 MCR p14,0,Rd,c0,CRm,4
101	0000-1111	80-95	BCR	MRC p14,0,Rd,c0,CRm,5 MCR p14,0,Rd,c0,CRm,5
110	0000-1111	96-111	WVR	MRC p14,0,Rd,c0,CRm,6 MCR p14,0,Rd,c0,CRm,6
111	0000-1111	112-127	WCR	MRC p14,0,Rd,c0,CRm,7 MCR p14,0,Rd,c0,CRm,7

a. Rd は任意の汎用 ARM レジスタ（R0-R14）です。

表 D3-2 で、MRC p14,0,Rd,c0,c5,0 と STC p14,c5,<adressing mode> は rDTR を、MCR p14,0,Rd,c0,c5,0 と LDC p14,c5,<adressing mode> は wDTR を指しています。詳細については、P. D3-14 *「レジスタ 5: データ転送レジスタ（DTR）」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.2.1 DSCR フラグの CPSR フラグへの転送

命令 `MRC p14,0,R15,c0,c1,0` は、CPSR フラグを次のように設定します。

- N フラグ = DSCR[31] これは予測不能な値です。
- Z フラグ = DSCR[30] これは、rDTRfull フラグの値です。
- C フラグ = DSCR[29] これは、wDTRfull フラグの値です。
- V フラグ = DSCR[28] これは予測不能な値です。

CPSR フラグは、次の条件付き命令の制御に使用できます。

D3.2.2 コプロセッサ 14 のデバッグ命令の実行

CP14 のデバッグ命令を実行した結果を表 D3-3 に示します。

表 D3-3 CP14 のデバッグ命令の実行結果

プロセッサモード	デバッグステート	DSCR[15:14]（デバッグモードが有効で選択されている）	DSCR[12]（DCC ユーザアクセスが禁止）	DIDR 読み出し、DSCR 読み出し、DTR 読み出し / 書き込み	DSCR 書き込み	読み出し / 書き込み 他のレジスタ
x	はい	xx	x	実行	実行	実行
ユーザ	いいえ	xx	0	実行	未定義例外	未定義例外
ユーザ	いいえ	xx	1	未定義例外	未定義例外	未定義例外
特権	いいえ	00（なし）	x	実行	実行	未定義例外
特権	いいえ	01（ホールト中）	x	実行	実行	未定義例外
特権	いいえ	10（モニタ）	x	実行	実行	実行
特権	いいえ	11（ホールト中）	x	実行	実行	未定義例外

未実装

プロセッサが P. D3-5 表 D3-2 にない CP14 のデバッグ命令を実行した場合、または未実装 BVR など予約されているレジスタを対象とした場合、未定義命令例外が取得されます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

デバッグリセット（外部デバッグインターフェース）

デバッグリセットが有効なときにプロセッサが CP14 のデバッグ命令を実行すると、この命令は正常に進行します。ただし、次の動作が行われます。

- CP14 のデバッグレジスタを読み出す CP14 のデバッグ命令は、これらのレジスタのリセット時の値を返します。詳細については、P. D3-24「*CP14 のデバッグレジスタのリセット時の値*」を参照して下さい。
- CP14 のデバッグレジスタに書き込む CP14 のデバッグ命令は無視されます。この命令は実行されますが、デバッグリセットが有効なため、CP14 のデバッグレジスタはリセット時の値に保たれます。

特権

デバッグ通信チャネルへのアクセス（DIDR 読み出し、DSCR 読み出し、DTR 読み出し / 書き込み）はユーザモードで実行可能な必要があります。他の CP14 デバッグ命令はすべて特権モード専用です。プロセッサがユーザモードでこれらの命令を実行すると、未定義命令例外が取得されます。

DSCR[12] の、デバッグ通信チャネルへのユーザモードアクセスのディセーブルビットがセットされている場合、すべての CP14 のデバッグ命令は特権命令として扱われ、ユーザモードから CP14 のデバッグレジスタへのアクセスはすべて未定義命令例外を生成します。

デバッグステート

プロセッサがデバッグステートにある場合（P. D2-8「*ホールトデバッグモード*」参照）、CP14 のデバッグ命令はすべて、プロセッサのモードに関係なく実行可能です。

DSCR[15:14] ビットの値

DSCR[14] ビットが設定されている（ホールトデバッグモードが選択され、許可されている）場合、プロセッサで実行されているソフトウェアが DIDR、DSCR、DTR 以外のレジスタにアクセスを試みると、プロセッサは未定義命令例外を取得します。プロセッサがモニタまたはホールトデバッグモードにない（DSCR[15:14] = 0b00）場合も同じ動作が行われます。

このロックアウト機構によって、プロセッサで実行中のソフトウェアが、外部デバッガよりプログラムされているデバッグイベントの設定を変更できないことが保証されます。

D3.2.3 CP14 のデバッグ命令の同期

プログラムで、明示的なメモリ操作の後にある CP14 のデバッグレジスタへの変更はすべて、前のメモリ操作に影響を与えないことが保証されます。

CP14 のレジスタへの変更が、以後の命令に対して可視となることが保証されるのは、PrefetchFlush 操作の実行後か、例外の取得または例外からの復帰後のみです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.3 デバッグレジスタリファレンス

このセクションでは、次のコードと用語が使用されます。

R 読み出し専用。書き込まれた値は無視されます。書き込む値は 0、または同じプロセッサの同じフィールドから以前に読み出したのと同じ値にする必要があります。

W 書き込み専用。このビットを読み出した場合、返される値は予測不能です。

RW 読み出し / 書き込み用。

C 読み出し時クリア。レジスタが読み出されるたびに値がクリアされます。

UNP/SBZP 予測不能 / 常に 0 または状態保持。このビットを読み出した場合、返される値は予測不能です。書き込む値は 0、または同じプロセッサの同じフィールドから以前に読み出したのと同じ値にする必要があります。これらのビットは通常、将来の拡張のため予約されています。

RAZ 読み出し値ゼロ。このビットを読み出した場合、0 が返されます。

コアビュー この列は、指定されたビットのコアアクセス許可を定義しています。

外部ビュー この列は、指定のビットがどのような外部デバッガビューを持つ必要があるかを定義しています。

読み出し / 書き込み属性

コアと外部デバッガビューが同一の場合に使用されます。

- 予測不能

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.3.1 レジスタ 0: デバッグ ID レジスタ（DIDR）

デバッグ ID レジスタの配置を表 D3-4 に示します。

表 D3-4 デバッグ ID レジスタのビット定義

ビット	コアビュー	外部ビュー	値	説明[a]
[3:0]	R	R		実装定義のリビジョン番号。この番号は、修正時にインクリメントされます。
[7:4]	R	R		実装定義のバリエーション番号。この番号は、機能の変更時にインクリメントされます。
[15:8]	UNP/SBZP	UNP/SBZP		予約
[19:16]	R	R		デバッグアーキテクチャのバージョン :
			0x1	このマニュアルで説明されているデバッグアーキテクチャ。
[23:20]	R	R		コンテキスト ID 比較機能を持つ、実装されている BRP の数。
			0000	1 つの BRP がコンテキスト ID 比較機能を持っています。
			0001	2 つの BRP がコンテキスト ID 比較機能を持っています。
			...	
			1111	16 個の BRP がコンテキスト ID 比較機能を持っています。
[27:24]	R	R		実装されている BRP の数。
			0000	予約（BRP の最少数は 2 です）
			0001	2 つの BRP が実装されています。
			0010	3 つの BRP が実装されています。
			...	
			1111	16 個の BRP が実装されています。
[31:28]	R	R		実装されている WRP の数。
			0000	1 つの WRP が実装されています。
			0001	2 つの WRP が実装されています。
			...	
			1111	16 個の WRP が実装されています。

a. BRP: ブレークポイントレジスタペア、WRP: ウォッチポイントレジスタペア

このレジスタのフィールドの値は実装定義です。ただし、選択された値は CP15 のメイン ID レジスタの値と一致している必要があります。

- メイン ID レジスタのビット [3:0] は、DIDR[3:0] と等しい必要があります。
- メイン ID レジスタのビット [23:20] は、DIDR[7:4] と等しい必要があります。

メイン ID レジスタの説明については、P. B3-7「*レジスタ 0: ID コード*」を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.3.2 レジスタ 1: デバッグステータス / 制御レジスタ（DSCR）

デバッグステータス / 制御レジスタの配置を表 D3-5 に示します。

表 D3-5 デバッグステータス / 制御レジスタのビット定義

ビット	コアビュー	外部ビュー	リセット時の値	説明[a]
0	R	R	a	コアがホールト中
1	R	R	a	コアが再起動された
[5:2]	RW	R	-	デバッグエントリの方式
6	R	RC	0	スティッキーな正確アボートビット
7	R	RC	0	スティッキーな不正確アボートビット
[9:8]	UNP/SBZP	UNP/SBZP	-	予約
10	R	RW	0	DbgAck: デバッグ応答
11	R	RW	0	割り込み禁止
12	RW	R	0	コムチャネルへのユーザモードアクセス禁止
13	R	RW	0	ARM 命令の実行許可
14	R	RW	0	ホールト / モニタデバッグモードの選択
15	RW	R	0	モニタデバッグモード許可
[28:16]	UNP/SBZP	UNP/SBZP	-	予約
29	R	R	0	wDTRfull: wDTR レジスタフル
30	R	R	0	rDTRfull: rDTR レジスタフル
31	UNP/SBZP	UNP/SBZP	-	予約

a. 詳細については、P. D3-24「*CP14 のデバッグレジスタのリセット時の値*」を参照して下さい。

コアがホールト中 : ビット [0]

デバッグイベントのプログラム後に、外部デバッガはこのビットが 1 にセットされるまでポーリングを行い、プロセッサがデバッグステートに入ったことを確認する必要があります。デバッグステートの定義については、P. D2-8「*ホールトデバッグモード*」を参照して下さい。

0 プロセッサは通常ステートです。

1 プロセッサはデバッグステートです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

コアが再起動した：ビット [1]

プロセッサのデバッグステートを解除した後で、外部デバッガはこのビットが 1 にセットされるまでポーリングを行い、イグジット（終了）コマンドが実行されてプロセッサがデバッグステートからイグジット（終了）したことを確認する必要があります。DSCR[0] が 0 にセットされるまでポーリングを行うのは安全ではありません。これは、外部デバッガが DSCR をサンプリングする前に、プロセッサが別のデバッグイベントによりデバッグステートに再度入る可能性があるためです。デバッグステートの定義については、P. D2-8 *「ホールトデバッグモード」* を参照して下さい。

0 プロセッサはデバッグステートからイグジット（終了）中です。

1 プロセッサはデバッグステートからイグジット（終了）しました。

デバッグエントリの方式：ビット [5:2]

デバッグエントリ方式のビットの値が示す意味を表 D3-6 に示します。

表 D3-6 デバッグエントリ方式のビットの値の意味

値	説明
0000	デバッグステートエントリ要求コマンドが発生した
0001	ブレークポイントが発生した
0010	ウォッチポイントが発生した
0011	BKPT 命令が実行された
0100	外部デバッグ要求シグナルがアクティブになった
0101	ベクタキャッチが発生した
0110	D- サイドアボートが発生した
0111	I- サイドアボートが発生した
1xxx	予約

これらのビットは、次の内容を示します。

- プリフェッチまたはデータアボートベクタにジャンプした原因
- デバッグステートに入った原因

これによって、プリフェッチアボートやデータアボートのハンドラは、デバッグモニタにジャンプするべきかどうかを判断できます。同時に、外部デバッガやデバッグモニタは、デバッグステート / デバッグ例外エントリを発生したデバッグイベントを特定できます。

具体例は次のようなものです。

- 通常ステートでデバッグも禁止されている状態で BKPT 命令が実行された場合でも、このフィールドは *BKPT 命令が実行された*に設定され、IFSR（P. B4-19 *「フォルトアドレスレジスタとフォルトステータスレジスタ」* 参照）はデバッグイベントを示す値に設定されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

スティッキーな正確アボート : ビット [6]

このフラグは、外部デバッグインターフェースを使用してプロセッサに対して発行された命令により生成された、正確なデータアボートの検出に使用されます。詳細については、P. D3-25 *「外部デバッグインターフェースから CP14 のデバッグレジスタへのアクセス」*を参照して下さい。DSCR[13] の ARM 命令実行イネーブルビットがクリアされている場合、またはコアがデバッグステートでない場合、スティッキーな正確アボートフラグの値は予測不能です。このフラグは、外部デバッガによる DSCR の読み出し時にクリアされます。

0 このビットが最後にクリアされて以降に、正確なデータアボート例外が発生していません。

1 このビットが最後にクリアされて以降に、正確なデータアボート例外が発生しました。

スティッキーな不正確アボート : ビット [7]

このフラグは、外部デバッグインターフェースを使用してプロセッサに対して発行された命令により生成または取得された、不正確なデータアボートの検出に使用されます。詳細については、P. D3-25 *「外部デバッグインターフェースから CP14 のデバッグレジスタへのアクセス」*を参照して下さい。DSCR[13] の ARM 命令実行イネーブルビットがクリアされている場合、またはコアがデバッグステートでない場合、スティッキーな不正確アボートフラグの値は予測不能です。このフラグは、外部デバッガによる DSCR の読み出し時にクリアされます。

0 このビットが最後にクリアされて以降に、不正確なデータアボート例外が発生していません。

1 このビットが最後にクリアされて以降に、不正確なデータアボート例外が発生しました。

DbgAck: ビット [10]

このビットがセットされている場合、**DBGACK** 出力信号（P. D2-13 *「外部デバッグインターフェース」*参照）は、プロセッサのステートにかかわらず HIGH に固定されます。

外部デバッガが、デバッグ処理の一部として通常ステートで一連のコードを実行する必要があり、かつシステムの他の部分はプロセッサがデバッグステートとして動作する必要がある場合、外部デバッガでこのビットを 1 にセットする必要があります。

割り込み無効 : ビット [11]

このビットがアサートされている場合、IRQ と FIQ の入力信号は禁止されます。

0 割り込みが許可されています。

1 割り込みが禁止されています。

外部デバッガが、デバッグ処理の一部として通常ステートで一連のコードを実行する必要があり、かつそのコードに割り込みが発生しない必要がある場合、外部デバッガでこのビットを 1 にセットする必要があります。

例として、OS サービスルーチンを実行してページをディスクからメモリにページインし、アプリケーションに戻ってこのステート変更の影響を確認する必要がある場合が挙げられます。このルーチンの実行中に、どのような割り込みもサービスされることは望ましくありません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

コムチャネルへのユーザモードアクセス禁止：ビット [12]

このビットがセットされている場合、ユーザモードのプロセスが DIDR、DSCR、DTR のいずれかにアクセスを試みた場合、未定義命令例外が取得されます。

このビットをセットすると、ユーザモードプロセスはどの CP14 のデバッグレジスタにもアクセスできなくなります。

0 コムチャネルへのユーザモードアクセスが許可されています。

1 コムチャネルへのユーザモードアクセスが禁止されています。

ARM 命令の実行許可：ビット [13]

0 禁止。

1 コアが外部デバッグインターフェース経由で、ARM 命令をデバッグステートで強制的に実行するようにする機構が許可されます。外部デバッグインターフェースにそのような機構がない場合、このビットからは常に 0 が読み出され、書き込みは無視されます。

コアがデバッグステートでない場合にこのビットをセットした場合の動作は、予測不能です。

ホールト / モニタデバッグモードの選択：ビット [14]

0 モニタデバッグモードが選択されています。

1 ホールトデバッグモードが選択され、許可されています。

モニタデバッグモードイネーブル：ビット [15]

0 モニタデバッグモードが禁止されています。

1 モニタデバッグモードが許可されています。

——— 注 ———

- モニタデバッグモードでコアがデバッグ例外を取得するには、選択と許可の両方が行われている（ビット 14 がクリアで、ビット 15 がセットされている）必要があります。
- 外部インターフェース入力 **DBGEN** が LOW の場合、DSCR[15:14] からは 0b00 が読み出されます。プログラムされた値は、**DBGEN** が HIGH になるまでマスクされ、その時点で値が読み出され、動作はプログラムされた値に戻ります。

———————

wDTRfull: wDTR レジスタフル：ビット [29]

0 wDTR レジスタは空です。

1 wDTR レジスタはフルです。

このフラグは、外部デバッガが wDTR を読み出したときに自動的にクリアされ、コアが同じレジスタに書き込んだときに自動的にセットされます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

rDTRfull: rDTR レジスタフル : ビット [30]

0 rDTR レジスタは空です。

1 rDTR レジスタはフルです。

このフラグは、外部デバッガが rDTR に書き込んだときに自動的にセットされ、コアが同じレジスタを読み出したときに自動的にクリアされます。rDTRfull フラグがセットされている場合、rDTR への書き込みは許可されません。

D3.3.3 レジスタ 5: データ転送レジスタ（DTR）

このレジスタは、*読み取り専用 DRT*（rDTR）と*書き込み専用 DTR*（wDTR）という 2 つの独立した物理レジスタで構成されます。読み出しと書き込みはコアから見たものである点に注意して下さい。rDTR は MRC または STC 命令によりアクセスされ、wDRT は MCR または LDC 命令によりアクセスされます。詳細については、P. D3-5 表 D3-2 を参照して下さい。より詳細な説明については、表 D3-7 を参照して下さい。

表 D3-7 データ転送レジスタのビット定義

ビット	コアビュー	外部ビュー	リセット時の値	説明
31:0	R	W	-	読み出し専用データ転送レジスタ
31:0	W	R	-	書き込み専用データ転送レジスタ

これらのレジスタの用法と、rDTRfull および wDTRfull フラグの関連の詳細については、P. D3-10 *「レジスタ 1: デバッグステータス / 制御レジスタ（DSCR）」*を参照して下さい。

これらのレジスタの組は、wDTRfull および rDTRfull フラグとともに、デバッグコムチャネルのプロセッサからのビューになります。

D3.3.4 レジスタ 6: ウォッチポイントフォルトアドレスレジスタ（WFAR）

WFAR には、すべてのウォッチポイントデバッグイベント時に、フォルトの発生した命令の仮想アドレスが書き込まれます。

表 D3-8 ウォッチポイントフォルトアドレスレジスタのビット定義

ビット	コアビュー	外部ビュー	リセット時の値	説明
31:0	RW	-	-	ウォッチポイント。フォルトの発生した命令のアドレス。

WFAR は、CP15 経由でもアクセスできます。詳細については、P. B4-44 *「レジスタ 6: フォルトアドレスレジスタ」*を参照して下さい。CP15 経由のアクセスは、ARMv6 では推奨されません。CP14 レジスタ経由のアクセスを推奨します。

WFAR への CP14 アクセスはオプションですが、推奨です。ARMv6 の初期の実装の一部には、この機能が含まれていません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.3.5 レジスタ 7: ベクタキャッチレジスタ(VCR)

レジスタ 7 はベクタキャッチレジスタ(VCR)です。詳細については、表 D3-9 を参照して下さい。VCR のビットがセットされている場合、対応するベクタがプリフェッチされ、命令の実行がコミットされた場合に、デバッグ例外またはデバッグステートエントリが生成されることがあります(DSCR[15:14] ビットの値に依存します。詳細については、P. D3-7 *「DSCR[15:14] ビットの値」*を参照して下さい)。

完全な詳細については、P. D2-3 *「ソフトウェアデバッグイベント」*を参照して下さい。

——— 注 ———

このモデルでは、例外エントリだけでなく、例外ベクタのどのような種類のプリフェッチでもベクタキャッチがトリガされる可能性があります。

表 D3-9 データキャッチレジスタのビット定義

ビット	読み出し / 書き込み属性	リセット時の値	説明	通常アドレス	ハイベクタアドレス
0	RW	0	ベクタキャッチイネーブル - リセット	0x00000000	0xFFFF0000
1	RW	0	ベクタキャッチイネーブル - 未定義命令	0x00000004	0xFFFF0004
2	RW	0	ベクタキャッチイネーブル - SWI	0xC0000008	0xFFFF0008
3	RW	0	ベクタキャッチイネーブル - プリフェッチアボート	0x0000000C	0xFFFF000C
4	RW	0	ベクタキャッチイネーブル - データアボート	0x00000010	0xFFFF0010
5	UNP/SBZP	-	予約	-	-
6	RW	0	ベクタキャッチイネーブル - IRQ	最近の IRQ アドレス	最近の IRQ アドレス
7	RW	0	ベクタキャッチイネーブル - FIQ	最近の FIQ アドレス	最近の FIQ アドレス
31:8	UNP/SBZP	-	予約	-	-

VCR ビット [6] がセットされている場合、デバッグロジックはデバッグが有効になっていた(DSCR[15:14] = 2b00)間に発生した最近の IRQ 割り込みに対応するベクタアドレスをキャッチします。これによって、標準(0x00000018)、ハイベクタ(0xFFFF0018)、ベクタ割り込みについて、P. A2-16 *「例外」*に説明されているように割り込みアドレスが確実にキャプチャできることが保証されます。VCR ビット [7] と FIQ 割り込みにも同じ関係が適用されます。

VCR の更新が、以後の命令に対して可視となることが保証されるのは、PrefetchFlush 操作の実行後か、例外の取得または例外からの復帰後のみです。詳細については、P. D3-7 *「CP14 のデバッグ命令の同期」*を参照して下さい。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.3.6 レジスタ 64 〜 79: ブレークポイント値レジスタ（BVR）

各 BVR は、BCR レジスタに関連付けられています。BVR0 は BCR0 に、BVR1 は BCR1 に、以下同様に BVR15 は BCR15 に関連付けられています。これらは、P. D3-2 *「コプロセッサ 14 のデバッグレジスタ」*で定義されている BVRy と BCRy です。一組のブレークポイントレジスタ BVRy/BCRy は、ブレークポイントレジスタペア（BRP）と呼ばれます。

このレジスタに保持されているブレークポイントの値は、IVA またはコンテキスト ID に対応しています。ブレークポイントは、IVA、コンテキスト ID、IVA/ コンテキスト ID のペアのいずれかに設定できます。3 番目の場合、2 つの BRP がそれぞれに対応する BCR を使用してリンクされる必要があります。デバッグイベントは、IVA とコンテキスト ID のペアが同時に一致する場合に生成されます。

——— 注 ———

BVR でのコンテキスト ID 比較はサポートされていない場合があります。詳細については、P. D3-2 *「コプロセッサ 14 のデバッグレジスタ」*を参照して下さい。

表 D3-10 ブレークポイント値レジスタのビット定義

ビット	読み出し / 書き込み属性	リセット時の値	説明
31:0	RW	-（予測不能）	ブレークポイントの値

BVR[1:0] の定義は、使用法に依存します。BVP がブレークポイント（IVA）比較としてセットアップされている場合、BVR[1:0] は RAZ/SBZP として定義されています。BVP がコンテキスト ID 比較としてセットアップされている場合、BVR[1:0] は有効で、比較の一部として使用されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.3.7 レジスタ 80 ～ 95: ブレークポイント制御レジスタ（BCR）

ブレークポイント制御レジスタの配置を表 D3-11 に示します。

表 D3-11 ブレークポイント制御レジスタのビット定義

ビット	読み出し / 書き込み属性	リセット時の値	説明[a]
[0]	RW	0	ブレークポイントが許可されているかどうかを示します。0: 禁止。1: 許可。
[2:1]	RW	-	スーパバイザアクセス。詳細については、*「スーパバイザアクセス : ビット [2:1]」*を参照して下さい。
[4:3]	UNP/SBZP	-	予約
[8:5]	RW	-	バイトアドレス選択。詳細については、P. D3-18 *「バイトアドレス選択 : ビット [8:5]」*を参照して下さい。
[15:9]	UNP/SBZP	-	予約
[19:16]	RW	-	リンク付き BRP 番号。詳細については、P. D3-19 *「リンク付き BRP 番号 : ビット [19:16]」*を参照して下さい。
[20]	RW	-	リンクイネーブル。詳細については、P. D3-19 *「リンクイネーブル : ビット [20]」*を参照して下さい。
[22:21]	RW[b]	- (-)	BVR の意味。詳細については、P. D3-19 *「BVR の意味 : ビット [22:21]」*を参照して下さい。
[31:23]	UNP/SBZP	-	予約

a. それぞれの詳細については、以下のサブセクションを参照して下さい。
b. ビット [21] は常に 0 が読み出されることがあります。詳細については、P. D3-19 *「BVR の意味 : ビット [22:21]」*を参照して下さい。

スーパバイザアクセス : ビット [2:1]

ブレークポイントの条件を、行われるアクセスの特権に関して指定できます。

00 予約
01 特権
10 ユーザ
11 特権とユーザの両方

この BRP がコンテキスト ID 比較とリンク用にプログラムされている場合（BCR[21:20] == 0b11）、IVA を保持する BRP の BCR[2:1] フィールドが優先されます。

この BRP にリンクされている WRP の WCR[2:1] フィールドも、このフィールドよりも優先されます。

どちらの場合も、この BRP の BCR[2:1] フィールドは*特権とユーザの両方*にプログラムされている必要があります。

この BRP が IVA 比較用にプログラムされている（BCR[21] == 0b0）、またはリンクされていないコンテキスト ID 比較用にプログラムされている（BCR[21:20] == 0b10）場合、比較が成功したかどうかにかかわらず、スーパバイザアクセス条件に適合しないとブレークポイントはヒットしません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

バイトアドレス選択 : ビット [8:5]

BVR にはワードアドレスがプログラムされています。このフィールドを使用して、特定のバイトアドレスが実行にコミットされた場合のみヒットするようにブレークポイントをプログラムできます。

0000 ブレークポイントは一切ヒットしません。

xxx1 ブレークポイントは、アドレス (BVR AND 0xFFFFFFFC) + 0 のバイトが実行にコミットされている場合のみヒットします。

xx1x ブレークポイントは、アドレス (BVR AND 0xFFFFFFFC) + 1 のバイトが実行にコミットされている場合のみヒットします。

x1xx ブレークポイントは、アドレス (BVR AND 0xFFFFFFFC) + 2 のバイトが実行にコミットされている場合のみヒットします。

1xxx ブレークポイントは、アドレス (BVR AND +0xFFFFFFFC) + 3 のバイトが実行にコミットされている場合のみヒットします。

1111 ブレークポイントは、アドレス (BVR AND 0xFFFFFFFC) で始まる 4 バイトのいずれかが実行にコミットされている場合にヒットします。

この BRP がコンテキスト ID 比較用にプログラムされている（BCR[21:20] == 0b1x）場合、このフィールドは 0b1111 に設定する必要があります。それ以外の場合、ブレークポイントまたはウォッチポイントデバッグイベントが期待したように生成されない場合があります。

バイトアドレス選択比較は、IVA とブレークポイントリソースの比較の一部です。ブレークポイントが命令仮想アドレス不一致用に構成されている場合、バイトアドレス選択比較も反転されます。

0000 ブレークポイントは常にヒットします。

xxx1 ブレークポイントは、アドレス (BVR AND 0xFFFFFFFC) + 0 のバイトが実行にコミットされている場合にはヒットしません。

xx1x ブレークポイントは、アドレス (BVR AND 0xFFFFFFFC) + 1 のバイトが実行にコミットされている場合にはヒットしません。

x1xx ブレークポイントは、アドレス (BVR AND 0xFFFFFFFC) + 2 のバイトが実行にコミットされている場合にはヒットしません。

1xxx ブレークポイントは、アドレス (BVR AND 0xFFFFFFFC) + 3 のバイトが実行にコミットされている場合にはヒットしません。

——— 注 ———

- ARMv6 準拠のプロセッサで Jazelle® オペコードがアクセラレートされない場合、BCR[8] … BCR[7] または BCR[6] … BCR[5] となるような値を BCR[8:5] に書き込んだ場合の結果は予測不能です。
- これらのバイトアドレスはリトルエンディアンです。これによって、命令フェッチのエンディアン形式にかかわらずブレークポイントがトリガされることが保証されます。たとえば、BCR[8:5] = 0b0011 で Thumb® 命令にブレークポイントが設定されている場合、リトルエンディアンで IVA[1:0] = 0b00 またはビッグエンディアンで IVA[1:0] = 0b10 のフェッチが行われた場合にこのブレークポイントはトリガされます。
- Jazelle ステートでのブレークポイントは、ブレークポイントの対象となるアドレスがオペコードとしてフェッチされた場合のみ取得されます。オペランドへのブレークポイントは無視されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

リンク付き BRP 番号 : ビット [19:16]

ここにエンコードされるバイナリ番号は、この BRP とリンクされる別の BRP を示しています。BRP が自分自身とリンクされている場合、ブレークポイントデバッグイベントが生成されるかどうかは予測不能です。

リンクイネーブル : ビット [20]

0 リンクは禁止されています。
1 リンクは許可されています。

このビットがセットされている場合、ブレークポイントはビット [19:16] で定義されたエントリにリンクされています。

BVR の意味 : ビット [22:21]

00 命令仮想アドレス一致。対応する BVR[31:2] と、バイトアドレスセレクトビット BCR[8:5] が、IVA バスと比較されます。ブレークポイントは、これらが一致する場合のみヒットします。

01 コンテキスト ID 一致。対応する BVR が、CP15 のコンテキスト ID（レジスタ 13）と比較されます。ブレークポイントは、これらが一致する場合のみヒットします。

10 命令仮想アドレス不一致。対応する BVR[31:2] と、バイトアドレスセレクトビット BCR[8:5] が、IVA バスと比較されます。ブレークポイントは、これらが一致しない場合のみヒットします。命令アドレス不一致を選択したことによる BCR[8:5] の意味の変化の詳細については、P. D3-18 *「バイトアドレス選択 : ビット [8:5]」*を参照して下さい。命令アドレス不一致の選択は、スーパバイザアクセス制御ビットの意味には影響しません。

11 予約

この BRP にコンテキスト ID 比較機能がない場合、ビット [21] は適用されず、0 が読み出されます。

IVA 不一致機能をサポートするかどうかは実装定義です。プロセッサが IVA 不一致をサポートしていない場合、ビット [22] からは 0 が読み出されます。プロセッサが IVA 不一致をサポートしている場合、すべての BRP がそうであることが必要です。

―― 注 ――

- BRP がリンクなしコンテキスト ID 比較用に設定されている場合も、BCR[8:5] と BCR[2:1] フィールドは適用されます。
- ブレークポイントが IVA 一致または IVA 不一致用に構成されている場合、比較に仮想アドレスと修飾仮想アドレスのどちらを使用するかは実装定義です。詳細については、P. B8-3 *「修飾仮想アドレス」*を参照して下さい。
- CP15 のコンテキスト ID レジスタに対して比較可能な BRP の個数は実装定義です。これは、DIDR[23:20] フィールドにより示されます。詳細については、P. D3-9 *「レジスタ 0: デバッグ ID レジスタ（DIDR）」*を参照して下さい。
- BRP が IVA 不一致またはリンクなしコンテキスト ID 比較用に構成されており、モニタデバッグモードが選択され、プロセッサが特権モードの場合、コアが回復不能なステートになることを避けるため、BRP から生成されるブレークポイントイベントはプロセッサにより無視されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ビット [21:20] の意味

00 この BVR を IVA バスに対して比較します。この BRP は他の BRP とリンクされていません。IVA 一致時にブレークポイントデバッグイベントを生成します。

01 この BVR を IVA バスに対して比較します。この BRP は BCR[19:16] で示される BRP とリンクされています。IVA とコンテキスト ID の組み合わせの一致時に、ブレークポイントデバッグイベントを生成します。

10 この BVR を、CP15 のコンテキスト ID（レジスタ 13）に対して比較します。この BRP は他の BRP とリンクされていません。コンテキスト ID 一致時に、ブレークポイントデバッグイベントを生成します。

11 この BVR を、CP15 のコンテキスト ID（レジスタ 13）に対して比較します。この BRP は、別の BRP（BCR[21:20] = 0b01 タイプの）または WRP（WRC[20] = 0b1 の）とリンクされています。IVA/DVA とコンテキスト ID の組み合わせの一致時にブレークポイント / ウォッチポイントデバッグイベントを生成します。

ARMv6 のブレークポイントデバッグイベントの生成

ブレークポイントデバッグイベントの生成には、次の規則が適用されます。

- BVR または BCR の更新が、以後の命令に対して可視となることが保証されるのは、PrefetchFlush 操作の実行後か、例外の取得または例外からの復帰後のみです。詳細については、P. D3-7「*CP14 のデバッグ命令の同期*」を参照して下さい。
- CP15 のコンテキスト ID レジスタ 13 の更新は、対応する MCR よりも数命令後に効果が発揮されることがあります。ただし、実装では例外からの復帰の前に書き込みが発生することを保証する必要があります。これは、ユーザモードのプロセスが CPU スケジューラによってスイッチインされた場合に、最初の命令でブレークが発生可能なことを保証するためです。
- 任意の BRP（IVA を保持している）は、コンテキスト ID 機能を持つ他の任意のものとリンク可能です。数個の BRP（IVA を保持している）を、コンテキスト ID 機能を持つ同じ BRP とリンクすることも可能です。
- BRP（IVA を保持している）がコンテキスト ID の比較とリンクを行うように構成されていない BRP とリンクされている場合、ブレークポイントデバッグイベントが生成されるかどうかは予測不能です。2 番目の BRP の BCR[21:20] フィールドは、0b11 に設定する必要があります。
- BRP（IVA を保持している）が実装されていない BRP とリンクされている場合、ブレークポイントデバッグイベントが生成されるかどうかは予測不能です。
- BRP が自分自身とリンクされている場合、ブレークポイントデバッグイベントが生成されるかどうかは予測不能です。
- BRP（IVA を保持している）が別の BRP（コンテキスト ID 値を保持している）とリンクされており、両方とも許可されていない（両方の BCR[0] ビットがセットされている）場合、ブレークポイントデバッグイベントは生成されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.3.8 レジスタ 96 ～ 111: ウォッチポイント値レジスタ（WVR）

各 WVR は、WCR レジスタに関連付けられています。WVR0 は WCR0 に、WVR1 は WCR1 に、以下同様に WVR15 は WCR15 に関連付けられています。これらは、P. D3-2 表 D3-1 で定義されている WVRy と WCRy です。一組のウォッチポイントレジスタ WVRy/WCRy は、*ウォッチポイントレジスタペア*（WRP）と呼ばれます。

このレジスタに含まれているウォッチポイントの値は、常に DVA に対応しています。ウオッチポイントは DVA、または DVA とコンテキスト ID のペアに設定できます。2 番目の場合、WRP とコンテキスト ID 比較機能を持つ BRP をリンクする必要があります。詳細については、P. D3-21 *「レジスタ 112 ～ 127: ウォッチポイント制御レジスタ（WCR）」*を参照して下さい。デバッグイベントは、DVA とコンテキスト ID のペアが同時に一致する場合に生成されます。WVR ビットの詳細については、表 D3-12 を参照して下さい。

表 D3-12 ウォッチポイント値レジスタのビット定義

ビット	読み出し / 書き込み属性	リセット時の値	説明
[31:2]	RW	-	ウォッチポイント値
[1:0]	RAZ/SBZP	-	-

D3.3.9 レジスタ 112 ～ 127: ウォッチポイント制御レジスタ（WCR）

これらの各レジスタは、単純なウォッチポイントまたはリンクされたウォッチポイントを適切に設定するために必要なすべての制御ビットを含んでいます。詳細については表 D3-13 を参照して下さい。

表 D3-13 ウォッチポイント値レジスタのビット定義

ビット	読み出し / 書き込み属性	リセット時の値	説明
[0]	RW	0	ウォッチポイントが許可されているかどうか。0: ウォッチポイントが禁止されています。1: ウォッチポイントが許可されています。
[2:1]	RW	-	スーパバイザアクセス。詳細については、P. D3-22 *「スーパバイザアクセス : ビット [2:1]」*を参照して下さい。
[4:3]	RW	-	ロード / ストアアクセス。詳細については、P. D3-22 *「ロード / ストアアクセス : ビット [4:3]」*を参照して下さい。
[8:5]	RW	-	バイトアドレス選択。詳細については、P. D3-22 *「バイトアドレス選択 : ビット [8:5]」*を参照して下さい。
[15:9]	SBZ	-	予約
[19:16]	RW	-	リンク付き BRP 番号。詳細については、P. D3-17 表 D3-11 を参照して下さい。
[20]	RW	-	リンクイネーブル。詳細については、P. D3-17 表 D3-11 を参照して下さい。
[31:21]	UNP/SBZP	-	予約

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

スーパバイザアクセス：ビット [2:1]

ウォッチポイントの条件を、行われるアクセスの特権に関して指定できます。

00 予約
01 特権
10 ユーザ
11 特権とユーザの両方

ロード / ストアアクセス：ビット [4:3]

ウォッチポイントの条件を、行われるアクセスのタイプに関して指定できます。

00 予約
01 ロード
10 ストア
11 ロードとストアの両方

—— 注 ——

- `SWP` または `SWPB` は、ビット [4:3] の値が 0b01、0b10、0b11 のいずれかに設定されている場合にトリガされます。
- 排他ロード命令（`LDREX`）は、ビット [4:3] の値が 0b01 または 0b11 に設定されている場合にトリガされます。
- 排他的ストア命令（`STREX`) は、ビット [4:3] の値が 0b10 または 0b11 に設定されている場合に、ストアが成功するかどうかにかかわらずトリガされます。

バイトアドレス選択：ビット [8:5]

WVR にはワードアドレスがプログラムされています。このフィールドを使用して、特定のバイトアドレスがアクセスされた場合のみヒットするようにウォッチポイントをプログラムできます。

0000 ウォッチポイントは一切ヒットしません。
xxx1 ウォッチポイントは、アドレス (WVR AND 0xFFFFFFFC) + 0 のバイトがアクセスされた場合にヒットします。
xx1x ウォッチポイントは、アドレス (WVR AND 0xFFFFFFFC) + 1 のバイトがアクセスされた場合にヒットします。
x1xx ウォッチポイントは、アドレス (WVR AND 0xFFFFFFFC) + 2 のバイトがアクセスされた場合にヒットします。
1xxx ウォッチポイントは、アドレス (WVR AND 0xFFFFFFFC) + 3 のバイトがアクセスされた場合にヒットします。

—— 注 ——

これらのバイトアドレスはリトルエンディアンです。これによって、メモリ位置のアクセス形式にかかわらずウォッチポイントがトリガされることが保証されます。たとえば、WCR[8:5] = 0b0001 の設定によりウォッチポイントがメモリのバイトに設定され、ワードアドレスが 0x0 であれば、ウォッチポイントは次のいずれの条件でもトリガされます。

- リトルエンディアンの構成で、BE-8（バイト固定ビッグエンディアンモデル）の場合

```
LDRB r0, #0x0
```

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

- BE-32（ワード不変ビッグエンディアンモデル）の場合

```
LDRB r0, #0x3
```

比較されるアドレスが仮想アドレスと修飾仮想アドレスのどちらかは実装定義です。

リンク付き BRP 番号：ビット [19:16]

ここにエンコードされるバイナリ番号は、この WRP とリンクされる BRP を示しています。

リンクイネーブル：ビット [20]

0 リンクが禁止されています。
1 リンクが許可されています。

このビットがセットされている場合、このウォッチポイントはリンク付き BRP 番号フィールドで選択された BRP にリンクされています。

ARMv6 のウォッチポイントデバッグイベントの生成

ウォッチポイントデバッグイベントの生成には、次の規則が適用されます。

- WVRまたはWCRの更新が、以後の命令に対して可視となることが保証されるのは、PrefetchFlush操作の実行後か、例外の取得または例外からの復帰後のみです。詳細については、P. D3-7「*CP14 のデバッグ命令の同期*」を参照して下さい。
- 任意の WRP は、コンテキスト ID 比較機能を持つ任意の BRP とリンクできます。数個の BRP（IVA を保持している）と WRP を、コンテキスト ID 機能を持つ同じ BRP とリンクすることも可能です。
- WRP が、コンテキスト ID の比較とリンクを行うように構成されていない BRP とリンクされている場合、ウォッチポイントデバッグイベントが生成されるかどうかは予測不能です。BRP の BCR[21:20] フィールドは、0b11 に設定する必要があります。
- WRP が実装されていない BRP とリンクされている場合、ウォッチポイントデバッグイベントが生成されるかどうかは予測不能です。
- WRP が BRP とリンクされており、両方とも有効でない（BCR[0] と WRC[0] がセットされている）場合、ウォッチポイントデバッグイベントは生成されません。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.4 CP14 のデバッグレジスタのリセット時の値

CP14 のデバッグには、2 種類のリセット信号が関係します。

システムリセット

メインプロセッサのリセット信号のアクティブ化により生成されるリセットです。

デバッグロジックリセット

デバッグロジックに固有のリセットで、外部デバッグインターフェース経由で生成されます。

デバッグロジックリセット時には、CP14 のデバッグレジスタはすべて P. D3-10 表 D3-5、P. D3-14 表 D3-7、P. D3-15 表 D3-9、P. D3-16 表 D3-10、P. D3-17 表 D3-11、P. D3-21 表 D3-12、P. D3-21 表 D3-13 の「リセット時の値」列に記載されている値に設定されます。DSCR[30:29] と DSCR[1:0] は例外で、次のように動作します。

DSCR[1]: コアが再起動した

デバッグロジックリセットによって変更されません。

DSCR[0]: コアがホールト中

デバッグロジックリセットによって変更されません。

DSCR[29]: wDTR レジスタフル

デバッグロジックリセットによって変更されません。

DSCR[30]: rDTR レジスタフル

デバッグロジックリセットによって変更されません。

プロセッサのシステムリセット時には、すべての CP14 デバッグレジスタの値は保持されます。例外は次のとおりです。

DSCR[1]: コアが再起動した

システムリセット時には、プロセッサは強制的にデバッグステートを解除されます。このため、このフラグは常にセットされます。

DSCR[0]: コアがホールト中

システムリセット時には、プロセッサは強制的にデバッグステートを解除されます。このため、このフラグは常にクリアされます。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

D3.5 外部デバッグインターフェースから CP14 のデバッグレジスタへのアクセス

このセクションで定義されている CP14 のデバッグレジスタはすべて、外部デバッグインターフェースからアクセス可能な必要があります。これらは、プロセッサの状態（ARM/Thumb/Jazelle/ デバッグ）に関係なくアクセス可能な必要があります。

本章の表で「外部ビュー」列が定義されている部分ではすべて、これらは外部デバッグインターフェースに対して読み出し / 書き込み属性の必要があります。「読み出し / 書き込み」列のみが存在する部分では、それらはコアと外部デバッガ両方に対する属性です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

用語集

アボート 不正なメモリアクセスにより発生します。アボートは、外部メモリシステムまたは MMU により発生します。

アボートモデル

データアボート例外の発生時にプロセッサステートに発生する動作の説明。アボートモデルが異なると、ベースレジスタのライトバックを指定するロード / ストア命令に関する動作が異なります。詳細については、P. A2-21*「データアボートが発生した命令の影響」*を参照して下さい。

アドレシングモード

一般に、命令によって使用される値の生成のため、多くの異なる命令によって共有されるプロシージャを意味します。ARM アドレシングモードの 4 つでは、生成される値はメモリアドレスです。これはアドレシングモードの伝統的な役割です。5 番目のアドレシングモードは、データ処理命令でオペランドとして使用される値を生成します。

境界アラインメント

データアイテムのアドレスが、あるサイズを割り切る最大の 2 のべき乗で割り切れる場合、そのデータアイテムはそのサイズに境界アラインしている、またはアラインドと呼びます。したがって、ハーフワード、ワード、ダブルワードに境界アラインしているデータアイテムのアドレスは、それぞれ 2、4、8 の倍数です。

AL（常時）

命令が条件コードフラグの値に関係なく実行されることを示します。命令のニーモニックで条件コードが指定されていない場合、AL 条件コードが使用されます。

ALU 演算ロジックユニット（Arighmetic Logic Unit）の略。

AND ビット単位の論理積を実行します。

Arithmetic_Shift_Right

右シフトを実行し、左側の空いたビットに元の左端ビット（符号ビット）を繰り返し挿入します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ARM 命令

ARM プロセッサが実行する動作を示すワード。ARM 命令はワード境界にアラインしている必要があります。

アサート文

擬似コード内で、特定の条件が満たされたことを示すために使用されます。

代入 = で示されます。

バンクレジスタ

レジスタ番号で、対応する物理レジスタは現在のプロセッサモードにより決定されます。バンクレジスタはレジスタ R8 ～ R14 です。

ベースレジスタ

ロード / ストア命令で指定され、命令のアドレス計算の基礎として使用されるレジスタ。命令とアドレシングモードにより、ベースレジスタの値にオフセットを加算または減算して仮想アドレスが構築され、メモリに送信されます。

ベースレジスタライトバック

アドレス計算に使用されるベースレジスタに、修飾された値が書き込まれること。

ビッグエンディアンメモリ

次のような条件が満たされるメモリ。

- ワード境界アライメントのアドレスにあるバイトまたはハーフワードが、そのアドレスのワード内の最上位バイトまたはハーフワードである。
- ハーフワード境界アライメントのアドレスにあるバイトが、そのアドレスのハーフワード内の最上位バイトである。

バイナリ番号

最初に 0b が付くことで示されます。

ブロッキング

キャッシュのある範囲のアドレスについてクリーニング、無効化、または両方を行うため、キャッシュのブロック転送操作をブロックすることをブロッキング操作と呼び、この操作が進行中は、以降の命令を実行できません。

ノンブロッキング操作では、操作の完了前に以降の命令の実行が許可され、例外が発生した場合にコアに命令が通知されません。これによって、実装は正確なプロセッサステートを保持する必要なしに、ノンブロッキング操作の実行中に以後の命令を終了できます。

ブール AND

`AND` 演算子で示されます。

ブール OR

`OR` 演算子で示されます。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

桁下がり パラメータとして指定された減算で桁下がりが発生した場合（真の結果が 0 より小さく、オペランドが符号なし整数として扱われる場合）は 1 を、それ以外の場合はすべて 0 を返します。これによって、擬似コードで以前に発生した減算についてより詳細な情報が得られます。減算の繰り返しは行われません。

分岐予測 ARM 実装で、プリフェッチを行う将来の実行パスを選択すること（プリフェッチ参照）。たとえば、分岐命令の後で、分岐命令に続く命令と、分岐先の命令のどちらかを選択してプリフェッチできます。

バイト 8 ビットのデータアイテム。

バイト固定

リトルエンディアンとビッグエンディアンの操作の間で、バイトアクセスが変更されないように切り替えを行う方式。このようなエンディアン形式の切り替えでは、他のデータサイズへのアクセスは必ず影響を受けます。

キャッシュ

高速なメモリのブロックで、プロセッサがアクセスしているメモリロケーションに応じてアドレスが自動的に変更され、メモリアクセスの平均速度の高速化を目的とするもの。

キャッシュの競合

頻繁に使用されるメモリキャッシュラインで、特定のキャッシュセットを使用するものの数がセットアソシアティビティを超えること。この場合、メインメモリの動作が増大し、パフォーマンスが低下します。

キャッシュヒット

メモリアクセスでアドレス指定されたデータが既にキャッシュに存在するため、高速な処理が可能な状態。

キャッシュライン

キャッシュの記憶域の基本単位。キャッシュラインのサイズは常に 2 のべき乗（通常は 4 または 8 ワード）で、適切なメモリ境界にアラインしている必要があります。*メモリキャッシュライン*は、キャッシュラインと同じサイズと境界アライメントのメモリロケーションのブロックを指します。メモリキャッシュラインは、漠然とキャッシュラインと呼ばれることもあります。

キャッシュラインインデクス

キャッシュセットの各キャッシュラインに関連付けられた番号。各キャッシュセット内のキャッシュラインには、0 から（セットアソシアティビティ）- 1 までの番号が付けられます。

キャッシュロックダウン

ワーストケースの状況でキャッシュへのアクセスにより発生する遅延を軽減する手法。キャッシュロックダウンを使用すると、重要なコードとデータをキャッシュにロードし、それらを保持するキャッシュラインが後で再割り当てされないようにできます。これにより、以降にそれらのコードやデータに対して行われるアクセスは必ずキャッシュヒットとなり、迅速に完了することが保証されます。

キャッシュロックダウンブロック

各キャッシュセットからそれぞれ 1 つのラインで構成されます。キャッシュロックダウンは、キャッシュロックダウンブロック単位で行われます。

キャッシュミス

メモリアクセスでアドレス指定されたデータがキャッシュに存在しないため、高速に処理できない状態。

キャッシュセット

キャッシュの領域で、キャッシュヒットが発生するかどうかの判断の処理を簡素化し高速化するために分割されています。キャッシュセットの数は常に 2 のべき乗です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

キャッシュウェイ

キャッシュウェイは、各キャッシュセットからそれぞれ 1 つのキャッシュラインで構成されます。キャッシュウェイには、0 から ASSOCIATIVITY - 1 までのインデクスが付けられます。キャッシュウェイのキャッシュラインは、キャッシュウェイと同じインデクス番号を持つように選択されます。したがって、たとえばキャッシュウェイ 0 は、各キャッシュセットでインデクス 0 を持つキャッシュラインで構成され、キャッシュウェイ *n* は各キャッシュセットでインデクス *n* を持つキャッシュラインで構成されます。

呼び出し先保存レジスタ

呼び出されたプロシージャが保存する必要があるレジスタ。呼び出し先保存レジスタを保存するには、呼び出されたプロシージャは通常はそのレジスタを一切使用しないか、またはプロシージャのエントリ時にレジスタをスタックにストアし、プロシージャのイグジット（終了）時にスタックから再ロードします。

呼び出し元保存レジスタ

呼び出されたプロシージャが保存する必要がないレジスタ。呼び出し元がそれらのレジスタの値を保持する必要がある場合、呼び出し元自身がストアと再ロードを行う必要があります。

CarryFrom

パラメータとして指定された加算で桁上がりが発生した場合（真の結果が $2^{32} - 1$ より大きく、オペランドが符号なし整数として扱われる場合）は 1 を、それ以外の場合はすべて 0 を返します。これによって、擬似コードで以前に発生した加算についてより詳細な情報が得られます。加算の繰り返しは行われません。

CarryFrom16

パラメータとして指定された加算で桁上がりが発生した場合（真の結果が $2^{16} - 1$ より大きく、オペランドが符号なし整数として扱われる場合）は 1 を、それ以外の場合はすべて 0 を返します。これによって、擬似コードで以前に発生した加算についてより詳細な情報が得られます。加算の繰り返しは行われません。

case ... endcase 文

多数の実行オプションから 1 つを選択することを示します。各オプションの文の範囲は、段下げによって示されます。

ClearExclusiveByAddress(<physical_address>, <processor_id>,<size>)

すべてのプロセッサによる、アドレス `<physical_address>` を排他的アクセスとしてマークする要求をすべてクリアします。詳細については、P. A2-49 *「操作の概要」* を参照して下さい。

ClearExclusiveLocal(processor_id>)

排他的アクセスのローカルレコードをクリアします。詳細については、P. A2-49 *「操作の概要」* を参照して下さい。

コメント /* */ でくくって示されます。

条件フィールド

命令に含まれる 4 ビットフィールドで、その命令がどの条件で実行されるかを指定します。

条件付き実行

命令の実行開始時に、条件コードフラグにより対応する条件が真であることが示されている場合、命令が通常に実行されることを意味します。それ以外の場合、命令は何も実行しません。

ConditionPassed(cond)

N、Z、C、V フラグのステートが、cond 引数にエンコードされている条件を満たす場合は TRUE を、それ以外の場合はすべて FALSE を返します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

構成　リセット時またはリセットの直後に行われる設定で、通常はプログラムの実行中に変化しません。

制御ビット
プログラムステータスレジスタ（PSR）の下位 8 ビット。制御ビットは例外の発生時に変更され、ソフトウェアからはプロセッサが特権モードの場合のみ変更可能です。

CPSR　カレントプログラムステータスレジスタ。

CurrentModeHasSPSR()
現在のプロセッサモードがユーザモードまたはシステムモードではない場合に TRUE を、現在のモードがユーザモードまたはシステムモードの場合は FALSE を返します。

データキャッシュ
データのロードとストアの処理のみに使用される独立したキャッシュ。

デコードビット
ARM 命令のビット [27:20] とビット [7:4] で、実行される命令のタイプを判定するために使用される主要なビットです。

デジタル信号処理
サンプリングされ、デジタル形式に変換された信号の処理に使用される各種のアルゴリズム。このようなアルゴリズムでは、飽和演算が多用されます。

ダイレクトマップキャッシュ
1 ウェイセットアソシアティブのキャッシュ。各キャッシュセットは単一のキャッシュラインで構成されるため、キャッシュのルックアップ時は 1 つのキャッシュラインのみを選択してチェックする必要があります。

ダイレクトメモリアクセス
プロセッサによる目的のデータへのアクセスを実行せず、メインメモリに直接アクセスする動作。

ドメイン　メモリのセクション、ラージページ、スモールページの集まりで、ドメインアクセス制御レジスタ（CP15 のレジスタ 3）への書き込みによりアクセス許可を高速に切り替えることができます。

修正不可フィールド（DNM）
ソフトウェアで変更が許可されない値。DNM フィールドから読み出される値は予測不能で、同じプロセッサの同じフィールドから読み出された値のみを書き込む必要があります。

このマニュアルを通して、DNM フィールドの後に、かっこに入った RAZ や RAO などの表記がされている部分があります。これは、将来の互換性のためにこのビットの読み出し値をどのようにするべきかを実装者に示すものですが、プログラマはこの動作に依存しないようにする必要があります。

倍精度数値
2 つの 32 ビット値で構成され、これら 2 つの値はメモリ上で連続して、両方ともワード境界にアラインしている必要があります。この値は IEEE 754-1985 標準に準拠した基本倍精度浮動小数点数値として解釈されます。

ダブルワード
64 ビットのデータアイテム。ARM システムでは、ダブルワードは通常は最低でもワード境界にアラインしている必要があります。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ダブルワード境界アライメント
アドレスが 8 の倍数であることを意味します。

DSP デジタル信号処理*参照*。

エレメント
変数の可能な値の一覧で、| により区切られた値。

エンディアン形式
システムのメモリマッピング形式の要素。ビッグエンディアンとリトルエンディアン参照。

EOR ビット単位の排他的論理和を実行します。

例外 イベントの処理。たとえば、例外により外部割り込みや未定義命令の処理が実行できます。

例外モード
特定の例外が発生したときにプロセッサが移行する特権モード。

例外ベクタ
低位メモリ（ハイベクタが構成されている場合は高位メモリ）の固定アドレスにある番号の 1 つ。

ExecutingProcessor()
動作を実行しているプロセッサに対応する値を返します。詳細については、P. A2-49*「操作の概要」*を参照して下さい。

明示的アクセス
CPU で実行されるロード / ストア命令により生成されるメモリからの読み出し、またはメモリへの書き込み。L1 DMA アクセスや、ハードウェアページテーブルアクセスによって生成される読み出し / 書き込みは明示的アクセスではありません。

外部アボート
外部メモリシステムにより生成されるアボート。

フォルト MMU により生成されるアボート。

高速コンテキストスイッチ拡張機能（FCSE）
ARM メモリシステムの動作を変更し、ARM プロセッサで実行されている複数のプログラムが同じアドレス範囲を使用し、同時にそれらがメモリシステムの他の部分に対して示しているアドレスが異なることを保証する機能。

フラットアドレスマッピング
すべてのアクセスについて、物理アドレスが仮想アドレスと等しくなるマッピング。

Flush-to-Zero モード
非正規化オペランドと中間結果を 0 で置き換え、最終結果の精度に大きな影響を与えることなく一部の VFP アルゴリズムのパフォーマンスを改善する、特別な処理モード。

浮動小数点例外レジスタ
読み出し / 書き込みレジスタで、このレジスタにある 2 つのビットがシステムレベルのステータスと制御に使用されます。このレジスタの他のビットは、実装のハードウェアとソフトウェアのコンポーネントの間で例外の情報を通知するために使用可能です。この方法は実装定義です。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

浮動小数点ステータス / 制御レジスタ
読み出し / 書き込みレジスタで、浮動小数点システムのすべてのユーザレベルのステータスと制御に使用されます。

浮動小数点システム ID レジスタ
読み出し専用レジスタで、どの VFP 実装が使用されているかを示します。

for ... 文 数値の範囲に対してループを実行することを示します。ループに含まれる文の範囲は、段下げによって示されます。

FPEXC 浮動小数点例外レジスタ *参照*。

FPSCR 浮動小数点ステータス / 制御レジスタ *参照*。

FPSID 浮動小数点システム ID レジスタ *参照*。

完全アソシアティブキャッシュ
キャッシュ全体が 1 つのキャッシュセットで構成されているもの。ダイレクトマップキャッシュも参照して下さい。

汎用レジスタ
32 ビットの汎用整数レジスタ R0 〜 R15 の 1 つ。R15 はプログラムカウンタを保持しているため、多くの点で R0 〜 R14 には適用されない使用制限があることに注意して下さい。

ハーフワード
16 ビットのデータアイテム。ARM システムでは、ハーフワードは通常はハーフワード境界にアラインしています。

ハーフワード境界アライメント
アドレスが 2 の倍数であること。

16 進数 等間隔のフォントで、最初に `0x` を付けて示されます。

上位レジスタ
ARM レジスタの 8 〜 15 で、一部の Thumb 命令によりアクセス可能です。

ハイベクタ
例外ベクタを置くための代替ロケーション。ハイベクタのアドレス範囲はアドレス空間の最下部ではなく、最上位近くにあります。

if ... else if ... else 文
条件付きの文を示すために使用されます。各オプションの文の範囲は、段下げによって示されます。

IGNORE フィールド（IGN）
書き込みを無視する必要があります。

IMB 命令メモリバリア *参照*。

イミディエートとオフセットフィールド
特に記載されていない限り、符号なしです。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

イミディエート値

命令に直接エンコードされ、命令の実行時に数値データとして使用される値。多くの ARM 命令と Thumb 命令では、小さな数値を、その数値を操作する命令の中にイミディエート値としてエンコードできます。

IMP そのビットの動作が実装定義であることを示すため、図中で使用される略号。

実装定義フィールド

動作がアーキテクチャによって定義されておらず、個々の実装により定義とドキュメント化が行われることを意味します。

InAPrivilegedMode()

現在のプロセッサモードがユーザモードではない場合に TRUE を、現在のモードがユーザモードの場合に FALSE を返します。

インデクスレジスタ

一部のロード / ストア命令で指定されるレジスタ。このレジスタの値はオフセットとして、ベースレジスタの値に加算または減算して仮想アドレスが構築され、メモリに送信されます。一部のアドレシングモードでは、オプションとしてインデクスレジスタの値を加算 / 減算の前にシフトできます。

インラインリテラル

コード自身と同じ領域に保持されている、定数アドレスや他のデータアイテム。これらはコンパイラにより自動的に生成され、アセンブラコードにも出現することがあります。

命令キャッシュ

命令フェッチの処理のみに使用される独立したキャッシュ。

命令メモリバリア

以降のすべての命令が、以前の命令の影響がすべて完了した後でフェッチされ、実行されることを保証するための動作のシーケンス。詳細については、P. B2-20 *「メモリのコヒーレンシとアクセスの考慮点」*を参照して下さい。

インタワーキング

ARM コードと Thumb コードの間で分岐を許可するための手法。

IsExclusiveGlobal(<physical_address>,<processor_id>,<size>)

アドレスが排他的アクセス要求にマークされているかどうかを返します。詳細については、P. A2-49 *「操作の概要」*を参照して下さい。

IsExclusiveLocal(<physical_address>,<processor_id>,<size>)

アドレスが排他的アクセス要求にマークされているかどうかを返します。詳細については、P. A2-49 *「操作の概要」*を参照して下さい。

リトルエンディアンメモリ

次のような条件が満たされるメモリ。

- ワード境界でアラインしたアドレスにあるバイトまたはハーフワードが、そのアドレスにあるワード内の最下位バイトまたはハーフワードである。
- ハーフワード境界アライメントのアドレスにあるバイトが、そのアドレスのハーフワード内の最下位バイトである。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ロード / ストアアーキテクチャ

データ処理操作がメモリの内容に対して直接行われず、レジスタの内容に対してのみ行われるアーキテクチャ。

Logical_Shift_Left

左シフトを実行し、右側の空きになったビット位置に 0 を挿入します。<< は `Logical_Shift_Left` の省略形として使用されます。

Logical_Shift_Right

右シフトを実行し、左側の空きになったビット位置に 0 を挿入します。

ロング分岐

4GB のアドレス空間のどこへでも分岐できるロード命令の使用法。

LR（リンクレジスタ）

整数レジスタ R14。

MarkExclusiveGlobal(<physical_address>,<processor_id>,<size>)

排他的アクセスが要求されることを記録します。詳細については、P. A2-49「*操作の概要*」を参照して下さい。

MarkExclusiveLocal(<physical_address>,<processor_id>,<size>)

排他的アクセスが要求されることを記録します。詳細については、P. A2-49「*操作の概要*」を参照して下さい。

Memory[<address>,<size>]

メモリのアドレス `<address>` にある長さ `<size>` のデータアイテムを指します。データアイテムは 32 ビットにゼロ拡張されます。定義されているサイズは次のとおりです。

1: バイト

2: ハーフワード

4: ワード

ARMv6 以前は、Memory[] は `<size>` バイトの境界にアラインしていました。`<size>` 境界にアラインするため、ハーフワードアクセスでは `<address>`[0] が、ワードアクセスでは `<address>`[1:0] が無視されます。

ARMv6 では、命令の定義で特に記載されていない限り、単一のアクセスについて境界アラインしていないハーフワードやワードのアクセスがサポートされています。複数ワードへのアクセスは、ワード境界にアラインしている必要があります。

アクセスのバイトの順序は、MemoryAccess(B-bit, E-bit) で定義されます。

MemoryAccess(B-bit, E-bit)

関数 `Memory[<address>, <size>]` で使用されるバイト順序アクセスモデルを、次の表に従って定義します。

B-Bit	E-bit	Endian model
0	0	LE
0	1	BE-8
1	0	BE-32
1	1	reserved (results are UNPREDICTABLE)

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

B-bit は CP15 のレジスタ 1 にあり、P. B3-12 *「制御レジスタ」*で定義されています。E-bit は CPSR にあり、P. A2-13 *「E ビット」*と P. A2-34 *「エンディアンの構成と制御」*で定義されています。BE-32、BE-8、LE は P. A2-31 *「エンディアン形式の概要」*で定義されています。

メモリバリア

詳細については、P. B2-18 *「メモリバリア」*を参照して下さい。

メモリコヒーレンシ

データ読み出しまたは命令フェッチでメモリロケーションが読み出されたとき、取得される値が常に、そのロケーションに最後に書き込まれた値と等しいことを保証する問題。複数の可能な物理ロケーション、たとえばメインメモリ、ライトバッファ、キャッシュなどが存在する場合、この保証が困難な場合があります。

メモリ管理ユニット

メモリシステムの詳細な制御を行うユニット。ほとんどの制御は、メモリに保持されている変換テーブルによって行われます。

メモリマップ I/O

特別なメモリアドレスを使用し、そのアドレスにロード / ストアが行われたときに I/O 機能を実行する方式。

混在エンディアン

プロセッサがビッグエンディアンのデータとリトルエンディアンのデータへのアクセスを自由に混在でき、それによって大きなパフォーマンスやコードサイズへの悪影響がない場合、混在エンディアンのメモリアクセスをサポートしていると呼びます。

修飾仮想アドレス

FCSE により生成され、メモリシステムの他の部分に送信されて通常の仮想アドレスの代わりに使用されるアドレス。新しい設計では FCSE の使用は推奨されません。

MMU メモリ管理ユニット*参照*。

MVA 修飾仮想アドレス*参照*。

NaN 非数を意味し、浮動小数点数値の一種です。

neg(arg)

IEEE 754-1985 標準の付録で関数 `-x` として定義されているように、浮動小数点引数の符号ビットを反転したコピーを返します。

NaN の処理と Flush-to-Zero モードに関して、これは非浮動小数点演算として扱われます。引数が NaN 値の場合も含め、結果はすべての値について、引数をコピーして符号ビットを反転することで生成されます。この操作は、引数がシグナル NaN の場合も無効演算例外を生成*しません*。

NOT ビット単位の補数を実行します。

NotFinished(CP_number)

`CP_number` 引数で指定されたコプロセッサから現在の操作が完了していないことが通知された場合は TRUE を、操作が完了している場合は FALSE を返します。

NumberOfSetBitsIn（ビットフィールド）

ビットフィールド引数に含まれる、セットされているビットの数を計算します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

オブジェクト [開始 : 終了]

ビットフィールドが、オブジェクトの「開始」ビットから「終了」ビットまでの部分から抽出される（開始と終了のビットも含む）ことを示します。

オフセットアドレシング

メモリアドレスが、ベースレジスタの値にオフセットを加算または減算して計算されることを意味します。

命令のオプション部分

{ と } で囲まれます。

OR
ビット単位の非排他的論理和を実行します。

OverflowFrom

パラメータとして指定された加算または減算により、32 ビット符号付きのオーバフローが発生した場合、1 を返します。加算は両方のオペランドの符号（ビット [31]）が同じで、結果の符合がどちらのオペランドとも異なる場合、オーバフローを生成します。減算はオペランドの符号が異なり、最初のオペランドと結果の符合が異なる場合にオーバフローを生成します。

これによって、擬似コードで以前に発生した加算または減算についてより詳細な情報が得られます。加算または減算の繰り返しは行われません。

PC（プログラムカウンタ）

整数レジスタ R15。

PCB（プロセス制御ブロック）

複数のソフトウェアプロセスをサポートしているソフトウェアシステムの場合、PCB は各プロセスに関連付けられ、プロセスが実行中でないときにプロセスのステートを保持するデータ構造体です。

物理アドレス

メインメモリのロケーション。

予測可能な以後の実行

通常のシーケンシャルな実行と、静的にターゲットが決定される分岐の実行の任意の組み合わせによって以後に到達可能な任意の命令の実行を意味します。レジスタの値に依存するロケーションに分岐する命令（`MOV PC,LR` など）は、予測可能な以後の実行を中断します。

ポストインデクスアドレシング

メモリアドレスがベースレジスタの値であるが、ベースレジスタの値にオフセットが加算または減算され、結果がベースレジスタに書き戻されることを意味します。

プリフェッチ

命令を、それ以前の命令の実行が完了する前にメモリからフェッチする処理。プリフェッチされた命令は必ず実行されるとは限りません。

プリインデクスアドレシング

メモリアドレスがオフセットアドレシングと同じ方法で計算され、そのメモリアドレスがベースレジスタに書き戻されることを意味します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

特権モード
ユーザモード以外のすべてのプロセッサモード。メモリシステムは一般に、特権モードからのアクセスをスーパバイザアクセス許可に対してチェックし、より制限の多いユーザアクセス許可に対してはチェックしません。一部の命令の使用も、特権モードに制限されます。

プロセス ID
FCSE で、現在のプロセスがロードされているプロセスブロックを識別する 7 ビットの番号。

保護領域 位置、サイズ、その他のプロパティが保護ユニットレジスタにより定義されているメモリ範囲。

保護ユニット
レジスタにより、メモリ上にある制限された数の保護領域について単純な制御を行うハードウェアユニット。

PSR CPSR、または SPSR の 1 つ。

クワイエット NaN
NaN の一種で、ほとんどの浮動小数点演算で変更されず維持されます。

読み出し割り当てキャッシュ
データのストア時にキャッシュミスが発生した場合、データがメインメモリに書き込まれるキャッシュ。キャッシュラインは、データが読み出し / ロードされるときのにみメモリロケーションに割り当てられ、書き込み / ストアされるときには割り当てられません。

読み出し値ゼロフィールド（RAZ）
読み出した場合に 0 が読み出されるフィールド。

リード・モディファイ・ライトフィールド（RMW）
汎用レジスタに読み出され、関連するフィールドがレジスタ内で更新され、レジスタの値が書き戻されるフィールド。

予約 予約されているレジスタと命令は、特記されていない限り予測不能です。予約と記載されているビット位置は SBZP/UNP です。

RISC 縮小命令セットコンピュータ（Reduced Instruction Set Computer）。

Rotate_Right
右ローテートを実行し、右端から出た各ビットは左側に挿入されます。

丸め誤差
数値演算の丸められた結果と、演算の正確な結果との差で定義されます。

丸めモード
浮動小数点演算の正確な結果を、デスティネーションの形式で表現可能な値に丸める方法を指定します。

近似値への丸め（RN）モード
丸め結果が、丸め前の結果に最も近い表現可能な数値になることを意味します。

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

プラス無限大への丸め（RP）モード

丸め結果が、正確な結果に最も近い、かつ正確な結果以上の表現可能な数値になることを意味します。

マイナス無限大への丸め（RP）モード

丸め結果が、正確な結果に最も近い、かつ正確な結果以下の表現可能な数値になることを意味します。

ゼロへの丸め（RN）モード

丸め結果が、丸め前の結果に最も近い、かつ絶対値がより大きくない表現可能な数値になることを意味します。

飽和演算 整数演算で、結果が表現可能な最大の数値よりも大きい場合は表現可能な最大の数値に設定され、結果が表現可能な最小の数値よりも小さい場合は表現可能な最小の数値に設定されるもの。符号付き飽和演算は、DSP アルゴリズムで多用されます。これは、ARM プロセッサで使用される通常の符号付き整数演算とは大きく異なっています。通常の演算では、オーバフローした結果は $+2^{31} - 1$ から -2^{31} にラップアラウンドし、逆の場合も同様のラップアラウンドが行われます。

セキュリティホール

システムの保護をバイパスする不正な機構。

自己変更型のコード

1 つ以上の命令をメモリに書き込み、それを実行するコード。このタイプのコードは、同期を保証するための命令なしに実行できません。詳細については、P. B2-21*「メモリオーダモデル内でのキャッシュ保守操作の順序付け」*を参照して下さい。

セットアソシアティビティ

キャッシュにある各キャッシュセットのキャッシュライン数。この数は ≥1 の任意の数で、2 のべき乗には制限されません。

Shared(<Rm>)

<Rm> の仮想アドレスが共有であることを示します。詳細については、P. A2-49*「操作の概要」*を参照して下さい。

シフタオペランド

ARM データ処理命令のソースオペランドの 1 つ。イミディエート値またはレジスタです。

常に 1 のフィールド（SBO）

ソフトウェアで常に 1（ビットフィールドの場合はすべてのビットが 1）として書き込む必要のあるフィールド。1 以外の値を書き込んだ場合、結果は予測不能です。

常に 1 または保持のフィールド（SBOP）

常に 1（ビットフィールドの場合はすべてのビットが 1）を書き込む、または同じプロセッサの同じフィールドから以前に読み出したのと同じ値を書き込んで保持する必要があるフィールド。

常に 0 のフィールド（SBZ）

ソフトウェアで常に 0（ビットフィールドの場合はすべてのビットが 0）として書き込む必要のあるフィールド。0 以外の値を書き込んだ場合、結果は予測不能です。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

常に 0 または保持のフィールド（SBZP）

常に 0（ビットフィールドの場合はすべてのビットが 0）を書き込む、または同じプロセッサの同じフィールドから以前に読み出したのと同じ値を書き込んで保持する必要があるフィールド。

シグナル NaN

浮動小数点演算のオペランドにシグナル NaN が指定された場合、常に無効演算例外が発生します。シグナル NaN はデバッグ時に、一部の初期化されていない変数の使用を追跡するために使用できます。

符号付きデータタイプ

-2^{N-1} ～ $+2^{N-1} - 1$ までの範囲の整数を、2 の補数形式で表現します。

符号付きイミディエートとオフセットフィールド

特に記載されていない限り、2 の補数表記でエンコードされます。

SignedDoesSat(x,n)

x が n ビットの符号付き整数の範囲内（つまり、$-2^{(n-1)} \leq x \leq 2^{(n-1)} - 1$ の場合）は 0 を、それ以外の場合は 1 を返します。

この操作によって、擬似コードで以前に発生した SignedSat(x, n) 演算についてより詳細な情報が得られます。x または n を計算するために使用される演算はいずれも繰り返されません。

SignExtend(arg)

引数を 32 ビットに符号拡張（符号ビットを伝播）します。

SignedSat(x,n)

n ビット符号付き整数の範囲に飽和した x を返します。つまり、次の値が返されます。

- $-2x < -2^{(n-1)}$ の場合、$^{(n-1)}$
- $-2^{(n-1)} <= x <= 2^{(n-1)} - 1$ の場合、x
- $2x > 2^{(n-1)} - 1$ の場合、$^{(n-1)}$

SIMD 単一命令・複数データ処理（Single-Instruction, Multple-Data）を意味します。

単精度数値

32 ビット値で、メモリ上でワード境界にアラインしている必要があります。この値は IEEE 754-1985 標準に準拠した基本単精度浮動小数点数値として解釈されます。

SP（スタックポインタ）

整数レジスタ R13。

空間的局所性

プログラムがメモリロケーションにアクセスした後で、直後にそれに近いメモリロケーションにもアクセスする可能性が高いという性質。複数ワードのキャッシュラインを持つキャッシュは、パフォーマンス向上のためにこの性質を利用しています。

SPSR 保存プログラムステータスレジスタ。現在のプロセッサモードに関連付けられており、ユーザモードやシステムモードのように保存プログラムステータスレジスタがない場合は未定義です。

SWI ソフトウェア割り込み。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ステータスレジスタ
CPSR と SPSR *参照*。

タグビット
仮想アドレスのビット [31:L+S]。ここで、L と S はそれぞれキャッシュライン長とキャッシュのセット数の、2 を底とする対数です。キャッシュヒットは、ARM プロセッサにより指定された仮想アドレスのタグビットが、選択されたキャッシュセットの有効なラインに関連付けられたタグビットと一致する場合に発生します。

時間的局所性
プログラムがメモリロケーションにアクセスした後で、短い時間の間に同じメモリロケーションに再度アクセスする可能性が高いという性質。キャッシュはパフォーマンス向上のためにこの性質を利用しています。

等価テスト
== で示されます。

Thumb 命令
ARM プロセッサが Thumb ステートで実行する動作を示すハーフワード。Thumb 命令はハーフワード境界にアラインしている必要があります。

TLB 変換ルックアサイドバッファ *参照*。

TLB ロックダウン
特定の変換テーブルウォーク結果がアクセスされることを防止する方法。これにより、関連するメモリ領域へのアクセスによって変換テーブルウォークが発生しないことが保証されます。

TLB(<Rm>)
<Rm> の物理アドレスを返します。詳細については、P. A2-49 *「操作の概要」*を参照して下さい。

変換ルックアサイドバッファ
変換テーブルウォークの結果を保持するメモリ構造。これによって、メモリアクセスの平均コストが減少します。通常は、ARM 実装の各メモリインターフェースごとに TLB が存在します。

変換テーブル
メモリに保持されるテーブル。1KB から 1MB まで、各種のサイズのメモリ領域のプロパティを定義しています。

変換テーブルウォーク
完全な変換テーブルルックアップを行う処理。ハードウェアによって自動的に実行されます。

トラップイネーブルビット
例外のトラップ処理と非トラップ処理のどちらが選択されているかを判定します。例外のトラップ処理が選択されている場合、実現方法は実装定義です。

影響を受けないアイテム
特定の操作により変更されないアイテム。

アンアラインド
アンアラインドトランザクションとは、トランザクションのアドレスがトランザクションのエレメントのサイズに境界アラインしていない場合を指します。

アンアラインドメモリアクセス
適切にハーフワード、ワード、ダブルワード境界にアラインしていない、またはその可能性があるメモリアクセス。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

バンクなしレジスタ

すべてのプロセッサモードで同じ 32 ビット物理レジスタを指す汎用レジスタ。バンクなしレジスタはレジスタ R0 ～ R7 です。

未定義 未定義命令例外を生成する命令。未定義命令例外の詳細については、P. A2-19「*未定義命令例外*」を参照して下さい。

統合キャッシュ

命令フェッチとデータのロード / ストアの処理の両方に使用されるキャッシュ。

インデクスなしアドレシング

ベースレジスタの値が直接仮想アドレスとして使用され、オフセットの加算や減算なしにメモリに送信されるアドレシング。ほとんどのタイプのアドレシングモードで、インデクスなしアドレシングはオフセットアドレシングで、イミディエートオフセットを 0 に設定することで実行されます。ARM アドレシングモード 5（LDC 命令と STC 命令で使用されます）には明示的なインデクスなしアドレシングモードがあり、命令のオフセットフィールドを使用して追加のコプロセッサオプションを指定できます。

予測不能 命令の結果を使用できないことを意味します。予測不能な命令や結果は、セキュリティホールを示すものであってはいけません。予測不能な命令がプロセッサや、システムのいずれかの部分のホールトやハングを引き起こすことは許されません。

予測不能フィールド（UNP）

有効なデータを含まず、値が時間ごと、命令ごと、実装ごとに異なる可能性があるフィールド。

符号なしデータタイプ

0 から $+2^N - 1$ までの負でない整数を、通常のバイナリ形式で表現します。

UnsignedDoesSat(x,n)

x が n ビットの符号なし整数の範囲内（つまり、$0 \leq x < 2^n$）の場合は 0、それ以外の場合は 1 を返します。

この操作によって、擬似コードで以前に発生した UnsignedSat(x,n) 演算についてより詳細な情報が得られます。x または n を計算するために使用される演算はいずれも繰り返されません。

UnsignedSat(x,n)

n ビット符号なし整数の範囲に飽和した x を返します。つまり、次の値が返されます。

- $x < 0$ の場合、0
- $0 <= x < 2^n$ の場合、x
- $x > 2^n - 1$ の場合、$2^n - 1$

命令の変数部分

< と > で囲んで表記されます。

VFP ベクタ浮動小数点アーキテクチャ *参照*。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

ベクタ浮動小数点アーキテクチャ
ARM アーキテクチャのコプロセッサ拡張機能。単精度と倍精度の浮動小数点演算を実行します。

VFP エミュレータ
ソフトウェアのみで構成され、すべての浮動小数点演算が ARM ルーチンによりエミュレートされる実装。

仮想アドレス
ARM プロセッサにより生成されるアドレス。

while ... 文
ループを示すために使用されます。ループの文の範囲は、段下げによって示されます。

ワード 32 ビットのデータアイテム。ARM システムでは、ワードは通常はワード境界にアラインしています。

ワード境界アライメント
アドレスが 4 の倍数であることを意味します。

ワード不変
リトルエンディアンとビッグエンディアンの操作の間で、境界アラインしているワードアクセスが変更されないように切り替えを行う方式。このようなエンディアン形式の切り替えでは、他のデータサイズとアンアラインドワードへのアクセスは必ず影響を受けます。

書き込み割り当てキャッシュ
データストア時のキャッシュミスによりキャッシュラインが割り当てられ、メインメモリの内容が読み込まれ、ストアされるデータがキャッシュラインに書き込まれるキャッシュ。

ライトバックキャッシュ
ストアアクセスでキャッシュヒットが発生した場合、データがキャッシュにのみ書き込まれるキャッシュ。このため、キャッシュのデータはメインメモリのデータより新しい内容になります。このようなデータは、キャッシュラインがクリアまたは再割り当てされるときにメインメモリに書き戻されます。ライトバックキャッシュは、*コピーバックキャッシュ*とも呼ばれます。

ライトスルーキャッシュ
ストアアクセスでキャッシュヒットが発生した場合、データがキャッシュとメインメモリの両方に書き込まれるキャッシュ。これは通常、プロセッサの速度低下を避けるためライトバッファにより行われます。

ライトバッファ
メインメモリへのストアを最適化することを目的とする高速なメモリのブロック。

{msbyte,..lsbyte}
バイト連結操作で、右側のバス [msbit:lsbit] サブバス名の最下位バイトと、msbit から lsbit までのビットフィールドが連結されます。

b3...b0 小文字の b が最初に付いている場合、リトルエンディアンのバイト解釈を示します。

B3...N0 大文字の B が最初に付いている場合、ビッグエンディアンのバイト解釈を示します。

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

 Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.