IA-64
アプリケーション・デベロッパーズ・アーキテクチャ・ガイド

1999 年 5 月

資料番号: 245188J-001

intel®

本書は、市場性、他者の権利を侵害しないこと、特定目的への適合、特定の提案、仕様、サンプルから生じる保証を含むがこれに限定されないいかなる保証もなく「無保証で」提供されます。

【輸出規制に関する告知と注意事項】

本資料に掲載されている製品のうち、外国為替および外国為替管理法に定める戦略物資等または役務に該当するものについては、輸出または再輸出する場合、同法に基づく日本政府の輸出許可が必要です。また、米国産品である当社製品は日本からの輸出または再輸出に際し、原則として米国政府の事前許可が必要です。

【資料内容に関する注意事項】

・ 本ドキュメントの内容を予告なしに変更することがあります。

・インテルでは、この資料に掲載された内容について、市販製品に使用した場合の保証あるいは特別な目的に合うことの保証等は、いかなる場合についてもいたしかねます。また、このドキュメント内の誤りについても責任を負いかねる場合があります。

・インテルでは、インテル製品の内部回路以外の使用にて責任を負いません。また、外部回路の特許についても関知いたしません。

・本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。

インテルは、製品について「取引条件」で提示されている場合を除き、インテル製品の販売や使用に関して、いかなる特許または著作権の侵害をも含み、あらゆる責任を負わないものとします。

・いかなる形および方法によっても、インテルの文書による許可なく、この資料の一部またはすべてを複写することは禁じられています。

本資料の内容についてのお問い合わせは、下記までご連絡下さい。

インテル株式会社　資料販売センタ

〒 305-8603　筑波学園郵便局　私書箱 115 号

Fax：0120-478832

注文番号を記載した文書あるいは本書で参照したその他のインテル文書は 0120-478832 に FAX でお問い合わせになるか http://www.intel.co.jp にアクセスされると入手することができます。

Copyright © Intel Corporation, 1999

* 一般にブランド名または商品名は各社の商標または登録商標です。

intel®

目次

第 1 章 IA-64 アプリケーション・デベロッパーズ・アーキテクチャ・ガイドについて 1-1

1.1 本書の概要 1-1
1.2 用語の定義 1-3
1.3 関連文献 1-3

第 I 部 IA-64 アプリケーション・アーキテクチャ・ガイド

第 2 章 IA-64 プロセッサ・アーキテクチャの概要 2-1

2.1 IA-64 のオペレーティング環境 2-1
2.2 命令セット切り替えモデルの概要 2-2
2.3 IA-64 命令セットの特長 2-3
2.4 命令レベルの並列性 2-3
2.5 コンパイラからプロセッサへの通信 2-4
2.6 スペキュレーション 2-4
2.6.1 コントロール・スペキュレーション 2-4
2.6.2 データ・スペキュレーション 2-5
2.6.3 プレディケーション 2-6
2.7 レジスタ・スタック 2-7
2.8 分岐 2-7
2.9 レジスタ・ローテーション 2-7
2.10 浮動小数点アーキテクチャ 2-8
2.11 マルチメディアのサポート 2-8

第 3 章 IA-64 の実行環境 3-1

3.1 アプリケーション・レジスタの状態 3-1
3.1.1 予約レジスタと無視レジスタ 3-2
3.1.2 汎用レジスタ 3-3
3.1.3 浮動小数点レジスタ 3-5
3.1.4 プレディケート・レジスタ 3-5
3.1.5 分岐レジスタ 3-5
3.1.6 命令ポインタ 3-6
3.1.7 現在のフレーム・マーカ 3-6
3.1.8 アプリケーション・レジスタ 3-7
3.1.9 パフォーマンス・モニタ・データ・レジスタ (PMD) 3-12
3.1.10 ユーザ・マスク (UM) 3-12
3.1.11 プロセッサ識別レジスタ 3-13
3.2 メモリ 3-15
3.2.1 アプリケーションでのメモリ・アドレス指定モデル 3-16
3.2.2 アドレス指定可能なユニットとアライメント 3-16
3.2.3 バイト・オーダー 3-16
3.3 命令のエンコーディングの概要 3-18

3.4 命令シーケンス 3-20

第 4 章 IA-64 アプリケーション・プログラミング・モデル 4-1

4.1 レジスタ・スタック 4-1
4.1.1 レジスタ・スタック・オペレーション 4-2
4.1.2 レジスタ・スタック命令 4-4
4.2 整数演算命令 4-5
4.2.1 算術演算命令 4-5
4.2.2 論理命令 4-6
4.2.3 32 ビットのアドレスおよび整数 4-6
4.2.4 ビット・フィールド命令およびシフト命令 4-7
4.2.5 ラージ定数 4-8
4.3 比較命令とプレディケーション 4-8
4.3.1 プレディケーション 4-9
4.3.2 比較命令 4-9
4.3.3 比較タイプ 4-10
4.3.4 プレディケート・レジスタの転送 4-12
4.4 メモリ・アクセス命令 4-12
4.4.1 ロード命令 4-14
4.4.2 ストア命令 4-15
4.4.3 セマフォ命令 4-15
4.4.4 コントロール・スペキュレーション 4-16
4.4.5 データ・スペキュレーション 4-20
4.4.6 メモリ階層の制御と整合性 4-26
4.4.7 メモリ・アクセスの順序 4-30
4.5 分岐命令 4-32
4.5.1 モジュロ・スケジュール型ループのサポート 4-33
4.5.2 分岐予測ヒント 4-36
4.6 マルチメディア命令 4-37
4.6.1 並列算術演算 4-37
4.6.2 並列シフト 4-39
4.6.3 データ配列 4-40
4.7 レジスタ・ファイルの転送 4-40
4.8 文字列とポピュレーション・カウント 4-42
4.8.1 文字列 4-42
4.8.2 ポピュレーション・カウント 4-42

第 5 章 IA-64 浮動小数点プログラミング・モデル 5-1

5.1 データ型および形式 5-1
5.1.1 実数型 5-1
5.1.2 浮動小数点レジスタ形式 5-2
5.1.3 浮動小数点レジスタ値の表現 5-3
5.2 浮動小数点ステータス・レジスタ 5-6
5.3 浮動小数点命令 5-10
5.3.1 メモリ・アクセス命令 5-10
5.3.2 浮動小数点レジスタと汎用レジスタとの間の転送命令 5-15
5.3.3 算術命令 5-16

5.3.4 **非算術命令** ...5-18
5.3.5 **浮動小数点ステータス・レジスタ** (FPSR) **のステータス・フィールド命令** ...5-20
5.3.6 **整数積和命令** ...5-20
5.4 IEEE **についての補足事項** ...5-21
5.4.1 SNaN、QNaN **および** NaN **の伝播の定義** ...5-21
5.4.2 **ソフトウェアに委ねられる** IEEE **標準演算** ...5-21
5.4.3 IEEE **標準にない演算** ...5-21

第 6 章 IA-64 システム環境での IA-32 アプリケーション実行モデル ...6-1

6.1 **命令セット・モード** ...6-1
6.1.1 IA-64 **命令セットの実行** ...6-2
6.1.2 IA-32 **命令セットの実行** ...6-3
6.1.3 **命令セットの移行** ...6-3
6.1.4 IA-32 **動作モードの移行** ...6-4
6.2 IA-32 **アプリケーション・レジスタの状態モデル** ...6-5
6.2.1 IA-32 **汎用レジスタ** ...6-10
6.2.2 IA-32 **命令ポインタ** ...6-10
6.2.3 IA-32 **セグメント・レジスタ** ...6-10
6.2.4 IA-32 **アプリケーション** EFLAG **レジスタ** ...6-18
6.2.5 IA-32 **浮動小数点レジスタ** ...6-20
6.2.6 IA-32 MMX® **テクノロジ・レジスタ** ...6-27
6.2.7 IA-32 **ストリーミング** SIMD **拡張命令レジスタ** ...6-28
6.3 **メモリ・モデルの概要** ...6-28
6.3.1 **メモリ・エンディアン** ...6-29
6.3.2 IA-32 **のセグメンテーション** ...6-29
6.3.3 **自己修正コード** ...6-30
6.4 IA-32 **による** IA-64 **レジスタの使用法** ...6-31
6.4.1 IA-64 **レジスタ・スタック・エンジン** ...6-31
6.4.2 IA-64 ALAT ...6-31
6.4.3 IA-32 **命令での** IA-64 NaT/NaTVal **の応答** ...6-31

第 7 章 IA-64 命令リファレンス ...7-1

7.1 **命令リファレンス・ページに関する規則** ...7-1
7.2 **命令の説明** ...7-2

第 II 部 IA-64 最適化ガイド

第 8 章 IA-64 最適化ガイドについて ...8-1

8.1 IA-64 **最適化ガイドの概要** ...8-1

第 9 章 IA-64 プログラミングの概要 ...9-1

9.1 **概要** ...9-1
9.2 **レジスタ** ...9-1
9.3 IA-64 **命令の使用法** ...9-2
9.3.1 **書式** ...9-2
9.3.2 **並列表現** ...9-3

9.3.3 バンドルとテンプレート 9-3
9.4 メモリ・アクセスとスペキュレーション 9-4
9.4.1 機能 9-4
9.4.2 スペキュレーション 9-5
9.4.3 コントロール・スペキュレーション 9-5
9.4.4 データ・スペキュレーション 9-6
9.5 プレディケーション 9-6
9.6 IA-64 によるプロシージャ・コールのサポート 9-7
9.6.1 スタックされたレジスタ 9-7
9.6.2 レジスタ・スタック・エンジン 9-8
9.7 分岐とヒント 9-8
9.7.1 分岐命令 9-8
9.7.2 ループおよびソフトウェアによるパイプライン化 9-9
9.7.3 レジスタのローテート 9-9
9.8 要約 9-10

第 10 章 メモリ参照 10-1

10.1 概要 10-1
10.2 非スペキュレーティブなメモリ参照 10-1
10.2.1 メモリへのストア 10-1
10.2.2 メモリからのロード 10-1
10.2.3 データ・プリフェッチ・ヒント 10-2
10.3 命令の依存関係 10-2
10.3.1 コントロール依存 10-2
10.3.2 データ依存 10-3
10.4 依存性を解消するための IA-64 スペキュレーションの使用法 10-6
10.4.1 IA-64 スペキュレーション・モデル 10-6
10.4.2 IA-64 のデータ・スペキュレーションの使用法 10-7
10.4.3 IA-64 のコントロール・スペキュレーションの使用法 10-10
10.4.4 データおよびコントロールのスペキュレーションの組み合わせ 10-12
10.5 メモリ参照の最適化 10-12
10.5.1 スペキュレーションに関する考慮事項 10-13
10.5.2 データの干渉 10-14
10.5.3 コード・サイズの最適化 10-15
10.5.4 ポスト・インクリメントのロードおよびストアの使用法 10-16
10.5.5 ループの最適化 10-17
10.5.6 チェック・コードの最小化 10-18
10.6 要約 10-19

第 11 章 プレディケーション、コントロール・フロー、命令ストリーム 11-1

11.1 概要 11-1
11.2 プレディケーション 11-1
11.2.1 分岐のパフォーマンス・コスト 11-1
11.2.2 IA-64 におけるプレディケーション 11-3
11.2.3 プレディケーションによるプログラム・パフォーマンスの最適化 11-4

11.2.4 プレディケーションに関する考慮事項 11-7
11.2.5 分岐削除に関するガイドライン 11-10
11.3 コントロール・フローの最適化 11-11
11.3.1 並列比較によるクリティカル・パスの短縮 11-11
11.3.2 マルチウェイ分岐によるクリティカル・パスの短縮 11-13
11.3.3 プレディケーションによる、複数の値からの変数または レジスタの選択 11-14
11.3.4 命令ストリームのフェッチの改善 11-16
11.4 分岐とプリフェッチ・ヒント 11-17
11.5 要約 11-17

第 12 章 ソフトウェアによるパイプライン化とループのサポート 12-1

12.1 概要 12-1
12.2 ループ関連の用語と基本的なループ・サポート 12-1
12.3 ループの最適化 12-1
12.3.1 ループのアンロール 12-2
12.3.2 ソフトウェアによるパイプライン化 12-3
12.4 IA-64 のループ・サポート機能 12-5
12.4.1 レジスタ・ローテーション 12-5
12.4.2 ローテート・プレディケートの初期化に関する注意事項 12-7
12.4.3 ソフトウェア・パイプライン・ループ分岐 12-7
12.4.4 用語の確認 12-11
12.5 IA-64 におけるループの最適化 12-12
12.5.1 while ループ 12-12
12.5.2 プレディケート付き命令を含むループ 12-15
12.5.3 複数の出口を持つループ 12-15
12.5.4 ソフトウェアによるパイプライン化に関する考慮事項 12-18
12.5.5 ソフトウェアによるパイプライン化とアドバンスド・ロード 12-19
12.5.6 ソフトウェアによるパイプライン化の前のループの アンロール 12-21
12.5.7 リダクションのサポート 12-23
12.5.8 明示的なプロローグとエピローグ 12-24
12.5.9 ループ内の余分なロードの除去 12-27
12.6 要約 12-28

第 13 章 浮動小数点アプリケーション 13-1

13.1 概要 13-1
13.2 FP アプリケーションのパフォーマンスの制限要因 13-1
13.2.1 実行レイテンシ 13-1
13.2.2 実行帯域幅 13-2
13.2.3 メモリ・レイテンシ 13-2
13.2.4 メモリ帯域幅 13-3
13.3 IA-64 の浮動小数点機能 13-3
13.3.1 ラージおよびワイド浮動小数点レジスタ・セット 13-4
13.3.2 積和命令 13-7
13.3.3 ソフトウェアによる除算 / 平方根のシーケンス 13-8

13.3.4 計算モデル 13-10
13.3.5 複数のステータス・フィールド 13-10
13.3.6 その他の機能 13-11
13.3.7 メモリ・アクセス制御 13-14
13.4 要約 13-15

第 III 部 付録

A 命令シーケンスについて A-1

A.1 RAW 順序の例外 A-3
A.2 WAW 順序の例外 A-4
A.3 WAR 順序の例外 A-5

B IA-64 擬似コード関数 B-1

C IA-64 命令形式 C-1

C.1 形式の要約 C-3
C.2 A ユニット命令エンコーディング C-10
C.2.1 整数 ALU C-10
C.2.2 整数比較 C-13
C.2.3 マルチメディア C-18
C.3 ユニット命令エンコーディング C-22
C.3.1 マルチメディアおよび変数シフト C-22
C.3.2 整数シフト C-30
C.3.3 ビット・テスト C-32
C.3.4 その他の I ユニット命令 C-33
C.3.5 GR/BR 移動 C-35
C.3.6 GR/ プレディケート /IP 移動 C-36
C.3.7 GR/AR 移動 (I ユニット) C-37
C.3.8 符号拡張 / ゼロ拡張 / ゼロ・インデックス計算 C-38
C.4 M ユニット命令エンコーディング C-38
C.4.1 ロードとストア C-38
C.4.2 ライン・プリフェッチ C-59
C.4.3 セマフォ C-61
C.4.4 FR 設定 / 取得 C-63
C.4.5 スペキュレーションおよびアドバンスド・ロード・チェック C-63
C.4.6 キャッシュ / 同期 /RSE/ALAT C-65
C.4.7 GR/AR 移動 (M ユニット) C-66
C.4.8 その他の M ユニット命令 C-67
C.4.9 メモリ管理 C-68
C.5 B ユニット命令エンコーディング C-71
C.5.1 分岐 C-71
C.5.2 Nop C-77
C.5.3 その他の B ユニット命令 C-78
C.6 F ユニット命令エンコーディング C-78
C.6.1 算術演算 C-81

C.6.2 並列浮動小数点 Select C-83
C.6.3 比較と分類 C-83
C.6.4 近似 C-85
C.6.5 最小値 / 最大値と並列比較 C-86
C.6.6 マージと論理 C-87
C.6.7 変換 C-88
C.6.8 ステータス・フィールド操作 C-89
C.6.9 その他の F ユニット命令 C-90
C.7 ユニット命令エンコーディング C-90
C.7.1 その他の X ユニット命令 C-90
C.7.2 ロング型即値 $_{64}$ 移動 C-91
C.8 即値の生成 C-92

3-1 アプリケーション・レジスタのモデル 3-4
3-2 フレーム・マーカの形式 3-6
3-3 RSC の形式 3-9
3-4 BSP レジスタの形式 3-10
3-5 BSPSTORE レジスタの形式 3-10
3-6 RNAT レジスタの形式 3-10
3-7 PFS の形式 3-11
3-8 エピローグ・カウント・レジスタの形式 3-12
3-9 ユーザ・マスクの形式 3-13
3-10 CPUID レジスタ 0 と 1 - ベンダ情報 3-14
3-11 CPUID レジスタ 2 - プロセッサ・シリアル番号 3-14
3-12 CPUID レジスタ 3 - バージョン情報 3-14
3-13 CPUID レジスタ 4 - 一般的な機能ビット 3-15
3-14 リトル・エンディアン形式のロード 3-17
3-15 ビッグ・エンディアン形式のロード 3-17
3-16 バンドルの形式 3-18
4-1 プロシージャのコールおよびリターン時のレジスタ・スタックの動作 4-3
4-2 ld.c によるデータ・スペキュレーションのリカバリ 4-21
4-3 chk.a によるデータ・スペキュレーションのリカバリ 4-22
4-4 メモリ階層 4-27
4-5 メモリ階層でサポートされるアロケーション・パス 4-28
5-1 浮動小数点レジスタ形式 5-2
5-2 浮動小数点ステータス・レジスタの形式 5-6
5-3 浮動小数点ステータス・フィールドの形式 5-7
5-4 メモリから浮動小数点レジスタへのデータ変換 - 単精度 5-11
5-5 メモリから浮動小数点レジスタへのデータ変換 - 倍精度 5-12
5-6 メモリから浮動小数点レジスタへのデータ変換 - 拡張倍精度、整数、およびフィル 5-13
5-7 浮動小数点レジスタからメモリへのデータ変換 5-14
5-8 スピル / フィルおよび拡張倍精度 (80 ビット) での浮動小数点メモリ形式 5-15
6-1 IA-64 プロセッサ命令セットの移行モデル 6-2
6-2 IA-32 アプリケーション・レジスタのモデル 6-6
6-3 IA-32 汎用レジスタ (GR8 〜 GR15) 6-10
6-4 IA-32 セグメント・レジスタ・セレクタの形式 6-11
6-5 IA-32 コード / データ・セグメント・レジスタ・ディスクリプタの形式 6-11
6-6 IA-32 EFLAG レジスタ (AR24) 6-18
6-7 IA-32 浮動小数点コントロール・レジスタ (FCR) 6-23
6-8 IA-32 IA-32 浮動小数点ステータス・レジスタ (FSR) 6-23
6-9 浮動小数点データ・レジスタ (FDR) 6-26
6-10 浮動小数点命令レジスタ (FIR) 6-26
6-11 IA-32 MMX® テクノロジ・レジスタ (MM0 〜 MM7) 6-27
6-12 ストリーミング SIMD 拡張命令レジスタ (XMM0 〜 XMM7) 6-28

6-13 メモリ・アドレス指定モデル 6-29
7-1 ポインタの加算 7-4
7-2 スタック・フレーム 7-5
7-3 br.ctop および br.cexit の操作 7-13
7-4 br.wtop および br.wexit の操作 7-14
7-5 デポジットの例 7-31
7-6 抽出の例 7-33
7-7 浮動小数点マージでの符号否定操作 7-55
7-8 浮動小数点マージでの符号操作 7-55
7-9 浮動小数点マージでの符号および指数操作 7-55
7-10 浮動小数点左ミックス 7-58
7-11 浮動小数点右ミックス 7-58
7-12 浮動小数点左右ミックス 7-58
7-13 浮動小数点パック 7-69
7-14 浮動小数点並列マージでの符号否定操作 7-82
7-15 浮動小数点並列マージでの符号操作 7-82
7-16 浮動小数点並列マージでの符号および指数操作 7-83
7-17 浮動小数点スワップ 7-104
7-18 浮動小数点スワップの左否定と右否定 7-104
7-19 浮動小数点左符号拡張 7-106
7-20 浮動小数点右符号拡張 7-106
7-21 getf.exp の機能 7-108
7-22 getf.sig の機能 7-108
7-23 ミックスの例 7-126
7-24 Mux1 の操作 (8 ビット要素) 7-140
7-25 Mux2 の例 (16 ビット要素) 7-141
7-26 パックの操作 7-145
7-27 並列加算の例 7-148
7-28 並列平均の例 7-150
7-29 並列平均でのゼロから離れる方向の丸めの例 7-151
7-30 並列平均減算の例 7-153
7-31 並列比較の例 7-155
7-32 並列最大値の例 7-157
7-33 並列最小値の例 7-158
7-34 並列乗算の操作 7-159
7-35 並列乗算右シフトの操作 7-160
7-36 並列絶対差累計の例 7-163
7-37 並列右シフトの例 7-164
7-38 並列減算の例 7-170
7-39 setf.exp の機能 7-173
7-40 setf.sig の機能 7-173
7-41 ポインタの左シフトおよび加算 7-177
7-42 ペア右シフト 7-179
7-43 アンパックの操作 7-193
C-1 バンドルの形式 C-1

表

2-1 IA-64 プロセッサのオペレーティング環境 2-2
3-1 予約 / 無視レジスタおよび予約 / 無視フィールド 3-2
3-2 フレーム・マーカのフィールドの説明 3-6
3-3 アプリケーション・レジスタ 3-8
3-4 RSC のフィールドの説明 3-9
3-5 PFS のフィールドの説明 3-12
3-6 ユーザ・マスクのフィールドの説明 3-13
3-7 CPUID レジスタ 3 のフィールド 3-15
3-8 命令タイプと実行ユニット・タイプの関係 3-18
3-9 テンプレート・フィールドのエンコーディングと命令スロットのマッピング a 3-19
4-1 アーキテクチャ上で参照可能なレジスタ・スタック関連の状態 4-4
4-2 レジスタ・スタック管理命令 4-5
4-3 整数算術演算命令 4-5
4-4 整数論理命令 4-6
4-5 32 ビット・ポインタ命令および 32 ビット整数命令 4-6
4-6 ビット・フィールド命令およびシフト命令 4-8
4-7 ラージ定数生成命令 4-8
4-8 比較命令 4-9
4-9 比較タイプの機能 4-10
4-10 ソース入力に NaT が含まれる場合の比較結果 4-11
4-11 命令および利用できる比較タイプ 4-11
4-12 メモリ・アクセス命令 4-13
4-13 メモリ・アクセス関連の状態 4-16
4-14 コントロール・スペキュレーション関連の状態 4-19
4-15 コントロール・スペキュレーションに関連する命令 4-20
4-16 データ・スペキュレーション関連の状態 4-25
4-17 データ・スペキュレーションに関連する命令 4-26
4-18 各命令クラスによって指定される局所性ヒント 4-28
4-19 メモリ階層の制御命令とヒントのメカニズム 4-29
4-20 メモリ順序規則 4-31
4-21 メモリ順序命令 4-31
4-22 分岐タイプ 4-32
4-23 分岐関連の状態 4-33
4-24 分岐に関係する命令 4-33
4-25 RRB を変更する命令 4-34
4-26 分岐有無予測ヒント 4-36
4-27 シーケンシャル・プリフェッチ・ヒント 4-37
4-28 予測リソースの割り当て解除ヒント 4-37
4-29 並列演算命令 4-38
4-30 並列シフト命令 4-39
4-31 並列データ配列命令 4-40
4-32 レジスタ・ファイル転送命令 4-41
4-33 文字列サポート命令 4-42
5-1 IEEE 実数型のプロパティ 5-1

5-2 浮動小数点レジスタのエンコーディング 5-4
5-3 浮動小数点ステータス・レジスタのフィールドの説明 5-6
5-4 浮動小数点ステータス・レジスタのステータス・フィールドの説明 .. 5-7
5-5 浮動小数点の丸め制御の定義 5-9
5-6 浮動小数点演算モデルの制御の定義 5-9
5-7 浮動小数点メモリ・アクセス命令 5-10
5-8 浮動小数点レジスタ転送命令 5-16
5-9 汎用レジスタ (整数) から浮動小数点レジスタへのデータ変換 5-16
5-10 浮動小数点レジスタから汎用レジスタ (整数) へのデータ変換 5-16
5-11 浮動小数点命令のステータス・フィールドの指定子の定義 5-17
5-12 浮動小数点算術命令 5-17
5-13 浮動小数点擬似演算 5-18
5-14 非算術浮動小数点命令 5-19
5-15 FPSR ステータス・フィールド命令 5-20
5-16 整数積和命令 5-21
6-1 IA-32 アプリケーション・レジスタのマッピング 6-7
6-2 IA-32 セグメント・レジスタのフィールド 6-11
6-3 IA-32 環境の初期レジスタ状態 6-13
6-4 IA-32 環境の実行時整合性チェック 6-17
6-5 IA-32 EFLAG レジスタのフィールド 6-19
6-6 IA-32 浮動小数点レジスタのマッピング 6-20
6-7 IA-32 浮動小数点コントロール・レジスタのマッピング (FCR) 6-24
6-8 IA-32 浮動小数点ステータス・レジスタのマッピング (FSR) 6-25
7-1 命令リファレンス・ページの説明 7-1
7-2 命令リファレンス・ページの字体に関する規則 7-1
7-3 レジスタ・ファイルの表記法 7-2
7-4 C シンタックスの相違点 7-2
7-5 分岐のタイプ 7-9
7-6 分岐有無予測ヒント 7-15
7-7 シーケンシャル・プリフェッチ・ヒント 7-15
7-8 分岐キャッシュ割り当て解除ヒント 7-15
7-9 ALAT クリアのためのコンプリータ 7-19
7-10 比較タイプ 7-22
7-11 通常および unc の比較での 64 ビット比較関係 7-23
7-12 並列比較での 64 ビット比較関係 7-24
7-13 32 ビット比較での即値範囲 7-26
7-14 比較交換のメモリ・サイズ 7-28
7-15 比較交換セマフォのタイプ 7-28
7-16 czx の結果の範囲 7-30
7-17 *pc ニーモニックの指定値 7-35*
7-18 *sf* ニーモニックの値 7-35
7-19 浮動小数点クラスの関係 7-42
7-20 浮動小数点のクラス 7-42
7-21 浮動小数点の比較タイプ 7-45
7-22 浮動小数点の比較関係 7-45
7-23 フェッチおよび加算セマフォのタイプ 7-50

7-24 浮動小数点並列比較の結果 7-74
7-25 浮動小数点並列比較関係 7-74
7-26 sz コンプリータ 7-111
7-27 ロード・タイプ 7-111
7-28 ロードのヒント 7-113
7-29 fsz コンプリータ 7-115
7-30 FP ロードのタイプ 7-116
7-31 lftype のニーモニック値 7-122
7-32 lfhint のニーモニック値 7-123
7-33 間接レジスタ・ファイルのニーモニック 7-134
7-34 ８ビット要素に対する Mux による置換 7-139
7-35 パックでの飽和の上下限値 7-145
7-36 並列加算の飽和コンプリータ 7-147
7-37 並列加算における飽和の制限 7-147
7-38 並列の比較関係 7-155
7-39 pmpyshr2 のシフト・オプション 7-160
7-40 並列減算の飽和コンプリータ 7-169
7-41 並列減算での飽和の上下限値 7-169
7-42 ストアのタイプ 7-181
7-43 ストアのヒント 7-181
7-44 xsz ニーモニック値 7-186
7-45 通常および unc タイプのビット判定での関係 7-188
7-46 並列タイプのビット判定での関係 7-188
7-47 通常および unc タイプの Nat 判定での関係 7-190
7-48 並列タイプの NaT 判定での関係 7-190
7-49 メモリ交換のサイズ 7-195
B-1 擬似コード関数 B-1
C-1 命令タイプと実行ユニット・タイプの関係 C-1
C-2 テンプレート・フィールドのエンコーディングと命令スロットのマッピング C-2
C-3 メジャー・オペコードの割り当て C-3
C-4 命令形式の要約 C-5
C-5 命令フィールドのカラー・キー C-7
C-6 命令フィールド名 C-8
C-7 特殊な命令表記 C-9
C-8 整数 ALU の 2 ビット +1 ビット・オペコード拡張 C-10
C-9 整数 ALU の 4 ビット +2 ビット・オペコード拡張 C-11
C-10 整数比較オペコード拡張 C-14
C-11 整数比較即値オペコード拡張 C-14
C-12 マルチメディア ALU の 2 ビット +1 ビット・オペコード拡張 C-18
C-13 マルチメディア ALU サイズ 1 の 4 ビット +2 ビット・オペコード拡張 C-18
C-14 マルチメディア ALU サイズ 2 の 4 ビット +2 ビット・オペコード拡張 C-19
C-15 マルチメディア ALU サイズ 4 の 4 ビット +2 ビット・オペコード拡張 C-20
C-16 マルチメディアおよび変数シフトの 1 ビット・オペコード拡張 C-22

C-17 マルチメディア最大値 / 最小値 / ミックス / パック / アンパックのサイズ 1 の 2 ビット・オペコード拡張 C-23
C-18 マルチメディア乗算 / シフト / 最大値 / 最小値 / ミックス / パック / アンパックのサイズ 2 の 2 ビット・オペコード拡張 C-24
C-19 マルチメディア・シフト / ミックス / パック / アンパックのサイズ 4 の 2 ビット・オペコード拡張 C-25
C-20 変数シフトの 2 ビット・オペコード拡張 C-26
C-21 整数シフト / ビット・テスト / NaT テストの 2 ビット・オペコード拡張 C-30
C-22 デポジットのオペコード拡張 C-30
C-23 ビット・テストのオペコード拡張 C-32
C-24 その他の I ユニットの 3 ビット・オペコード拡張 C-34
C-25 その他の I ユニットの 6 ビット・オペコード拡張 C-34
C-26 整数ロード / ストア / セマフォ /FR 取得の 1 ビット・オペコード拡張 C-39
C-27 浮動小数点ロード / ストア / ペア・ロード /FR 設定の 1 ビット・オペコード拡張 C-39
C-28 整数ロード / ストアのオペコード拡張 C-40
C-29 整数ロード +Reg のオペコード拡張 C-41
C-30 整数ロード / ストア +Imm のオペコード拡張 C-42
C-31 セマフォ /FR 取得のオペコード拡張 C-43
C-32 浮動小数点ロード / ストア /Lfetch のオペコード拡張 C-44
C-33 浮動小数点ロード /Lfetch+Reg のオペコード拡張 C-45
C-34 浮動小数点ロード / ストア /Lfetch+Imm のオペコード拡張 C-46
C-35 浮動小数点ペア・ロード /SR 設定のオペコード拡張 C-47
C-36 浮動小数点ペア・ロード +Imm のオペコード拡張 C-48
C-37 ロード・ヒント・コンプリータ C-49
C-38 ストア・ヒント・コンプリータ C-49
C-39 ライン・プリフェッチ・ヒント・コンプリータ C-60
C-40 オペコード 0 のメモリ管理の 3 ビット・オペコード拡張 C-69
C-41 オペコード 0 のメモリ管理の 4 ビット +2 ビット・オペコード拡張 C-69
C-42 オペコード 1 のメモリ管理の 3 ビット・オペコード拡張 C-70
C-43 オペコード 1 のメモリ管理の 6 ビット・オペコード拡張 C-70
C-44 IP 相対分岐タイプ C-72
C-45 間接 / その他の分岐のオペコード拡張 C-73
C-46 間接分岐タイプ C-73
C-47 間接リターン分岐タイプ C-74
C-48 シーケンシャル・プリフェッチ・ヒント・コンプリータ C-74
C-49 分岐有無ヒント・コンプリータ C-74
C-50 間接コール有無ヒント・コンプリータ C-75
C-51 分岐キャッシュ割り当て解除ヒント・コンプリータ C-75
C-52 間接予測 /nop のオペコード拡張 C-77
C-53 その他の浮動小数点の 1 ビット・オペコード拡張 C-79
C-54 オペコード 0 のその他の浮動小数点の 6 ビット・オペコード拡張 .. C-79
C-55 オペコード 1 のその他の浮動小数点の 6 ビット・オペコード拡張 .. C-80
C-56 逆数近似の 1 ビット・オペコード拡張 C-80
C-57 浮動小数点ステータス・フィールド・コンプリータ C-81

C-58　**浮動小数点算術の 1 ビット・オペコード拡張** C-81
C-59　**固定小数点の積和および選択のオペコード拡張** C-81
C-60　**浮動小数点比較のオペコード拡張** C-83
C-61　**浮動小数点分類の 1 ビット・オペコード拡張** C-84
C-62　**その他の X ユニットの 3 ビット・オペコード拡張** C-90
C-63　**その他の X ユニットの 6 ビット・オペコード拡張** C-91
C-64　**ロング型移動の 1 ビット・オペコード拡張** C-92
C-65　**即値の生成** C-92

IA-64 アプリケーション・デベロッパーズ・アーキテクチャ・ガイドについて 1

IA-64 アーキテクチャは、明示的に並列化を記述した命令、プレディケーション、スペキュレーションをはじめとする多くの斬新な機能を組み合わせたユニークなアーキテクチャである。このアーキテクチャは、種々のサーバおよびワークステーション市場において増大しつづけるパフォーマンス要求に対応できるよう、すぐれたスケーラビリティを備えている。IA-64 アーキテクチャは、EPIC (Explicitly Parallel Instrucution Computing) と呼ばれる新しいプロセッサ・アーキテクチャ・テクノロジを応用した革新的な 64 ビット命令セット・アーキテクチャ (ISA) を特長としている。IA-64 アーキテクチャの主な特長は、IA-32 命令セットとの互換性である。

本書の*第 I 部「IA-64 アプリケーション・アーキテクチャ・ガイド」*では、アプリケーション・ソフトウェア開発のために公開されている IA-64 アーキテクチャについて包括的に説明する。これは、アプリケーション・レベルのリソース (レジスタなど) やアプリケーション環境に関する情報、アプリケーション (非特権) 命令の詳細説明、形式、およびエンコーディングなどである。IA-64 アーキテクチャは IA-32 命令セットとの互換性をサポートしており、これについても本書で説明している。

本書の*第 II 部「IA-64 最適化ガイド」*では、IA-64 アプリケーション・アーキテクチャについて簡単に復習した後、IA-64 アーキテクチャのいくつかの機能について説明し、これらの機能を使って高度に最適化されたコードを生成する方法について詳しく述べる。各章では、IA-64 のそれぞれの機能を使ってパフォーマンス上の障害を軽減または排除する方法について説明する。

システム・アーキテクチャおよびソフトウェアの表記法などの IA-64 プログラミング環境の詳細は、今後刊行される『IA-64 Programmer's Reference Manual』で説明する。

1.1 本書の概要

第 1 章「IA-64 アプリケーション・デベロッパーズ・アーキテクチャ・ガイドについて」。このガイドの概要を示す。

第 2 章「IA-64 プロセッサ・アーキテクチャの 概要」。IA-64 アーキテクチャの主な機能の概要を示す。

第 3 章「IA-64 の実行環境」。IA-64 アプリケーション・アーキテクチャの状態 (レジスタ、メモリなど) について説明する。

第 4 章「IA-64 アプリケーション・プログラミング・モデル」。IA-64 のアーキテクチャをアプリケーション・プログラマの観点から説明する。IA-64 命令を関連する機能ごとにまとめ、その動作の概要を示す。

第 5 章「IA-64 浮動小数点プログラミング・モデル」。IA-64 浮動小数点レジスタ、データ型とデータ形式、および浮動小数点命令について説明する。

第 6 章「IA-64 システム環境での IA-32 アプリケーション実行モデル」。IA-64 システム環境での IA-32 アプリケーションの実行について説明する。

第 7 章「IA-64 命令リファレンス」。IA-64 アプリケーション命令とその操作および形式について詳しく説明する。

第 8 章「IA-64 最適化ガイドについて」。「IA-64 最適化ガイド」の概要を示す。

第 9 章「IA-64 プログラミングの概要」。IA-64 アプリケーション・プログラミング環境の概要を示す。

第 10 章「メモリ参照」。コントロールおよびデータのスペキュレーションに関連する機能と最適化について説明する。

第 11 章「プレディケーション、コントロール・フロー、命令ストリーム」。プレディケーション、コントロール・フロー、分岐ヒントに関連する機能および最適化について説明する。

第 12 章「ソフトウェアによるパイプライン化とループのサポート」。ループに対してソフトウェアによるパイプライン処理を使って最適化する方法について詳しく説明する。

第 13 章「浮動小数点アプリケーション」。浮動小数点アプリケーションの現時点でのパフォーマンス上の制約と、これらの制約に関わる IA-64 の機能について説明する。

付録 A「命令シーケンスについて」。IA-64 アーキテクチャでの命令シーケンスについて説明する。

付録 B「IA-64 擬似コード関数」。第 7 章「IA-64 命令リファレンス」で使用している擬似コード関数を示す。

付録 C「IA-64 命令形式」。第 7 章で説明している命令のエンコーディングおよび形式を示す。

1.2 用語の定義

以下の定義は IA-64 アーキテクチャに関連する用語についてであり、本書ではこの定義を用いる。

命令セット・アーキテクチャ (ISA) －アプリケーション－レベルおよびシステム・レベルのリソースを定義する。これらのリソースには、命令とレジスタが含まれる。

IA-64 **アーキテクチャ**－ 64 ビット命令機能、新しいパフォーマンス強化機能、IA-32 命令セットのサポートを備えた新しい ISA。

IA-32 **アーキクチャ**－インテルの 32 ビットおよび 16 ビット・アーキテクチャ。『*インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル*』を参照。

IA-64 **プロセッサ**－ IA-64 および IA-32 の両方の命令セットを実装している Intel の 64 ビット・プロセッサ。

IA-64 **システム環境** － IA-64 および IA-32 の両方のコードを実行できる IA-64 オペレーティング・システム特権環境

IA-32 **システム環境** －『*インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル*』で定義されているオペレーティング・システム特権環境およびリソース。リソースには、仮想ページング、コントロール・レジスタ、デバッグ、パフォーマンス・モニタ、マシン・チェック、および一連の特権命令が含まれる。

1.3 関連文献

- 『***インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル***』－このリファレンス・セットは Intel 32 ビット・アーキテクチャについての詳しい情報を掲載しており、インテル資料センタから入手できる。

第 I 部： IA-64 アプリケーション・アーキテクチャ・ガイド

intel®

IA-64 プロセッサ・アーキテクチャの概要 2

IA-64 アーキテクチャは、従来のアーキテクチャのパフォーマンスの限界を超えて、将来の拡張に備えて最大限の余裕を提供できるように設計されている。そのために IA-64 は、より強力な命令レベルの並列処理が可能な一連の画期的な新機能を備えている。これには、スペキュレーション、プレディケーション、ラージ・レジスタ・ファイル、レジスタ・スタック、先行分岐アーキテクチャなどの多くの機能が含まれる。このアーキテクチャでは、データ・ウェアハウスの利用や電子取引などのハイパフォーマンスのサーバ・アプリケーションにおいて要求されるますます大きなメモリ領域に対応できるよう、64 ビットでのメモリ・アドレス指定が可能になっている。IA-64 アーキテクチャにはまた、デジタル・コンテンツの作成、設計エンジニアリング、科学解析などのワークステーション・アプリケーションによって要求されるハイパフォーマンスをサポートするために、斬新な浮動小数点アーキテクチャおよびその他の機能拡張が組み込まれている。

また、IA-64 アーキテクチャは、IA-32 命令セットとのバイナリでの互換性がある。IA-64 プロセッサは、IA-32 アプリケーションの実行に対応できる IA-64 オペレーティング・システム上で IA-32 アプリケーションを実行できる。また IA-64 プロセッサは、従来の IA-32 オペレーティング・システム上で IA-32 アプリケーション・バイナリを実行できる (システムにプラットフォームおよびファームウェアがサポートされている場合)。IA-64 アーキテクチャは、IA-32 と IA 64 の混合コードの実行もサポートしている。

2.1 IA-64 のオペレーティング環境

IA-64 アーキテクチャは、以下の 2 つのオペレーティング・システム環境をサポートしている。

- IA-32 システム環境。これは IA-32 の 32 ビット・オペレーティング・システムをサポートする。
- IA-64 システム環境。これは IA-64 オペレーティング・システムをサポートする。

このアーキテクチャ・モデルはまた、1 つの IA-64 オペレーティング・システムで IA-32 および IA-64 のアプリケーションの組み合わせをサポートできる。表 2-1 に、IA-64 プロセッサでサポートされる主なオペレーティング環境を示す。

表 2-1. IA-64 プロセッサのオペレーティング環境

システム環境	アプリケーション環境	用途
IA-32	IA-32 命令セット	IA-32 保護モード、リアル・モード、および仮想 8086 モードのアプリケーションおよびオペレーティング・システム環境。IA-32 の Pentium®、Pentium Pro、Pentium II、Pentium III プロセッサとの互換性がある。
IA-64	IA-32 保護モード	IA-64 システム環境での IA-32 保護モード・アプリケーション (OS によってサポートされている場合)。
	IA-32 リアル・モード	IA-64 システム環境での IA-32 リアル・モード・アプリケーション (OS によってサポートされている場合)。
	IA-32 仮想モード	IA-64 システム環境での IA-32 仮想 86 モード・アプリケーション (OS によってサポートされている場合)。
	IA-64 命令セット	IA-64 オペレーティング・システム上の IA-64 アプリケーション。

2.2 命令セット切り替えモデルの概要

IA-64 システム環境 においては、プロセッサは、いつでも IA-32 と IA-64 の命令を実行できる。プロセッサを IA-32 命令セットと IA-64 命令セットとの間で切り替えるために、3 つの特別の命令と割り込みが定義されている。

- `jmpe` (IA-32 命令)。IA-64 のターゲット命令にジャンプし、命令セットを IA-64 に変更する。
- `br.ia` (IA-64 命令)。IA-64 から I A-32 のターゲット命令に分岐し、命令セットを IA-32 に変更する。
- 割り込みは、すべての割り込み条件に対してプロセッサを IA-64 命令セットに切り替える。
- `rfi` (IA-64 命令)。`rfi` (" 割り込みからのリターン ") は、IA-32 命令または IA-64 命令にリターンするように定義される。

`jmpe` 命令と `br.ia` 命令は、小さなオーバーヘッドで命令セット間のコントロールの受け渡しを行う。通常、これらの命令は、動的または静的にリンクされたライブラリをコールするために必要なコール・リンケージおよび呼び出し規則を実現する "thunks" または "stubs" に組み込まれている。詳細は、付録 6、「IA-64 システム環境での IA-32 アプリケーション実行モデル」を参照のこと。

2.3 IA-64 命令セットの特長

IA-64 のアーキテクチャは、持続的にハイパフォーマンスを実現でき、将来のパフォーマンス向上に対する障害を取り除いていくためのアーキテクチャ機能が組み込まれている。IA-64 アーキテクチャは、以下の原則を基礎としている。

- 明示的に並列化を記述した命令
  - — コンパイラとプロセッサの連携のメカニズム
  - — 命令レベルの並列性を活用するための大量のリソース
  - — 128 個の整数レジスタおよび浮動小数点レジスタ、64 個の 1 ビット・プレディケート・レジスタ、8 個の分岐レジスタ
  - — 多数の実行ユニットおよびメモリ・ポートのサポート
- 命令レベルの並列性の強化
  - — スペキュレーション (メモリ・レイテンシの影響を最小限に抑える)
  - — プレディケーション (分岐を排除する)
  - — オーバーヘッドが小さい、ソフトウェアによるループのパイプライン化
  - — 分岐予測による分岐コストの削減
- ソフトウェア・パフォーマンスの改善のための重点的な機能拡張
  - — ソフトウェアのモジュール化のための特別のサポート
  - — ハイパフォーマンスの浮動小数点アーキテクチャ
  - — マルチメディアのための特別の命令

以下の項では、これらの IA-64 の重要な特長について説明する。

2.4 命令レベルの並列性

命令レベルの並列性 (Instruction Level Parallelism、ILP) によって複数の命令を同時に実行できる。IA-64 アーキテクチャでは、個々の命令を並列に実行するために 3 つの命令を 1 つのバンドルにまとめ、1 クロックで複数のバンドルを発行することができる。IA-64 アーキテクチャでは、ラージ・レジスタ・ファイルや複数の実行ユニットなどの膨大な数の並列リソースを利用して、コンパイラが進行中の作業を管理し、同時に発生する種々の計算をスケジュールすることが可能である。

IA-64 アーキテクチャには、ILP を活用するメカニズムが組み込まれている。従来のアーキテクチャのコンパイラでは、スペキュレートされた (見込み) 情報が常に正確であることを保証できなかったために、そのような情報を利用する上で制約があった。IA-64 アーキテクチャでは、コンパイラはスペキュレートされた情報を利用してアプリケーションを正確に実行できる (2.6

節を参照)。さらに従来のアーキテクチャでは、プロシージャ・コールを行うと、レジスタをスピルおよびフィルする必要があるために、パフォーマンスが制約されていた。IA-64 では、プロシージャが使用するレジスタを明示的にプロセッサに知らせることができるので、プロセッサは、低レベルの ILP を使用している場合でも、プロシージャ・レジスタの動作をスケジュールできる。2-7 ページの 2.7 節「レジスタ・スタック」を参照。

2.5 コンパイラからプロセッサへの通信

IA-64 アーキテクチャでは、命令テンプレート、分岐ヒント、キャッシュ・ヒントなどのメカニズムによって、コンパイラがコンパイル時の情報をプロセッサに知らせることが可能である。IA-64 では、コンパイルされたコードによって、ランタイム情報を使ってプロセッサ・ハードウェアを管理できる。この通信メカニズムは、分岐やキャッシュの失敗に伴なうパフォーマンスの低下を最小限にする上で非常に重要である。

IA-64 では、メモリに対するすべてのロード命令およびストア命令には 2 ビットのキャッシュ・ヒント・フィールドがあり、コンパイラはアクセスされるメモリ領域の空間的または時間的な局所性の予測をそこにコード化する。IA-64 プロセッサは、この情報を使ってキャッシュ階層の中のキャッシュ・ラインの位置を判断できる。これによって、階層をより効率的に活用できる。なぜならキャッシュ・ミスの相対的コストは継続的に増加するからである。

2.6 スペキュレーション

スペキュレーションにはコントロール・スペキュレーションとデータ・スペキュレーションの 2 つのタイプがある。どちらのスペキュレーションでも、コンパイラは、早い段階でオペレーションを発行し、クリティカル・パスからこのオペレーションのレイテンシを除去することによって ILP を実現する。コンパイラは、スペキュレーションが有効であると推定される場合に、そのスペキュレーションを使ってオペレーションを発行する。スペキュレーションが有効であるためには 2 つの条件が満たされている必要がある。オペレーションが統計的に見て十分に頻繁に行われていてリカバリを必要とする確率が小さいこと、および、オペレーションを早い段階で発行することが ILP の強化による最適化に役立つことである。スペキュレーションは、コンパイラがオペレーションのレイテンシをオーバーラップさせる (それによってレイテンシを許容する) ことによって統計的 ILP を利用する主要なメカニズムの 1 つである。

2.6.1 コントロール・スペキュレーション

コントロール・スペキュレーションとは、オペレーションの見張り役となる分岐の前にそのオペレーションを実行することを言う。次のコード・シーケンスで検討する。

```
if (a>b) load(ld_addr1,target1)
else load(ld_addr2, target2)
```

(a>b) が決定する前にオペレーション load(ld_addr1,target1) を実行する場合、このオペレーションはコントロール条件 (a>b) に関してコントロール・スペキュレーションとなる。通常の実行では、オペレーション load(ld_addr1,target1) は、実行される場合とされない場合がある。新しいコントロール・スペキュレーションによるロード命令で例外が発生した場合には、(a>b) が真であるときにだけその例外が処理される。コンパイラは、コントロール・スペキュレーションを使用しているときには、元の位置にチェック・オペレーションを残しておく。このチェック命令により、例外が発生したかどうかを確認し、例外が発生した場合はリカバリコードに分岐する。上記のコード・シーケンスは、以下のように解釈される。

```
/* off critical path */
sload(ld_addr1,target1)
sload(ld_addr2,target2)

/* other operations including uses of target1/target2 */
if (a>b) scheck(target1,recovery_addr1)
else scheck(target2, recovery_addr2)
```

2.6.2 データ・スペキュレーション

データ・スペキュレーションとは、ストアに先だってメモリからのロードを実行することである。本来、このストア命令はロード命令の前に行われ、また、その別名となる可能性がある。データ・スペキュレーションによるロードを「アドバンスド・ロード」とも言う。次のコード・シーケンスで検討する。

```
store(st_addr,data)
load(ld_addr,target)
use(target)
```

コンパイル時にメモリ・アドレス間の関係を決定するプロセスを一義化 (Disambiguation) という。上の例では、ld_addr および st_addr を一義化することができず、ストアの前にロードが実行された場合、このロードは、このストアに関してデータ・スペキュレーションをすることになる。実行中にメモリ・アドレスがオーバーラップした場合は、ストア命令の前に発行されたデータ・スペキュレーションによるロード命令の結果が、ストア命令の後に発行される通常のロード命令と異なる値になることがある。そのため、コントロール・スペキュレーションの場合と同様に、コンパイラはデータ・スペキュレーションによってロードするときには、元のロード位置にチェック命令を残しておく。このチェック命令により、オーバーラップが発生したかどうかを確認し、オーバーラップが発生した場合にはリカバリコードに分岐する。上記のコード・シーケンスは以下のように解釈される。

```
/* off critical path */
aload(ld_addr,target)
```

```
/* other operations including uses of target */
store(st_addr,data)
acheck(target,recovery_addr)
use(target)
```

2.6.3 プレディケーション

プレディケーションとは、命令を条件付きで実行することである。従来のアーキテクチャでは、命令の条件付き実行は分岐によって実現されていた。IA-64 では、この機能をプレディケート付きの命令によって実現している。このプレディケーションによって、条件付き実行のために分岐を使用する必要がなくなり、その結果、より大きい基本ブロックが実現でき、これに関する予測ミスによるペナルティを排除できる。

たとえば、プレディケーションを行わない場合の次の命令を想定する。

```
r1 = r2 + r3
```

この命令は、プレディケーションを行った場合には、次の形式になる。

```
if (p5) r1 = r2 + r3
```

この例では、p5 がコントロール・プレディケートとなり、命令を実行して状態を更新するかどうかを決定する。プレディケートの値が真である場合は、命令を実行して状態を更新する。そうでない場合は、通常はこの命令は nop として動作する。プレディケートの値は比較命令によって割り当てられる。

プレディケーションによって分岐を排除することができ、コントロール依存をデータ依存に変換することになり、コンパイラの最適化が簡単になる。たとえば、元のコードとして次のコードを想定する。

```
if (a>b) c = c + 1
else d = d * e + f
```

上記のコードを次のようにプレディケートされたコードに変換すれば、(a>b) の分岐を排除することができる。

```
pT, pF = compare(a>b)
if (pT) c = c + 1
if (pF) d = d * e + f
```

条件が真である場合はプレディケート pT は 1 に設定され、条件が偽である場合はプレディケート pT は 0 に設定される。プレディケート pF は、pT の補数である。分岐時のコントロール依存の命令 c = c ＋ 1 および d = d*e + f は、プレディケート pT および pF によって compare (a>b) におけるデータ依存に変換され、分岐が排除される。これによる追加的な利点として、コンパイラは pT および pF より後の命令を並列で実行するようにスケジュールできる。また、種々の比較命令によって種々の方法でプレディケートを作成できる (例えば、無条件比較、並列比較など)。

2.7 レジスタ・スタック

IA-64 は、コンパイラ制御によるレジスタのリネームによって、プロシージャのコールおよびリターン時のレジスタに対する不必要なスピルおよびフィルを回避している。コール時には、コールされたプロシージャで新しいレジスタ・フレームを使用でき、コールした側 (呼び出し元) やコールされた側 (呼ばれた側) がレジスタをスピルおよびフィルする必要はない。レジスタへのアクセスは、ベース・レジスタによって、命令の中の仮想レジスタ識別子を物理レジスタ識別子にリネームすることによって行われる。呼ばれた側は使用可能なレジスタを自由に使用でき、呼び出し元のレジスタのスピルおよび復元を行う必要はない。呼ばれた側は `alloc` 命令を実行し、使用するレジスタの数を指定して、十分な数のレジスタを確保する。使用できるレジスタの数が足りない場合 (スタック・オーバーフロー) には、`alloc` 命令はプロセッサを停止し、必要な数のレジスタが使用できるようになるまでコール元のレジスタをスピルする。

リターン時には、ベース・レジスタは呼び出し元がコールする前にレジスタにアクセスするために使用していた値に復元される。呼び出し元の一部のレジスタは、ハードウェアによってスピルされ、まだ復元されていないことがある。この場合 (スタック・アンダーフロー)、リターン時にプロセッサは呼び出し元の必要な数のレジスタが復元されるまで停止する。ハードウェアは、明示指定されたレジスタ・スタック・フレーム情報を利用して、最も適切な時点で (呼び出し元のプロシージャおよび呼ばれた側のプロシージャとは無関係に) レジスタ・スタックからメモリへレジスタをスピルおよびフィルすることができる。

2.8 分岐

プレディケーションによって分岐を排除するほかに、分岐予測ミスの確率を減らし、排除されずに残った分岐に対する予測ミスのコストを減らすために、いくつかのメカニズムが導入されている。これらのメカニズムによって、コンパイラは分岐条件に関する情報をプロセッサに知らせることができる。

間接分岐の場合は、分岐レジスタを使って分岐先アドレスを保持する。

カウント指定ループやモジュロ・スケジュール型ループを高速に処理するために特別のループ終了分岐を使用できる。これらの分岐によってループの終了を完全に予測するための情報が得られ、それによって予測ミスのコストを削減し、ループのオーバーヘッドを減らすことができる。

2.9 レジスタ・ローテーション

ループに対するモジュロ・スケジューリングは、機能ユニットに対するハードウェアによるパイプライン処理と似ており、ループにおいて前の反復が完了する前に次の反復が開始する。反復は、実行パイプライン処理のステージと同様のステージに分割される。このモジュロ・スケジューリングによって、

コンパイラはループの反復を順次にではなく並列に処理できる。従来は、複数の反復を同時に実行するには、ループのアンロールとソフトウェアによるレジスタのリネームを必要とした。IA-64 ではレジスタのリネームが可能で、これによって反復ごとに別々のレジスタ・セットを使用することができ、アンロールの必要がなくなった。このようなレジスタのリネームをレジスタ・ローテーションと言う。これによって、ソフトウェアによるパイプライン処理を非常に広範なループ (大小のループ) に適用でき、オーバーヘッドを大幅に減らすことできる。

2.10 浮動小数点アーキテクチャ

IA-64 では、単精度、倍精度 および拡張倍精度 (80 ビット) のデータ型に対応する完全な IEEE サポートを備えた浮動小数点アーキテクチャを定義している。乗算 / 加算の混合演算、最小 / 最大関数、拡張倍精度型メモリ形式より大きい範囲のレジスタ・ファイル形式などの機能が拡張されている。128 個の浮動小数点レジスタが定義されており、これらのうちの 96 個のレジスタはローテート可能なレジスタであり (スタックされない)、これを使ってループを小さいモジュロ・スケジュール型ループにすることができる。スペキュレーションのために複数の浮動小数点ステータス・レジスタがある。

IA-64 には並列操作を行う FP 命令が用意されており、1 つの浮動小数点レジスタの中にある 2 つの 32 ビット単精度数を並列 (独立) に処理する。これらの命令は単精度浮動小数点演算のスループットを大幅に向上させ、大量の 3D 処理を行うアプリケーションやゲームでのパフォーマンスを向上させる。

2.11 マルチメディアのサポート

IA-64 には、汎用レジスタを 8 つの 8 ビット要素、4 つの 16 ビット要素、または 2 つの 32 ビット要素として処理するマルチメディア命令がある。これらの命令は、各要素を並列に (相互に独立に) 操作する。IA-64 マルチメディア命令は、語彙上は Intel の MMX® テクノロジ命令およびストリーミング SIMD 拡張命令テクノロジと互換である。

IA-64 の実行環境 3

アーキテクチャの状態は、レジスタとメモリで構成される。命令を実行した結果は、一連の実行シーケンス規則に従ってアーキテクチャに反映される。本章では、IA-64 アプリケーションのアーキテクチャ状態と実行シーケンス規則について説明する。

3.1 アプリケーション・レジスタの状態

以下に、アプリケーション・プログラムで使用できるレジスタを示す (図 3-1 を参照)。

- **汎用レジスタ** (GR) ー汎用の 64 ビット・レジスタ・ファイル、GR0 ～ GR127。IA-32 命令を実行するときには、IA-32 の整数レジスタおよびセグメント・レジスタが GR8 ～ GR31 を使用する。
- **浮動小数点レジスタ** (FR) ー浮動小数点レジスタ・ファイル、FR0 ～ FR127。IA-32 命令を実行するときには、IA-32 の浮動小数点レジスタおよびマルチメディア・レジスタが FR8 ～ FR31 を使用する。
- **プレディケート・レジスタ** (PR) ー IA-64 のプレディケーションと分岐処理に使用するシングル・ビット・レジスタ、PR0 ～ PR63。
- **分岐レジスタ** (BR) ー IA-64 の分岐処理に使用するレジスタ、BR0 ～ BR7。
- **命令ポインタ** (IP) ー実行中の IA-64 命令のバンドル・アドレスまたは実行中の IA-32 命令のバイト・アドレスを保持するレジスタ 。
- **現在のフレーム・マーカ** (CFM) ー現在の汎用レジスタのスタック・フレーム、および FR/PR のローテーションを示す状態レジスタ。
- **アプリケーション・レジスタ** (AR) ー特定目的の IA-64 および IA-32 のアプリケーション・レジスタ・セット
- **パフォーマンス・モニタ・データ・レジスタ** (PMD) ー パフォーマンス・モニタ・ハードウェア用のデータ・レジスタ
- **ユーザ・マスク** (UM) ーアライメント・トラップ、パフォーマンス・モニタ、および浮動小数点レジスタの使用状況のモニタに使用する一連のシングル・ビット値
- **プロセッサ識別子** (CPUID) ープロセッサのインプリメンテーションによって異なる IA-64 機能を示すレジスタ

IA-32 アプリケーション・レジスタの状態はすべて IA-64 アプリケーション・レジスタ・セットに完全に格納され、IA-64 命令を使ってアクセスできる。IA-32 命令を使って IA-64 固有のレジスタ・セットにアクセスすることはできない。

3.1.1 予約レジスタと無視レジスタ

定義されていないレジスタは、予約レジスタまたは無視レジスタである。**予約レジスタ**にアクセスすると、Illegal Operation (無効操作) フォルトが発生する。**無視レジスタ**を読み取るとゼロが戻される。ソフトウェアによって無視レジスタに任意の値を書きこむことはできるが、ハードウェアは書き込まれた値を無視する。可変サイズのレジスタ・セットでは、プロセッサにインプリメントされていないレジスタも予約レジスタである。これらのレジスタにアクセスすると Reserved Register/Field (予約レジスタ / フィールド) フォルトが発生する。

定義されているレジスタの中の未定義フィールドは、予約フィールドまたは無視フィールドである。**予約フィールド**を読み取ると、ハードウェアは常にゼロを戻す。これらのフィールドには常にソフトウェアによってゼロを書き込まなければならない。予約フィールドに非ゼロの値を書き込もうとすると、Reserved Register/Field (予約レジスタ / フィールド) フォルトが発生する。**予約**フィールドは将来使用される可能性がある。

無視フィールドを読み取ると、別途に記載してある場合を除いて、ハードウェアはゼロを戻す。ソフトウェアによってこれらのフィールドに任意の値を書き込むことはできるが、ハードウェアは書き込まれた値を無視する。別途に記載してある場合を除いて、一部の IA-32 の無視フィールドは将来使用される可能性がある。

表 3-1 に、プロセッサが予約 / 無視レジスタおよび予約 / 無視フィールドを扱う方法をまとめて示す。

表 3-1. 予約 / 無視レジスタおよび予約 / 無視フィールド

種類	読み取り	書き込み
予約レジスタ	Illegal Operation (無効操作) フォルト	Illegal Operation (無効操作) フォルト
無視レジスタ	ゼロ	書き込まれた値は破棄される
予約フィールド	ゼロ	非ゼロ値を書き込むと Reserved Register/Field (予約レジスタ / フィールド) フォルトが発生する
無視フィールド	ゼロ (別途に記載されている場合を除く)	書き込まれた値は破棄される

レジスタ内の定義されているフィールドでは、定義されていない値は予約値である。これらのフィールドには、ソフトウェアによって、必ず、定義された値を書き込まなければならない。**予約値**を書き込もうとすると、Reserved

Register/Field (予約レジスタ / フィールド) フォルトが発生する。一部のレジスタは**読み取り専用レジスタ**である。読み取り専用レジスタに書き込もうとすると、Illegal Operation (無効操作) フォルトが発生する。

フィールドに reserved (予約) というマークが付いている場合には、将来のプロセッサとの互換性を確保するために、ソフトウェア上では、そのフィールドを将来使用する可能性があるがその影響は未知であるものとして扱うべきである。ソフトウェア上で**予約**フィールドを扱う場合は、以下のガイドラインに従う必要がある。

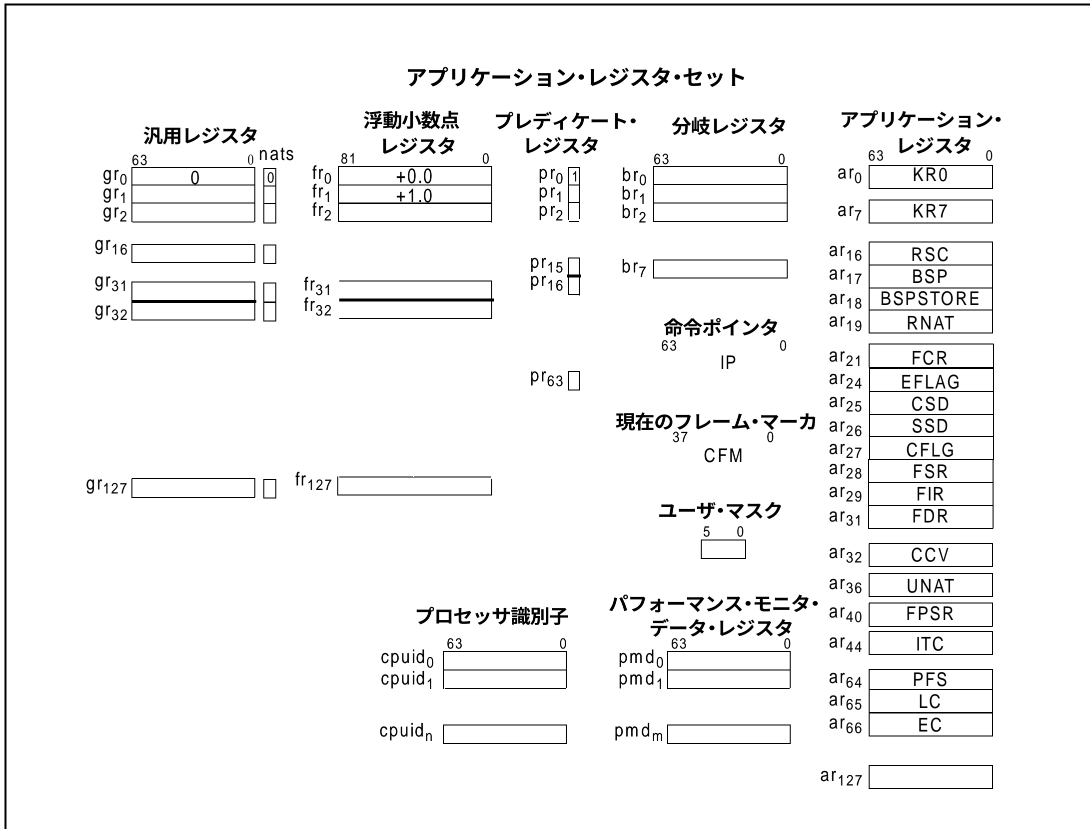
- 予約フィールドの状態に依存しないこと。テストの前にすべての予約フィールドをマスクすること。
- メモリまたはレジスタにストアする場合は、予約フィールドの状態に依存しないこと。
- 予約フィールドまたは無視フィールドに書き込まれた情報が保存できるという前提を用いないこと。
- 可能であれば、予約フィールドまたは無視フィールドに、前に同じレジスタから戻された値を再ロードすること。それができない場合は、ゼロをロードすること。

3.1.2 汎用レジスタ

128 個の 64 ビット**汎用レジスタ**・セットは、すべての整数演算と整数マルチメディア演算の中心的なリソースとなる。これらのレジスタには GR0 ～ GR127 の番号が割り当てられ、特権レベルに関わりなくすべてのプログラムで利用できる。各汎用レジスタには 64 ビットの通常のデータ記憶域のほかに NaT (ノット・ア・シング) ビットがある。NaT ビットはスペキュレーションによって据え置かれた例外の追跡に使用する。

汎用レジスタは 2 つのサブセットに分割される。汎用レジスタの 0 ～ 31 を**スタティック汎用レジスタ**と言う。この中のレジスタ GR0 は特殊であり、ソース・オペランドとして使用されると常にゼロが読み取られ、GR0 に書き込むと Illegal Operation (無効操作) フォルトが発生する。汎用レジスタ 32 ～ 127 を**スタック汎用レジスタと**言う。このスタックされたレジスタは、指定した数のローカル・レジスタや出力レジスタから成るレジスタ・スタック・フレームを割り当てることによってプログラムから使用できる。詳細は、4.1 節を参照のこと。ループを高速に処理するために、スタックされたレジスタの一部をプログラムによってリネームすることができる。4-33 ページの「モジュロ・スケジュール型ループのサポート」を参照のこと。

図 3-1. アプリケーション・レジスタのモデル



IA-32 命令の実行時には、汎用レジスタ 8 ～ 31 には IA-32 の整数レジスタ、セグメント・セレクタ・レジスタ、およびセグメント・ディスクリプタ・レジスタが格納される。

3.1.3 浮動小数点レジスタ

128 個の 82 ビット**浮動小数点レジスタ**・セットは、すべての浮動小数点演算に使用する。これらのレジスタには FR0 〜 FR127 の番号が割り当てられており、特権レベルに関わりなくすべてのプログラムで利用できる。浮動小数点レジスタは 2 つのサブセットに分かれている。浮動小数点レジスタ 0 〜 31 を**スタティック浮動小数点レジスタ**と言う。これらのレジスタの中の FR0 と FR1 は特殊であり、FR0 をソース・オペランドとして使用すると常に +0.0 が読み取られ、FR1 は常に +1.0 が読み取られる。これらのレジスタのどちらかをデスティネーションとして使用すると、フォルトが発生する。スペキュレーションによって据え置かれた例外は、特殊なレジスタ値 NaTVal (**ノット・ア・シング値**) で記録される。

浮動小数点レジスタ 32 〜 127 を**ローテート浮動小数点レジスタ**と言う。ループを高速に処理するために、これらのレジスタをプログラムによってリネームできる。4-33 ページの「モジュロ・スケジュール型ループのサポート」を参照のこと。

IA-32 命令の実行時には、浮動小数点レジスタ 8 〜 31 には IA-32 の浮動小数点レジスタおよびマルチメディア・レジスタが格納される。

3.1.4 プレディケート・レジスタ

64 個の 1 ビット・**プレディケート・レジスタ**・セットは、IA-64 の比較命令の結果を保持するために使用する。これらのレジスタには PR0 〜 PR63 の番号が割り当てられ、特権レベルに関わりなくすべてのプログラムで利用できる。

プレディケート・レジスタは、2 つのサブセットに分かれている。**プレディケート・レジスタ** 0 〜 15 をスタティック・プレディケート・レジスタと言う。この中の PR0 は、ソース・オペランドとして使用されると常に "1" が読み取られ、ディスティネーションとして使用した場合は、その結果が破棄される。スタティック・プレディケート・レジスタは、条件付き分岐でも使用される。4-9 ページの「プレディケーション」を参照のこと。

プレディケート・レジスタ 16 〜 63 を**ローテート・プレディケート・レジスタ**と言う。ループを高速に処理するために、これらのレジスタをプログラムによってリネームすることができる。4-33 ページの「モジュロ・スケジュール型ループのサポート」を参照のこと。

3.1.5 分岐レジスタ

8 つの 64 ビット**分岐レジスタ**・セットは、IA-64 の分岐情報を保持するために使用する。これらのレジスタには BR0 〜 BR7 の番号が割り当てられており、特権レベルに関わりなくすべてのプログラムで利用できる。分岐レジスタは、間接分岐の際の分岐先アドレスを指定するために使用する。詳細は、4-32 ページの「分岐命令」を参照のこと。

3.1.6 命令ポインタ

命令ポインタ (IP) は、現在実行中の IA-64 命令を含むバンドルのアドレスを保持する。IP は、mov ip 命令によって直接に読み取ることができる。IP に直接に書き込むことはできない。IP は命令が実行されるたびにインクリメントされ、分岐命令が実行されると新しい値に設定される。IA-64 の命令バンドルは 16 バイトであり、16 バイト単位でアライメントされるため、IP の下位 4 ビットは常にゼロである。3-18 ページの「命令のエンコーディングの概要」を参照のこと。IA-32 命令セットを実行する場合は、IP は現在実行中の IA-32 命令の 32 ビット仮想リニア・アドレスをゼロ拡張して保持する。IA-32 命令はバイト単位でアライメントされるため、IP の下位 4 ビットは IA-32 命令セットの実行のために保持される。

3.1.7 現在のフレーム・マーカ

汎用レジスタのそれぞれのスタック・フレームには、フレーム・マーカが関連付けられている。フレーム・マーカによって、IA-64 汎用レジスタ・スタックの状態を示す。現在のフレーム・マーカ (CFM) は、現在のスタック・フレームの状態を保持する。CFM は直接に読み取りまたは書き込みを行うことはできない (4-1 ページの「レジスタ・スタック」を参照)。

フレーム・マーカには、スタック・フレームの種々の部分のサイズと、レジスタ・ローテーションで使用する 3 つのレジスタ・リネーム・ベース値が格納される。図 3-2 にフレーム・マーカのレイアウトを示し、表 3-2 にそのフィールドを示す。

コール時に、CFM は AR.pfs (以前のファンクションの状態) レジスタ (3.1.8.10 を参照) の pfm (以前のフレーム・マーカ) フィールドにコピーされる。CFM に新しい値が書き込まれ、新しいスタック・フレームが作成される。このスタック・フレームにはローカル・レジスタやローテートするレジスタは格納されておらず、出力レジスタ・セット (呼び出し元の出力レジスタ) が格納される。さらに、すべての CFM.rrb (レジスタ・リネーム・ベース) レジスタがゼロに設定される。4-33 ページの「モジュロ・スケジュール型ループのサポート」を参照のこと。

図 3-2. フレーム・マーカの形式

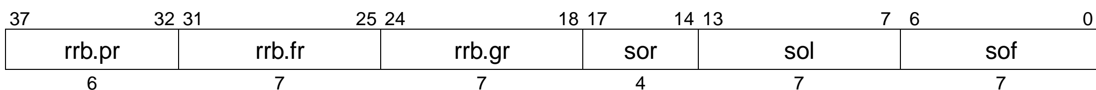


表 3-2. フレーム・マーカのフィールドの説明

フィールド	ビット範囲	説明
sof	6:0	スタック・フレームのサイズ
sol	13:7	スタック・フレームのローカル領域のサイズ
sor	17:14	スタック・フレームのローテート領域のサイズ (ローテートするレジスタの数は、8 * sor)

表 3-2. フレーム・マーカのフィールドの説明 (続き)

フィールド	ビット範囲	説明
rrb.gr	24:18	汎用レジスタのレジスタ・リネーム・ベース
rrb.fr	31:25	浮動小数点レジスタのレジスタ・リネーム・ベース
rrb.pr	37:32	プレディケート・レジスタのレジスタ・リネーム・ベース

3.1.8 アプリケーション・レジスタ

アプリケーション・レジスタ・ファイルには、特定用途のデータ・レジスタと IA-32 および IA-64 の命令セットのためのプロセッサ機能のコントロール・レジスタが含まれ、これらはアプリケーションから読み取り可能である。これらのレジスタは IA-64 アプリケーション・ソフトウェアによってアクセスできる (別途に記載している場合を除く)。表 3-3 に、アプリケーション・レジスタの一覧を示す。

アプリケーション・レジスタは M または I の実行ユニットからだけアクセスできる。これは、表の最後の列に指定されている。無視レジスタは、将来の拡張での下位互換性のために確保されている。

表 3-3. アプリケーション・レジスタ

レジスタ	名前	説明	実行ユニットのタイプ
AR 0-7	KR 0-7[a]	カーネル・レジスタ 0-7	M
AR 8-15		予約	M
AR 16	RSC	レジスタ・スタック設定レジスタ	M
AR 17	BSP	バッキング・ストア・ポインタ (読み取りのみ)	M
AR 18	BSPSTORE	メモリ・ストア用バッキング・ストア・ポインタ	M
AR 19	RNAT	RSE NAT コレクション・レジスタ	M
AR 20		予約	M
AR 21	FCR	浮動小数点コントロール・レジスタ	M
AR 22 – AR 23		予約	M
AR 24	EFLAG[b]	IA-32 EFLAG レジスタ	M
AR 25	CSD	IA-32 コード・セグメント・ディスクリプタ	M
AR 26	SSD	IA-32 スタック・セグメント・ディスクリプタ	M
AR 27	CFLG[a]	IA-32 CR0/CR4 結合レジスタ	M
AR 28	FSR	IA-32 浮動小数点ステータス・レジスタ	M
AR 29	FIR	IA-32 浮動小数点命令レジスタ	M
AR 30	FDR	IA-32 浮動小数点データ・レジスタ	M
AR 31		予約	M
AR 32	CCV	比較交換での比較値レジスタ	M
AR 33 – AR 35		予約	M
AR 36	UNAT	ユーザ NAT コレクション・レジスタ	M
AR 37 – AR 39		予約	M
AR 40	FPSR	浮動小数点ステータス・レジスタ	M
AR 41 – AR 43		予約	M
AR 44	ITC	間隔時間カウンタ	M
AR 45 – AR 47		予約	M
AR 48 – AR 63		無視	M または I
AR 64	PFS	以前のファンクションの状態	I
AR 65	LC	ループ・カウント・レジスタ	I
AR 66	EC	エピローグ・カウント・レジスタ	I
AR 67 – AR 111		予約	I
AR 112 – AR 127		無視	M または I

a. 特権レベルがゼロでないときにこれらのレジスタに書き込むと、Privileged Register (特権レジスタ) フォルトが発生する。読み取りは常に許可される。
b. 特権レベルがゼロでない場合は、一部の IA-32 EFLAG フィールドに書き込みを行っても単に無視される。

3.1.8.1 カーネル・レジスタ (KR07 - AR07)

ユーザが参照可能な 8 つの IA-64 の 64 ビット・データ・カーネル・レジスタは、オペレーティング・システムからの情報をアプリケーションに伝える。これらのレジスタはどの特権レベルでも読み取ることができるが、書き込みには最上位の特権レベルが必要である。KR0 ～ KR2 はまた、IA-32 命令セットの実行時に、IA-32 レジスタの追加的な状態情報を保持する。

3.1.8.2 レジスタ・スタック・コンフィギュレーション・レジスタ (RSC - AR16)

レジスタ・スタック・コンフィギュレーション (RSC) レジスタは、IA-64 レジスタ・スタック・エンジン (RSE) の動作を制御する 64 ビット・レジスタである。RSC の形式を図 3-3 に示し、各フィールドの説明を表 3-4 に示す。RSC を変更する命令を使っても、pl (特権レベル) フィールドを現在実行中のプロセスよりも高いレベルに設定することはできない。

図 3-3. RSC の形式

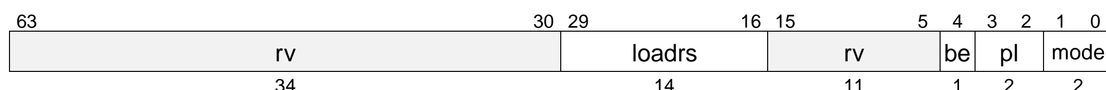


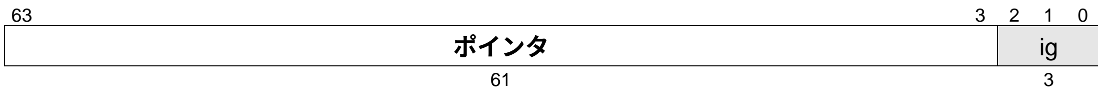
表 3-4. RSC のフィールドの説明

<table>
<tr><th>フィールド</th><th>ビット範囲</th><th colspan="4">説明</th></tr>
<tr><td rowspan="3">mode</td><td rowspan="3">1:0</td><td colspan="4">RSE モード ― RSE がどの程度の頻度でレジスタ・フレームを保存および復元するかを制御する。eager (頻度高) および intensive (頻度中) の設定はヒントであり、lazy (頻度底) としてインプリメントすることもできる。</td></tr>
<tr><td>ビット・パターン</td><td>RSE モード</td><td>ビット 1 :
eager ロード</td><td>ビット 0 :
eager ストア</td></tr>
<tr><td>00
10
01
11</td><td>lazy の強制
intensive ロード
intensive ストア
eager</td><td>使用不能
使用可能
使用不能
使用可能</td><td>使用不能
使用不能
使用可能
使用可能</td></tr>
<tr><td>pl</td><td>3:2</td><td colspan="4">RSE 特権レベル ― RSE によって発行されるロードおよびストアの特権レベル。</td></tr>
<tr><td>be</td><td>4</td><td colspan="4">RSE エンディアン・モード ― RSE によって発行されるロードおよびストアで使用されるバイト順序 (0 : リトル・エンディアン、1: ビッグ・エンディアン)</td></tr>
<tr><td>loadrs</td><td>29:16</td><td colspan="4">RSE ロードのティア・ポイントへの距離 ― RSE をティア・ポイントに同期化するために loadrs 命令で使用する値。</td></tr>
<tr><td>rv</td><td>15:5, 63:30</td><td colspan="4">予約</td></tr>
</table>

3.1.8.3 RSE バッキング・ストア・ポインタ (BSP - AR17)

RSE バッキング・ストア・ポインタ (BSP) は 64 ビットの読取り専用レジスタである (図 3-4)。これは現在のスタック・フレームでの GR 32 の保存場所のメモリ・アドレスを保持する。

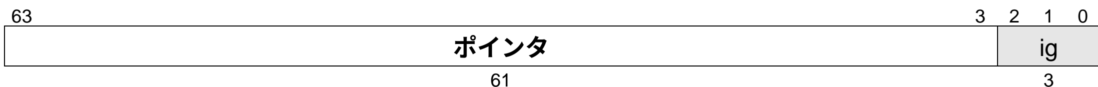
図 3-4. BSP レジスタの形式



3.1.8.4 メモリ・ストア用 RSE バッキング・ストア・ポインタ (BSPSTORE - AR18)

メモリ・ストア用 RSE バッキング・ストア・ポインタ (BSPSTORE) は 64 ビットのレジスタである (図 3-5)。これは RSE が次の値を退避する場所のメモリ・アドレスを保持する。

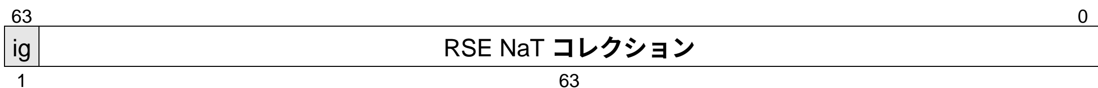
図 3-5. BSPSTORE レジスタの形式



3.1.8.5 RSE NAT コレクション・レジスタ (RNAT - AR19)

RES NaT コレクション (RNAT) レジスタは 64 ビットのレジスタで、RSE が汎用レジスタを退避するときに NaT ビットを一時的に保持するために使用する (図 3-6)。ビット 63 は常にゼロに設定され、書き込まれても無視される。

図 3-6. RNAT レジスタの形式



3.1.8.6 比較交換での比較値レジスタ (CCV - AR32)

比較交換での比較値 (CCV) レジスタは 64 ビットのレジスタで、IA-64 の `cmpxchg` 命令の中の 3 番目のソース・オペランドとして使用する比較値が格納される。

3.1.8.7 ユーザ NAT コレクション・レジスタ (UNAT - AR36)

ユーザ NaT コレクション (UNAT) レジスタは 64 ビットのレジスタで、IA-64 の `ld8.fill` 命令および `st8.spill` 命令によって汎用レジスタを保存および復元するときに NaT ビットを一時的に保持するために使用する。

3.1.8.8 浮動小数点ステータス・レジスタ (FPSR - AR40)

浮動小数点ステータス・レジスタ (FPSR) は、IA-64 浮動小数点命令のトラップ、丸めモード、精度指定、フラグ、および他のコントロール・ビットを制御する。FPSR は、IA-32 浮動小数点命令のステータスについては制御せず、反映しない。FPSR に関する詳細は、5.2 節を参照のこと。

3.1.8.9 間隔時間カウンタ (ITC - AR44)

間隔時間カウンタ (ITC) は、プロセッサのクロック周波数と連動してカウントアップする 64 ビットのレジスタである。アプリケーションでは、時間ベースの計算やパフォーマンス測定の際に ITC を直接サンプリングできる。システム・ソフトウェアを利用して、ITC を権限のない IA-64 アクセスから保護できる。アクセスを保護した場合、最高の特権レベル以外の特権レベルで ITC を読み取ると Privileged Register (特権レジスタ) フォルトが発生する。ITC への書き込みには最高の特権レベルが必要である。IA-32 タイム・スタンプ・カウンタ (TSC) は ITC と同じ働きをする。ITC は、IA-32 の `rdtsc` (read time stamp counter) 命令によって直接読み取ることができる。システム・ソフトウェアを利用して、ITC を権限のない IA-32 アクセスから保護できる。アクセスを保護した場合、IA-32 で最高の特権レベル以外の特権レベルで ITC を読み取ると、IA-32_Exception (GPfault) が発生する。

3.1.8.10 以前のファンクションの状態 (PFS - AR64)

IA-64 の以前のファンクションの状態 (PFS) レジスタには、pfm (以前のフレーム・マーカ)、pec (以前のエピローグ・カウント)、ppl (以前の特権レベル) の各フィールドがある。図 3-7 に PFS の形式を示し、図 3-5 に PFS の各フィールドを示す。プロシージャ・コールを高速に処理するために、これらの値はコール時に CFM レジスタ、EC (エピローグ・カウント) レジスタ、および PSR.cpl (プロセッサ・ステータス・レジスタの現在の特権レベル) から自動的にコピーされる。

IA-64 の `br.call` 命令を実行すると、CFM、EC、および PSR.cpl が PFS にコピーされ、PFS の古いカウントが破棄される。IA-64 の `br.ret` 命令を実行すると、PFS が CFM と EC にコピーされ、この命令によって特権レベルが大きくなる場合を除いて、PFS.ppl が PSR.cpl にコピーされる。

PFS.pfm は CFM (3.1.7 節を参照) と同じレイアウトで、PFS.pec は EC (3.1.8.12 節を参照) と同じレイアウトである。

図 3-7. PFS の形式

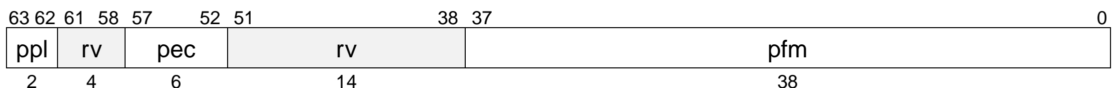


表 3-5. PFS のフィールドの説明

フィールド	ビット範囲	説明
pfm	37:0	以前のフレーム・マーカ
pec	57:52	以前のエピローグ・カウント
ppl	63:62	以前の特権レベル
rv	51:38, 61:58	予約

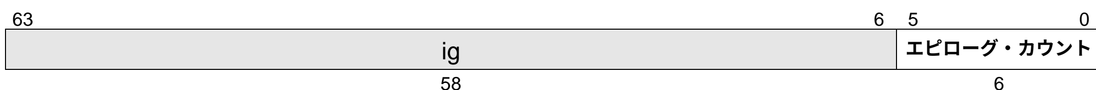
3.1.8.11 ループ・カウント・レジスタ (LC - AR65)

ループ・カウント・レジスタ (LC) は、IA-64 のカウント (反復回数) 指定ループで使用する 64 ビットのレジスタである。LC は、カウント指定ループの分岐によってデクリメントされる。

3.1.8.12 エピローグ・カウント・レジスタ (EC - AR66)

エピローグ・カウント・レジスタ (EC) は、IA-64 のモジュロ・スケジュール型ループの最終 (エピローグ) ステージで使用する 6 ビットのレジスタである。4-33 ページの「モジュロ・スケジュール型ループのサポート」を参照のこと。図 3-8 に EC レジスタを示す。

図 3-8. エピローグ・カウント・レジスタの形式



3.1.9 パフォーマンス・モニタ・データ・レジスタ (PMD)

パフォーマンス・モニタ・レジスタ・セットは、権限をもつソフトウェアによって、すべての特権レベルでアクセスできるように設定できる。パフォーマンス・モニタ・データはアプリケーションの中から直接にサンプリングすることができる。オペレーティング・システムによって、ユーザ設定のパフォーマンス・モニタを保護することができる。

保護されている場合には、現在の特権レベルに関わりなく、パフォーマンス・カウンタを読み取るとゼロが戻される。パフォーマンス・モニタの書き込みには、最高の特権レベルが必要である。パフォーマンス・モニタを使って、IA-32 および IA-64 の命令セットの実行に関するパフォーマンス情報を収集することができる。

3.1.10 ユーザ・マスク (UM)

ユーザ・マスク (UM) は、プロセッサ・ステータス・レジスタのサブセットであり、IA-64 アプリケーション・プログラムからアクセスできる。ユーザ・マスクによって、メモリ・アクセスのアライメント、バイト・オーダー、お

intel®

よびユーザ設定のパフォーマンス・モニタを制御する。また、IA-64 の浮動小数点レジスタの変更状態を記録する。図 3-9 にユーザ・マスクの形式を示し、表 3-6 にユーザ・マスクの各フィールドの説明を示す。

図 3-9. ユーザ・マスクの形式

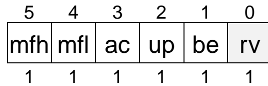


表 3-6. ユーザ・マスクのフィールドの説明

フィールド	ビット範囲	説明
rv	0	予約
be	1	IA-64 でのビッグ・エンディアン形式のメモリ・アクセスの使用 (ロードおよびストアは制御するが RSE メモリ・アクセスは制御しない) 0：アクセスがリトル・エンディアン方式で行われる 1：アクセスがビッグ・エンディアン方式で行われる IA-32 のデータ・メモリ・アクセスではこのビットは無視される。IA-32 でのデータ参照は常にリトル・エンディアン方式で行われる
up	2	IA-32 および IA-64 の命令セットの実行の際のユーザ・パフォーマンス・モニタの使用 0：ユーザ・パフォーマンス・モニタは使用不能 1：ユーザ・パフォーマンス・モニタは使用可能
ac	3	IA-32 および IA-64 のデータ・メモリ参照の際のアライメント・チェック 0：アライメントされていないデータ・メモリを参照すると Unaligned Data Reference (非アライメント・データ参照) フォルトが発生することがある (参照方法による) 1：アライメントされていないデータ・メモリを参照すると必ず Unaligned Data Reference (非アライメント・データ参照) フォルトが発生する
mfl	4	浮動小数点レジスタの下位レジスタ (f2 .. f31) への書き込み－このビットは、レジスタ f2 .. f31 をターゲット・レジスタとして使用する IA 64 の命令が完了したときに 1 にセットされる。このビットは スティッキーであり、ユーザ・マスクに明示的に書き込むことによってのみクリアされる。
mfh	5	浮動小数点レジスタの上位レジスタ (f32 .. f127) への書き込み－このビットは、レジスタ f32..f127 をターゲット・レジスタとして使用する IA 64 の命令が完了したときに 1 にセットされる。このビットは スティッキーであり、ユーザ・マスクに明示的に書き込むことによってのみクリアされる。

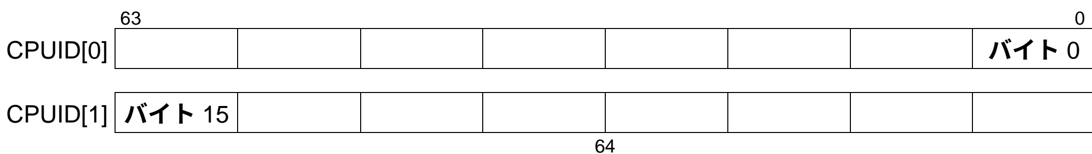
3.1.11 プロセッサ識別レジスタ

アプリケーション・レベルのプロセッサ識別情報は、CPUID という IA-64 レジスタ・ファイルに格納される。このレジスタ・ファイルは、固定領域 (レジスタ 0 ～ 4) と可変領域 (レジスタ 5 以上) に分かれている。CPUID[3].number フィールドは、プロセッサ固有の情報が格納されている 8 バイト・レジスタの最大数を示す。

CPUID レジスタには特権が割り当てられておらず、間接的に mov (from) 命令を使ってアクセスされる。レジスタの CPUID[3].number の個数を超えるすべてのレジスタは予約されており、アクセスすると Reserved Register/Field (予約レジスタ / フィールド) フォルトが発生する。書き込みは許可されておらず、書き込むための命令はない。

CPUID レジスタ 0 と 1 にはベンダ情報が格納され、プロセッサのベンダ名 (ASCII 形式) を示す (図 3-10)。文字列の終りから 16 番目のバイトまでのすべてのバイトはゼロである。入力した ASCII 文字は、若い番号のレジスタの若い番号のバイト位置から順に格納される。

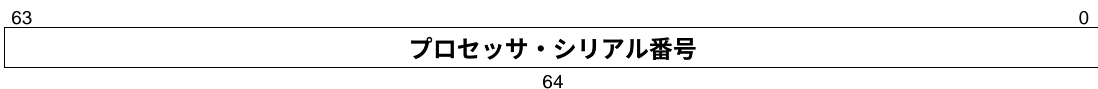
図 3-10. CPUID レジスタ 0 と 1 - ベンダ情報



CPUID レジスタ 2 にはプロセッサのシリアル番号が格納される。プロセッサのシリアル番号がプロセッサ・モデルによってサポートされていて使用不能になっていない場合、このレジスタはプロセッサのシリアル番号に対応する 64 ビットの数値を戻す (図 3-11)。この条件に当てはまらない場合はゼロを戻す。64 ビットのシリアル番号 (CPUID レジスタ 2) と 32 ビットのバージョン情報 (CPUID レジスタ 3 のアーキテクチャ・リビジョン、ファミリ、モデル、およびリビジョン番号) とが組み合わさって 96 ビットのプロセッサ識別子となる。

96 ビットのプロセッサ識別子は一意な識別子で、重複することはない。

図 3-11. CPUID レジスタ 2 - プロセッサ・シリアル番号



CPUID レジスタ 3 には、プロセッサに関連するバージョン情報を示すいくつかのフィールドが含まれる。図 3-12 および 表 3-7 に各フィールドの定義を示す。

図 3-12. CPUID レジスタ 3 - バージョン情報

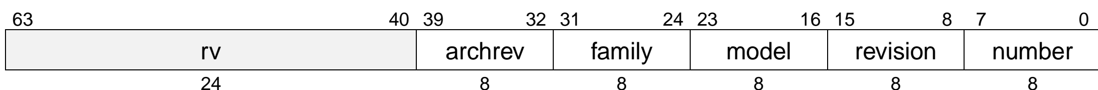


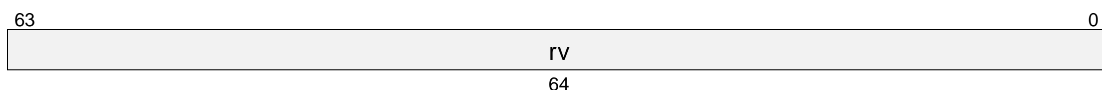
表 3-7. CPUID レジスタ 3 のフィールド

フィールド	ビット	説明
number	7:0	提供されている CPUID レジスタの最大のインデックス (CPUID レジスタの数より 1 小さい数)。この値は 4 以上である。
revision	15:8	プロセッサのリビジョン番号。プロセッサ・モデルにおけるこのプロセッサのリビジョンまたはステッピングを表す 8 ビットの値。
model	23:16	プロセッサ・モデル番号。プロセッサ・ファミリにおけるプロセッサ・モデルを表す一意な 8 ビット値。
family	31:24	プロセッサ・ファミリ番号。プロセッサ・ファミリを表す一意な 8 ビット値。
archrev	39:32	アーキテクチャ・リビジョン。プロセッサがサポートしているアーキテクチャ・リビジョン番号を表す 8 ビットの値。
rv	63:40	予約

CPUID レジスタ 4 には、IA-64 の機能に関する一般的なアプリケーション・レベルの情報が格納されている。図 3-13 に示すように、所定の IA-64 機能がそのプロセッサ・モデルでサポートされているかどうかを示す一連のフラグ・ビットである。ビットが 1 であればその機能はサポートされており、ビットが 0 であればその機能はサポートされていない。このレジスタは IA-32 の命令セットの機能については含んでいない。IA-32 の命令セットの機能は、IA-32 の cpuid 命令によって取得できる。現在のアーキテクチャでは、定義されている機能ビットはない。

将来のプロセッサ・モデルで新しい機能が追加または削除された時点で、新しい機能の有無が新しい機能ビットによって示される。このレジスタの値がゼロであれば、IA-64 アーキテクチャの最初のリビジョンにおいて定義されたすべての機能がサポートされていることを示す。

図 3-13. CPUID レジスタ 4 - 一般的な機能ビット



3.2 メモリ

この節では、IA-64 アプリケーション・プログラムから見たメモリについて説明する。32 ビットおよび 64 ビットのアプリケーションでのメモリ・アクセス方法についても説明する。また、メモリ内のアドレス指定可能なユニットのサイズおよびアライメント、およびバイト・オーダーの操作方法についても説明する。

3.2.1 アプリケーションでのメモリ・アドレス指定モデル

メモリはバイト単位でアドレス指定でき、64 ビット・ポインタによってアクセスされる。ハードウェア・モードがない 32 ビット・ポインタ・モデルは、アーキテクチャによってサポートされる。メモリ上で 32 ビットであるポインタは、64 ビットのレジスタにロードされ、処理される。ソフトウェアでは、使用前に 32 ビット・ポインタを明示的に 64 ビット・ポインタに変換しておかなければならない。

3.2.2 アドレス指定可能なユニットとアライメント

1、2、4 、8、10、および 16 バイト単位でメモリをアドレス指定できる。

すべてのアドレス指定可能なユニットを、本来のアライメントされた境界上にストアすることを推奨する。ハードウェアやオペレーティング・システム・ソフトウェアでは、アライメントされていないアクセスもサポートしているが、ある程度のパフォーマンス低下を伴う。10 バイトの浮動小数点値は、16 バイトでアライメントされた境界上にストアする必要がある。

大きなユニット内のビットには、最下位ビットから順に、ゼロから始まる番号が割り当てられる。メモリから汎用レジスタにロードされる数は、常に、レジスタの最下位部分に置かれる (ロードされた値は、ターゲットの汎用レジスタに右揃えで配置される)。

命令バンドル (1 つのバンドルにつき 3 つの IA-64 命令が入る) は 16 バイト・ユニットで、常に 16 バイト境界上にアライメントされる。

3.2.3 バイト・オーダー

IA-64 でのメモリ参照においては、ユーザ・マスクの UM.be ビットを使って、ロードおよびストア操作でリトル・エンディアン方式とビッグ・エンディアン方式のどちらを使用するかを設定する。UM.be ビットが 0 であれば、1 バイトより多いロードおよびストア操作はリトル・エンディアン方式で行われる (メモリ内の下位アドレスのバイトがレジスタの下位バイトに対応する)。UM.be ビットが 1 であれば、ビッグ・エンディアン方式で行われる (メモリ内の下位アドレスのバイトがレジスタの上位バイトに対応する)。バイト単位のロードおよびストアでは、UM.be ビットの影響を受けない。UM.be ビットは命令のフェッチ、IA-32 でのメモリ参照、および RSE には影響を与えない。IA-64 命令は常に、プロセッサからリトル・エンディアン形式のユニットとしてアクセスされる。命令がビッグ・エンディアン形式のデータとして参照された場合は、その命令は、レジスタでは逆順で表される。

図 3-14 に、リトル・エンディアン形式での種々のロード操作を示す。図 3-15 に、ビッグ・エンディアン形式での種々のロード操作を示す。ストア操作については示していないが、ロード操作と同様である。

図 3-14. リトル・エンディアン形式のロード

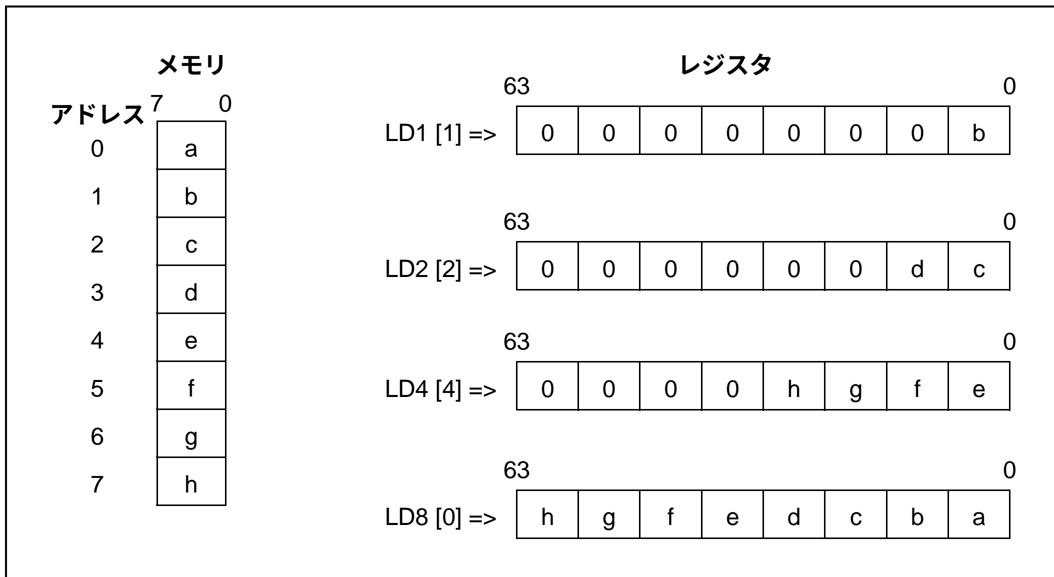
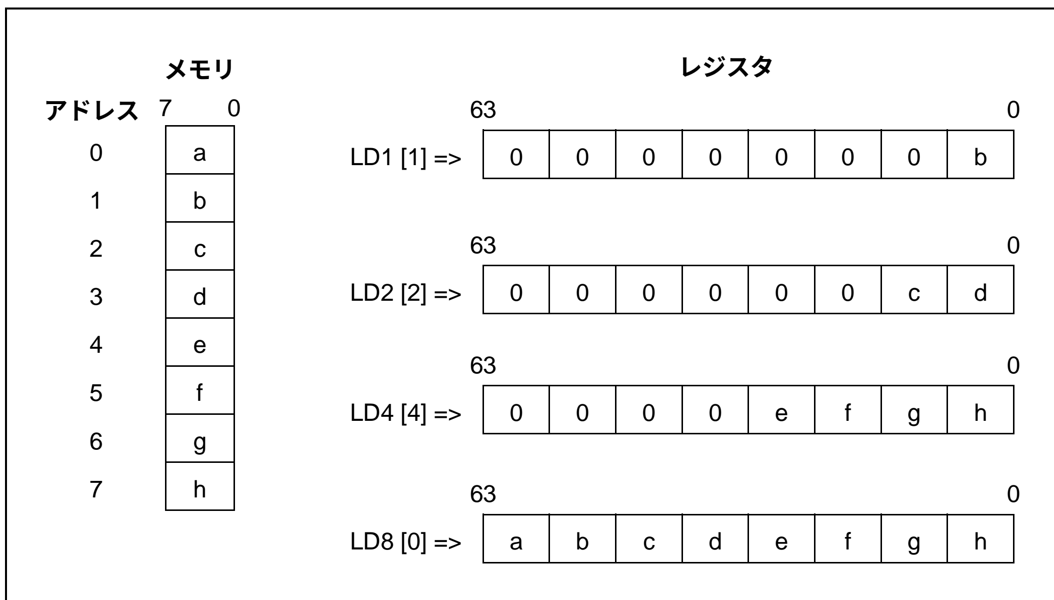


図 3-15. ビッグ・エンディアン形式のロード



3.3 命令のエンコーディングの概要

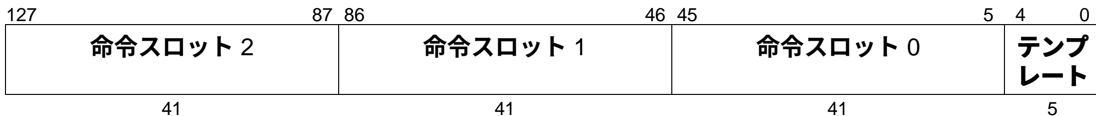
IA-64 命令は、6 つのタイプに分類される。各命令タイプは、1 つ以上のタイプの実行ユニットで実行できる。表 3-8 に、命令タイプとそれが実行される命令ユニットのタイプの一覧を示す。

表 3-8. 命令タイプと実行ユニット・タイプの関係

命令タイプ	説明	実行ユニット・タイプ
A	整数 ALU	I ユニット または M ユニット
I	非 ALU 整数	I ユニット
M	メモリ	M ユニット
F	浮動小数点	F ユニット
B	分岐	B ユニット
L+X	拡張	I ユニット

3 つの命令がグループ化されて、128 ビット・サイズのアラインメントされたコンテナ (**バンドル**) に入れられる。各バンドルには、3 つの 41 ビット**命令スロット**と、5 ビットのテンプレート・フィールドがある。バンドルの形式を図 3-16 に示す。

図 3-16. バンドルの形式



実行中に、プログラム内のアーキテクチャ上の**停止**によって、停止前の 1 つ以上の命令と停止後の 1 つ以上の命令との間に何らかのリソースの依存関係があることをハードウェアに知らせる。停止は、表 3-9 で右側に二重線があるスロットの後に入る。たとえば、テンプレート 00 には停止はない。テンプレート 03 にはスロット 1 の後に停止があり、スロット 2 の後にも停止がある。

テンプレート・フィールドは、停止の位置のほかに、命令スロットの実行ユニット・タイプへのマッピングを指定する。命令とユニットの間の可能なすべてのマッピングが利用できるわけではない。表 3-9 に、定義されている組合せを示す。右側の 3 つの列は、バンドルの中の 3 つの命令スロットに対応している。各列には、その命令スロットによってコントロールされる実行ユニットのタイプを示している。

intel

表 3-9. テンプレート・フィールドのエンコーディングと命令スロットのマッピング[a]

テンプレート	スロット 0	スロット 1	スロット 2
00	M ユニット	I ユニット	I ユニット
01	M ユニット	I ユニット	I ユニット
02	M ユニット	I ユニット	I ユニット
03	M ユニット	I ユニット	I ユニット
04	M ユニット	L ユニット	X ユニット
05	M ユニット	L ユニット	X ユニット
06			
07			
08	M ユニット	M ユニット	I ユニット
09	M ユニット	M ユニット	I ユニット
0A	M ユニット	M ユニット	I ユニット
0B	M ユニット	M ユニット	I ユニット
0C	M ユニット	F ユニット	I ユニット
0D	M ユニット	F ユニット	I ユニット
0E	M ユニット	M ユニット	F ユニット
0F	M ユニット	M ユニット	F ユニット
10	M ユニット	I ユニット	B ユニット
11	M ユニット	I ユニット	B ユニット
12	M ユニット	B ユニット	B ユニット
13	M ユニット	B ユニット	B ユニット
14			
15			
16	B ユニット	B ユニット	B ユニット
17	B ユニット	B ユニット	B ユニット
18	M ユニット	M ユニット	B ユニット
19	M ユニット	M ユニット	B ユニット
1A			
1B			
1C	M ユニット	F ユニット	B ユニット
1D	M ユニット	F ユニット	B ユニット
1E			
1F			

3.4 命令シーケンス

IA-64 プログラムは、バンドルにパックされた一連の命令と停止指示で構成される。命令の実行は以下の順序で行われる。

- バンドルは、最下位のメモリ・アドレスから最上位のメモリ・アドレスの順に実行される。同じバンドルの中の下位のメモリ・アドレスをもつ命令は、上位のメモリ・アドレスをもつバンドルに優先する。メモリ上の各バンドルのバイト順序はリトル・エンディアン方式である (テンプレート・フィールドは、各バンドルのバイト 0 の中に含まれる)。
- 1 つのバンドルの中では、命令は図 3-16 に示すように、命令スロット 0 から命令スロット 2 の順に実行される。

命令シーケンスに関する詳細は、付録 A「命令シーケンスについて」を参照のこと。

intel®

IA-64 アプリケーション・プログラミング・モデル　4

本章では、アプリケーション・プログラマの観点から見た IA-64 のアーキテクチャ上の機能について説明する。IA-64 命令を関連する機能ごとにまとめて、その動作の概要を示している。別途の記述がない限り、すべての即値は、使用前に 64 ビットに符号拡張される。浮動小数点プログラミング・モデルについては、第 5 章「IA-64 浮動小数点プログラミング・モデル」で説明する。

本章で扱う IA-64 プログラミング・モデルの主要な機能は以下の通りである。

- 汎用レジスタ・スタック
- 整数演算命令
- 比較命令およびプレディケーション
- メモリ・アクセス命令およびスペキュレーション
- 分岐命令および分岐予測
- マルチメディア命令
- レジスタ・ファイル転送命令
- 文字列およびポピュレーション・カウント

4.1　レジスタ・スタック

3-3 ページの「汎用レジスタ」で説明しているように、汎用レジスタ・ファイルはスタティックな汎用レジスタとスタックされた汎用レジスタの 2 つのサブセットに分かれている。スタティックなサブセットは、すべてのプロシージャから参照可能な部分であり、GR 0 ～ GR 31 の 32 個のレジスタから成っている。スタックされたサブセットは、それぞれのプロシージャでローカルに使用でき、GR 32 から始まる 0 ～ 96 個の任意の数のレジスタを利用できる。レジスタ・スタックのメカニズムは、プロシージャのコールおよびリターンの際のレジスタ・アドレスのリネームによって実現される。このリネームのメカニズムは、それ以外にはアプリケーション・プログラムからは参照できない部分である。レジスタ・スタックは、IA-32 命令セットの実行時には使用不能になる。

スタティックなサブセットは、ソフトウェアの規則に従ってプロシージャ境界で保存および復元しなければならない。スタックされたサブセットは、ソフトウェアによる明示的な操作を行わずに、レジスタ・スタック・エンジン (RSE) によって自動的に保存および復元される。

これ以外のすべてのレジスタ・ファイルは、すべてのプロシージャから参照可能であり、ソフトウェアの規則に従ってソフトウェアによって保存および復元しなければならない。

4.1.1 レジスタ・スタック・オペレーション

特定のプロシージャから参照可能なスタックされたサブセット内のレジスタをレジスタ・スタック・フレームと言う。このフレームは、ローカル領域と出力領域の 2 つの可変サイズの領域に分割される。コールの直後には新しく起動されたフレームのローカル領域のサイズはゼロであり、出力領域のサイズはコール元の出力領域のサイズと等しく、コール元の出力領域にオーバーレイする。

フレームのローカル領域および出力領域のサイズは `alloc` 命令によって変更できる。この命令では、フレームのサイズ (sof) およびローカル領域のサイズ (sol) を即値で指定する。

注: アセンブリ言語では、`alloc` 命令で、入力領域サイズの即値、ローカル領域サイズの即値、出力領域サイズの即値の 3 つのオペランドを指定する。sol の値は、入力領域サイズの即値とローカル領域サイズの即値との和である。sof の値は 3 つの即値の総和である。

sof の値には、現在のプロシージャから検出可能な、スタックされたサブセットの全体のサイズを指定する。sol の値は、ローカル領域のサイズを指定する。出力領域のサイズは、sof と sol の差である。現在アクティブなプロシージャについて、これらのパラメタの値が CFM (現在のフレーム・マーカ) に保持される。

スタックされたレジスタのうち現在のフレームの範囲外のものを読み込むと、未定義の結果が返される。これらのレジスタに書き込むと Illegal Operation (無効操作) フォルトが発生する。

コール・タイプの分岐が実行された場合には、CFM が PFS (以前のファンクション状態) アプリケーション・レジスタの PFM (以前のフレーム・マーカ) フィールドにコピーされ、以下のように呼ばれた側のフレームが作成される。

- 呼び出し元の出力領域の最初のレジスタが呼ばれた側の GR 32 になるように、スタックされたレジスタがリネームされる。
- ローカル領域のサイズがゼロに設定される。
- 呼ばれた側のフレームのサイズ (sof_{b1}) が呼び出し元の出力領域のサイズ ($sof_a - sol_a$) に設定される。

呼び出し元のレジスタ・スタック・フレームの出力領域の値が呼ばれた側から参照可能になる。このオーバーラップによって、プロシージャ間でのパラメタと戻り値の受け渡しがすべてレジスタ内で実行できる。

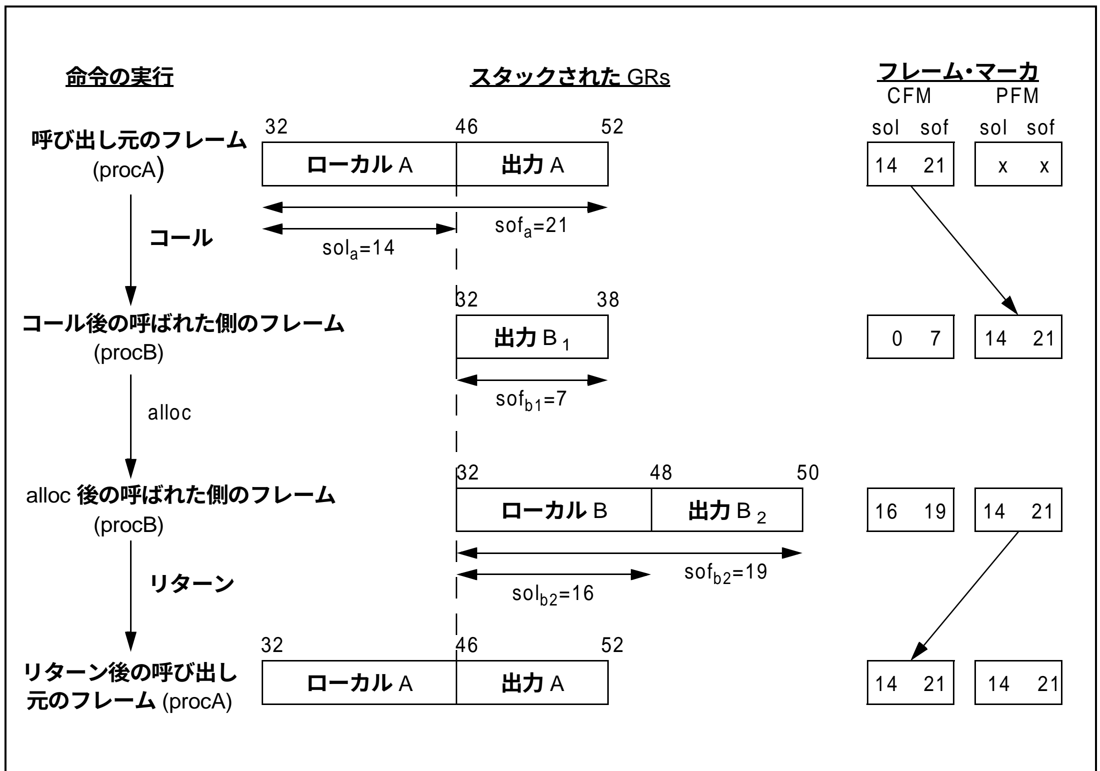
`alloc` 命令を発行することによって、プロシージャ・フレームのサイズを動的に変更できる。`alloc` 命令はレジスタのリネームは行わずに、レジスタ・スタック・フレームのサイズと、ローカル領域と出力領域の分割だけを変更

する。一般的には、プロシージャがコールされると、プロシージャは自分が使用するいくつかのローカル・レジスタ (呼び出し元の出力レジスタで渡されたパラメタを含む) と出力領域 (コールするプロシージャに渡すパラメタのために使用する) を割り当てる。新しく割り当てられたレジスタ (NaT ビットを含む) には未定義の値が含まれる。

リターン・タイプの分岐が実行された場合には、CFM が PFM から復元され、レジスタのリネームが、呼び出し元の設定に合わせて復元される。PFM はプロシージャのローカル状態であり、non-leaf プロシージャによって保存および復元されなければならない。CFM はアプリケーション・プログラムでは直接にはアクセスできず、コール、リターン、`alloc` 命令および `clrrrb` 命令の実行によってのみ更新される。

図 4-1 に、procA (呼び出し元) から procB (呼ばれた側) へのプロシージャ・コールの際のレジスタ・スタックの動作を示す。コール前、コール直後、procB による `alloc` の実行後、および procB から procA へのリターン後の 4 つ時点でのレジスタ・スタックの状態を示している。

図 4-1. プロシージャのコールおよびリターン時のレジスタ・スタックの動作



大部分のアプリケーション・プログラムでは、レジスタ・スタックを効率よく利用するためには、alloc 命令を発行し、PMF を保存および復元するだけでよい。RSE (レジスタ・スタック・エンジン) についての詳細は、ユーザ・レベルのスレッド・パッケージ、デバッガなどのある種の特殊なアプリケーション・ソフトウェアでだけ必要となる。

4.1.2 レジスタ・スタック命令

alloc 命令を使用して、現在のレジスタ・スタック・フレームのサイズを変更できる。alloc 命令は命令グループの最初の命令でなければならない。そうでない場合には、未定義の結果が返される。alloc 命令は、命令グループの中のすべての命令 (alloc 命令自身を含む) から参照可能なレジスタ・スタック・フレームに影響する。alloc 命令はプレディケートを使用できない。alloc 命令は、割り当てられたレジスタの値や NaT ビットには影響しない。レジスタ・スタック・フレームが拡張されると、新しく割り当てられたレジスタの NaT ビットがセットされることがある。

さらに、レジスタ・スタックの状態を明示的に制御するための 3 つの命令がある。これらの命令は、スレッドおよびコンテクストの切り替えに使用される。これには、レジスタ・スタック用のバッキング・ストアの切り替えが対応していなければならない。

flushrs 命令は、以前のすべてのスタック・フレームをバッキング・ストア・メモリへ強制的に出力する。この命令は、物理レジスタ・スタック内の現在のフレームまでのすべてのアクティブ・フレーム (現在のフレームを含まない) が RSE によってバッキング・ストアに退避されるまで命令の実行を停止する。flushrs 命令は命令グループの最初の命令でなければならない。そうでない場合には、未定義の結果が返される。flushrs 命令はプレディケートを使用できない。

表 4-1 に、アーキテクチャ上で参照可能なレジスタ・スタック関連の状態を示す。表 4-2 に、レジスタ・スタック管理命令を示す。スタックに影響するコール・タイプおよびリターン・タイプの分岐については、4-32 ページの「分岐命令」を参照のこと。

表 4-1. アーキテクチャ上で参照可能なレジスタ・スタック関連の状態

レジスタ	説明
AR[PFS].pfm	以前のフレーム・マーカ・フィールド
AR[RSC]	レジスタ・スタック設定アプリケーション・レジスタ
AR[BSP]	バッキング・ストア・ポインタ・アプリケーション・レジスタ
AR[BSPSTORE]	メモリ・ストア用バッキング・ストア・ポインタ・アプリケーション・レジスタ
AR[RNAT]	RSE NaT コレクション・アプリケーション・レジスタ

表 4-2. レジスタ・スタック管理命令

ニーモニック	操作
alloc	レジスタ・スタック・フレームを割り当てる
flushrs	レジスタ・スタックをバッキング・ストアにフラッシュする

4.2 整数演算命令

整数演算ユニットは、一連の算術演算命令、論理命令、シフト命令、および ビット・フィールド操作命令を提供している。また、32 ビットのデータおよびポインタの操作を高速処理するための一連の命令も提供している。

算術演算命令、論理命令、および 32 ビット高速処理命令は、I ユニットでも M ユニットでも実行できる。

4.2.1 算術演算命令

加算および減算 (add、sub) は、通常の 2 入力形式と、特別の 3 入力形式がサポートされている。3 入力の加算形式では、2 つの入力レジスタの和に定数 1 を加算する。3 入力の減算形式では、2 つの入力レジスタの差から定数 1 を減算する。3 入力形式では 2 入力形式と同じニーモニックを使用し、第 3 ソース・オペランドとして "1" を 追加指定する。

加算および減算の即値形式では、レジスタおよび 15 ビット即値を使用する。即値形式では、第 1 オペランドとしてレジスタではなく即値を指定する。また、レジスタと 22 ビット即値との間で加算を実行できる。ただし、ソース・レジスタは GR 0、1、2、または 3 でなければならない。

左シフトおよび加算命令 (shladd) は、1 つのレジスタ・オペランドを 1 ～ 4 ビット左にシフトし、結果をもう 1 つのレジスタ・オペランドに加算する。表 4-3 に、整数算術演算命令を示す。

表 4-3. 整数算術演算命令

ニーモニック	操作
add	加算
add ...,1	3 入力加算
sub	減算
sub ...,1	3 入力減算
shladd	左シフトおよび加算

浮動小数点レジスタを使用する整数乗算命令が定義されている。詳細は 5-20 ページの「整数積和命令」を参照のこと。整数の除算は、ソフトウェアでは浮動小数点除算と同様の方法で実行される。

4.2.2 論理命令

2 つのレジスタの間、またはレジスタと即値との間の論理積 AND (and)、論理和 OR (or)、および排他的論理和 XOR (xor) を計算する命令が定義されている。andcm 命令は、レジスタまたは即値と、別のレジスタの補数との間の論理積 AND をとる。
表 4-4 に整数論理命令を示す。

表 4-4. 整数論理命令

ニーモニック	操作
and	論理積
or	論理和
andcm	補数の論理積
xor	排他的論理和

4.2.3 32 ビットのアドレスおよび整数

IA-64 の 32 ビット・アドレスのサポートは、リージョン・ビット・コピーを実行する加算命令の形で提供されている。これは仮想アドレス変換モデルをサポートするものである。32 ビット・ポインタ加算命令 (addp) は、2 つのレジスタ、またはレジスタと即値とを加算し、結果の上位 32 ビットをゼロにし、第 2 ソース・オペランドの 31:30 ビットを結果の 62:61 ビットにコピーする。shladdp 命令も同様の動作を行うが、加算を実行する前に第 1 ソース・オペランドを 1 〜 4 ビット左にシフトする。shladdp 命令は 2 レジスタ形式だけが提供されている。

また、32 ビット整数のサポートとしては、32 ビットの比較命令や、符号拡張およびゼロ拡張を実行する命令が提供されている。比較命令については 4-8 ページの「比較命令とプレディケーション」で説明している。符号拡張およびゼロ拡張 (sxt、zxt) 命令は、レジスタ内の 8 ビット、16 ビット、または 32 ビットの値を取り出し、正しく拡張された 64 ビットの結果を生成する。

表 4-5 に、32 ビット・ポインタ命令および 32 ビット整数命令を示す。

表 4-5. 32 ビット・ポインタ命令および 32 ビット整数命令

ニーモニック	操作
addp	32 ビット・ポインタの加算
shladdp	32 ビット・ポインタの左シフトおよび加算
sxt	符号拡張
zxt	ゼロ拡張

4.2.4 ビット・フィールド命令およびシフト命令

汎用レジスタ内のビット・フィールドのシフトと操作のために、可変シフト命令、固定シフトおよびマスク命令、128 ビット入力ファネル・シフト命令、および汎用レジスタ内の個々のビットをテストする特別比較演算命令の 4 つのクラスの命令が定義されている。ビット・テスト (tbit) または NaT ビット・テスト (tnat) のための比較命令については、4-8 ページの「比較命令とプレディケーション」で説明している。

可変シフト命令は、汎用レジスタの内容を、もう 1 つの汎用レジスタで指定されるビット数だけシフトする。符号付き右シフト (shr) 命令および符号なし右シフト (shr.u) 命令は、レジスタの内容を右にシフトし、空のビット位置にそれぞれ符号ビットまたはゼロを埋め込む。左シフト (shl) 命令は、レジスタの内容を左にシフトする。

固定シフトおよびマスク命令 (extr,dep) は、固定シフトの一般的な形式である。抽出命令 (extr) は、任意のビット・フィールドを汎用レジスタからターゲット・レジスタの下位ビットにコピーする。ターゲット・レジスタの残りのビットには、ビット・フィールドの符号 (extr の場合)、またはゼロ (extr.u の場合) が書き込まれる。フィールドの長さおよび開始位置は、2 つの即値で指定する。これは基本的には右シフトおよびマスク操作である。シフトするビット数が固定している単純な右シフトは、即値をシフト量とする shr 命令によって指定できる。これは単に抽出命令のアセンブリ擬似オペコードである (抽出されるフィールドがレジスタ・ビットの左端まで拡張される)。

デポジット命令 (dep) は、汎用レジスタの下位ビット、またはオール 0 もしくはオール 1 の即値からビット・フィールドを取り出し、それを任意の位置に挿入し、そのフィールドの左側および右側にはもう 1 つの汎用レジスタのビット (dep の場合) またはゼロ (dep.z の場合) をフィルする。フィールドの長さおよび開始位置は、2 つの即値で指定する。これは基本的には左シフト・マスク・マージ操作である。シフトするビット数が固定している単純な左シフトは、即値をシフト量とする shl 命令によって指定できる。これは単に dep.z 命令のアセンブリ擬似オペコードである (デポジットされるフィールドがレジスタ・ビットの左端まで拡張される)。

ペアの右シフト (shrp) 命令は、128 ビット入力ファネル・シフトを実行する。この命令は、2 つのソース汎用レジスタを連結して生成される 128 ビット・フィールドから任意の 64 ビット・フィールドを抽出する。開始位置は、即値で指定する。これによって、アライメントされていないデータの調整が迅速になる。shrp 命令を使って、両方のオペランドに同じレジスタを指定することによってビットのローテート操作を実行できる。

表 4-6 に、ビット・フィールド命令およびシフト命令を示す。

表 4-6. ビット・フィールド命令およびシフト命令

ニーモニック	操作
shr	符号付き右シフト
shr.u	符号なし右シフト
shl	左シフト
extr	符号付き抽出 (右シフトおよびマスク)
extr.u	符号なし抽出 (右シフトおよびマスク)
dep	デポジット (左シフト、マスク、およびマージ)
dep.z	ゼロのデポジット (左シフトおよびマスク)
shrp	ペアの右シフト

4.2.5 ラージ定数

ラージ定数を生成するために特別の命令が定義されている (表 4-7 参照)。サイズが 22 ビットまでの定数については、add 命令または mov 擬似オペコード (GRO での add の擬似オペコード、GR0 は常にゼロ) が使用できる。23 ビット以上の定数については、64 ビット即値を汎用レジスタに書き込むためのロング型即値移動命令 (movl) が定義されている。この命令は同じバンドル内の 2 つの命令スロットを使用するが、そのような動作をする唯一の命令である。

表 4-7. ラージ定数生成命令

ニーモニック	操作
mov	22 ビット即値の移動
movl	64 ビット即値の移動

4.3 比較命令とプレディケーション

一連の比較命令を使って種々の条件をテストし、命令の動的な実行に影響を与えることができる。比較命令は、指定された条件をテストして、結果のブール値を生成する。これらの結果は、プレディケート・レジスタに書き込まれる。このプレディケート・レジスタは条件付き分岐の条件として、またはプレディケーションの修飾プレディケートとして、動的な実行に影響を及ぼす。

4.3.1 プレディケーション

プレディケーションは、命令の条件付き実行である。ほとんどの IA-64 命令の実行は、修飾プレディケートによって判断される。プレディケートが真の場合、命令は通常通りに実行される。プレディケートが偽の場合、命令はアーキテクチャ上の状態を変更しない (無条件タイプの比較命令、浮動小数点近似命令、および while ループ分岐を除く)。プレディケートは 1 ビット値であり、プレディケート・レジスタ・ファイルにストアされる。プレディケートの値 0 は偽として解釈され、1 は真として解釈される (プレディケート・レジスタ PR0 は 1 にハード配線されている)。

一部の IA-64 命令、たとえば、スタック・フレームの割り当て (`alloc`)、rrb のクリア (`clrrrb`)、レジスタ・スタックのフラッシュ (`flushrs`)、カウント指定の分岐 (`cloop`、`ctop`、`cexit`) などの命令ではプレディケーションを使用できない。

4.3.2 比較命令

プレディケート・レジスタは、汎用レジスタ比較 (`cmp`、`cmp4`)、浮動小数点レジスタ比較 (`fcmp`)、ビット・テストおよび NaT テスト (`tbit`、`tnat`)、浮動小数点分類 (`fclass`)、浮動小数点の逆数近似および平方根逆数近似 (`frcpa`、`frsqrta`) の命令で書き込まれる。`frcpa` と `frsqrta` を除き、これらの比較命令では、比較の結果に基づいて 2 つのプレディケート・レジスタをセットする。2 つのターゲット・レジスタの設定については、4-10 ページの「比較タイプ」で説明している。表 4-8 に比較命令を示す。

表 4-8. 比較命令

ニーモニック	操作
`cmp`, `cmp4`	GR 比較
`tbit`	GR のビット・テスト
`tnat`	GR の NaT ビット・テスト
`fcmp`	FR 比較
`fclass`	FR 分類
`frcpa`, `fprcpa`	浮動小数点逆数近似
`frsqrta`, `fprsqrta`	浮動小数点平方根逆数近似

64 ビット (`cmp`) および 32 ビット (`cmp4`) 比較命令は、10 種類の比較関係式 (>、<= など) のいずれかについて、2 つのレジスタ、またはレジスタと即値を比較する。比較命令は、結果に従って 2 つのプレディケート・ターゲットを設定する。`cmp4` 命令は、両方のソース・オペランドの下位 32 ビットを比較する (上位 32 ビットは無視される)。

ビット・テスト (tbit) 命令は、汎用レジスタの単一ビットの状態に従って 2 つのプレディケート・レジスタを設定する (どの位置のビットを使うかは即値によって指定される)。NaT テスト (tnat) 命令は、汎用レジスタに対応する NaT ビットの状態に従って 2 つのプレディケート・レジスタを設定する。

fcmp 命令は、8 種類の比較関係式のいずれかに従って 2 つの浮動小数点レジスタを比較し、2 つのプレディケート・ターゲットを設定する。fclass 命令は、浮動小数点レジスタのソース・オペランドに含まれる数値の分類に従って 2 つのプレディケート・ターゲットを設定する。

frcpa 命令および frsqrta 命令は、浮動小数点レジスタ・ソース・オペランドが有効な近似値を作成できる場合には単一のプレディケート・ターゲットをセットし、そうでない場合にはプレディケート・ターゲットをクリアする。

4.3.3 比較タイプ

比較命令には通常、無条件、AND、OR、および DeMorgan (ド・モーガン) の 5 つの比較タイプがある。これらの比較タイプによって、比較命令が比較の結果や修飾プレディケートに基づいてターゲットのプレディケート・レジスタに書き込む方法を定義する。これらのタイプを表 4-9 に示す。この表では、"qp" は比較の修飾プレディケートの値を示し、"result" は比較関係式の結果を示す (関係式が真ならば 1、関係式が偽ならば 0)。

表 4-9. 比較タイプの機能

比較タイプ	コンプリータ	操作	
		第 1 プレディケート・ターゲット	第 2 プレディケート・ターゲット
通常	*none*	if (qp) {target = result}	if (qp) {target = !result}
無条件	unc	if (qp) {target = result} else {target = 0}	if (qp) {target = !result} else {target = 0}
AND	and	if (qp && !result) {target = 0}	if (qp && !result) {target = 0}
	andcm	if (qp && result) {target = 0}	if (qp && result) {target = 0}
OR	or	if (qp && result) {target = 1}	if (qp && result) {target = 1}
	orcm	if (qp && !result) {target = 1}	if (qp && !result) {target = 1}
DeMorgan	or.andcm	if (qp && result) {target = 1}	if (qp && result) {target = 0}
	and.orcm	if (qp && !result) {target = 0}	if (qp && !result) {target = 1}

通常タイプでは、比較結果を第 1 プレディケート・ターゲットに、結果の補数を第 2 プレディケート・ターゲットに書き込む。

無条件タイプは、通常タイプと同様であるが、修飾プレディケートが 0 の場合に両方のプレディケート・ターゲットに 0 が書き込まれる。これは、プレディケート・ターゲットの初期設定と通常比較を合わせた操作である。無条件タイプの比較命令では、命令の修飾プレディケートが偽のときにアーキテクチャ上の状態を変更する。

AND、OR、および DeMorgan タイプを " 並列 " 比較タイプと言う。なぜなら、これらのタイプでは 1 つのプレディケート・レジスタに対して複数の (同じタイプの) 比較を同時に行うことができるからである。これによって単一のサイクルで `p5 = (r4 == 0) || (r5 == r6)` のような論理式を計算できる (前のサイクルで p5 が 0 に初期設定されていると仮定する)。DeMorgan タイプは、一方のプレディケート・ターゲットに対する OR タイプの比較と他方のプレディケート・ターゲットに対する AND タイプの比較を組み合わせた比較タイプである。複数の OR タイプの比較 (DeMorgan タイプの OR 部分を含む) で同じ命令グループの同じプレディケート・ターゲットを指定することができる。また、複数の AND タイプ比較 (DeMorgan タイプの AND 部分を含む) で同じ命令グループの同じプレディケート・ターゲットを指定することもできる。

`tnat` 命令および `fclass` 命令を除くすべての比較命令では、一方または両方のソース・レジスタに据え置き例外トークン (NaT または NaTVal ― 4-16 ページの「コントロール・スペキュレーション」を参照) が含まれる場合は、比較結果は異なる。両方のプレディケート・ターゲットは同様に扱われ、0 が書き込まれるか、そのままにされる。これとスペキュレーションを組み合わせることによって、据え置き例外が検出された場合にはプレディケートされたコードをオフにすることができる (`fclass` 命令においても、NaTVal がテスト対象のクラスでない場合には同様の処理が行われる)。表 4-10 に、この動作の説明を示す。

表 4-10.　ソース入力に NaT が含まれる場合の比較結果

比較タイプ	操作
通常	if (qp) {target = 0}
無条件	target = 0
AND	if (qp) {target = 0}
OR	(not written)
DeMorgan	(not written)

一部の比較命令については、比較タイプのサブセットだけを示している。表 4-11 に、それぞれの命令で利用できる比較タイプを示す。

表 4-11.　命令および利用できる比較タイプ

命令	関係式	利用できる比較タイプ
`cmp`, `cmp4`	a == b, a != b, a > 0, a >= 0, a < 0, a <= 0, 0 > a, 0 >= a, 0 < a, 0 <= a	通常、無条件、AND、OR、DeMorgan
	上記以外のすべての関係式	通常、無条件

表 4-11. 命令および利用できる比較タイプ (続き)

命令	関係式	利用できる比較タイプ
tbit, tnat	すべて	通常、無条件、AND、OR、DeMorgan
fcmp, fclass	すべて	通常、無条件
frcpa, frsqrta, fprcpa, fprsqrta	適用外	無条件

4.3.4 プレディケート・レジスタの転送

プレディケート・レジスタ・ファイルと汎用レジスタとの間の転送のための命令が用意されている。これらの命令は " ブロードサイド " 方式で動作する。つまり、複数のプレディケート・レジスタが並列に転送され、プレディケート・レジスタ N が汎用レジスタのビット N に (またはビット N から) 転送される。

プレディケートへの移動命令 (mov pr=) は、即値によって指定されるマスクに従って、汎用レジスタから複数のプレディケート・レジスタをロードする。マスクは、PR1 〜 PR15 のそれぞれに対する各 1 ビット (PR0 は 1 にハード配線されている) と、PR16 〜 PR63 の全体についての 1 ビット (ローテートするプレディケート) から成る。プレディケート・レジスタは、対応するマスク・ビットが 1 であれば汎用レジスタの対応するビットから書き込まれ、対応するマスク・ビットが 0 であれば変更されない。

ローテートするレジスタへの移動命令 (mov pr.rot=) は、即値で与えられる 48 ビットを 48 個のローテートするレジスタ (PR16 〜 PR63 まで) にコピーする。即値は、28 個のビットを符号拡張したものである。したがって PR16 〜 PR42 は個々に新しい値に設定でき、PR43 〜 PR63 はオール 0 または 1 にセットされる。

プレディケートからの移動命令 (mov =pr) は、プレディケート・レジスタ・ファイルの全体を汎用レジスタ・ターゲットに転送する。

これらのすべてのプレディケート・レジスタ転送においては、プレディケート・レジスタは、あたかもレジスタ・リネーム・ベース (CFM.rrb.pr) がゼロであるかのようにアクセスされる。したがって、通常はローテートするプレディケートを初期化する前に、ソフトウェアによって CFM.rrb.pr をクリアしなければならない。

4.4 メモリ・アクセス命令

メモリは、単純なロード、ストア、およびセマフォ命令でアクセスされる。これらの命令は、汎用レジスタまたは浮動小数点レジスタとの間でデータを転送する。メモリ・アドレスは汎用レジスタの内容によって指定される。

また、ほとんどのロード命令およびストア命令は、ベース・アドレス・レジスタの更新を指定できる。ベースの更新は、即値または汎用レジスタの内容をアドレス・レジスタに追加し、その結果をアドレス・レジスタに戻す。更新は、ロード操作やストア操作の後に実行される。すなわちアドレスのポスト・インクリメントとして実行される。

最高のパフォーマンスを実現するには、データを自然境界にアライメントしなければばならない。UM.ac (ユーザ・マスク・レジスタのアライメント・チェック・ビット) が 1 ならば、4K バイト境界の中で、自然境界にアライメントされていないデータ・アクセスは失敗する。UM.ac が 0 ならば、アライメントされていないデータ・アクセスは、プロセッサによってサポートされている場合は成功する。サポートされていない場合には、Unaligned Data Reference (非アライメント・データ参照) フォルトが発生する。

4K バイト境界を超えるすべてのメモリ・アクセスは、UM.ac に関わりなく Unaligned Data Reference (非アライメント・データ参照) フォルトになる。また、すべてのセマフォ命令においては、アクセスが自然境界にアライメントされていない場合は、UM.ac に関わりなく Unaligned Data Reference (非アライメント・データ参照) フォルトになる。

1 バイトより大きいメモリ幅でのアクセスは、ビッグ・エンディアン方式またはリトル・エンディアン方式で実行できる。すべてのメモリ・アクセスのバイト順序は、IA-64 メモリ参照の場合はユーザ・マスク・レジスタの UM.be フィールドによって決定される。IA-32 メモリ参照はすべてリトル・エンディアン方式で実行される。

表 4-12 に、ロード命令、ストア命令、およびセマフォ命令を示す。

表 4-12. メモリ・アクセス命令

ニーモニック			操作
汎用	浮動小数点		
	通常	ロード・ペア	
ld	ldf	ldfp	ロード
ld.s	ldf.s	ldfp.s	スペキュレーティブ・ロード
ld.a	ldf.a	ldfp.a	アドバンスド・ロード
ld.sa	ldf.sa	ldfp.sa	スペキュレーティブ・アドバンスド・ロード
ld.c.nc, ld.c.clr	ldf.c.nc, ldf.c.clr	ldfp.c.nc, ldfp.c.clr	チェック・ロード
ld.c.clr.acq			順序付けされたチェック・ロード
ld.acq			順序付けされたロード
ld.bias			バイアスされたロード
ld8.fill	ldf.fill		フィル

表 4-12. メモリ・アクセス命令 (続き)

ニーモニック			操作
汎用	浮動小数点		
	通常	ロード・ペア	
st	stf		ストア
st.rel			順序付けされたストア
st.spill	stf.spill		退避
cmpxchg			比較および交換
xchg			メモリと GR の交換
fetchadd			フェッチおよび加算

4.4.1 ロード命令

ロード命令は、データをメモリから汎用レジスタ、浮動小数点レジスタ、または浮動小数点レジスタ・ペアに転送する。

汎用レジスタへのロードでは、アクセス・サイズとして 1、2、4、および 8 バイトが定義されている。8 バイト未満のサイズでは、ロードされた値はゼロ拡張されて 64 ビットになる。

浮動小数点ロードでは、単精度 (4 バイト)、倍精度 (8 バイト)、拡張倍精度 (10 バイト)、単精度ペア (8 バイト)、および倍精度ペア (16 バイト) の 5 つのアクセス・サイズが定義されている。メモリからロードされた値は浮動小数点レジスタ形式に変換される (詳細は 5-10 ページの「メモリ・アクセス命令」を参照)。浮動小数点のペア・ロード命令では、隣接する 2 つの単精度または倍精度の数値が 2 つの独立した浮動小数点レジスタにロードされる (ターゲット・レジスタ指定子に関する制限については、ldfp[s/d] 命令の説明を参照)。浮動小数点のペア・ロード命令では、ベース・レジスタの更新は指定できない。

コンパイラによって指示されたコントロールおよびデータのスペキュレーションをサポートするために、汎用レジスタおよび浮動小数点レジスタのロード命令の応用形式が定義されている。これらの応用形式は、汎用レジスタの NaT ビットおよび ALAT を使用する。4-16 ページの「コントロール・スペキュレーション」および 4-20 ページの「データ・スペキュレーション」を参照のこと。

また、メモリ / キャッシュ・サブシステムを制御するための応用形式も定義されている。順序付けされたロード命令を使って、メモリ・アクセスの順序を強制することができる。4-30 ページの「メモリ・アクセスの順序」を参照のこと。バイアスされたロード命令は、アクセスしたラインの排他的所有権を取得するためのヒントを提供する。4-20 ページの「メモリ階層の制御と整合性」を参照のこと。

メモリに退避されたレジスタ値を復元するための専用ロード命令が定義されている。ld8.fill 命令は、汎用レジスタおよび対応する NaT ビット (8 バイト・アクセスのみに定義されている) をロードする。ldf.fill 命令は、メモリの値を浮動小数点レジスタ形式で変換なしにロードする (16 バイト・アクセスのみに定義されている)。4-18 ページの「レジスタのスピルおよびフィル」を参照のこと。

4.4.2 ストア命令

ストア命令は、汎用レジスタまたは浮動小数点レジスタからメモリにデータを転送する。ストア命令ではスペキュレーションは行われない。ストア命令は、ベース・アドレス・レジスタの更新を指定できるが、それには即値だけを使用できる。メモリ / キャッシュ・サブシステムの制御のための応用形式も定義されている。順序付けされたストア命令を使って、メモリ・アクセスの順序を強制することができる。

汎用レジスタと浮動小数点レジスタのストアは、それらのレジスタのロードと同じアクセス・サイズで定義される。唯一の例外として、浮動小数点のペア・ストア命令はない。

レジスタ値をメモリに退避するための専用ストア命令が定義されている。st8.spill 命令は、汎用レジスタおよび対応する NaT ビット (8 バイト・アクセスのみに定義されている) をストアする。それによってスペキュレーティブ計算の結果をメモリに退避し、復元することができる。stf.spill 命令は、浮動小数点レジスタを浮動小数点レジスタ形式で変換なしにメモリにストアする。それによってレジスタの退避および復元用コードが、将来の浮動小数点レジスタ形式の拡張に適合できる。stf.spill 命令はまた、レジスタに NaTVal がある場合でもフォルトにならない (これは 16 バイト・アクセスのみに定義されている)。4-15 ページの「レジスタのスピルおよびフィル」を参照のこと。

4.4.3 セマフォ命令

セマフォ命令は自動的にメモリから汎用レジスタをロードし、操作を実行し、結果を同じメモリ位置にストアする。セマフォ命令ではスペキュレーションは行われない。また、ベース・レジスタは更新されない。

アトミックなセマフォ操作として、交換 (xchg)、比較交換 (cmpxchg)、およびフェッチおよび加算 (fetchadd) の 3 つのタイプの操作が定義されている。

xchg 命令では、第 1 ソース・オペランドによって指定されるメモリ位置の内容がゼロ拡張されてターゲットにロードされ、次に第 2 ソース・オペランドが同じメモリ位置にストアされる。

cmpxchg 命令では、第 1 ソース・オペランドによって指定されるメモリ位置の内容がゼロ拡張されてターゲットにロードされ、ゼロ拡張された値が CCV (比較交換での比較値) アプリケーション・レジスタの内容と等しい場合に、第 2 ソース・オペランドが同じメモリ位置にストアされる。

fetchadd 命令では、1 つの汎用レジスタ・ソース・オペランド、1 つの汎用レジスタ・ターゲット、および即値を指定する。この命令では、ソース・オペランドによって指定されたメモリ位置の内容がゼロ拡張されてターゲットにロードされ、ロードされた値に即値が加算され、その結果が同じメモリ位置にストアされる。

表 4-13. メモリ・アクセス関連の状態

レジスタ	機能
UM.be	ユーザ・マスクのバイト順序
UM.ac	ユーザ・マスクの非アライメント・データの参照フォルトのイネーブル
UNAT	GR NaT コレクション
CCV	比較交換での比較値アプリケーション・レジスタ

4.4.4 コントロール・スペキュレーション

コンパイラによって指示されたスペキュレーションを実行するための特別のメカニズムが提供されている。このスペキュレーションには、コントロール・スペキュレーションとデータ・スペキュレーションの 2 通りの方法があり、それぞれ異なるメカニズムを使用する。4-20 ページの「データ・スペキュレーション」を参照のこと。

4.4.4.1 コントロール・スペキュレーションの概念

コントロール・スペキュレーションはコンパイラの最適化の 1 つの方法である。コントロール・スペキュレーションでは、命令または一連の命令は、プログラムの動的コントロール・フローが実際にプログラム中のその命令シーケンスを必要とする箇所に到達するかどうかに関わりなく実行される。これは、実行レイテンシが長い命令シーケンスで行われる。早い段階で実行を開始することによって、コンパイラはその実行を他の処理とオーバーラップでき、より効率的な並行処理が行われ、全体の実行時間が短縮される。コンパイラは、プログラムの動的コントロール・フローで最終的にはこの計算を必要とすると判定した時点で、この最適化を実行する。コントロール・フローでこの計算が必要でないと判定した場合には、その結果は廃棄される (プロセッサ・レジスタにロードされた結果は使用されない)。

スペキュレーティブな命令シーケンスがプログラム上で必要でないことがあるので、プログラムから検出可能な例外が発生した場合でも、プログラムのコントロール・フローが実際にこの命令シーケンスを必要としていることがわかるまでは、例外を知らせる信号は生成されない。そのため、後で例外を知らせる信号が必要になった時点でそれを生成できるように、例外の発生を記録するメカニズムが用意されている。このような例外を、据え置き例外と言う。命令によって例外が据え置かれた場合には、ターゲット・レジスタに、プログラム内に据え置き例外があることを知らせる特別なトークンが書き込まれる。

据え置き例外トークンは、汎用レジスタ・ファイルと浮動小数点レジスタ・ファイルとで異なる。汎用レジスタでは、各レジスタに NaT (ノット・ア・シング) という追加ビットが定義されている。したがって、汎用レジスタのビット長は 65 になる。NaT ビットが 1 のとき、レジスタに据え置き例外トークンがあり、64 ビットのデータ部分にはソフトウェアでは依存できない固有の値が含まれている。浮動小数点レジスタでは、据え置き例外は、NaTVal という特別の擬似ゼロ・コードによって示される (詳細については 5-3 ページの「浮動小数点レジスタ値の表現」を参照)。

4.4.4.2 コントロール・スペキュレーションと命令

命令にはスペキュレーティブ命令 (スペキュレーションを行うことができる) と非スペキュレーティブ命令 (スペキュレーションを行うことができない) がある。非スペキュレーティブ命令では、例外が発生した時点で例外が生成されるので、これらの命令が実行されるとわかる前にそれをスケジュールするのは危険である。スペキュレーティブ命令では、例外は据え置かれる (生成されない) ので、命令が実行されることがわかる前にそれをスケジュールしても安全である。

汎用レジスタおよび浮動小数点レジスタへのロードには、非スペキュレーティブ命令 (`ld`、`ldf`、`ldfp`) とスペキュレーティブ命令 (`ld.s`、`ldf.s`、`ldfp.s`) を使用できる。結果を汎用レジスタまたは浮動小数点レジスタに書き込む計算命令は、一般にはスペキュレーティブ命令である。汎用レジスタおよび浮動小数点レジスタ以外の状態を変更する命令は非スペキュレーティブ命令である。なぜなら、据え置き例外を表現する方法がないからである (いくつかの例外がある)。

据え置き例外トークンは、プログラム内ではデータ・フローと同様の方法で伝播される。スペキュレーティブ命令によって据え置き例外トークンが入っているレジスタを読み込むと、据え置き例外トークンをそのターゲットに伝播する。このようにスペキュレーションを使って一連の命令を実行でき、例外が発生したかどうかは、結果レジスタをチェックして据え置き例外トークンの有無を調べるだけでわかる。

プログラムの中で、スペキュレーティブ計算の結果が必要であるとわかった時点で、スペキュレーション・チェック (`chk.s`) 命令が使用される。この命令は、据え置き例外トークンの有無を調べる。据え置き例外トークンがなかった場合は、スペキュレーティブ計算は正常に完了しており、通常通りにプログラムの実行が続行される。据え置き例外トークンが見つかった場合は、スペキュレーティブ計算が失敗しており、再実行しなければならない。この場合、`chk.s` 命令は新しいアドレス (`chk.s` 命令の即値オフセットによって指定される) に分岐する。ソフトウェア上でこのメカニズムを利用することにより、スペキュレーティブ計算のコピーを含むコードを (非スペキュレーティブ・ロードで) 呼び出すことができる。すでにこの計算が必要なことがわかっているので、このときに発生する例外によって信号を生成し、通常の方法で処理を行う。

一般には計算命令で例外が発生することはないので、据え置き例外トークンを生成する命令はスペキュレーティブ・ロードだけである (IEEE 浮動小数点例外は、一連の代替状態フィールドを使って特別に処理される。5-6 ページの「浮動小数点ステータス・レジスタ」を参照)。これ以外のスペキュレーティブ命令は、据え置き例外トークンを伝播するが、それを生成することはない。

4.4.4.3 コントロール・スペキュレーションと比較

すでに説明したように、汎用レジスタと浮動小数点レジスタ以外のレジスタ・ファイルに書き込む命令の大部分は、非スペキュレーティブ命令である。比較 (`cmp`、`cmp4`、`fcmp`)、ビット・テスト (`tbit`)、浮動小数点分類 (`fclass`)、および浮動小数点近似 (`frcpa`、`frsqrta`) 命令は特別のケースである。これらの命令は汎用レジスタまたは浮動小数点レジスタを読み込み、1 つまたは 2 つのプレディケート・レジスタに書き込む。

これらの命令では、いずれかのソース・オペランドに据え置き例外トークンがある場合、すべてのプレディケート・ターゲットがクリアされるかそのままにされる (どちらになるかは比較のタイプによって決まる。4-11 ページの表 を参照)。ソフトウェアでは、この動作を利用して、依存関係のある条件付き分岐を行わずに、依存関係のあるプレディケートされた命令を無効にするように設定できる。4-9 ページの「プレディケーション」を参照のこと。

据え置き例外トークンは、いくつかの比較命令によってもテストできる。NaT テスト (`tnat`) 命令は、指定した汎用レジスタに対応する NaT ビットをテストし、2 つのプレディケート結果を書き込む。浮動小数点分類 (`fclass`) 命令は、浮動小数点レジスタの NaTVal をテストし、結果を 2 つのプレディケート・レジスタに書き込む (`fclass` 命令では NaTVal がテスト対象のクラスの 1 つである場合には、NaTVal 入力があると両方のプレディケート・ターゲットをクリアしない)。

4.4.4.4 リカバリなしのコントロール・スペキュレーション

非スペキュレーティブ命令で据え置き例外トークンが入っているレジスタを読み込むと、Register NaT Consumption (レジスタ Nat 参照) フォルトが生成される。このような命令は、リカバリ不可能なスペキュレーション・チェック操作と同じような働きをする。コンパイル環境によっては、据え置かれる例外がすべて致命的エラーである場合もある。そのようなプログラムでは、スペキュレーティブ計算の結果がチェックされ、据え置き例外トークンが見つかった場合に、プログラムの実行が中止される。このようなプログラムでは、スペキュレーティブ計算の結果は、非スペキュレーティブ命令を使用するだけでチェックできる。

4.4.4.5 レジスタのスピルおよびフィル

レジスタをメモリにスピル (退避) し、据え置き例外トークンを保持し、スピル (退避) されたレジスタを復元するための、特別のストアおよびロード命令が用意されている。

汎用レジスタのスピル (退避) およびフィル (復元) 命令 (st8.spill、ld8.fill) は、汎用レジスタと、対応する NaT ビットを保存および復元する。

st8.spill 命令は、汎用レジスタの NaT ビットを UNAT (ユーザ NaT コレクション) アプリケーション・レジスタに書き込み、NaT ビットが 0 の場合にはレジスタの 64 ビットのデータ部分をメモリに書き込む。レジスタの NaT ビットが 1 の場合には、UNAT は更新されるが、メモリの更新はプロセッサのバージョンに固有であり、下記の 3 通りのスピルの動作のいずれかになる。

- st8.spill は、レジスタの 64 ビットのデータ部分をメモリを書き込めない。
- st8.spill は、指定したメモリ位置にゼロを書き込むことができる。
- st8.spill は、プロセッサが NaT テストされたすべてのスペキュレーティブ・ロードのターゲット・レジスタにゼロを戻し、すべての NaT 伝播命令が命令ページで指定されているすべての計算を実行することが保証されている場合にだけ、64 ビットのデータ部分をメモリに書き込むことができる。

メモリ・アドレスのビット 8:3 によって、UNAT レジスタのどのビットに書き込まれるかが決まる。

ld8.fill 命令は、メモリから汎用レジスタをロードし、メモリ・アドレスのビット 8:3 によって指定される UNAT レジスタのビットから、対応する NaT ビットを取り出す。UNAT レジスタの保存および復元は、ソフトウェアによって行わなければならない。ソフトウェア上では、st8.spill 命令および st8.fill 命令の実行中に UNAT レジスタの内容が常に正しいことを保証しなければならない。

浮動小数点レジスタのスピルおよびフィル命令 (stf.spill、ldf.fill) は、浮動小数点レジスタ (16 バイトとして保存されている) を保存または復元し、FR に NaTVal が含まれている場合にも、例外を生成しない (これらの命令は UNAT レジスタには影響を及ぼさない)。

汎用および浮動小数点のスピル / フィル命令では、スペキュレーティブ命令のターゲットであるレジスタのスピルおよびフィルが可能であり、したがって、据え置き例外トークンが含まれることがある。また、汎用レジスタ・ファイルと浮動小数点レジスタ・ファイルの間の転送によって、2 つの据え置き例外トークンの形式の変換が行われる。

表 4-14 に、コントロール・スペキュレーション関連の状態を示す。表 4-15 に、コントロール・スペキュレーションに関連する命令を示す。

表 4-14. コントロール・スペキュレーション関連の状態

レジスタ	説明
NaT bits	各 GR に関連付けられている 65 番目のビット、据え置き例外を示す
NaTVal	FR の擬似ゼロ・エンコーディング、据え置き例外を示す
UNAT	ユーザ NaT コレクション・アプリケーション・レジスタ

表 4-15. コントロール・スペキュレーションに関連する命令

ニーモニック	操作
ld.s, ldf.s, ldfp.s	GR および FR のスペキュレーティブ・ロード
ld8.fill, ldf.fill	GR のフィルと Nat の収集、FR のフィル
st8.spill, stf.spill	GR のスピルと Nat の収集、FR のスピル
chk.s	GR または FR の据え置き例外トークンのテスト
tnat	GR Nat ビットのテストおよびプレディケートの設定

4.4.5 データ・スペキュレーション

コントロール・スペキュレーションによるロードおよびチェックによってコンパイラが複数のコントロール依存を越えて命令をスケジュールできるのと同様に、データ・スペキュレーションによるロードおよびチェックによってコンパイラはいくつかのタイプの曖昧なデータ依存を越えて命令をスケジュールできる。この項では、使用モデル、データ・スペキュレーションおよび関連する命令の語彙を詳しく説明する。

4.4.5.1 データ・スペキュレーションの概念

ロードとストア (またはメモリ状態を更新する何らかの操作) がアクセスするメモリの領域がオーバラップするかどうかをスタティックに判断できないとき、このストアとロードの間には曖昧なメモリ依存関係があると言う。便宜的に、特定のロードに対してスタティックに明確にできないストアを、そのロードに対して曖昧であると言う。このような場合には、コンパイラは、プログラム内での元のロード命令とストア命令の順序を変更することはできない。このスケジューリング上の制約を取り除くために、アドバンスド・ロードという特別のロード命令を、そのロードに対して曖昧である 1 つ以上のストア命令よりも前に実行することができる。

コントロール・スペキュレーションの場合と同様に、コンパイラは、アドバンスド・ロードに依存する操作を予測し、後でその予測が成功したかどうかを調べるためのチェック命令を挿入することができる。データ・スペキュレーションでは、このチェックは、本来のデータ・スペキュレーテションによらないロードがスケジュールされる任意の箇所に挿入できる。

このように、データ・スペキュレーションによる命令シーケンスは、アドバンスド・ロード、そのロードの値に依存する 0 個以上の命令、およびチェック命令によって構成される。つまり、一連のストアの後にロードが続くシーケンスは、アドバンスド・ロードの後に一連のストアとチェックが続くシーケンスに変換できる。このような変換を実行するかどうかは、データ・スペキュレーションが失敗したときのリカバリの可能性とコストによって決まる。

4.4.5.2 データ・スペキュレーションと命令

アドバンスド・ロードには、整数 (ld.a)、浮動小数点 (ldf.a)、および浮動小数点ペア (ldfp.a) の形式がある。アドバンスド・ロードを実行すると、ALAT (アドバンスド・ロード・アドレス・テーブル) という構造にエントリが割り当てられる。後に、対応するチェックを実行したときにエントリが存在していれば、データ・スペキュレーションは成功している。エントリが存在しない場合は、スペキュレーションは失敗し、下記の 2 通りのコンパイラ生成のリカバリ処理のどちらかが実行される。

1. アドバンスド・ロードに対して 曖昧であるストアの前にスケジュールされている唯一の命令がそのアドバンスド・ロードである場合は、チェック・ロード命令 (ld.c、ldf.c または ldpf.c) を使ってリカバリを行う。このチェック・ロード命令では、ALAT を使って対応するエントリを検索する。エントリが見つかった場合は、スペキュレーションは成功していたことになる。エントリが見つからない場合、スペキュレーションは失敗し、このチェック・ロード命令によってメモリから正しい値を再ロードする。図 4-2 にこの変換を示す。

図 4-2. ld.c によるデータ・スペキュレーションのリカバリ

データ・スペキュレーションの前	データ・スペキュレーションの後

// other instructions
st8 [r4] = r12
ld8 r6 = [r8];;
add r5 = r6, r7;;
st8 [r18] = r5
	ld8.a r6 = [r8];; // advanced load
	// other instructions
	st8 [r4] = r12
	ld8.c.clr r6 = [r8] // check load
	add r5 = r6, r7;;
	st8 [r18] = r5

2. アドバンスド・ロードに対して曖昧であるストアの前に、ロードされている値に依存する 1 つのアドバンスド・ロード命令といくつかの命令がスケジュールされている場合は、アドバンスド・ロード・チェック命令 (chk.a) を使用する。アドバンスド・ロード・チェック命令では、スペキュレーション・チェック命令 (chk.s) と同様に、スペキュレーションが成功した場合はインラインで実行が継続され、リカバリは不要である。スペキュレーションが失敗した場合、chk.a 命令はコンパイラ生成のリカバリコードへ分岐する。リカバリコードには、失敗したデータ・スペキュレーティブ・ロードに依存していたすべての作業 (チェック命令の時点まで) を再実行する命令が含まれる。チェック・ロードの場合と同様に、アドバンスド・ロード・チェックを使用してのデータ・スペキュレーションの成否は、ALAT を使って対応するエントリを検索することによって確認される。図 4-3 にこの変換を示す。

図 4-3. chk.a によるデータ・スペキュレーションのリカバリ

<table>
<tr><th>データ・スペキュレーションの前</th><th>データ・スペキュレーションの後</th></tr>
<tr><td><pre>
    // other instructions
    st8   [r4] = r12
    ld8   r6 = [r8];;
    add   r5 = r6, r7;;
    st8   [r18] = r5
</pre></td><td><pre>
    ld8.a r6 = [r8];;
    // other instructions
    add   r5 = r6, r7;;
    // other instructions
    st8   [r4] = r12
    chk.a.clr r6, recover
back:
    st8   [r18] = r5

    // somewhere else in program
recover:
    ld8   r6 = [r8];;
    add   r5 = r6, r7
    br   back
</pre></td></tr>
</table>

リカバリコードでは、通常のロードまたはアドバンスド・ロードを使用して、失敗したアドバンスド・ロードの正しい値を取得することができる。アドバンスド・ロードを使用するのは、スペキュレーションが失敗した後で ALAT エントリを再割り当てすることに利点がある場合だけである。リカバリコードの最後の命令は、`chk.a` 命令の次の命令に分岐しなければならない。

4.4.5.3 ALAT および関連する命令の機能の詳細

ALAT は、アドバンスド・ロードおよびそのチェックが正しく機能するのに必要な状態を保持するための構造である。ALAT の検索には、物理アドレスまたは ALAT レジスタ・タグを使用する。ALAT レジスタ・タグは、物理ターゲット・レジスタの番号とタイプ、および他のプロセッサ固有の状態から生成される一意な番号である。プロセッサ固有の状態には、レジスタ・スタックのラップ・アラウンド情報が含まれており、RSE によってスピルされた可能性がある物理レジスタのインスタンスと、そのレジスタのカレント・インスタンスとは区別されるので、すべてのレジスタ・スタックのラップ・アラウンドにおいて ALAT をパージする必要はない。

IA-32 命令セットを実行したときは、ALAT の内容は未定義のままになる。ソフトウェア上では、ALAT 値が命令セットの移行の前後で保持されていることを前提としてはならない。IA-32 命令セットに移行すると、ALAT の既存のエントリは無視される。

4.4.5.3.1 ALAT エントリの割り当ておよびチェック

アドバンスド・ロードでは、以下のアクションを実行する。

1. アドバンスド・ロードの ALAT レジスタ・タグを計算する (`ldfp.a` 命令では、タグは最初のターゲット・レジスタについてだけ計算される)。
2. 同じ ALAT レジスタ・タグを持つエントリがある場合には、それを削除する。

3. ALAT に新しいエントリを割り当てる。これには、新しい ALAT レジスタ・タグ、ロード・アクセス・サイズ、および物理メモリ・アドレスから導出されたタグが含まれる。
4. アドバンスド・ロードで指定されているアドレスの値をターゲット・レジスタにロードする。(指定されている場合には) ベース・レジスタを更新し、暗黙のプリフェッチを実行する。

チェックの成否は、ALAT の中で一致するレジスタ・タグが見つかるかどうかで決まるので、`chk.a` 命令と `ld.c` 命令のターゲット・レジスタの両方に、対応するアドバンスド・ロードと同じレジスタを指定しなければならない。さらに、チェック・ロードでは、対応するアドバンスド・ロードと同じアドレスおよびオペランド・サイズを使用しなければならない。そうでない場合、チェック・ロードによってターゲット・レジスタに書き込まれる値は定義されない。

アドバンスド・ロード・チェックでは、以下のアクションを実行する。

1. 一致する ALAT エントリを検索し、それが見つかれば次の命令に進む。
2. 一致するエントリが見つからない場合、`chk.a` 命令は指定されたアドレスに分岐する。

失敗したアドバンスド・ロード・チェックを、直接に分岐として実行するか、あるいは、フォルトとして解釈してフォルト・ハンドラが分岐をエミュレートするようにするかはコードによって選択できる。予期される動作モードは、コード上でチェック中に ALAT 内に一致するエントリが見つかることであるが、場合によっては一致する ALAT レジスタ・タグを持つエントリがあってもチェック命令に失敗することがある。これは稀にしか起こらないが、ソフトウェア上では、ALAT にそのようなエントリがないと想定してはならない。

チェック・ロードでは、ALAT 内で一致するエントリをチェックする。一致するエントリが見つからない場合、メモリから値を再ロードし、メモリ参照中に発生したフォルトが生成される。一致するエントリが見つかった場合、ターゲット・レジスタはそのままの状態に置かれる。

チェック・ロードが順序付けされたチェック・ロード (`ld.c.clr.acq`) である場合には、順序付けされたロード命令 (`ld.acq`) の語彙を使って実行される。アドバンスド・ロード・チェックおよびそのチェック・ロードによる ALAT レジスタ・タグの検索は、4-30 ページの「メモリ・アクセスの順序」で説明しているメモリ順序の制約条件に従う。

上記のような柔軟性のほかに、ALAT のサイズ、編成、照合アルゴリズム、および置換アルゴリズムも手法によって異なる。したがって、プログラム内での個々のアドバンスド・ロードおよびそのチェックの成否は、プログラムが実行されるプロセッサによっても異なるし、同じプロセッサでも 1 つのプログラムの実行の中で、あるいは、プログラムを実行するたびに異なることがある。

4.4.5.3.2 ALAT エントリの無効化

アドバンスド・ロードによるエントリの削除のほかに、ALAT エントリはメモリ状態を変更するイベントによって暗黙的に、または `ld.c.clr` 命令、`ld.c.clr.acq` 命令、`chk.a.clr` 命令、または `invala.e` 命令によって明示的に無効に設定されることがある。暗黙的に ALAT エントリを無効にするイベントには、下記のような、メモリ状態またはメモリ移行状態を変更するイベントが含まれる。

1. コヒーレンス・ドメイン内の他のプロセッサ上でのストアまたはセマフォの実行
2. ローカル・プロセッサ上で発行されたストア命令またはセマフォ命令の実行

これらのイベントのいずれかが発生すると、ハードウェアによって ALAT のエントリが表す各メモリ領域をチェックして、それがエントリを無効にするイベントの影響を受ける位置とオーバーラップするかどうかを調べる。メモリ領域が無効化イベントの位置とオーバーラップする ALAT エントリは削除される。

4.4.5.4 コントロール・スペキュレーションとデータ・スペキュレーションの組み合わせ

コントロール・スペキュレーションとデータ・スペキュレーションは相互に排他的ではない。あるロードが、コントロール・スペキュレーティブ・ロードであり、しかもデータ・スペキュレティブ・ロードである場合もある。汎用レジスタおよび浮動小数点レジスタについて、コントロール・スペキュレーティブなアドバンスド・ロード (`ld.sa`、`ldf.sa`、`ldfp.sa`) と非コントロール・スペキュレーティブなアドバンスド・ロード (`ld.a`、`ldf.a`、`ldfp.a`) が定義されている。スペキュレーティブ・アドバンスド・ロードで据え置き例外トークンが生成された場合は、以下のようになる。

1. 同じ ALAT レジスタ・タグがある既存の ALAT はすべて無効になる。
2. 新しい ALAT エントリは割り当てられない。
3. ロードのターゲットが汎用レジスタであれば、その NaT ビットがセットされる。
4. ロードのターゲットが浮動小数点レジスタであれば、ターゲット・レジスタに NaTVal が書き込まれる。

スペキュレーティブ・アドバンスド・ロードが据え置き例外を生成しない場合は、その動作は、対応する非コントロール・スペキュレーティブ・アドバンスド・ロードと同じようになる。

据え置きフォルトの後には ALAT に一致するエントリはないので、1 つのアドバンスド・ロード・チェックまたはチェック・ロードだけで、データ・スペキュレーション障害のチェックと据え置き例外の検出を行うことができる。

4.4.5.5 ALAT 管理のための命令コンプリータ

コンパイラによる ALAT エントリの割り当てと割り当て解除の管理を支援するために、それぞれのアドバンスド・ロード・チェックおよびチェック・ロードに 2 つのコンプリータが提供されている。クリアを行う命令形式 (`chk.a.clr`、`ld.c.clr`、`ld.c.clr.acq`,`ldf.c.clr`、`ldfp.c.clr`) と、クリアを行わない命令形式 (`chk.a.nc`、`ld.c.nc`、`ldf.c.nc`、`ldfp.c.nc`) である。

クリア形式は、ALAT エントリが再使用されないことがわかっていて、エントリを明示的に削除したい場合に使用する。これによってソフトウェア上では、エントリが不必要になったことを知らせることができ、すべてのエントリが割り当てられているために有用なエントリを強制的に削除してしまう可能性が少なくなる。

チェック・ロードのクリア形式では、同じ ALAT レジスタ・タグがある ALAT エントリは、チェック・ロードおよび対応するアドバンスド・ロードのアドレスおよびサイズ・フィールドが一致するかどうかとは無関係に無効にされる。`chk.a.clr` 命令では、命令がフォール・スルーになった（リカバリコードは実行されない）ときにだけエントリが無効になる。したがって、`chk.a.clr` 命令が失敗したときには、一致する ALAT エントリがクリアされる場合とクリアされない場合がある。このようなケースでは、プログラムが正しく実行されるためには `chk.a.clr` 命令が失敗した後にエントリが存在してはならない場合は、リカバリコードによって問題のエントリを明示的に無効にしなければならない。

両方のデータ・スペキュレーション・チェックの非クリア形式では、既存のエントリを ALAT の中に保持しなければならないこと、あるいは一致する ALAT エントリが存在しないときに新しいエントリを割り当てなければならないことをプロセッサに知らせる。このような命令形式をループの中で使用することにより、ループ不変と見なされてコンパイラによってループから除外されたアドバンスド・ロードをチェックすることができる。この動作によって、チェック・ロードが 1 回めの反復で失敗しても、それ以降のすべての反復において必ずしも失敗しないようにすることができる。新しいエントリが ALAT に挿入されたとき、またはエントリの内容が更新されたときに、ALAT に書き込まれた情報のうちチェック・ロードからの情報だけを使用し、前のエントリからの情報は使用しない。`chk.a` 命令の非クリア形式である `chk.a.nc` 命令では、エントリを割り当てない。'nc' コンプリータを指定することにより、エントリをクリアしなければならないことをプロセッサに知らせる。

表 4-16. データ・スペキュレーション関連の状態

構造	機能
ALAT	アドバンスド・ロード・アドレス・テーブル

表 4-17. データ・スペキュレーションに関連する命令

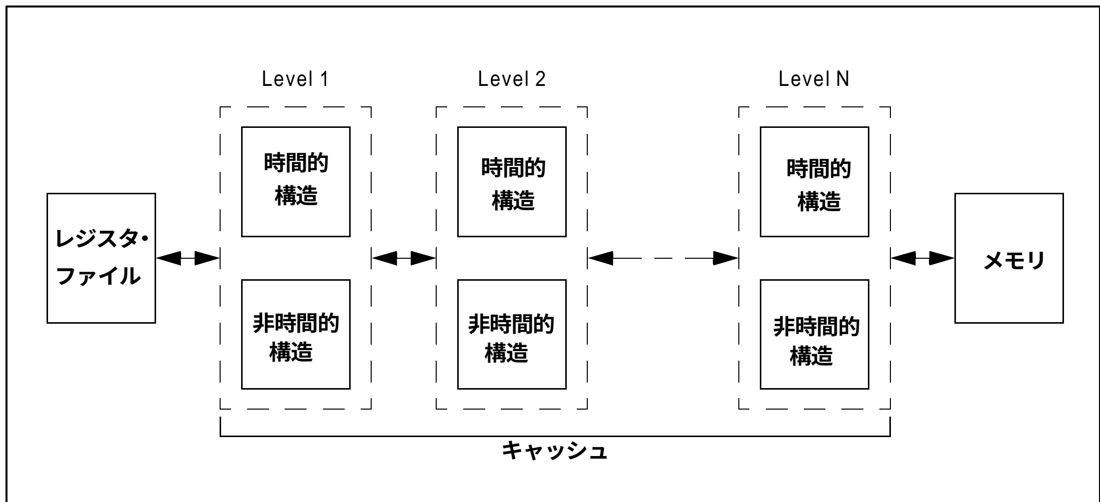
ニーモニック	操作
ld.a, ldf.a, ldfp.a	GR および FR のアドバンスド・ロード
st, st.rel, st8.spill, stf, stf.spill	GR および FR のストア
cmpxchg, fetchadd, xchg	GR セマフォ
ld.c.clr, ld.c.clr.acq, ldf.c.clr, ldfp.c.clr	GR および FR のチェック・ロード、ALAT のヒットでクリアされる
ld.c.nc, ldf.c.nc, ldfp.c.nc	GR および FR のチェック・ロード、ALAT のミスで再割り当てされる
ld.sa, ldf.sa, ldfp.sa	GR および FR のスペキュレーティブ・アドバンスド・ロード
chk.a.clr, chk.a.nc	GR および FR のアドバンスド・ロード・チェック
invala	すべての ALAT エントリを無効にする
invala.e	GR または FR の個々の ALAT エントリを無効にする

4.4.6 メモリ階層の制御と整合性

4.4.6.1 階層の制御とヒント

アクセスしているデータに時間的局所性を持たせるかどうかを指定するための IA-64 メモリ・アクセス命令が定義されている。また、メモリ・アクセス命令によって、そのアクセスがどのメモリ階層レベルに影響を及ぼすかを指定できる。これを図 4-4 のメモリ階層アーキテックチャ図に示す。このアーキテクチャでは、レジスタ・ファイルとメモリの間に 0 個以上のレベルのキャッシュがあり、各レベルは 2 つの並列の構造、すなわち時間的構造と非時間的構造から成っている 。この図はデータ・アクセスに適用され、命令アクセスには適用されない。

図 4-4. メモリ階層



時間的構造においては、時間的局所性によってアクセスされたメモリをキャッシュする。非時間的構造においては、時間的局所性なしにアクセスされたメモリをキャッシュする。どちらの構造も、メモリ・アクセスが空間的局所性を持っているものと想定している。個々の時間的構造や非時間的構造の有無とキャッシュのレベル数は、プロセッサによって異なる。

アロケーションの制御のために 3 つのメカニズムが定義されている。局所性ヒント、明示的プリフェッチ、および暗黙的プリフェッチである。局所性ヒントは、ロード、ストア、および明示的プリフェッチ (`lfetch`) 命令によって指定される。1 つの局所性ヒントで、1 つの階層レベル (例えば、1、2 、すべて) を指定する。ある階層レベルに対して時間的であるアクセスは、より下位の (番号が大きい) すべてのレベルに対して時間的であると見なされる。ある階層レベルに対して非時間的であるアクセスは、より下位のすべてのレベルに対しては時間的であると見なされる。階層の中でヒントで指定されているキャッシュ・ラインよりも近いキャッシュ・ラインが見つかった場合でも、そのラインのレベルは下げられない。これによって `lfetch` 命令を使ってラインを正確に管理でき、その後に `.nta` コンプリータを指定してロードおよびストアすることで、階層の中でのそのレベルを保持することができる。たとえば、プリフェッチによって `.nt2` ヒントを指定した場合、データはレベル 3 にキャッシュされる。その後のロードおよびストアで `.nta` を指定して、データをレベル 3 に保持しておくことができる。

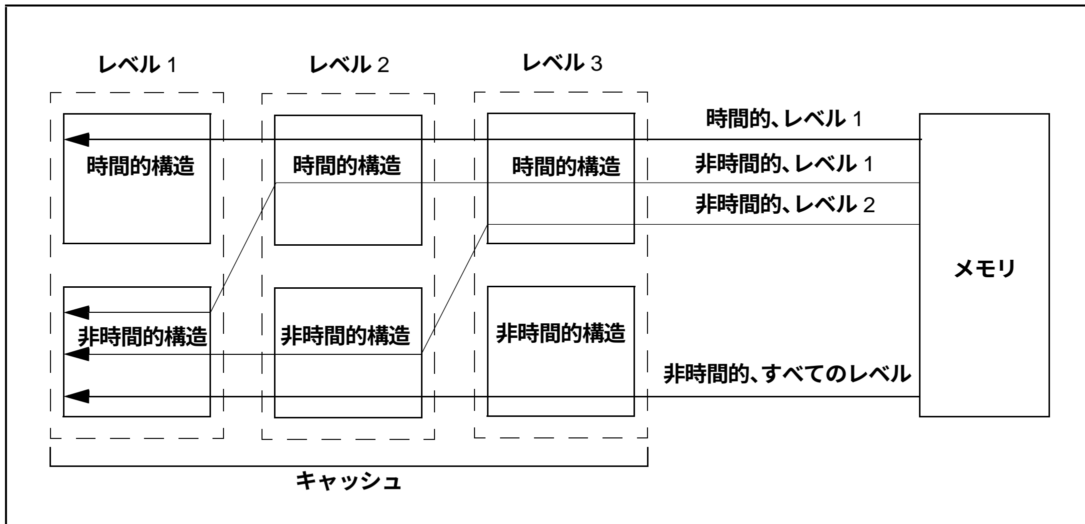
局所性ヒントは、プログラムの機能上の動作には影響を与えず、コードの機能設計では無視してもよい。局所性ヒントは、表 4-18 に示すように、ロード、ストアおよび明示的プリフェッチの命令で使用できる。命令アクセスは、レベル 1 に対して時間的局所性と非時間的局所性の両方を持っているとみなされる。

表 4-18. 各命令クラスによって指定される局所性ヒント

ニーモニック	局所性ヒント	命令タイプ		
		ロード	ストア	lfetc、lfetch.fault
none	時間的、レベル 1	x	x	x
nt1	非時間的、レベル 1	x		x
nt2	非時間的、レベル 2			x
nta	非時間的、すべてのレベル	x	x	x

それぞれの局所性ヒントによって、メモリ階層の中の特定のアロケーション・パスを暗黙に指定する。局所性ヒントに対応するアロケーション・パスを図 4-5 に示す。アロケーション・パスにより、参照するデータを含むラインが最適に割り当てられる構造を指定する。そのラインがすでに階層の中の同じレベルまたはそれ以上のレベルにある場合、移動は行われない。データを時間的構造にキャッシュするべきであるというヒントは、それが近い将来に読み取られることを想定している。

図 4-5. メモリ階層でサポートされるアロケーション・パス



明示的プリフェッチは、ライン・プリフェッチ命令 (lfetch、lfetch.fault) によって定義される。lfetch 命令は、アドレス指定されたバイトを含むラインをメモリ階層の中の局所性ヒントによって指定された位置に移動させる。そのラインがすでに階層の中の同じレベルまたはそれ以上のレベルにある場合、移動は行われない。lfetch 命令および lfetch.fault 命令には即値およびレジスタのポスト・インクリメントが定義されている。lfetch 命令は例外を発生せず、プログラムの動作に影響を

与えず、プログラム設計上無視してもよい。`lfetch.fault` 命令は、`lfetch` 命令と全く同じようにメモリ階層に影響を与えるが、1 バイトのロード命令と同様の例外を生成する。

暗黙のプリフェッチでは、ロード、ストア、`lfetch` 命令および `lfetch.fault` 命令でのアドレスのポスト・インクリメントに基づく。ポスト・インクリメントされたアドレスを含むラインは、元のロード、ストア、`lfetch` 命令または `lfetch.fault` 命令の局所性ヒントに従ってメモリ階層内で移動する。ラインがすでに階層の中の同じレベルまたはそれ以上のレベルにある場合、移動は行われない。暗黙のプリフェッチは例外を発生せず、プログラムの動作に影響を与えず、プログラム設計上無視してもよい。

ロードではこのほかに、`ld.bias` ロード・タイプのヒントを使用できる。これはアドレス指定されたデータを含むラインの排他的所有権を取得するよう指示する。bias ヒントはプログラムの機能に影響を与えず、プログラム設計上無視してもよい。

`fc` 命令は、メモリ階層でのメモリより上位のすべてのレベルのキャッシュ・ラインを無効にする。キャッシュ・ラインがメモリと整合しない場合は、無効にする前にキャッシュ・ラインがメモリにコピーされる。

表 4-19 に、メモリ階層の制御命令とヒントのメカニズムを示す。

表 4-19. メモリ階層の制御命令とヒントのメカニズム

ニーモニック	操作
ロードの .nt1 および .nta コンプリータ	使用ヒントのロード
ストアの .nta コンプリータ	使用ヒントのストア
ロードおよびストアのポスト・インクリメント・アドレスでのプリフェッチ・ライン	ヒントのプリフェッチ
.nt1、.nt2、および .nta ヒントによる lfetch、lfetch.fault	ラインのプリフェッチ
`fc`	キャッシュのフラッシュ

4.4.6.2 メモリの整合性

あるプロセッサによる IA-64 命令アクセスは、他のプロセッサによる命令アクセスやデータ・アクセスに対してコヒーレントでない。また、あるプロセッサによる命令アクセスは、そのプロセッサによるデータ・アクセスに対してもコヒーレントでない。そのため、ハードウェアは、プロセッサの命令キャッシュを (プロセッサ自身のデータ・キャッシュを含めた) どのプロセッサのデータ・キャッシュに対しても整合させる必要はない。また、ハードウェアは、プロセッサの命令キャッシュを他のプロセッサの命令キャッシュと整合させる必要もない。同じコヒーレンス・ドメインの中の異なるプロセッサからのデータ・アクセスは、相互にコヒーレントである。この整合性は、ハードウェアによって提供される。同じプロセッサからのデータ・アクセスは、データ依存の規則に従う。後述の「メモリ・アクセスの順序」を参照のこと。

コヒーレンスが保持されるメカニズムは、プロセッサによって異なる。データおよび命令をキャッシュするための個別の構造または統合された構造は、アーキテクチャ上では認識されない。このコンテキスト内では、アロケーションとフラッシュという 2 通りのデータ・メモリ階層に対する制御がある。アロケーションとは階層の中でのプロセッサに近づく移動 (小さい番号のレベルへの移動) を言い、フラッシュとは階層の中でのプロセッサから離れる移動 (大きい番号のレベルへの移動) を言う。アロケーションとフラッシュの単位はライン・サイズで表される。アーキテクチャ的に検出可能な最小のライン・サイズは 32 バイトである (32 バイト境界にアライメントされる)。ハードウェア設計上のライン・サイズはこれより小さくてもよい。その場合、1 つのアロケーション・イベントおよびフラッシュ・イベントに対して複数のラインを移動させる必要がある。また、設計によっては、32 バイトより大きい単位でアロケーションおよびフラッシュを行うことができる。

あるプロセッサからの書き込みがそのプロセッサおよび他のプロセッサの命令ストリームから検出可能であるためには、書き込まれるラインをメモリにフラッシュする必要がある。ソフトウェア上では、そのために `fc` 命令を使用できる。DMA デバイスによるメモリ更新は、プロセッサの命令アクセスおよびデータ・アクセスに対してコヒーレントである。プロセッサの命令キャッシュおよびデータ・キャッシュ間の DMA デバイスによるメモリ更新に対するコーヒレンシは、ハードウェアによって提供される。プログラム上で自分の命令を変更する場合には、`sync.i` 命令よび `srlz.i` 命令を使って、プログラムの所定の時点で、事前にコーヒレンシ・アクションが行われるようにする。自己修正コードの例は、7-187 ページの `sync.i` の項に示している。

4.4.7 メモリ・アクセスの順序

メモリ・データ・アクセスの順序は、同じメモリ位置への RAW (リード・アフター・ライト)、WAW (ライト・アフター・ライト)、および WAR (ライト・アフター・リード) のデータ依存関係を満たしていなければならない。また、メモリへの書き込みおよびフラッシュは、コントロール依存に従わなければならない。読み取り、書き込み、フラッシュは、これらの制約に従っている限りは、指定されたプログラム順序とは違う順序で発生してもよい。命令アクセスとデータ・アクセスの間、または 2 つの命令アクセス間には定義された順序はない。以下で説明するメカニズムは、特定のメモリ・アクセス順序を強制するために定義されたものである。以下の説明では、" 先行の " および " 後続の " という用語は、プログラムによって指定された順序を指す。" 可視 " という用語は、アーキテクチャ的に参照可能なすべてのメモリ・アクセスの影響を指す (これは少なくともメモリの読み取りまたは書き込みを含む)。

メモリ・アクセスは、4 つのメモリ順序付けの語彙、すなわち unordered (順序付けなし)、release (解放)、acquire (取得)、fence (フェンス) のいずれかに従う。unordered でのデータ・アクセスは、任意の順序で可視になる。release でのデータ・アクセスは、先行のデータ・アクセスがすべて可視になった後で可視になる。acquire でのデータ・アクセスは、後続のデータ・アクセスが可視になる前に可視になる。fence オペレーションは、release および acquire の語彙を双方向のフェンスに組み合せる。つまり、先行のすべてのデータ・アクセスが可視になった後で、後続のデータ・アクセスが可視になる。

明示的なメモリ順序は、一連の命令セットの形で定義される。すなわち、順序付けされたロード命令および順序付けされたチェック・ロード命令 (`ld.acq`、`ld.c.clr.acq`)、順序付けされたストア命令 (`st.rel`)、セマフォ命令 (`cmpxchg`、`xchg`,`fetchadd`)、およびメモリ・フェンス命令 (`mf`) である。`ld.acq`、および `ld.c.clr.acq` 命令は、acquire 語彙に従う。`st.rel` 命令は、release 語彙に従う。`mf` 命令は、`fence` オペレーションである。`xchg`、`fetchadd.acq` および `cmpxchg.acq` の各命令は、acquire 語彙に従う。`cmpxchg.rel`、および `fetchadd.rel` 命令は、release 語彙に従う。セマフォ命令はまた、暗黙の順序をもつ。書き込みがある場合は、常に読み取りの後に行われる。読み取りと書き込みはアトミックに行われ、同じメモリ領域への割り込みアクセスは行われない。

表 4-20 に、順序付けにおけるメモリ・アクセスと種々の順序付けの語彙の相互関係を示す。"O" は、最初の参照と 2 番目の参照がその順序で行われることを示す。"-" は、データ依存 (および書き込みとフラッシュのコントロール依存) 以外に暗黙の順序がないことを示す。

表 4-20. メモリ順序規則

最初の参照	2 番目の参照			
	fence	acquire	release	unordered
fence	O	O	O	O
acquire	O	O	O	O
release	O	–	O	–
unordered	O	–	O	–

表 4-21 に、キャッシュ可能なメモリに関連するメモリ順序命令を示す。

表 4-21. メモリ順序命令

ニーモニック	操作
`ld.acq, ld.c.clr.acq`	順序付けされたロードと順序付けされたチェック・ロード
`st.rel`	順序付けされたストア
`xchg`	メモリと汎用レジスタの交換
`cmpxchg.acq, cmpxchg.rel`	メモリと汎用レジスタの条件付き交換
`fetchadd.acq,fetchadd.rel`	メモリへの即値の追加
`mf`	メモリの順序付けフェンス

4.5 分岐命令

分岐命令は、新しいアドレスにコントロール・フローを渡す。分岐ターゲットはバンドルにアライメントされている。つまり、コントロールは常にターゲット・バンドルの最初の命令スロット (スロット 0) に渡される。分岐命令は命令グループの中の最後の命令でなくてもよい。

実際、命令グループには、任意の数の分岐を含めることができる (通常の RAW および WAW の依存関係の必要条件が満たされている場合に限る)。分岐が行われた場合、その分岐までの命令だけが実行される。分岐の後の最初に実行される命令は、分岐のターゲットにある命令である。

分岐には IP 相対分岐と間接分岐の 2 つの種類がある。IP 相対分岐では、そのターゲットを符号付き 21 ビット変位で指定する。分岐を含んでいるバンドルの IP にこの変位が加算されてターゲット・バンドルのアドレスとなる。この変位によって ± 16M バイトの範囲での分岐が可能で、分岐はバンドルにアライメントされる。間接分岐は、分岐レジスタを使ってターゲット・アドレスを指定する。

表 4-22 に示すように、いくつかの分岐タイプがある。条件付き分岐 `br` では、指定したプレディケートが 1 である場合に分岐が行われ、そうでない場合は分岐は行われない。条件付きコール分岐 `br.call` は、`br` と同じ操作を行うほかに、指定した分岐レジスタにリンク・アドレスを書き込み、汎用レジスタ・スタックを調整する (4-1 ページの「レジスタ・スタック」を参照)。条件付きリターン `br.ret` は、間接の条件付き分岐と同じ操作を行うほかに、汎用レジスタ・スタックを調整する。無条件分岐、コール、およびリターンは、分岐命令のプレディケートとして PR 0 (常に 1) を指定することによって実行される。

表 4-22. 分岐タイプ

ニーモニック	機能	分岐条件	分岐先アドレス
`br.cond` または `br`	条件付き分岐	修飾プレディケート	IP 相対分岐または間接
`br.call`	条件付きプロシージャ・コール	修飾プレディケート	IP 相対分岐または間接
`br.ret`	条件付きプロシージャ・リターン	修飾プレディケート	間接
`br.ia`	IA-32 命令セットの呼び出し	無条件	間接
`br.cloop`	カウント指定ループ分岐	ループ・カウント	IP 相対
`br.ctop`, `br.cexit`	モジュロ・スケジュール型のカウント指定ループ	ループ・カウントおよびエピローグ・カウント	IP 相対
`br.wtop`, `br.wexit`	モジュロ・スケジュール型の while ループ	修飾プレディケートおよびエピローグ・カウント	IP 相対

カウント指定ループ・タイプ (CLOOP) では、ループ・カウント (LC) アプリケーション・レジスタを使用する。LC が非ゼロならばデクリメントされ、分岐が処理される。LC がゼロならば分岐は行われない。モジュロ・スケジュー

ル型のループ・タイプの分岐 (CTOP、CEXIT、WTOP、WEXIT) については、4-33 ページの「モジュロ・スケジュール型ループのサポート」で説明している。ループ・タイプの分岐 (CLOOP、CTOP、CEXIT、WTOP、WEXIT) は、バンドルのスロット 2 でだけ実行できる。スロット 0 または 1 でループ・タイプの分岐を実行すると Illegal Operation (無効操作) フォルトが発生する。

分岐レジスタと汎用レジスタの間でデータを移動する命令が定義されている (`mov = br`,`mov br =`)。表 4-23 に、分岐関連の状態および命令を示す。

表 4-23. 分岐関連の状態

レジスタ	機能
BRs	分岐レジスタ
PRs	プレディケート・レジスタ
CFM	現在のフレーム・マーカ
PFS	以前のファンクション状態アプリケーション・レジスタ
LC	ループ・カウント・アプリケーション・レジスタ
EC	エピローグ・カウント・アプリケーション・レジスタ

表 4-24. 分岐に関係する命令

ニーモニック	操作
`br`	分岐
`mov =br`	BR から GR への移動
`mov br=`	GR から BR への移動

4.5.1 モジュロ・スケジュール型ループのサポート

ソフトウェアでパイプライン化されるループは、ローテートするレジスタおよびループ・タイプの分岐によってサポートされる。ループに対するソフトウェアによるパイプライン処理は、1 機能ユニットのハードウェアによるパイプライン処理と同じである。ループ本体は、複数の " ステージ " に分割され、各ステージに 0 個 以上の命令が含まれる。モジュロ・スケジュール型ループにはプロローグ、カーネル、エピローグの 3 つのフェーズがある。プロローグ・フェーズでは、一巡するごとにループの新しい反復が開始する (ソフトウェア・パイプラインをフィルする)。カーネル・フェーズでは、パイプラインはいっぱいになっている。一巡するごとにループの新しい反復が開始し、前の反復が終了する。エピローグ・フェーズでは、新しい反復は開始されず、直前の反復が完了する (ソフトウェア・パイプラインをドレインする)。

各ステージに、そのステージにある命令の起動を制御するプレディケートが割り当てられる (このプレディケートを " ステージ・プレディケート " と言う)。ソフトウェア・パイプライン・ループでのステージ・プレディケートおよびレジスタのパイプライン効果をサポートするために、プレディケート・レジスタ・ファイルおよび浮動小数点レジスタ・ファイル (PR16 ～ PR63 お

および FR32 〜 FR127) の固定サイズ領域、および汎用レジスタ・ファイルの可変サイズ領域が " ローテート " 領域として定義されている。汎用レジスタ・ファイルの中のローテート領域のサイズは、`alloc` 命令の即値によって決定される。この即値は 0 または 8 の倍数でなければならない。汎用レジスタのローテート領域は、GR32 から始まり、その相対サイズに応じてローカル領域および出力領域をオーバレイするように定義される。ステージ・プレディケートは、プレディケート・レジスタ・ファイルのローテート領域に割り当てられる。カウント指定ループでは、PR16 がアーキテクチャ的に最初のステージ・プレディケートとして定義され、これ以降のステージ・プレディケートには、より大きいプレディケート・レジスタ番号が割り当てられる。while ループでは、最初のステージ・プレディケートは、どのローテート・プレディケートでもかまわず、後続のステージ・プレディケートには、より大きいプレディケート・レジスタ番号が割り当てられる。ソフトウェア上では、ループに入る前にステージ (ローテート)・プレディケートを初期設定しなければならない。`alloc` 命令では、CFM の中のすべてのローテート・レジスタ・ベース (rrb) がゼロでない限り、レジスタ・スタック・フレームのローテート部分のサイズを変更できない。`clrrrb` 命令を使ってすべての rrb をゼロにクリアできる。`clrrrb.pr` 形式を使えば、プレディケート・レジスタの rrb だけをクリアできる。`clrrrb` 命令は命令グループの中の最後の命令でなければならない。

ソフトウェア・パイプライン・ループ・タイプの分岐が実行されると、1 レジスタ分だけローテートが行われる。レジスタはラップ・アラウンド方式で大きなレジスタ番号に向かってローテートする。たとえば、1 回のローテートの後、レジスタ X の値はレジスタ X+1 に置かれる。X がローテート・レジスタの最上位アドレスである場合には、その値は最下位アドレスのローテート・レジスタにラップされる。ローテートは、CFM の中のローテート・レジスタ・ベース (rrb) の値に従ってレジスタをリネームすることによって実現される。rrb は、3 つのローテート・レジスタ・ファイルのそれぞれについて定義されている。すなわち、汎用レジスタは CFM.rrb.gr、浮動小数点レジスタは CFM.rrb.fr、プレディケート・レジスタは CFM.rrb.pr である。汎用レジスタは、ローテート領域のサイズがゼロでないときだけローテートする。浮動小数点レジスタおよびプレディケート・レジスタは常にローテートする。ローテートが行われる場合、3 つの rrb のうち 2 つまたはすべてが同時にデクリメントされる。各 rrb は、それぞれのローテート領域のサイズのモジュロ (例、rrb.fr では 96) に従ってデクリメントされる。ローテート・レジスタのリネーム・メカニズムの上記以外の動作は、ソフトウェアからは認識されない。rrb を変更する命令を表 4-25 に示す。

表 4-25. RRB を変更する命令

ニーモニック	操作
`clrrrb`	すべての rrb のクリア
`clrrrb.pr`	rrb.pr のクリア
`br.call`	すべての rrb のクリア
`br.ret`	PFM.rrb からの CFM.rrb の復元
`br.ctop, br.cexit, br.wtop, and br.wexit`	すべての rrb のデクリメント

ソフトウェア・パイプライン・ループには、カウント指定ループと while ループの 2 つの分岐タイプがある。どちらのタイプにも top と exit の 2 通りの形式がある。"top" 形式は、ループ判定がループ本体の最後に置かれるときに使用される。分岐が発生するとループが続行し、分岐が発生しないとループが終了する。"exit" 形式は、ループ判定がループ本体の最後以外の場所に置かれるときに使用される。分岐が発生しないとループが続行し、分岐が発生するとループが終了する。"exit" 形式は、展開型のパイプライン・ループの中間点でも使用される。

カウント指定ループの分岐における分岐条件は、カウント指定ループのタイプ (ctop または cexit)、ループ・カウント・アプリケーション・レジスタ (LC) の値、およびエピローグ・カウント・アプリケーション・レジスタ (EC) の値によって決定される。カウント指定ループの分岐においては、修飾プレディケートは使用しない。LC はカウント指定ループの反復数より 1 小さい値に初期設定され、EC はループ本体を分割するステージの数に初期設定される。LC が 0 より大きいときに、分岐判定によりループが続行され、LC がデクリメントされ、レジスタがローテートされ (rrb がデクリメントされる)、ローテートの後に PR 16 が 1 にセットされる (ループ・タイプの分岐のたびに、分岐によって PR63 が書き込まれ、ローテートの後にこの値が PR 16 に置かれる)。

LC が 0 のときにカウント指定ループの分岐が実行されると、エピローグ・フェーズが開始される。エピローグ・フェーズでは、EC が 1 より大きい場合、分岐判定によりループが続行され、EC がデクリメントされ、レジスタがローテートされ、ローテートの後に PR 16 が 0 にクリアされる。LC が 0 で EC が 1 のときにカウント指定ループの分岐が実行されると、ループが終了する。つまり、分岐判定によりループが終了し、EC がデクリメントされ、レジスタがローテートされ、ローテートの後に PR 16 が 0 にクリアされる。LC と EC の両方が 0 のときにカウント指定ループの分岐が実行されると、分岐判定によってループが終了する。LC、EC、および rrb は変更されず (ローテートされない)、PR 63 が 0 にクリアされる。LC と EC の両方が 0 になるのは、一部の最適化された展開型のソフトウェア・パイプライン・ループで、cexit 分岐のターゲットが次のシーケンシャル・バンドルに設定され、ループ・トリップ・カウントが展開数で割り切れない場合である。

while ループの分岐の判定は while ループのタイプ (wtop または wexit)、修飾プレディケートの値、および EC の値 によって決まる。while ループの分岐においては LC は使用しない。修飾プレディケートの値が 1 であるとき、分岐判定によりループは続行され、レジスタがローテートされ、ローテートの後に PR 16 が 0 にクリアされる。修飾プレディケートが 0 で EC が 1 より大きいとき、分岐判定によりループは続行され、EC がデクリメントされ、レジスタがローテートされ、ローテートの後に PR 16 が 0 にクリアされる。修飾プレディケートは、カーネル・フェーズでは 1 であり、エピローグ・フェーズでは 0 である。プロローグ・フェーズでは、修飾プレディケートは、パイプライン化された while ループのプログラミングに使用したスキームに従って、0 または 1 になる。修飾プレディケートが 0 で、EC が 1 であるときに while ループ分岐が実行されると、ループが終了する。つまり、分岐判定によりループが終了し、EC がデクリメントされ、レジスタがローテートされ、ローテートの後に PR 16 が 0 にクリアされる。修飾プレディケートが 0 で、EC が

0 のときに while ループ分岐が実行されると、分岐判定によりループが終了する。EC および rrb は変更されず (ローテートされない)、PR 63 は 0 にクリアされる。

while ループでは、EC の初期設定は、パイプライン化された while ループのプログラミングに使用したスキームに従って異なる。多くの場合、while ループの分岐における最初の有効な条件は、プロローグのいくつかのステージまで計算されない。そのため、while ループのソフトウェア・パイプラインには、多くの場合、いくつかのスペキュレーティブ・プロローグ・ステージがある。これらのステージ中に、修飾プレディケートに対しては、ループのプログラミングに使用したスキームに従って、0 または 1 に設定できる。修飾プレディケートがプロローグの全期間にわたって 1 である場合、EC はエピローグ・フェーズでだけデクリメントされ、エピローグ・ステージの数より 1 大きい数に初期設定される。修飾プレディケートがプロローグのスペキュレーティブ・ステージの間だけ 0 である場合、プロローグのこのステージの間 EC はデクリメントされ、それに対応して EC の初期設定値が大きくなる。

4.5.2 分岐予測ヒント

分岐予測を改善するために、分岐の動作に関する情報をプロセッサに提供することができる。この情報は、分岐命令の一部として分岐ヒントを使ってコード化できる (これをヒントと言う)。ヒントは、プログラムの機能上の動作には影響を与えず、プロセッサからは無視される。

分岐命令によって 3 つのタイプのヒントを提供できる。

- **分岐有無予測ストラテジ**：COND、CALL および RET タイプの分岐において、プロセッサが分岐条件を予測する方法を記述する (ループ・タイプの分岐では、予測は LC および EC に基づく)。ヒントとして提供できる推奨ストラテジを表 4-26 に示す。

表 4-26. 分岐有無予測ヒント

コンプリータ	ストラテジ	操作
spnt	静的に処理されない	この分岐を無視し、この分岐に予測リソースを割り当てない。
sptk	静的に処理する	常に分岐すると予測される。この分岐に予測リソースを割り当てない。
dpnt	動的に処理されない	動的予測ハードウェアを使用する。この分岐に関する動的履歴情報がない場合、分岐しないと予測する。
dptk	動的に処理する	動的予測ハードウェアを使用する。この分岐に関する動的履歴情報がない場合、分岐すると予測する。

- **シーケンシャル・プリフェッチ**：プロセッサが分岐ターゲットでプリフェッチするコードの量を示す (表 4-27 を参照)。

表 4-27.　シーケンシャル・プリフェッチ・ヒント

コンプリータ	シーケンシャル・プリフェッチ・ヒント	操作
few	プリフェッチするラインが少ない	分岐ターゲットでコードをプリフェッチするとき、少数のラインをプリフェッチしてから停止する (ライン数はプロセッサによって異なる)。
many	プリフェッチするラインが多い	分岐ターゲットでコードをプリフェッチするとき、多数のラインをプリフェッチする (ライン数はプロセッサによって異なる)。

- **予測リソースの割り当て解除：**ハードウェアが分岐予測リソースをより効率的に管理できるように、情報を再利用できるようにする。通常は、予測リソースによって、最後に実行された分岐の記録を保持する。しかし、最後に実行した分岐を記録しても役に立たない場合がある (その分岐をしばらくは再使用しない、あるいはこの分岐を再使用するより前にヒント命令によってその情報が再提供される)。このような場合には、このヒントを使って、予測リソースを解放することができる。

表 4-28.　予測リソースの割り当て解除ヒント

コンプリータ	操作
none	割り当て解除を行わない
clr	分岐情報の割り当てを解除する

4.6　マルチメディア命令

マルチメディア命令 (表 4-29 を参照) は、汎用レジスタを 8 個の 8 ビット要素、4 個の 16 ビット要素、または 2 個 の 32 ビット要素の連結として扱い、各要素を独立かつ並列に操作する。要素は常に、汎用レジスタ内の自然境界にアライメントされる。ほとんどのマルチメディア命令は、複数の要素サイズを扱うように定義されている。マルチメディア命令として、算術演算、シフト、データ配列の 3 つのクラスが定義されている。

4.6.1　並列算術演算

3 つの形式の並列加算 / 減算命令が定義されている。モジュロ (padd、psub)、符号付き飽和 (padd.sss、psub.sss)、および符号なし飽和 (padd.uuu、padd.uus、psub.uuu、psub.uus) である。モジュロ形式では、結果が結果要素の範囲内で表現可能な最大値または最小値でラップ・アラウンドする。飽和形式では、結果が結果要素の範囲内で表現可能な最大値より大きい、または最小値より小さい場合、それぞれ結果要素の範囲内の最大値または最小値にクランプされる。符号付き飽和形式は、両方のソースを符号付きとして取り扱い、結果を符号付きの範囲の上限 / 下限にクランプする。符号なし飽和形式は、1 つのソース・オペランドを符号なしとして扱い、

結果を符号なし範囲の上限 / 下限にクランプする。この形式には、2 番目のソース・オペランドを符号付きとして扱うための .uus と符号なしとして扱うための .uuu の 2 種類の応用形式が定義されている。

並列平均命令 (pavg、pavg.raz) は、各ソース・オペランドの対応する要素を加算し、それぞれの結果を 1 ビット右にシフトする。この命令の簡単な形式として、それぞれの和の最上位ビットの繰り上がりが、結果要素の最上位ビットに書き込まれる。ゼロから離れる方向の丸め形式では、シフトの前に、それぞれの和に 1 が加算される。並列平均減算命令 (pavgsub) は、ソース・オペランドの差に対して同様の操作を行う。

並列左シフトおよび加算命令 (pshladd) は、第 1 ソース・オペランドの要素に左シフトを実行し、次にそれを第 2 ソース・オペランドの対応する要素に加算する。符号付き飽和がシフト演算と加算演算の両方に実行される。並列右シフトおよび加算命令 (pshradd) は、pshladd と同様の操作を行う。これらの命令はどちらも、2 バイト要素に対してだけ定義されている。

並列比較命令 (pcmp) は、両方のソース・オペランドの対応する要素を比較し、ターゲットの対応する要素に、== または > の関係に従って、オール 1 (真の場合) またはオール 0 (偽の場合) を書き込む。

並列乗算および右シフト命令 (pmpy.r) は、両方のソース・オペランドの対応する 2 つの偶数の符号付き 2 バイト要素を乗算し、結果をターゲットの 2 つの 4 バイト要素に書き込む。pmpy.l 命令は、奇数の 2 バイト要素に対して同様の操作を実行する。並列乗算および右シフト命令 (pmpyshr、pmpyshr.u) は、両方のソース・オペランドの対応する 2 バイト要素を乗算し、4 つの 4 バイトの結果を生成する。4 バイトの結果は、命令によって指定された数だけ (0、7、15、または 16 ビット) 右にシフトされる。次に、シフトされた 4 バイトの結果の下位 2 バイトがターゲット・レジスタにストアされる。

並列絶対差累積命令 (psad) は、対応する 1 バイト要素の絶対差を累算し、結果をターゲットに書き込む。

並列最小値命令 (pmin.u,pmin) および並列最大値命令 (pmax.u、pmax) はそれぞれ、ターゲットの対応する 1 バイト、または 2 バイトの要素の最小値または最大値を生成する。1 バイト要素は符号なしの値として扱われ、2 バイト要素は符号付きの値として扱われる。

表 4-29. 並列演算命令

ニーモニック	操作	1 バイト	2 バイト	4 バイト
padd	並列モジュロ加算	x	x	x
padd.sss	符号付き飽和形式の並列加算	x	x	
padd.uuu, padd.uus	符号なし飽和形式の並列加算	x	x	
psub	並列のモジュロ減算	x	x	x
psub.sss	符号付き飽和形式の並列減算	x	x	

表 4-29. 並列演算命令 (続き)

ニーモニック	操作	1 バイト	2 バイト	4 バイト
psub.uuu, psub.uus	符号なし飽和形式の並列減算	x	x	
pavg	並列算術平均	x	x	
pavg.raz	ゼロから離れる方向の丸め形式での並列算術平均	x	x	
pavgsub	並列平均減算	x	x	
pshladd	符号付き飽和形式の並列左シフトおよび加算		x	
pshradd	符号なし飽和形式の並列右シフトおよび加算		x	
pcmp	並列比較	x	x	x
pmpy.l	符号付き奇数要素の並列乗算			x
pmpy.r	符号付き偶数要素の並列乗算			x
pmpyshr	符号付き並列乗算および右シフト		x	
pmpyshr.u	符号なし並列乗算および右シフト		x	
psad	並列絶対差累積	x		
pmin	並列最小値	x	x	
pmax	並列最大値	x	x	

4.6.2 並列シフト

並列左シフト命令 (pshl) は、第 1 ソース・オペランドの各要素を、汎用レジスタまたは即値に指定されているカウントだけ左シフトする。並列右シフト命令 (pshr) は、1 つのソース・オペランドの各要素に対して、汎用レジスタまたは即値に指定されているカウントをもとに算術的右シフトを実行する。pshr.u 命令は、符号なし右シフトを実行する。表 4-30 に、並列シフト命令を示す。

表 4-30. 並列シフト命令

ニーモニック	操作	1 バイト	2 バイト	4 バイト
pshl	並列左シフト		x	x
pshr	並列符号付き右シフト		x	x
pshr.u	並列列符号なし右シフト		x	x

4.6.3 データ配列

ミックス右命令 (mix.r) は、両方のソース・オペランドから偶数の要素をターゲットにインタリーブする。ミックス左命令 (mix.l) は、奇数の要素をインタリーブする。アンパック・ロー命令 (unpack.l) は、各ソース・オペランドの下位 4 バイトの要素をターゲット・レジスタにインタリーブする。アンパック・ハイ命令 (unpack.h）は、上位 4 バイトの要素をインタリーブする。パック命令 (pack.sss、pack.uss) は、32 ビットまたは 16 ビット要素をそれぞれ 16 ビットまたは 8 ビット要素に変換する。両方のソース・オペランドの要素の下半分が抽出されて、ターゲット・レジスタの要素に書き込まれる。pack.sss 命令は、抽出した要素を符号付きの値として扱い、この値に符号付き飽和を実行する。pack.uss 命令は、符号なし飽和を実行する。mux 命令 (mux) は、ソース・オペランドの中の個別の 2 バイト または 4 バイト要素を指定された機能に従ってターゲットの中の任意の位置にコピーする。2 バイト要素の場合、8 ビットの即値によって可能なすべての組み合わせを指定できる。1 バイト要素の場合、可能な 5 つの機能 (反転、ミックス、シャッフル、交互、ブロードキャスト) からコピー機能が選択される。表 4-31 に、種々のタイプの並列データ配列命令を示す。

表 4-31. 並列データ配列命令

ニーモニック	操作	1 バイト	2 バイト	4 バイト
mix.l	両方のソース・オペランドから奇数の要素をインタリーブする	x	x	x
mix.r	両方のソース・オペランドから偶数の要素をインタリーブする	x	x	x
mux	個々のソース・オペランド要素を任意にコピーする	x	x	
pack.sss	大きい要素を小さい要素に変換して符号付き飽和を実行する		x	x
pack.uss	大きい要素を小さい要素に変換して符号なし飽和を実行する		x	
unpack.l	両方のソース・オペランドから下位要素をインタリーブする	x	x	x
unpack.h	両方のソース・オペランドから上位要素をインタリーブする	x	x	x

4.7 レジスタ・ファイルの転送

表 4-32 に、汎用レジスタ・ファイルと、浮動小数点、分岐、プレディケート、パフォーマンス・モニタ、プロセッサ識別およびアプリケーションの各レジスタ・ファイルとの間で値を移動させるために定義されている命令を示す。いくつかの転送命令は同じニーモニック (mov) を共用する。オペランドの値で、アクセスするレジスタ・ファイルを識別する。

表 4-32. レジスタ・ファイル転送命令

ニーモニック	操作
`getf.exp, getf.sig`	FR の指数または仮数を GR に移動
`getf.s, getf.d`	単 / 倍精度メモリ形式を FR から GR に移動
`setf.s, setf.d`	単 / 倍精度メモリ形式を GR から FR に移動
`setf.exp, setf.sig`	GR を FR の指数または仮数に移動
`mov =br`	BR から GR に移動
`mov br=`	GR から BR に移動
`mov =pr`	プレディケートから GR に移動
`mov pr=, mov pr.rot=`	GR からプレディケートに移動
`mov ar=`	GR から AR に移動
`mov =ar`	AR から GR に移動
`sum, rum`	ユーザ・マークの設定およびリセット
`mov =pmd[...]`	パフォーマンス・モニタ・データ・レジスタから GR に移動
`mov =cpuid[...]`	プロセッサ識別レジスタから GR に移動
`mov =ip`	命令ポインタからの移動

メモリ・アクセス命令は、汎用レジスタ・ファイルおよび浮動小数点レジスタ・ファイルだけをターゲットまたはソースとする。メモリと他のすべてのレジスタ・ファイル
(浮動小数点レジスタ・ファイルを除く) の間での転送には、汎用レジスタ・ファイルを媒介として使用する必要がある。

汎用レジスタと浮動小数点レジスタの間の移動のために 2 つのクラスが定義されている。最初のタイプは仮数または符号 / 指数を移動する (`getf.sig`、`setf.sig`、`getf.exp`、`setf.exp`)。2 番目のタイプは、単精度または倍精度の数値全体を移動する (`getf.s`、`setf.s`、`getf.d`、`setf.d`)。また、これらの命令は、据え置き例外トークン形式の変換を実行する。

分岐レジスタと汎用レジスタの間での転送のための命令が定義されている。

プレディケート・レジスタ・ファイルと汎用レジスタの間の転送のための命令が定義されている。これらの命令はブロードサイド方式で実行し、複数のプレディケート・レジスタが並列で転送される (プレディケート・レジスタ N と汎用レジスタの ビット N の間で転送が行われる)。プレディケートへの移動命令 (`mov pr=`) では、即値によって指定されるマスクに従って、汎用レジスタを複数のプレディケート・レジスタに転送する。マスクには、それぞれのスタティック・プレディケート・レジスタ (PR 1 ～ PR 15。PR 0 は 1 にハード配線されている) について各 1 ビットと、すべてのローテート・プレディケート・レジスタ (PR16 ～ PR 63) について 1 ビットが含まれている。プレディケート・レジスタは、対応するマスク・ビットがセットされている場合に、汎用レジスタの対応するビットから書き込まれる。マスク・ビットが

クリアされている場合は、プレディケート・レジスタは変更されない。ローテート・プレディケートは、CFM.rrb.pr がゼロであるかのように転送される。CFM.rrb.pr の実際の値は無視され、変更されない。プレディケートからの移動命令 (mov =pr) は、プレディケート・レジスタ・ファイル全体を汎用レジスタ・ターゲットに転送する。

mov =pmd[ ] 命令は、パフォーマンス・モニタ・データ (PMD) レジスタから汎用レジスタに移動する。オペレーティング・システムがユーザ・レベルでのパフォーマンス・モニタ・データ・レジスタの読み取りを許可していない場合には、オール・ゼロが戻される。mov =cpuid[ ] 命令は、プロセッサ識別レジスタから汎用レジスタに移動する。

mov =ip 命令は、命令ポインタ (IP) の現在値を汎用レジスタにコピーする。

4.8 文字列とポピュレーション・カウント

特別な命令セットによって、文字データおよびビット・フィールド・データの操作を迅速に処理できる。

4.8.1 文字列

ゼロ・インデックス算出命令 (czx.l、czx.r) は、汎用レジスタ・ソース・オペランドを 8 つの 1 バイト要素または 4 つの 2 バイト要素として扱い、最初に検出したゼロ要素のインデックスを汎用レジスタ・ターゲットに書き込む。ソース・オペランドにゼロ要素がない場合は、ターゲットには可能な最大のインデックス (1 バイト形式では 8、2 バイト形式では 4) より 1 大きい定数が書き込まれる。czx.l 命令では、ソース・オペランドを左から右にスキャンする (左端の要素がインデックス 0)。czx.r 命令では、ソース・オペランドを右から左にスキャンする (右端の要素がインデックス 0)。表 4-33 に、ゼロ・インデックス算出命令を示す。

表 4-33. 文字列サポート命令

ニーモニック	操作	1 バイト	2 バイト
czx.l	最初のゼロ要素を左から右に探索する	x	x
czx.r	最初のゼロ要素を右から左に探索する	x	x

4.8.2 ポピュレーション・カウント

ポピュレーション・カウント命令 (popcnt) は、ソース・オペランド・レジスタの中の値が 1 であるビット数をターゲット・レジスタに書き込む。

intel®

IA-64 浮動小数点プログラミング・モデル 5

IA-64 浮動小数点アーキテクチャは、ANSI/IEEE のバイナリ浮動小数点演算標準 (Std.754-1985) に完全に準拠している。IEEE 標準の単精度、倍精度、拡張倍精度の実数形式が全面的にサポートされている。丸め精度を制御するための 2 つの IEEE 標準の方法がサポートされている。第 1 の方法では、結果を拡張倍精度の指数範囲に変換する。第 2 の方法では、結果を出力先の精度に変換する。積和演算、最小値・最大値演算、最小拡張倍精度形式より大きい範囲をもつレジスタ・ファイル形式などの IEEE 拡張標準もサポートされている。

5.1 データ型および形式

単精度、倍精度、拡張倍実数 (IEEE 実数型)、64 ビット符号付き整数、64 ビット符号なし整数、82 ビット浮動小数点レジスタ形式の 6 つのデータ型が直接にサポートされている。" 並列 FP" 形式、すなわち 1 対の IEEE の単精度値が浮動小数点レジスタの仮数部を使用する形式もサポートされている。7 番目のデータ型として、IEEE 式の 4 倍精度がソフトウェア・ルーチンによってサポートされる。将来のアーキテクチャ拡張によって、4 倍精度実数型のサポートが追加される可能性がある。

5.1.1 実数型

表 5-1 に、サポートされている IEEE 実数型のパラメータを示す。

表 5-1. IEEE 実数型のプロパティ

	単精度	倍精度	拡張倍精度	4 倍精度
IEEE 実数型パラメータ				
符号	+ or –	+ or –	+ or –	+ or –
E_{max}	+127	+1023	+16383	+16383
E_{min}	–126	–1022	–16382	–16382
指数バイアス	+127	+1023	+16383	+16383
精度 (ビット数)	24	53	64	113

表 5-1. IEEE 実数型のプロパティ (続き)

	単精度	倍精度	拡張倍精度	4 倍精度
IEEE メモリ形式				
メモリ形式全体のビット数	32	64	80	128
符号フィールドのビット数	1	1	1	1
指数フィールドのビット数	8	11	15	15
仮数フィールドのビット数	23	52	64	112

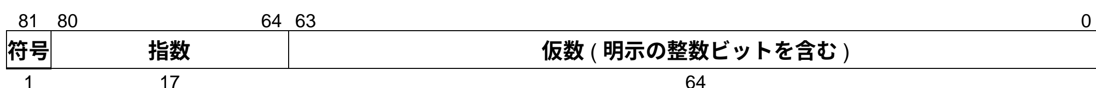
5.1.2 浮動小数点レジスタ形式

浮動小数点レジスタには整数型または実数型のデータを格納することができる。浮動小数点レジスタのデータ形式は、どちらのデータ型でも認識できる (データの情報が失われない) ように設計されている。

実数は、82 ビットの浮動小数点レジスタに 3 つのフィールドからなるバイナリ形式で格納される (図 5-1 参照)。3 つのフィールドは以下の通りである。

- 64 ビットの**仮数**フィールド (b_{63}、b_{62}、b_{61}、...b_1、b_0) には、その数の有効桁が格納される。このフィールドは、明示の整数ビット (仮数部 {63}) と 63 ビットの小数部 (仮数部 {62:0}) から成る。並列 FP データでは、仮数フィールドに 1 対の 32 ビットの IEEE 単精度実数が格納される。
- 17 ビットの**指数**フィールドは、有効桁の範囲内または範囲外の小数点の位置を指す (つまり、このフィールドで数値の大きさが決まる)。指数フィールドは、65535 (0xFFFF) でバイアスされる。指数フィールドがすべて 1 のとき、IEEE の符号付き無限大および NaN を表す。指数フィールドがすべて 0 で、仮数フィールドがすべて 0 のとき、IEEE の符号付き 0 を表す。指数フィールドがすべて 0 で、仮数フィールドが非 0 のとき、拡張倍精度デノーマル実数および拡張倍精度擬似デノーマル実数を表す。
- 1 ビットの**符号**フィールドは、数値が正 (sign=0) か負 (sign=1) かを示す。並列 FP データでは、このビットは常に 0 である。

図 5-1. 浮動小数点レジスタ形式



指数フィールドが非ゼロである有限浮動小数点数の値は、次の式を使って計算される。

$$(-1)^{(sign)} * 2^{(exponent - 65535)} * (significand\{63\}.significand\{62:0\}_2)$$

指数フィールドがゼロである有限浮動小数点数の値は、次の式を使って計算される。

$$(-1)^{(sign)} * 2^{(-16382)} * (significand\{63\}.significand\{62:0\}_2)$$

整数 (符号付きまたは符号なし 64 ビット) および 並列 FP 数は、64 ビットの仮数フィールドに置かれる。標準の形式では、指数フィールドは 0x1003E (バイアスされた 63) に設定され、符号フィールドはゼロに設定される。

5.1.3 浮動小数点レジスタ値の表現

浮動小数点レジスタのエンコーディングはクラスおよびサブクラスにグループ分けされる。表 5-2 にそれを示す (灰色で表示されているエンコーディングはサポートされていない)。最後の 2 つの項には、固定浮動小数点レジスタ FR 0 および FR 1 の値が含まれている。FR 1 の定数は、並列単精度命令でも整数積累算命令でも変更されず、一般的には使用されない。

表 5-2. 浮動小数点レジスタのエンコーディング

クラスまたはサブクラス	符号 (1 ビット)	バイアスされた指数 (17 ビット)	仮数 i.bb...bb (明示の整数ビットを含む) (64 ビット)
NaNs	0/1	0x1FFFF	1.000...01 ～ 1.111...11
クワイエット型 NaN	0/1	0x1FFFF	1.100...00 ～ 1.111...11
クワイエット型 NaN 不定数[a]	1	0x1FFFF	1.100...00
シグナル型 NaN	0/1	0x1FFFF	1.000...01 ～ 1.011...11
無限大	0/1	0x1FFFF	1.000...00
擬似 NaN	0/1	0x1FFFF	0.000...01 ～ 0.111...11
擬似無限大	0/1	0x1FFFF	0.000...00
正規数 (浮動小数点レジスタ形式のノーマル数)	0/1	0x00001 ≀ 0x1FFFE	1.000...00 ～ 1.111...11
整数または 並列 FP (符号なしラージ整数または負の符号付き整数)	0	0x1003E	1.000...00 ～ 1.111...11
不定整数[b]	0	0x1003E	1.000...00
IEEE 単精度ノーマル実数	0/1	0x0FF81 ≀ 0x1007E	1.000...00...(40)0s ≀ 1.111...11...(40)0s
IEEE 倍精度ノーマル実数	0/1	0x0FC01 ≀ 0x103FE	1.000...00...(11)0s ≀ 1.111...11...(11)0s
IEEE 拡張倍精度ノーマル実数	0/1	0x0C001 ≀ 0x13FFE	1.000...00 ～ 1.111...11
拡張倍精度擬似デノーマル実数と同じ値をもつノーマル数	0/1	0x0C001	1.000...00 ～ 1.111...11
IA-32 スタック単精度ノーマル実数 (計算モデルが IA-32 スタック単精度のときに生成される)	0/1	0x0C001 ≀ 0x13FFE	1.000...00...(40)0s ≀ 1.111...11...(40)0s
IA-32 スタック倍精度ノーマル実数 (計算モデルが IA-32 スタック倍精度のときに生成される)	0/1	0x0C001 ≀ 0x13FFE	1.000...00...(11)0s ≀ 1.111...11...(11)0s
非正規数 (浮動小数点レジスタ形式の非正規数)	0/1	0x00000	0.000...01 ～ 1.111...11
		0x00001 ≀ 0x1FFFE	0.000...01 ～ 0.111...11
		0x00001 ≀ 0x1FFFD	0.000...00
	1	0x1FFFE	0.000...00

表 5-2. 浮動小数点レジスタのエンコーディング (続き)

クラスまたはサブクラス	符号 (1 ビット)	バイアスされた指数 (17 ビット)	仮数 i.bb...bb (明示の整数ビットを含む) (64 ビット)
整数または並列 FP (正の符号付きまたは符号なし整数)	0	0x1003E	0.000...00 ～ 0.111...11
単精度デノーマル実数	0/1	0x0FF81	0.000...01...(40)0s ≀ 0.111...11...(40)0s
倍精度デノーマル実数	0/1	0x0FC01	0.000...01...(11)0s ≀ 0.111...11...(11)0s
レジスタ形式のデノーマル数	0/1	0x00001	0.000...01 ～ 0.111...11
拡張倍精度デノーマル実数	0/1	0x00000	0.000...01 ～ 0.111...11
拡張倍精度デノーマル実数と同じ値を持つアンノーマル数	0/1	0x0C001	0.000...01 ～ 0.111...11
拡張倍精度擬似デノーマル実数 (IA-32 スタックおよびメモリ形式)	0/1	0x00000	1.000...00 ～ 1.111...11
IA-32 スタック単精度デノーマル実数 (計算モデルが IA-32 スタック単精度のときに生成される)	0/1	0x00000	0.000...01...(40)0s ≀ 0.111...11...(40)0s
IA-32 スタック倍精度デノーマル実数 (計算モデルが IA-32 スタック倍精度のときに生成される)	0/1	0x00000	0.000...01...(11)0s ≀ 0.111...11...(11)0s
擬似ゼロ	0/1	0x00001 ≀ 0x1FFFD	0.000...00
	1	0x1FFFE	0.000...00
NaTVal[c]	0	0x1FFFE	0.000...00
ゼロ	0/1	0x00000	0.000...00
FR 0 (正の 0)	0	0x00000	0.000...00
FR 1 (正の 1)	0	0x0FFFF	1.000...00

a. マスクされた実数無効操作のデフォルト応答
b. マスクされた整数無効操作のデフォルト応答
c. スペキュレーティブなメモリ操作が失敗したときに生成される。

すべてのレジスタ・ファイルのエンコーディングは、算術演算の入力として使用できる。算術演算の結果は常に、算出された値を最大限に正規化したレジスタ・ファイル表現であり、指数範囲は出力先の型の Emin から Emax の範囲に限定され、仮数の精度は出力先の型の精度ビット数に制限される。これらの境界を超える算出値、すなわちゼロ、無限大、NaN などの値は、対応する固有のレジスタ・ファイルのエンコーディングによって表現される。拡張倍精度デノーマル実数の結果は、レジスタ・ファイルの指数 0x00000 にマッ

プされる (0x0C001 ではない)。サポートされていないエンコーディング (擬似 NaN、擬似無限大)、擬似ゼロ、および拡張倍精度擬似デノーマル実数が算術演算の結果として生成されることはない。

擬似ゼロに対する算術演算は、同じ符号の符号付きゼロと同じように処理される。ただし、擬似ゼロに無限大を掛ける場合は例外で、同じ符号の無限大が返され、Invalid Operation Floating-point Exception (無効操作浮動小数点例外) フォルト (および QNaN) は返されない。また擬似ゼロはゼロではなく、非正規数として分類される。

5.2 浮動小数点ステータス・レジスタ

浮動小数点ステータス・レジスタ (FPSR) には、浮動小数点演算のための動的なコントロール情報とステータス情報が含まれる。メイン・セットであるコントロール / ステータス情報 (FPSR.sf0) と 3 つの代替セット (FPSR.sf1、FPSR.sf2、FPSRsf3) がある。図 5-2 に FPSR のレイアウトを、表 5-3 に各フィールドの定義を示す。表 5-4 に FPSR のステータス・フィールドの説明を、図 5-3 にそのレイアウトを示す。

図 5-2. 浮動小数点ステータス・レジスタの形式

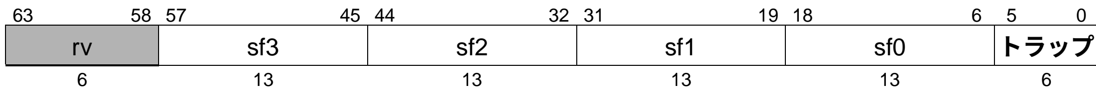


表 5-3. 浮動小数点ステータス・レジスタのフィールドの説明

フィールド	ビット	説明
traps.vd	0	4 このビットがセットされているときは、Invalid Operation Floating-point Exception (無効操作浮動小数点例外) フォルト (IEEE トラップ) がディセーブルである
traps.dd	1	このビットがセットされているときは、Denormal/Unnormal Operand Floating-point Exception (デノーマル / アンノーマル・オペランド浮動小数点例外) フォルトがディセーブルである
traps.zd	2	このビットがセットされているときは、Zero Divide Floating-point Exception (ゼロ除算浮動小数点例外) フォルト (IEEE トラップ) がディセーブルである
traps.od	3	このビットがセットされているときは、Overflow Floating-point Exception (オーバーフロー浮動小数点例外) トラップ (IEEE トラップ) がディセーブルである
traps.ud	4	このビットがセットされているときは、Underflow Floating-point Exception (アンダーフロー浮動小数点例外) トラップ (IEEE トラップ) がディセーブルである
traps.id	5	このビットがセットされているときは、Inexact Floating-point Exception (不正確浮動小数点例外) トラップ (IEEE トラップ) がディセーブルである

表 5-3. 浮動小数点ステータス・レジスタのフィールドの説明 (続き)

フィールド	ビット	説明
sf0	18:6	メイン・ステータス・フィールド
sf1	31:19	代替ステータス・フィールド 1
sf2	44:32	代替ステータス・フィールド 2
sf3	57:45	代替ステータス・フィールド 3
rv	63:58	予約

図 5-3. 浮動小数点ステータス・フィールドの形式

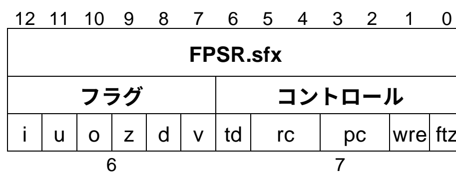


表 5-4. 浮動小数点ステータス・レジスタのステータス・フィールドの説明

フィールド	ビット	説明
ftz	0	ゼロ・フラッシュ・モード
wre	1	最大範囲の指数 (表 5-6 参照)
pc	3:2	精度制御 (表 5-6 参照)
rc	5:4	丸め制御 (表 5-5 参照)
td	6	トラップ・ディセーブル[a]
v	7	無効操作 (IEEE フラグ)
d	8	デノーマル / アンノーマル・オペランド
z	9	ゼロ除算 (IEEE フラグ)
o	10	オーバーフロー (IEEE フラグ)
u	11	アンダーフロー (IEEE フラグ)
i	12	不正確 (IEEE フラグ)

a. td はメイン・ステータス・フィールド FPSR.sf0 の予約ビットである。

デノーマル / アンノーマル・オペランド・ステータス・フラグは IEEE 形式のスティッキー・フラグで、その値が算術命令および算術演算で使用される場合にセットされる。たとえば、アンノーマル (*) NaN では、d フラグはセットされない。標準の単精度 / 倍精度 / 拡張倍精度デノーマル / 拡張倍精度擬似デノーマル / レジスタ形式デノーマルのエンコーディングは、浮動小数点レジスタ形式の非正規数のサブセットである。

注: 浮動小数点例外フォルト / トラップは、命令の実行中に、イネーブルになっている例外が生じた場合にだけ発生する。したがって、ソフトウェアでステータス・フィールドのフラグ・ビットを 1 にセットしても、割り込みは発生しない。ステータス・フィールドの各フラグは、単に、浮動小数点例外の発生を示すだけである。

ゼロ・フラッシュ (FTZ) モードでは、" 極小値 " を含む結果が、対応する符号が付いた符号付きゼロに切り捨てられる。ゼロ・フラッシュ・モードは、アンダーフローがディセーブルにされている場合にだけイネーブルにできる。アンダーフローをディセーブルにするには、すべてのトラップをディセーブルにする (FPSR.sfx.td を 1 にセットする) か、個別にディセーブルにする (FPSR.traps.ud を 1 にセットする)。アンダーフローがイネーブルにされている場合は、それが優先され、ゼロ・フラッシュは無視される。ただし、ソフトウェア例外ハンドラを使ってゼロ・フラッシュ・モード・ビットを調べ、イネーブルにされているアンダーフロー例外が発生したときにゼロ・フラッシュ演算をエミュレートすることができる。

ゼロ・フラッシュ・モードによって、得られた結果が対応する符号が付いた符号付きゼロにフラッシュされると、FPSR.sfx.u ビットおよび FPSR.sfx.i ビットが 1 にセットされる。不正確結果例外がイネーブルにされている場合は、この例外が通知される。

浮動小数点の結果は、命令の *.pc* コンプリータおよびステータス・フィールドの *wre*、*pc*、および *rc* コントロール・フィールドに基づいて丸められる。得られる結果の仮数の精度および指数範囲は、表 5-6 のように決定される。結果が正確でない場合、FPSR.sfx.*rc* によって丸め方向を指定する (表 5-5 を参照)。

表 5-5. 浮動小数点の丸め制御の定義

	最近値方向 (偶数)	マイナス無限大方向 (切り下げ)	プラス無限大方向 (切り上げ)	ゼロ方向 (切り捨て / チョップ)
FPSR.sfx.rc	00	01	10	11

表 5-6. 浮動小数点演算モデルの制御の定義

計算モデルのコントロール・フィールド			選択した計算モデル		
命令の *.pc* コンプリータ	FPSR.sfx の動的 *pc* フィールド	FPSR.sfx の動的 *wre* フィールド	仮数の精度	指数範囲	演算形式
.s	無視	0	24 ビット	8 ビット	IEEE 実数単精度
.d	無視	0	53 ビット	11 ビット	IEEE 実数倍精度
.s	無視	1	24 ビット	17 ビット	レジスタ・ファイルの範囲、単精度
.d	無視	1	53 ビット	17 ビット	レジスタ・ファイルの範囲、倍精度
なし	00	0	24 ビット	15 ビット	IA-32 スタック単精度
なし	01	0	未定義	未定義	予約
なし	10	0	53 ビット	15 ビット	IA-32 スタック倍精度
なし	11	0	64 ビット	15 ビット	IA-32 拡張倍精度
なし	00	1	24 ビット	17 ビット	レジスタ・ファイルの範囲、単精度
なし	01	1	未定義	未定義	予約
なし	10	1	53 ビット	17 ビット	レジスタ・ファイルの範囲、倍精度
なし	11	1	64 ビット	17 ビット	レジスタ・ファイルの範囲、拡張倍精度
適応せず[a]	無視	無視	24 ビット	8 ビット	IEEE 実数単精度のペア
適応せず[b]	無視	無視	64 ビット	17 ビット	レジスタ・ファイルの範囲、拡張倍精度

a. .pc コンプリータを使用しない並列 FP 命令 (例、fpma)
b. .pc コンプリータを使用しない非並列 FP 命令 (例、fmerge)

トラップ・ディセーブル (*sfx.td*) 制御ビットを使って、IEEE 例外トラップについてローカルなデフォルト環境を簡単に設定できる。FPSR.*sfx*.td がクリア (イネーブル) されている場合は、FPSR.trap ビットが使用される。FPSR.sfx.td がセットされている場合には、FPSR.trap ビットがすべてセット (ディセーブル) されているように扱われる。FPSR.sf0.td は予約フィールドであり、読み込まれるとゼロを返す。

5.3 浮動小数点命令

この節では、IA-64 浮動小数点命令について説明する。

5.3.1 メモリ・アクセス命令

単精度、倍精度、拡張倍精度の浮動小数点実数型データ、および並列 FP または符号付き / 符号なし整数型データについての浮動小数点ロード命令およびストア命令がある。浮動小数点ロード命令およびストア命令のアドレス指定モードは、整数のロード命令およびストア命令と同様であるが、浮動小数点ペア・ロード命令ではベース・レジスタの暗黙のポスト・インクリメントを利用できる。浮動小数点のロード命令およびストア命令のメモリ・ヒント・オプションは、整数のロード命令およびストア命令と同じである (4-26 ページの「メモリ階層の制御と整合性」を参照)。表 5-7 に、浮動小数点のロード命令およびストア命令の型を示す。

浮動小数点ペア・ロード命令は、2 つのターゲット・レジスタが奇数番と偶数番または偶数番と奇数番でなければならない。浮動小数点ストア命令 (stfs、stfd、stfe) は、形式変換が正しく行われるためには、浮動小数点レジスタの値がストア命令と同じ型でなければならない。

表 5-7. 浮動小数点メモリ・アクセス命令

操作	FR へのロード	FR へのペアのロード	FR からのストア
単精度	ldfs	ldfps	stfs
整数 / 並列 FP	ldf8	ldfp8	stf8
倍精度	ldfd	ldfpd	stfd
拡張倍精度	ldfe		stfe
スピル / フィル	ldf.fill		stf.spill

スペキュレーティブ・ロードが失敗した場合は、NaTVal が出力先レジスタに書き込まれる (4.4.4 項を参照)。スピル命令 (stf.spill) 以外の命令では、NaTVal をメモリにストアすると Register NaT Consumption (レジスタ NaT 参照) フォルトが発生する。

浮動小数点レジスタの保存および復元は、スピル命令およびフィル命令 (stf.spill、ldf.fill) によって 16 バイト・メモリ・コンテナを使って行われる。これらの命令だけが実際のレジスタの内容を保存および復元できる (これらの命令は NaTVal でフォルトしない)。これらの命令はすべての型の値 (単精度、倍精度、拡張倍精度、レジスタ形式、および整数または 並列 FP) を保存および復元できるので、将来のアーキテクチャの拡張に対応できる。

図 5-4 ～ 5-7 に、浮動小数点レジスタとメモリの間のデータ転送時に、単精度、倍精度、拡張倍精度、およびスピルやフィル時のデータがどのように変換されるかを示す。

図 5-4. メモリから浮動小数点レジスタへのデータ変換 - 単精度

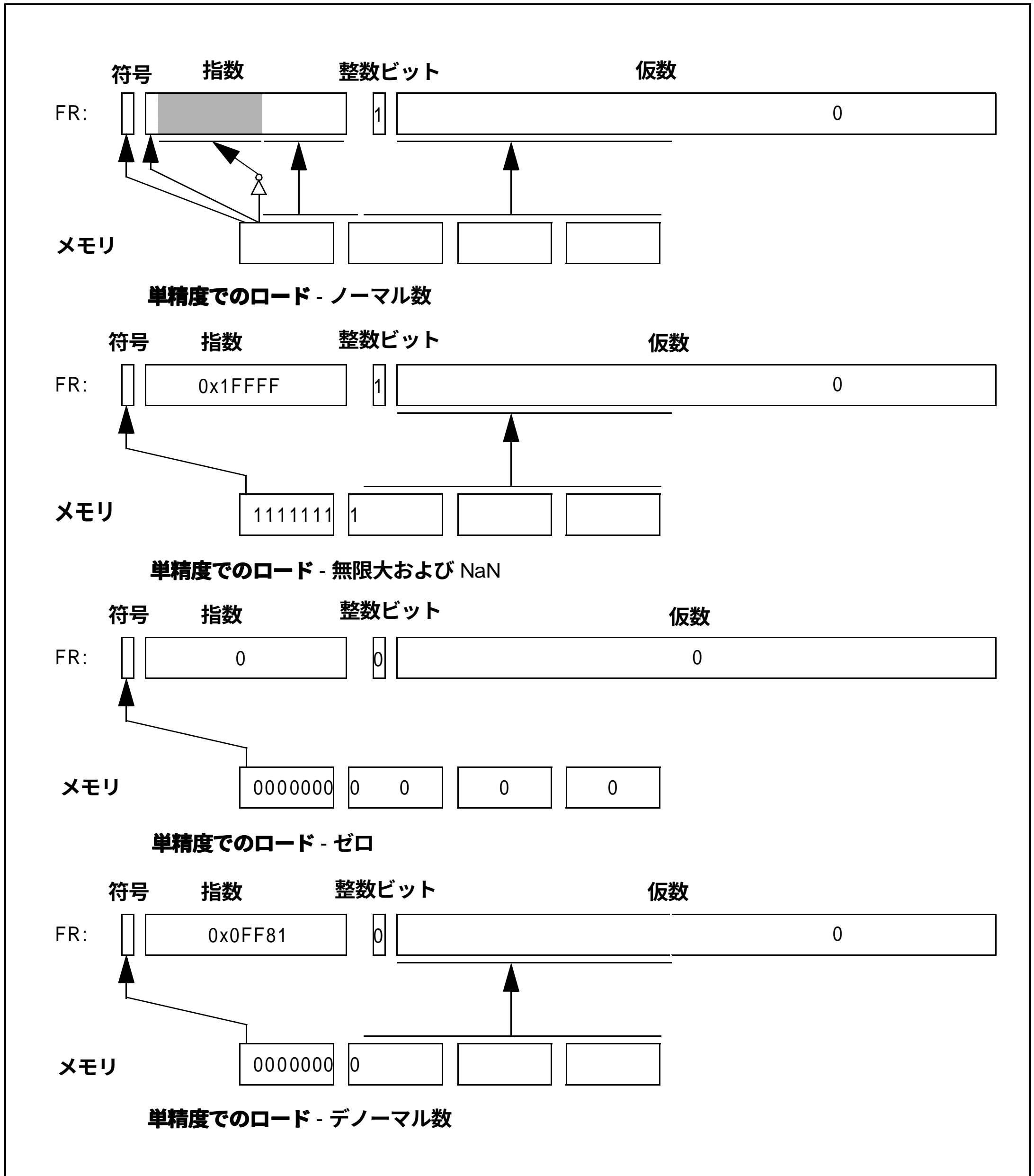


図 5-5. メモリから浮動小数点レジスタへのデータ変換 - 倍精度

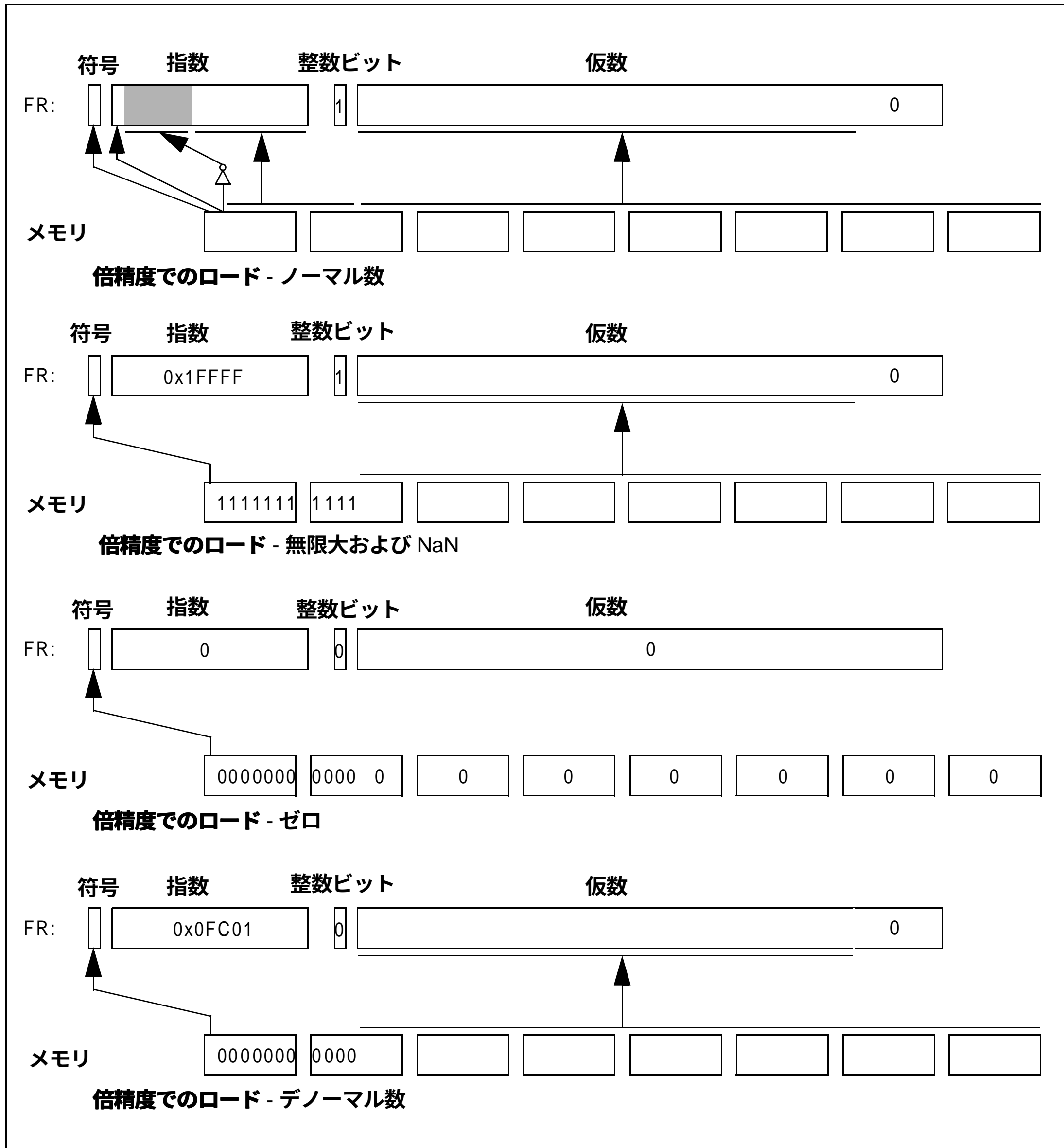


図 5-6.　メモリから浮動小数点レジスタへのデータ変換 - 拡張倍精度、整数、およびフィル

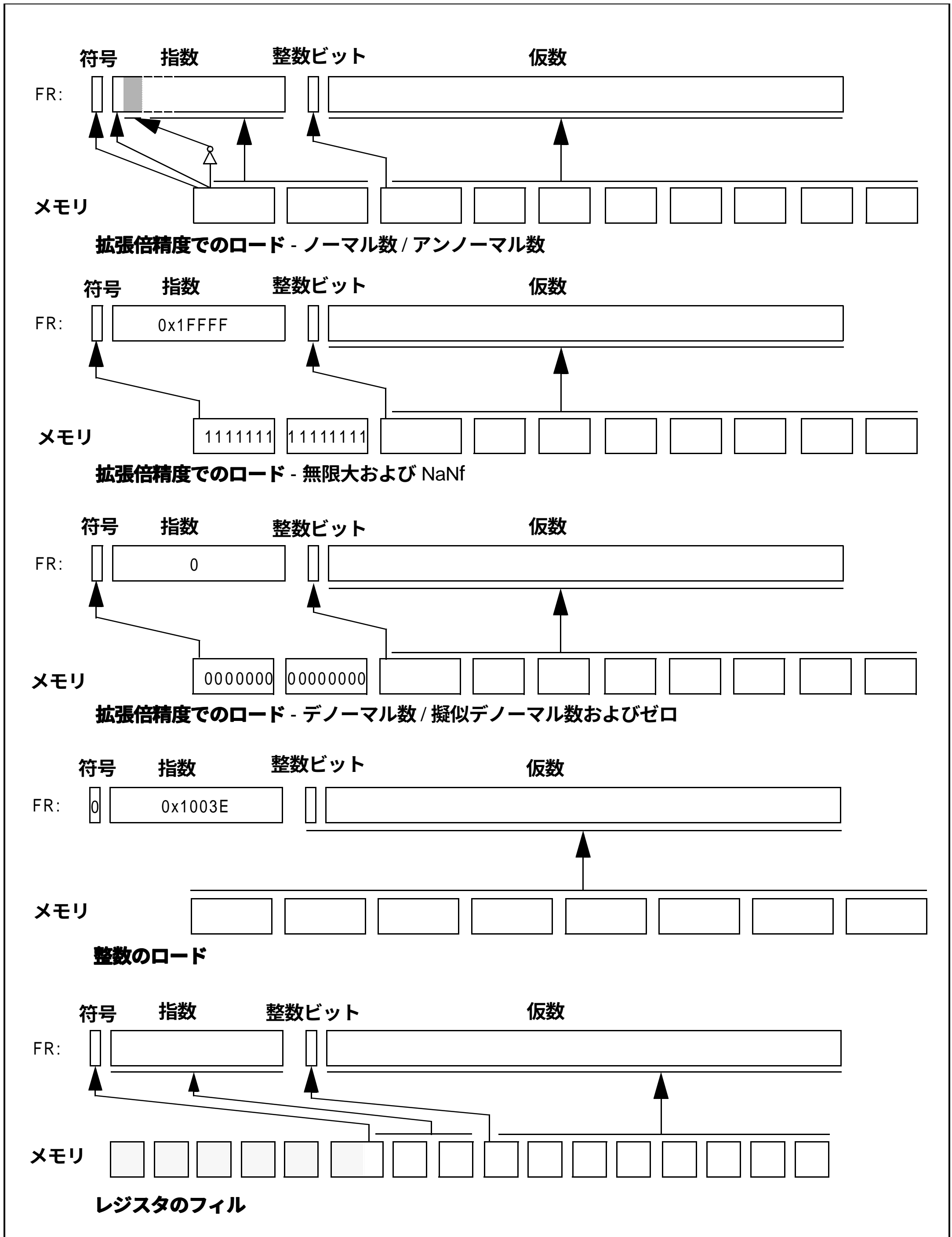
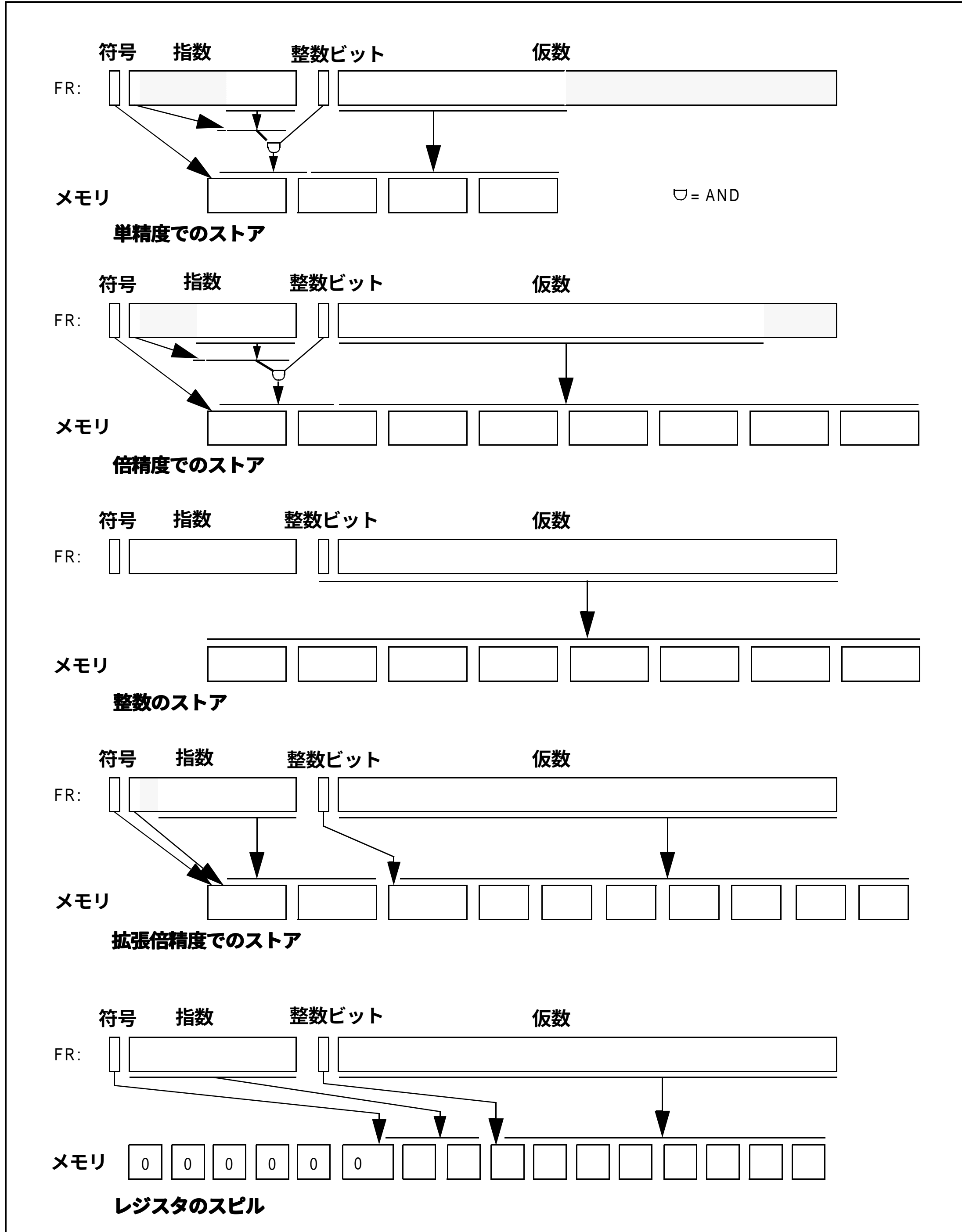
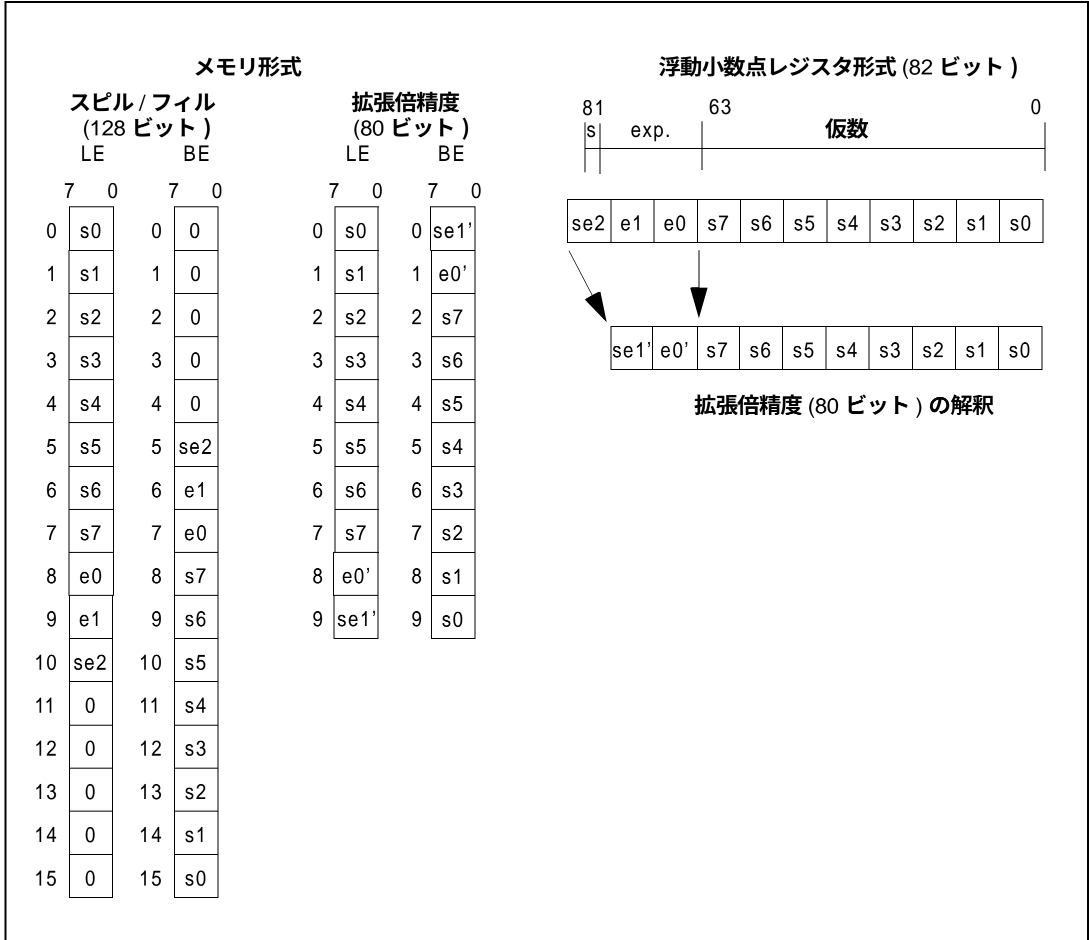


図 5-7. 浮動小数点レジスタからメモリへのデータ変換



浮動小数点のロードおよびストアでは、リトル・エンディアン型およびビッグ・エンディアン型の両方のバイト・オーダーがサポートされている。単精度および倍精度のメモリ形式では、バイト順序は 32 ビットおよび 64 ビットの整数データ型と同じである (3.2.3 項を参照)。スピル / フィル時のメモリ形式および拡張倍精度のメモリ形式のバイト順序を、図 5-8 に示す。

図 5-8. スピル / フィルおよび拡張倍精度 (80 ビット) での浮動小数点メモリ形式



5.3.2 浮動小数点レジスタと汎用レジスタとの間の転送命令

`setf` 命令および `getf` 命令 (表 5-8 を参照) は、浮動小数点レジスタ (FR) と汎用レジスタ (GR) の間でデータを転送する。これらの命令は、汎用レジスタの NaT を浮動小数点レジスタ NaTVal に変換したり、NaTVal から NaT に変換する。それ以外の命令の場合は、図 5-4、5-5、5-7 に示すように `setf` と `getf` 命令の変形である .s および .d は FR に変換されたり、FR から他のものに変換される。メモリ表現は、GR から読み込まれたり GR に書き込まれたりする。

`setf` および `getf` 命令の変形である `.exp` および `.sig` はそれぞれ、浮動小数点レジスタの符号 / 指数部分および仮数部分を処理する。それらの変換形式を表 5-9 および 表 5-10 に示す。

表 5-8. 浮動小数点レジスタ転送命令

操作	GR から FR へ	FR から GR へ
Single	`setf.s`	`getf.s`
Double	`setf.d`	`getf.d`
Sign and Exponent	`setf.exp`	`getf.exp`
Significand/Integer	`setf.sig`	`getf.sig`

表 5-9. 汎用レジスタ (整数) から浮動小数点レジスタへのデータ変換

クラス	汎用レジスタ		浮動小数点レジスタ (.sig)			浮動小数点レジスタ (.exp)		
	NaT	整数	符号	指数	仮数	符号	指数	仮数
NaT	1	無視	NaTVal			NaTVal		
整数	0	000...00 ~ 111...11	0	0x1003E	整数	整数 {17}	整数 {16:0}	0x8000000000000000

表 5-10. 浮動小数点レジスタから汎用レジスタ (整数) へのデータ変換

クラス	浮動小数点レジスタ			汎用レジスタ (.sig)		汎用レジスタ (.exp)	
	符号	指数	仮数	NaT	整数	NaT	整数
NaTVal	0	0x1FFFE	0.000...00	1	0x0000000000000000	1	0x1FFFE
整数または並列 FP	0	0x1003E	0.000...00 ~ 1.111...11	0	仮数	0	0x1003E
その他	無視	無視	無視	0	仮数	0	((sign<<17) | 指数)

5.3.3 算術命令

すべての算術浮動小数点命令には *.sf* 指定子がある (ただし、`fcvt.xf` 命令は除く)。この指定子によって、4 つの FPSR のステータス・フィールドのどれで命令実行のステータスを制御および記録するかを指定する (表 5-11 参照)。ステータス・フィールドには、イネーブルにされている例外、丸めモード、指数のビット数、精度制御、および更新するステータス・フィールドのフラグを指定する。5-6 ページの「浮動小数点ステータス・レジスタ」を参照のこと。

表 5-11. 浮動小数点命令のステータス・フィールドの指定子の定義

.sf 指定子	.s0	.s1	.s2	.s3
ステータス・フィールド	FPSR.sf0	FPSR.sf1	FPSR.sf2	FPSR.sf3

ほとんどの算術浮動小数点命令は、結果の精度を *.pc* コンプリータを使って静的に、あるいは FPSR ステータス・フィールドの *.pc* フィールドを使って動的に指定することができる (表 5-6 を参照)。*.pc* コンプリータが指定されていない算術命令では、浮動小数点レジスタ・ファイルに指定されている範囲および精度を使用する。

表 5-12 に浮動小数点算術命令を、表 5-13 に擬似演算の定義を示す。

表 5-12. 浮動小数点算術命令

操作	通常の FP ニーモニック	並列 FP ニーモニック
浮動小数点積和	fma.*pc*.*sf*	fpma.*sf*
浮動小数点積差	fms.*pc*.*sf*	fpms.*sf*
浮動小数点積和否定	fnma.*pc*.*sf*	fpnma.*sf*
浮動小数点逆数近似	frcpa.*sf*	fprcpa.*sf*
浮動小数点平方根逆数近似	frsqrta.*sf*	fprsqrta.*sf*
浮動小数点比較	fcmp.*frel*.*fctype*.*sf*	fpcmp.*frel*.*sf*
浮動小数点最小値	fmin.*sf*	fpmin.*sf*
浮動小数点最大値	fmax.*sf*	fpmax.*sf*
浮動小数点絶対最小値	famin.*sf*	fpamin.*sf*
浮動小数点絶対最大値	famax.*sf*	fpamax.*sf*
浮動小数点から符号付き整数への変換	fcvt.fx.*sf* fcvt.fx.trunc.*sf*	fpcvt.fx.*sf* fpcvt.fx.trunc.*sf*
浮動小数点から符号なし整数への変換	fcvt.fxu.*sf* fcvt.fxu.trunc.*sf*	fpcvt.fxu.*sf* fpcvt.fxu.trunc.*sf*
符号付き整数から浮動小数点への変換	fcvt.xf	N.A.

表 5-13. 浮動小数点擬似演算

操作	ニーモニック	使用する演算
浮動小数点乗算 (IEEE) 並列 FP 乗算	fmpy.*pc*.*sf* fpmpy.*sf*	fma、FR 0 を加数として使用する fpma、FR 0 を加数として使用する
浮動小数点乗算否定 (IEEE) 並列 FP 乗算否定	fnmpy.*pc*.*sf* fpnmpy.*sf*	fnma、FR 0 を加数として使用する fpnma、FR 0 を加数として使用する
浮動小数点加算 (IEEE)	fadd.*pc*.*sf*	fma、FR 1 を被乗数として使用する
浮動小数点減算 (IEEE)	fsub.*pc*.*sf*	fms、FR 1 を被乗数として使用する
浮動小数点否定	fnma.*pc*.*sf*	fnma、FR 1 を被乗数として使用し、FR 0 を加数として使用する
浮動小数点絶対値 並列 FP 絶対値	fabs fpabs	fmerge.s、FR 0 の符号を使用する fpmerge.s、FR 0 の符号を使用する
浮動小数点否定 並列 FP 否定	fneg fpneg	fmerge.ns fpmerge.ns
浮動小数点絶対値否定 並列 FP 絶対値否定	fnegabs fpnegabs	fmerge.ns、FR 0 の符号を使用する fpmerge.ns、FR 0 の符号を使用する
浮動小数点正規化	fnorm.*pc*.*sf*	fma、FR1 を被乗数として使用し、FR0 を加数として使用する
符号なし整数から浮動小数点への変換	fcvt.xuf.*pc*.*sf*	fma、FR1 を被乗数として使用し、FR0 を加数として使用する

FR 1 にはパックされた単精度の 1.0 値のペアが格納されていないので、並列 FP の加算、減算、否定、および正規化の擬似演算はない。並列 FP の加算を実行するには、はじめにレジスタに 1.0 値のペアを生成し (fpack 命令を使用する)、次に fpma 命令を使用する。同様に、整数加算演算を実行するには、はじめに浮動小数点レジスタに整数 1 を生成し、次に xma 命令を使用する。

5.3.4 非算術命令

表 5-14 に非算術浮動小数点命令を示す。fclass 命令は、浮動小数点レジスタの内容を分類するために使用する。fmerge 命令は、2 つの浮動小数点レジスタのデータを 1 つの浮動小数点レジスタにマージするために使用する。fmix、fsxt、fpack、および fswap 命令は、浮動小数点仮数の 並列 FP

データを操作するために使用する。fand、fandcm、for、および fxor 命令は、浮動小数点仮数の論理演算を実行するために使用する。fselect 命令は条件付き選択のために使用する。

非算術浮動小数点命令には *.pc* コンプリータおよび *.sf* 指定子がないので、常に浮動小数点レジスタ (82 ビット) の精度を使用する。

表 5-14. 非算術浮動小数点命令

操作	ニーモニック
浮動小数点分類	fclass.*fcrel*.*fctype*
浮動小数点符号マージ 並列 FP 符号マージ	fmerge.s fpmerge.s
浮動小数点符号否定マージ 並列 FP 符号否定マージ	fmerge.ns fpmerge.ns
浮動小数点符号および指数マージ 並列 FP 符号および指数マージ	fmerge.se fpmerge.se
浮動小数点左ミックス	fmix.l
浮動小数点右ミックス	fmix.r
浮動小数点左右ミックス	fmix.lr
浮動小数点左符号拡張	fsxt.l
浮動小数点右符号拡張	fsxt.r
浮動小数点パック	fpack
浮動小数点スワップ	fswap
浮動小数点のスワップおよび左否定	fswap.nl
浮動小数点のスワップおよび右否定	fswap.nr
浮動小数点論理積	fand
浮動小数点補数の論理積	fandcm
浮動小数点論理和	for
浮動小数点排他的論理和	fxor
浮動小数点選択	fselect

5.3.5 浮動小数点ステータス・レジスタ (FPSR) のステータス・フィールド命令

浮動小数点演算のスペキュレーションでは、ステータス・フラグが一時的に 1 つの代替ステータス・フィールド (FPSR.sf0 以外のステータス・フィールド) にストアされなければならない。スペキュレーティブ実行チェーンがコミットされた後で `fchkf` 命令を使って通常のフラグ (FPSR.sf0.flag) を更新できる。この演算によって、IEEE フラグが常に正しく保たれる。`fchkf` 命令では、そのためにステータス・フィールドの各フラグを FPSR.sf0.flag および FPSR.trap と比較する。代替ステータス・フィールドのフラグが FPSR.trap の中のイネーブルにされている浮動小数点例外に対応するイベントや、FPSR.sf0.flag の中のまだ登録されていないイベント (つまり、FPSR.sf0.flag の中のそのイベントのフラグがクリアされている) が発生したことを示している場合は、`fchkf` 命令によって Speculative Operation (スペキュレーティブ操作) フォルトが発生する。このどちらも起こらない場合には、`fchkf` 命令は何も行わない。

`fsetc` 命令では、ステータス・フィールドのコントロール・ビットをビットごとに変更できる。FPSR.sf0.control が 7 ビットの即値の論理積マスクとの間で論理積が取られ、さらに 7 ビット の即値の論理和マスクとの間で論理和が取られてステータス・フィールドのコントロール・ビットが生成される。`fclrf` 命令は、すべてのステータス・フィールドのフラグを 0 にクリアする。

表 5-15. FPSR ステータス・フィールド命令

操作	ニーモニック
浮動小数点フラグ・チェック	`fchkf.`*sf*
浮動小数点フラグ・クリア	`fclrf.`*sf*
浮動小数点コントロール・ビット設定	`fsetc.`*sf*

5.3.6 整数積和命令

整数 (固定小数点) 乗算は、浮動小数点ユニットで、3 つのオペランドを使用する xma 命令によって実行される。これらの命令のオペランドおよび結果は浮動小数点レジスタである。xma 命令は、NaTVal チェックを除いて、浮動小数点レジスタの符号フィールドおよび指数フィールドを無視する。2 つの 64 ビット・ソース・オペランドの仮数の積が 3 番目の 64 ビット・ソース・オペランドの仮数 (ゼロ拡張されている) に加算されて 128 ビットの結果が生成される。この命令には下位バージョンと上位バージョンがあり、それぞれ 128 ビット結果の下位または上位 64 ビットを選択し、それを標準の整数として出力先レジスタに書き込む。この命令の符号付きバージョンおよび符号なしバージョンではそれぞれ、入力レジスタを符号付きおよび符号なしの 64 ビット整数として扱う。

表 5-16. 整数積和命令

整数積和	下位	上位
符号付き	xma.l	xma.h
符号なし	xma.lu （擬似オペコード）	xma.hu

5.4 IEEE についての補足事項

5.4.1 SNaN、QNaN および NaN の伝播の定義

シグナル型 NaN は、仮数の最上位の小数ビットが 0 である。クワイエット型 NaN は、仮数の最上位の小数ビットが 1 である。シグナル型 NaN と クワイエット型 NaN がこのように定義されていることによって、異なる精度の間で変換する時でも "NaN" としての性質を容易に維持できる。2 つ以上の NaN オペランドがある演算で NaN を伝播する場合に、得られた結果の NaN がどちらのオペランドを選択するかについては、レジスタ・エンコーディング・フィールドに基づく優先順位 (f4、f2、f3) で決まる。

5.4.2 ソフトウェアに委ねられる IEEE 標準演算

以下の IEEE 演算がソフトウェアによってサポートされる。

- 文字列から浮動小数点への変換
- 浮動小数点から文字列への変換
- 除算 (`frcpa` または `fprcpa` 命令を利用する)
- 平方根 (`frsqrta` または `fprsqrta` 命令を利用する)
- 剰余 (`frcpa` または `fprcpa` 命令を利用する)
- 浮動小数点から整数値浮動小数点への変換
- 単精度、倍精度、および拡張倍精度のオーバーフローおよびアンダーフロー値での適切な指数ラッピング (IEEE 標準での推奨に従う)。

5.4.3 IEEE 標準にない演算

- 乗算・加算の混合演算 (`fma`、`fms`、`fnma`、`fpma`、`fpms`、`fpnma`) によって、効率良い除算、平方根、および剰余についてのソフトウェア・アルゴリズムが可能である。
- レジスタ・ファイル形式での 17 ビットの拡張指数範囲によって、多くの基本的な数値アルゴリズムの設計が容易になる。
- NaTVal は、IEEE の NaN の概念を自然に拡張したものである。これにより、スペキュレーティブな実行がサポートされる。

- ゼロ・フラッシュ・モードは、業界標準の付加演算である。
- 最小値および最大値命令によって、よく使われる Fortran 固有関数の MIN()、MAX()、AMIN()、AMAX() や、a<b?a:b などの C 言語イディオムを効率よく実行できる。
- すべての混合精度オペランドが可能である。IEEE 標準では、低い精度のオペランドから高い精度の結果を生成可能であることが提唱されており、これがサポートされている。また IEEE 標準では、高い精度のオペランドから低い精度の結果を生成することはできないことが提唱されているが、これについては従っていない。
- IEEE スタイルの 4 倍精度実数型がソフトウェアでサポートされる。

intel®

IA-64 システム環境での IA-32 アプリケーション実行モデル 6

IA-64 アーキテクチャでは、システムに必要なプラットフォームおよびファームウェアがサポートされている場合には、従来の IA-32 オペレーティング・システム上で IA-32 アプリケーション・バイナリを変更なしに実行できる。

この章では、IA-64 システム環境での IA-32 命令の実行について説明する。IA-64 アーキテクチャでは、IA-64 オペレーティング・システム上で実行する 16 ビット・リアル・モード、16 ビット VM86、および 16 ビット /32 ビット保護モードでの IA-32 アプリケーションをサポートしている。これらの機能に対する IA-64 オペレーティング・システムによるサポートは、オペレーティング・システム・ベンダによって定義される。

本章で取り上げる主な機能は以下の通りである。

- IA-32 および IA-64 の命令セットの移行
- IA-32 の整数、セグメント、浮動小数点、MMX® テクノロジ、およびストリーミング SIMD 拡張命令の各レジスタの状態マッピング
- IA-32 のメモリおよびアドレス指定モデルの概要

この章では IA-32 アプリケーション・プログラミング・モデル、IA-32 の命令およびレジスタについての詳細には触れていない。IA-32 アプリケーション・プログラミング・モデルの詳細については、『*インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル*』を参照のこと。

6.1 命令セット・モード

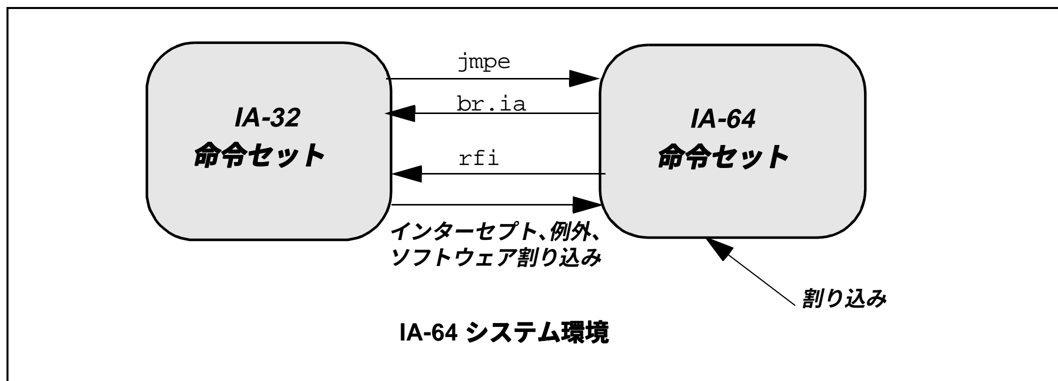
プロセッサは、IA-32 命令または IA-64 命令を実行できる。プロセッサ・ステータス・レジスタ (PSR) のビットで、現在実行中の命令セットを指定する。プロセッサの IA-32 命令セットと IA-64 命令セットの間の移行 (図 6-1 に示す) のために、3 つの特別な命令と割り込みが定義されている。

- `jmpe` (IA-32 命令)。IA-64 のターゲット命令にジャンプし、命令セットを IA-64 に変更する。
- `br.ia` (IA-64 命令)。IA-64 から IA-32 のターゲット命令に分岐し、命令セットを IA-32 に変更する。

- rfi (IA-64 命令)。rfi (割り込みからのリターン) は、割り込みから実行中の操作を再開するときに、IA-32 命令または IA-64 命令にリターンするように定義される。
- 割り込みは、すべての割り込み条件に対してプロセッサを IA-64 命令セットに切り換える。

jmpe 命令および br.ia 命令は、小さなオーバヘッドで、命令セット間の制御の受け渡しを行う。通常、これらのプリミティブは "thunks" または "stubs" に組み込まれている。"thunks" と "stubs" は、動的または静的にリンクされたライブラリをコールするために必要なコール・リンケージおよび呼び出し規則をサポートしている。

図 6-1. IA-64 プロセッサ命令セットの移行モデル



6.1.1 IA-64 命令セットの実行

プロセッサが IA-64 命令セットで実行している場合、

- IA-64 命令が、プロセッサによってフェッチ、デコードおよび実行される。
- IA-64 命令で、IA-64 および IA-32 のアプリケーション・レジスタのすべての状態にアクセスできる。これには IA-32 のセグメント・ディスクリプタ、セレクタ、汎用レジスタ、浮動小数点レジスタ、MMX テクノロジ・レジスタ、およびストリーミング SIMD 拡張命令レジスタが含まれる。レジスタ状態のマッピングについては 6.2 項を参照のこと。
- セグメンテーションは使用不能となる。セグメンテーション保護チェックは適用されず、セグメント・ベースの加算による仮想アドレスの計算も行われない。つまり、すべての算出アドレスは仮想アドレスである。
- 2^{64} 仮想アドレスを生成することができ、また IA-64 メモリ管理を使ってすべてのメモリ参照および I/O 参照を行う。

6.1.2 IA-32 命令セットの実行

プロセッサが、IA-64 システム環境で IA-32 命令を実行している場合は、Pentium® III プロセッサによって定義されている IA-32 アプリケーション・アーキテクチャが使用される。つまり、下記のようになる。

- IA-32 の 16/32 ビットのアプリケーション・レベル、MMX テクノロジ命令、およびストリーミング SIMD 拡張命令がプロセッサによってフェッチ、デコードおよび実行される。命令は 32/16 ビット・オペレーションに限定される。
- IA-32 アプリケーション・レベルのレジスタ状態だけが参照可能である (これには IA-32 汎用レジスタ、MMX テクノロジ・レジスタ、およびストリーミング SIMD 拡張命令レジスタ、セレクタ、EFLAGS、FP レジスタ、および FP コントロール・レジスタが含まれる)。IA-64 のアプリケーション状態および制御状態 (例、分岐、プレディケート、アプリケーションなど) は参照できない。
- IA-32 のリアル・モード、VM86 および保護モードのセグメンテーションが有効である。セグメント保護チェックが適用され、IA-32 セグメント規則に従って仮想アドレスが生成される。IA-32 セグメント・アプリケーションをサポートするために GDT および LDT セグメントが定義されている。セグメンテーションされた 16 および 32 ビット・コードが完全にサポートされている。
- すべての IA-32 命令セットおよび I/O ポート参照のための仮想アドレスから物理的アドレスへの変換は IA-64 メモリ管理を使って行われる。
- 命令およびデータのメモリ参照は、強制的にリトル・エンディアン型で行われる。メモリの順序付けは、Pentium III プロセッサのメモリ順序付けモデルを使用する。
- IA-32 オペレーティング・システム・リソース (IA-32 ページング、MTRR、IDT、コントロール・レジスタ、デバック・レジスタ、および特権命令) は、IA-64 用に定義されているリソースによって置き換えられる。これらのリソースにアクセスすると、インターセプション・フォルトになる。

6.1.3 命令セットの移行

以下の項では、各命令セットの移行時の動作の概要を示す。詳細については、`jmpe` (IA-32 命令) および `br.ia` (IA-64 命令) に関する詳細な説明を参照のこと。

6.1.3.1 JMPE 命令

`jmpe reg16/32`；`jmpe disp16/32` は、IA-64 命令セットへのジャンプと制御の引き渡しのために使用する。レジスタ間接形式と絶対形式の 2 種類がある。絶対形式では、仮想 IA-64 ターゲット・アドレスを以下のように計算する。

```
IP{31:0} =disp16/32 + CSD.base
IP{63:32} = 0
```

間接形式では、16/32 ビット・レジスタ位置を読み取り、次に IA-64 ターゲット・アドレスを以下のように計算する。

```
IP{31:0} = [reg16/32] + CSD.base
IP{63:32} = 0
```

IA-64 jmpe のターゲットは強制的に 16 バイトにアライメントされ、IA-32 でのアドレス指定の制約のため、64 ビットの仮想アドレス空間の下位 4G バイトに制約される。未処理の IA-32 数値例外がある場合は、jmpe は無効になり、IA-32 浮動小数点例外フォルトが発生する。

6.1.3.2 IA 命令への分岐

IA-32 命令セットへの無条件分岐には、IA-64 用に定義されている間接分岐メカニズムを使用する。IA-32 ターゲットは 32 ビットの仮想アドレス・ターゲット (実効アドレスではない) によって指定される。IA-32 仮想アドレスは、32 ビットに切り捨てられる。br.ia 命令の分岐ヒントは常に、静的に処理すると予測されるように設定しなければならない。IA-32 命令セットへのプロセッサの移行は以下のように設定される。

```
IP{31:0} = BR[b]{31:0}
IP{63:32} = 0
EIP{31:0} = IP{31:0} - CSD.base
```

IA-32 命令セットに移行してもプロセッサの特権レベルは変更されない。

分岐を発行する前に、コード・セグメントのディスクリプタおよびセレクタが適切にロードされていることをソフトウェアによって確実にしなければならない。ターゲットの EIP 値がコード・セグメント制限を超えている場合、あるいはコード・セグメント特権違反がある場合は、ターゲットの IA-32 命令で IA-32 GPFault(0) 例外が報告される。

プロセッサが IA-64 命令セットによって IA-32 命令ストリームへの書き込みを検出することは保証されていない。詳細は、6-30 ページの「自己修正コード」を参照のこと。IA-32 命令セットを入力する前に、IA-64 ソフトウェア上で、先行のすべてのレジスタ・スタック・フレームをメモリにフラッシュさせなければならない。現在および以前のレジスタ・スタック・フレームの中の残りのすべてのレジスタは、IA-32 命令セットの実行中に変更される。詳細は、6-31 ページの「IA-64 レジスタ・スタック・エンジン」を参照のこと。

6.1.4 IA-32 動作モードの移行

6-3 ページの「IA-32 命令セットの実行」で説明しているように、jmpe、br.ia、および rfi の各命令と割り込みを使ってプロセッサを 2 つの命令セット・

モードの間で移行できる。ほとんどの IA-32 モードと IA-64 の間で移行が可能である。jmpe 命令および割り込みでは、プロセッサを IA-32 の VM86、リアル・モード、または保護モードから IA-64 命令セット・モードに移行させる。IA-32 のリアル・モード、保護モード、および VM86 の定義の間のモードの移行方法は、『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』で定義されている方法と同じである。

IA-64 インタフェース・コードでは、6-12 ページの「セグメント・ディスクリプタと環境の整合性」で定義されている方法で、保護モード、リアル・モード、または VM86 の整合した環境をセットアップし、ロードする (たとえば、セグメント・セレクタおよびセグメント・ディスクリプタをロードする、など) 必要がある。プロセッサは、整合した方法で動作が行われることを保証するために、追加的なセグメント・ディスクリプタ・チェックを適用する。

6.2 IA-32 アプリケーション・レジスタの状態モデル

図 6-2 および表 6-1 に示すように、IA-32 汎用レジスタ、セグメント・セレクタ、およびセグメント・ディスクリプタは、IA-64 汎用レジスタの GR 8 〜 GR31 の下位 32 ビットにマップされる。浮動小数点レジスタ・スタック、MMX テクノロジ・レジスタ、およびストリーミング SIMD 拡張命令レジスタは、IA-64 浮動小数点レジスタの FR8 〜 FR31 にマップされる。

パラメータを直接に受け渡しできるように、IA-32 および IA-64 の整数レジスタおよび IEEE 浮動小数点レジスタとメモリのデータ型には、IA-32 命令セットと IA-64 命令セット間のバイナリ互換性が保持されている。

図 6-2 および 表 6-1 に示すように、IA-32 命令セットの実行中に、副次的作用として一部の IA-64 レジスタが変更される。通常は、IA-64 のシステム状態は IA-32 命令セットの実行の影響を受けない。IA-64 コードですべての IA-64 および IA-32 のレジスタを参照することはできるが、IA-32 命令セットによる参照は、IA-32 で参照可能なアプリケーション・レジスタの状態だけに限定される。

IA-32 命令セットと IA-64 命令セットの間の移行中に、レジスタには以下の規則が割り当てられる。

- **IA-32 状態**：IA-32 命令セットの実行中は、レジスタには IA-32 レジスタが格納される。IA-32 命令セットに切り替える前に、必要な IA-32 の値がロードされている必要がある。IA-32 命令の完了後は、これらのレジスタには IA-32 命令の実行結果が格納される。これらのレジスタには、IA-64 命令の実行中は IA-64 ソフトウェア規則に従って任意の値を格納することができる。ソフトウェア上では、これらのレジスタに対して IA-32 および IA-64 のコール規則に準拠させる必要がある。
- **変更**：" 変更される " というマークが付いたレジスタは、プロセッサが IA-32 命令の実行用にスクラッチ領域として使用する。このレジスタの値は、命令セットの移行の前後で変更される場合がある。

- **共有**：共有レジスタには、どちらの命令セットでも同様の機能を持つ値が格納される。たとえば、スタック・ポインタ (ESP) や命令ポインタ (IP) は共有される。
- **変更されない：**これらのレジスタは、IA-32 の実行によって変更されない。IA-64 コード上では、IA-32 命令セットの実行中にこれらのレジスタが変更されていないと想定してよい。IA-32 命令セットの開始時と IA-32 命令セットの終了時で、レジスタには同じ内容が格納されている。

図 6-2. IA-32 アプリケーション・レジスタのモデル

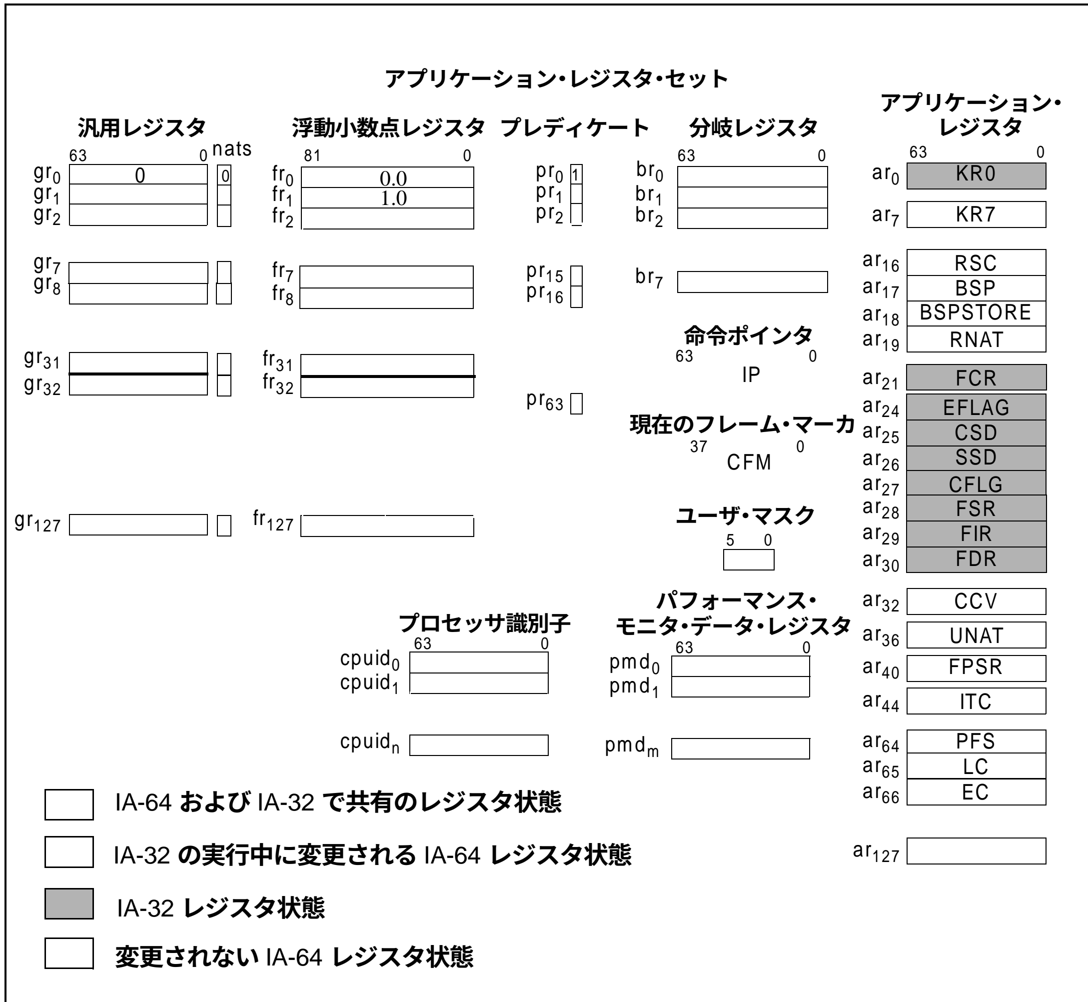


表 6-1. IA-32 アプリケーション・レジスタのマッピング

IA-64 レジスタ	IA-32 レジスタ	規則	サイズ	説明
汎用整数レジスタ				
GR0				定数 0
GR1-3		変更 [f]		IA-32 実行用のスクラッチ
GR4-7		変更されない		IA-64 の保持レジスタ
GR8	EAX	IA-32 状態	32[a]	IA-32 の汎用レジスタ
GR9	ECX			
GR10	EDX			
GR11	EBX			
GR12	ESP			
GR13	EBP			
GR14	ESI			
GR15	EDI			
GR16{15:0}	DS		64	IA-32 のセレクタ
GR16{31:16}	ES			
GR16{47:32}	FS			
GR16{63:48}	GS			
GR17{15:0}	CS			
GR17{31:16}	SS			
GR17{47:32}	LDT			
GR17{63:48}	TSS			
GR18-23		変更 [f]		IA-32 実行用のスクラッチ
GR24	ESD	IA-32 状態	64	IA-32 のセグメント・ディスクリプタ (レジスタ形式)[b]
GR25-26		変更 [f]		IA-32 実行用のスクラッチ
GR27	DSD	IA-32 状態	64	IA-32 のセグメント・ディスクリプタ (レジスタ形式)[b]
GR28	FSD			
GR29	GSD			
GR30	LDTD[c]			
GR31	GDTD			
GR32-127		変更 [d]		IA-32 コード実行空間
浮動小数点レジスタ				
IP	IP	共有	64	IA-32 および IA-64 で共有の仮想命令ポインタ

表 6-1. IA-32 アプリケーション・レジスタのマッピング (続き)

IA-64 レジスタ	IA-32 レジスタ	規則	サイズ	説明
Floating-point Registers				
FR0				定数 +0.0
FR1				定数 +1.0
FR2-5		変更されない[d]		IA-64 の保持レジスタ
FR6-7		変更[e]		IA-32 コード実行空間
FR8	MM0/FP0	IA-32 状態	64/ 80	IA-32 の MMX® テクノロジ・レジスタ (64 ビットの FP 仮数部に別名が付けられる) IA-32 の FP レジスタ (物理的レジスタのマッピング)[e]
FR9	MM1/ FP1			
FR10	MM2/FP2			
FR11	MM3/FP3			
FR12	MM4/ FP4			
FR13	MM5/FP5			
FR14	MM6/FP6			
FR15	MM7/FP7			
FR16-17	XMM0	IA-32 状態	64	IA-32 のストリーミング SIMD 拡張命令レジスタ XMM0 の下位の 64 ビットは FR16(63:0) にマップされる。 XMM0 の上位の 64 ビットは FR17(63:0) にマップされる。
FR18-19	XMM1			
FR20-21	XMM2			
FR22-23	XMM3			
FR24-25	XMM4			
FR26-27	XMM5			
FR28-29	XMM6			
FR30-31	XMM7			
FR32-127		変更[f]		IA-32 コード実行空間
Predicate Registers				
PR0				定数 1
PR1-63		変更[f]		IA-32 コード実行空間
Branch Registers				
BR0-5		変更されない		IA-64 の保持レジスタ
BR6-7		変更		IA-32 コード実行空間
Application Registers				
RSC		変更されない		IA-32 の実行には使用されない、IA-64 の保持レジスタ
BSP				
BSPSTORE				
RNAT				

表 6-1. IA-32 アプリケーション・レジスタのマッピング (続き)

IA-64 レジスタ	IA-32 レジスタ	規則	サイズ	説明
CCV		変更[f]	64	IA-32 コード実行空間
UNAT		変更されない		IA-32 の実行には使用されない、IA-64 の保持レジスタ
FPSR.sf0		変更されない		IA-64 の数値ステータスおよびコントロール
FPSR.sf1,2,3		変更[f]		IA-32 コード実行空間、IA-32 実行中に変更される
FSR	FSW,FTW, MXCSR	IA-32 状態	64	IA-32 の数値ステータス、タグ・ワード、およびストリーミング SIMD 拡張命令ステータス
FCR	FCW, MXCSR		64	IA-32 の数値およびストリーミング SIMD 拡張命令コントロール
FIR	FOP, FIP, FCS		64	IA-32 の x87 数値環境のオペコード、コード・セレクタ、および IP
FDR	FEA, FDS		64	IA-32 の x87 数値環境のデータ・セレクタおよびオフセット
ITC	TSC	共有	64	IA-32 のタイム・スタンプ・カウンタ (TSC) および IA-64 のインターバル・タイマで共有
PFS		変更されない		IA-32 コードの実行には使用されない。以前の EC が PFM に保持される
LC				
EC				
EFLAG	EFLAG	IA-32 状態	32	IA-32 のシステム / 演算フラグ、CPL および EFLAG.iopl によるいくつかのビット条件の書き込み
CSD	CSD		64	IA-32 のコード・セグメント (レジスタ形式)[b]
SSD	SSD			IA-32 のスタック・セグメント (レジスタ形式)[b]
CFLG	CR0/CR4		64	IA-32 のコントロール・フラグ CR0=CFLG(31:0)、CR4=CFLG(63:32)、CPL=0 でのみ書込み可能

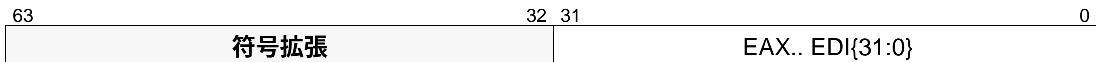
a. IA-32 命令セットへの移行時に、上位 32 ビットは無視される。終了時に上位 32 ビットはビット 31 から符号拡張される。
b. セグメント・ディスクリプタ形式は IA-32 メモリ形式と異なる。詳細は、6-10 ページの「IA-32 セグメント・レジスタ」を参照。セレクタまたはディスクリプタが変更されても、メモリ内のアクセス / ビジー・ビットはセットされない。
c. GDT/LDT ディスクリプタは、IA-64 ユーザ・レベルのコードによる変更から保護されない。
d. 現在および以前のレジスタ・フレームの中のすべてのレジスタは、IA-32 の実行中に変更される。
e. IA-32 浮動小数点レジスタのマッピングは物理的なマッピングであり、IA-32 のスタックのトップの値を反映しない。

6.2.1 IA-32 汎用レジスタ

整数レジスタは、IA-64 汎用レジスタ GR8 ～ GR15 の下位 32 ビットにマップされる。GR 8 ～ GR15 の上位 32 ビットは、IA-32 の実行時には無視される。IA-32 命令セットの実行が完了した後、GR 8 ～ GR15 の上位 32 ビットは、ビット 31 から符号拡張される。

IA-32 および IA-64 のコール規則に基づき、IA-32 命令セットを開始する前に、IA-64 コードによって必要な IA-32 状態をメモリまたはレジスタにロードしておかなければならない。

図 6-3. IA-32 汎用レジスタ (GR8 ～ GR15)



6.2.2 IA-32 命令ポインタ

プロセッサは、IA-32 命令セットの参照のための 2 つの命令ポインタとして、EIP (32 ビット実効アドレス) と IP (IA-64 命令セットの IP と同じ値の 64 ビット仮想アドレス) を持っている。IP は、EIP にコード・セグメント・ベースを加算し、64 ビットにゼロ拡張することによって生成される。IP を 8086 の 16 ビット実効アドレス命令ポインタと混同してはならない。

EIP は現在のコード・セグメント内でのオフセットであり、IP は IA-64 命令セットと共有する 64 ビット仮想ポインタである。IA-32 命令の実行中に、EIP と IP の間で下記の関係が定義される。

```
IP{63:32} = 0;
IP{31:0} = EIP{31:0} + CSD.Base;
```

EIP は IA-32 命令のフェッチごとにコード・セグメント・ベースに加算され、64 ビットの仮想アドレスにゼロ拡張される。IA-32 命令のフェッチ中に EIP がコード・セグメント制限を超えた場合は、命令の参照時に GPFault が生成される。4G バイトより上の実効命令アドレス (シーケンシャル値またはジャンプ・ターゲット) は 32 ビットに切り捨てられ、4G バイトのラップ・ラウンド条件が発生する。

6.2.3 IA-32 セグメント・レジスタ

IA-32 のセグメント・セレクタおよびセグメント・ディスクリプタは GR16 ～ GR29、および AR25 ～ AR26 にマップされる。ディスクリプタは、図 6-5 に示すように、スクランブルされない形式で保持される。この形式は、IA-32 のスクランブルされたメモリ・ディスクリプタ形式とは異なる。スクランブルされないレジスタ形式は、IA-64 コードによる IA-32 のセグメンテーションされた 16/32 ビット・ポインタの仮想アドレスへの迅速な変換をサポートするために設計されている。IA-32 セグメント・レジスタのロード命令では、セグメント・レジスタのロード時に、GDT/LDT メモリ形式をスルランブル解除して、ディスクリプタ・レジスタ形式に変換する。さらに、ディスクリプ

タ・レジスタがソフトウェアによって適切にスクランブル解除されている場合は、IA-64 ソフトウェアによってディスクリプタ・レジスタを直接にロードできる。全ビット・フィールドとフィールド・セマンティクスの定義の詳細は、「インテル・アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル」を参照のこと。

図 6-4. IA-32 セグメント・レジスタ・セレクタの形式

63 … 48	47 … 32	31 … 16	15 … 0	
GS	FS	ES	DS	GR16
TSS	LDT	SS	CS	GR17

図 6-5. IA-32 コード / データ・セグメント・レジスタ・ディスクリプタの形式

63	62	61	60	59	58 57	56	55 … 52	51 … 32	31 … 0
g	d/b	ig	av	p	dpl	s	type	lim{19:0}	base{31:0}

表 6-2. IA-32 セグメント・レジスタのフィールド

フィールド	ビット	説明
selector	15:0	セグメント・セレクタ値。ビットの定義については『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。
base	31:0	セグメント・ベース値。この値は、64 ビットにゼロ拡張されて、IA-32 命令セットでのメモリ参照のための 64 制限ビット仮想アドレス空間内のセグメントの開始位置を指す。
lim	51:32	セグメント制限。IA-32 命令セットでのメモリ参照で、上に拡張するセグメントに対するセグメント内の最大実効アドレス値が入る。下に拡張するセグメントに対しては、セグメント内の最小実効アドレス値を定義する。セグメント制限の詳細およびセグメント制限のフォルト条件については『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。セグメントの g ビットが 1 のとき、セグメントは (lim << 12) ｜ 0xFFF 倍に拡大される。
type	55:52	データおよびコードのセグメント・タイプの識別子で、アクセス・ビット (ビット 52) が入る。エンコーディングおよび定義については『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。
s	56	非システム・セグメント。1 のときはデータ・セグメント、0 のときはシステム・セグメント。
dpl	58:57	ディスクリプタ特権レベル。IA-32 命令セットでのメモリ参照において、DPL をチェックしてメモリ・アクセス許可を調べる。
p	59	セグメント存在ビット。0 のときに IA-32 でのメモリ参照でこのセグメントを使用すると、データ・セグメント (CS、DS、ES、FS、GS) については IA-32_Exception (GPFault) が生成され、SS については IA-32_Exception (StackFault) が生成される。
av	60	無視。CS および SS ディスクリプタでは、このフィールドを読み取るとゼロが返される。DS、ES、FS および GS ディスクリプタでは、このフィールドを読み取ると、最後に IA-64 コードによって書き込まれた値が返される。このフィールドが IA-32 ディスクリプタのロードによって書き込まれた場合は、このフィールドを読み取るとゼロが返される。IA-32 命令セットの実行時には、プロセッサはこのフィールドを無視する。ソフトウェア上で使用可能で、将来はこのフィールドは使用されなくなる。

表 6-2. IA-32 セグメント・レジスタのフィールド (続き)

フィールド	ビット	説明
ig	61	無視。CS および SS ディスクリプタでは、このフィールドを読み取るとゼロが返される。DS、ES、FS および GS ディスクリプタでは、このフィールドを読み取ると、最後に IA-64 コードによって書き込まれた値が返される。このフィールドが IA-32 ディスクリプタのロードによって書き込まれた場合は、このフィールドを読み取るとゼロが返される。IA-32 命令セットの実行時には、プロセッサはこのフィールドを無視する。このフィールドは将来使用される可能性があり、ソフトウェアによってゼロに設定しなければならない。
d/b	62	セグメント・サイズ。0 のとき、セグメント内の IA-32 命令セットの実効アドレスは 16 ビットに切り捨てられる。1 の場合、実効アドレスは 32 ビットである。コード・セグメントの d/b ビットはまた、IA-32 命令のデフォルトのオペランド・サイズを制御する。1 のとき、デフォルトのオペランド・サイズは 32 ビットで、0 の場合は 16 ビットである。
g	63	セグメント制限の単位。1 のとき、IA-32 命令セットによるメモリ参照においてセグメント制限を lim=(lim<<12)|0xFFF のスケールで拡大する。IA-64 命令セットによるメモリ参照では、このフィールドは無視される。

6.2.3.1 データ・セグメントおよびコード・セグメント

IA-32 コードへの移行の際に、IA-32 セグメント・ディスクリプタ・レジスタおよびセグメント・セレクタ・レジスタ (GDT、LDT、DS、ES、CS、SS、FS および GS) は、IA-32 および IA-64 のコール規則と、使用するセグメンテーションモデルに基づいて、IA-64 コードによって、要求される値に初期設定されなければならない。

IA-64 コードによって手動で LDT/GDT から 8 バイトを取り出してディスクリプタをロードし、ディスクリプタのスクランブルを解除し、セグメント・ベース、制限および属性を書き込むことができる。別の方法として、IA-64 ソフトウェア上で IA-32 命令セットに切り替え、IA-32 の `Mov Sreg` 命令を使って要求されるセグメント・ロードを実行することができる。IA-64 コードで明示的にセグメント・ディスクリプタをロードする場合、IA-64 コード上でセグメント・ディスクリプタの整合性を保証する必要がある。

プロセッサは、メモリ内のディスクリプタとディスクリプタ・レジスタの間の一貫性を保証していない。また、プロセッサは、セグメント・レジスタが IA-64 命令によってロードされた場合には LDT/GDT のセグメント・アクセス・ビットをセットしない。

6.2.3.2 セグメント・ディスクリプタと環境の整合性

IA-32 命令セットの実行の際に、大部分のセグメント保護チェックは、セグメント・ディスクリプタが IA-32 命令によってセグメント・レジスタにロードされるときにプロセッサによって行われる。しかし、セグメント・ディスクリプタが IA-64 命令セットによって汎用レジスタ・ファイルにロードされるときは、そのような保護チェックは行われず、セグメント・アクセス・ビットは更新されない。

IA-64 ソフトウェアによってディスクリプタを直接にロードする場合は、そのソフトウェアによってディスクリプタの有効性を保証し、IA-32 の保護モード、リアル・モードまたは VM86 環境の整合性を保証しなければならない。表 6-3 に、初期の IA-32 環境を確立するためのソフトウェア上のガイドラインを示す。プロセッサは、6-16 ページの 6.2.3.3 項「IA-32 環境の実行時整合性チェック」で定義されている方法で IA-32 環境の整合性をチェックする。IA-64 コードと IA-32 コードの間の移行の際には、プロセッサはセグメント・ディスクリプタのベース、制限および属性値を変更せず、特権レベルも変更されない。

表 6-3. IA-32 環境の初期レジスタ状態

レジスタ	フィールド	リアル・モード	保護モード	VM86 モード
PSR	**cpl**	**0**	**特権レベル**	**3**
EFLAG	**vm**	**0**	**0**	**1**
CR0	**pe**	**0**	**1**	**1**
CS	selector	base >> 4[a]	selector	base >> 4
	base	selector << 4[b]	base	selector << 4
	dpl	PSR.cpl (0)	PSR.cpl[c]	PSR.cpl (3)
	d-bit	16-bit[d]	16/32-bit	16-bit
	type	data rd/wr, expand up	execute	data rd/wr, expand up
	s-bit	1	1	1
	p-bit	1	1	1
	a-bit	1	1	1
	g-bit/limit	0xFFFF[e]	limit	0xFFFF
SS	selector	base >> 4[a]	selector	base >> 4
	base	selector << 4[b]	base	selector << 4
	dpl	PSR.cpl (0)	PSR.cpl	PSR.cpl (3)
	d-bit	16-bit[d]	16/32-bit size	16-bit
	type	data rd/wr, expand up	data types	data rd/wr, expand up
	s-bit	1	1	1
	p-bit	1	1	1
	a-bit	1	1	1
	g-bit/limit	0xFFFF[e]	limit	0xFFFF

表 6-3. IA-32 環境の初期レジスタ状態 (続き)

レジスタ	フィールド	リアル・モード	保護モード	VM86 モード
PSR	**cpl**	**0**	**特権レベル**	**3**
EFLAG	**vm**	**0**	**0**	**1**
CR0	**pe**	**0**	**1**	**1**
DS, ES, FS, GS	selector	base >> 4[a]	selector	base >> 4
	base	selector << 4[b]	base	selector << 4
	dpl	dpl >= PSR.cpl (0)	dpl >= PSR.cpl	dpl >= PSR.cpl (3)
	d-bit	16-bit[d]	16/32-bit	0
	type	data rd/wr, expand up	data types	data rd/wr, expand up
	s-bit	1	1	1
	a-bit	1	1	1
	p-bit	1	1/0[f]	1
	g-bit/limit	0xFFFF[e]	limit	0xFFFF
LDT,GDT, TSS	selector	na	selector	
	base		base	
	dpl		dpl >= PSR.cpl	
	d-bit		0	
	type		ldt/gdt/tss types	
	s-bit		0	
	p-bit		1	
	a-bit		1	
	g-bit/limit		limit	

a. 通常の RM 64KB 動作では、セレクタは 16*base に設定する。
b. 通常の RM 64KB 動作では、セグメント・ベースは selector/16 に設定する。
c. 適合するコード・セグメントを指定しない場合
d. 通常の RM 64KB 動作では、セグメント・サイズは 16 ビットに設定する
e. 通常の RM 64KB 動作では、セグメント制限は 0xFFFF に設定する
f. 有効なセグメントについては p ビットは 1 にセットする。null セグメントについては p ビットは 0 にセットする。

6.2.3.2.1 保護モード

IA-32 命令セットに移行する前に、IA-64 ソフトウェアによって、下記の規則に従って、保護モード環境に対応するセグメント・ディスクリプタをセットアップしなければならない。

- スタック・セグメント・ディスクリプタ・レジスタが DPL==PSR.cpl であることを IA-64 ソフトウェアによって確実にしなければならない。

- DSD、ESD、FSD および GSD セグメント・ディスクリプタ・レジスタについて、DPL>=PSR.cpl であることを IA-64 ソフトウェアによって確実にしなければならない。
- CSD セグメント・ディスクリプタ・レジスタについて、DPL==PSR.cpl (適合するコード・セグメントを除く) であることを IA-64 ソフトウェアによって確実にしなければならない。
- すべてのコード、スタックおよびデータのセグメント・ディスクリプタ・レジスタがシステム・セグメントのエンコーディングを含んでいないことをソフトウェアによって確実にしなければならない。
- すべてのセグメント・ディスクリプタ・レジスタの a ビットが 1 に設定されていることをソフトウェアによって確実にしなければならない。
- すべての有効なデータ・セグメントの p ビットが 1 に設定され、すべての Null データ・セグメントの p ビットが 0 に設定されていることをソフトウェアによって確実にしなければならない。

6.2.3.2.2 VM86

IA-32 命令セットに移行する前に、IA-64 ソフトウェアによって、下記の規則に従って、VM86 環境に対応するセグメント・ディスクリプタをセットアップしなければならない。

- PSR.cpl は 3 でなければならない (`rfi` の場合は IPSR.cpl は 3 でなければならない)。
- スタック・セグメント・ディスクリプタ・レジスタが DPL==PSR.cpl==3 で、16 ビット、データ読み取り / 書き込み、上に拡張に設定されていることを IA-64 ソフトウェアによって確実にしなければならない。
- CSD、DSD、ESD、FSD および GSD セグメント・ディスクリプタ・レジスタについて、DPL==3 で、16 ビット、データ読み取り / 書き込み、上に拡張に設定されていることを IA-64 ソフトウェアによって確実にしなければならない。
- すべてのコード、スタックおよびデータのセグメント・ディスクリプタ・レジスタがシステム・セグメントのエンコーディングを含んでいないことをソフトウェアによって確実にしなければならない。
- すべてのセグメント・ディスクリプタ・レジスタの P ビットおよび A ビットが 1 に設定されていることをソフトウェアによって確実にしなければならない。
- すべての DSD、CSD、ESD、SSD、FSD および GSD セグメント・ディスクリプタ・レジスタについて Base=Selector*16 の関係が維持されていることをソフトウェアによって確実にしなければならない。そうでない場合のプロセッサの動作は予測できない。
- DSD、CSD、ESD、SSD、FSD および GSD セグメント・ディスクリプタ・レジスタの制限値が 0xFFFF に設定されていることをソフトウェアによって確実にしなければならない。そうでない場合には、擬似セグメント制限フォルト (GPFault または スタック・フォルト) が発生する。

- すべてのセグメント・ディスクリプタ・レジスタ (コード・セグメントを含む) がデータ読み取り / 書き込みであることをソフトウェアによって確実にしなければならない。

6.2.3.2.3 リアル・モード

IA-32 命令セットに移行する前に、IA-64 ソフトウェアによって、下記の規則に従って、リアル・モード環境に対応するセグメント・ディスクリプタをセットアップしなければならない。そうでない場合のソフトウェアの動作は予想できない。

- PSR.cpl が 1 であることを IP-64 ソフトウェアによって確実にしなければならない。
- スタック・セグメント・ディスクリプタ・レジスタの DPL が 0 であることを IA-64 ソフトウェアによって確実にしなければならない。
- すべてのコード、スタックおよびデータのセグメント・ディスクリプタ・レジスタがシステム・セグメントのエンコーディングを含んでいないことをソフトウェアによって確実にしなければならない。
- すべてのセグメント・ディスクリプタ・レジスタの P ビットおよび A ビットが 1 に設定されていることをソフトウェアによって確実にしなければならない。
- 通常のリアル・モードの 64K 動作で、すべての DSD、CSD、ESD、SSD、FSD および GSD セグメント・ディスクリプタ・レジスタについて Base=Selector*16 の関係が維持されていることをソフトウェアによって確実にしなければならない。
- 通常のリアル・モードの 64K 動作で、DSD、CSD、ESD、SSD、FSD および GSD セグメント・ディスクリプタ・レジスタの制限値が 0xFFFF に設定され、セグメント・サイズが 16 ビット (64K) に設定されていることをソフトウェアによって確実にしなければならない。
- 通常のリアル・モード動作で、すべてのセグメント・ディスクリプタ・レジスタ (コード・セグメントを含む) が読み取り可能 / 書き込み可能であることを IA-64 ソフトウェアによって確実にしなければならない。

6.2.3.3 IA-32 環境の実行時整合性チェック

IA-64 プロセッサは IA-32 環境の整合性をチェックするための追加的な実行時チェックを実行する。これらのチェックは、IA-32 プロセッサについて定義されている実行時チェックに追加されるものであり、表 6-4 で強調表示されている。既存の IA-32 実行時チェックも示しているが、強調表示はされていない。表に示していないディスクリプタ・フィールドはチェックされない。表に示すように、実行時チェックは、IA-32 命令コードのフェッチ時、あるいは指定されたいずれかのセグメント・レジスタに対する IA-32 でのデータ・メモリ参照時に実行される。

表 6-4. IA-32 環境の実行時整合性チェック

<table>
<tr><th>参照</th><th>リソース</th><th>リアル・モード</th><th>保護モード</th><th>VM86 モード</th><th>フォルト</th></tr>
<tr><td rowspan="3">すべての
コードの
フェッチ</td><td>PSR.cpl</td><td>0 でない</td><td>無視</td><td>3 でない</td><td rowspan="3">コード・
フェッチ・
フォルト
(GPFault
(0))[a]</td></tr>
<tr><td>EFLAG.vm
CFLG.pe</td><td colspan="3">EFLAG.vm が 1 で、CFLG.pe が 0</td></tr>
<tr><td>EFLAG.vif
EFLAG.vip</td><td colspan="3">EFLAG.vip & EFLAG.vif & CFLG.pe &
PSR.cpl==3 &
(CFLG.pvi | (EFLAG.vm & CFLG.vme))</td></tr>
<tr><td rowspan="6">すべての
コードの
フェッチ、
CS</td><td>dpl</td><td colspan="2" rowspan="2">無視</td><td>dpi が 3 でない</td><td rowspan="6">コード・
フェッチ・
フォルト
(GPFault+
(0))</td></tr>
<tr><td>d-bit</td><td>16 ビットでない</td></tr>
<tr><td>type</td><td colspan="3">無視 (exec または data)</td></tr>
<tr><td></td><td colspan="3">データが下に拡張する場合、GPFault</td></tr>
<tr><td>s, p, a-bits</td><td colspan="3">1 でない</td></tr>
<tr><td>g-bit/limit</td><td colspan="3">セグメント制限違反</td></tr>
<tr><td rowspan="6">SS に対す
るデータ・
メモリ参照</td><td>dpl</td><td colspan="3">dpl!=PSR.cpl</td><td rowspan="6">スタック・
フォルト</td></tr>
<tr><td>d-bit</td><td colspan="2">無視</td><td>16 ビットでない</td></tr>
<tr><td rowspan="2">type</td><td colspan="2">無視</td><td>データが下に拡張</td></tr>
<tr><td colspan="3">読み込みと読み込み不可、書き込みと書き込み不可</td></tr>
<tr><td>s, p, a-bits</td><td colspan="3">1 でない</td></tr>
<tr><td>g-bit/limit</td><td colspan="3">セグメント制限違反</td></tr>
<tr><td rowspan="6">DS、ES、
FS および
GS に対す
るデータ・
メモリ参照</td><td>dpl</td><td colspan="3">無視</td><td rowspan="6">GPFault(0)</td></tr>
<tr><td>d-bit</td><td colspan="2">無視</td><td>16 ビットでない</td></tr>
<tr><td rowspan="2">type</td><td colspan="2">無視</td><td>データが下に拡張</td></tr>
<tr><td colspan="3">読み込みと読み込み不可、書き込みと書き込み不可</td></tr>
<tr><td>s, p, a-bits</td><td colspan="3">1 でない</td></tr>
<tr><td>g-bit/limit</td><td colspan="3">セグメント制限違反</td></tr>
<tr><td rowspan="6">CS に対す
るデータ・
メモリ参照</td><td>dpl</td><td colspan="3">無視</td><td rowspan="6">GPFault(0)</td></tr>
<tr><td>d-bit</td><td colspan="2">無視</td><td>16 ビットでない</td></tr>
<tr><td rowspan="2">type</td><td colspan="2">無視</td><td>データが下に拡張</td></tr>
<tr><td>rd/wr チェック
は無視される</td><td>rd チェック
で読み取り
不可、
wr チェック
で書き込み
不可</td><td>rd/wr チェックは
無視される</td></tr>
<tr><td>s, p, a-bits</td><td colspan="3">1 でない</td></tr>
<tr><td>g-bit/limit</td><td colspan="3">セグメント制限違反</td></tr>
</table>

表 6-4. IA-32 環境の実行時整合性チェック (続き)

参照	リソース	リアル・モード	保護モード	VM86 モード	フォルト
すべてのコードのフェッチ	PSR.cpl	0 でない	無視	3 でない	コード・フェッチ・フォルト (GPFault (0))[a]
	EFLAG.vm CFLG.pe	EFLAG.vm が 1 で、CFLG.pe が 0			
	EFLAG.vif EFLAG.vip	EFLAG.vip & EFLAG.vif & CFLG.pe & PSR.cpl==3 & (CFLG.pvi | (EFLAG.vm & CFLG.vme))			
LDT、GDT、TSS に対するメモリ参照	dpl	無視			GPFault (Selector/ 0)[b]
	type	無視			
	s-bit	1 でない			
	a, d-bits	無視			
	p-bit	1 でない			
	g-bit/limit	セグメント制限違反			

a. コード・フェッチ・フォルトはより高い優先順位の GPFault(0) として渡される。
b. GDT または LDT を参照しているときは GP フォルトのエラー・コードはセレクタ値である。そうでない場合のエラー・コードはゼロである。

6.2.4 IA-32 アプリケーション EFLAG レジスタ

EFLAG (AR24) レジスタは、ユーザ算術フラグ (CF、PF、AF、ZF、SF、OF および ID) と、システム・コントロール・フラグ (TF、IF、IOPL、NT、RF、VM、AC、VIF、VIP) の 2 つの主要要素から成っている。算術フラグもシステム・フラグも IA-64 命令の実行には影響を及ぼさない。

図 6-6. IA-32 EFLAG レジスタ (AR24)

31–22	21	20	19	18	17	16	15	14	13–12	11	10	9	8	7	6	5	4	3	2	1	0
予約 (0 にセットされる)	id	vip	vif	ac	vm	rf	0	nt	iopl	of	df	if	tf	sf	zf	0	af	0	pf	1	cf

63–32
予約 (0 にセットされる)

IA-32 命令セットでは、算術フラグを使用して、IA-32 演算の状態を反映し、IA-32 ストリング操作を制御し、IA-32 命令の分岐条件を制御している。これらのフラグは IA-64 命令では無視される。ID、OF、DF、SF、ZF、AF、PF および CF の各フラグは『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』で定義されている。

表 6-5. IA-32 EFLAG レジスタのフィールド

EFLAG[a]	ビット	説明
cf	0	IA-32 キャリー・フラグ。詳細は『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。
	1	無視 - 書き込みがあっても無視される。読み取ると、IA-32 命令および IA-64 命令は 1 を返す。
	3,5, 15	無視 - 書き込みがあっても無視される。読み取ると、IA-32 命令および IA-64 命令は 0 を返す。ソフトウェアでこれらのビットを 0 に設定しなければならない。
pf	2	IA-32 パリティ・フラグ。詳細は『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。
af	4	IA-32 予備フラグ。詳細は『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。
zf	6	IA-32 ゼロ・フラグ。詳細は『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。
sf	7	IA-32 符号フラグ。詳細は『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。
tf	8	IA-32 システム EFLAG レジスタ。
if	9	
df	10	IA-32 方向フラグ。詳細は『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。
of	11	IA-32 オーバフロー・フラグ。詳細は『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。
iopl	13:12	IA-32 システム EFLAG レジスタ。
nt	14	
rf	16	
vm	17	
ac	18	
vif	19	
vip	20	
id	21	
	63:22	予約。0 に設定しなければならない。

a. IA-32 命令セットに移行すると、すべてのビットは後続の IA-32 命令で読み取り可能になり、IA-32 命令セットを終了すると、これらのビットはそれまでのすべての IA-32 命令の結果を表す。EFLAG ビットは IA-64 命令セットの実行動作には影響を及ぼさない。

6.2.5 IA-32 浮動小数点レジスタ

IA-32 浮動小数点レジスタのスタック、数値コントロール、および環境は、表 6-6 に示すように IA-64 浮動小数点レジスタの FR8 ～ FR15 およびアプリケーション・レジスタの名前空間にマップされる。

表 6-6. IA-32 浮動小数点レジスタのマッピング

IA-64 レジスタ	IA-32 レジスタ	サイズ (ビット)	説明
FR8	ST[(TOS + N)==0]	80	IA-32 数値レジスタ・スタック IA-64 で FR8 ～ FR15 にアクセスすると、IA-32 の TOS 調整は無視される。 IA-32 でアクセスすると、指定されたレジスタ N について TOS 調整を使用する
FR9	ST[(TOS + N)==1]		
FR10	ST[(TOS + N)==2]		
FR11	ST[(TOS + N)==3]		
FR12	ST[(TOS + N)==4]		
FR13	ST[(TOS + N)==5]		
FR14	ST[(TOS + N)==6]		
FR15	ST[(TOS + N)==7]		
FCR (AR21)	FCW, MXCSR	64	IA-32 数値およびストリーミング SIMD 拡張命令のコントロール・レジスタ
FSR (AR28)	FSW,FTW, MXCSR	64	IA-32 数値およびストリーミング SIMD 拡張命令のステータスおよびタグワード
FIR (AR29)	FOP, FCS, FIP	64	IA-32 数値命令ポインタ
FDR (AR30)	FDS, FEA	48	IA-32 数値データ・ポインタ

6.2.5.1 IA-32 浮動小数点スタック

IA-32 浮動小数点レジスタは、以下のように定義される。

- IA-32 数値レジスタ・スタックは、Intel 8087 の 80 ビット IEEE 浮動小数点形式を使って FR8 ～ FR15 にマップされる。
- IA-32 命令セットによる参照では、浮動小数点レジスタは FCR.top に保持されている IA-32 のスタックのトップ (TOS) ポインタをもとに、FR8 ～ FR15 に論理的にマップされる。TOS 調整の後、FR8 は物理レジスタを表し、必ずしも論理的浮動小数点レジスタ・スタックのトップではなくなる。
- IA-64 命令セットによる参照では、浮動小数点レジスタの番号は物理的番号であり、数値 TOS ポインタとは関係しない。たとえば、FR8 を参照すると、TOS の値に関わりなく常に物理レジスタ FR8 の値が返される。IA-64 ソフトウェア上で、必ずしも FR8 に IA-32 の論理レジスタ ST(0) が格納されていると想定することはできない。標準的な IA-32 コール規則を使って、浮動小数点値をメモリを介して渡すことを強く推奨する。

6.2.5.2 IA-32/IA-64 の特殊なケース

IA-32 浮動小数点命令で単精度または倍精度のデノーマル数をロードすると、正規化された拡張倍精度値がターゲットの浮動小数点レジスタに入る。IA-64 命令で、単精度または倍精度のデノーマル数をロードすると、正規化されないデノーマル値がターゲットの浮動小数点レジスタに入る。2 つの IA-64 標準指数値があり、単精度デノーマル数と倍精度デノーマル数を示す。

IA-64 から IA-32 命令に浮動小数点値を渡すときは、標準的な IA-32 コール規則を使って、浮動小数点値をメモリ・スタックを介して渡すことを強く推奨する。ソフトウェア上で浮動小数点レジスタを介して IA-64 コードから IA-32 コードに浮動小数点値を渡す場合には、ソフトウェア上で以下の条件を確実にしなければならない。

- IA-64 の単精度または倍精度のデノーマル数は IA-32 命令に必要な正規化された拡張倍精度値に変換しなければならない。ソフトウェアによって、IA-64 のデノーマル数に拡張倍精度の 1.0 を掛けることによって変換することができる (`fma.sfx fr = fr、f1、f0`)。IA-32 浮動小数点操作で無効な単精度または倍精度のデノーマル数が検出された場合は、IA-32 例外 (FPError Invalid Operand) フォルトが生成される。
- 浮動小数点値は、IA-32 の 80 ビット (15 ビット指数) 拡張倍精度形式の範囲内でなければならない。IA-64 では、中間計算で 82 ビット (最大指数範囲 17 ビット) を使用できる。IA-32 命令に渡されるすべての IA-64 浮動小数点レジスタ値が拡張倍精度の 80 ビット形式で表現できることを保証しなければならない。そうでないと、プロセッサの動作はモデル固有になり、定義されない。定義されない動作として、以下のことが発生する可能性がある (これら以外の状態が発生する可能性もある)。つまり、IA-32 浮動小数点命令で使用したときに IA-32_Exception (FPError Invalid Operation) フォルトが発生したり、範囲外の値がゼロ / デノーマル数 / 無限大に丸められ、IA-32_Exception (FPError Overflow/Underflow) フォルトが発生したり、あるいは、フォルトが発生せずに範囲外の値を含む浮動小数点レジスタが QNAN または SNAN に変換されたりする (この変換は IA-32 命令セットへの移行時、または IA-32 浮動小数点命令による使用時に行われる) ことがある。ソフトウェアによって、`fma.sfx fr = fr、f1、f0` を使って拡張倍精度形式の 1.0 を掛けることによって (このとき最大範囲の指数フィールドをディセーブルにしておく)、渡されるすべての浮動小数点レジスタ値が範囲内であることを保証することができる。
- IA-64 浮動小数点 NaTVal 値を IA-32 浮動小数点命令に伝播してはならない。そうしないと、プロセッサの動作はモデル固有になり、定義されない。プロセッサは、フォルトを発生せずに NaTVal を含む浮動小数点レジスタを SNAN に変換することがある (この変換は IA-32 命令セットへの移行時、または IA-32 浮動小数点命令による使用時に行われる)。IA-32 命令セットによって、伝播された NaTVal 値を直接または間接的に使用すると、NaTVal 値を伝播するか、あるいは IA-32_Exception (FPError Invalid Operand) フォルトを生成する。プロセッサがフォルトを生成するか、NaTVal 値を伝播するかはモデルによって異なる。プロセッサが NaTVal が格納されているレジスタの使用を許可する場合には、必ず、NaTVal 値を伝播するか、IA-32_Exception (FPError Invalid Operand) フォルトを生成する。

注： IA-32 コードで IA-32 浮動小数点ロード命令を使ってメモリ位置から NaTVal 値を読み取ることはできない。なぜなら NaTVal 値は 80 ビット拡張倍精度値では表現できないからである。

NaTVal および IA-64 デノーマル数の数値に関する問題を避けるために、標準的な IA-32 コール規則を使って浮動小数点値をメモリ・スタックに渡すことを強く推奨する。

6.2.5.3 IA-32 浮動小数点コントロール・レジスタ

FPSR は、IA-64 浮動小数点命令のコントロール / ステータス・ビットを制御する。FPSR は IA-32 浮動小数点命令は制御せず、また、IA-32 浮動小数点命令のステータスは反映しない。IA-32 浮動小数点命令とストリーミング SIMD 拡張命令には、別個のコントロール・レジスタとステータス・レジスタ、つまり、FCR (浮動小数点コントロール・レジスタ) および FSR (浮動小数点ステータス・レジスタ) がある。

FCR には、図 6-7 に示すように、IA-32 FCW ビットと、すべてのストリーミング SIMD 拡張命令コントロール・ビットが含まれる。

FSR には、図 6-8 に示すように、IA-32 浮動小数点ステータス・フラグ FSW、FTW と、ストリーミング SIMD 拡張命令ステータス・ビットが含まれる。タグ・フィールドは、対応する IA-32 論理浮動小数点レジスタが空白かどうかを示す。それぞれの IA-32 論理浮動小数点レジスタのゼロおよび NaN、無限大、デノーマル数などの特殊条件のタグ・エンコーディングはサポートされていない。ただし、IA-32 命令セットによって FTW を読み取ると、それぞれの IA-32 浮動小数点レジスタについて追加的な特殊条件が計算される。IA-64 コードで浮動小数点分類命令を発行して、それぞれの IA-32 浮動小数点レジスタの処置を決定することができる。

FCR と FSR はすべての IA-32 浮動小数点のコントロール、ステータスおよびタグに関する情報をまとめて保持する。MXSCR、FCR、FSW および FTAG によって更新および制御される IA-32 命令は、実質的には FSR を更新し、FSR によって制御される。MXCSR、FSW、FCW および FTW に対する IA-32 読み取り / 書き込みは、FSR および FCR に対する IA-64 読み取り / 書き込みと同じ情報を返す。

IA-32 命令セットに移行する前に、FCR および FSR が IA-32 数値命令の実行のために適切にロードされていることをソフトウェアによって確実にしなければならない。

図 6-7. IA-32 浮動小数点コントロール・レジスタ (FCR)

IA-32 FCW{12:0}

ビット	31–13	12	11–10	9–8	7	6	5	4	3	2	1	0
フィールド	予約 (0 にセット)	IC	RC	PC	0	1	PM	UM	OM	ZM	DM	IM

ビット	63–48	47	46–45	44	43	42	41	40	39	38	37–32
フィールド	予約 (0 にセット)	FZ	RC	PM	UM	OM	ZM	DM	IM	rv	無視

IA-32 MXCSR (コントロール)

図 6-8. IA-32 IA-32 浮動小数点ステータス・レジスタ (FSR)

IA-32 FTW{15:0} | IA-32 FSW{15:0}

ビット	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
フィールド	0	TG7	0	TG6	0	TG5	0	TG4	0	TG3	0	TG2	0	TG1	0	TG0

ビット	15	14	13–11	10	9	8	7	6	5	4	3	2	1	0
フィールド	B	C3	TOP	C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE

ビット	63–48	47–39	38	37	36	35	34	33	32
フィールド	予約 (0 にセット)	無視	rv	PE	UE	OE	ZE	DE	IE

IA-32 MXCSR (ステータス)

表 6-7. IA-32 浮動小数点コントロール・レジスタのマッピング (FCR)

IA-32 の状態	IA-64 の状態	ビット	IA-32 での使用法	IA-64 での使用法
FCR レジスタでの FCW、MXCSR の状態				
FCW.im	FCR.im	0	無効操作マスク	これらの IA-32 の数値およびストリーミング SIMD 拡張命令のコントロール・ビットは IA-64 浮動小数点命令の実行に影響を及ぼさない。 各フィールドの詳細については『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。
FCW.dm	FCR.dm	1	デノーマル・オペランド・マスク	
FCW.zm	FCR.zm	2	ゼロ除算マスク	
FCW.om	FCR.om	3	オーバーフロー・マスク	
FCW.um	FCR.um	4	アンダーフロー・マスク	
FCW.pm	FCR.pm	5	精度マスク	
無視		6	無視 - 書き込みがあっても無視される、読み取ると 1 を返す	
無視		7, 32:37	無視 - 書き込みがあっても無視される、読み取ると 0 を返す	
予約		13:31,38, 48:63	予約	
FCW.pc	FCR.pc	8:9	精度制御 (00 - 単精度、10 - 倍精度、11 - 拡張倍精度)	
FCW.rc	FCR.rc	10:11	丸め (00 - 偶数、01 - 切り下げ、10 - 切り上げ、11 - 切り捨て)	
FCW.ic	FCR.ic	12	(無限大の制御) - すべての IA-64 プロセッサで無視される。IA-32 プロセッサとの互換性のために提供されている。	
MXCSR.im	FCR.im	39	ストリーミング SIMD 拡張命令の無効操作マスク	
MXCSR.dm	FCR.dm	40	ストリーミング SIMD 拡張命令のデノーマル・オペランド・マスク	
MXCSR.zm	FCR.zm	41	ストリーミング SIMD 拡張命令のゼロ除算マスク	
MXCSR.om	FCR.om	42	ストリーミング SIMD 拡張命令のオーバーフロー・マスク	
MXCSR.um	FCR.um	43	ストリーミング SIMD 拡張命令のアンダーフロー・マスク	
MXCSR.pm	FCR.pm	44	ストリーミング SIMD 拡張命令の精度マスク	
MXCSR.rc	FCR.rc	45:46	ストリーミング SIMD 拡張命令の丸めマスク (00 - 偶数、01 - 切り下げ、10 - 切り上げ、11 - 切り捨て)	
MXCSR.fz	FCR.fz	47	ストリーミング SIMD 拡張命令のゼロ・フラッシュ	

表 6-8. IA-32 浮動小数点ステータス・レジスタのマッピング (FSR)

IA-32 の状態	A-64 の状態	ビット	IA-32 での使用法	IA-64 での使用法
FSR レジスタでの FSW、FTW、MXCSR の状態				
FSW.ie	FSR.ie	0	無効操作例外	これらのビットは IA-64 浮動小数点の実行のステータスを反映しない。 IA-32 数値フラグの詳細については『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。
FSW.de	FSR.de	1	デノーマル・オペランド例外	
FSW.ze	FSR.ze	2	ゼロ除算例外	
FSW.oe	FSR.oe	3	オーバーフロー例外	
FSW.ue	FSR.ue	4	アンダーフロー例外	
FSW.pe	FSR.pe	5	精度例外	
FSW.sf	FSR.sf	6	スタック・フォルト	
FSW.es	FSR.es[a]	7	エラー・サマリ	
FSW.c3:0	FSR.c3:0	8:10,14	数値条件コード	
FSW.top	FSR.top	11:13	IA-32 数値スタックのトップ	
FSW.b	FSR.b	15	IA-32 の FPU Busy は常に FSW.ES の状態と同じ	
FTW	FSR.tg {7:0}[b]	16,18,20, 22,24,26, 28,30	数値タグ。0 - 空白でない、1 - 空白[c]	
0		17,19,21, 23,25,27, 29,31,39, 47	無視 - 書き込みがあっても無視される、読み取ると 0 を返す	
MXCSR.ie	FSR.ie	32	ストリーミング SIMD 拡張命令の無効操作例外	IA-64 浮動小数点の実行のステータスを反映しない。 詳細については『IA-32 インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照。
MXCSR.de	FSR.de	33	ストリーミング SIMD 拡張命令のデノーマル・オペランド例外	
MXCSR.ze	FSR.ze	34	ストリーミング SIMD 拡張命令のゼロ除算例外	
MXCSR.oe	FSR.oe	35	ストリーミング SIMD 拡張命令のオーバーフロー例外	
MXCSR.ue	FSR.ue	36	ストリーミング SIMD 拡張命令のアンダーフロー例外	
MXCSR.pe	FSR.pe	37	ストリーミング SIMD 拡張命令の精度例外	
予約		38, 48:63	予約	
無視		39:47	無視 - 書き込みがあっても無視される、読み取ると 0 を返す	

a. 例外サマリ・ビット。詳細は 6.2.5.4 項を参照。
b. タグのエンコーディングは、0、NaN、無限大またはデノーマル数を含んでいるそれぞれの IA-32 数値レジスタが IA-64 による FSR の読み取りでサポートされているかどうかを示す。
c. すべての MMX® テクノロジ命令では、すべての数値タグを 0 = NotEmpty に設定する。しかし、MMX テクノロジ命令 EMMS では、すべての数値タグを 1 = Empty に設定する。

6.2.5.4 浮動小数点環境

Intel 8087 遅延数値例外モデルをサポートするために、FSR、FDR、および FIR には数値例外に関連する保留情報が含まれている。FDR には、オペランドの実効アドレスおよびセグメント・セレクタが含まれている。FIR には、数値命令の実効アドレス、コード・セグメント・セレクタ、およびオペコード・ビットが含まれている。FSR は、数値例外のタイプのサマリを IE、DE、ZE、OE、UE、PE、SF、ES ビットに保持する。ES ビットは、下記のように IA-32 浮動小数点例外状態のサマリを保持する。

- FSR.es を IA-64 コードによって読み取ると、返される値は , FSR ビットに入っているマスクされていない未処理の例外 IE、DE、ZE、OE、UE、PE、および SF のサマリである。注意：ES ビットと FSR の中の他の未処理の例外ビットの間に整合性がない場合には、ES ビットを読み取っても最後に書き込まれた値が返されるとは限らない。
- FSR.es が IA-64 コードによって 1 にセットされているときは、FSR のビット IE、DE、ZE、OE、UE、PE および SF に書き込まれている数値例外情報に関わりなく、次の IA-32 浮動小数点命令で IA-32 遅延数値例外が生成される。
- FSR.es が FSR ビット (IE、DE、ZE、OE、UE、PE、および SF) との関係で整合性がない状態で書き込まれた場合は、その後の数値例外によって整合性がない浮動小数点ステータス・ビットを報告することがある。

数値例外を生成し、正確に報告するためには、FSR、FDR および FIR がコンテクスト・スイッチの前後で同じ値を保持しなければならない。

図 6-9. 浮動小数点データ・レジスタ (FDR)

31–0
オペランド・オフセット (fea)

63–48	47–32
予約 (0 にセットされる)	オペランド・セレクタ (fds)

図 6-10. 浮動小数点命令レジスタ (FIR)

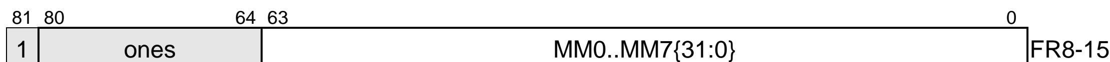
31–0
コード・オフセット (fip)

63–59	58–48	47–32
予約	オペコード {10:0} (fop)	コード・セレクタ (fcs)

6.2.6 IA-32 MMX® テクノロジ・レジスタ

8 つの IA-32 MMX テクノロジ・レジスタが 8 つの IA-64 浮動小数点レジスタ FR8 ~ FR15 にマップされる。このとき、MM0 が FR8 にマップされ、MM7 が FR15 にマップされる。MMX テクノロジ・レジスタの IA-32 浮動小数点スタック・ビューへのマッピングは、IA-32 スタックのトップの浮動小数点値によって異なる。

図 6-11. IA-32 MMX® テクノロジ・レジスタ (MM0 ～ MM7)



- IA-32 MMX テクノロジ命令によって MMX テクノロジ・レジスタに値を書き込むと、下記のように処理される。
  — 対応する浮動小数点レジスタの指数フィールド (ビット 80 ～ 64) とその符号ビット (ビット 81) がすべて 1 に設定される。
  — 仮数部 (ビット 63 ～ 0) は、MMX テクノロジのデータ値に設定される。
- IA-32 MMX テクノロジ命令によって MMX テクノロジ・レジスタの値を読み取ると、下記のように処理される。
  — 対応する浮動小数点レジスタの指数フィールド (ビット 80 ～ 64) とその符号ビット (ビット 81) は、NaTVal エンコーディングを含めて、無視される。

このマッピングの結果、IA-32 または IA-64 の浮動小数点命令によって書き込まれた浮動小数点値の仮数部が IA-32 MMX テクノロジ・レジスタにも表れる。IA-32 MMX テクノロジ・レジスタはまた、8 つのマッピングされた浮動小数点レジスタの仮数フィールドの 1 つにも表れる。

パフォーマンスの低下を避けるため、ソフトウェアのプログラミングの際に、IA-32 浮動小数点命令と IA-32 MMX テクノロジ命令を混用しないことを強く推奨する。MMX テクノロジのコーディングの詳細については『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照のこと。

6.2.7 IA-32 ストリーミング SIMD 拡張命令レジスタ

8 つの 128 ビット IA-32 ストリーミング SIMD 拡張命令レジスタ (XMM0 ～ 7) が 16 の IA-64 浮動小数点レジスタ・ペア FR16 ～ FR31 にマップされる。このとき、XMM0 の下位 64 ビットが FR16{63:0} にマップされ、XMM0 の上位 64 ビットが FR17{63:0} にマップされる。

図 6-12. ストリーミング SIMD 拡張命令レジスタ (XMM0 ～ XMM7)

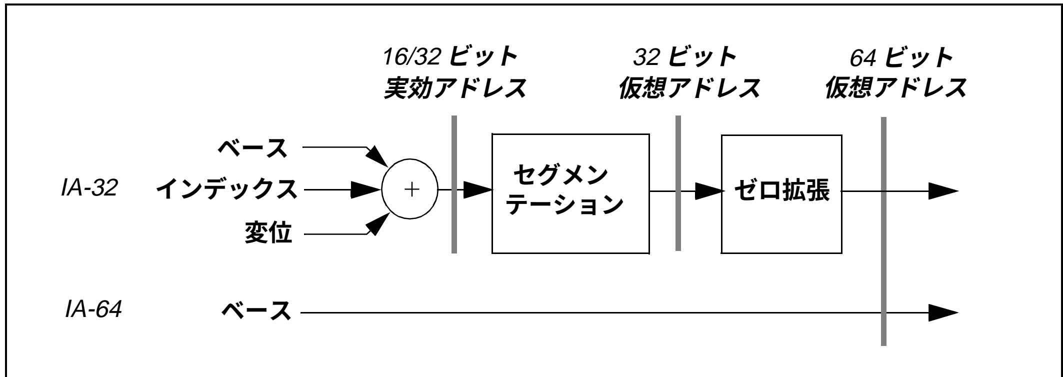
81	80 … 64	63 … 0	
0	0x1003E	XMM0-7{127:64}	FR17-31, 奇数
0	0x1003E	XMM0-7{63:0}	FR16-30, 偶数

- IA-32 ストリーミング SIMD 拡張命令によってストリーミング SIMD 拡張命令レジスタに値を書き込むと、下記のように処理される。
  - — 対応する IA-64 浮動小数点レジスタの指数フィールド (ビット 80 ～ 64) が 0x1003E に設定され、その符号ビット (ビット 81) が 0 に設定される。
  - — 仮数部 (ビット 63 ～ 0) は、偶数レジスタは XMM テクノロジのデータ値ビット {63:0} に設定され、奇数レジスタはビット {127:64} に設定される。
- A-32 ストリーミング SIMD 拡張命令によってストリーミング SIMD 拡張命令レジスタを読み取ると、下記のように処理される。
  - — 対応する IA-64 浮動小数点レジスタの指数フィールド (ビット 80 ～ 64) とその符号ビット (ビット 81) は、NaTVal エンコードを含めて、無視される。

6.3 メモリ・モデルの概要

IA-64 または IA-32 の命令セット内の仮想アドレスは、同じ物理的メモリ位置をアドレス指定するように定義されている。IA-64 命令は直接に 64 ビット仮想アドレスを生成する。IA-32 命令は 16 ビットまたは 32 ビットの実効アドレスを生成し、次にそれを 64 ビットにゼロ拡張することによって 64 ビット仮想アドレスに変換する。ゼロ拡張では、すべての IA-32 でのメモリ参照が 64 ビット仮想アドレス空間の下位 4G バイトに置かれる。どちらかの命令セットによって生成された仮想アドレスは次に、IA-64 のメモリ管理メカニズムを使って物理的アドレスに変換される。

図 6-13. メモリ・アドレス指定モデル



6.3.1 メモリ・エンディアン

メモリの整数および浮動小数点 (IEEE) のデータ型は、IA-32 命令セットと IA-64 命令セットの間でバイナリ互換性がある。IA-32 コードと相互動作する IA-64 アプリケーションおよびオペレーティング・システムでは、メモリ形式を同じにするために、リトル・エンディアン型のアクセスを使用しなければならない。すべての IA-32 における命令データおよび命令メモリの参照は強制的にリトル・エンディアン型として処理される。

6.3.2 IA-32 のセグメンテーション

IA-64 命令セットでのメモリ参照ではセグメンテーションは使用しない。セグメンテーションは、IA-32 命令セットでのメモリ参照の際に、EFLAG.vm および CFLG.pe の状態に基づいて実行される。『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』で定義されているリアル・モード、VM86、または保護モードのセグメンテーション規則に従う。特に下記の規則に従うこと。

- **IA-32 データの 16/32 ビット実効アドレス：**16 または 32 ビット実効アドレスは、CSD.d、SSD.b およびプリフィックスのオーバーライドに基づいて、ベース・レジスタ、スケールされたインデックス・レジスタ、および 16/32 ビット変位値を加算することによって生成される。開始実効アドレス (複数バイト・オペランドの最初のバイト) が 16 または 32 ビットより大きい場合は、16 または 32 ビットに切り捨てられる。最後の 16 ビット実効アドレス (複数バイト・オペランドの最後のバイト) は 64K バイト境界を超えて延長してもよいが、最後の 32 ビット実効アドレスは 32 ビットに切り捨てられ、4G バイトの実行アドレス境界を超えることはない。ラップ条件の詳細については、『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル』を参照のこと。
- **IA-32 コードの 16/32 ビット実効アドレス：**CSD.d に基づいた 16 または 32 ビット EIP が実効アドレスとして使用される。開始 EIP 値 (複数バイト命令の最初のバイト) が 16 または 32 ビットより大きい場合は、16 または 32 ビットに切り捨てられる。最後の 16 ビット実効アドレス (複数バ

イト命令の最後のバイト) は 64K バイト境界を超えて延長してもよいが、最後の 32 ビット EIP 値は 32 ビットに切り捨てられ、4G バイトの実行アドレス境界を超えることはない。

- **IA-32 における 32 ビット仮想アドレスの生成：**生成された 16 または 32 ビット実効アドレスは、セグメント・ベースを加算することによって 32 ビット仮想アドレス空間にマップされる。『*インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル*』で指定されている方法で完全なセグメント保護および制限チェックが検証され、この章で前述した追加的チェックが行われる。32 ビット開始仮想アドレスは、セグメント・ベースが加算された後で 32 ビットに切り捨てられる。最後の仮想アドレス (複数バイトのオペランドまたは命令の最後のバイト) は 4G バイト仮想境界で切り捨てられ、ラップされる。
- **IA-32 における 64 ビット・アドレスの生成：**生成された 32 ビット仮想アドレスは、64 ビットにゼロ拡張することによって 64 ビット仮想アドレスに変換される。これによって、すべての IA-32 命令セットによるメモリ参照が 64 ビット仮想アドレス空間の最初の 4G バイトの中に入る。

IA-32 コードがフラットにセグメント化されたコードを使用している (セグメント・ベースが 0 に設定されている) 場合、IA-32 および IA-64 のコードでは、ポインタが 64 ビットにゼロ拡張された後、自由にポインタを交換できる。セグメント化された IA-32 コードでは、実効アドレス・ポインタを IA-64 コードと共有するためには、あらかじめ仮想アドレスに変換しておかなければならない。

6.3.3 自己修正コード

IA-32 命令セットでの動作時に、すべての IA-32 プログラムに自己修正コードおよび命令キャッシュ・コヒーレンシ (ローカル・プロセッサのデータ・キャッシュに対するコヒーレンシ) がサポートされる。自動修正コードの検出は、Pentium プロセッサと同じ互換レベルで直接にサポートされている。更新された命令バイトが認識されるためには、ソフトウェアによって、ストア操作と修正された命令の間に IA-32 の分岐命令を挿入しなければならない。

IA-64 から IA-32 へ命令セットを切り替える場合、および IA-64 命令を実行しているときには、自己修正コードおよび命令キャッシュ・コヒーレンシはプロセッサ・ハードウェアによって直接にはサポートされない。とくに、IA-64 命令によって IA-32 命令の変更が行われた場合は、IA-64 コードは明示的に命令キャッシュを 4-29 ページの 4.4.6.2 項「メモリの整合性」で定義されているコード・シーケンスと同期化しなければならない。そうしないと、その後の IA-32 命令にその変更が反映されないことがある。

IA-32 から IA-64 へ命令セットを切り替える場合には、IA-32 命令によるローカル命令キャッシュの内容の変更がプロセッサ・ハードウェアによって検出される。プロセッサにより、命令キャッシュがその変更とのコヒーレンシを保ち、後続のすべての IA-64 命令のフェッチでこの変更が認識されることが保証される。

6.4 IA-32 による IA-64 レジスタの使用法

この節では、IA-64 の汎用レジスタおよび浮動小数点レジスタについてのソフトウェアに関連する考慮事項、および IA-32 と相互動作するときの ALAT を示す。

6.4.1 IA-64 レジスタ・スタック・エンジン

`br.ia` 命令または `rfi` 命令によって IA-32 命令を開始する前に、`flushrs` 命令を使ってレジスタ・スタックの中のすべてのダーティなレジスタがバッキング・ストアへフラッシュされていることをソフトウェアによって確実にしなければならない。現在または以前のレジスタ・スタック・フレームに残っているダーティなレジスタはすべて変更される。レジスタ・スタックについての詳細は、4-1 ページの 4.1 項「レジスタ・スタック」を参照のこと。

IA-32 命令セットの実行に移行した後は、RSE コントロール・レジスタのイネーブル条件に関わりなく、RSE は実質的にディセーブルになる。

`jmpe` 命令または割り込みによって IA-32 命令セットが終了した後、すべてのスタックされたレジスタは無効とマークされ、クリーンなレジスタの数は 0 に設定される。

6.4.2 IA-64 ALAT

IA-32 命令セットを実行したときには、ALAT の内容は未定義である。ソフトウェア上では、命令セットの移行の前後で ALAT の値が保持されることを想定してはならない。IA-32 コードに移行すると、ALAT の中の既存のエントリは無視される。ALAT についての詳細は、4-21 ページの 4.4.5.2 項「データ・スペキュレーションと命令」を参照のこと。

6.4.3 IA-32 命令での IA-64 NaT/NaTVal の応答

IA-32 命令セットに移行する前に、IA-64 コードが整数レジスタの NaT 条件、または浮動小数点レジスタ、MMX テクノロジ・レジスタまたはストリーミング SIMD 拡張命令レジスタの NaTVal 条件を設定している場合には、以下の条件が発生する可能性がある。

- 伝播された NaT が入っているレジスタを直接または間接的に使用する IA-32 依存の命令では、NaT を伝播するか、または NaT Register Consumption (Nat レジスタ消費) アボートとなる。プロセッサが NaT を伝播するか、NaT Register Consumption アボートとなるかはモデルによって異なる。IA-32 の実行中に NaT Register Consumption アボートが発生した場合、IA-32 命令が実行の途中で、一部のアーキテクチャ上の状態がすでに変更された状態で打ち切られることがある。プロセッサが NaT を含む入力レジスタの使用を許可する場合には、必ず、NaT 値を伝播するか、NaT Register Consumption フォルトを生成する。

- IA-64 浮動小数点の NaTVal 値を IA-32 浮動小数点命令に伝播してはならない。そうしないと、プロセッサの動作はモデル固有になり、定義されない。プロセッサは、フォルトを発生せずに NaTVal を含む浮動小数点レジスタを SNAN に変換することがある (この変換は IA-32 命令セットへの移行時、または IA-32 浮動小数点命令による使用時に行われる)。IA-32 命令セットによって、伝播された NaTVal 値を直接または間接的に使用すると、NaTVal 値を伝播するか、あるいは IA-32_Exception (FPError Invalid Operand) フォルトを生成する。プロセッサがフォルトを生成するか、NaTVal 値を伝播するかはモデルによって異なる。プロセッサが NaTVal が格納されているレジスタの使用を許可する場合には、必ず、NaTVal 値を伝播するか、IA-32_Exception (FPError Invalid Operand) フォルトを生成する。

  注：IA-32 コードで IA-32 浮動小数点ロード命令を使ってメモリ位置から NaTVal 値を読み取ることはできない。なぜなら、NaTVal 値は 80 ビット拡張倍精度値では表現できないからである。NaTVal および IA-64 デノーマル数の数値に関する問題を避けるために、標準的な IA-32 呼び出し規則を使って浮動小数点値をメモリ・スタックに渡すことを強く推奨する。

- NaTVal エンコーディングを含むレジスタを直接または間接的に使用する IA-32 ストリーミング SIMD 拡張命令では、NaTVal エンコーディングを無視し、レジスタの仮数フィールドを有効なデータ値と解釈する。

- NaTVal エンコーディングを含むレジスタを直接または間接的に使用する IA-32 MMX テクノロジ命令では、NaTVal エンコーディングを無視し、レジスタの仮数フィールドを有効なデータ値と解釈する。

ソフトウェア上で、IA-32 命令の実行中の NaT または NaTVal の動作に依存したり、NaT または NaTVal を IA-32 命令に伝播してはならない。

JMPE - IA-64 命令セットへのジャンプ

オペコード	命令	説明
0F 00 /6	JMPE *r/m16*	IA-64 **命令へのジャンプ、**r/m16 **によって指定される間接アドレス**
0F 00 /6	JMPE *r/m32*	A-64 **命令へのジャンプ、**r/m32 **によって指定される間接アドレス**
0F B8	JMPE *disp16*	IA-64 **命令へのジャンプ、**addr16 **によって指定される絶対アドレス**
0F B8	JMPE *disp32*	IA-64 **命令へのジャンプ、**addr32 **によって指定される絶対アドレス**

説明

この命令は、IA-64 システム環境の IA-64 プロセッサでのみ使用できる。

JMPE 命令はプロセッサを IA-64 命令セットに切り替え、指定されたターゲット・アドレスから実行を開始する。間接アドレス指定形式 r/m*r16/32* と符号なしの絶対アドレス指定形式 *disp16/32* の 2 つの方法がある。16 ビットと 32 ビットの両方の形式がサポートされている。

絶対アドレス指定形式では、IA-64 命令セットの中の 16 バイトにアラインメントされた 64 ビット仮想ターゲット・アドレスを、現在の CS ベースに符号なしの 16 または 32 ビット変位を加算することによって算出する (*IP{31:0}=disp16/32 + CSD.base*)。間接アドレス指定では、IA-64 仮想ターゲット・アドレスをレジスタの内容またはメモリ位置によって指定する (*IP{31:0}= [r/m16/32] + CSD.base*)

GR[1] には、JMPE 命令の次の順次命令アドレスがロードされる。

JMPE 命令は FWAIT オペレーションを実行する。マスクされていない IA-32 浮動小数点例外がすべて、JMPE 命令でフォルトとして報告される。

JMPE 命令はメモリ・フェンスあるいはシリアル化オペレーションを実行しない。

JMPE 命令が正常に実行されると、EFLAG.rf が 0 にクリアされる。

eager での実行のために IA-64 のレジスタ・スタック・エンジンがイネーブルにされている場合、レジスタ・スタック・エンジンは、プロセッサが IA-64 命令セットの実行に移行するとただちにレジスタのロードを開始する。

intel®

IA-64 命令リファレンス 7

本章では、IA-64 命令の機能を、アセンブリ言語ニーモニックのアルファベット順に説明する。

7.1 命令リファレンス・ページに関する規則

命令リファレンス・ページは、表 7-1 に示すように 4 つの項目に分かれている。最初の 3 つの項目はすべての命令リファレンス・ページにあり、最後の 1 項目は必要な場合に限られる。表 7-2 に、命令リファレンス・ページに使用されている印刷字体に関する規則を示す。

表 7-1. 命令リファレンス・ページの説明

項名	内容
書式	アセンブリ言語シンタックス、命令のタイプ、およびエンコーディング書式
説明	命令の機能説明
操作	C コード表記による命令の機能説明
FP 例外	IEEE 浮動小数点トラップ

表 7-2. 命令リファレンス・ページの字体に関する規則

字体	意味
普通	(書式の項) アセンブリ言語ニーモニックの必須文字
イタリック	(書式の項) 説明の項に示されている範囲内の有効な値で置き換えるべきアセンブリ言語フィールド名
`code`	(操作の項) 命令の動作を表す C コード
`code_italic`	(操作の項) 書式の項に示されているイタリック表記フィールドに対応するアセンブリ言語フィールド名

書式の項では、レジスタ・アドレスは、表 7-3 の第 3 欄に示されているアセンブリ・ニーモニック・フィールド名を使用して指定する。プレディケートを定義する命令については、説明の項では、修飾プレディケートが偽であるときにアーキテクチャ上の状態を変更する命令以外は、修飾プレディケートは真であるものとみなしている。該当する場合、修飾プレディケートの判定は操作の項に記載されている。

操作の項では、`reg[`*`addr`*`].field` という表記を使用してレジスタのアドレスを指定する。アクセスされるレジスタ・ファイルを *`reg`* に指定し、その値は表 7-3 の第 2 欄から選択する。`addr` フィールドには、アセンブリ言語フィールド名またはレジスタ・ニーモニックでレジスタ・アドレスを指定する。レジスタのリネームが行われる汎用

レジスタ、浮動小数点レジスタ、およびプレディケート・レジスタの各ファイルについては、*addr* はリネームされる前のレジスタ・アドレスであり、リネームは示していない。`field` オプションには、レジスタ内で命名されているビット・フィールドを指定する。`field` がない場合は、レジスタのすべてのフィールドがアクセスされる。唯一の例外は、`GR[addr]` 表記が使用される場合の、汎用レジスタのデータ・フィールド (NaT ビットを除く 64 ビット) を参照する場合である。操作の項に示されているコードと標準 C とのシンタックス上の相違を表 7-4 に示す。

表 7-3. レジスタ・ファイルの表記法

レジスタ・ファイル	C 表記	アセンブリ・ニーモニック	間接アクセス
アプリケーション・レジスタ	AR	ar	
分岐レジスタ	BR	b	
CPU 識別レジスタ	CPUID	cpuid	Y
浮動小数点レジスタ	FR	f	
汎用レジスタ	GR	r	
パフォーマンス・モニタ・データ・レジスタ	PMD	pmd	Y
プレディケート・レジスタ	PR	p	

表 7-4. C シンタックスの相違点

シンタックス	機能
{msb:lsb}, {bit}	ビット・フィールド指定子。これが変数に付加されると、"msb" で指定される最上位ビットから "lsb" で指定される最下位ビットまでの、"msb" および "lsb" ビットを含むビット・フィールドを表す。"msb" と "lsb" とが等しい場合は、単一ビットがアクセスされる。2 番目の形式は単一ビットを表す。
u>, u>=, u<, u<=	符号なし不等関係。演算子のどちら側の変数も符号なしとして扱われる。
u>>, u>>=	符号なし右シフト。最下位ビット位置からゼロがシフト・インされる。
u+	符号なし加算。両オペランドは符号なしとして扱われ、ゼロ拡張される。
u*	符号なし乗算。両オペランドは符号なしとして扱われる。

7.2 命令の説明

本章の以降では、IA-64 命令を説明する。

加算 (Add)

書式:

(qp) add $r_1 = r_2, r_3$	register_form	A1
(qp) add $r_1 = r_2, r_3$, 1	plus1_form, register_form	A1
(qp) add $r_1 = imm, r_3$	pseudo-op	
(qp) adds $r_1 = imm_{14}, r_3$	imm14_form	A4
(qp) addl $r_1 = imm_{22}, r_3$	imm22_form	A5

説明: 2 つのソース・オペランド (および任意指定の定数 1) が加算され、その結果が GRr_1 に格納される。register_form では、第 1 オペランドは GR r_2 であり、imm14_form では、第 1 オペランドは符号拡張された imm14 エンコーディング・フィールドで与えられ、imm22_form では、第 1 オペランドは符号拡張された imm_{22} エンコーディング・フィールドで与えられる。imm22_form では、GRr_3 は GR 0、1、2、および 3 しか指定できない。

plus1_form は register_form でのみ使用可能である (ただし、即値を調整することにより、即値形式でも同じ結果が得られる)。

即値形式の擬似オペコードでは、即値オペランドと GRr_3 の値の大きさに基づいて、imm14_form または imm22_form を選択する。

操作:

```
if (PR[qp]) {
    check_target_register(r1);

    if (register_form)                                  // register form
        tmp_src = GR[r2];
    else if (imm14_form)                                // 14-bit immediate form
        tmp_src = sign_ext(imm14, 14);
    else                                                // 22-bit immediate form
        tmp_src = sign_ext(imm22, 22);

    tmp_nat = (register_form ? GR[r2].nat : 0);

    if (plus1_form)
        GR[r1] = tmp_src + GR[r3] + 1;
    else
        GR[r1] = tmp_src + GR[r3];

    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

ポインタの加算 (Add Pointer)

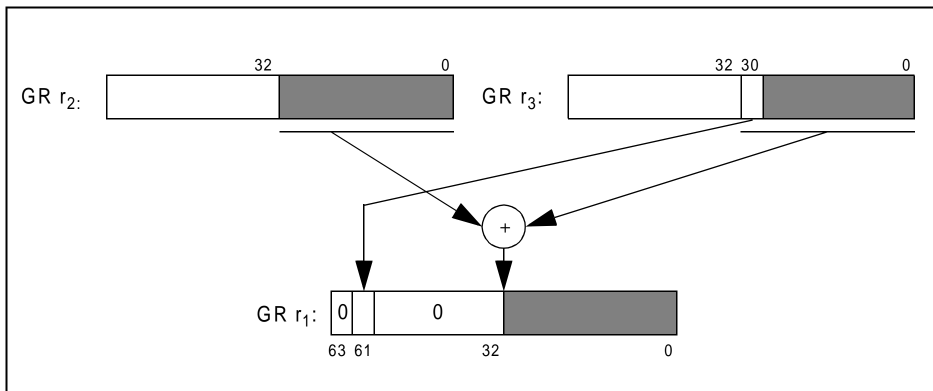
書式 :

(*qp*) addp4 $r_1 = r_2, r_3$ register_form A1
(*qp*) addp4 $r_1 = imm_{14}, r_3$ imm14_form A4

説明 :

2 つのソース・オペランドが加算され、その結果の上位 32 ビットがゼロ・クリアされ、次に GR r_3 のビット [31:30] が結果のビット [62:61] にコピーされる。この結果は GR r_1 に格納される。register_form では、第 1 オペランドは GR r_2 であり、imm14_form では、第 1 オペランドは符号拡張された imm_{14} エンコーディング・フィールドで与えられる。

図 7-1. ポインタの加算



操作 :

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm14, 14));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    tmp_res = tmp_src + GR[r3];
    tmp_res = zero_ext(tmp_res{31:0}, 32);
    tmp_res{62:61} = GR[r3]{31:30};
    GR[r1] = tmp_res;
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

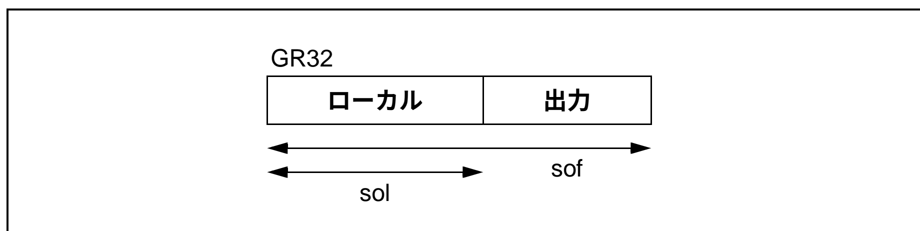
スタック・フレームの割り当て (Allocate Stack Frame)

書式: alloc r_1 = ar.pfs, i, l, o, r M34

説明: 説明 : 汎用レジスタ・スタック上に新しいスタック・フレームが割り当てられ、以前のファンクションの状態 (PFS) レジスタが GR r_1 にコピーされる。フレーム・サイズの変更値は即値で指定する。GR r_1 への書き込みと、同じ命令グループ内の以降の命令では、新しいフレームを使用する。この命令ではプレディケートは使用できない。

4 つのパラメータ i (入力領域のサイズ)、l (ローカル領域のサイズ)、o (出力領域のサイズ)、および r (ローテート領域のサイズ) でスタック・フレームの各領域のサイズを指定する。

図 7-2. スタック・フレーム



フレームのサイズ (sof) は $i + l + o$ によって決定される。この命令は、現在のレジスタ・スタック・フレームのサイズを増減させることがあるので注意されたい。ローカル領域のサイズ (sol) は $i + l$ によって与えられる。入力領域とローカル領域との間には実際的な相違はない。これらは、ローカル・レジスタがどのように使用されるかについてのヒントをアセンブラに示すために別々のオペランドとして指定できる。

ローテートするレジスタはスタック・フレーム内に収まらなければならず、それらの数は 8 の倍数でなければならない。この命令で CFM.sor のサイズを変更しようとしたときに、レジスタ・リネーム・ベース・レジスタ (CFM.rrb.gr、CFM.rrb.fr、CFM.rrb.pr) のすべてがゼロでなかった場合は、予約レジスタ / フィールド (Reserved Register/Field) フォルトが発生する。

アセンブラでは alloc の無効なオペランドの組み合わせを許可していないが、命令に無効な組み合わせのエンコーディングを使用することは可能である。96 レジスタを超えるスタック・フレームを割り当てようとしたり、スタック・フレームより大きなサイズのローテート領域やスタック・フレームより大きなサイズのローカル領域を割り当てようとしたりすると、無効操作 (Illegal Operation) フォルトが発生する。`alloc` 命令は、同一命令グループ内の最初の命令でなければならない。そうでない場合は、結果は不定になる。

使用可能なレジスタが足りないために必要なフレームが割り当てられない場合は、ダーティなレジスタが必要なだけバッキング・ストアに書き込まれるまで、`alloc` はプロセッサをストールさせる。このような RSE による強制的な格納によって、下に示すようなデータ関連のフォルトが発生することがある。

操作：

```
tmp_sof = i + l + o;
tmp_sol = i + l;
tmp_sor = r u>> 3;
check_target_register_sof(r1, tmp_sof);
if (tmp_sof u> 96 || r u> tmp_sof || tmp_sol u> tmp_sof)
    illegal_operation_fault();
if (tmp_sor != CFM.sor &&
                (CFM.rrb.gr != 0 || CFM.rrb.fr != 0 || CFM.rrb.pr != 0))
    reserved_register_field_fault();

alat_frame_update(0, tmp_sof - CFM.sof);
rse_new_frame(CFM.sof, tmp_sof);// Make room for new registers; Mandatory RSE
                                // stores can raise faults listed below.
CFM.sof = tmp_sof;
CFM.sol = tmp_sol;
CFM.sor = tmp_sor;

GR[r1] = AR[PFS];
GR[r1].nat = 0;
```

論理積 (Logical And)

書式:

(qp) and $r_1 = r_2, r_3$	register_form	A1
(qp) and $r_1 = imm_8, r_3$	imm8_form	A3

説明: 2 つのソース・オペランド間の論理積が取られ、その結果が GR r_1 に格納される。register_form では、第 1 オペランドは GR r_2 であり、imm8_form では、第 1 オペランドは imm_8 エンコーディング・フィールドで与えられる。

操作:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src & GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

補数の論理積 (And Complement)

書式: (qp) andcm $r_1 = r_2, r_3$ register_form A1
(qp) andcm $r_1 = imm_8, r_3$ imm8_form A3

説明: 第 1 ソース・オペランドと、第 2 ソース・オペランドの 1 の補数との間の論理積が取られ、その結果が GR r_1 に格納される。register_form では、第 1 オペランドは GR r_2 であり、imm8_form では、第 1 オペランドは imm_8 エンコーディング・フィールドで与えられる

操作:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src & ~GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

分岐 (Branch)

書式 :

(*qp*) br.*btype*.*bwh*.*ph*.*dh* $target_{25}$	ip_relative_form	B1
(*qp*) br.*btype*.*bwh*.*ph*.*dh* $b_1 = target_{25}$	call_form, ip_relative_form	B3
br.*btype*.*bwh*.*ph*.*dh* $target_{25}$	counted_form, ip_relative_form	B2
br.*ph*.*dh* $target_{25}$	pseudo-op	
(*qp*) br.*btype*.*bwh*.*ph*.*dh* b_2	indirect_form	B4
(*qp*) br.*btype*.*bwh*.*ph*.*dh* $b_1 = b_2$	call_form, indirect_form	B5
br.*ph*.*dh* b_2	pseudo-op	

説明 :

説明 : 分岐演算が評価され、その結果により、分岐が発生するか、あるいはシーケンス内の次の命令からプログラムの実行が継続される。分岐の実行は、論理的には、同じ命令グループ内の分岐命令より前のすべての非分岐命令が実行された後に行われる。分岐が発生すると、プログラムの実行はスロット 0 から開始される。

分岐のタイプとしては IP (命令ポインタ) 相対分岐と間接分岐がある。IP 相対分岐の場合は、アセンブリでは、$target_{25}$ オペランドに分岐先のラベルを指定する。これで、この命令を含むバンドルから分岐先バンドルまでの変位が計算されて、符号付き即値 (imm_{21}) として分岐命令にエンコードされる ($imm_{21} = target_{25}$ - IP >> 4)。間接分岐の場合は、分岐先アドレスは BR b_2 で与えられる。

表 7-5. 分岐のタイプ

btype	機能	分岐条件	分岐先アドレス
cond or *none*	条件付き分岐	修飾プレディケート	IP 相対または間接
call	条件付きプロシージャ・コール	修飾プレディケート	IP 相対または間接
ret	条件付きプロシージャ・リターン	修飾プレディケート	間接
ia	IA-32 命令セットの呼び出し	無条件	間接
cloop	カウント指定ループの分岐	ループ・カウント	IP 相対
ctop, cexit	モジュロ・スケジュール型のカウント指定ループ	ループ・カウントおよびエピローグ・カウント	IP 相対
wtop, wexit	モジュロ・スケジュール型の while ループ	修飾プレディケートおよびエピローグ・カウント	IP 相対

無条件分岐には 2 つの擬似オペコードがある。これらのオペコードは条件付き分岐 (*btype* = cond) のようにエンコードされ、*qp* フィールドには PR 0 を指定し、*bwh* ヒントには *sptk* を指定する。

分岐のタイプによって、分岐条件がどのように計算されるか、また (リンク・レジスタへの書き込みなどの) 分岐による他の効果があるかどうかが決まる。基本的分岐タイプの場合は、分岐条件は単に指定されたプレディケート・レジスタの値だけである。基本分岐タイプを以下に示す。

- cond: 修飾プレディケートが 1 の場合に分岐が発生する。1 でない場合は、発生しない。

- call: 修飾プレディケートが 1 の場合に分岐が発生し、さらに以下の処置がなされる。
  - — 現在のフレーム・マーカ (CFM)、EC アプリケーション・レジスタ、および現在の特権レベルの現在値が以前のファンクションの状態 (PFS) アプリケーション・レジスタにセーブされる。
  - — 呼び出し元のスタック・フレームが結果的にセーブされ、呼び出し元の出力領域だけが入ったフレームが呼ばれた側に渡される。
  - — CFM 内のローテーション用のリネーム・ベース・レジスタが 0 にリセットされる。
  - — RB b_1 にリターン・リンク値が格納される。
- return: 修飾プレディケートが 1 の場合に分岐が発生し、さらに以下の処置がなされる。
  - — CFM、EC、および現在の特権レベルが PFS から復元される。(特権レベルは、復元することによって特権レベルが大きくならない場合に限り復元される。)
  - — 呼び出し元のスタック・フレームが復元される。
  - — リターンによって特権レベルが下がり、かつ PSR.lp が 1 の場合は、Lower-previlege Transfer (下位特権遷移) トラップが発生する。
- ia: OS によってインターセプトされなくても、無条件に分岐が発生する。この分岐によって、(PSR.is を 1 にセットすることにより) IA-32 命令セットを呼び出して、BR b_2\{31:0\} 内の仮想リニア分岐先アドレスから IA-32 命令の処理を開始する。修飾プレディケートが PR 0 でない場合は、無効操作フォルトが発生する。

  IA-32 における分岐先の実効アドレスは、現在のコード・セグメントと相対的に、つまり、EIP\{31:0\}= BR b_2\{31:0\}- CSD.base として計算される。命令セットの移行がディスエーブルにされていなければ、IA-32 命令セットには任意の特権レベルで移行できる。命令セットの移行中には、レジスタ・バンクの切り換えも特権レベルの変更も生じない。

  ソフトウェア上では、分岐を発行する前に、必ずコード・セグメント・ディスクリプタ (CSD) およびコード・セグメント・セレクタ (CS) がロードされるようにしなければならない。分岐先の EIP 値がコード・セグメント制限を超えたり、その値にコード・セグメント特権違反があった場合は、分岐先の IA-32 命令で IA-32_Exception(GPFault) が発生する。16 ビットの IA-32 コードに移行する場合は、BR b_2 が CSD.base の 64K バイト以内でなければ、分岐先命令で GPFault が発生する。EFLAG.rf は、最初の IA-32 命令が正常に終了するまでは変更されない。EFLAG.rf は、分岐先の IA-32 命令が正常に終了するまではクリアされない。

  IA-32 でのプロセッサによる整合性のあるメモリ参照と IA-64 での順序付けられていないメモリ参照との間でメモリの順序付けが必要な場合は、ソフトウェアによって分岐の前に `mf` 命令を発行しなければならない。プロセッサは、IA-64 命令セットによって生じる命令ストリームへの書き込みが、後続の IA-32 命令フェッチから検出できるかどうかは保証していない。`br.ia` 命令では、命令のシリアル化操作は行わない。プロセッサは、(同一命令グループ内でも) 前に行われた GR や FR への書き込みが最初の IA-32 命令から検出できるかどうかは保証していない。`br.ia` 命

令ではすべての AR を暗黙的に読み込むことがあるので、br.ia 命令と同じ命令グループ内での AR への書き込みは許可されない。AR への書き込みと br.ia 命令との間に無効な RAW の依存関係が存在する場合は、最初の IA-32 命令のフェッチと実行の段階では、更新後の AR 値がわかることもわからないこともある。

IA-32 命令セットを実行すると、ALAT の内容は未定義のままである。ソフトウェア上では、命令セットの移行があった場合は、ALAT の値が保持されるものとみなしてはならない。IA-32 コードに移行すると、ALAT 内の既存エントリは無視される。レジスタ・スタック内にダーティなレジスタが存在する場合は、br.ia 命令で無効操作フォルトが発生する。現在のレジスタ・スタック・フレームに残っていたレジスタはすべて、IA-32 命令セットの実行中は未定義のままである。現在のレジスタ・スタック・フレームはゼロ・クリアされる。ダーティなレジスタのレジスタ・ファイルをフラッシュするには、命令グループ内で br.ia 命令より前に flushrs 命令を発行しなければならない。命令セットの移行のパフォーマンスを改善するには、ソフトウェア上では、1) flushrs 命令を必ず br.ia 命令の前の 1 つだけの命令グループに入れ、2) br.ia 命令を必ず最初の B スロットに入れて、IA-32 命令セットを起動するのと並行して IA-64 レジスタ・スタックのフラッシュを起動するとよい。br.ia 命令は、常に、" 静的に処理する " (デフォルト) のヒントを付けて最初の B スロットで実行するようにする。そうしないと、プロセッサのパフォーマンスが低下する。

もう 1 つの分岐タイプが、単純なカウント指定ループ向けに用意されている。この分岐タイプでは、ループ・カウント (LC) アプリケーション・レジスタを使用して分岐条件を判定し、修飾プレディケートは使用しない。

- cloop: LC レジスタがゼロでない場合は、この値がデクリメントされ、分岐が発生する。

これらの単純な分岐タイプの他に、モジュロ・スケジュール型ループを高速に処理するための 4 つのタイプがある。LC レジスタを使用するカウント指定ループ用が 2 つと、修飾プレディケートを使用する while ループ用が 2 つである。これらのループ・タイプでは、レジスタのローテーションを使用してレジスタのリネーミングを実現し、プレディケーションを使用して空のパイプライン・ステージに対応する命令をオフにする。

エピローグ・カウント (EC) アプリケーション・レジスタを使用してエピローグ・ステージがカウントされる。あるいは一部の while ループについては、プロローグ・ステージの一部がカウントされる。エピローグ・フェーズでは、ループが一巡するたびに EC がデクリメントされ、大部分のループでは、EC が 1 になると、パイプラインが空になってループが終了する。特定タイプの、ソフトウェアによってパイプライン化された展開型の最適化ループについては、br.cexit または br.wext の分岐先が次の順序のバンドルに設定される。この場合は、EC が 1 になっても、パイプラインは完全には空になっていないことがあり、EC がゼロであっても引き続きパイプラインから命令が排出される。

これらのモジュロ・スケジュール型ループについては、分岐が発生するかどうかの演算は、カーネル分岐条件 (カウント指定ループに対しては LC、while ループに対しては修飾プレディケート) およびエピローグ条件 (EC が 1 より大きいか、大きくないか) によって決まる。

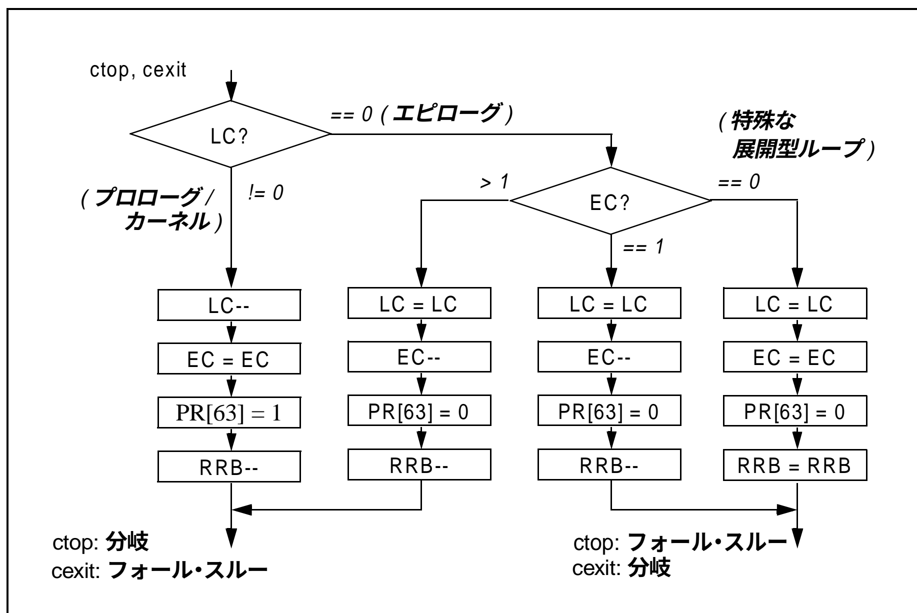
これらの分岐には、"top" と "exit" という 2 つのカテゴリがある。"top" タイプ (ctop および wtop) は、ループの判定がループ本体の一番下にあるときに使用され、したがって、分岐が発生するとループを続行するのに対し、フォール・スルー分岐ではループを終了する。"exit" タイプ (cexit および wexit) は、ループの判定がループの一番下以外のどこかにあるときに使用され、したがって、フォール・スルー分岐ではループを続行するのに対して、分岐が発生するとループを終了する。"exit" タイプは、展開型のパイプライン化されたループ内の中間点にも使用される。

モジュロ・スケジュール型ループのタイプを以下に示す。

- ctop および cexit: これらの分岐タイプの動作は、分岐するかどうかの判定以外は同じである。`br.ctop` については、LC がゼロでないか、EC が 1 より大きい場合に分岐が発生する。`br.cexit` については、その反対があてはまる。つまり、LC がゼロでないか、EC が 1 より大きい場合は分岐は発生せず、そうでない場合は分岐が発生する。

  これらの分岐タイプでは、レジスタのローテーションおよびプレディケートの初期化の制御にも LC と EC を使用する。プロローグ・フェーズおよびカーネル・フェーズ (LC が 0 ではない) では、LC がカウント・ダウンされる。LC が 0 で `br.ctop` または `br.cexit` が実行されるとエピローグ・フェーズになり、EC がカウント・ダウンされる。LC が 0、かつ EC が 1 のときに `br.ctop` または `br.cexit` が実行されると、EC の最後のデクリメント (カウント・ダウン) と最後のレジスタのローテーションが行われる。LC および EC が共にゼロの場合は、レジスタのローテーションが停止する。分岐の判定以外のこれらの効果はどちらの分岐タイプにおいても同じであり、図 7-3 に示すとおりである。

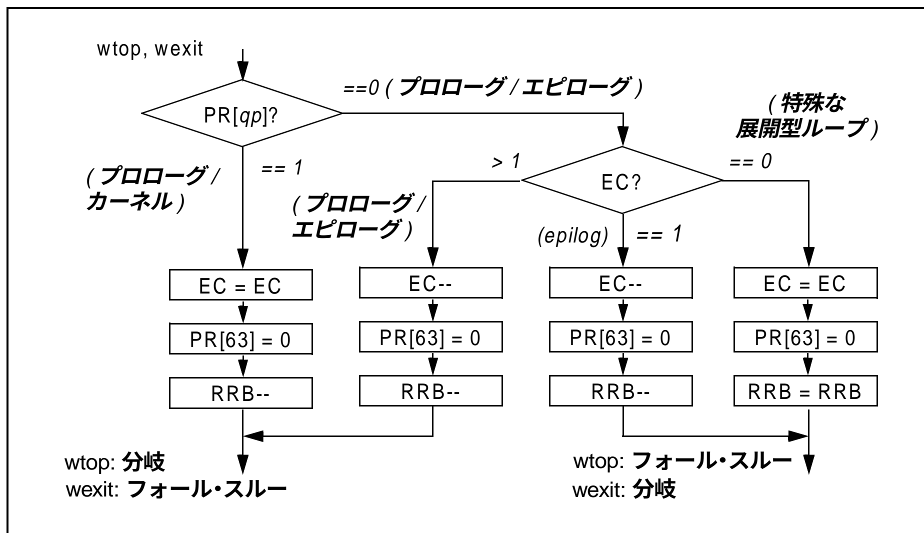
図 7-3. br.ctop および br.cexit の操作



wtop および wexit: これらの分岐タイプの動作は、分岐するかどうかの判定以外は同じである。br.wtop については、修飾プレディケートが 1 であるか、EC が 1 より大きい場合に分岐が発生する。br.cexit については、その反対があてはまる。つまり、修飾プレディケートが 1 であるか、EC が 1 より大きい場合は分岐は発生せず、そうでない場合は分岐が発生する。

これらの分岐タイプでは、レジスタのローテーションおよびプレディケートの初期化の制御にも修飾プレディケートと EC を使用する。プロローグ・フェーズでは、ループのプログラミングに使用される方式に応じて、修飾プレディケートはゼロか 1 である。カーネル・フェーズでは、修飾プレディケートは 1 である。エピローグ・フェーズでは、修飾プレディケートはゼロであり、EC がカウント・ダウンされる。修飾プレディケートがゼロ、かつ EC が 1 のときに br.wtop または br.wexit が実行されると、EC の最後のデクリメント (カウント・ダウン) と最後のレジスタのローテーションが行われる。修飾プレディケートおよび EC が共にゼロの場合は、レジスタのローテーションが停止する。分岐の判定以外のこれらの効果はどちらの分岐タイプにおいても同じであり、図 7-4 に示すとおりである。

図 7-4. br.wtop および br.wexit の操作



ループ・タイプの分岐 (`br.cloop`、`br.ctop`、`br.cexit`、`br.wtop`、および `br.wexit`) は、同一バンドル内の命令スロット 2 でしか実行できない。このような命令をスロット 0 か 1 で実行すると、分岐が発生したかどうかに関わらず、無効操作フォルトが発生する。

分岐命令においては、リード・アフター・ライト (RAW) およびライト・アフター・リード (WAR) の依存関係についての必要条件はわずかに異なる。非分岐命令による BR、PR、および PFS の変更は、同一命令グループ内のそれら以降の分岐命令からは検出されない (つまり、これらのリソースには制限された RAW が許可される)。したがって、例えば、低レイテンシの比較分岐シーケンスは使用できる。RAW に対する通常の必要条件は、LC および EC 両アプリケーション・レジスタや RPB にあてはまる。

読み取り命令や書き込み命令が分岐である場合は、同一の命令グループ内での PR63 に対する WAR の依存関係は許されない。例えば、`br.wtop` または `br.wexit` は、その修飾プレディケートとして PR[63] を使用してはならない。また、PR[63] は、同一命令グループ内の br.wtop または br.wexit より前にある分岐命令の修飾プレディケートであってはならない。

依存関係のために、分岐が発生するかどうかに関わらず、ループ・タイプの分岐は結果的に常にそれぞれが関連付けられているリソースに書き込む。cloop タイプは結果的に常に LC に書き込む。LC が 0 のときは、cloop 分岐は LC を変更しないが、ハードウェアが LC に同じ値を再書き込みする形でその更新を実現できる。同様に、`br.ctop` と `br.cexit` は結果的に常に LC、EC、RRB、および PR[63] に書き込む。`br.wtop` と `br.wexit` は結果的に常に EC、RRB、および PR[63] に書き込む。

各種の分岐ヒントのコンプリータの値を以下の表に示す。分岐有無の予測ストラテジ (Whether Prediction Strategy) ヒントを表 7-6 に、シーケンシャル・プ

リフェッチ (Sequential Prefetch) ヒントを表 7-7 に、また分岐キャッシュ割り当て解除 (Branch Cache Deallocation) ヒントを表 7-8 にそれぞれ示す。

表 7-6. 分岐有無予測ヒント

bwh Completer	分岐有無予測ヒント
spnt	静的に処理されない
sptk	静的に処理する
dpnt	動的に処理されない
dptk	動的に処理する

表 7-7. シーケンシャル・プリフェッチ・ヒント

ph Completer	シーケンシャル・プリフェッチ・ヒント
few もしくは *none*	少数ライン
many	多数ライン

表 7-8. 分岐キャッシュ割り当て解除ヒント

dh Completer	分岐キャッシュ割り当て解除ヒント
none	割り当て解除なし
clr	分岐情報割り当て解除あり

操作 :

```
if (ip_relative_form)                              // determine branch target
    tmp_IP = IP + sign_ext((imm21 << 4), 25);
else // indirect_form
    tmp_IP = BR[b2];

if (btype != 'ia')                                 // for IA-64 branches,
    tmp_IP = tmp_IP & ~0xf;                        //  ignore bottom 4 bits of target

lower_priv_transition = 0;

switch (btype) {
    case 'cond':                                   // simple conditional branch
        tmp_taken = PR[qp];
        break;

    case 'call':                                   // call saves a return link
        tmp_taken = PR[qp];
        if (tmp_taken) {
            BR[b1] = IP + 16;

            AR[PFS].pfm = CFM;                     // ... and saves the stack frame
            AR[PFS].pec = AR[EC];
            AR[PFS].ppl = PSR.cpl;

            alat_frame_update(CFM.sol, 0);
            rse_preserve_frame(CFM.sol);
            CFM.sof -= CFM.sol;                    // new frame size is size of outs
            CFM.sol = 0;
            CFM.sor = 0;
            CFM.rrb.gr = 0;
            CFM.rrb.fr = 0;
            CFM.rrb.pr = 0;
```

```
        }
        break;

    case 'ret':                                  // return restores stack frame
        tmp_taken = PR[qp];
        if (tmp_taken) {
            // tmp_growth indicates the amount to move logical TOP *up*:
            // tmp_growth = sizeof(previous out) - sizeof(current frame)
            // a negative amount indicates a shrinking stack
            tmp_growth = (AR[PFS].pfm.sof - AR[PFS].pfm.sol) - CFM.sof;
            alat_frame_update(-AR[PFS].pfm.sol, 0);
            rse_fatal = rse_restore_frame(AR[PFS].pfm.sol, tmp_growth,
CFM.sof);
            if (rse_fatal) {
                CFM.sof = 0;
                CFM.sol = 0;
                CFM.sor = 0;
                CFM.rrb.gr = 0;
                CFM.rrb.fr = 0;
                CFM.rrb.pr = 0;
            } else // normal branch return
                CFM = AR[PFS].pfm;

            rse_enable_current_frame_load();
            AR[EC] = AR[PFS].pec;
            if (PSR.cpl u< AR[PFS].ppl) {       // ... and restores privilege
                PSR.cpl = AR[PFS].ppl;
                lower_priv_transition = 1;
            }
        }
        break;

    case 'ia':                                   // switch to IA mode
        tmp_taken = 1;
        if (qp != 0)
            illegal_operation_fault();
        if (AR[BSPSTORE] != AR[BSP])
            illegal_operation_fault();
        if (PSR.di)
            disabled_instruction_set_transition_fault();
        PSR.is = 1;                              // set IA-32 Instruction Set Mode
        CFM.sof = 0;                             //force current stack frame
        CFM.sol = 0;                             //to zero
        CFM.sor = 0;
        CFM.rrb.gr = 0;
        CFM.rrb.fr = 0;
        CFM.rrb.pr = 0;
        rse_invalidate_non_current_regs();

    // Note the register stack is disabled during IA-32 instruction set
execution
        break;

    case 'cloop':                                // simple counted loop
        if (slot != 2)
            illegal_operation_fault();
        tmp_taken = (AR[LC] != 0);
        if (AR[LC] != 0)
            AR[LC]--;
        break;

    case 'ctop':
    case 'cexit':                                // SW pipelined counted loop
        if (slot != 2)
            illegal_operation_fault();
        if (btype == 'ctop')    tmp_taken =  ((AR[LC] != 0) || (AR[EC] u> 1));
        if (btype == 'cexit')   tmp_taken = !((AR[LC] != 0) || (AR[EC] u> 1));
        if (AR[LC] != 0) {
            AR[LC]--;
```

```
            AR[EC] = AR[EC];
            PR[63] = 1;
            rotate_regs();
        } else if (AR[EC] != 0) {
            AR[LC] = AR[LC];
            AR[EC]--;
            PR[63] = 0;
            rotate_regs();
        } else {
            AR[LC] = AR[LC];
            AR[EC] = AR[EC];
            PR[63] = 0;
            CFM.rrb.gr = CFM.rrb.gr;
            CFM.rrb.fr = CFM.rrb.fr;
            CFM.rrb.pr = CFM.rrb.pr;
        }
        break;

    case 'wtop':
    case 'wexit':                              // SW pipelined while loop
        if (slot != 2)
            illegal_operation_fault();
        if (btype == 'wtop')  tmp_taken =  (PR[qp] || (AR[EC] u> 1));
        if (btype == 'wexit') tmp_taken = !(PR[qp] || (AR[EC] u> 1));
        if (PR[qp]) {
            AR[EC] = AR[EC];
            PR[63] = 0;
            rotate_regs();
        } else if (AR[EC] != 0) {
            AR[EC]--;
            PR[63] = 0;
            rotate_regs();
        } else {
            AR[EC] = AR[EC];
            PR[63] = 0;
            CFM.rrb.gr = CFM.rrb.gr;
            CFM.rrb.fr = CFM.rrb.fr;
            CFM.rrb.pr = CFM.rrb.pr;
        }
        break;
}
if (tmp_taken) {
    taken_branch = 1;
    IP = tmp_IP;                               // set the new value for IP
    if ((PSR.it && unimplemented_virtual_address(tmp_IP))
         || (!PSR.it && unimplemented_physical_address(tmp_IP)))

unimplemented_instruction_address_trap(lower_priv_transition,tmp_IP);
    if (lower_priv_transition && PSR.lp)
        lower_privilege_transfer_trap();
    if (PSR.tb)
        taken_branch_trap();
}
```

ブレーク (Break)

書式:

(qp) break imm_{21}	pseudo-op	
(qp) break.i imm_{21}	i_unit_form	I19
(qp) break.b imm_{21}	b_unit_form	B9
(qp) break.m imm_{21}	m_unit_form	M37
(qp) break.f imm_{21}	f_unit_form	F15
(qp) break.x imm_{62}	x_unit_form	X1

説明: Break Instruction (ブレーク命令) フォルトが発生する。i_unit_form、f_unit_form、および m_unit_form では、imm_{21} によって指定される値がゼロ拡張され、割り込み即値 (Interruption Immediate: IIM) コントロール・レジスタに格納される。

b_unit_form では、imm_{21} が無視され、値 0 が IIM コントロール・レジスタに格納される。

x_unit_form では、imm_{62} によって指定される値の下位 21 ビットがゼロ拡張され、IIM コントロール・レジスタに格納される。バンドルの L スロットに imm_{62} の上位 41 ビットが入る。

この命令には 5 つの形式があるが、各形式は、それぞれ特定のタイプの実行ユニットに対してのみ実行できる。実行するユニットのタイプが重要でない場合は、擬似オペコードを使用してよい。

操作:

```
if (PR[qp]) {
    if (b_unit_form)
        immediate = 0;
    else if (x_unit_form)
        immediate = zero_ext(imm62, 21);
    else // i_unit_form || m_unit_form || f_unit_form
        immediate = zero_ext(imm21, 21);

    break_instruction_fault(immediate);
}
```

スペキュレーション・チェック (Speculation Check)

書式 :

(qp) chk.s r_2, $target_{25}$	pseudo-op	
(qp) chk.s.i r_2, $target_{25}$	control_form, i_unit_form, gr_form	I20
(qp) chk.s.m r_2, $target_{25}$	control_form, m_unit_form, gr_form	M20
(qp) chk.s f_2, $target_{25}$	control_form, fr_form	M21
(qp) chk.a.*aclr* r_1, $target_{25}$	data_form, gr_form	M22
(qp) chk.a.*aclr* f_1, $target_{25}$	data_form, fr_form	M23

説明 :

コントロール・スペキュレーションまたはデータ・スペキュレーションによる計算の結果をチェックして、成功か失敗かが確認される。チェックが失敗した場合は、$target_{25}$ への分岐が実行される。

control_form では、成否はソース・レジスタに対する NaT ビットにより判定される。gr_form において GR r_2 に対応する NaT ビットが 1 の場合、あるいは fr_form において FR f_2 の内容が NaTVal の場合は、チェックは失敗である。

data_form では、成否は ALAT によって判定される。ALAT は、gr_form の場合には 汎用レジスタ指定子 r_1 を、fr_form の場合には浮動小数点レジスタ指定子 f_1 を使用して照会される。一致する ALAT エントリがない場合は、チェックは失敗である。ALAT エントリが一致したかどうかに関わらず、チェックを必要に応じて失敗させることができる。

$target_{25}$ オペランドで、アセンブリで分岐先のラベルを指定する。この命令を含むバンドルから分岐先バンドルまでの変位が計算されて、符号付き即値 (imm_{21}) としてこの命令内にエンコードされる (imm_{21} = $target_{25}$ - IP >> 4)。

汎用レジスタ・チェック用のこの命令の control_form は、I ユニットに対しても M ユニットに対してもエンコード可能である。実行するユニットのタイプが重要でない場合は、擬似オペコードを使用してよい。

data_form では、一致する ALAT エントリがある場合は、*aclr* コンプリータの値に基づいて、一致する ALAT エントリを必要に応じて無効にすることができる (表 7-9 を参照)。

表 7-9. ALAT クリアのためのコンプリータ

aclr コンプリータ	ALAT に対する効果
clr	一致する ALAT エントリを無効にする
nc	無効にしない

aclr コンプリータに *clr* 値を指定してチェックが成功した場合は、ALAT の一致するエントリが無効にされる。ただし、チェックが失敗した (一致する ALAT エントリがある場合でも失敗することがある) 場合は、一致する ALAT エントリを必要に応じて無効にできるが、必ずしも無効にする必要はない。したがって、データ・スペキュレーションのためのリカバリコードにおいては、一致する ALAT エントリが存在しないものと想定してはならない。

操作:

```
if (PR[qp]) {
    if (control_form) {
        if (fr_form && (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0)))
            disabled_fp_register_fault(tmp_isrcode, 0);
        check_type = gr_form ? CHKS_GENERAL : CHKS_FLOAT;
        fail = (gr_form && GR[r2].nat) || (fr_form && FR[f2] == NATVAL);
    } else {                                                    // data_form
        reg_type = gr_form ? GENERAL : FLOAT;
        alat_index = gr_form ? r1 : (data_form ? f1 : f2);

        check_type = gr_form ? CHKA_GENERAL : CHKA_FLOAT;
        fail = !alat_cmp(reg_type, alat_index);
    }
    if (fail) {
            taken_branch = 1;
            IP = IP + sign_ext((imm21 << 4), 25);
            if ((PSR.it && unimplemented_virtual_address(IP))
                || (!PSR.it && unimplemented_physical_address(IP)))
                unimplemented_instruction_address_trap(0, IP);
            if (PSR.tb)
                taken_branch_trap();
    }
    if (!fail && data_form && (aclr == 'clr'))
        alat_inval_single_entry(reg_type, alat_index);
}
```

intel®

RRB のクリア (Clear RRB)

書式:

clrrrb	all_form	B8
clrrrb.pr	pred_form	B8

説明:

all_form では、レジスタ・リネーム・ベース・レジスタ (CFM.rrb.gr、CFM.rrb.pr、および CFM.rrb.pr) がクリアされる。pred_form では、プレディケート用のレジスタ・リネーム・ベース・レジスタ (CFM.rrb.pr) だけがクリアされる。

この命令は、同一の命令グループ内の最後の命令でなければならない。そうでない場合は、Illegal Operation (無効操作) フォルトが発生する。

この命令にはプレディケートは使用できない。

操作:

```
if (!followed_by_stop())
    illegal_operation_fault();

if (all_form) {
    CFM.rrb.gr = 0;
    CFM.rrb.fr = 0;
    CFM.rrb.pr = 0;
} else { // pred_form
    CFM.rrb.pr = 0;
}
```

比較 (Compare)

書式:

(*qp*) cmp.*crel*.*ctype* $p_1, p_2 = r_2, r_3$ register_form A6
(*qp*) cmp.*crel*.*ctype* $p_1, p_2 = imm_8, r_3$ imm8_form A8
(*qp*) cmp.*crel*.*ctype* $p_1, p_2 =$ r0, r_3 parallel_inequality_form A7
(*qp*) cmp.*crel*.*ctype* $p_1, p_2 = r_3$, r0 pseudo-op

説明:

2 つのソース・オペランドが、crel によって指定される 10 種類の関係のいずれか 1 つについて比較される。これによって、ブール結果が生成される。ブール結果は、比較条件が真の場合は 1 に、真でない場合は 0 になる。この結果は、2 つのプレディケート・レジスタ・デスティネーション p_1 および p_2 に書き込まれる。結果が両デスティネーションにどのように書き込まれるかは、*ctype* によって指定される比較タイプによって決まる。

比較タイプは、プレディケート・ターゲットがどのように比較結果に基づいて更新されるかを示す。通常タイプでは、比較結果を一方のターゲットに書き、結果の補数を他方のターゲットに書く。並列タイプでは、特定の比較結果に対してのみ両ターゲットを更新する。これにより、同じプレディケート・レジスタをターゲットとして、複数の OR タイプの同時比較や複数の AND タイプの同時比較が可能になる。

unc タイプは、*修飾プレディケートには関わらず*、まず最初に両プレディケート・ターゲットを 0 に初期化する点が特殊である。その後、通常タイプと同じ操作を実行する。これらの比較タイプの動作を表 7-10 に示す。空白のエントリは、プレディケート・ターゲットが変わらないことを示す。

表 7-10. 比較タイプ

ctype	擬似オペコード	PR[*qp*]==0		PR[*qp*]==1					
				結果 ==0、ソースに NaT なし		結果 ==1、ソースに NaT なし		ソースに 1 つ以上の NaT あり	
		PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]
none				0	1	1	0	0	0
unc		0	0	0	1	1	0	0	0
or						1	1		
and				0	0			0	0
or.andcm						1	0		
orcm	or			1	1				
andcm	and					0	0	0	0
and.orcm	or.andcm			0	1				

register_from では、第 1 オペランドは GR r_2 であり、imm8_form では、第 1 オペランドは符号拡張された imm8 エンコーディング・フィールドで与えられる。また、parallel_inequality_form では、第 1 オペランドは GR 0 でなければならない。parallel_inequality_form は、比較タイプが並列タイプのいずれか

であり、かつ関係が不等 (>、>=、<、<=) であるときに限り使用される。以下を参照のこと。

2 つのプレディケート・レジスタ・デスティネーションが同じ (p_1 と p_2 が同一のプレディケート・レジスタを指定する) 場合は、修飾プレディケートがセットされているか、あるいは比較タイプが unc であれば、この命令は Illegal Operation (無効操作) フォルトを発生する。

10 種類の関係がすべてハードウェアで直接サポートされているわけではない。一部は実際には擬似オペコードである。それらの関係に対しては、アセンブラは単にソース・オペランド指定子やプレディケート・ターゲット指定子を切り換えて、サポートされている関係を利用する。一部の imm8_form の擬似オペコード比較については、アセンブラは即値から 1 を引いて、許容される即値範囲をわずかに変更する。6 つの並列比較タイプのうち、3 つのタイプは実際には擬似オペコードである。アセンブラは、単に、サポートされているタイプと否定の関係を使用する。サポートされている関係と、擬似オペコードがそれらの関係にどのようにマップされているかを、通常タイプおよび unc タイプの比較については表 7-11 に、並列タイプの比較については表 7-12 にそれぞれ示す。

表 7-11. 通常および unc の比較での 64 ビット比較関係

crel	比較関係 (*a* rel *b*)		register_form での擬似オペコード			imm8_form での擬似オペコード			即値の範囲
eq	$a == b$								-128 .. 127
ne	$a != b$		eq		$p_1 \leftrightarrow p_2$	eq		$p_1 \leftrightarrow p_2$	-128 .. 127
lt	$a < b$	符号付き							-128 .. 127
le	$a <= b$		lt	a ↔ b	$p_1 \leftrightarrow p_2$	lt	a-1		-127 .. 128
gt	$a > b$		lt	a ↔ b		lt	a-1	$p_1 \leftrightarrow p_2$	-127 .. 128
ge	$a >= b$		lt		$p_1 \leftrightarrow p_2$	lt		$p_1 \leftrightarrow p_2$	-128 .. 127
ltu	$a < b$	符号なし							0 .. 127, 2^{64}-128 .. 2^{64}-1
leu	$a <= b$		ltu	a ↔ b	$p_1 \leftrightarrow p_2$	ltu	a-1		1 .. 128, 2^{64}-127 .. 2^{64}
gtu	$a > b$		ltu	a ↔ b		ltu	a-1	$p_1 \leftrightarrow p_2$	1 .. 128, 2^{64}-127 .. 2^{64}
geu	$a >= b$		ltu		$p_1 \leftrightarrow p_2$	ltu		$p_1 \leftrightarrow p_2$	0 .. 127, 2^{64}-128 .. 2^{64}-1

並列タイプは、限定された関係とオペランドのセットに対してのみ使用できる。これらのタイプでは、2 つのレジスタ間またはレジスタと即値との間の等および不等比較に使用できる。あるいは、レジスタと GR 0 との間の不等比較に使用できる。符号なしの関係は、どちらかのオペランドがゼロのときはあまり有用でないので用意されていない。並列の不等比較については、第 1 オペランド (GR r_2) が GR 0 である比較についてのみ、ハードウェアで直接サポートされている。第 2 オペランドが GR 0 である比較は擬似オペコードであり、アセンブラはそのレジスタ指定子を切り換えて、反対の関係を利用する

表 7-12. 並列比較での 64 ビット比較関係

crel	比較関係 (*a* rel *b*)		register_form での擬似オペコード	即値の範囲
eq	$a == b$			-128 .. 127
ne	$a != b$			-128 .. 127
lt	$0 < b$	符号付き		imm8_form なし
lt	$a < 0$		gt $a \leftrightarrow b$	
le	$0 <= b$			
le	$a <= 0$		ge $a \leftrightarrow b$	
gt	$0 > b$			
gt	$a > 0$		lt $a \leftrightarrow b$	
ge	$0 >= b$			
ge	$a >= 0$		le $a \leftrightarrow b$	

操作：

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    tmp_nat = (register_form ? GR[r2].nat : 0) || GR[r3].nat;
    if (register_form)
        tmp_src = GR[r2];
    else if (imm8_form)
        tmp_src = sign_ext(imm8, 8);
    else // parallel_inequality_form
        tmp_src = 0;

    if      (crel == 'eq')  tmp_rel = tmp_src == GR[r3];
    else if (crel == 'ne')  tmp_rel = tmp_src != GR[r3];
    else if (crel == 'lt')  tmp_rel = lesser_signed(tmp_src,  GR[r3]);
    else if (crel == 'le')  tmp_rel = lesser_equal_signed(tmp_src, GR[r3]);
    else if (crel == 'gt')  tmp_rel = greater_signed(tmp_src,  GR[r3]);
    else if (crel == 'ge')  tmp_rel = greater_equal_signed(tmp_src, GR[r3]);
    else if (crel == 'ltu') tmp_rel = lesser(tmp_src, GR[r3]);
    else if (crel == 'leu') tmp_rel = lesser_equal(tmp_src, GR[r3]);
    else if (crel == 'gtu') tmp_rel = greater(tmp_src, GR[r3]);
    else                    tmp_rel = greater_equal(tmp_src, GR[r3]);// 'geu'

    switch (ctype) {
        case 'and':                                 // and-type compare
            if (tmp_nat || !tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break
        case 'or':                                  // or-type compare
            if (!tmp_nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case 'or.andcm':                            // or.andcm-type compare
            if (!tmp_nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
            }
            break;
        case 'unc':                                 // unc-type compare
        default:                                    // normal compare
```

```
            if (tmp_nat) {
                PR[p1] = 0;
                PR[p2] = 0;
            } else {
                PR[p1] = tmp_rel;
                PR[p2] = !tmp_rel;
            }
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

ワード比較 (Compare Word)

書式：

(*qp*) cmp4.*crel*.*ctype* $p_1, p_2 = r_2, r_3$	register_form A6
(*qp*) cmp4.*crel*.*ctype* $p_1, p_2 = imm_8, r_3$	imm8_form A8
(*qp*) cmp4.*crel*.*ctype* $p_1, p_2 =$ r0, r_3	parallel_inequality_form A7
(*qp*) cmp4.*crel*.*ctype* $p_1, p_2 = r_3$, r0	pseudo-op

説明：

2 つのソース・オペランドのそれぞれの最下位 32 ビットが、*crel* によって指定される 10 種類の関係のいずれか 1 つについて比較される。この比較により、ブール結果が生成される。ブール結果は、比較条件が真の場合に 1 になり、真でない場合に 0 になる。この結果は、2 つのプレディケート・レジスタ・デスティネーション p_1 と p_2 に書き込まれる。結果がデスティネーションにどのように書き込まれるかは、*ctype* によって指定される比較タイプによって決まる。Compare 命令と 7-22 ページの表 7-10 を参照のこと。

register_form では、第 1 オペランドは GR r_2 であり、imm8_form では、第 1 オペランドは符号拡張された imm8 エンコーディング・フィールドで与えられる。また、parallel_inequality_form では、第 1 オペランドは GR 0 でなければならない。parallel_inequality_form は、比較タイプが並列タイプのいずれかであり、かつ関係が不等 (>、>=、<、<=) であるときに限り使用される。Compare 命令と 7-24 ページの表 7-12 を参照のこと。

2 つのプレディケート・レジスタ・デスティネーションが同じ (p_1 と p_2 が同一のプレディケート・レジスタを指定する) 場合は、修飾プレディケートがセットされているか、あるいは比較タイプが unc であれば、この命令は無効操作 (Illegal Operation) フォルトを発生する。

10 種類の関係がすべてハードウェアで直接サポートされているわけではない。一部は実際には擬似オペコードである。Compare 命令と 7-23 ページの表 7-11 および表 7-12 を参照のこと。即値の範囲を下の表に示す。

表 7-13. 32 ビット比較での即値範囲

crel	比較関係 (*a* rel *b*)		即値の範囲
eq	*a* == *b*		-128 .. 127
ne	*a* != *b*		-128 .. 127
lt	*a* < *b*	符号付き	-128 .. 127
le	*a* <= *b*		-127 .. 128
gt	*a* > *b*		-127 .. 128
ge	*a* >= *b*		-128 .. 127
ltu	*a* < *b*	符号なし	0 .. 127, 2^{32}-128 .. 2^{32}-1
leu	*a* <= *b*		1 .. 128, 2^{32}-127 .. 2^{32}
gtu	*a* > *b*		1 .. 128, 2^{32}-127 .. 2^{32}
geu	*a* >= *b*		0 .. 127, 2^{32}-128 .. 2^{32}-1

操作:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    tmp_nat = (register_form ? GR[r2].nat : 0) || GR[r3].nat;

    if (register_form)
        tmp_src = GR[r2];
    else if (imm8_form)
        tmp_src = sign_ext(imm8, 8);
    else // parallel_inequality_form
        tmp_src = 0;

    if      (crel == 'eq')        tmp_rel = tmp_src{31:0} == GR[r3]{31:0};
    else if  (crel == 'ne')       tmp_rel = tmp_src{31:0} != GR[r3]{31:0};
    else if (crel == 'lt')
        tmp_rel = lesser_signed(sign_ext(tmp_src, 32), sign_ext(GR[r3], 32));
    else if (crel == 'le')
        tmp_rel = lesser_equal_signed(sign_ext(tmp_src, 32), sign_ext(GR[r3], 32));
    else if (crel == 'gt')
        tmp_rel = greater_signed(sign_ext(tmp_src, 32), sign_ext(GR[r3], 32));
    else if (crel == 'ge')
        tmp_rel = greater_equal_signed(sign_ext(tmp_src, 32), sign_ext(GR[r3], 32));
    else if (crel == 'ltu')
        tmp_rel = lesser(zero_ext(tmp_src, 32), zero_ext(GR[r3], 32));
    else if (crel == 'leu')
        tmp_rel = lesser_equal(zero_ext(tmp_src, 32), zero_ext(GR[r3], 32));
    else if (crel == 'gtu')
        tmp_rel = greater(zero_ext(tmp_src, 32), zero_ext(GR[r3], 32));
    else        // 'geu'
        tmp_rel = greater_equal(zero_ext(tmp_src, 32), zero_ext(GR[r3], 32));

    switch (ctype) {
        case 'and':                                    // and-type compare
            if (tmp_nat || !tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;
        case 'or':                                     // or-type compare
            if (!tmp_nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case 'or.andcm':                               // or.andcm-type compare
            if (!tmp_nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
            }
            break;
        case 'unc':                                    // unc-type compare
        default:                                       // normal compare
            if (tmp_nat) {
                PR[p1] = 0;
                PR[p2] = 0;
            } else {
                PR[p1] = tmp_rel;
                PR[p2] = !tmp_rel;
            }
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

比較交換 (Compare And Exchange)

書式: (*qp*) cmpxchg*sz.sem.ldhint* r_1 = [r_3], r_2, ar.ccv M16

説明: GR r_3 の値によって指定されるアドレスから始まるメモリ位置から *sz* バイトの値が読み込まれる。この値はゼロ拡張され、比較交換での比較値 (cmpxchg Compare Value) アプリケーション・レジスタの内容 (AR[CCV]) と比較される。2 つが等しい場合は、GR r_2 の最下位の sz ビットが取り出され、GR r_3 の値によって指定されるアドレスから始まるメモリ位置に書き込まれる。メモリから読み込まれ、ゼロ拡張された値は GR r_1 に格納され、GR r_1 に対応する NaT ビットがクリアされる。

sz コンプリータの値を表 7-14 に示す。*sem* コンプリータでセマフォ操作のタイプを指定する。それらの操作の説明を、表 7-15 に示す

表 7-14. 比較交換のメモリ・サイズ

sz コンプリータ	アクセスされるバイト数
1	1
2	2
4	4
8	8

表 7-15. 比較交換セマフォのタイプ

sem コンプリータ	順序付けの語彙	セマフォ操作
acq	取得	後続のすべてのデータ・メモリ・アクセスの前に、メモリに対する読み取り / 書き込みが検出可能になる。
rel	解放	先行のすべてのデータ・メモリ・アクセスの後に、メモリに対する読み取り / 書き込みが検出可能になる。

GR r_3 の値によって指定されるアドレスが、メモリでアクセスされるサイズに自然にアライメントされていない場合は、ユーザ・マスク (User Mask) アライメント・チェック・ビット UM.ac (プロセッサ・ステータス・レジスタの PSR.ac) の状態に関わらず、非アライメント・データ参照 (Unaligned Data Reference) フォルトが発生する。

メモリに対する読み取りおよび書き込みはアトミックな操作であることが保証されている。

参照されるページに対する読み取りおよび書き込みの両方のアクセス特権が必要である。メモリへの書き込みが行われるかどうかに関わらず、書み込みアクセス特権のチェックが行われる。

ldhint コンプリータの値で、メモリ・アクセスの局所性を指定する。*ldhint* コンプリータの値の一覧を 7-113 ページの表 7-28 に示してある。局所性ヒントはプログラムの機能には影響せず、プログラム・コードでは無視することも

できる。詳細については、4-26 ページの「メモリ階層の制御と整合性」を参照のこと。

操作:

```
if (PR[qp]) {
    check_target_register(r1, SEMAPHORE);

    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(SEMAPHORE);

    paddr = tlb_translate(GR[r3], sz, SEMAPHORE, PSR.cpl, &mattr, &tmp_unused);

    if (!ma_supports_semaphores(mattr))
        unsupported_data_reference_fault(SEMAPHORE, GR[r3]);

    if (sem == 'acq') {
        val = mem_xchg_cond(AR[CCV], GR[r2], paddr, sz, UM.be, mattr, ACQUIRE,
                            ldhint);
    } else { // 'rel'
        val = mem_xchg_cond(AR[CCV], GR[r2], paddr, sz, UM.be, mattr, RELEASE,
                            ldhint);
    }
    val = zero_ext(val, sz * 8);

    if (AR[CCV] == val)
        alat_inval_multiple_entries(paddr, sz);

    GR[r1] = val;
    GR[r1].nat = 0;
}
```

ゼロ・インデックスの算出

書式：

(qp) czx1.l $r_1 = r_3$	one_byte_form, left_form	I29
(qp) czx1.r $r_1 = r_3$	one_byte_form, right_form	I29
(qp) czx2.l $r_1 = r_3$	two_byte_form, left_form	I29
(qp) czx2.r $r_1 = r_3$	two_byte_form, right_form	I29

説明： ゼロ要素を探して GR r_3 がスキャンされる。要素は、8 ビットにアライメントされた 1 バイト (one_byte_form) か 16 ビットにアライメントされた 2 バイト (two_byte_form) である。最初に検出されたゼロ要素のインデックスが GR r_1 に格納される。GR r_3 にゼロ要素がない場合は、デフォルト値が GR r_2 に格納される。表 7-16 に起こりうる結果を示す。left_form では、ソースが最上位要素から最下位要素にスキャンされ、right_form では、ソースが最下位要素から最上位要素にスキャンされる。

表 7-16. czx の結果の範囲

サイズ	要素幅	ゼロ要素が検出された場合の結果の範囲	ゼロ要素が検出されなかった場合のデフォルト結果
1	8 ビット	0-7	8
2	16 ビット	0-3	4

操作：

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {
        if (left_form) {                // scan from most significant down
            if      ((GR[r3] & 0xff00000000000000) == 0) GR[r1] = 0;
            else if ((GR[r3] & 0x00ff000000000000) == 0) GR[r1] = 1;
            else if ((GR[r3] & 0x0000ff0000000000) == 0) GR[r1] = 2;
            else if ((GR[r3] & 0x000000ff00000000) == 0) GR[r1] = 3;
            else if ((GR[r3] & 0x00000000ff000000) == 0) GR[r1] = 4;
            else if ((GR[r3] & 0x0000000000ff0000) == 0) GR[r1] = 5;
            else if ((GR[r3] & 0x000000000000ff00) == 0) GR[r1] = 6;
            else if ((GR[r3] & 0x00000000000000ff) == 0) GR[r1] = 7;
            else GR[r1] = 8;
        } else { // right_form          scan from least significant up
            if      ((GR[r3] & 0x00000000000000ff) == 0) GR[r1] = 0;
            else if ((GR[r3] & 0x000000000000ff00) == 0) GR[r1] = 1;
            else if ((GR[r3] & 0x0000000000ff0000) == 0) GR[r1] = 2;
            else if ((GR[r3] & 0x00000000ff000000) == 0) GR[r1] = 3;
            else if ((GR[r3] & 0x000000ff00000000) == 0) GR[r1] = 4;
            else if ((GR[r3] & 0x0000ff0000000000) == 0) GR[r1] = 5;
            else if ((GR[r3] & 0x00ff000000000000) == 0) GR[r1] = 6;
            else if ((GR[r3] & 0xff00000000000000) == 0) GR[r1] = 7;
            else GR[r1] = 8;
        }
    } else { // two_byte_form
        if (left_form) {                // scan from most significant down
            if      ((GR[r3] & 0xffff000000000000) == 0) GR[r1] = 0;
            else if ((GR[r3] & 0x0000ffff00000000) == 0) GR[r1] = 1;
            else if ((GR[r3] & 0x00000000ffff0000) == 0) GR[r1] = 2;
            else if ((GR[r3] & 0x000000000000ffff) == 0) GR[r1] = 3;
            else GR[r1] = 4;
        } else { // right_form          scan from least significant up
            if      ((GR[r3] & 0x000000000000ffff) == 0) GR[r1] = 0;
            else if ((GR[r3] & 0x00000000ffff0000) == 0) GR[r1] = 1;
            else if ((GR[r3] & 0x0000ffff00000000) == 0) GR[r1] = 2;
            else if ((GR[r3] & 0xffff000000000000) == 0) GR[r1] = 3;
            else GR[r1] = 4;
        }
    }
    GR[r1].nat = GR[r3].nat;
}
```

デポジット (Deposit)

書式:

(*qp*) dep $r_1 = r_2, r_3, pos_6, len_4$ merge_form, register_form I15
(*qp*) dep $r_1 = imm_1, r_3, pos_6, len_6$ merge_form, imm_form I14
(*qp*) dep.z $r_1 = r_2, pos_6, len_6$ zero_form, register_form I12
(*qp*) dep.z $r_1 = imm_8, pos_6, len_6$ zero_form, imm_form I13

説明:

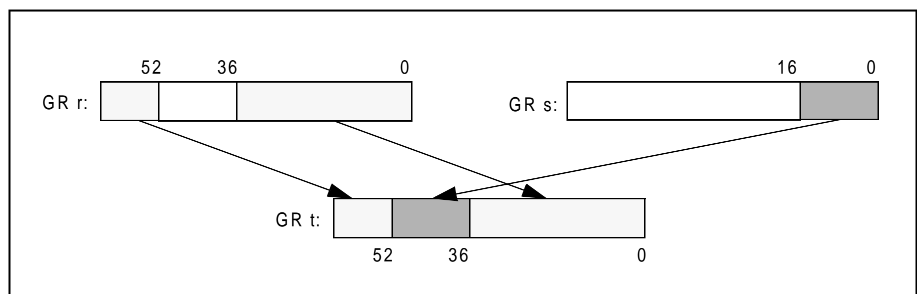
merge_form では、第 1 ソース・オペランドから右寄せビット・フィールドが取り出されて、それを GR r_3 の値の任意のビット位置に挿入した結果が GR r_1 に格納される。register_form では、第 1 ソース・オペランドは GR r_2 であり、imm_form では、第 1 オペランドは imm_1 を符号拡張した (つまり、すべて "1" かすべて "0") 値である。結果が挿入されるビット・フィールドは、即値 pos_6 によって指定されるビット位置から始まり、左方向に (最上位ビット方向に) 即値 len によって指定されるビット数のサイズをもつ。即値 *len* の範囲は、register_form では 1 ～ 16 であり、imm_form では 1 ～ 64 であることに注意されたい。即値 pos_6 の範囲は 0 ～ 63 である。

zero_form では、GR r_2 の値 (register_form の場合) か imm_8 を符号拡張した値 (imm_form の場合) で与えられる右寄せビット・フィールドが GR r_1 に挿入され、GR r_1 のその他のすべてのビットがゼロ・クリアされる。GR r_1 に挿入されるビット・フィールドは、即値 pos_6 によって指定されるビット位置から始まり、左方向に (最上位ビット方向に) 即値 *len* によって指定されるビット数のサイズをもつ。即値 *len* の範囲は 1 ～ 64 であり、即値 pos_6 の範囲は 0 ～ 63 である。

挿入されるビット・フィールドがターゲットのビット 63 を超える場合、つまり、$len + pos_6 > 64$ である場合は、挿入されるビット・フィールドの最上位の $len + pos_6 - 64$ ビットが切り捨てられる。即値 *len* は命令内では *len* - 1 としてエンコードされる。

`dep t = s、r、36、16` の操作の図解を図 7-5 に示す。

図 7-5. デポジットの例



操作:

```
if (PR[qp]) {
    check_target_register(r1);

    if (imm_form) {
        tmp_src = (merge_form ? sign_ext(imm1,1) : sign_ext(imm8, 8));
        tmp_nat = merge_form ? GR[r3].nat : 0;
        tmp_len = len6 ;
    } else {                                        // register_form
        tmp_src = GR[r2];
        tmp_nat = (merge_form ? GR[r3].nat : 0) || GR[r2].nat;
        tmp_len = merge_form ? len4 : len6 ;
    }
    if (pos6 + tmp_len u> 64)
        tmp_len = 64 - pos6;

    if (merge_form)
        GR[r1] = GR[r3];
    else // zero_form
        GR[r1] = 0;

    GR[r1]{(pos6 + tmp_len - 1):pos6} = tmp_src{(tmp_len - 1):0};
    GR[r1].nat = tmp_nat;
}
```

抽出 (Extract)

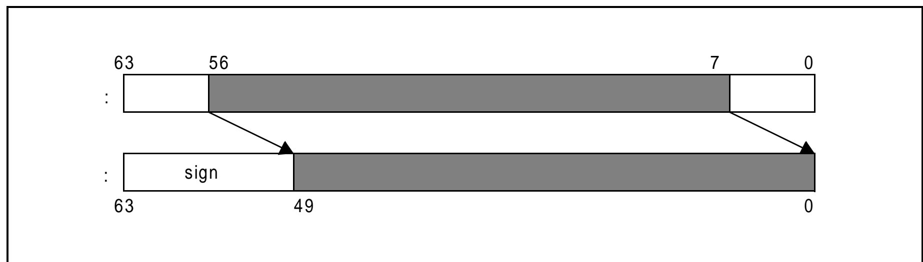
書式：

(qp) extr $r_1 = r_3, pos_6, len_6$	signed_form	I11
(qp) extr.u $r_1 = r_3, pos_6, len_6$	unsigned_form	I11

説明： GR r_3 からフィールドが抽出され、ゼロ拡張または符号拡張されて GR r_1 に格納される。抽出されるフィールドは、第 2 オペランドによって与えられるビット位置から始まり、左方向に len_6 のビット数のサイズをもつ。フィールドの開始ビット位置は即値 pos_6 によって指定される。抽出されたフィールドは、signed_form では符号拡張され、unsigned_form ではゼロ拡張される。符号は、抽出されたフィールドの最上位ビットから取られる。指定されたフィールドが GR r_3 の最上位ビットを超える場合は、符号は GR r_3 の最上位ビットとなる。即値 len_6 には、範囲 0 ～ 63 内の任意の値を使用でき、命令内では len_6 - 1 としてエンコードされる。即値 pos_6 は、範囲 1 ～ 64 内の任意の値を使用できる。

`extr t = r, 7, 50` の操作の図解を図 7-6 に示す。

図 7-6. 抽出の例



操作：

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_len = len6;

    if (pos6 + tmp_len u> 64)
        tmp_len = 64 - pos6;

    if (unsigned_form)
        GR[r1] = zero_ext(shift_right_unsigned(GR[r3], pos6), tmp_len);
    else // signed_form
        GR[r1] = sign_ext(shift_right_unsigned(GR[r3], pos6), tmp_len);

    GR[r1].nat = GR[r3].nat;
}
```

浮動小数点絶対値 (Floating-Point Absolute Value)

書式: (qp) fabs $f_1 = f_3$ (qp) fmerge.s f_1 = f0, f_3 の擬似オペコード

説明: FR f_3 の絶対値が計算され、その結果が FR f_1 に格納される。

FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

操作: 7-55 ページの「浮動小数点マージ (Floating-Point Merge)」を参照のこと。

浮動小数点加算 (Floating-Point Add)

書式 : (*qp*) fadd.*pc*.*sf* $f_1 = f_3, f_2$ (*qp*) fma.*pc*.*sf* $f_1 = f_3, f_1, f_2$ の擬似オペコード

説明 : FR f_3 と FR f_2 とが加算 (無限の精度で計算) され、FPSR.*sf*.*rc* に指定されている丸めモードと *pc* (さらに FPSR.*sf*.*pc* および FPSR.*sf*.*wre*) で指定される精度にしたがって丸められ、その結果が FR f_1 に格納される。FR f_3 か FR f_2 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

オペコードの *pc* のニーモニック値を表 7-17 に、*sf* のニーモニック値を表 7-18 にそれぞれ示す。ステータス・フィールドの *pc*、*wre*、および *rc* のエンコーディングと意味については、5-7 ページの表 5-4 および 5-9 ページの表 5-5 を参照のこと。

表 7-17. *pc* ニーモニックの指定値

pc ニーモニック	指定される精度
.s	単精度
.d	倍精度
無指定	動的 (つまり、ステータス・フィールドの pc 値を使用)

表 7-18. *sf* ニーモニックの値

sf ニーモニック	アクセスされるステータス・フィールド
.s0 または*無指定*	sf0
.s1	sf1
.s2	sf2
.s3	sf3

操作 : 7-53 ページの「浮動小数点積和 (Floating-Point Multiply Add)」を参照のこと。

浮動小数点絶対最大値 (Floating-Point Absolute Maximum)

書式 : (qp) famax.*sf* $f1 = f_2, f_3$ F8

説明 : 絶対値が大きい方のオペランドが FR f_1 に格納される。FR f_2 の絶対値が FR f_3 の絶対値と等しい場合は、FR f_1 には FR f_3 の値が格納される。

FR f_2 か FR f_3 が NaN である場合は、FR f_1 には FR f_3 の値が格納される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

この操作は、他の浮動小数点算術演算とは異なり、NaN を伝播させない。fcmp.lt 操作と同じ方法で無効操作 (Invalid Operation) が通知される。

sf のニーモニック値は、7-35 ページの表 7-18 に示してある。

操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_right = fp_reg_read(FR[f2]);
        tmp_left = fp_reg_read(FR[f3]);
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        FR[f1] = tmp_bool_res ? FR[f2] : FR[f3];

        fp_update_fpsr(sf, tmp_fp_env);
    }

    fp_update_psr(f1);
}
```

FP 例外 : 無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

浮動小数点絶対最小値 (Floating-Point Absolute Minimum)

書式 : (qp) famin.sf $f_1 = f_2, f_3$ F8

説明 : 絶対値が小さい方のオペランドが FR f_1 に格納される。FR f_2 の絶対値が FR f_3 の絶対値と等しい場合は、FR f_1 には FR f_3 の値が格納される。

FR f_2 か FR f_3 が NaN である場合は、FR f_1 には FR f_3 の値が格納される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

この操作は、他の浮動小数点算術演算とは異なり、NaN を伝播させない。fcmp.lt 操作と同じ方法で無効操作 (Invalid Operation) が通知される。

sf のニーモニック値は、7-35 ページの表 7-18 に示してある。

操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_left = fp_reg_read(FR[f2]);
        tmp_right = fp_reg_read(FR[f3]);
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        FR[f1] = tmp_bool_res ? FR[f2] : FR[f3];

        fp_update_fpsr(sf, tmp_fp_env);
    }

    fp_update_psr(f1);
}
```

FP 例外 : 無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

浮動小数点論理積 (Floating-Point Logical And)

書式 : (*qp*) fand $f_1 = f_2, f_3$ F9

説明 : FR f_2 と FR f_3 の両仮数フィールド間のビット単位の論理積が計算される。結果の値は FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f2].significand & FR[f3].significand;
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }
    fp_update_psr(f1);
}
```

FP 例外 : なし。

浮動小数点補数の論理積 (Floating-Point And Complemen)

書式: (qp) fandcm $f_1 = f_2, f_3$ F9

説明: FR f_2 の仮数フィールドと、FR f_3 の仮数フィールドをビット単位に補数をとったものとの間でビット単位の論理積が計算される。結果の値は FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f2].significand & ~FR[f3].significand;
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }
    fp_update_psr(f1);
}
```

FP 例外: なし。

キャッシュのフラッシュ (Flush Cache)

書式 : (*qp*) fc r_3 M28

説明 : GR r_3 の値によって指定されるアドレスに関連付けられるキャッシュ・ラインが、プロセッサ・キャッシュ階層のすべてのレベルから無効にされる。この無効化はコヒーレンス・ドメイン全体にブロードキャストされる。そのラインがキャッシュ階層のどのレベルにおいてもメモリと整合しない場合は、無効にされる前にメモリに書き込まれる。

操作の対象となるライン・サイズは最低 32 バイトである (32 バイト境界にアラインメントされる)。設計により、それより大きな領域をフラッシュできる。

この命令はデータ依存のルールに従う。つまり、同一ラインに対する先行および後続のメモリ参照に関して順序付けされる。このプロセッサによって行われる先行するすべてのストアが、メモリに書き戻されるデータに含まれるという点において、`fc` 命令にはデータ依存が存在する。`fc` 命令は順序付けのない操作であり、メモリ・フェンス (`mf`) 命令によって影響されない。`fc` 命令は `sync.i` 命令に関して順序付けられる。

操作 :

```
if (PR[qp]) {
    itype = NON_ACCESS|FC|READ;
    if (GR[r3].nat)
        register_nat_consumption_fault(itype);
    tmp_paddr = tlb_translate_nonaccess(GR[r3], itype);
    mem_flush(tmp_paddr);
}
```

浮動小数点フラグ・チェック (Floating-Point Check Flags)

書式： (*qp*) fchkf.*sf* $target_{25}$ F14

説明： FPSR.*sf*.flags のフラグが FPSR.s0.flags および FPSR.traps と比較される。FPSR.*sf*.flags 内のセットされているフラグのなかでイネーブルにされている FPSR.traps に対応するものがある場合、あるいは FPSR.*sf*.flags 内のセットされているフラグのなかで FPSR.s0.flags にセットされていないものがある場合には、$target_{25}$ への分岐が発生する。

$target_{25}$ オペランドには分岐先のラベルを指定する。これで、この命令を含むバンドルから分岐先バンドルまでの変位が計算されて、符号付き即値 (imm_{21}) として命令にエンコードされる (imm_{21} = $target_{25}$ - IP >> 4)。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。

操作：

```
if (PR[qp]) {
    switch (sf) {
        case 's0':
            tmp_flags = AR[FPSR].sf0.flags;
            break;
        case 's1':
            tmp_flags = AR[FPSR].sf1.flags;
            break;
        case 's2':
            tmp_flags = AR[FPSR].sf2.flags;
            break;
        case 's3':
            tmp_flags = AR[FPSR].sf3.flags;
            break;
    }
    if ((tmp_flags & ~AR[FPSR].traps) || (tmp_flags & ~AR[FPSR].sf0.flags)) {
        if (check_branch_implemented(FCHKF)) {
            taken_branch = 1;
            IP = IP + sign_ext((imm21 << 4), 25);
            if ((PSR.it && unimplemented_virtual_address(IP))
                || (!PSR.it && unimplemented_physical_address(IP)))
                unimplemented_instruction_address_trap(0, IP);
            if (PSR.tb)
                taken_branch_trap();
        } else
            speculation_fault(FCHKF, zero_ext(imm21, 21));
    }
}
```

FP 例外： なし。

浮動小数点分類 (Floating-Point Class)

書式: (*qp*) fclass.*fcrel*.*fctype* p_1, p_2 = f_2, *fclass*$_9$ F5

説明: FR f_2 の内容が、表 7-20 に示すように、*fclass*$_9$ コンプリータに従ってクラスに分類される。この操作により、*fcrel* コンプリータによる指定に従って、FR f_2 の内容が *fclass*$_9$ によって指定される浮動小数点数形式に一致しているかどうかが判定され、ブール結果が生成される。この結果は 2 つのプレディケート・レジスタ・デスティネーション p_1 および p_2 に書き込まれる。両デスティネーションに書き込まれる結果は、*fctype* によって指定される比較タイプによって決まる。

使用できるタイプは、通常 (または*無指定*) と unc である。7-45 ページの表 7-21 を参照のこと。アセンブリのシンタックスではメンバシップの有無の指定が可能であり、アセンブラがターゲット・プレディケートをスワップして、必要な結果を達成する。

表 7-19. 浮動小数点クラスの関係

fcrel	判定テストの関係
m	FR f_2 が *fclass9* によって指定されるパターンと一致する (メンバ資格あり)。
nm	FR f_2 が *fclass9* によって指定されるパターンと一致しない (メンバ資格なし)。

以下のいずれかの条件が成立する場合に、数値は *fclass*$_9$ によって指定されるパターンと一致していると判定される。

- 数値が NaTVal であり、かつ *fclass*$_9$ {8} が 1 である。
- 数値がクワイエット型 NaN であり、かつ *fclass*$_9$ {7} が 1 である。
- 数値がシグナル型 NaN であり、かつ *fclass*$_9$ {6} が 1 である。
- 数値の符号が、*fclass*$_9$ の下位 2 ビットのいずれかによって指定される符号と一致し、かつ数値の型 (符号は無視) が、表 7-20 に従って、*fclass*$_9$ の次の 4 ビットによって指定される数値の型と一致する。

注: *fclass*$_9$ を 0x1FF にすると、サポートされているいずれかのオペランドであるかどうかを判定することになる。

表 7-20 に使用されているクラス名は、5-4 ページの表 5-2 に定義してある。

表 7-20. 浮動小数点のクラス

fclass$_9$	クラス	ニーモニック
下のパターンで右側の型を判定できる。		
0x0100	NaTVal	@nat
0x080	クワイエット型 NaN	@qnan
0x040	シグナル型 NaN	@snan
下の 2 つのパターンの論理和で右側の型を判定できる。		

表 7-20. 浮動小数点のクラス (続き)

fclass9	クラス	ニーモニック
0x001	正	@pos
0x002	負	@neg
下の 4 つのパターンの論理和との論理積で右側の型を判定できる。		
0x004	ゼロ	@zero
0x008	非正規数	@unorm
0x010	正規数	@norm
0x020	無限大	@inf

操作 :

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    if (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    tmp_rel = ((fclass9{0} && !FR[f2].sign || fclass9{1} && FR[f2].sign)
                &&  ((fclass9{2} && fp_is_zero(FR[f2]))||
                     (fclass9{3} && fp_is_unorm(FR[f2])) ||
                     (fclass9{4} && fp_is_normal(FR[f2])) ||
                     (fclass9{5} && fp_is_inf(FR[f2]))
                    )
                )
            || (fclass9{6} && fp_is_snan(FR[f2]))
            || (fclass9{7} && fp_is_qnan(FR[f2]))
            || (fclass9{8} && fp_is_natval(FR[f2]));

    tmp_nat = fp_is_natval(FR[f2]) && (!fclass9{8});

    if (tmp_nat) {
        PR[p1] = 0;
        PR[p2] = 0;
    } else {
        PR[p1] = tmp_rel;
        PR[p2] = !tmp_rel;
    }
} else {
    if (fctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

FP 例外 : なし。

浮動小数点フラグ・クリア (Floating-Point Clear Flags)

書式 : (*qp*) fclrf.*sf* F13

説明 : ステータス・フィールドの 6 ビットのフラグ・フィールドをゼロにリセットする。*sf* のニーモニック値は 7-35 ページの表 7-18 に示してある。

操作 :

```
if (PR[qp]) {
    fp_set_sf_flags(sf, 0);
}
```

FP 例外 : なし。

浮動小数点比較 (Floating-Point Compare)

書式 : (qp) fcmp.*frel*.*fctype*.*sf* $p_1, p_2 = f_2, f_3$ F4

説明 : 2 つのソース・オペランドが、*frel* によって指定される 12 種類の比較関係のいずれか 1 つと比較される。この操作により、比較条件が真の場合にブール型の結果が 1 になり、真でない場合に 0 になる。この比較結果は、2 つのプレディケート・レジスタ・デスティネーション p_1 および p_2 に書き込まれる。結果がどのようにデスティネーションに書き込まれるかは、*fctype* によって指定される比較タイプによって決まる。使用できる比較タイプは通常 (または*無指定*) と unc である。

表 7-21. 浮動小数点の比較タイプ

fctype	PR[*qp*]==0		PR[*qp*]==1					
			結果 ==0、ソースに NaTVal なし		結果 ==1、ソースに NaTVal なし		ソースに NaTVal が 1 つ以上あり	
	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]
none			0	1	1	0	0	0
unc	0	0	0	1	1	0	0	0

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。

各比較タイプに対して定義されている比較関係の一覧を表 7-22 に示す。12 種類の比較関係のすべてがハードウェアで直接サポートされているわけではない。一部は実際には擬似オペコードである。それらの比較関係に対しては、アセンブラが単にソース・オペランド指定子やプレディケート・ターゲット指定子を切り換えて、サポートされている比較関係を利用する。

表 7-22. 浮動小数点の比較関係

frel	frel コンプリータの意味	関係	擬似オペコード	オペランドがクワイエット型 NaN の場合の無効通知の有無
eq	等しい	$f_2 == f_3$		なし
lt	より小	$f_2 < f_3$		あり
le	より小か等しい	$f_2 <= f_3$		あり
gt	より大	$f_2 > f_3$	lt $f_2 \leftrightarrow f_3$	あり
ge	より大か等しい	$f_2 >= f_3$	le $f_2 \leftrightarrow f_3$	あり
unord	非順序化	$f_2 ? f_3$		なし
neq	等しくない	$!(f_2 == f_3)$	eq $p_1 \leftrightarrow p_2$	なし
nlt	より小でない	$!(f_2 < f_3)$	lt $p_1 \leftrightarrow p_2$	あり
nle	より小でないか等しい	$!(f_2 <= f_3)$	le $p_1 \leftrightarrow p_2$	あり
ngt	より大でない	$!(f_2 > f_3)$	lt $f_2 \leftrightarrow f_3$ $p_1 \leftrightarrow p_2$	あり

表 7-22. 浮動小数点の比較関係

frel	frel コンプリータの意味	関係	擬似オペコード	オペランドがクワイエット型 NaN の場合の無効通知の有無
nge	より大でないか等しい	$!(f_2 >= f_3)$	le $f_2 \leftrightarrow f_3$ $p_1 \leftrightarrow p_2$	あり
ord	順序化	$!(f_2 ? f_3)$	unord $p_1 \leftrightarrow p_2$	なし

操作：

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    if (tmp_isrcode = fp_reg_disabled(f2, f3, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        PR[p1] = 0;
        PR[p2] = 0;
    } else {
        fcmp_exception_fault_check(f2, f3, frel, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = fp_reg_read(FR[f2]);
        tmp_fr3 = fp_reg_read(FR[f3]);

        if      (frel == 'eq') tmp_rel = fp_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'lt') tmp_rel = fp_less_than(tmp_fr2, tmp_fr3);
        else if (frel == 'le') tmp_rel = fp_lesser_or_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'gt') tmp_rel = fp_less_than(tmp_fr3, tmp_fr2);
        else if (frel == 'ge') tmp_rel = fp_lesser_or_equal(tmp_fr3, tmp_fr2);
        else if (frel == 'unord') tmp_rel = fp_unordered(tmp_fr2, tmp_fr3);
        else if (frel == 'neq') tmp_rel = !fp_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'nlt') tmp_rel = !fp_less_than(tmp_fr2, tmp_fr3);
        else if (frel == 'nle') tmp_rel = !fp_lesser_or_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'ngt') tmp_rel = !fp_less_than(tmp_fr3, tmp_fr2);
        else if (frel == 'nge') tmp_rel = !fp_lesser_or_equal(tmp_fr3, tmp_fr2);
        else                    tmp_rel = !fp_unordered(tmp_fr2, tmp_fr3); //
'ord'

        PR[p1] = tmp_rel;
        PR[p2] = !tmp_rel;

        fp_update_fpsr(sf, tmp_fp_env);
    }
} else {
    if (fctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

FP 例外：

無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

intel®

浮動小数点から整数への変換 (Convert Floating-Point to Integer)

書式：

(qp) fcvt.fx.sf $f_1 = f_2$	signed_form	F10
(qp) fcvt.fx.trunc.sf $f_1 = f_2$	signed_form, trunc_form	F10
(qp) fcvt.fxu.sf $f_1 = f_2$	unsigned_form	F10
(qp) fcvt.fxu.trunc.sf $f_1 = f_2$	unsigned_form, trunc_form	F10

説明： FR f_2 がレジスタ形式の浮動小数点値として扱われ、FPSR.*sf.rc* によって指定される丸めモードを使用するか、あるいは trunc_form の場合にはゼロ側への丸めモードを使用して、符号付き (signed_form の場合) または符号なし (unsigned_form の場合) の整数に変換される。結果は、FR f_1 の 64 ビットの仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_2 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。

操作：

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fcvt_exception_fault_check(f2, sf,
                                      signed_form, trunc_form, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan(tmp_default_result)) {
            FR[f1].significand = INTEGER_INDEFINITE;
            FR[f1].exponent = FP_INTEGER_EXP;
            FR[f1].sign = FP_SIGN_POSITIVE;
        } else {
            tmp_res = fp_ieee_rnd_to_int(fp_reg_read(FR[f2]), &tmp_fp_env);
            if (tmp_res.exponent)
                tmp_res.significand = fp_U64_rsh(
                    tmp_res.significand, (FP_INTEGER_EXP - tmp_res.exponent));
            if (signed_form && tmp_res.sign)
                tmp_res.significand = (~tmp_res.significand) + 1;

            FR[f1].significand = tmp_res.significand;
            FR[f1].exponent = FP_INTEGER_EXP;
            FR[f1].sign = FP_SIGN_POSITIVE;
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP 例外： 無効操作 (Invalid Operation: V)　　不正確結果 (Inexact: I)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

符号付き整数から浮動小数点への変換 (Convert Signed Integer to Floating-point)

書式：　(*qp*)　fcvt.xf $f_1 = f_2$　　F11

説明：　FR f_2 の 64 ビットの仮数が符号付き整数として扱われ、そのレジスタ・ファイル精度の浮動小数点表現が FR f_1 に格納される。

FR f_2 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

この操作は常に正確であり、丸めモードによる影響を受けない。

操作：

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2])) {
        FR[f1] = NATVAL;
    } else {
        tmp_res = FR[f2];
        if (tmp_res.significand{63}) {
            tmp_res.significand = (~tmp_res.significand) + 1;
            tmp_res.sign = 1;
        } else
            tmp_res.sign = 0;

        tmp_res.exponent = FP_INTEGER_EXP;
        tmp_res = fp_normalize(tmp_res);

        FR[f1].significand = tmp_res.significand;
        FR[f1].exponent = tmp_res.exponent;
        FR[f1].sign = tmp_res.sign;
    }
    fp_update_psr(f1);
}
```

FP 例外：　なし。

符号付き整数から浮動小数点への変換 (Convert Unsigned Integer to Floating-point)

書式: (*qp*) fcvt.xuf.*pc*.*sf* $f_1 = f_3$ (unsigned_form) (*qp*) fma.pc.*sf* $f_1 = f_3, f_1$, f0 の擬似オペコード

説明: FR f_3 が FR 1 と乗算され、FPSR.*sf*.*rc* によって指定される丸めモードと *pc* (さらに FPSR.*sf*.*pc* および FPSR.*sf*.*wre*) によって指定される精度にしたがって丸められ、その結果が FR f_1 に格納される。

注： FR f_3 を FR 1 (1.0) と乗算すると、浮動小数点レジスタ・ファイル内で整数の標準表現が正規化され、通常の浮動小数点数が生成される。

FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

オペコードの *pc* のニーモニック値は 7-35 ページの表 7-17 に示してある。*sf* のニーモニック値は 7-35 ページの表 7-18 に示してある。ステータス・フィールドの *pc*、*wre*、および *rc* のエンコーディングと意味については、5-7 ページの表 5-4 および 5-9 ページの表 5-5 を参照のこと。

操作: 7-53 ページの「浮動小数点積和 (Floating-Point Multiply Add)」を参照のこと。

即値のフェッチおよび加算 (Fetch And Add Immediate)

書式 :

(qp) fetchadd4.*sem*.*ldhint* r_1 = [r_3], inc_3 four_byte_form M17
(qp) fetchadd8.*sem*.*ldhint* r_1 = [r_3], inc_3 eight_byte_form M17

説明 :

4 または 8 バイトからなる値が、GR r_3 の値によって指定されるアドレスから始まるメモリ位置から読み込まれる。この値がゼロ拡張され、inc_3 によって指定される符号拡張された即値に加算される。inc_3 によって指定できる値は、-16、-8、-4、-1、1、4、8、16 である。次に、加算結果の最下位 4 または 8 バイトが、GR r_3 の値によって指定されるアドレスから始まるメモリ位置に書き込まれる。メモリから読み込まれたゼロ拡張された値が GR r_1 に格納され、GR r_1 に対応する NaT ビットがクリアされる。

sem コンプリータでセマフォ操作を指定する。これらの操作の説明を表 7-23 に示す。

表 7-23. フェッチおよび加算セマフォのタイプ

sem コンプリータ	順序付けの語彙	セマフォ操作
acq	取得	後続のすべてのデータ・メモリ・アクセスの前に、メモリに対する読み取り / 書き込みが検出可能になる。
rel	解放	後続のすべてのデータ・メモリ・アクセスの後に、メモリに対する読み取り / 書き込みが検出可能になる。

メモリに対する読み取りおよび書き込みはアトミックな操作であることが保証されている。

GR r_3 の値によって指定されるアドレスが、メモリでアクセスされるサイズに自然にアライメントされていない場合は、ユーザ・マスク (User Mask) アライメント・チェック・ビット UM.ac (プロセッサ・ステータス・レジスタの PSR.ac) の状態に関わらず、非アライメント・データ参照 (Unaligned Data Reference) フォルトが発生する。

参照されるページに対する読み取りおよび書き込みの両方のアクセス特権が必要である。メモリへの書き込が行われるかどうかに関わらず、書き込みアクセス特権のチェックが行われる。

ldhint コンプリータの値でメモリ・アクセスの局所性を指定する。*ldhint* コンプリータの値の一覧が、7-113 ページの表 7-28 に示してある。局所性ヒントはプログラムの機能には影響せず、プログラム・コードにより無視することもできる。

操作 :

```
if (PR[qp]) {
    check_target_register(r1, SEMAPHORE);

    if (GR[r3].nat)
        register_nat_consumption_fault(SEMAPHORE);

    size = four_byte_form ? 4 : 8;

    paddr = tlb_translate(GR[r3], size, SEMAPHORE, PSR.cpl, &mattr,
&tmp_unused);
    if (!ma_supports_fetchadd(mattr))
```

```
        unsupported_data_reference_fault(SEMAPHORE, GR[r3]);

    if (sem == 'acq')
        val = mem_xchg_add(inc3, paddr, size, UM.be, mattr, ACQUIRE, ldhint);
    else // 'rel'
        val = mem_xchg_add(inc3, paddr, size, UM.be, mattr, RELEASE, ldhint);

    alat_inval_multiple_entries(paddr, size);

    GR[r1] = zero_ext(val, size * 8);
    GR[r1].nat = 0;
}
```

レジスタ・スタックのフラッシュ (Flush Register Stack)

書式: flushrs M25

説明: レジスタ・スタックのダーティなパーティションにスタックされているすべての汎用レジスタがバッキング・ストアに書き込まれてから、実行が継続される。ダーティなパーティションには、以前のプロシージャ・フレームの、まだバッキング・ストアにセーブされていないレジスタが含まれている。

この命令の実行を終了した後は、AR[BSPSTORE] が AR[BSP] に等しくなる。

この命令は、同一命令グループ内の最初の命令でなければならない。そうでない場合は、結果は不定になる。この命令ではプレディケートは使用できない。

操作:

```
while (AR[BSPSTORE] != AR[BSP]) {
    rse_store(MANDATORY);                  // increments AR[BSPSTORE]
    deliver_unmasked_pending_external_interrupt();
}
```

浮動小数点積和 (Floating-Point Multiply Add)

書式： (qp) fma.*pc*.*sf* $f_1 = f_3, f_4, f_2$ F1

説明： FR f_3 と FR f_4 との積が無限の精度で計算され、次に FR f_2 がこの積にやはり無限の精度で加算される。次に、結果の値が、FPSR.*sf*.*rc* によって指定される丸めモードと *pc* (さらに FPSR.*sf*.*pc* および FPSR.*sf*.*wre*) によって指定される精度にしたがって丸められ、その結果が FR f_1 に格納される。

FR f_3、FR f_4、および FR f_2 のどれか 1 つでも NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

f_2 が f0 の場合は、積和演算の代わりに IEEE の乗算が行われる。7-60 ページの「浮動小数点乗算 (Floating-Point Multiply)」を参照のこと。

オペコードの *pc* のニーモニック値は 7-35 ページの表 7-17 に示してある。*sf* のニーモニック値は 7-35 ページの表 7-18 に示してある。ステータス・フィールドの *pc*、*wre*、および *rc* のエンコーディングと意味については、5-7 ページの表 5-4 および 5-9 ページの表 5-5 を参照のこと。

操作：

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) || fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fma_exception_fault_check(f2, f3, f4,
                                                       pc, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[f3]), fp_reg_read(FR[f4]));
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read(FR[f2]), tmp_fp_env);
            FR[f1] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP 例外：

無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド
(Denormal/Unnormal Operand: D)
ソフトウェア・アシスト
(Software Assist: SWA) フォルト
アンダーフロー (Underflow: U)

オーバーフロー (Overflow: O)
不正確結果 (Inexact: I)

ソフトウェア・アシスト
(Software Assist: SWA) トラップ

浮動小数点最大値 (Floating-Point Maximum)

書式: (qp) fmax.sf $f_1 = f_2, f_3$ F8

説明: 2 つのオペランドの値が大きい方が FR f_1 に格納される。FR f_2 が FR f_3 と等しい場合は、FR f_1 には FR f_3 が格納される。

FR f_2 か FR f_3 が NaN である場合は、FR f_1 には FR f_3 が格納される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

この操作は、他の浮動小数点算術演算とは異なり、NaN を伝播させない。fcmp.lt 操作と同じ方法で無効操作 (Invalid Operation) が通知される。

sf のニーモニック値は、7-35 ページの表 7-18 に示してある。

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_bool_res = fp_less_than(fp_reg_read(FR[f3]), fp_reg_read(FR[f2]));
        FR[f1] = (tmp_bool_res ? FR[f2] : FR[f3]);

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP 例外: 無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

浮動小数点マージ (Floating-Point Merge)

書式 :

(qp) fmerge.ns $f_1 = f_2, f_3$	neg_sign_form	F9
(qp) fmerge.s $f_1 = f_2, f_3$	sign_form	F9
(qp) fmerge.se $f_1 = f_2, f_3$	sign_exp_form	F9

説明 : FR f_2 および FR f_3 から、それぞれ符号、指数、および仮数の各フィールドが抽出され、それらがマージされて、その結果が FR f_1 に格納される。

neg_sign_form では、FR f_2 の符号が否定され (つまり反転され)、FR f_3 の指数および仮数と連結される。この形式を使用すると、FR f_2 と FR f_3 に同じレジスタを使用することにより、浮動小数点数の否定を求めることができる。

sign_form では、FR f_2 の符号が FR f_3 の指数および仮数と連結される。

sign_exp_form では、FR f_2 の符号と指数が FR f_3 の仮数と連結される。

すべての形式について、FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

図 7-7. 浮動小数点マージでの符号否定操作

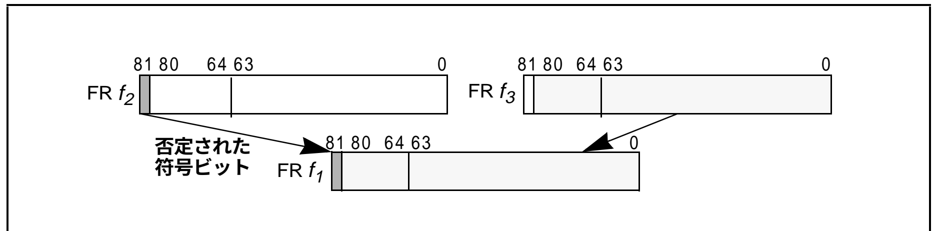


図 7-8. 浮動小数点マージでの符号操作

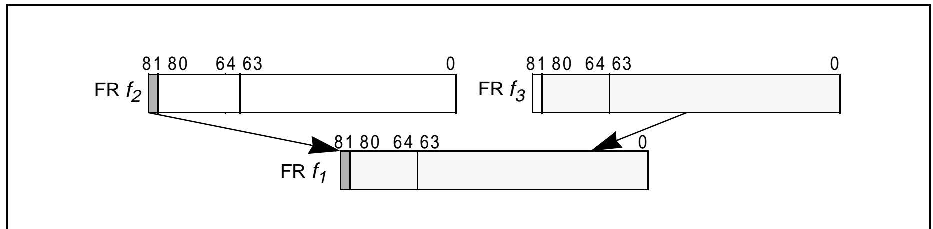
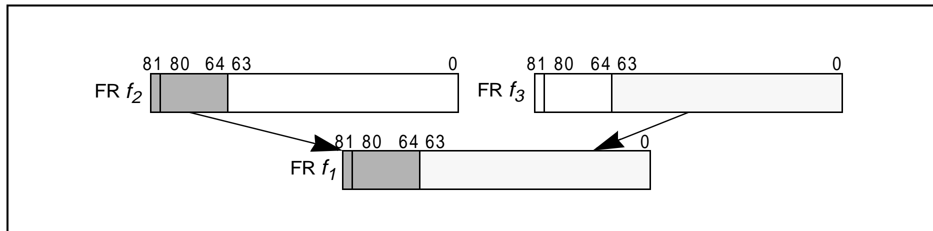


図 7-9. 浮動小数点マージでの符号および指数操作



操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f3].significand;
        if (neg_sign_form) {
            FR[f1].exponent = FR[f3].exponent;
            FR[f1].sign = !FR[f2].sign;
        } else if (sign_form) {
            FR[f1].exponent = FR[f3].exponent;
            FR[f1].sign = FR[f2].sign;
        } else {                                        // sign_exp_form
            FR[f1].exponent = FR[f2].exponent;
            FR[f1].sign = FR[f2].sign;
        }
    }

    fp_update_psr(f1);
}
```

FP 例外: なし。

浮動小数点最小値 (Floating-Point Minimum)

書式 : (*qp*) fmin.*sf* $f_1 = f_2, f_3$ F8

説明 : 2 つのオペランドの値が小さい方が FR f_1 に格納される。FR f_2 が FR f_3 と等しい場合は、FR f_1 には FR f_3 が格納される。

FR f_2 か FR f_3 が NaN である場合は、FR f_1 には FR f_3 が格納される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

この操作は、他の浮動小数点算術演算とは異なり、NaN を伝播させない。fcmp.lt 操作の場合と同じ方法で無効操作 (Invalid Operation) が通知される。

sf のニーモニック値は、7-35 ページの表 7-18 に示してある

操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_bool_res = fp_less_than(fp_reg_read(FR[f2]), fp_reg_read(FR[f3]));
        FR[f1] = tmp_bool_res ? FR[f2] : FR[f3];

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP 例外 : 無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

浮動小数点並列ミックス (Floating-Point Parallel Mix)

書式:

(qp) fmix.l $f_1 = f_2, f_3$	mix_l_form	F9
(qp) fmix.r $f_1 = f_2, f_3$	mix_r_form	F9
(qp) fmix.lr $f_1 = f_2, f_3$	mix_lr_form	F9

説明: mix_l_form (mix_r_form) では、FR f_2 の左側 (右側) の単精度値が FR f_3 の左側 (右側) の単精度値と連結される。mix_lr_form では、FR f_2 の左側の単精度値が FR f_3 の右側の単精度値と連結される。

すべての形式について、FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

すべての形式について、FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

図 7-10. 浮動小数点左ミックス

図 7-11. 浮動小数点右ミックス

図 7-12. 浮動小数点左右ミックス

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        if (mix_l_form) {
            tmp_res_hi = FR[f2].significand{63:32};
            tmp_res_lo = FR[f3].significand{63:32};
        } else if (mix_r_form) {
            tmp_res_hi = FR[f2].significand{31:0};
            tmp_res_lo = FR[f3].significand{31:0};
        } else {                                          // mix_lr_form
            tmp_res_hi = FR[f2].significand{63:32};
            tmp_res_lo = FR[f3].significand{31:0};
        }
        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP 例外: なし。

浮動小数点乗算 (Floating-Point Multiply)

書式: (qp) fmpy.pc.sf $f_1 = f_3, f_4$　　pseudo-op of: (qp) fma.pc.sf $f_1 = f_3, f_4$, f0

説明: FR f_3 と FR f_4 との積が無限の精度で計算される。次に、結果の値が、FPSR.sf.rc によって指定される丸めモードと pc (さらに FPSR.sf.pc および FPSR.sf.wre) によって指定される精度にしたがって丸められ、その結果が FR f_1 に格納される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

オペコードの pc のニーモニック値は 7-35 ページの表 7-17 に示してある。sf のニーモニック値は 7-35 ページの表 7-18 に示してある。ステータス・フィールドの pc、wre、および rc のエンコーディングと意味については、5-7 ページの表 5-4 および 5-9 ページの表 5-5 を参照のこと。

操作: 7-53 ページの「浮動小数点積和 (Floating-Point Multiply Add)」を参照のこと。

浮動小数点積差 (Floating-Point Multiply Subtract)

書式 : (qp) fms.*pc*.*sf* $f_1 = f_3, f_4, f_2$ F1

説明 : FR f_3 と FR f_4 との積が無限の精度で計算され、次に FR f_2 がこの積からやはり無限の精度で減算される。この結果の値が、FPSR.*sf*.*rc* によって指定される丸めモードと *pc* (さらに FPSR.*sf*.*pc* および FPSR.*sf*.*wre*) によって指示される精度にしたがって丸められ、その結果が FR f_1 に格納される。

FR *f3*、FR f_4、および FR f_2 のどれか 1 つでも NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

f_2 が f0 の場合は、積差演算の代わりに IEEE の乗算が行われる。7-60 ページの「浮動小数点乗算 (Floating-Point Multiply)」を参照のこと。

オペコードの *pc* のニーモニック値は 7-35 ページの表 7-17 に示してある。*sf* のニーモニック値は 7-35 ページの表 7-18 に示してある。ステータス・フィールドの *pc*、*wre*、および *rc* のエンコーディングと意味については、5-7 ページの表 5-4 および 5-9 ページの表 5-5 を参照のこと。

操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) || fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fms_fnma_exception_fault_check(f2, f3, f4,
                                                            pc, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[f3]), fp_reg_read(FR[f4]));
            tmp_fr2 = fp_reg_read(FR[f2]);
            tmp_fr2.sign = !tmp_fr2.sign;
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, tmp_fr2, tmp_fp_env);
            FR[f1] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP 例外 :

無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド
(Denormal/Unnormal Operand: D)
ソフトウェア・アシスト
(Software Assist: SWA) フォルト
アンダーフロー
(Underflow: U)

オーバーフロー (Overflow: O)
不正確結果 (Inexact: I)
ソフトウェア・アシスト
(Software Assist: SWA) トラップ

浮動小数点否定 (Floating-Point Negate)

書式: (qp) fneg $f_1 = f_3$ pseudo-op of: (qp) fmerge.ns $f_1 = f_3, f_3$

説明: FR f_3 の値が否定され、FR f_1 に格納される。

FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

操作: 7-55 ページの「浮動小数点マージ (Floating-Point Merge)」を参照のこと。

浮動小数点絶対値の否定 (Floating-Point Negate Absolute Value)

書式: (qp) fnegabs $f_1 = f_3$ pseudo-op of: (qp) fmerge.ns f_1 = f0, f_3

説明: FR f_3 の値の絶対値が計算され、それが否定されて、FR f_1 に格納される。

FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

操作: 7-55 ページの「浮動小数点マージ (Floating-Point Merge)」を参照のこと。

浮動小数点積和の否定 (Floating-Point Negative Multiply Add)

書式 : (qp) fnma.pc.sf $f_1 = f_3, f_4, f_2$ F1

説明 : FR f_3 と FR f_4 との積が無限の精度で計算され、否定され、次に FR f_2 がこの積にやはり無限の精度で加算される。その後、結果の値が、FPSR.*sf.rc* によって指定される丸めモードと *pc* (さらに FPSR.*sf.pc* および FPSR.*sf.wre*) によって指定される精度にしたがって丸められ、その結果が FR f_1 に格納される。

FR f_3、FR f_4、および FR f_2 のどれか 1 つでも NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

f_2 が f0 の場合は、IEEE の乗算が行われた後、その乗算結果が否定される。

オペコードの *pc* のニーモニック値は 7-35 ページの表 7-17 に示してある。*sf* のニーモニック値は 7-35 ページの表 7-18 に示してある。ステータス・フィールドの *pc*、*wre*、および *rc* のエンコーディングと意味については、5-9 ページの表 5-5 および 5-9 ページの表 5-6 を参照のこと。

操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) || fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fms_fnma_exception_fault_check(f2, f3, f4,
                                                            pc, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[f3]), fp_reg_read(FR[f4]));
            tmp_res.sign = !tmp_res.sign;
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read(FR[f2]), tmp_fp_env);
            FR[f1] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP 例外 :

無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト
アンダーフロー (Underflow: U)
オーバーフロー (Overflow: O)
不正確結果 (Inexact: I)
ソフトウェア・アシスト (Software Assist: SWA) トラップ

浮動小数点乗算の否定 (Floating-Point Negative Multiply)

書式: (*qp*) fnmpy.*pc*.*sf* $f_1 = f_3, f_4$ pseudo-op of: (*qp*) fnma.*pc*.*sf* $f_1 = f_3, f_4$,f0

説明: FR f_3 と FR f_4 との積が無限の精度で計算され、次に否定される。その後、結果の値が、FPSR.*sf*.*rc* によって指定される丸めモードと *pc* (さらに FPSR.*sf*.*pc* および FPSR.*sf*.*wre*) によって指定される精度にしたがって丸められ、その結果が FR f_1 に格納される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

オペコードの *pc* のニーモニック値は 7-35 ページの表 7-17 に示してある。*sf* のニーモニック値は 7-35 ページの表 7-18 に示してある。ステータス・フィールドの *pc*、*wre*、および *rc* のエンコーディングと意味については、5-9 ページの表 5-5 および 5-9 ページの表 5-6 を参照のこと。

操作: 7-64 ページの「浮動小数点積和の否定 (Floating-Point Negative Multiply Add)」を参照のこと。

浮動小数点正規化 (Floating-Point Normalize)

書式: (*qp*) fnorm.*pc*.*sf* $f_1 = f_3$ pseudo-op of: (*qp*) fma.*pc*.*sf* $f_1 = f_3, f_1$, f0

説明: FR f_3 がノーマライズされ、FPSR.*sf*.*rc* によって指定される丸めモードと pc (さらに FPSR.*sf*.*pc* および FPSR.*sf*.*wre*) によって指定される精度にしたがって丸められ、その結果が FR f_1 に格納される。

FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

オペコードの *pc* のニーモニック値は 7-35 ページの表 7-17 に示してある。*sf* のニーモニック値は 7-35 ページの表 7-18 に示してある。ステータス・フィールドの *pc*、*wre*、および *rc* のエンコーディングと意味については、5-7 ページの表 5-4 および 5-9 ページの表 5-5 を参照のこと。

操作: 7-53 ページの「浮動小数点積和 (Floating-Point Multiply Add)」を参照のこと。

浮動小数点論理和 (Floating-Point Logical Or)

書式: (*qp*) for $f_1 = f_2, f_3$ F9

説明: FR f_2 および FR f_3 の両仮数フィールド間のビット単位の論理和が計算され、結果の値が FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f2].significand | FR[f3].significand;
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP 例外: なし。

浮動小数点並列絶対値 (Floating-Point Parallel Absolute Value)

書式 : (qp) fpabs $f_1 = f_3$　　pseudo-op of: (qp) fpmerge.s f_1 = f0, f_3

説明 : FR f_3 の仮数フィールド内の単精度値ペアの絶対値が計算され、結果が FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

操作 : 7-82 ページの「浮動小数点並列マージ (Floating-Point Parallel Merge)」を参照のこと。

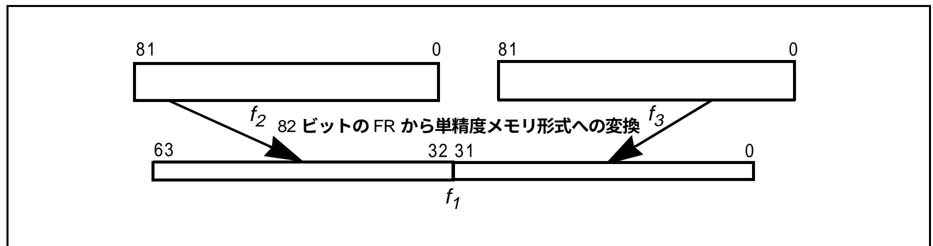
浮動小数点パック (Floating-Point Pack)

書式: (qp) fpack $f_1 = f_2, f_3$ pack_form F9

説明: FR f_2 および FR f_3 に格納されているレジスタ形式の 2 つの数値が単精度のメモリ形式に変換される。これら 2 つの単精度数値が連結され、FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

図 7-13. 浮動小数点パック



操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        tmp_res_hi = fp_single(FR[f2]);
        tmp_res_lo = fp_single(FR[f3]);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }
    fp_update_psr(f1);
}
```

FP 例外: なし。

浮動小数点並列最大絶対値 (Floating-Point Parallel Absolute Maximum)

書式 : (qp) fpamax.sf $f_1 = f_2, f_3$ F8

説明 : FR f_2 および FR f_3 の両仮数フィールドに格納されている単精度値のペアがそれぞれ比較され、絶対値が大きい方のオペランドが FR f_1 の仮数フィールドに返される。

FR f_3 の上 (下) 位の絶対値が FR f_2 の上 (下) 位の絶対値よりも小さい場合は、FR f_1 の上 (下) 位には FR f_2 の上 (下) 位が格納される。そうでない場合は、FR f_1 の上 (下) 位には FR f_3 の上 (下) 位が格納される。

FR f_2 の上 (下) 位または FR f_3 の上 (下) 位が NaN であって、かつ FR f_2 も FR f_3 も NaTVal でない場合は、FR f_1 の上 (下) 位には FR f_3 の上 (下) 位が格納される。

FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

この操作は、他の浮動小数点算術演算とは異なり、NaN を伝播させない。fcmp.lt 操作の場合と同じ方法で無効操作 (Invalid Operation) が通知される。

sf のニーモニック値は、7-35 ページの表 7-18 に示してある。

操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_right = fp_reg_read_hi(f2);
        tmp_fr3 = tmp_left = fp_reg_read_hi(f3);
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        tmp_fr2 = tmp_right = fp_reg_read_lo(f2);
        tmp_fr3 = tmp_left = fp_reg_read_lo(f3);
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
```

```
        }
        fp_update_psr(f1);
}
```

FP 例外: 無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

浮動小数点並列絶対最小値 (Floating-Point Parallel Absolute Minimum)

書式 : (qp) fpamin.sf $f_1 = f_2, f_3$ F8

説明 : FR f_2 および FR f_3 の両仮数フィールドに格納されている単精度値のペアがそれぞれ比較され、絶対値が小さい方のオペランドが FR f_1 の仮数フィールドに返される。

FR f_2 の上 (下) 位の絶対値が FR f_3 の上 (下) 位の絶対値よりも小さい場合は、FR f_1 の上 (下) 位には FR f_2 の上 (下) 位が格納される。そうでない場合は、FR f_1 の上 (下) 位には FR f_3 の上 (下) 位が格納される。

FR f_2 の上 (下) 位または FR f_3 の上 (下) 位が NaN であって、かつ FR f_2 も FR f_3 も NaTVal でない場合は、FR f_1 の上 (下) 位には上 (下) 位の FR f_3 が格納される。

FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

この操作は、他の浮動小数点算術演算とは異なり、NaN を伝播させない。fcmp.lt 操作の場合と同じ方法で無効操作 (Invalid Operation) が通知される。

sf のニーモニック値は、7-35 ページの表 7-18 に示してある。

操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_left = fp_reg_read_hi(f2);
        tmp_fr3 = tmp_right = fp_reg_read_hi(f3);
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        tmp_fr2 = tmp_left = fp_reg_read_lo(f2);
        tmp_fr3 = tmp_right = fp_reg_read_lo(f3);
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
```

```
    }
    fp_update_psr(f1);
}
```

FP 例外 :　無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

浮動小数点並列比較 (Floating-Point Parallel Compare)

書式 : (qp) fpcmp.*frel*.*sf* $f_1 = f_2, f_3$ F8

説明 : FR f_2 および FR f_3 の両仮数フィールドに格納されている単精度ソース・オペランドの各ペアが、frel によって指定される 12 種類の関係のいずれか 1 つについて比較される。この操作によりブール結果が生成される。結果は、比較条件が真である場合は 32 個の "1" からなるマスクになり、偽の場合は 32 個の "0" からなるマスクになる。この結果は、FR f_1 の仮数フィールドに 32 ビット整数のペアとして書き込まれる。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される

表 7-24. 浮動小数点並列比較の結果

PR[*qp*]==0	PR[*qp*]==1		
	結果 ==0、ソースに NaTVal なし	結果 ==1、ソースに NaTVal なし	ソースに NaTVal が 1 つ以上あり
無変更	0...0	1...1	NaTVal

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。

各比較タイプに対して定義されている比較関係の一覧を表 7-25 に示す。12 種類の関係のすべてがハードウェアで直接サポートされているわけではない。一部は実際には擬似オペコードである。それらの関係に対しては、アセンブラは単にソース・オペランド指定子やプレディケート・タイプ指定子を切り換えて、サポートされている関係を利用する。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

表 7-25. 浮動小数点並列比較関係

frel	frel コンプリータの意味	関係	擬似オペコード	オペランドがクワイエット型 NaN の場合の無効通知の有無
eq	等しい	$f_2 == f_3$		なし
lt	より小	$f_2 < f_3$		あり
le	より小か等しい	$f_2 <= f_3$		あり
gt	より大	$f_2 > f_3$	lt $f_2 \leftrightarrow f_3$	あり
ge	より大か等しい	$f_2 >= f_3$	le $f_2 \leftrightarrow f_3$	あり
unord	非順序化	$f_2 ? f_3$		なし
neq	等しくない	$!(f_2 == f_3)$		なし
nlt	より小でない	$!(f_2 < f_3)$		あり
nle	より小でないか等しい	$!(f_2 <= f_3)$		あり

intel®

表 7-25. 浮動小数点並列比較関係 (続き)

frel	frel コンプリータの意味	関係	擬似オペコード	オペランドがクワイエット型 NaN の場合の無効通知の有無
ngt	より大でない	$!(f_2 > f_3)$	nlt $f_2 \leftrightarrow f_3$	あり
nge	より大でないか等しい	$!(f_2 >= f_3)$	nle $f_2 \leftrightarrow f_3$	あり
ord	順序化	$!(f_2 ? f_3)$		なし

操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2,f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpcmp_exception_fault_check(f2, f3, frel, sf, &tmp_fp_env);

        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = fp_reg_read_hi(f2);
        tmp_fr3 = fp_reg_read_hi(f3);

        if      (frel == 'eq') tmp_rel = fp_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'lt') tmp_rel = fp_less_than(tmp_fr2, tmp_fr3);
        else if (frel == 'le') tmp_rel = fp_lesser_or_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'gt') tmp_rel = fp_less_than(tmp_fr3, tmp_fr2);
        else if (frel == 'ge') tmp_rel = fp_lesser_or_equal(tmp_fr3, tmp_fr2);
        else if (frel == 'unord') tmp_rel = fp_unordered(tmp_fr2, tmp_fr3);
        else if (frel == 'neq') tmp_rel = !fp_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'nlt') tmp_rel = !fp_less_than(tmp_fr2, tmp_fr3);
        else if (frel == 'nle') tmp_rel = !fp_lesser_or_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'ngt') tmp_rel = !fp_less_than(tmp_fr3, tmp_fr2);
        else if (frel == 'nge') tmp_rel = !fp_lesser_or_equal(tmp_fr3, tmp_fr2);
        else                         tmp_rel = !fp_unordered(tmp_fr2, tmp_fr3); //
'ord'

        tmp_res_hi = (tmp_rel ? 0xFFFFFFFF : 0x00000000);

        tmp_fr2 = fp_reg_read_lo(f2);
        tmp_fr3 = fp_reg_read_lo(f3);

        if      (frel == 'eq') tmp_rel = fp_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'lt') tmp_rel = fp_less_than(tmp_fr2, tmp_fr3);
        else if (frel == 'le') tmp_rel = fp_lesser_or_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'gt') tmp_rel = fp_less_than(tmp_fr3, tmp_fr2);
        else if (frel == 'ge') tmp_rel = fp_lesser_or_equal(tmp_fr3, tmp_fr2);
        else if (frel == 'unord') tmp_rel = fp_unordered(tmp_fr2, tmp_fr3);
        else if (frel == 'neq') tmp_rel = !fp_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'nlt') tmp_rel = !fp_less_than(tmp_fr2, tmp_fr3);
        else if (frel == 'nle') tmp_rel = !fp_lesser_or_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'ngt') tmp_rel = !fp_less_than(tmp_fr3, tmp_fr2);
        else if (frel == 'nge') tmp_rel = !fp_lesser_or_equal(tmp_fr3, tmp_fr2);
        else                    tmp_rel = !fp_unordered(tmp_fr2, tmp_fr3); //'ord'

        tmp_res_lo = (tmp_rel ? 0xFFFFFFFF : 0x00000000);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
```

```
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP 例外 : 無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

並列浮動小数点から整数への変換 (Convert Parallel Floating-Point to Integer)

書式：

(qp) fpcvt.fx.sf $f_1 = f_2$	signed_form	F10
(qp) fpcvt.fx.trunc.sf $f_1 = f_2$	signed_form, trunc_form	F10
(qp) fpcvt.fxu.sf $f_1 = f_2$	unsigned_form	F10
(qp) fpcvt.fxu.trunc.sf $f_1 = f_2$	unsigned_form, trunc_form	F10

説明：

FR f_2 の仮数フィールド内の単精度値のペアが、FPSR.*sf.rc* によって指定されるも丸めモードを使用して、あるいはこの命令の trunc_form が使用された場合はゼロ側への丸めモードを使用して、符号付き (signed_form の場合) または符号なし (unsigned_form の場合) の 32 ビット整数のペアに変換される。この結果が、32 ビット整数のペアとして FR f_1 の仮数フィールドに書き込まれる。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。IEEE 無効操作浮動小数点例外 (Invalid Operation Floating-Point Exception) フォルトがディスエーブルにされていて、変換の結果が 32 ビット整数に収まらなかった場合は、結果として 32 ビットの不定整数値 0x80000000 が使用される。

FR f_2 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result_pair = fpcvt_exception_fault_check(f2, sf,
                                        signed_form, trunc_form, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan(tmp_default_result_pair.hi)) {
            tmp_res_hi = INTEGER_INDEFINITE_32_BIT;
        } else {
            tmp_res = fp_ieee_rnd_to_int_sp(fp_reg_read_hi(f2), HIGH,
&tmp_fp_env);
            if (tmp_res.exponent)
                tmp_res.significand = fp_U64_rsh(
                    tmp_res.significand, (FP_INTEGER_EXP - tmp_res.exponent));
            if (signed_form && tmp_res.sign)
                tmp_res.significand = (~tmp_res.significand) + 1;

            tmp_res_hi = tmp_res.significand{31:0};
        }

        if (fp_is_nan(tmp_default_result_pair.lo)) {
            tmp_res_lo = INTEGER_INDEFINITE_32_BIT;
        } else {
            tmp_res = fp_ieee_rnd_to_int_sp(fp_reg_read_lo(f2), LOW,
&tmp_fp_env);
            if (tmp_res.exponent)
                tmp_res.significand = fp_U64_rsh(
                    tmp_res.significand, (FP_INTEGER_EXP -
tmp_res.exponent));
            if (signed_form && tmp_res.sign)
                tmp_res.significand = (~tmp_res.significand) + 1;

            tmp_res_lo = tmp_res.significand{31:0};
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP 例外:

無効操作 (Invalid Operation: V)　　不正確結果 (Inexact: I)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

浮動小数点並列積和 (Floating-Point Parallel Multiply Add)

書式: (qp) fpma.*sf* $f_1 = f_3, f_4, f_2$ F1

説明: FR f_3 および FR f_4 の両仮数フィールドに格納されている単精度値のペアのそれぞれの積が無限の精度で計算され、次に、FR f_2 の仮数フィールドの単精度値のペアが、計算結果の 2 つの積にやはり無限の精度で加算される。その後、得られた 2 つの値が FPSR.*sf.rc* によって指定される丸めモードを使用して単精度に丸められる。丸められた結果のペアが FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_3、FR f_4、および FR f_2 のどれか 1 つでも NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

注： `fmpa` 命令の f_2 が f0 の場合は、単に IEEE の乗算が行われる (7-85 ページの「浮動小数点並列乗算 (Floating-Point Parallel Multiply)」を参照のこと)。FR f_1 は、オペランドとしては、パックされた値 1.0 のペアではない。単に、レジスタ・ファイル形式の値 1.0 である。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。
ステータス・フィールドの *rc* のエンコーディングと意味は、5-9 ページの表 5-5 に示してある。

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) || fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result_pair = fpma_exception_fault_check(f2,
                                        f3, f4, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
        } else {
            tmp_res = fp_mul(fp_reg_read_hi(f3), fp_reg_read_hi(f4));
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read_hi(f2), tmp_fp_env);
            tmp_res_hi = fp_ieee_round_sp(tmp_res, HIGH, &tmp_fp_env);
        }

        if (fp_is_nan_or_inf(tmp_default_result_pair.lo)) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
        } else {
            tmp_res = fp_mul(fp_reg_read_lo(f3), fp_reg_read_lo(f4));
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read_lo(f2), tmp_fp_env);
            tmp_res_lo = fp_ieee_round_sp(tmp_res, LOW, &tmp_fp_env);
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
```

```
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP 例外:

無効操作 (Invalid Operation: V)

デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)

ソフトウェア・アシスト (Software Assist: SWA) フォルト

アンダーフロー (Underflow: U)

オーバーフロー (Overflow: O)

不正確結果 (Inexact: I)

ソフトウェア・アシスト (Software Assist: SWA) トラップ

浮動小数点並列最大値 (Floating-Point Parallel Maximum)

書式 : (qp) fpmax.sf $f1 = f_2, f_3$ F8

説明 : FR f_2 および FR f_3 の両仮数フィールドに格納されている単精度値のペアがそれぞれ比較される。値が大きい方のオペランドが FR f_1 の仮数フィールドに返される。

FR f_3 の上 (下) 位の値が FR f_2 の上 (下) 位の値よりも小さい場合は、FR f_1 の上 (下) 位には FR f_2 の上 (下) 位が格納される。そうでない場合は、FR f_1 の上 (下) 位には FR f_3 の上 (下) 位が格納される。

FR f_2 の上 (下) 位または FR f_3 の上 (下) 位が NaN である場合は、FR f_1 の上 (下) 位にはの FR f_3 の上 (下) 位が格納される。

FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

この操作は、他の浮動小数点算術演算とは異なり、NaN を伝播させない。fcmp.lt 操作の場合と同じ方法で無効操作 (Invalid Operation) が通知される。

sf のニーモニック値は、7-35 ページの表 7-18 に示してある。

操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_right = fp_reg_read_hi(f2);
        tmp_fr3 = tmp_left = fp_reg_read_hi(f3);
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2 : tmp_fr3);

        tmp_fr2 = tmp_right = fp_reg_read_lo(f2);
        tmp_fr3 = tmp_left = fp_reg_read_lo(f3);
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP 例外 : 無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

浮動小数点並列マージ (Floating-Point Parallel Merge)

書式 :

(qp) fpmerge.ns $f_1 = f_2, f_3$	neg_sign_form	F9
(qp) fpmerge.s $f_1 = f_2, f_3$	sign_form	F9
(qp) fpmerge.se $f_1 = f_2, f_3$	sign_exp_form	F9

説明 : neg_sign_form では、FR f_2 の仮数フィールド内の単精度値ペアの両符号が否定され、FR f_3 の仮数フィールド内の単精度値ペアの両指数および両仮数と連結され、FR f_1 の仮数フィールドに格納される。この形式を使用すると、f_2 と f$_3$ に同じレジスタを使用して、単精度浮動小数点数ペアの否定を得ることができる。

sign_from では、FR f_2 の仮数フィールド内の単精度値ペアの両符号が、FR f_3 の仮数フィールド内の単精度値ペアの両指数および両仮数と連結され、FR f_1 の仮数フィールドに格納される。

sign_exp_form では、FR f_2 の仮数フィールド内の単精度値ペアの両符号および両指数が、FR f_3 の仮数フィールド内の単精度値ペアのそれぞれの単精度仮数と連結され、FR f_1 の仮数フィールドに格納される。

すべての形式について、FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

すべての形式について、FR f_2 か FR f_3 が NaTVal である場合は、FR $f1$ は計算結果ではなく NaTVal に設定される。

図 7-14. 浮動小数点並列マージでの符号否定操作

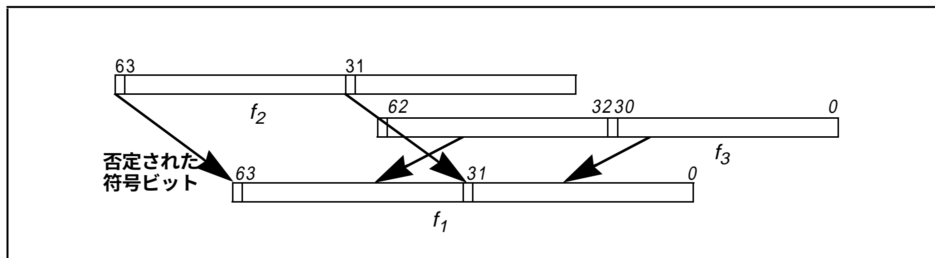


図 7-15. 浮動小数点並列マージでの符号操作

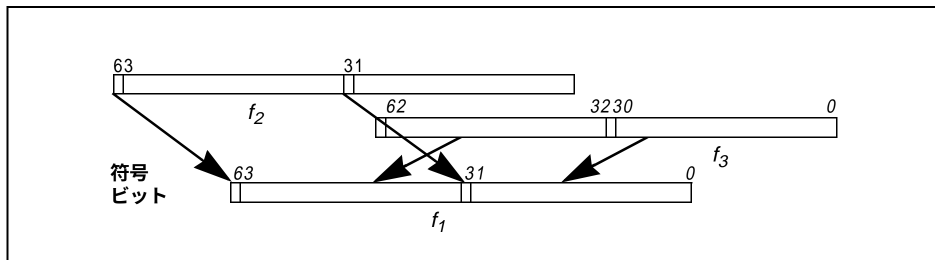
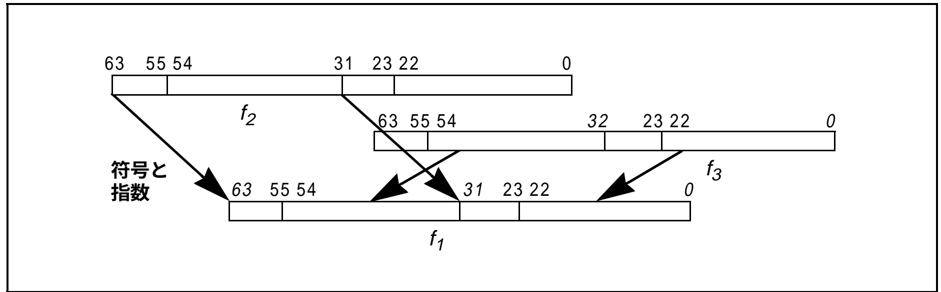


図 7-16. 浮動小数点並列マージでの符号および指数操作



操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        if (neg_sign_form) {
            tmp_res_hi = (!FR[f2].significand{63} << 31)
                        | (FR[f3].significand{62:32});
            tmp_res_lo = (!FR[f2].significand{31} << 31)
                        | (FR[f3].significand{30:0});
        } else if (sign_form) {
            tmp_res_hi = (FR[f2].significand{63} << 31)
                        | (FR[f3].significand{62:32});
            tmp_res_lo = (FR[f2].significand{31} << 31)
                        | (FR[f3].significand{30:0});
        } else {                                            // sign_exp_form
            tmp_res_hi = (FR[f2].significand{63:55} << 23)
                        | (FR[f3].significand{54:32});
            tmp_res_lo = (FR[f2].significand{31:23} << 23)
                        | (FR[f3].significand{22:0});
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP 例外 : なし。

浮動小数点並列最小値 (Floating-Point Parallel Minimum)

書式 : (qp) fpmin.sf $f_1 = f_2, f_3$ F8

説明 : FR f_2 および FR f_3 の両仮数フィールドに格納されている単精度値のペアがそれぞれ比較される。値が小さい方のオペランドが FR f_1 の仮数フィールドに返される。

FR f_2 の上 (下) 位の値が FR f_3 の上 (下) 位の値よりも小さい場合は、FR f_1 の上 (下) 位には FR f_2 の上 (下) 位が格納される。そうでない場合は、FR f_1 の上 (下) 位には FR f_3 の上 (下) 位が格納される。

FR f_2 の上 (下) 位または FR f_3 の上 (下) 位が NaN である場合は、FR f_1 の上 (下) 位には FR f_3 の上 (下) 位が格納される。

FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

この操作は、他の浮動小数点算術演算とは異なり、NaN を伝播させない。fcmp.lt 操作の場合と同じ方法で無効操作 (Invalid Operation) が通知される。

sf のニーモニック値は、7-35 ページの表 7-18 に示してある。

操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_left = fp_reg_read_hi(f2);
        tmp_fr3 = tmp_right = fp_reg_read_hi(f3);
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        tmp_fr2 = tmp_left = fp_reg_read_lo(f2);
        tmp_fr3 = tmp_right = fp_reg_read_lo(f3);
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP 例外 : 無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

浮動小数点並列乗算 (Floating-Point Parallel Multiply)

書式 : (qp) fpmpy.*sf* $f_1 = f_3, f_4$ pseudo-op of: (qp) fpma.*sf* $f_1 = f_3, f_4$, f0

説明 : FR f_3 および FR f_4 の両仮数フィールドに格納されている単精度値ペア間の積が無限の精度で計算される。次に、それぞれの結果の値が、FPSR.*sf.rc* によって指定される丸めモードを使用して、単精度に丸められる。丸められた結果のペアが FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_3 か FR f_4 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。
ステータス・フィールドの *rc* のエンコーディングと意味は、5-9 ページの表 5-5 に示してある。

操作 : 7-79 ページの「浮動小数点並列積和 (Floating-Point Parallel Multiply Add)」を参照のこと。

浮動小数点並列積差 (Floating-Point Parallel Multiply Subtract)

書式: (qp) fpms.sf $f_1 = f_3, f_4, f_2$ F1

説明: FR f_3 および FR f_4 の両仮数フィールドに格納されている単精度値ペア間の積が無限の精度で計算され、次にそれぞれの積から FR f_2 の仮数フィールド内の単精度値ペアがやはり無限の精度で減算される。次に、それぞれの結果の値が、FPSR.$sf.rc$ によって指定される丸めモードを使用して単精度に丸められる。丸められた結果のペアが FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_3 か FR f_4 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

注： fmps 命令の f_2 が f0 の場合は、単に IEEE の乗算が実行される。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。
ステータス・フィールドの rc のエンコーディングと解釈は、5-9 ページの表 5-5 に示してある。

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result_pair = fpms_fpnma_exception_fault_check(f2, f3,
f4,
                                                        sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
        } else {
            tmp_res = fp_mul(fp_reg_read_hi(f3), fp_reg_read_hi(f4));
            if (f2 != 0) {
                tmp_sub = fp_reg_read_hi(f2);
                tmp_sub.sign = !tmp_sub.sign;
                tmp_res = fp_add(tmp_res, tmp_sub, tmp_fp_env);
            }
            tmp_res_hi = fp_ieee_round_sp(tmp_res, HIGH, &tmp_fp_env);
        }

        if (fp_is_nan_or_inf(tmp_default_result_pair.lo)) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
        } else {
            tmp_res = fp_mul(fp_reg_read_lo(f3), fp_reg_read_lo(f4));
            if (f2 != 0) {
                tmp_sub = fp_reg_read_lo(f2);
                tmp_sub.sign = !tmp_sub.sign;
                tmp_res = fp_add(tmp_res, tmp_sub, tmp_fp_env);
            }
            tmp_res_lo = fp_ieee_round_sp(tmp_res, LOW, &tmp_fp_env);
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
```

```
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP 例外:

無効操作 (Invalid Operation: V)

デノーマル / アンノーマル・オペランド)
(Denormal/Unnormal Operand: D)

ソフトウェア・アシスト
(Software Assist: SWA) フォルト

アンダーフロー (Underflow: U)

オーバーフロー (Overflow: O

不正確結果 (Inexact: I)

ソフトウェア・アシスト
(Software Assist: SWA) トラップ

浮動小数点並列否定 (Floating-Point Parallel Negate)

書式: (qp) fpneg $f_1 = f_3$ pseudo-op of: (qp) fpmerge.ns $f_1 = f_3, f_3$

説明: FR f_3 の仮数フィールド内の単精度値のペアが否定され、FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

操作: 7-82 ページの「浮動小数点並列マージ (Floating-Point Parallel Merge)」を参照のこと。

浮動小数点並列絶対値否定 (Floating-Point Parallel Negate Absolute Value)

書式： (*qp*) fpnegabs $f_1 = f_3$ pseudo-op of: (*qp*) fpmerge.ns f_1 = f0, f_3

説明： FR f_3 の仮数フィールド内の単精度値ペアの絶対値が計算され、否定され、次に FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

操作： 7-82 ページの「浮動小数点並列マージ (Floating-Point Parallel Merge)」を参照のこと。

浮動小数点並列積和否定 (Floating-Point Parallel Negative Multiply Add)

書式: (*qp*) fpnma.*sf* $f_1 = f_3, f_4, f_2$ F1

説明: FR f_3 および FR f_4 の両仮数フィールドに格納されている単精度値ペア間の積が最大精度まで計算され、否定され、次に FR f_2 の仮数フィールド内の単精度値ペアが、それぞれの否定された積にやはり無限の精度で加算される。その後、加算結果のそれぞれの値が FPSR.*sf*.*rc* によって指定される丸めモードを使用して単精度に丸められる。丸められた結果のペアが FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_3、FR f_4、および FR f_2 のどれか 1 つでも NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

> **注：** fmnma 命令の f_2 が f0 の場合は、単に IEEE の乗算が実行される (積が否定された後に丸められる)。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。
ステータス・フィールドの *rc* のエンコーディングと意味は、5-9 ページの表 5-5 に示してある。

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) || fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result_pair = fpms_fpnma_exception_fault_check(f2, f3,
f4,
                                                       sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
        } else {
            tmp_res = fp_mul(fp_reg_read_hi(f3), fp_reg_read_hi(f4));
            tmp_res.sign = !tmp_res.sign;
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read_hi(f2), tmp_fp_env);
            tmp_res_hi = fp_ieee_round_sp(tmp_res, HIGH, &tmp_fp_env);
        }

        if (fp_is_nan_or_inf(tmp_default_result_pair.lo)) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
        } else {
            tmp_res = fp_mul(fp_reg_read_lo(f3), fp_reg_read_lo(f4));
            tmp_res.sign = !tmp_res.sign;
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read_lo(f2), tmp_fp_env);
            tmp_res_lo = fp_ieee_round_sp(tmp_res, LOW, &tmp_fp_env);
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
```

```
            fp_update_fpsr(sf, tmp_fp_env);
            fp_update_psr(f1);
            if (fp_raise_traps(tmp_fp_env))
                fp_exception_trap(fp_decode_trap(tmp_fp_env));
        }
    }
```

FP 例外 :

無効操作 (Invalid Operation: V)

デノーマル / アンノーマル・オペランド
(Denormal/Unnormal Operand: D)

ソフトウェア・アシスト
(Software Assist: SWA) フォルト

アンダーフロー (Underflow: U)

オーバーフロー (Overflow: O)

不正確結果 js (Inexact: I)

ソフトウェア・アシスト
(Software Assist: SWA) トラップ

浮動小数点並列乗算否定 (Floating-Point Parallel Negative Multiply)

書式: (qp) fpnmpy.sf $f_1 = f_3, f_4$ pseudo-op of: (qp) fpnma.sf $f_1 = f_3, f_4$,f0

説明: FR f_3 および FR f_4 の両仮数フィールドに格納されている単精度値ペア間の積が最大精度まで計算され、次に否定される。その後、それぞれの結果の値が FPSR.$sf.rc$ によって指定される丸めモードを使用して単精度に丸められる。丸められた結果のペアが FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_3 か FR f_4 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。
ステータス・フィールドの rc のエンコーディングと意味は、5-6 ページの表 5-5 に示してある。

操作: 7-90 ページの「浮動小数点並列積和否定 (Floating-Point Parallel Negative Multiply Add)」を参照のこと。

浮動小数点並列逆数近似 (Floating-Point Parallel Reciprocal Approximation)

書式: (qp) fprcpa.sf f_1,p_2 = f_2, f_3 F6

説明: PR qp が 0 の場合、RP p_2 がクリアされる。FR f_1 はそのまま変わらない。

PR qp が 1 の場合、以下のことが行われる。

- FR f_1 の仮数の上下各半分が、FR f_3 の対応する半分の逆数の近似値 (相対誤差 $<2^{-8.886}$) に設定される。あるいは、FR f_2 または FR f_3 の対応する半分が {- ∞、-0、+0、+ ∞、NaN} の集合に属している場合は、その対応する半分の FR f_2/FR f_3 の商に対する IEEE 754 準拠の応答に設定される。
- FR f_1 のいずれかの半分が IEEE 754 準拠の商に設定された場合、あるいは、ニュートン・ラフソンの反復計算で IEEE 754 準拠の正しい除算結果を生成できないような逆数の近似値に設定された場合は、PR p_2 が 0 に設定され、そうでない場合は、1 に設定される。

  正しい IEEE 除算結果を保証するために、RP p_2 がクリアされるときは、ユーザ・ソフトウェア上で、(並列でない `frcpa` 命令を使用して) 各半分の商 (FR f_2/FR f_3) を計算し、それぞれの結果をマージして FR f_1 に格納し、RP p_2 はクリア状態に維持することが望まれる。
- FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。
- FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定され、PR p_2 はクリアされる。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
        PR[p2] = 0;
    } else {
        tmp_default_result_pair = fprcpa_exception_fault_check(f2, f3, sf,
                                                  &tmp_fp_env, &limits_check);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result_pair.hi) || limits_check.hi_fr3) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
            tmp_pred_hi = 0;
        } else {
            num = fp_normalize(fp_reg_read_hi(f2));
            den = fp_normalize(fp_reg_read_hi(f3));
            if (fp_is_inf(num) && fp_is_finite(den)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_hi = 0;
            } else if (fp_is_finite(num) && fp_is_inf(den)) {
                tmp_res = FP_ZERO;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_hi = 0;
            } else if (fp_is_zero(num) && fp_is_finite(den)) {
```

```
                tmp_res = FP_ZERO;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_hi = 0;
            } else {
                tmp_res = fp_ieee_recip(den);
                if (limits_check.hi_fr2_or_quot)
                    tmp_pred_hi = 0;
                else
                    tmp_pred_hi = 1;
            }
            tmp_res_hi = fp_single(tmp_res);
        }
        if (fp_is_nan_or_inf(tmp_default_result_pair.lo) || limits_check.lo_fr3) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
            tmp_pred_lo = 0;
        } else {
            num = fp_normalize(fp_reg_read_lo(f2));
            den = fp_normalize(fp_reg_read_lo(f3));
            if (fp_is_inf(num) && fp_is_finite(den)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_lo = 0;
            } else if (fp_is_finite(num) && fp_is_inf(den)) {
                tmp_res = FP_ZERO;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_lo = 0;
            } else if (fp_is_zero(num) && fp_is_finite(den)) {
                tmp_res = FP_ZERO;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_lo = 0;
            } else {
                tmp_res = fp_ieee_recip(den);
                if (limits_check.lo_fr2_or_quot)
                    tmp_pred_lo = 0;
                else
                    tmp_pred_lo = 1;
            }
            tmp_res_lo = fp_single(tmp_res);
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
        PR[p2] = tmp_pred_hi && tmp_pred_lo;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
}
```

FP 例外: 無効操作 (Invalid Operation: V)
ゼロ除算 (Zero Divide: Z)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

intel®

浮動小数点並列平方根逆数近似 (Floating-Point Parallel Reciprocal Square Root Approximation)

書式： (qp) fprsqrta.*sf* f_1,p_2 = f_3 F7

説明： PR *qp* が 0 の場合、RP p_2 がクリアされる。FR f_1 はそのまま変わらない。

PR *qp* が 1 の場合、以下のことが行われる。

- FR f_1 の仮数の上下各半分が、FR f_3 の対応する半分の平方根の逆数の近似値 (相対誤差 <$2^{-8.831}$) に設定される。あるいは、FR f_3 の対応する半分が {- ∞、- 有限数、-0、+0、+ ∞、NaN} の集合に属している場合は、FR f_3 のその対応する半分の平方根の逆数に対する IEEE 754 準拠の応答に設定される。
- FR f_1 のいずれかの半分が、IEEE 754 準拠の平方根の逆数に設定された場合、あるいは、ニュートン・ラフソンの反復計算で IEEE 754 準拠の正しい平方根結果を生成できないような平方根の逆数の近似値に設定された場合は、PR p_2 が 0 に設定され、そうでない場合は、1 に設定される。

  正しい IEEE の平方根結果を保証するために、RP p_2 がクリアされるときは、ユーザ・ソフトウェア上で、(非並列の frsqrta 命令を使用して) 各半分の平方根を計算し、それぞれの結果をマージして FR f_1 に格納し、RP p_2 はクリア状態に維持することが望まれる。
- FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。
- FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定され、PR p_2 はクリアされる。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。

操作：

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f3, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
        PR[p2] = 0;
    } else {
        tmp_default_result_pair = fprsqrta_exception_fault_check(f3, sf,
                                                        &tmp_fp_env, &limits_check);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
            tmp_pred_hi = 0;
        } else {
            tmp_fr3 = fp_normalize(fp_reg_read_hi(f3));
            if (fp_is_zero(tmp_fr3)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = tmp_fr3.sign;
                tmp_pred_hi = 0;
            } else if (fp_is_pos_inf(tmp_fr3)) {
                tmp_res = FP_ZERO;
                tmp_pred_hi = 0;
            } else {
```

```
                tmp_res = fp_ieee_recip_sqrt(tmp_fr3);
                if (limits_check.hi)
                    tmp_pred_hi = 0;
                else
                    tmp_pred_hi = 1;
            }
            tmp_res_hi = fp_single(tmp_res);
        }

        if (fp_is_nan(tmp_default_result_pair.lo)) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
            tmp_pred_lo = 0;
        } else {
            tmp_fr3 = fp_normalize(fp_reg_read_lo(f3));
            if (fp_is_zero(tmp_fr3)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = tmp_fr3.sign;
                tmp_pred_lo = 0;
            } else if (fp_is_pos_inf(tmp_fr3)) {
                tmp_res = FP_ZERO;
                tmp_pred_lo = 0;
            } else {
                tmp_res = fp_ieee_recip_sqrt(tmp_fr3);
                if (limits_check.lo)
                    tmp_pred_lo = 0;
                else
                    tmp_pred_lo = 1;
            }
            tmp_res_lo = fp_single(tmp_res);
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
        PR[p2] = tmp_pred_hi && tmp_pred_lo;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
}
```

FP 例外: 無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

浮動小数点逆数近似 (Floating-Point Reciprocal Approximation)

書式: (qp) frcpa.sf $f_1, p_2 = f_2, f_3$ F6

説明: PR qp が 0 の場合、RP p_2 がクリアされる。FR f_1 はそのまま変わらない。

PR qp が 1 の場合、以下のことが行われる。

- FR f_1 が、FR f_3 の逆数の近似値 (相対誤差 $<2^{-8.886}$) に設定される。あるいは、FR f_2 か FR f_3 が {- ∞、-0、擬似ゼロ、+0、+ ∞、非サポート数 } の集合に属している場合は、FR f_2/FR f_3 の IEEE 754 準拠の商に設定される。
- FR f_1 が FR f_3 の逆数の近似値に設定された場合は、PR p_2 が 1 に設定され、そうでない場合は、0 に設定される。
- FR f_3 の逆数の近似によって、ニュートン・ラフソンの反復計算で FR f_2/FR f_3 に対して IEEE 754 準拠の正しい結果を生成できないような場合は、ソフトウェア・アシストを要求する浮動小数点例外 (Floating-point Exception) フォルトが発生する。

  システム・ソフトウェア上で、IEEE 754 準拠の商 (FR f_2/FR f_3) を計算し、その結果を FR f_1 に返し、PR p_2 を 0 に設定することが望まれる。
- FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定され、PR p_2 はクリアされる。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
        PR[p2] = 0;
    } else {
        tmp_default_result = frcpa_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
            PR[p2] = 0;
        } else {
            num = fp_normalize(fp_reg_read(FR[f2]));
            den = fp_normalize(fp_reg_read(FR[f3]));
            if (fp_is_inf(num) && fp_is_finite(den)) {
                FR[f1] = FP_INFINITY;
                FR[f1].sign = num.sign ^ den.sign;
                PR[p2] = 0;
            } else if (fp_is_finite(num) && fp_is_inf(den)) {
                FR[f1] = FP_ZERO;
                FR[f1].sign = num.sign ^ den.sign;
                PR[p2] = 0;
            } else if (fp_is_zero(num) && fp_is_finite(den)) {
                FR[f1] = FP_ZERO;
                FR[f1].sign = num.sign ^ den.sign;
                PR[p2] = 0;
            } else {
                FR[f1] = fp_ieee_recip(den);
                PR[p2] = 1;
            }
```

```
        }
        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
}

// fp_ieee_recip()

fp_ieee_recip(den)
{
    const EM_uint_t RECIP_TABLE[256] = {
        0x3fc, 0x3f4, 0x3ec, 0x3e4, 0x3dd, 0x3d5, 0x3cd, 0x3c6,
        0x3be, 0x3b7, 0x3af, 0x3a8, 0x3a1, 0x399, 0x392, 0x38b,
        0x384, 0x37d, 0x376, 0x36f, 0x368, 0x361, 0x35b, 0x354,
        0x34d, 0x346, 0x340, 0x339, 0x333, 0x32c, 0x326, 0x320,
        0x319, 0x313, 0x30d, 0x307, 0x300, 0x2fa, 0x2f4, 0x2ee,
        0x2e8, 0x2e2, 0x2dc, 0x2d7, 0x2d1, 0x2cb, 0x2c5, 0x2bf,
        0x2ba, 0x2b4, 0x2af, 0x2a9, 0x2a3, 0x29e, 0x299, 0x293,
        0x28e, 0x288, 0x283, 0x27e, 0x279, 0x273, 0x26e, 0x269,
        0x264, 0x25f, 0x25a, 0x255, 0x250, 0x24b, 0x246, 0x241,
        0x23c, 0x237, 0x232, 0x22e, 0x229, 0x224, 0x21f, 0x21b,
        0x216, 0x211, 0x20d, 0x208, 0x204, 0x1ff, 0x1fb, 0x1f6,
        0x1f2, 0x1ed, 0x1e9, 0x1e5, 0x1e0, 0x1dc, 0x1d8, 0x1d4,
        0x1cf, 0x1cb, 0x1c7, 0x1c3, 0x1bf, 0x1bb, 0x1b6, 0x1b2,
        0x1ae, 0x1aa, 0x1a6, 0x1a2, 0x19e, 0x19a, 0x197, 0x193,
        0x18f, 0x18b, 0x187, 0x183, 0x17f, 0x17c, 0x178, 0x174,
        0x171, 0x16d, 0x169, 0x166, 0x162, 0x15e, 0x15b, 0x157,
        0x154, 0x150, 0x14d, 0x149, 0x146, 0x142, 0x13f, 0x13b,
        0x138, 0x134, 0x131, 0x12e, 0x12a, 0x127, 0x124, 0x120,
        0x11d, 0x11a, 0x117, 0x113, 0x110, 0x10d, 0x10a, 0x107,
        0x103, 0x100, 0x0fd, 0x0fa, 0x0f7, 0x0f4, 0x0f1, 0x0ee,
        0x0eb, 0x0e8, 0x0e5, 0x0e2, 0x0df, 0x0dc, 0x0d9, 0x0d6,
        0x0d3, 0x0d0, 0x0cd, 0x0ca, 0x0c8, 0x0c5, 0x0c2, 0x0bf,
        0x0bc, 0x0b9, 0x0b7, 0x0b4, 0x0b1, 0x0ae, 0x0ac, 0x0a9,
        0x0a6, 0x0a4, 0x0a1, 0x09e, 0x09c, 0x099, 0x096, 0x094,
        0x091, 0x08e, 0x08c, 0x089, 0x087, 0x084, 0x082, 0x07f,
        0x07c, 0x07a, 0x077, 0x075, 0x073, 0x070, 0x06e, 0x06b,
        0x069, 0x066, 0x064, 0x061, 0x05f, 0x05d, 0x05a, 0x058,
        0x056, 0x053, 0x051, 0x04f, 0x04c, 0x04a, 0x048, 0x045,
        0x043, 0x041, 0x03f, 0x03c, 0x03a, 0x038, 0x036, 0x033,
        0x031, 0x02f, 0x02d, 0x02b, 0x029, 0x026, 0x024, 0x022,
        0x020, 0x01e, 0x01c, 0x01a, 0x018, 0x015, 0x013, 0x011,
        0x00f, 0x00d, 0x00b, 0x009, 0x007, 0x005, 0x003, 0x001,
    };

    tmp_index = den.significand{62:55};
    tmp_res.significand = (1 << 63) | (RECIP_TABLE[tmp_index] << 53);
    tmp_res.exponent = FP_REG_EXP_ONES - 2 - den.exponent;
    tmp_res.sign = den.sign;
    return (tmp_res);
}
```

FP 例外:

無効操作 (Invalid Operation: V)
ゼロ除算 (Zero Divide: Z)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

浮動小数点平方根逆数近似 (Floating-Point Reciprocal Square Root Approximation)

書式: (qp) frsqrta.sf f_1, p_2 = f_3 F7

説明: PR qp が 0 の場合、RP p_2 がクリアされる。FR f_1 はそのまま変わらない。

PR qp が 1 の場合、以下のことが行われる。

- FR f_1 が、FR f_3 の平方根の逆数の近似値 (相対誤差 $<2^{-8.831}$) に設定される。あるいは、FR f_3 が {- ∞、- 有限数、-0、擬似ゼロ、+0、+ ∞、NaN、非サポート数 } の集合に属している場合は、FR f_3 の IEEE 754 準拠の平方根に設定される。
- FR f_1 が FR f_3 の平方根の逆数の近似値に設定された場合は、PR p_2 が 1 に設定され、そうでない場合は、0 に設定される。
- FR f_3 の平方根の逆数の近似によって、ニュートン・ラフソンの反復計算で IEEE 754 準拠の正しい平方根結果を生成できないような場合は、ソフトウェア・アシストを必要とする小数点例外 (Floating-point Exception) フォルトが発生する。

  システム・ソフトウェア上で、IEEE 754 準拠の平方根を計算し、その結果を FR f_1 に返し、PR p_2 に 0 を設定することが望まれる。
- FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定され、PR p_2 はクリアされる。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f3, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
        PR[p2] = 0;
    } else {
        tmp_default_result = frsqrta_exception_fault_check(f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan(tmp_default_result)) {
            FR[f1] = tmp_default_result;
            PR[p2] = 0;
        } else {
            tmp_fr3 = fp_normalize(fp_reg_read(FR[f3]));
            if (fp_is_zero(tmp_fr3)) {
                FR[f1] = tmp_fr3;
                PR[p2] = 0;
            } else if (fp_is_pos_inf(tmp_fr3)) {
                FR[f1] = tmp_fr3;
                PR[p2] = 0;
            } else {
                FR[f1] = fp_ieee_recip_sqrt(tmp_fr3);
                PR[p2] = 1;
            }
        }
        fp_update_fpsr(sf, tmp_fp_env);
    }
```

```
        fp_update_psr(f1);
    } else {
        PR[p2] = 0;
    }

    // fp_ieee_recip_sqrt()

    fp_ieee_recip_sqrt(root)
    {
        const EM_uint_t RECIP_SQRT_TABLE[256] = {
            0x1a5, 0x1a0, 0x19a, 0x195, 0x18f, 0x18a, 0x185, 0x180,
            0x17a, 0x175, 0x170, 0x16b, 0x166, 0x161, 0x15d, 0x158,
            0x153, 0x14e, 0x14a, 0x145, 0x140, 0x13c, 0x138, 0x133,
            0x12f, 0x12a, 0x126, 0x122, 0x11e, 0x11a, 0x115, 0x111,
            0x10d, 0x109, 0x105, 0x101, 0x0fd, 0x0fa, 0x0f6, 0x0f2,
            0x0ee, 0x0ea, 0x0e7, 0x0e3, 0x0df, 0x0dc, 0x0d8, 0x0d5,
            0x0d1, 0x0ce, 0x0ca, 0x0c7, 0x0c3, 0x0c0, 0x0bd, 0x0b9,
            0x0b6, 0x0b3, 0x0b0, 0x0ad, 0x0a9, 0x0a6, 0x0a3, 0x0a0,
            0x09d, 0x09a, 0x097, 0x094, 0x091, 0x08e, 0x08b, 0x088,
            0x085, 0x082, 0x07f, 0x07d, 0x07a, 0x077, 0x074, 0x071,
            0x06f, 0x06c, 0x069, 0x067, 0x064, 0x061, 0x05f, 0x05c,
            0x05a, 0x057, 0x054, 0x052, 0x04f, 0x04d, 0x04a, 0x048,
            0x045, 0x043, 0x041, 0x03e, 0x03c, 0x03a, 0x037, 0x035,
            0x033, 0x030, 0x02e, 0x02c, 0x029, 0x027, 0x025, 0x023,
            0x020, 0x01e, 0x01c, 0x01a, 0x018, 0x016, 0x014, 0x011,
            0x00f, 0x00d, 0x00b, 0x009, 0x007, 0x005, 0x003, 0x001,
            0x3fc, 0x3f4, 0x3ec, 0x3e5, 0x3dd, 0x3d5, 0x3ce, 0x3c7,
            0x3bf, 0x3b8, 0x3b1, 0x3aa, 0x3a3, 0x39c, 0x395, 0x38e,
            0x388, 0x381, 0x37a, 0x374, 0x36d, 0x367, 0x361, 0x35a,
            0x354, 0x34e, 0x348, 0x342, 0x33c, 0x336, 0x330, 0x32b,
            0x325, 0x31f, 0x31a, 0x314, 0x30f, 0x309, 0x304, 0x2fe,
            0x2f9, 0x2f4, 0x2ee, 0x2e9, 0x2e4, 0x2df, 0x2da, 0x2d5,
            0x2d0, 0x2cb, 0x2c6, 0x2c1, 0x2bd, 0x2b8, 0x2b3, 0x2ae,
            0x2aa, 0x2a5, 0x2a1, 0x29c, 0x298, 0x293, 0x28f, 0x28a,
            0x286, 0x282, 0x27d, 0x279, 0x275, 0x271, 0x26d, 0x268,
            0x264, 0x260, 0x25c, 0x258, 0x254, 0x250, 0x24c, 0x249,
            0x245, 0x241, 0x23d, 0x239, 0x235, 0x232, 0x22e, 0x22a,
            0x227, 0x223, 0x220, 0x21c, 0x218, 0x215, 0x211, 0x20e,
            0x20a, 0x207, 0x204, 0x200, 0x1fd, 0x1f9, 0x1f6, 0x1f3,
            0x1f0, 0x1ec, 0x1e9, 0x1e6, 0x1e3, 0x1df, 0x1dc, 0x1d9,
            0x1d6, 0x1d3, 0x1d0, 0x1cd, 0x1ca, 0x1c7, 0x1c4, 0x1c1,
            0x1be, 0x1bb, 0x1b8, 0x1b5, 0x1b2, 0x1af, 0x1ac, 0x1aa,
        };

        tmp_index = (root.exponent{0} << 7) | root.significand{62:56};
        tmp_res.significand = (1 << 63) | (RECIP_SQRT_TABLE[tmp_index] << 53);
        tmp_res.exponent = FP_REG_EXP_HALF - ((root.exponent - FP_REG_BIAS) >> 1);
        tmp_res.sign = FP_SIGN_POSITIVE;
        return (tmp_res);
    }
```

FP 例外: 無効操作 (Invalid Operation: V)
デノーマル / アンノーマル・オペランド (Denormal/Unnormal Operand: D)
ソフトウェア・アシスト (Software Assist: SWA) フォルト

浮動小数点選択 (Floating-Point Select)

書式: (*qp*) fselect $f_1 = f_3, f_4, f_2$ F3

説明: FR f_3 の仮数フィールドと FR f_2 の仮数フィールドとの論理積が取られ、FR f_4 の仮数フィールドと FR f_2 の仮数フィールドの "1" の補数との論理積が取られる。次に、2 つの結果間の論理和が取られ、その結果が FR f_1 の仮数フィールドに格納される。

FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_3、FR f_4、および FR f_2 のどれか 1 つでも NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) || fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand          = (FR[f3].significand & FR[f2].significand)
                                    | (FR[f4].significand & ~FR[f2].significand);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP 例外: なし。

浮動小数点コントロール設定 (Floating-Point Set Controls)

書式 : (*qp*) fsetc.*sf* *amask7, omask7* F12

説明 : sf0.controls と *amask7* 即値フィールドとの論理積を取り、その結果と *omask7* 即値フィールドとの論理和を取ることにより得られる値にステータス・フィールドのコントロール・ビットが初期化される。

sf のニーモニック値は 7-35 ページの表 7-18 に示してある。

操作 :

```
if (PR[qp]) {
    tmp_controls = (AR[FPSR].sf0.controls & amask7) | omask7;
    if (is_reserved_field(FSETC, sf, tmp_controls))
        reserved_register_field_fault();
    fp_set_sf_controls(sf, tmp_controls);
}
```

FP 例外 : なし。

浮動小数点減算 (Floating-Point Subtract)

書式 : (qp) fsub.pc.sf $f_1 = f_3, f_2$ pseudo-op of: (qp) fms.pc.sf $f_1 = f_3, f_1, f_2$

説明 : FR f_3 から FR f_2 が減算され (無限の精度で計算される)、FPSR.*sf*.*rc* によって指定される丸めモードと *pc* (さらに FPSR.*sf*.*pc* および FPSR.*sf*.*wre*) によって指示される精度にしたがって丸められ、その結果が FR f_1 に格納される。

FR f_3 か FR f_2 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

オペコードの *pc* のニーモニック値は 7-35 ページの表 7-17 に示してある。*sf* のニーモニック値は 7-35 ページの表 7-18 に示してある。ステータス・フィールドの *pc*、*wre*、および *rc* のエンコーディングと意味については、5-7 ページの表 5-4 および 5-9 ページの表 5-5 を参照のこと。

操作 : 7-61 ページの「浮動小数点積差 (Floating-Point Multiply Subtract)」を参照のこと。

浮動小数点スワップ (Floating-Point Swap)

書式:

(qp) fswap $f_1 = f_2, f_3$	swap_form	F9
(qp) fswap.nl $f_1 = f_2, f_3$	swap_nl_form	F9
(qp) fswap.nr $f_1 = f_2, f_3$	swap_nr_form	F9

説明: swap_form では、FR f_2 内の左側の単精度値が FR f_3 内の右側の単精度値と連結される。次に、連結されたペアがスワップされる。

swap_nl_form では、FR f_2 内の左側の単精度値が FR f_3 内の右側の単精度値と連結される。次に、連結されたペアがスワップされ、左側の単精度値が否定される。

swap_nr_form では、FR f_2 内の左側の単精度値が FR f_3 内の右側の単精度値と連結される。次に、連結されたペアがスワップされ、右側の単精度値が否定される。

すべての形式について、FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

すべての形式について、FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

図 7-17. 浮動小数点スワップ

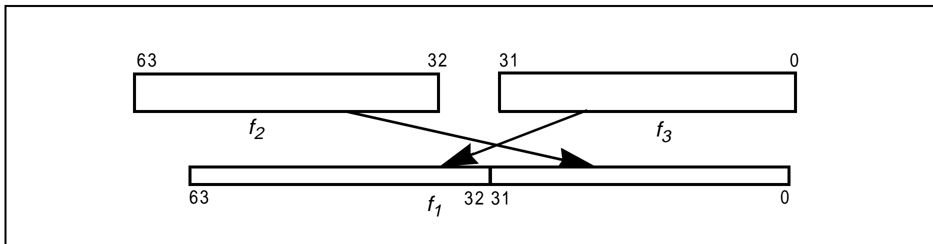
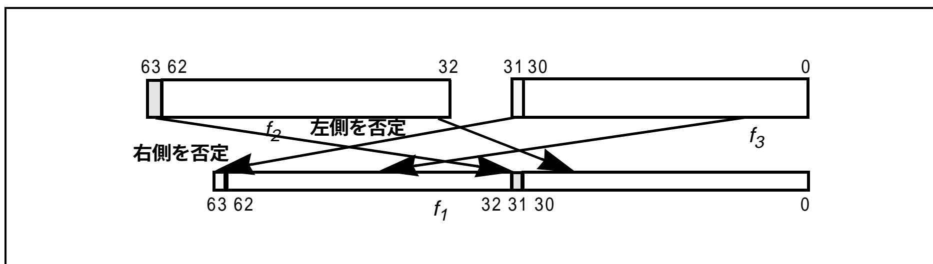


図 7-18. 浮動小数点スワップの左否定と右否定



操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        if (swap_form) {
            tmp_res_hi = FR[f3].significand{31:0};
            tmp_res_lo = FR[f2].significand{63:32};
        } else if (swap_nl_form) {
            tmp_res_hi = (!FR[f3].significand{31} << 31)
                         | (FR[f3].significand{30:0});
            tmp_res_lo = FR[f2].significand{63:32};
        } else { // swap_nr_form
            tmp_res_hi = FR[f3].significand{31:0};
            tmp_res_lo = (!FR[f2].significand{63} << 31)
                         | (FR[f2].significand{62:32});
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP 例外: なし。

浮動小数点符号拡張 (Floating-Point Sign Extend)

書式: (qp) fsxt.l $f_1 = f_2, f_3$ sxt_l_form F9
(qp) fsxt.r $f_1 = f_2, f_3$ sxt_r_form F9

説明: sxt_l_form (sxt_r_form) では、FR f_2 内の左 (右) 側の単精度値の符号が 32 ビットに拡張され、FR f_3 内の左 (右) 側の単精度値と連結される。

すべての形式について、FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

すべての形式について、FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

図 7-19. 浮動小数点左符号拡張

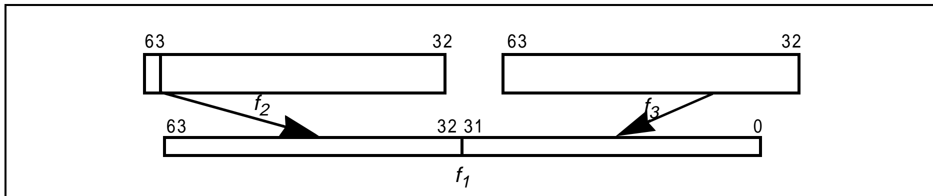
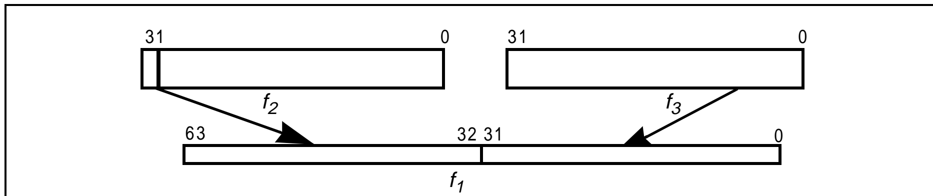


図 7-20. 浮動小数点右符号拡張



操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        if (sxt_l_form) {
            tmp_res_hi = (FR[f2].significand{63} ? 0xFFFFFFFF :
0x00000000);
            tmp_res_lo = FR[f3].significand{63:32};
        } else {                                    // sxt_r_form
            tmp_res_hi = (FR[f2].significand{31} ? 0xFFFFFFFF :
0x00000000);
            tmp_res_lo = FR[f3].significand{31:0};
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP 例外: なし。

浮動小数点排他的論理和 (Floating-Point Exclusive Or)

書式 : (*qp*) fxor $f_1 = f_2, f_3$ F9

説明 : FR f_2 および FR f_3 の両仮数フィールド間のビット単位の排他的論理和が計算される。結果の値は FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

FR f_2 か FR f_3 が NaTVal である場合は、FR f_1 は計算結果ではなく NaTVal に設定される。

操作 :

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f2].significand ^ FR[f3].significand;
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP 例外 : なし。

浮動小数点値、指数、仮数の取得 (Get Floating-Point Value or Exponent or Significand)

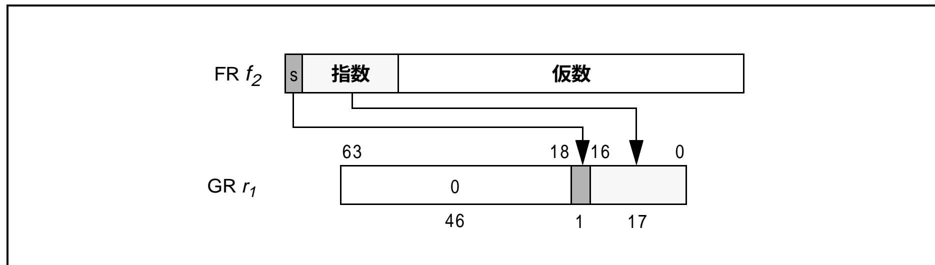
書式:

(qp) getf.s $r_1 = f_2$	single_form	M19
(qp) getf.d $r_1 = f_2$	double_form	M19
(qp) getf.exp $r_1 = f_2$	exponent_form	M19
(qp) getf.sig $r_1 = f_2$	significand_form	M19

説明: single_form および double_form では、FR f_2 の値が単精度 (single_form) または倍精度 (double_form) メモリ表現に変換され、GR r_1 に格納される。single_form では、GR r_1 の最上位の 32 ビットが 0 に設定される。

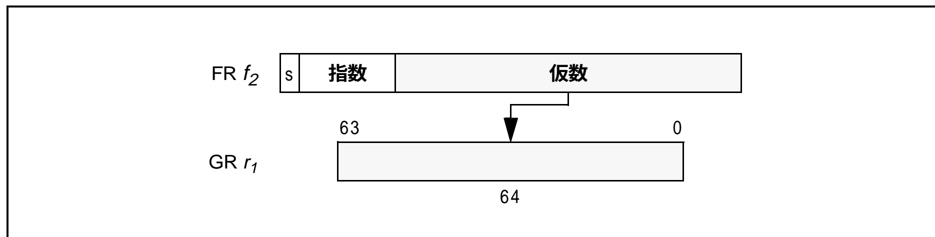
exponent_form では、FR f_2 の指数フィールドが GR r_1 のビット 16:0 にコピーされ、FR f_2 の値の符号ビットが GR r_1 のビット 17 にコピーされる。GR r_1 の最上位の 46 ビットが 0 に設定される。

図 7-21. getf.exp の機能



significand_form では、FR f_2 の値の仮数フィールドが GR r_1 にコピーされる。

図 7-22. getf.sig の機能



すべての形式について、FR f_2 が NaTVal である場合は、GR r_1 に対応する NaT ビットが 1 に設定される。

操作:

```
if (PR[qp]) {
    check_target_register(r1);
    if (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (single_form) {
        GR[r1]{31:0} = fp_fr_to_mem_format(FR[f2], 4, 0);
        GR[r1]{63:32} = 0;
    } else if (double_form) {
        GR[r1] = fp_fr_to_mem_format(FR[f2], 8, 0);
    } else if (exponent_form) {
        GR[r1]{63:18} = 0;
        GR[r1]{16:0} = FR[f2].exponent;
        GR[r1]{17} = FR[f2].sign;
    } else // significand_form
        GR[r1] = FR[f2].significand;
    if (fp_is_natval(FR[f2]))
        GR[r1].nat = 1;
    else
        GR[r1].nat = 0;
}
```

ALAT の無効化 (Invalidate ALAT)

書式:

(*qp*) invala	complete_form	M24
(*qp*) invala.e r_1	gr_form, entry_form	M26
(*qp*) invala.e f_1	fr_form, entry_form	M27

説明: ALAT 内の選択された単一または複数のエントリが無効にされる。

complete_form では、ALAT のすべてのエントリが無効にされる。entry_form では、汎用レジスタ指定子 r_1 (gr_form の場合)、あるいは浮動小数点レジスタ指定子 f_1 (fr_form の場合) を使用して ALAT が照会され、いずれかの ALAT エントリが一致した場合に、そのエントリが無効にされる。

操作:

```
if (PR[qp]) {
    if (complete_form)
        alat_inval();
    else { // entry_form
        if (gr_form)
            alat_inval_single_entry(GENERAL, r1);
        else // fr_form
            alat_inval_single_entry(FLOAT, f1);
    }
}
```

intel®

Load

書式:

(qp) ldsz.$ldtype$.$ldhint$ r_1 = [r_3]	no_base_update_form	M1
(qp) ldsz.$ldtype$.$ldhint$ r_1 = [r_3], r_2	reg_base_update_form	M2
(qp) ldsz.$ldtype$.$ldhint$ r_1 = [r_3], imm_9	imm_base_update_form	M3
(qp) ld8.fill.$ldhint$ r_1 = [r_3]	fill_form, no_base_update_form	M1
(qp) ld8.fill.$ldhint$ r_1 = [r_3], r_2	fill_form, reg_base_update_form	M2
(qp) ld8.fill.$ldhint$ r_1 = [r_3], imm_9	fill_form, imm_base_update_form	M3

説明:

sz バイトからなる値が、GR r_3 の値によって指定されるアドレスから始まるメモリ位置から読み込まれる。次に、この値がゼロ拡張され、GR r_1 に格納される。*sz* コンプリータの値を表 7-26 に示す。後述のアドバンスド・ロードの場合を除いて、GR r_1 に対応する NaT ビットがクリアされる。*ldtype* コンプリータで特殊なロード操作を指定する。それらの操作については、表 7-27 に説明してある。

fill_form では、8 バイトの値がロードされ、UNAT アプリケーション・レジスタの特定ビットがターゲット・レジスタの NaT ビットにコピーされる。この命令は、スピルしたレジスタと Nat のペアをロードする場合に使用される。詳細については、4-16 ページの「コントロール・スペキュレーション」を参照のこと。

ベース更新形式では、CR r_3 の値が符号付き即値 (imm_9) または GR r_2 の値に加算され、その結果が GR r_3 に戻される。このベース・レジスタの更新はロード後に行われ、ロード・アドレスには影響しない。reg_update_form では、GR r_2 に対応する NaT ビットがセットされている場合は、GR r_3 に対応する NaT ビットがセットされ、フォルトは発生しない。

表 7-26. sz コンプリータ

sz コンプリータ	アクセスされるバイト数
1	1 バイト
2	2 バイト
4	4 バイト
8	8 バイト

表 7-27. ロード・タイプ

ldtype コンプリータ	意味	特殊 Load 操作
none	通常のロード	
s	スペキュレーティブ・ロード	特定の例外を、フォルトを発生させないで据え置かせることができる。例外を据え置くと、ターゲット・レジスタの NaT ビットがセットされる。NaT ビットは後で据え置きの検出に利用される。

表 7-27. ロード・タイプ (続き)

ldtype コンプリータ	意味	特殊 Load 操作
a	アドバンスド・ロード	ALAT にエントリが追加される。それにより、以降の命令でストアの衝突の有無を確認できる。参照されたデータ・ページに非スペキュレーティブ属性があった場合は、ターゲット・レジスタの NaT ビットがクリアされ、プロセッサによって、ターゲット・レジスタに対する ALAT エントリが存在しないことが保証される。ALAT エントリが存在しないということを利用して、後で据え置きまたは衝突を検出する。
sa	スペキュレーティブ・アドバンスド・ロード	ALAT にエントリが追加され、特定の例外を据え置かせることができる。例外を据え置くと、ターゲット・レジスタの NaT ビットがセットされ、プロセッサによって、ターゲット・レジスタに対する ALAT エントリが存在しないことが保証される。ALAT エントリが存在しないことを利用して、後で据え置きまたは衝突を検出する。
c.nc	チェック・ロード - クリアなし	ALAT を検索して一致するエントリが探され、見つかった場合は、ロードは実行されず、ターゲット・レジスタは変更されない。指定されている場合には、ALAT のヒットまたはミスに関わらず、ベース・レジスタの更新が実行される。ALAT エントリの一致の有無に関わらず、必要に応じて ALAT ルックアップを失敗させることができる。見つからなかった場合は、ロードが実行され、参照されたデータ・ページに非スペキュレーティブ属性がない場合は、エントリが ALAT に追加される。非スペキュレーティブ属性があった場合は、ALAT エントリは割り当てられない。
c.clr	チェック・ロード - クリアあり	ALAT が検索されて一致するエントリが探され、見つかった場合は、そのエントリが削除され、ロードは実行されず、ターゲット・レジスタは変更されない。指定されている場合には、ALAT のヒットまたはミスに関わらず、ベース・レジスタ更新が実行される。ALAT エントリの一致の有無に関わらず、ALAT ルックアップを必要に応じて失敗させることができる。見つからなかった場合は、クリア・チェック・ロードの動作は通常のロードと同様になる。
c.clr.acq	順序付けされたチェック・ロード - クリアあり	このタイプの動作は、ALAT ルックアップ (および、ALAT エントリが見つからなかった場合、結果のロード) が acquire (取得) の語彙を使用して実行される以外は、順序付けなしのクリア形式と同じである。
acq	順序付けされたロード	acquire の語彙を使用して順序付けされたロードが実行される。
bias	バイアスされたロード	アクセスされるキャッシュ・ラインの排他的所有権を取得するよう指示するヒントが提示される。

「順序付けされた」、「バイアスされた」、「スペキュレーティブ」、「アドバンスド」、および「チェック型」の各タイプのロードの詳細については、4-16 ページの「コントロール・スペキュレーション」と 4-20 ページの「データ・スペキュレーション」を参照のこと。順序付けされたロードの詳細については、4-23 ページの「メモリ・アクセスの順序」を参照のこと。また、バイアスされたロードの詳細については、4-26 ページの「メモリ階層の制御と整合性」を参照のこと。

非スペキュレーティブ・ロード・タイプについては、GR r_3 に対応する NaT ビットが 1 の場合は、レジスタ NaT 参照 (Register NaT Consumption) フォルトが発生する。スペキュレーティブ・ロードおよびスペキュレーティブ・アドバンスド・ロードでは、フォルトは発生せず、例外が据え置かれる。ベース更新の計算においては、GR r_2 に対応する NaT ビットが 1 の場合、GR r_3 に対応する NaT ビットが 1 に設定され、フォルトは発生しない。

ldhint コンプリータの値で、メモリ・アクセスの局所性を指定する。*ldhint* コンプリータの値を表 7-28 に示す。ベース更新形式には、暗黙的にプリフェッチ・ヒントの意味がある。ベース更新後の GR r_3 の値によって指定されるアドレスには、指定されたキャッシュ・ラインをプリフェッチするよう指示するヒントの働きがある。このプリフェッチは、*ldhint* によって指定される局所性ヒントを使用する。プリフェッチと局所性のヒントはプログラムの機能には影響せず、インプリメンテーションによって無視することもできる。詳細については、4-26 ページの「メモリ階層の制御と整合性」を参照のこと。

表 7-28. ロードのヒント

ldhint コンプリータ	意味
none	時間的局所性、レベル 1
nt1	時間的局所性なし、レベル 1
nta	時間的局所性なし、全レベル

no_base_update 形式では GR r_3 の値は変更されず、この形式にはプリフェッチ・ヒントの暗黙的意味もない。

ベース更新形式の場合は、r_1 と r_3 に同じレジスタ・アドレスを指定すると、無効操作 (Illegal Operation) フォルトが発生する。

操作 :

```
if (PR[qp]) {
    size = fill_form ? 8 : sz;

    speculative = (ldtype == 's' || ldtype == 'sa');
    advanced = (ldtype == 'a' || ldtype == 'sa');
    check_clear = (ldtype == 'c.clr' || ldtype == 'c.clr.acq');
    check_no_clear = (ldtype == 'c.nc');
    check = check_clear || check_no_clear;
    acquire = (ldtype == 'acq' || ldtype == 'c.clr.acq');
    bias = (ldtype == 'bias') ? BIAS : 0 ;

    itype = READ;
    if (speculative) itype |= SPEC ;
    if (advanced)itype |= ADVANCE ;

    if ((reg_base_update_form || imm_base_update_form) && (r1 == r3))
        illegal_operation_fault();
    check_target_register(r1, itype);
    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);

    if (reg_base_update_form) {
        tmp_r2 = GR[r2];
        tmp_r2nat = GR[r2].nat;
    }

    if (!speculative && GR[r3].nat)                  // fault on NaT address
        register_nat_consumption_fault(itype);
```

```
    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(GENERAL, r1)) {          // no load on ld.c & ALAT hit
        if (check_clear)                  // remove entry on ld.c.clr or ld.c.clr.acq
            alat_inval_single_entry(GENERAL, r1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                  &defer);
            if (!defer) {
                otype = acquire ? ACQUIRE : UNORDERED;
                val = mem_read(paddr, size, UM.be, mattr, otype, bias | ldhint);
            }
        }
        if (check_clear || advanced)               // remove any old ALAT entry
            alat_inval_single_entry(GENERAL, r1);
        if (defer) {
            if (speculative) {
                GR[r1] = natd_gr_read(paddr, size, UM.be, mattr, otype,
                                      bias | ldhint);
                GR[r1].nat = 1;
            } else {
                GR[r1] = 0;                        // ld.a to sequential memory
                GR[r1].nat = 0;
            }
        } else {                                   // execute load normally
            if (fill_form) {                       // fill NaT on ld8.fill
                bit_pos = GR[r3]{8:3};
                GR[r1] = val;
                GR[r1].nat = AR[UNAT]{bit_pos};
            } else {                               // clear NaT on other types
                GR[r1] = zero_ext(val, size * 8);
                GR[r1].nat = 0;
            }
            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                   // add entry to ALAT
                alat_write(GENERAL, r1, paddr, size);
        }
    }

    if (imm_base_update_form) {                    // update base register
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = GR[r3].nat;
    } else if (reg_base_update_form) {
        GR[r3] = GR[r3] + tmp_r2;
        GR[r3].nat = GR[r3].nat || tmp_r2nat;
    }

    if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
        mem_implicit_prefetch(GR[r3], bias | ldhint);
}
```

浮動小数点ロード (Floating-Point Load)

書式:

(qp) ldf*fsz.fldtype.ldhint* f_1 = [r_3]	no_base_update_form	M6
(qp) ldf*fsz.fldtype.ldhint* f_1 = [r_3], r_2	reg_base_update_form	M7
(qp) ldf*fsz.fldtype.ldhint* f_1 = [r_3], imm_9	imm_base_update_form	M8
(qp) ldf8*.fldtype.ldhint* f_1 = [r_3]	integer_form, no_base_update_form	M6
(qp) ldf8*.fldtype.ldhint* f_1 = [r_3], r_2	integer_form, reg_base_update_form	M7
(qp) ldf8*.fldtype.ldhint* f_1 = [r_3], imm_9	integer_form, imm_base_update_form	M8
(qp) ldf.fill*.ldhint* f_1 = [r_3]	fill_form, no_base_update_form	M6
(qp) ldf.fill*.ldhint* f_1 = [r_3], r_2	fill_form, reg_base_update_form	M7
(qp) ldf.fill*.ldhint* f_1 = [r_3], imm_9	fill_form, imm_base_update_form	M8

説明:

fsz バイトからなる値が、GR r_3 の値によって指定されるアドレスから始まるメモリ位置から読み込まれる。次に、この値が浮動小数点レジスタ形式に変換され、FR f_1 に格納される。浮動小数点レジスタ形式への変換の詳細については、5-1 ページの「データ型および形式」を参照のこと。*fsz* コンプリータの値を表 7-29 に示す。*fldtype* コンプリータで特殊なロード操作を指定する。それらの操作については、表 7-30 に説明してある。

integer_form については、8 バイト値がロードされ、変換されずに FR f_1 の仮数フィールドに格納される。FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

fill_form については、16 バイト値がロードされ、FR f_1 の対応するフィールドに変換されずに格納される。この命令は、スピルしたレジスタを再ロードする場合に使用される。詳細については、4-16 ページの「コントロール・スペキュレーション」を参照のこと。

ベース更新形式では、CR r_3 の値が符号付き即値 (imm_9) または GR r_2 の値に加算され、その結果が GR r_3 に戻される。このベース・レジスタの更新はロード後に実行され、ロード・アドレスには影響しない。reg_update_form では、GR r_2 に対応する NaT ビットがセットされている場合は、GR r_3 に対応する NaT ビットがセットされ、フォルトは発生しない。

表 7-29. fsz コンプリータ

fsz コンプリータ	アクセスされるバイト数	メモリ形式
s	4 バイト	単精度
d	8 バイト	倍精度
e	10 バイト	拡張精度

表 7-30. FP ロードのタイプ

fldtype コンプリータ	意味	特殊ロード操作
null	通常のロード	
s	スペキュレーティブ・ロード	特定の例外を、フォルトを発生させないで据え置かせることができる。例外を据え置くと、ターゲット・レジスタが NaTVal に設定される。NaTVal ビットは後で据え置きの検出に利用される。
a	アドバンスド・ロード	ALAT にエントリが追加される。それにより、以降の命令でストアの衝突の有無を確認できる。参照されるデータ・ページに非スペキュレーティブ属性があった場合は、ALAT にエントリが追加されず、ターゲット・レジスタは次のように設定される。integer_form では、指数が 0x1003E に設定され、符号および仮数がゼロに設定される。これ以外の形式では、符号、指数、および仮数がゼロに設定される。ALAT エントリが存在しないということを利用して、後で据え置きまたは衝突を検出する。
sa	スペキュレーティブ・アドバンスド・ロード	ALAT にエントリが追加され、特定の例外を据え置かせることができる。例外を据え置くと、ターゲット・レジスタが NaTVal に設定され、プロセッサによって、ターゲット・レジスタに対する ALAT エントリが存在しないことが保証される。ALAT エントリが存在しないということを利用して、後で据え置きまたは衝突を検出する。
c.nc	チェック・ロード - クリアなし	ALAT を検索して一致するエントリが探され、見つかった場合は、ロードは実行されず、ターゲット・レジスタは変更されない。指定されている場合には、ALAT のヒットまたはミスに関わらず、ベース・レジスタの更新が実行される。ALAT エントリの一致の有無に関わらず、必要に応じて ALAT ルックアップを失敗させることができる。見つからなかった場合は、ロードが実行され、参照されたデータ・ページに非スペキュレーティブ属性がない場合は、エントリが ALAT に追加される。非スペキュレーティブ属性があった場合は、ALAT エントリは割り当てられない。
c.clr	チェック・ロード - クリアあり	ALAT が検索されて一致するエントリが探され、見つかった場合は、そのエントリが削除され、ロードは実行されず、ターゲット・レジスタは変更されない。指定されている場合には、ALAT のヒットまたはミスに関わらず、ベース・レジスタの更新が実行される。ALAT エントリの一致の有無に関わらず、必要に応じて ALAT ルックアップを失敗させることができる。見つからなかった場合は、クリア・チェック・ロードの動作は通常のロードと同様になる。

「スペキュレーティブ」、「アドバンスド」、および「チェック型」の各タイプのロードの詳細については、4-16 ページの「コントロール・スペキュレーション」と 4-16 ページの「データ・スペキュレーション」を参照のこと。

非スペキュレーティブ・ロード・タイプについては、GR r_3 に対応する NaT ビットが 1 の場合は、レジスタ NaT 参照 (Register NaT Consumption) フォルトが発生する。スペキュレーティブ・ロードおよびスペキュレーティブ・アドバンスド・ロードでは、フォルトは発生せず、例外が据え置かれる。ベース

intel®

更新の計算については、GR r_2 に対応する NaT ビットが 1 の場合は、GR r_3 に対応する NaT ビットが 1 に設定され、フォルトは発生しない。

ldhint コンプリータの値でメモリ・アクセスの局所性を指定する。*ldhint* コンプリータの値は 7-113 ページの表 7-28 に示してある。ベース更新形式には、暗黙的にプリフェッチ・ヒントの意味がある。ベース更新後の GR r_3 の値によって指定されるアドレスには、指定されたキャッシュ・ラインをプリフェッチするよう指示するヒントの働きがある。このプリフェッチは、*ldhint* によって指定される局所性ヒントを使用する。プリフェッチと局所性のヒントはプログラムの機能性には影響せず、プログラム・コードによって無視することもできる。詳細については、4-26 ページの「メモリ階層の制御と整合性」を参照のこと。

no_base_update 形式では GR r_3 の値は変更されず、この形式にはプリフェッチ・ヒントの暗黙的意味もない。

PSR.mfl ビットおよび PSR.mfh ビットが更新されて、FR f_1 の変更を記録する。

操作：

```
if (PR[qp]) {
    size = (fill_form ? 16 : (integer_form ? 8 : fsz));
    speculative = (fldtype == 's' || fldtype == 'sa');
    advanced = (fldtype == 'a' || fldtype == 'sa');
    check_clear = (fldtype == 'c.clr' );
    check_no_clear = (fldtype == 'c.nc');
    check = check_clear || check_no_clear;

    itype = READ;
    if (speculative) itype |= SPEC ;
    if (advanced) itype |= ADVANCE ;

    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, itype);

    if (!speculative && GR[r3].nat)              // fault on NaT address
        register_nat_consumption_fault(itype);

    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(FLOAT, f1)) {          // no load on ldf.c & ALAT hit
        if (check_clear)                         // remove entry on ldf.c.clr
            alat_inval_single_entry(FLOAT, f1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                  &defer);
            if (!defer)
                val = mem_read(paddr, size, UM.be, mattr, UNORDERED, ldhint);
        }
        if (check_clear || advanced)             // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
        if (speculative && defer) {
            FR[f1] = NATVAL;
        } else if (advanced && !speculative && defer) {
            FR[f1] = (integer_form ? FP_INT_ZERO : FP_ZERO);
        } else {                                 // execute load normally
            FR[f1] = fp_mem_to_fr_format(val, size, integer_form);

            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                   // add entry to ALAT
```

```
                alat_write(FLOAT, f1, paddr, size);
            }
        }

        if (imm_base_update_form) {                     // update base register
            GR[r3] = GR[r3] + sign_ext(imm9, 9);
            GR[r3].nat = GR[r3].nat;
        } else if (reg_base_update_form) {
            GR[r3] = GR[r3] + GR[r2];
            GR[r3].nat = GR[r3].nat || GR[r2].nat;
        }

        if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
            mem_implicit_prefetch(GR[r3], ldhint);

        fp_update_psr(f1);
    }
```

浮動小数点ペア・ロード (Floating-Point Load Pair)

書式:

(qp) ldfps.*fldtype*.*ldhint* $f_1, f_2 = [r_3]$	single_form, no_base_update_form	M11
(qp) ldfps.*fldtype*.*ldhint* $f_1, f_2 = [r_3]$, 8	single_form, base_update_form	M12
(qp) ldfpd.*fldtype*.*ldhint* $f_1, f_2 = [r_3]$	double_form, no_base_update_form	M11
(qp) ldfpd.*fldtype*.*ldhint* $f_1, f_2 = [r_3]$, 16	double_form, base_update_form	M12
(qp) ldfp8.*fldtype*.*ldhint* $f_1, f_2 = [r_3]$	integer_form, no_base_update_form	M11
(qp) ldfp8.*fldtype*.*ldhint* $f_1, f_2 = [r_3]$, 16	integer_form, base_update_form	M12

説明:

8 (single_form の場合) または 16 (double_form/integer_form の場合) バイトが、GR r_3 の値によって指定されるアドレスから始まるメモリ位置から読み込まれる。読み込まれた値は、single_form/double_form については連続した浮動小数点数ペアとして、integer_form については整数 / 並列 FP データとして扱われる。各数値は浮動小数点レジスタ形式に変換される。下位アドレスの値は、FR f_1 に格納され、上位アドレスの値は FR f_2 に格納される。浮動小数点レジスタ形式への変換の詳細については、5-1 ページの「データ型および形式」を参照のこと。*fldtype* コンプリータで特殊なロード操作を指定する。それらの操作については、7-116 ページの表 7-30 に説明してある。

「スペキュレーティブ」、「アドバンスド」、および「チェック」の各タイプのロードの詳細については、4-16 ページの「コントロール・スペキュレーション」および 4-20 ページの「データ・スペキュレーション」を参照のこと。

非スペキュレーティブ・ロード・タイプでは、GR r_3 に対応する NaT ビットが 1 であった場合、レジスタ NaT 参照 (Register NaT Consumption) フォルトが発生する。スペキュレーティブ・ロードおよびスペキュレーティブ・アドバンスド・ロードでは、フォルトは発生せず、例外が据え置かれる。

base_update_form では、GR r_3 の値が暗黙指定の (データ・サイズの 2 倍に等しい) 即値に加算され、その結果が GR f_3 に格納される。このベース・レジスタの更新は、ロードの後に行われ、ロード・アドレスには影響しない。

ldhint コンプリータの値で、メモリ・アクセスの局所性を指定する。*ldhint* のニーモニック値は 7-113 ページの表 7-28 に示してある。ベース更新形式には、暗黙的にプリフェッチ・ヒントの意味がある。ベース更新後の GR r_3 の値によって指定されるアドレスには、指定されたキャッシュ・ラインをプリフェッチするよう指示するヒントの働きがある。このプリフェッチは、*ldhint* によって指定される局所性ヒントを使用する。プリフェッチと局所性のヒントはプログラムの機能には影響せず、プログラム・コードによって無視することもできる。詳細については、4-26 ページの「メモリ階層の制御と整合性」を参照のこと。

no_base_update 形式では GR r_3 の値は変更されず、この形式にはプリフェッチ・ヒントの暗黙的意味もない。

RSP.mfl ビットおよび PSR.mfh ビットが更新されて、FR f_1 と FR f_2 の変更を記録する。

ターゲット・レジスタの選択については制約がある。レジスタ指定子 f_1 と f_2 は、それぞれ、奇数番号の物理 FR と偶数番号の物理 FR を 1 つずつ指定しなければならない。奇数または偶数番号だけのレジスタを 2 つ選択すると、無

効操作 (Illegal Operation) フォルトが発生する。この制約は、レジスタ・ローテーション後の物理レジスタ番号に課せられる。つまり、f_1 と f_2 に共にスタティック・レジスタを指定したり、共にローテート・レジスタを指定する場合は、f_1 と f_2 は奇数 / 偶数または偶数 / 奇数の組み合わせでなければならない。f_1 と f_2 とが同一のスタティック・レジスタまたは同一のローテート・レジスタを指定する場合は、制約は CFM.rrb.fr に応じて異なる。CFM.rrb.fr が偶数の場合は、制約条件は同じであり、つまり f_1 と f_2 は奇数 / 偶数または偶数 / 奇数でなければならない。CFM.rrb.fr が奇数の場合は、f_1 と f_2 は偶数 / 偶数または奇数 / 奇数でなければならない。スタティック・レジスタとローテート・レジスタを 1 つずつ指定するのは、CFM.rrb.fr の値が予測可能な値 (0 など) になるときに限られる。

操作:

```
if (PR[qp]) {
    size = single_form ? 8 : 16;

    speculative = (fldtype == 's' || fldtype == 'sa');
    advanced = (fldtype == 'a' || fldtype == 'sa');
    check_clear = (fldtype == 'c.clr');
    check_no_clear = (fldtype == 'c.nc');
    check = check_clear || check_no_clear;

    itype = READ;
    if (speculative) itype |= SPEC;
    if (advanced) itype |= ADVANCE;

    if (fp_reg_bank_conflict(f1, f2))
        illegal_operation_fault();

    if (base_update_form)
        check_target_register(r3);

    fp_check_target_register(f1);
    fp_check_target_register(f2);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, itype);

    if (!speculative && GR[r3].nat)              // fault on NaT address
        register_nat_consumption_fault(itype);

    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(FLOAT, f1)) {          // no load on ldfp.c & ALAT hit
        if (check_clear)                         // remove entry on ldfp.c.clr
            alat_inval_single_entry(FLOAT, f1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                  &defer);
            if (!defer)
                val = mem_read(paddr, size, UM.be, mattr, UNORDERED, ldhint);
        }
        if (check_clear || advanced)             // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
        if (speculative && defer) {
            FR[f1] = NATVAL;
            FR[f2] = NATVAL;
        } else if (advanced && !speculative && defer) {
            FR[f1] = (integer_form ? FP_INT_ZERO : FP_ZERO);
            FR[f2] = (integer_form ? FP_INT_ZERO : FP_ZERO);
        } else {                                 // execute load normally
            if (UM.be) {
                FR[f1] = fp_mem_to_fr_format(val u>> (size/2*8), size/2,
```

```
                    integer_form);
                FR[f2] = fp_mem_to_fr_format(val, size/2, integer_form);
            } else {
                FR[f1] = fp_mem_to_fr_format(val, size/2, integer_form);
                FR[f2] = fp_mem_to_fr_format(val u>> (size/2*8), size/2,
                                                integer_form);
            }

            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                    // add entry to ALAT
                alat_write(FLOAT, f1, paddr, size);
        }
    }

    if (base_update_form) {                         // update base register
        GR[r3] = GR[r3] + size;
        GR[r3].nat = GR[r3].nat;
        if (!GR[r3].nat)
            mem_implicit_prefetch(GR[r3], ldhint);
    }

    fp_update_psr(f1);
    fp_update_psr(f2);
}
```

ライン・プリフェッチ (Line Prefetch)

書式:

(*qp*) lfetch.*lftype*.*lfhint* [r_3]	no_base_update_form	M13
(*qp*) lfetch.*lftype*.*lfhint* [r_3], r_2	reg_base_update_form	M14
(*qp*) lfetch.*lftype*.*lfhint* [r_3], imm_9	imm_base_update_form	M15
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3]	no_base_update_form, exclusive_form	M13
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3], r_2	reg_base_update_form, exclusive_form	M14
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3], imm_9	imm_base_update_form, exclusive_form	M15

説明:

GR r_3 の値によって指定されるアドレスのラインが、データ・メモリ階層の最上位レベルに移動される。*lfhint* 変更子でメモリ・アクセスの局所性を指定する。*lfhint* のニーモニック値を表 7-32 に示す。

メモリ読み取りの動作は、アクセスされるページに関連付けられているメモリ属性によっても決まる。ライン・サイズはインプリメンテーションに依存するが、32 バイト以上の 2 の倍数でなければならない。排他形式では、キャッシュ・ラインは排他的状態でマーキングできる。この修飾子は、プログラムがそのライン上の特定のメモリ位置をすぐに変更するものと予想する場合に使用される。そのラインがあるページのメモリ属性がキャッシング不可能である場合は、参照は行われない。

lftype コンプリータで、この命令が、正規のロードに通常に関連付けられているフォルトを発生するかどうかを指定する。表 7-32 に、それらの 2 つのオプションの定義を示す。

表 7-31. lftype のニーモニック値

lftype ニーモニック	意味
none	フォルトを無視する
fault	フォルトを発生する

ベース更新形式では、GR r_3 の値がメモリのアドレス指定に使用された後に、imm_9 の符号拡張値 (imm_base_update_form の場合) または GR r_2 の値 (reg_base_update_form の場合) だけインクリメントされる。reg_base_update_form では、GR r_2 に対応する NaT ビットがセットされている場合は、GR r_3 に対応する NaT ビットがセットされて、フォルトは発生しない。

reg_base_update_form と imm_base_update_form では、GR r_3 に対応する NaT ビットがクリアされている場合は、ポスト・インクリメント処理後の GR r_3 の値によって指定されるアドレスには、指定されたキャッシュ・ラインをプリフェッチするよう暗黙的に指示するヒントの働きがある。この暗黙指定のプリフェッチでは、*lfhint* によって指定される局所性ヒントを使用する。暗黙指定プリフェッチと局所性のヒントはプログラムの機能には影響せず、プログラム・コードによって無視することもできる。

no_base_update_form では GR r_3 の値は変更されず、この形式にはプリフェッチ・ヒントの暗黙的意味もない。

GR r_3 に対応する NaT ビットがセットされている場合は、メモリの状態は影響を受けない。reg_base_update_form と imm_base_update_form では、GR r_3 のポスト・インクリメントが行われ、上記のようにヒントによってプリフェッチが指示される。

表 7-32. lfhint のニーモニック値

lfhint ニーモニック	意味
none	時間的局所性、レベル 1
nt1	時間的局所性なし、レベル 1
nt2	時間的局所性なし、レベル 2
nta	時間的局所性なし、全レベル

操作：

```
if (PR[qp]) {
    itype = READ|NON_ACCESS;
    itype |= (lftype == 'fault') ? LFETCH_FAULT : LFETCH;

    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);

    if (lftype == 'fault') {                // faulting form
        if (GR[r3].nat && !PSR.ed)          // fault on NaT address
            register_nat_consumption_fault(itype);
    }

    if (exclusive_form)
        excl_hint = EXCLUSIVE;
    else
        excl_hint = 0;

    if (!GR[r3].nat && !PSR.ed) {// faulting form already faulted if r3 is
nat'ed
        paddr = tlb_translate(GR[r3], 1, itype, PSR.cpl, &mattr, &defer);
        if (!defer)
            mem_promote(paddr, mattr, lfhint | excl_hint);
    }

    if (imm_base_update_form) {
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = GR[r3].nat;
    } else if (reg_base_update_form) {
        GR[r3] = GR[r3] + GR[r2];
        GR[r3].nat = GR[r2].nat || GR[r3].nat;
    }

    if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
        mem_implicit_prefetch(GR[r3], lfhint | excl_hint);
}
```

メモリ・フェンス (Memory Fence)

書式:

(*qp*) mf	ordering_form	M24
(*qp*) mf.a	acceptance_form	M24

説明: この命令は、先行と後続のメモリ・アクセスの間の順序付けを強制する。ordering_form では、先行するすべてのデータ・メモリ・アクセスが参照可能になってから、後続のデータ・メモリ・アクセスが参照可能になることが保証される。この形式では、先行するデータ・メモリ参照が外部プラットフォームに受け入れられたどうか、さらに、先行するデータ・メモリ参照が検出可能であるかどうかについては保証されない。

acceptance_form では、以下の処置が終了するまで、プロセッサによる後続のデータ・メモリ・アクセスによって外部プラットフォームへのトランザクションを開始するのを禁止する。

- すべての先行ロードがデータを返した。
- すべての先行ストアが外部プラットフォームによって受け入れられた。

" 受け入れ " の定義はプラットフォームによって異なる。acceptance_form は、一般的に、プロセッサが、メモリ・マップド IO トランザクションが「受け入れられる」まで「待って」から、次の外部トランザクションを開始するように保証する場合に使用される。acceptance_form では、順序付けは保証していない。

操作:

```
if (PR[qp]){
    if (acceptance_form)
        acceptance_fence();
    else
        ordering_fence();
}
```

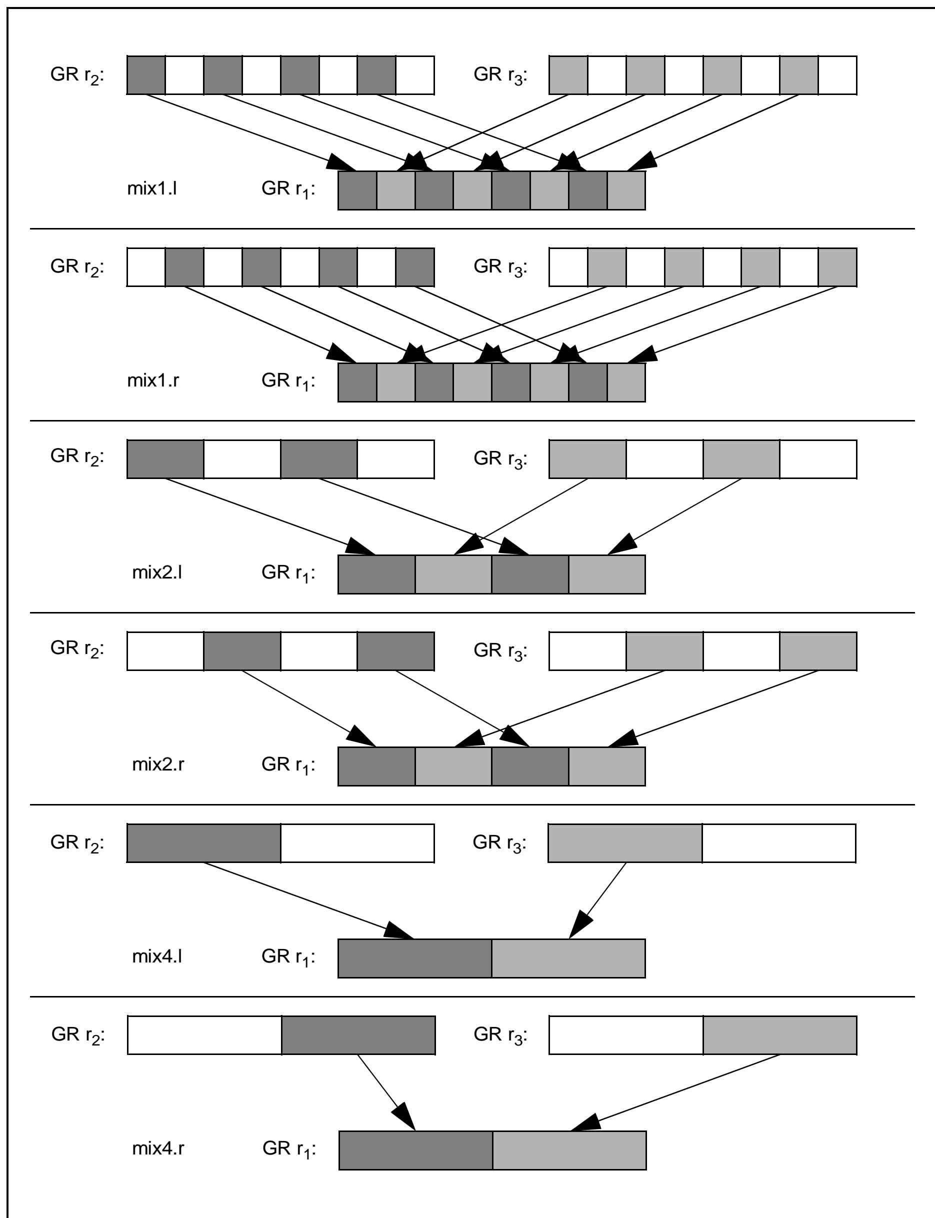
ミックス (Mix)

書式 :

(qp) mix1.l $r_1 = r_2, r_3$	one_byte_form, left_form	I2
(qp) mix2.l $r_1 = r_2, r_3$	two_byte_form, left_form	I2
(qp) mix4.l $r_1 = r_2, r_3$	four_byte_form, left_form	I2
(qp) mix1.r $r_1 = r_2, r_3$	one_byte_form, right_form	I2
(qp) mix2.r $r_1 = r_2, r_3$	two_byte_form, right_form	I2
(qp) mix4.r $r_1 = r_2, r_3$	four_byte_form, right_form	I2

説明 : GR r_2 および r_3 のデータ要素が図 7-23 に示すようにミックスされ、その結果が GR r_1 に格納される。両方のソース・レジスタ内のソース・オペランドは、1、2、あるいは 4 対のグループに分けられ、各ペアから 1 要素ずつ選択されて結果に組み込まれる。left_form では、各ペアの左側要素からミックスされる。right_form では、結果は各右側要素から形成される。要素は 2 つのソース・レジスタから交互に選択される。

図 7-23. ミックスの例



操作：

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                        // one-byte elements
        x[0] = GR[r2]{7:0};         y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};        y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};       y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};       y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};       y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};       y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};       y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};       y[7] = GR[r3]{63:56};

        if (left_form)
            GR[r1] = concatenate8( x[7], y[7], x[5], y[5],
                                   x[3], y[3], x[1], y[1]);
        else
            GR[r1] = concatenate8( x[6], y[6], x[4], y[4],
                                   x[2], y[2], x[0], y[0]);

    } else if (two_byte_form) {                                 // two-byte elements
        x[0] = GR[r2]{15:0};        y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};       y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};       y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};       y[3] = GR[r3]{63:48};

        if (left_form)
            GR[r1] = concatenate4(x[3], y[3], x[1], y[1]);
        else
            GR[r1] = concatenate4(x[2], y[2], x[0], y[0]);

    } else {                                                    // four-byte elements
        x[0] = GR[r2]{31:0};        y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};       y[1] = GR[r3]{63:32};

        if (left_form)
            GR[r1] = concatenate2(x[1], y[1]);
        else
            GR[r1] = concatenate2(x[0], y[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

アプリケーション・レジスタの移動 (Move Application Register)

書式:

(qp) mov $r_1 = ar_3$	pseudo-op	
(qp) mov $ar_3 = r_2$	pseudo-op	
(qp) mov $ar_3 = imm_8$	pseudo-op	
(qp) mov.i $r_1 = ar_3$	i_form, from_form	I28
(qp) mov.i $ar_3 = r_2$	i_form, register_form, to_form	I26
(qp) mov.i $ar_3 = imm_8$	i_form, immediate_form, to_form	I27
(qp) mov.m $r_1 = ar_3$	m_form, from_form	M31
(qp) mov.m $ar_3 = r_2$	m_form, register_form, to_form	M29
(qp) mov.m $ar_3 = imm_8$	m_form, immediate_form, to_form	M30

説明: ソース・オペランドがデスティネーション・レジスタにコピーされる。

from_form では、ar_3 レジスタによって指定されるアプリケーション・レジスタが GR r_1 にコピーされ、対応する NaT ビットがクリアされる。

to_form では、GR r_2 の値 (register_form の場合)、あるいは imm8 の符号拡張された値 (immediate_form の場合) が AR ar_3 に格納される。register_form では、GR r_2 に対応する NaT ビットがセットされている場合は、レジスタ NaT 参照 (Register NaT Consumption) フォルトが発生する。

各実行ユニット (M または I) からは、それぞれアプリケーション・レジスタの特定のサブセットに対してしかアクセスできない。3-8 ページの表 3-3 に、どの実行ユニット・タイプからどのアプリケーション・レジスタにアクセスできるかが示してある。間違ったユニット・タイプからアプリケーション・レジスタにアクセスされると、無効操作 (Illegal Operation) フォルトが発生する。

この命令には、擬似オペコードが対応していて、したがって実行ユニットを指定する必要がない形式が 3 つある。AR へのアクセスは常に暗黙的にシリアル化される。暗黙的にシリアル化が行われるときは、リード・アフター・ライトの依存関係とライト・アフター・ライトの依存関係は避けなければならない。例えば、同一命令グループ内で CCV を設定してから `cmpxchg` 命令を使用したり、`ld.fill` 命令による UNAT レジスタへの書き込みと UNAT への移動を同時に行ったりすることなどが挙げられる。

操作:

```
if (PR[qp]) {
    tmp_type = (i_form ? AR_I_TYPE : AR_M_TYPE);
    if (is_reserved_reg(tmp_type, ar3))
        illegal_operation_fault();

    if (from_form) {
        check_target_register(r1);
        if (((ar3 == BSPSTORE) || (ar3 == RNAT)) && (AR[RSC].mode != 0))
            illegal_operation_fault();

        if (ar3 == ITC && PSR.si && PSR.cpl != 0)
            privileged_register_fault();

        GR[r1] = (is_ignored_reg(ar3)) ? 0 : AR[ar3];
        GR[r1].nat = 0;
    } else {                                                // to_form
        tmp_val = (register_form) ? GR[r2] : sign_ext(imm8, 8);

        if (ar3 == BSP)
            illegal_operation_fault();

        if (((ar3 == BSPSTORE) || (ar3 == RNAT)) && (AR[RSC].mode != 0))
            illegal_operation_fault();

        if (register_form && GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(AR_TYPE, ar3, tmp_val))
            reserved_register_field_fault();

        if ((is_kernel_reg(ar3) || ar3 == ITC) && (PSR.cpl != 0))
            privileged_register_fault();

        if (!is_ignored_reg(ar3)) {
            tmp_val = ignored_field_mask(AR_TYPE, ar3, tmp_val);
            // check for illegal promotion
            if (ar3 == RSC && tmp_val{3:2} u< PSR.cpl)
                tmp_val{3:2} = PSR.cpl;
            AR[ar3] = tmp_val;

            if (ar3 == BSPSTORE) {
                AR[BSP] = rse_update_internal_stack_pointers(tmp_val);
                AR[RNAT] = undefined();
            }
        }
    }
}
```

分岐レジスタの移動 (Move Branch Register)

書式:

(*qp*) mov $r_1 = b_2$	from_form	I22
(*qp*) mov$b_1 = r_2$	to_form	I21
(*qp*) mov.ret.$b_1 = r_2$	return_form, to_form	I21

説明: ソース・オペランドがデスティネーション・レジスタにコピーされる。

from_form では、b_2 によって指定される分岐レジスタが GR r_1 にコピーされる。GR r_1 に対応する NaT ビットがクリアされる。

to_form では、GR r_2 の値が BR b_1 にコピーされる。GR r_2 に対応する NaT ビットが 1 である場合は、レジスタ NaT 参照 (Register NaT Consumption) フォルトが発生する。

操作:

```
if (PR[qp]) {
    if (from_form) {
        check_target_register(r1);
        GR[r1] = BR[b2];
        GR[r1].nat = 0;
    } else { // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);
        BR[b1] = GR[r2];
    }
}
```

浮動小数点レジスタの移動 (Move Floating-Point Register)

書式: (*qp*) mov $f_1 = f_3$ pseudo-op of: (*qp*) fmerge.s $f_1 = f_3, f_3$

説明: FR f_3 の値が FR f_1 にコピーされる。

操作: 7-55 ページの「浮動小数点マージ (Floating-Point Merge)」を参照のこと。

汎用レジスタの移動 (Move General Register)

書式 : (*qp*) mov $r_1 = r_3$ pseudo-op of: (*qp*) adds $r_1 = 0, r_3$

説明 : GR r_3 の値が GR r_1 にコピーされる。

操作 : 7-3 ページの「加算 (Add)」を参照のこと。

即値の移動 (Move Immediate)

書式: (*qp*) mov $r_1 = imm_{22}$ pseudo-op of: (*qp*) addl $r_1 = imm_{22}$, r0

説明: 即値 imm22 が 64 ビットに符号拡張され、GR r_1 に格納される。

操作: 7-3 ページの「加算 (Add)」を参照のこと。

間接レジスタの移動 (Move Indirect Register)

書式 : (*qp*) mov r_1 = *ireg*[r_3] from_form M43

説明 : ソース・オペランドがデスティネーション・レジスタにコピーされる。

間接レジスタからの移動では、GR r_3 が読み込まれ、その値が、*ireg* によって指定されるレジスタ・ファイル (下の表 7-33 を参照) へのインデックスとして使用される。インデックスされたレジスタが読み込まれ、その値が GR r_1 にコピーされる。

表 7-33. 間接レジスタ・ファイルのニーモニック

ireg	レジスタ・ファイル
cpuid	プロセッサ識別 (PI) レジスタ
pmd	パフォーマンス・モニタ・データ (PMD) レジスタ

GR r_3 のビット {7:0} がインデックスとして使用される。その他のビットは無視される。

PMD レジスタ・ファイルを除いて、存在しないレジスタにアクセスすると、予約レジスタ / フィールド (Reserved Register/Field) フォルトが発生する。PMD レジスタ・ファイルのプロセッサ依存部分へのアクセスはすべて、プロセッサによって異なる動作を生じるが、フォルトは発生しない。

操作 :

```
if (PR[qp]) {
    tmp_index = GR[r3]{7:0};

    if (from_form) {
        check_target_register(r1);

        if (GR[r3].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_reg(ireg, tmp_index))
            reserved_register_field_fault();

        if (ireg == PMD_TYPE) {
            GR[r1] = pmd_read(tmp_index);
        } else
            switch (ireg) {
                case CPUID_TYPE:   GR[r1] = CPUID[tmp_index]; break;
            }
        GR[r1].nat = 0;
    }
}
```

命令ポインタの移動 (Move Instruction Pointer)

書式: (*qp*) mov r_1 = ip　　I25

説明: この命令があるバンドルの命令ポインタ (IP) が GR r_1 にコピーされる。

操作:

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = IP;
    GR[r1].nat = 0;
}
```

プレディケートの移動 (Move Predicates)

書式:

(qp) mov r_1 = pr	from_form	I25
(qp) mov pr = r_2, $mask_{17}$	to_form	I23
(qp) mov pr.rot = imm_{44}	to_rotate_form	I24

説明: ソース・オペランドがデスティネーション・レジスタにコピーされる。

GR へのプレディケートの移動では、PR i が GR r_1 内のビット位置 i にコピーされる。

プレディケートへの移動では、ソースには汎用レジスタと即値が使用できる。to_form では、ソース・オペランドは GR r_2 であり、即値 $mask_{17}$ によって指定されるプレディケートだけが書き込まれる。$mask_{17}$ の値は、$imm_{16} = mask_{17}$ >> 1 になるように命令内で imm_{16} フィールドにエンコードされる。プレディケート・レジスタ 0 は常に 1 である。$mask_{17}$ の値は符号拡張される。したがって、$mask_{17}$ の最上位ビットはローテート・プレディケートのすべてに対するマスク・ビットである。GR r_2 に対して据え置き例外が存在していた (NaT ビットが 1) 場合は、レジスタ NaT 参照 (Register NaT Consumption) フォルトが発生する。

to_rotate_form では、48 のローテート・プレディケートしか書けない。ソース・オペランドは $mask_{44}$ オペランドで与えられる (命令内では、$mask_{28}$= imm_{44} >> 16 になるように、imm_{28} フィールドにエンコードされる)。下位 16 ビットはスタティックなプレディケート・レジスタに対応する。この即値が符号拡張されて、上位 21 のプレディケートを設定する。ソース・オペランドのビット位置 i が PR i にコピーされる。

この命令の操作は、現在のフレーム・マーカ (CFM.rrb.pr) 内のプレディケート・ローテーション・ベースがゼロであった場合と同様になる。

操作:

```
if (PR[qp]) {
    if (from_form) {
        check_target_register(r1);
        GR[r1] = 1;                             // PR[0] is always 1
        for (i = 1; i <= 63; i++) {
            GR[r1]{i} = PR[pr_phys_to_virt(i)];
        }
        GR[r1].nat = 0;
    } else if (to_form) {
        if (GR[r2].nat)
            register_nat_consumption_fault(0);
        tmp_src = sign_ext(mask17, 17);
        for (i = 1; i <= 63; i++) {
            if (tmp_src{i})
                PR[pr_phys_to_virt(i)] = GR[r2]{i};
        }
    } else {    // to_rotate_form
        tmp_src = sign_ext(imm44, 44);
        for (i = 16; i <= 63; i++) {
            PR[pr_phys_to_virt(i)] = tmp_src{i};
        }
    }
}
```

ユーザ・マスクの移動 (Move User Mask)

書式：

(*qp*) mov r_1 = psr.um	from_form	M36
(*qp*) mov psr.um = r_2	to_form	M35

説明：ソース・オペランドがデスティネーション・レジスタにコピーされる

ユーザ・マスクからの移動では、PSR {5:0} が読み込まれ、ゼロ拡張されて、GR r_1 にコピーされる。

ユーザ・マスクへの移動では、GR r_2 のビット {5:0} が PSR {5:0} に書き込まれる。

操作：

```
if (PR[qp]) {
    if (from_form) {
        check_target_register(r1);

        GR[r1] = zero_ext(PSR{5:0}, 6);
        GR[r1].nat = 0;
    } else {                                          // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(PSR_TYPE, PSR_UM, GR[r2]))
            reserved_register_field_fault();

        PSR{5:0} = GR[r2]{5:0};
    }
}
```

ロング型即値の移動 (Move Long Immediate)

書式: (qp) movl $r_1 = imm_{64}$ X2

説明: 即値 imm_{64} が GR r_1 にコピーされる。バンドルの L スロットに imm_{64} の 41 ビットが入る。

操作:

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = imm64;
    GR[r1].nat = 0;
}
```

置換 (Mux)

書式:

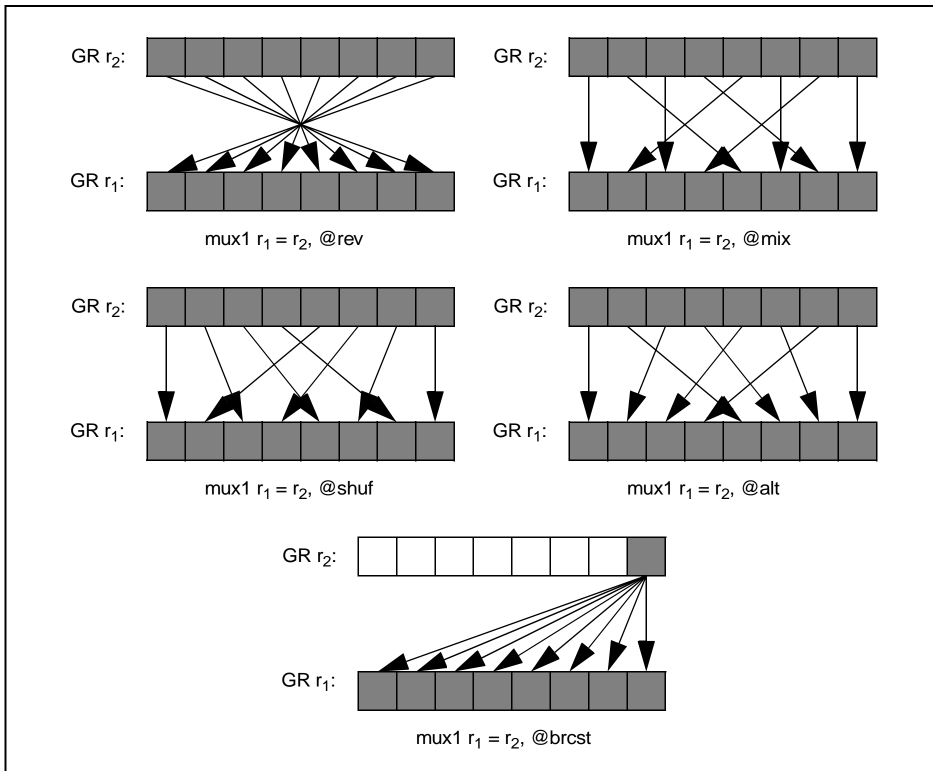
(qp) mux1 $r_1 = r_2$, $mbtype_4$	one_byte_form	I3
(qp) mux2 $r_1 = r_2$, $mhtype_8$	two_byte_form	I4

説明: 1 つのソース・レジスタ GR r_2 内のパック形式の要素に対して置換が行われ、その結果が
GR r_1 に格納される。8 ビット要素に対しては、可能なすべての置換タイプのなかの一部しか指定できない。5 つの可能な置換とそれらの図解を、それぞれ表 7-37 と図 7-24 に示す。

表 7-34. 8 ビット要素に対する Mux による置換

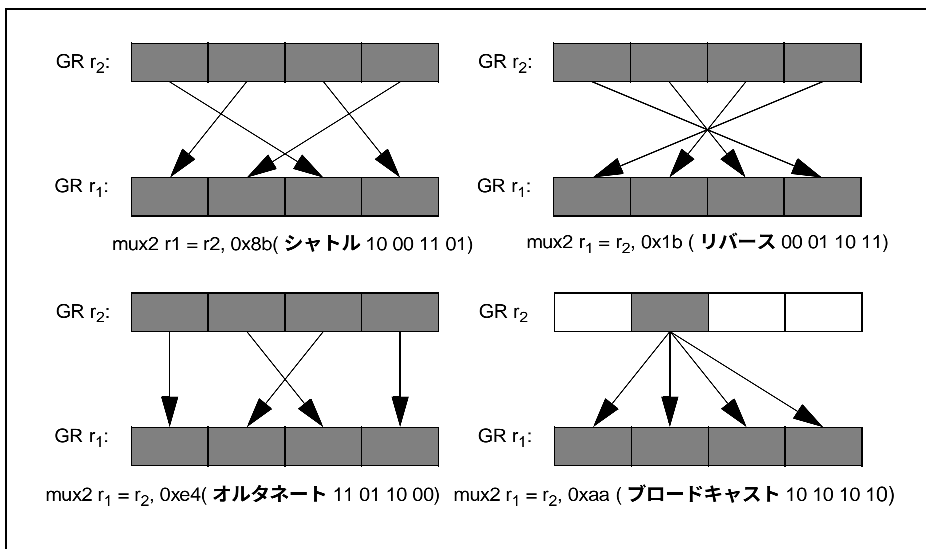
$mbtype_4$	機能
@rev	バイトの順序を逆にする。
@mix	GR r_2 の両半分に対してミックス操作を実行する。
@shuf	GR r_2 の両半分に対してシャッフル操作を実行する。
@alt	GR r_2 の両半分に対して交互操作を実行する。
@brcst	GR r_2 の最下位バイトに対してブロードキャスト操作を実行する。

図 7-24. Mux1 の操作 (8 ビット要素)



16 ビット要素に対しては、反復の有無にかかわらず、可能なすべての置換タイプを指定できる。これらの置換は、4 つの 16 ビット・データ要素のインデックスをエンコーディングしたものである 8 ビットの $mhtype_8$ フィールドで表現される。インデックスされた GR r_2 の 16 ビット要素が、ターゲット・レジスタ GR r_1 の対応する 16 ビット位置にコピーされる。インデックスはリトル・エンディアン型の順序でエンコードされる。($mhtype_8$ の 8 ビット [7:0] がビット対のグループに分けられ、操作の項では、$mhtype_8$[3]、$mhtype_8$[2]、$mhtype_8$[1]、$mhtype_8$[0] と命名されている。)

図 7-25.　Mux2 の例 (16 ビット要素)



操作:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {
        x[0] = GR[r2]{7:0};
        x[1] = GR[r2]{15:8};
        x[2] = GR[r2]{23:16};
        x[3] = GR[r2]{31:24};
        x[4] = GR[r2]{39:32};
        x[5] = GR[r2]{47:40};
        x[6] = GR[r2]{55:48};
        x[7] = GR[r2]{63:56};

        switch (mbtype) {
            case '@rev':
                GR[r1] = concatenate8( x[0], x[1], x[2], x[3],
                                       x[4], x[5], x[6], x[7]);
                break;

            case '@mix':
                GR[r1] = concatenate8( x[7], x[3], x[5], x[1],
                                       x[6], x[2], x[4], x[0]);
                break;

            case '@shuf':
                GR[r1] = concatenate8( x[7], x[3], x[6], x[2],
                                       x[5], x[1], x[4], x[0]);
                break;

            case '@alt':
                GR[r1] = concatenate8( x[7], x[5], x[3], x[1],
                                       x[6], x[4], x[2], x[0]);
                break;

            case '@brcst':
                GR[r1] = concatenate8( x[0], x[0], x[0], x[0],
                                       x[0], x[0], x[0], x[0]);
                break;
        }
    } else {                                              // two_byte_form
        x[0] = GR[r2]{15:0};
        x[1] = GR[r2]{31:16};
        x[2] = GR[r2]{47:32};
        x[3] = GR[r2]{63:48};

        res[0] = x[mhtype8{1:0}];
        res[1] = x[mhtype8{3:2}];
        res[2] = x[mhtype8{5:4}];
        res[3] = x[mhtype8{7:6}];

        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat;
}
```

ノー・オペレーション (No Operation)

書式：

(qp) nop imm_{21}	pseudo-op	
(qp) nop.i imm_{21}	i_unit_form	I19
(qp) nop.b imm_{21}	b_unit_form	B9
(qp) nop.m imm_{21}	m_unit_form	M37
(qp) nop.f imm_{21}	f_unit_form	F15
(qp) nop.x imm_{62}	x_unit_form	X1

説明： 何の操作も行われない。

ソフトウェアは、即値 imm_{21} または imm_{62} をプログラム・コード内でマーカとして使用できる。ハードウェアはこのマーカを無視する。

x_unit_form では、バンドルの L スロットに imm_{62} の上位 41 ビットが入る。

この命令には 5 つの形式があるが、これらの形式はそれぞれ特定の実行ユニット・タイプに対してしか実行できない。実行するユニット・タイプが重要でない場合は、擬似オペコードを使用できる。

操作：

```
if (PR[qp]) {
    ; // no operation
}
```

論理和 (Logical Or)

書式 :

(*qp*) or $r_1 = r_2, r_3$ register_form A1

(*qp*) or $r_1 = imm_8, r_3$ imm8_form A3

説明 :

2 つのソース・オペランドの論理和が取られ、結果が GR r_1 に格納される。レジスタ形式では、第 1 オペランドは GR r_2 であり、即値形式では、第 1 オペランドは imm_8 のエンコーディング・フィールドで与えられる。

操作 :

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src | GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

パック (Pack)

書式 :

(qp) pack2.sss $r_1 = r_2, r_3$ two_byte_form, signed_saturation_form I2
(qp) pack2.uss $r_1 = r_2, r_3$ two_byte_form, unsigned_saturation_form I2
(qp) pack4.sss $r_1 = r_2, r_3$ four_byte_form, signed_saturation_form I2

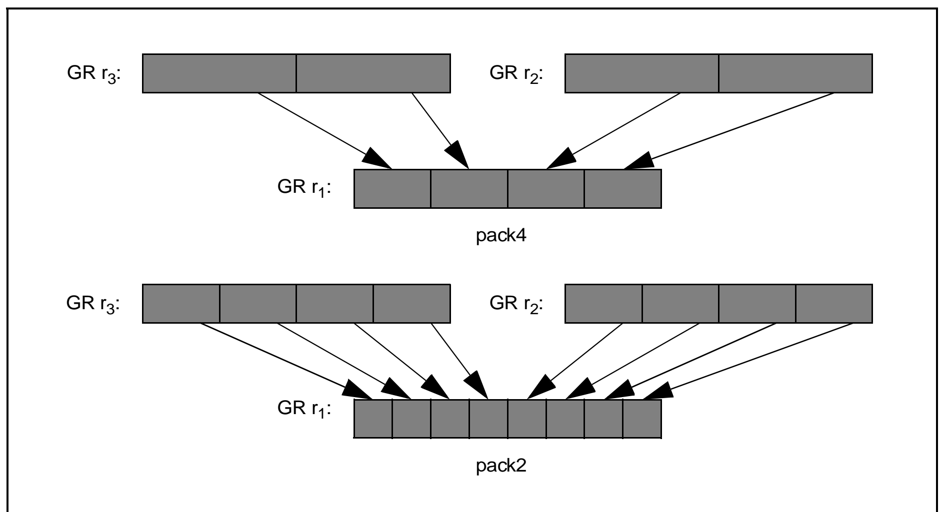
説明 :

GR r_2 および GR r_3 の 32 ビットまたは 16 ビット要素が、それぞれ、16 ビットまたは 8 ビットの要素に変換され、その結果が GR r_1 に格納される。ソース要素は符号付きの値として扱われる。ソース要素が結果の要素として表現できない場合は、飽和によるクリッピングが行われる。飽和は、符号付き、符号なしのいずれの場合もあり得る。要素が上限値より大きい場合は、結果は上限値になる。要素が下限値より小さい場合は、結果は下限値になる。飽和の上下限値を表 7-35 に示す。

表 7-35. パックでの飽和の上下限値

サイズ	ソース要素の幅	結果要素の幅	飽和の区別	上限値	下限値
2	16 ビット	8 ビット	符号付き	0x7f	0x80
2	16 ビット	8 ビット	符号なし	0xff	0x00
4	32 ビット	16 ビット	符号付き	0x7fff	0x8000

図 7-26. パックの操作



操作:

```
if (PR[qp]) {
    check_target_register(r1);

    if (two_byte_form) {                                    // two_byte_form
        if (signed_saturation_form) {                   // signed_saturation_form
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);
        } else {                                    // unsigned_saturation_form
            max = 0xff;
            min = 0x00;
        }
        temp[0] = sign_ext(GR[r2]{15:0},  16);
        temp[1] = sign_ext(GR[r2]{31:16}, 16);
        temp[2] = sign_ext(GR[r2]{47:32}, 16);
        temp[3] = sign_ext(GR[r2]{63:48}, 16);
        temp[4] = sign_ext(GR[r3]{15:0},  16);
        temp[5] = sign_ext(GR[r3]{31:16}, 16);
        temp[6] = sign_ext(GR[r3]{47:32}, 16);
        temp[7] = sign_ext(GR[r3]{63:48}, 16);

        for (i = 0; i < 8; i++) {
            if (temp[i] > max)
                temp[i] = max;

            if (temp[i] < min)
                temp[i] = min;
        }

        GR[r1] = concatenate8(temp[7], temp[6], temp[5], temp[4],
                              temp[3], temp[2], temp[1], temp[0]);

    } else {                                                // four_byte_form
        max = sign_ext(0x7fff, 16);                     // signed_saturation_form
        min = sign_ext(0x8000, 16);
        temp[0] = sign_ext(GR[r2]{31:0},  32);
        temp[1] = sign_ext(GR[r2]{63:32}, 32);
        temp[2] = sign_ext(GR[r3]{31:0},  32);
        temp[3] = sign_ext(GR[r3]{63:32}, 32);

        for (i = 0; i < 4; i++) {
            if (temp[i] > max)
                temp[i] = max;

            if (temp[i] < min)
                temp[i] = min;
        }

        GR[r1] = concatenate4(temp[3], temp[2], temp[1], temp[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

並列加算 (Parallel Add)

書式:

(qp) padd1 $r_1 = r_2, r_3$	one_byte_form, modulo_form	A9
(qp) padd1.sss $r_1 = r_2, r_3$	one_byte_form, sss_saturation_form	A9
(qp) padd1.uus $r_1 = r_2, r_3$	one_byte_form, uus_saturation_form	A9
(qp) padd1.uuu $r_1 = r_2, r_3$	one_byte_form, uuu_saturation_form	A9
(qp) padd2 $r_1 = r_2, r_3$	two_byte_form, modulo_form	A9
(qp) padd2.sss $r_1 = r_2, r_3$	two_byte_form, sss_saturation_form	A9
(qp) padd2.uus $r_1 = r_2, r_3$	two_byte_form, uus_saturation_form	A9
(qp) padd2.uuu $r_1 = r_2, r_3$	two_byte_form, uuu_saturation_form	A9
(qp) padd4 $r_1 = r_2, r_3$	four_byte_form, modulo_form	A9

説明: 2 つのソース・オペランドの要素セットが加え合わされ、その結果が GR r_1 に格納される。

2 つの要素の和が結果要素として表現できず、飽和コンプリータが指定されている場合は、飽和によるクリッピングが行われる。飽和は、表 7-36 に示すように、符号付き、符号なしのいずれの場合も指定できる。要素が上限値より大きい場合は、結果は上限値になる。要素が下限値より小さい場合は、結果は下限値になる。飽和の上下限値を表 7-37 に示す。

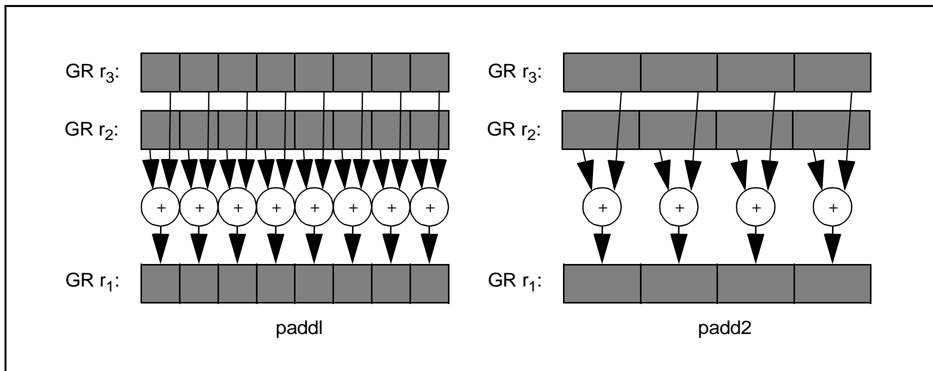
表 7-36. 並列加算の飽和コンプリータ

コンプリータ	結果 r_1 の処理	ソース r_2 の処理	ソース r_3 の処理
sss	符号付き	符号付き	符号付き
uus	符号なし	符号なし	符号付き
uuu	符号なし	符号なし	符号なし

表 7-37. 並列加算における飽和の制限

サイズ	要素幅	符号付きの結果 r_1		符号なしの結果 r_1	
		上限値	下限値	上限値	下限値
1	8 ビット	0x7f	0x80	0xff	0x00
2	16 ビット	0x7fff	0x8000	0xffff	0x0000

図 7-27. 並列加算の例



操作 :

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                          // one-byte elements
        x[0] = GR[r2]{7:0};          y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};         y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};        y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};        y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};        y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};        y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};        y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};        y[7] = GR[r3]{63:56};

        if (sss_saturation_form) {                                // sss_saturation_form
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);

            for (i = 0; i < 8; i++) {
                temp[i] = sign_ext(x[i], 8) + sign_ext(y[i], 8);
            }
        } else if (uus_saturation_form) {                         // uus_saturation_form
            max = 0xff;
            min = 0x00;

            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + sign_ext(y[i], 8);
            }
        } else if (uuu_saturation_form) {                         // uuu_saturation_form
            max = 0xff;
            min = 0x00;

            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
            }
        } else {                                                  // modulo_form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
            }
        }

        if (sss_saturation_form || uus_saturation_form || uuu_saturation_form) {
            for (i = 0; i < 8; i++) {
                if (temp[i] > max)
                    temp[i] = max;

                if (temp[i] < min)
                    temp[i] = min;
            }
```

```
        }
        GR[r1] = concatenate8(temp[7], temp[6], temp[5], temp[4],
                              temp[3], temp[2], temp[1], temp[0]);

    } else if (two_byte_form) {                         // 2-byte elements
        x[0] = GR[r2]{15:0};      y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};     y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};     y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};     y[3] = GR[r3]{63:48};

        if (sss_saturation_form) {                      // sss_saturation_form
            max = sign_ext(0x7fff, 16);
            min = sign_ext(0x8000, 16);

            for (i = 0; i < 4; i++) {
                temp[i] = sign_ext(x[i], 16) + sign_ext(y[i], 16);
            }
        } else if (uus_saturation_form) {               // uus_saturation_form
            max = 0xffff;
            min = 0x0000;

            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + sign_ext(y[i], 16);
            }
        } else if (uuu_saturation_form) {               // uuu_saturation_form
            max = 0xffff;
            min = 0x0000;

            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
            }
        } else {                                        // modulo_form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
            }
        }

        if (sss_saturation_form || uus_saturation_form || uuu_saturation_form) {
            for (i = 0; i < 4; i++) {
                if (temp[i] > max)
                    temp[i] = max;

                if (temp[i] < min)
                    temp[i] = min;
            }
        }
        GR[r1] = concatenate4(temp[3], temp[2], temp[1], temp[0]);

    } else {                                            // four-byte elements
        x[0] = GR[r2]{31:0};      y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};     y[1] = GR[r3]{63:32};

        for (i = 0; i < 2; i++) {                       // modulo_form
            temp[i] = zero_ext(x[i], 32) + zero_ext(y[i], 32);
        }

        GR[r1] = concatenate2(temp[1], temp[0]);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

並列平均 (Parallel Average)

書式 :

(qp) pavg1 $r_1 = r_2, r_3$	normal_form, one_byte_form	A9
(qp) pavg1.raz $r_1 = r_2, r_3$	raz_form, one_byte_form	A9
(qp) pavg2 $r_1 = r_2, r_3$	normal_form, two_byte_form	A9
(qp) pavg2.raz $r_1 = r_2, r_3$	raz_form, two_byte_form	A9

説明 : GR r_2 の符号なしの各データ要素が GR r_3 の対応する符号なしデータ要素に加算され、次にこの加算結果がそれぞれ独立に右に 1 ビットだけシフトされる。各要素の最上位ビットには加算のキャリー・ビットが埋められる。丸め誤差を累積させないために、平均操作が行われる。符号なしの結果が GR r_1 に格納される。

平均操作は次のとおりである。normal_form では、対応する加算結果の最下位 2 ビットの少なくともいずれかが 1 の場合、それぞれの加算結果の最下位ビットが 1 に設定される。raz_form (ゼロから離れる方向の丸め形式) では、それぞれの加算結果に 1 を加算して、平均がゼロから丸められる。

図 7-28. 並列平均の例

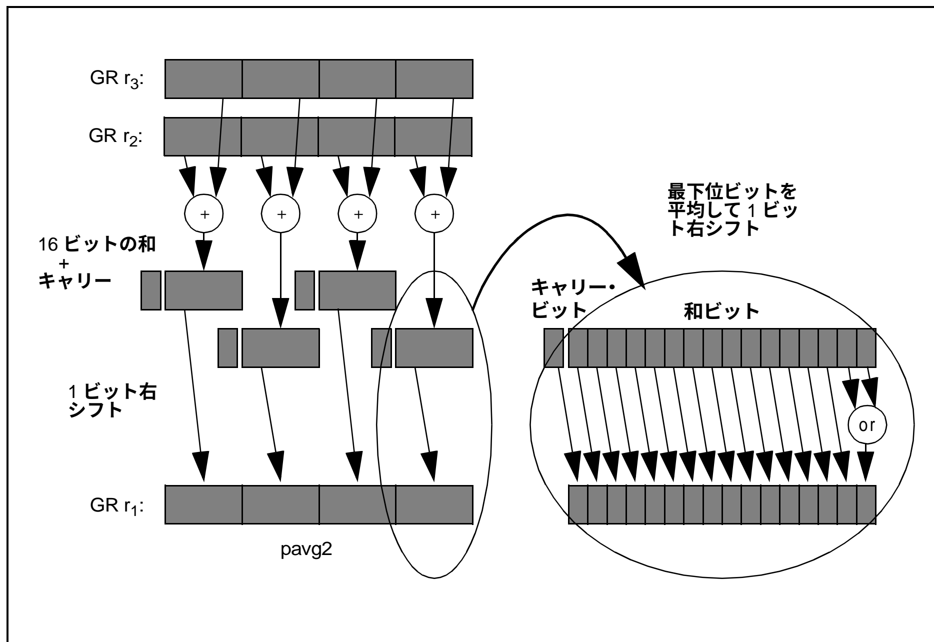
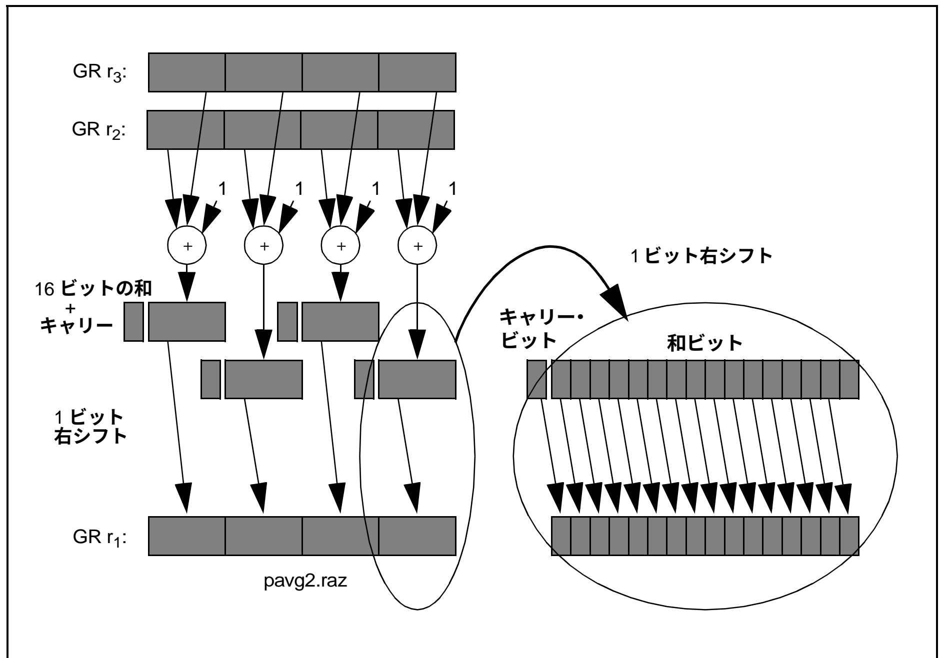


図 7-29. 並列平均でのゼロから離れる方向の丸めの例



操作:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                        // one_byte_form
        x[0] = GR[r2]{7:0};          y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};         y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};        y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};        y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};        y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};        y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};        y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};        y[7] = GR[r3]{63:56};

        if (raz_form) {
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8) + 1;
                res[i] = shift_right_unsigned(temp[i], 1);
            }
        } else {                                                // normal form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
                res[i] = shift_right_unsigned(temp[i], 1) | (temp[i]{0});
            }
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);

    } else {                                                    // two_byte_form
        x[0] = GR[r2]{15:0};         y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};        y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};        y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};        y[3] = GR[r3]{63:48};

        if (raz_form) {
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16) + 1;
                res[i] = shift_right_unsigned(temp[i], 1);
            }
        } else {                                                // normal form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
                res[i] = shift_right_unsigned(temp[i], 1) | (temp[i]{0});
            }
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

並列平均減算 (Parallel Average Subtract)

書式:
(*qp*) pavgsub1 $r_1 = r_2, r_3$ one_byte_form A9
(*qp*) pavgsub2 $r_1 = r_2, r_3$ two_byte_form A9

説明:
GR r_3 の符号なしの各データ要素が GR r_2 の対応する符号なしデータ要素から減算され、次にこの減算結果がそれぞれ独立に右に 1 ビット位置だけシフトされる。各要素の最上位ビットには減算のボロー・ビット (ALU のキャリーの補数) が埋められる。丸め誤差を累積させないために、平均操作が行われる。対応する減算結果の最下位 2 ビットの少なくともいずれかが 1 の場合、それぞれの減算結果の最下位ビットが 1 に設定される。符号付きの結果が GR r_1 に格納される。

図 7-30. 並列平均減算の例

操作 :

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                              // one_byte_form
        x[0] = GR[r2]{7:0};         y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};        y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};       y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};       y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};       y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};       y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};       y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};       y[7] = GR[r3]{63:56};

        for (i = 0; i < 8; i++) {
            temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            res[i] = (temp[i]{8:0} u>> 1) | (temp[i]{0});
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                             res[3], res[2], res[1], res[0]);

    } else {                                                          // two_byte_form
        x[0] = GR[r2]{15:0};        y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};       y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};       y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};       y[3] = GR[r3]{63:48};

        for (i = 0; i < 4; i++) {
            temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            res[i] = (temp[i]{16:0} u>> 1) | (temp[i]{0});
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

並列比較 (Parallel Compare)

書式 :

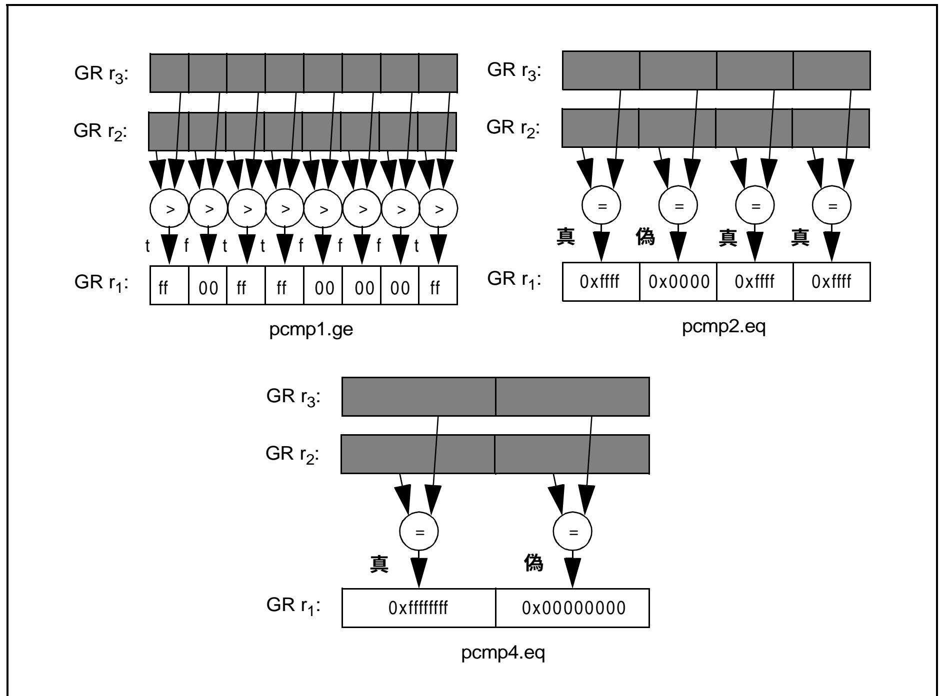
(qp) pcmp1.*prel* $r_1 = r_2, r_3$	one_byte_form	A9
(qp) pcmp2.*prel* $r_1 = r_2, r_3$	two_byte_form	A9
(qp) pcmp4.*prel* $r_1 = r_2, r_3$	four_byte_form	A9

説明 : 2 つのソース・オペランドが、表 7-38 に示す 12 種類の関係のいずれかについて比較される。GR r_2 と GR r_3 の対応するデータ要素について比較条件が真である場合は、GR r_1 の対応するデータ要素がすべて 1 ビットに設定される。比較条件が偽である場合は、GR r_1 の対応するデータ要素がすべて 0 ビットに設定される。'>' の関係については、両オペランドが符号付きとして解釈される。

表 7-38. 並列の比較関係

prel	比較関係 (r_2 *prel* r_3)
eq	$r_2 == r_3$
gt	$r_2 > r_3$ (符号付き)

図 7-31. 並列比較の例



操作:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                        // one-byte elements
        x[0] = GR[r2]{7:0};         y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};        y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};       y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};       y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};       y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};       y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};       y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};       y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            if (prel == 'eq')
                tmp_rel = x[i] == y[i];
            else
                tmp_rel = greater_signed(sign_ext(x[i], 8), sign_ext(y[i], 8));

            if (tmp_rel)
                res[i] = 0xff;
            else
                res[i] = 0x00;
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);
    } else if (two_byte_form) {                                 // two-byte elements
        x[0] = GR[r2]{15:0};        y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};       y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};       y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};       y[3] = GR[r3]{63:48};
        for (i = 0; i < 4; i++) {
            if (prel == 'eq')
                tmp_rel = x[i] == y[i];
            else
                tmp_rel = greater_signed(sign_ext(x[i], 16), sign_ext(y[i], 16));

            if (tmp_rel)
                res[i] = 0xffff;
            else
                res[i] = 0x0000;
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    } else {                                                    // four-byte elements
        x[0] = GR[r2]{31:0};        y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};       y[1] = GR[r3]{63:32};
        for (i = 0; i < 2; i++) {
            if (prel == 'eq')
                tmp_rel = x[i] == y[i];
            else
                tmp_rel = greater_signed(sign_ext(x[i], 32), sign_ext(y[i], 32));

            if (tmp_rel)
                res[i] = 0xffffffff;
            else
                res[i] = 0x00000000;
        }
        GR[r1] = concatenate2(res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

並列最大値 (Parallel Maximum)

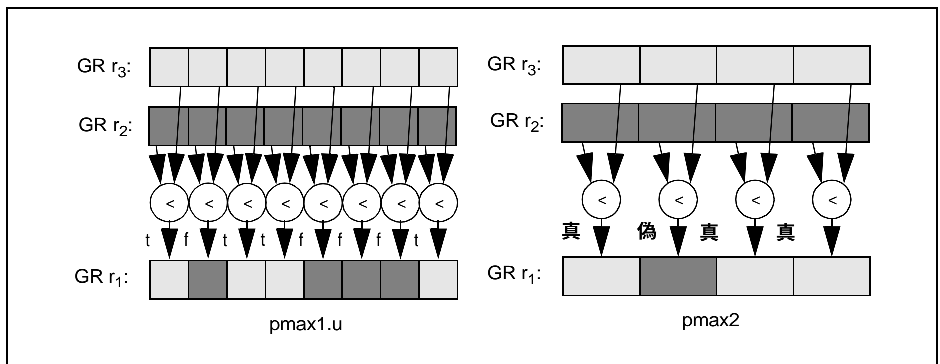
書式:

(qp) pmax1.u $r_1 = r_2, r_3$ one_byte_form   I2
(qp) pmax2 $r_1 = r_2, r_3$ two_byte_form   I2

説明:

2 つのソース・オペランドの最大値が結果レジスタに格納される。one_byte_form では、GR r_2 の符号なし 8 ビットの各要素が GR r_3 の対応する符号なし 8 ビット要素と比較され、要素の大きい方が GR r_1 の対応する 8 ビット要素に格納される。two_byte_form では、GR r_2 の符号付き 16 ビットの各要素が GR r_3 の対応する符号付き 16 ビット要素と比較され、要素の大きい方が GR r_1 の対応する 16 ビット要素に格納される。

図 7-32. 並列最大値の例



操作:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                    // one-byte elements
        x[0] = GR[r2]{7:0};         y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};        y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};       y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};       y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};       y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};       y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};       y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};       y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            res[i] = (zero_ext(x[i],8) < zero_ext(y[i],8)) ? y[i] : x[i];
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);
    } else {                                                // two-byte elements
        x[0] = GR[r2]{15:0};        y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};       y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};       y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};       y[3] = GR[r3]{63:48};
        for (i = 0; i < 4; i++) {
            res[i] = (sign_ext(x[i],16) < sign_ext(y[i],16)) ? y[i] :
x[i];
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

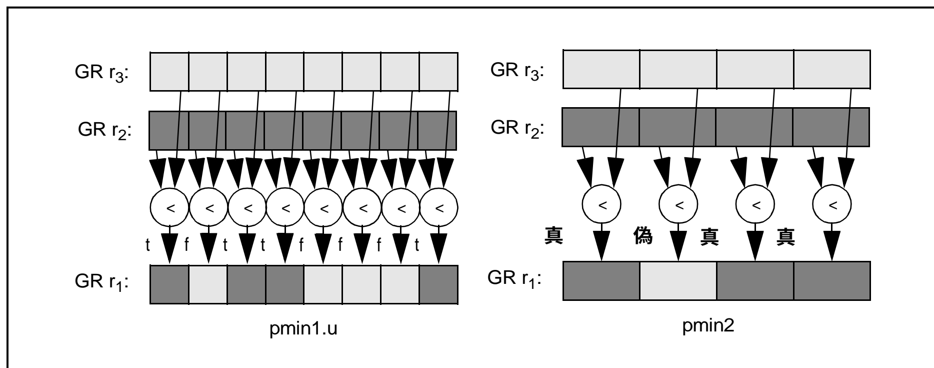
並列最小値 (Parallel Minimum)

書式:

(qp) pmin1.u $r_1 = r_2, r_3$	one_byte_form	I2
(qp) pmin2 $r_1 = r_2, r_3$	two_byte_form	I2

説明: 2 つのソース・オペランドの最小値が結果レジスタに格納される。one_byte_form では、GR r_2 の符号なし 8 ビットの各要素が GR r_3 の対応する符号なし 8 ビット要素と比較され、要素の小さい方が GR r_1 の対応する 8 ビット要素に格納される。two_byte_form では、GR r_2 の符号付き 16 ビットの各要素が GR r_3 の対応する符号付き 16 ビット要素と比較され、要素の小さい方が GR r_1 の対応する 16 ビット要素に格納される。

図 7-33. 並列最小値の例



操作:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                    // one-byte elements
        x[0] = GR[r2]{7:0};         y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};        y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};       y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};       y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};       y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};       y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};       y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};       y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            res[i] = (zero_ext(x[i],8) < zero_ext(y[i],8)) ? x[i] : y[i];
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                             res[3], res[2], res[1], res[0]);
    } else {                                                // two-byte elements
        x[0] = GR[r2]{15:0};        y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};       y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};       y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};       y[3] = GR[r3]{63:48};
        for (i = 0; i < 4; i++) {
            res[i] = (sign_ext(x[i],16) < sign_ext(y[i],16)) ? x[i] :
y[i];
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

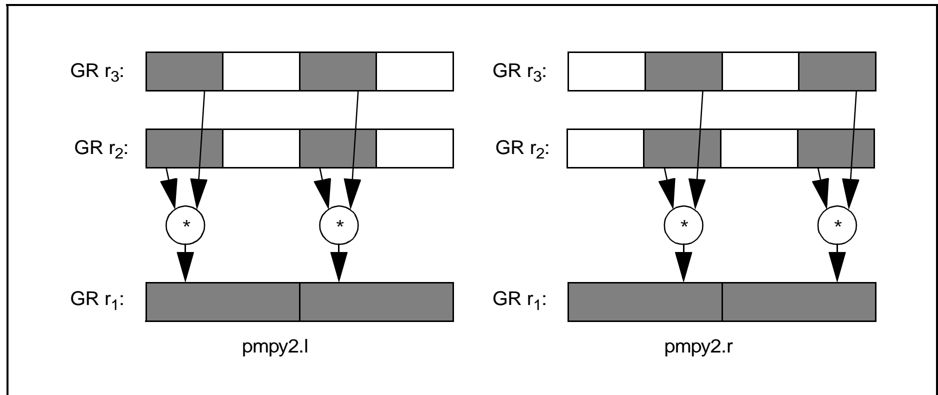
並列乗算 (Parallel Multiply)

書式 :

(qp)	pmpy2.r $r_1 = r_2, r_3$	right_form	I2
(qp)	pmpy2.l $r_1 = r_2, r_3$	left_form	I2

説明 : GR r_2 の 2 つの符号付き 16 ビット・データ要素が、図 7-34 に示すように、GR r_3 の対応する 2 つの符号付き 16 ビット・データ要素と乗算される。2 つの 32 ビットの結果が GR r_1 に格納される。

図 7-34. 並列乗算の操作



操作 :

```
if (PR[qp]) {
    check_target_register(r1);

    if (right_form) {
        GR[r1]{31:0} = sign_ext(GR[r2]{15:0}, 16) * sign_ext(GR[r3]{15:0},
16);
        GR[r1]{63:32} = sign_ext(GR[r2]{47:32}, 16) *
sign_ext(GR[r3]{47:32}, 16);
    } else {                                            // left_form
        GR[r1]{31:0} = sign_ext(GR[r2]{31:16}, 16) *
sign_ext(GR[r3]{31:16}, 16);
        GR[r1]{63:32} = sign_ext(GR[r2]{63:48}, 16) *
sign_ext(GR[r3]{63:48}, 16);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

並列乗算および右シフト (Parallel Multiply and Shift Right)

書式 :

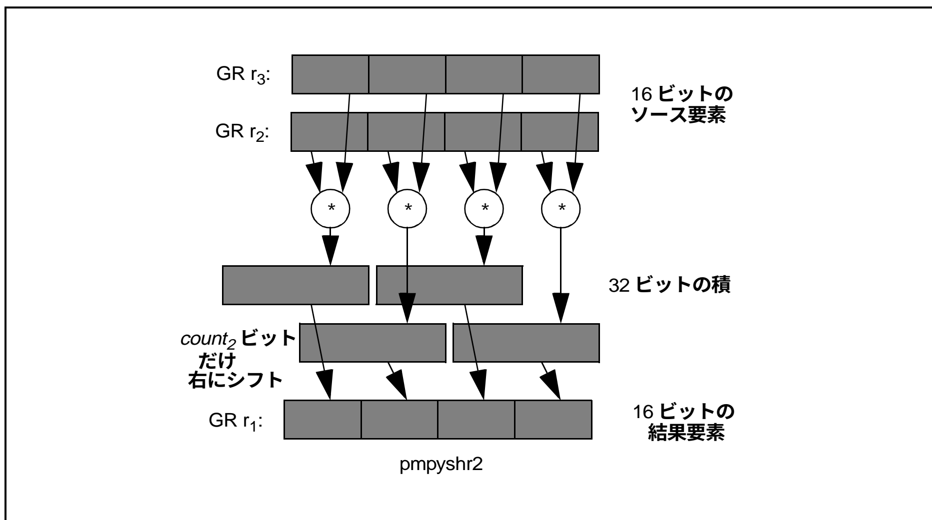
(qp) pmpyshr2 $r_1 = r_2, r_3, count_2$	signed_form	I1
(qp) pmpyshr2.u $r_1 = r_2, r_3, count_2$	unsigned_form	I1

説明 : GR r_2 の 4 つの 16 ビット・データ要素が、図 7-35 に示すように、GR r_3 の対応する 4 つの 16 ビット・データ要素と乗算される。この乗算には、符号付き (pmpyshr2) と符号なし (pmpyshr2.u) を指定できる。次に、各積が $count_2$ ビットだけ右にシフトされ、シフトされたそれぞれの積の最下位 16 ビットから 4 つの 16 ビットの結果が形成される。それらの結果が GR r_1 に格納される。$count_2$ が 0 である場合は、各結果は積の下位 16 ビットになり、$count_2$ が 16 の場合は、各結果は積の上位 16 ビットになる。$count_2$ に使用できる値を表 7-39 に示す。

表 7-39. pmpyshr2 のシフト・オプション

$count_2$	32 ビットの積から選択されるビット・フィールド
0	15:0
7	22:7
15	30:15
16	31:16

図 7-35. 並列乗算右シフトの操作



操作：

```
if (PR[qp]) {
    check_target_register(r1);
    x[0] = GR[r2]{15:0};        y[0] = GR[r3]{15:0};
    x[1] = GR[r2]{31:16};       y[1] = GR[r3]{31:16};
    x[2] = GR[r2]{47:32};       y[2] = GR[r3]{47:32};
    x[3] = GR[r2]{63:48};       y[3] = GR[r3]{63:48};
    for (i = 0; i < 4; i++) {
        if (unsigned_form)                          // unsigned multiplication
            temp[i] = zero_ext(x[i], 16) * zero_ext(y[i], 16);
        else                                        // signed multiplication
            temp[i] = sign_ext(x[i], 16) * sign_ext(y[i], 16);

        res[i] = temp[i]{(count2 + 15):count2};
    }

    GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

ポピュレーション・カウント (Population Count)

書式 : (qp) popcnt $r_1 = r_3$ I9

説明 : GR r_3 の値が 1 であるビットがカウントされ、結果の合計値が GR r_1 に格納される。

操作 :

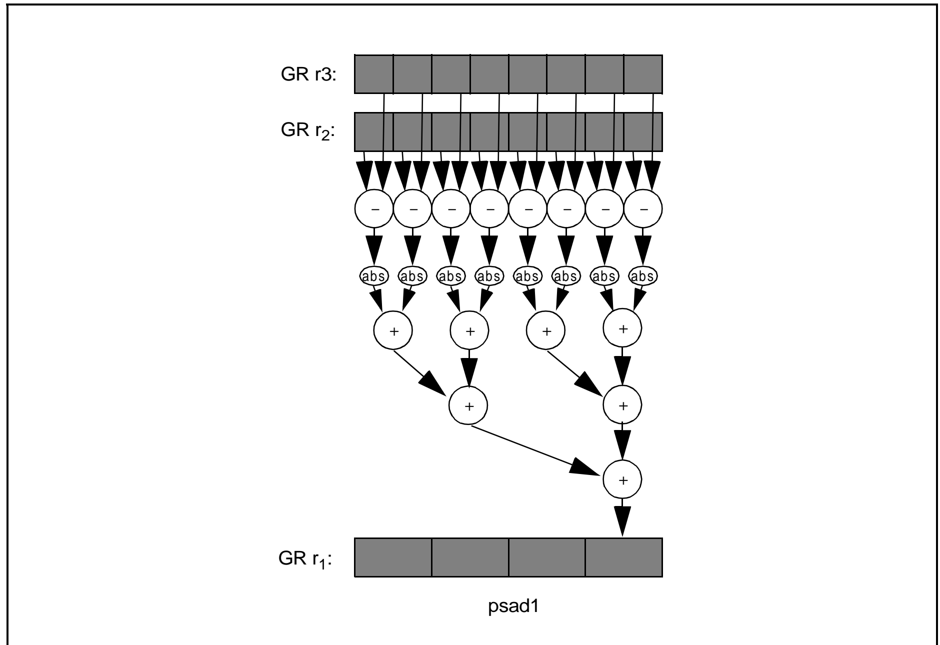
```
if (PR[qp]) {
    check_target_register(r1);

    res = 0;
    // Count up all the one bits
    for (i = 0; i < 64; i++) {
        res += GR[r3]{i};
    }

    GR[r1] = res;
    GR[r1].nat = GR[r3].nat;
}
```

並列絶対差累計 (Parallel Sum of Absolute Difference)

書式: (qp) psad1 $r_1 = r_2, r_3$ I2

説明: GR r_2 の符号なし 8 ビットの各要素が GR r_3 の対応する符号なし 8 ビット要素から減算される。各減算結果の絶対値がすべての要素について累計され、GR r_1 に格納される。

図 7-36. 並列絶対差累計の例



操作:

```
if (PR[qp]) {
    check_target_register(r1);

    x[0] = GR[r2]{7:0};          y[0] = GR[r3]{7:0};
    x[1] = GR[r2]{15:8};         y[1] = GR[r3]{15:8};
    x[2] = GR[r2]{23:16};        y[2] = GR[r3]{23:16};
    x[3] = GR[r2]{31:24};        y[3] = GR[r3]{31:24};
    x[4] = GR[r2]{39:32};        y[4] = GR[r3]{39:32};
    x[5] = GR[r2]{47:40};        y[5] = GR[r3]{47:40};
    x[6] = GR[r2]{55:48};        y[6] = GR[r3]{55:48};
    x[7] = GR[r2]{63:56};        y[7] = GR[r3]{63:56};

    GR[r1] = 0;
    for (i = 0; i < 8; i++) {
        temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
        if (temp[i] < 0)
            temp[i] = -temp[i];
        GR[r1] += temp[i];
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

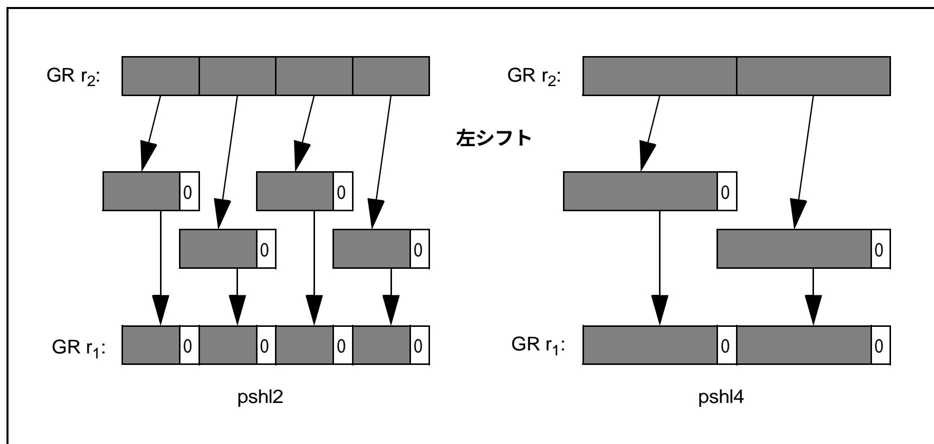
並列右シフト (Parallel Shift Left)

書式 :

(qp) pshl2 $r_1 = r_2, r_3$	two_byte_form, variable_form	I7
(qp) pshl2 $r_1 = r_2, count_5$	two_byte_form, fixed_form	I8
(qp) pshl4 $r_1 = r_2, r_3$	four_byte_form, variable_form	I7
(qp) pshl4 $r_1 = r_2, count_5$	four_byte_form, fixed_form	I8

説明 : GR r_2 の各データ要素が、GR r_3 または即値フィールド $count_5$ のスカラ・シフト・カウントだけ、それぞれ独立に左にシフトされる。各要素の下位ビットにはゼロが埋められる。シフト・カウントは符号なしとして解釈される。シフト・カウントが 15 (16 ビットの数値の場合) あるいは 31 (32 ビットの数値の場合) を超える場合は、結果はすべてゼロ・ビットになる。結果は GR r_1 に格納される。

図 7-37. 並列右シフトの例



操作 :

```
if (PR[qp]) {
    check_target_register(r1);

    shift_count = (variable_form ? GR[r3]  : count5);
    tmp_nat = (variable_form ? GR[r3].nat : 0);

    if (two_byte_form) {                                  // two_byte_form
        if (shift_count u> 16)
            shift_count = 16;
        GR[r1]{15:0}  = GR[r2]{15:0}  << shift_count;
        GR[r1]{31:16} = GR[r2]{31:16} << shift_count;
        GR[r1]{47:32} = GR[r2]{47:32} << shift_count;
        GR[r1]{63:48} = GR[r2]{63:48} << shift_count;
    } else {                                              // four_byte_form
        if (shift_count u> 32)
            shift_count = 32;
        GR[r1]{31:0}  = GR[r2]{31:0}  << shift_count;
        GR[r1]{63:32} = GR[r2]{63:32} << shift_count;
    }

    GR[r1].nat = GR[r2].nat || tmp_nat;
}
```

並列左シフトおよび加算 (Parallel Shift Left and Add)

書式 : (*qp*) pshladd2 $r_1 = r_2$, $count_2$, r_3 A10

説明 : GR r_2 の 4 つの符号付き 16 ビット・データ要素が、$count_2$ ビットだけ、それぞれ独立に左にシフト (下位ビットにゼロがシフト・イン) され、GR r_3 の 4 つの符号付き 16 ビット・データ要素に加算される。左シフトと加算の両操作は共に飽和する。つまり、シフトまたは加算の結果が符号付き 16 ビット値として表現できない場合には、最終結果が飽和する。4 つの符号付き 16 ビット結果は GR r_1 に格納される。第 1 オペランドに対してシフトできるビット数は、1、2、あるいは 3 に限られる。

操作 :

```
if (PR[qp]) {
    check_target_register(r1);

    x[0] = GR[r2]{15:0};        y[0] = GR[r3]{15:0};
    x[1] = GR[r2]{31:16};       y[1] = GR[r3]{31:16};
    x[2] = GR[r2]{47:32};       y[2] = GR[r3]{47:32};
    x[3] = GR[r2]{63:48};       y[3] = GR[r3]{63:48};

    max = sign_ext(0x7fff, 16);
    min = sign_ext(0x8000, 16);

    for (i = 0; i < 4; i++) {
        temp[i] = sign_ext(x[i], 16) << count2;

        if (temp[i] > max)
            res[i] = max;
        else if (temp[i] < min)
            res[i] = min;
        else {
            res[i] = temp[i] + sign_ext(y[i], 16);
            if (res[i] > max)
                res[i] = max;
            if (res[i] < min)
                res[i] = min;
        }
    }

    GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

並列右シフト (Parallel Shift Right)

書式 :

(qp) pshr2 $r_1 = r_3, r_2$	signed_form, two_byte_form, variable_form	I5
(qp) pshr2 $r_1 = r_3, count_5$	signed_form, two_byte_form, fixed_form	I6
(qp) pshr2.u $r_1 = r_3, r_2$	unsigned_form, two_byte_form, variable_form	I5
(qp) pshr2.u $r_1 = r_3, count_5$	unsigned_form, two_byte_form, fixed_form	I6
(qp) pshr4 $r_1 = r_3, r_2$	signed_form, four_byte_form, variable_form	I5
(qp) pshr4 $r_1 = r_3, count_5$	signed_form, four_byte_form, fixed_form	I6
(qp) pshr4.u $r_1 = r_3, r_2$	unsigned_form, four_byte_form, variable_form	I5
(qp) pshr4.u $r_1 = r_3, count_5$	unsigned_form, four_byte_form, fixed_form	I6

説明 :

GR r_2 の各データ要素が、GR r_3 または即値フィールド $count_5$ のスカラ・シフト・カウントだけ、それぞれ独立に右にシフトされる。各要素の上位ビットには、算術シフトの場合は GR r_3 の各データ要素の符号ビットの初期値が埋められ、論理シフトの場合はゼロが埋められる。シフト・カウントは符号なしとして解釈される。シフト・カウントが 15 (16 ビットの数値の場合) あるいは 31 (32 ビットの数値の場合) を超える場合は、GR r_3 の各データ要素の符号ビットの初期値、および符号付き、符号なしのどちらのシフトが行われたかによって、結果はすべて 0 ビットかすべて 1 ビットになる。結果は GR r_1 に格納される。

操作 :

```
if (PR[qp]) {
    check_target_register(r1);

    shift_count = (variable_form ? GR[r2] : count5);
    tmp_nat = (variable_form ? GR[r2].nat : 0);

    if (two_byte_form) {                                       // two_byte_form
        if (shift_count u> 16)
            shift_count = 16;
        if (unsigned_form) {                                   // unsigned
shift
            GR[r1]{15:0}  = shift_right_unsigned(zero_ext(GR[r3]{15:0},  16),
                                                          shift_count);
            GR[r1]{31:16} = shift_right_unsigned(zero_ext(GR[r3]{31:16}, 16),
                                                          shift_count);
            GR[r1]{47:32} = shift_right_unsigned(zero_ext(GR[r3]{47:32}, 16),
                                                          shift_count);
            GR[r1]{63:48} = shift_right_unsigned(zero_ext(GR[r3]{63:48}, 16),
                                                          shift_count);
        } else {                                               // signed shift
            GR[r1]{15:0}  = shift_right_signed(sign_ext(GR[r3]{15:0},  16),
                                                          shift_count);
            GR[r1]{31:16} = shift_right_signed(sign_ext(GR[r3]{31:16}, 16),
                                                          shift_count);
            GR[r1]{47:32} = shift_right_signed(sign_ext(GR[r3]{47:32}, 16),
                                                          shift_count);
            GR[r1]{63:48} = shift_right_signed(sign_ext(GR[r3]{63:48}, 16),
                                                          shift_count);
        }
    } else {                                                   //
four_byte_form
        if (shift_count > 32)
            shift_count = 32;
        if (unsigned_form) {                                   // unsigned
shift
            GR[r1]{31:0}  = shift_right_unsigned(zero_ext(GR[r3]{31:0},  32),
                                                          shift_count);
            GR[r1]{63:32} = shift_right_unsigned(zero_ext(GR[r3]{63:32}, 32),
                                                          shift_count);
        } else {                                               // signed shift
            GR[r1]{31:0}  = shift_right_signed(sign_ext(GR[r3]{31:0},  32),
```

```
                                                             shift_count);
            GR[r1]{63:32} = shift_right_signed(sign_ext(GR[r3]{63:32}, 32),
                                                             shift_count);
        }
    }

    GR[r1].nat = GR[r3].nat || tmp_nat;
}
```

並列右シフトおよび加算 (Parallel Shift Right and Add)

書式: (*qp*) pshradd2 $r_1 = r_2, count_2, r_3$ A10

説明: GR r_2 の 4 つの符号付き 16 ビット・データ要素が、$count_2$ ビットだけ、それぞれ独立に右にシフトされ、GR r_3 の 4 つの符号付き 16 ビット・データ要素に加算される。この右シフト操作では、各要素の上位ビットに GR r_2 の各データ要素の符号ビットの初期値が埋められる。加算操作の実行には符号付きの飽和が行われる。4 つの符号付き 16 ビットの加算結果は GR r_1 に格納される。第 1 オペランドに対してシフトできるビット数は、1、2、あるいは 3 に限られる。

操作:

```
if (PR[qp]) {
    check_target_register(r1);

    x[0] = GR[r2]{15:0};        y[0] = GR[r3]{15:0};
    x[1] = GR[r2]{31:16};       y[1] = GR[r3]{31:16};
    x[2] = GR[r2]{47:32};       y[2] = GR[r3]{47:32};
    x[3] = GR[r2]{63:48};       y[3] = GR[r3]{63:48};

    max = sign_ext(0x7fff, 16);
    min = sign_ext(0x8000, 16);

    for (i = 0; i < 4; i++) {
        temp[i] = shift_right_signed(sign_ext(x[i], 16), count2);

        res[i] = temp[i] + sign_ext(y[i], 16);
        if (res[i] > max)
            res[i] = max;
        if (res[i] < min)
            res[i] = min;
    }

    GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

並列減算 (Parallel Subtract)

書式:

(qp) psub1 $r_1 = r_2, r_3$	one_byte_form, modulo_form A9
(qp) psub1.sss $r_1 = r_2, r_3$	one_byte_form, sss_saturation_form A9
(qp) psub1.uus $r_1 = r_2, r_3$	one_byte_form, uus_saturation_form A9
(qp) psub1.uuu $r_1 = r_2, r_3$	one_byte_form, uuu_saturation_form A9
(qp) psub2 $r_1 = r_2, r_3$	two_byte_form, modulo_form A9
(qp) psub2.sss $r_1 = r_2, r_3$	two_byte_form, sss_saturation_form A9
(qp) psub2.uus $r_1 = r_2, r_3$	two_byte_form, uus_saturation_form A9
(qp) psub2.uuu $r_1 = r_2, r_3$	two_byte_form, uuu_saturation_form A9
(qp) psub4 $r_1 = r_2, r_3$	four_byte_form, modulo_form A9

説明: 2 つのソース・オペランドのの要素セット間の減算が行われ、結果が GR r_1 に格納される。

対応する 2 つの要素間の差が結果要素として表現できないで、かつ飽和コンプリータが指定されている場合は、飽和によるクリッピングが行われる。飽和は、表 7-40 に示すように、符号付き、符号なしを指定できる。2 つの要素の差が上限値より大きい場合は、結果は上限値になる。下限値より小さい場合は、結果は下限値になる。この操作の飽和の上下限値を表 7-41 に示す。

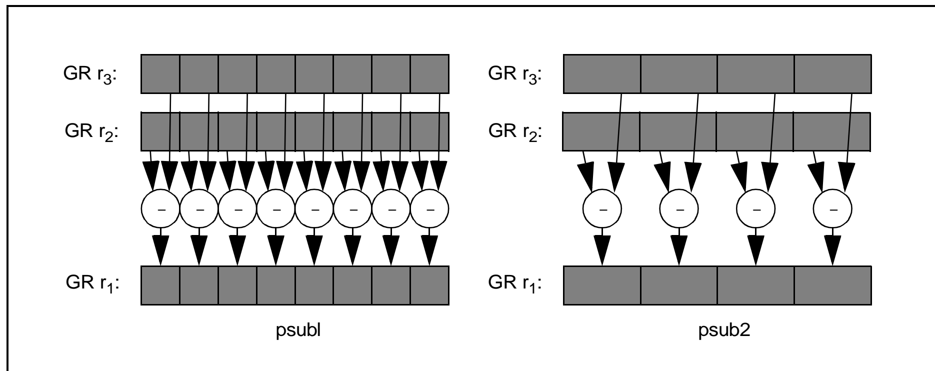
表 7-40. 並列減算の飽和コンプリータ

コンプリータ	結果 r_1 の処理	ソース r_2 の処理	ソース r_3 の処理
sss	符号付き	符号付き	符号付き
uus	符号なし	符号なし	符号付き
uuu	符号なし	符号なし	符号なし

表 7-41. 並列減算での飽和の上下限値

サイズ	要素幅	符号付きの結果 r_1		符号なしの結果 r_1	
		上限値	下限値	上限値	下限値
1	8 bit	0x7f	0x80	0xff	0x00
2	16 bit	0x7fff	0x8000	0xffff	0x0000

図 7-38. 並列減算の例



操作 :

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                    // one-byte elements
        x[0] = GR[r2]{7:0};        y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};       y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};      y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};      y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};      y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};      y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};      y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};      y[7] = GR[r3]{63:56};

        if (sss_saturation_form) {                          // sss_saturation_form
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);
            for (i = 0; i < 8; i++) {
                temp[i] = sign_ext(x[i], 8) - sign_ext(y[i], 8);
            }
        } else if (uus_saturation_form) {                   // uus_saturation_form
            max = 0xff;
            min = 0x00;
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - sign_ext(y[i], 8);
            }
        } else if (uuu_saturation_form) {                   // uuu_saturation_form
            max = 0xff;
            min = 0x00;
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            }
        } else {                                            // modulo_form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            }
        }

        if (sss_saturation_form || uus_saturation_form || uuu_saturation_form) {
            for (i = 0; i < 8; i++) {
                if (temp[i] > max)
                    temp[i] = max;
                if (temp[i] < min)
                    temp[i] = min;
            }
        }

        GR[r1] = concatenate8(temp[7], temp[6], temp[5], temp[4],
                              temp[3], temp[2], temp[1], temp[0]);
```

```
    } else if (two_byte_form) {                             // two-byte elements
        x[0] = GR[r2]{15:0};       y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};      y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};      y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};      y[3] = GR[r3]{63:48};

        if (sss_saturation_form) {                          // sss_saturation_form
            max = sign_ext(0x7fff, 16);
            min = sign_ext(0x8000, 16);
            for (i = 0; i < 4; i++) {
                temp[i] = sign_ext(x[i], 16) - sign_ext(y[i], 16);
            }
        } else if (uus_saturation_form) {                   // uus_saturation_form
            max = 0xffff;
            min = 0x0000;
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - sign_ext(y[i], 16);
            }
        } else if (uuu_saturation_form) {                   // uuu_saturation_form
            max = 0xffff;
            min = 0x0000;
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            }
        } else {                                            // modulo_form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            }
        }

        if (sss_saturation_form || uus_saturation_form || uuu_saturation_form) {
            for (i = 0; i < 4; i++) {
                if (temp[i] > max)
                    temp[i] = max;
                if (temp[i] < min)
                    temp[i] = min;
            }
        }

        GR[r1] = concatenate4(temp[3], temp[2], temp[1], temp[0]);
    } else {                                                // four-byte elements
        x[0] = GR[r2]{31:0};       y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};      y[1] = GR[r3]{63:32};

        for (i = 0; i < 2; i++) {                           // modulo_form
            temp[i] = zero_ext(x[i], 32) - zero_ext(y[i], 32);
        }

        GR[r1] = concatenate2(temp[1], temp[0]);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

ユーザ・マスクのリセット (Reset User Mask)

書式 : (*qp*) rum imm_{24} M44

説明 : imm_{24} オペランドの補数とユーザ・マスク (PSR{5:0}) との論理積が取られ、その結果がユーザ・マスクに格納される。

PSR.up は、セキュリティ・パフォーマンス・モニタ・ビット (PSR.sp) がゼロの場合に限りクリアされる。そうでない場合は、RSP.up は変更されない

操作 :

```
if (PR[qp]) {
    if (is_reserved_field(PSR_TYPE, PSR_UM, imm24))
        reserved_register_field_fault();

    if (imm24{1})       PSR{1} = 0;
    if (imm24{2} && PSR.sp == 0)            //non-secure perf monitor
                        PSR{2} = 0;
    if (imm24{3})       PSR{3} = 0;
    if (imm24{4})       PSR{4} = 0;
    if (imm24{5})       PSR{5} = 0;
}
```

浮動小数点数、指数、仮数の設定 (Set Floating-Point Value, Exponent, or Significand)

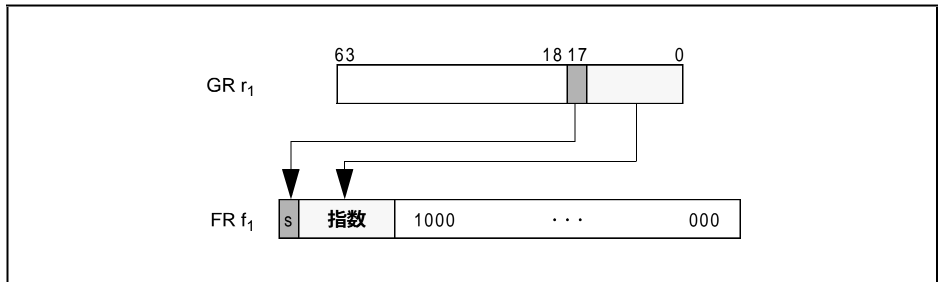
書式 :

(*qp*) setf.s $f_1 = r_2$	single_form	M18
(*qp*) setf.d $f_1 = r_2$	double_form	M18
(*qp*) setf.exp $f_1 = r_2$	exponent_form	M18
(*qp*) setf.sig $f_1 = r_2$	significand_form	M18

説明 : 単精度および倍精度形式では、GR r_2 がそれぞれ単精度 (single_form の場合) または倍精度 (double_form の場合) のメモリ表現として扱われ、浮動小数点レジスタ形式に変換され、FR f_1 に格納される。

exponent_from では、GR r_2 のビット 16:0 が FR f_1 の指数フィールドにコピーされ、GR r_2 のビット 17 が FR f_1 の符号ビットにコピーされる。FR f_1 の仮数フィールドは 1 (0x800...000) に設定される。

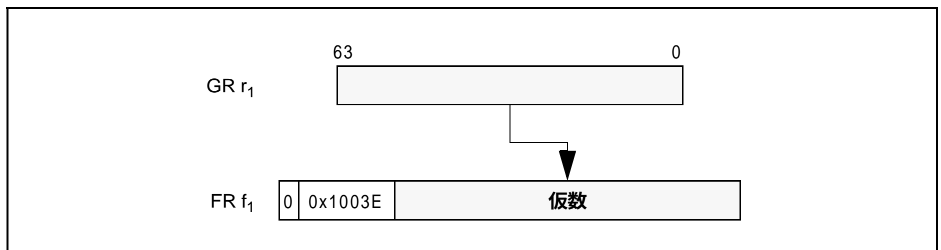
図 7-39. setf.exp の機能



significand_form では、GR r_2 の値が FR f_1 の仮数フィールドにコピーされる。

FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

図 7-40. setf.sig の機能



すべての形式について、r_2 に対応する NaT ビットが 1 の場合は、FR f_1 は計算結果ではなく NatVal に設定される。

操作:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (!GR[r2].nat) {
        if (single_form)
            FR[f1] = fp_mem_to_fr_format(GR[r2], 4, 0);
        else if (double_form)
            FR[f1] = fp_mem_to_fr_format(GR[r2], 8, 0);
        else if (significand_form) {
            FR[f1].significand = GR[r2];
            FR[f1].exponent = FP_INTEGER_EXP;
            FR[f1].sign = 0;
        } else {                                              // exponent_form
            FR[f1].significand = 0x8000000000000000;
            FR[f1].exp = GR[r2]{16:0};
            FR[f1].sign = GR[r2]{17};
        }
    } else
        FR[f1] = NATVAL;

    fp_update_psr(f1);
}
```

左シフト (Shift Left)

書式：

(qp) shl $r_1 = r_2, r_3$ I7

(qp) shl $r_1 = r_2, count_6$ pseudo-op of: (qp) dep.z $r_1 = r_2, count_6$, 64–$count_6$

説明：

GR r_2 の値が左にシフトされ、空きになったビット位置にゼロが埋められ、GR r_1 に格納される。シフトされるビット数は、GR r_3 または即値 $count_6$ によって指定される。このシフト・カウントは符号なし数値として解釈される。GR r_3 の値が 63 より大きい場合は、結果はすべてゼロ・ビットになる。

即値形式については、7-31 ページの「デポジット (Deposit)」を参照のこと。

操作：

```
if (PR[qp]) {
    check_target_register(r1);

    count = GR[r3];
    GR[r1] = (count > 63) ? 0: GR[r2] << count;

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

左シフトおよび加算 (Shift Left and Add)

書式: (qp) shladd $r_1 = r_2$, $count_2$, r_3 A2

説明: 第 1 ソース・オペランドが、$count_2$ ビットだけ左にシフトされ、次に、第 2 ソース・オペランドに加算され、その結果が GR r_1 に格納される。第 1 オペランドに対しては 1、2、3、あるいは 4 ビットだけシフトできる。

操作:

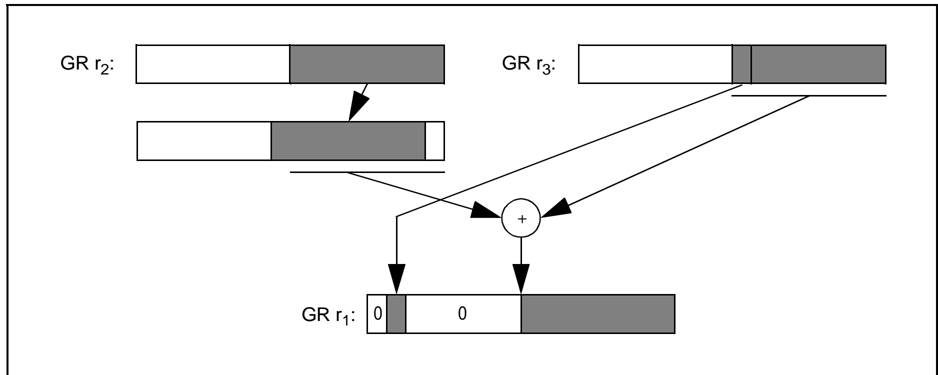
```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = (GR[r2] << count2) + GR[r3];
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

ポインタの左シフトおよび加算 (Shift Left and Add Pointer)

書式 : (qp) shladdp4 $r_1 = r_2, count_2, r_3$ A2

説明 : 第 1 ソース・オペランドが、$count_2$ ビットだけ左にシフトされ、次に、第 2 ソース・オペランドに加算される。結果の上位 32 ビットがゼロにクリアされ、次に、GR r_3 のビット {31:30} が結果のビット {62:61} にコピーされる。この結果が GR r_1 に格納される。第 1 オペランドは、1、2、3、あるいは 4 ビットだけシフトできる。

図 7-41. ポインタの左シフトおよび加算



操作 :

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_res = (GR[r2] << count2) + GR[r3];
    tmp_res = zero_ext(tmp_res{31:0}, 32);
    tmp_res{62:61} = GR[r3]{31:30};
    GR[r1] = tmp_res;
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

右シフト (Shift Right)

書式:

(qp) shr $r_1 = r_3, r_2$	signed_form	I5
(qp) shr.u $r_1 = r_3, r_2$	unsigned_form	I5
(qp) shr $r_1 = r_3, count_6$	pseudo-op of: (qp) extr $r_1 = r_3, count_6, 64–count_6$	
(qp) shr.u $r_1 = r_3, count_6$	pseudo-op of: (qp) extr.u $r_1 = r_3, count_6, 64–count_6$	

説明:

GR r_3 内の値が右にシフトされ、GR r_1 に格納される。signed_form では、空きになったビット位置に GR r_3 のビット 63 が埋められ、unsigned_form では、空きになったビット位置にゼロが埋められる。シフトされるビット数は、GR r_2 の値または即値 $count_6$ によって指定される。シフト・カウントは符号なし数値として解釈される。GR r_2 の値が 63 より大きい場合の結果は、unsigned_form の場合あるいは GR r_3 のビット 63 が 0 の場合はすべて 0 ビットに、signed_form で GR r_3 のビット 63 が 1 の場合にはすべて 1 ビットになる。

.u コンプリータが指定された場合はシフトは符号なし (論理) シフトになり、指定されない場合は符号付き (算術) シフトになる。

即値形式については、7-33 ページの「抽出 (Extract)」を参照のこと。

操作:

```
if (PR[qp]) {
    check_target_register(r1);

    if (signed_form) {
        count = (GR[r2] > 63) ? 63 : GR[r2];
        GR[r1] = shift_right_signed(GR[r3], count);
    } else {
        count = GR[r2];
        GR[r1] = (count > 63) ? 0 : shift_right_unsigned(GR[r3], count);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

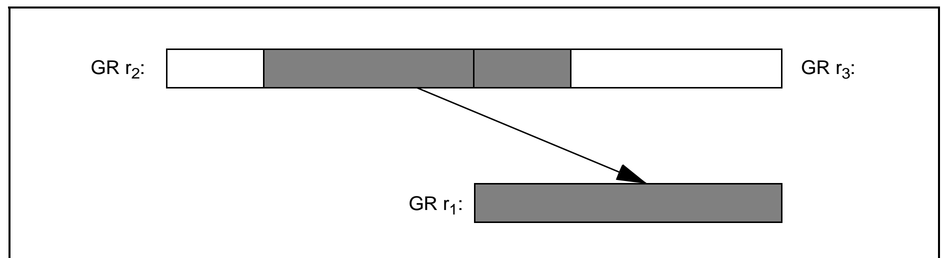
ペア右シフト (Shift Right Pair)

書式: (*qp*) shrp $r_1 = r_2, r_3, count_6$ I10

説明: 2 つのソース・オペランド GR r_2 と GR r_3 が連結されて 128 ビット値が形成され、$count_6$ ビットだけ右にシフトされる。結果の最下位 64 ビットが GR r_1 に格納される。

即値 $count_6$ には 0 〜 63 の範囲内の任意の数値を使用できる。

図 7-42. ペア右シフト



操作:

```
if (PR[qp]) {
    check_target_register(r1);

    temp1 = shift_right_unsigned(GR[r3], count6);
    temp2 = GR[r2] << (64 - count6);
    GR[r1] = zero_ext(temp1, 64 - count6) | temp2;
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

シリアル化 (Serialize)

書式: (*qp*) srlz.i M24

説明: 命令のシリアル化 (`srlz.i`) によって、以下のことが保証される。

- 後続の命令グループのフェッチに影響するプロセッサ・レジスタ・リソースに対する、先行する変更が検出できる。
- 後続のプログラム実行またはデータ・メモリ・アクセスに影響するプロセッサ・レジスタ・リソースに対する、先行する変更が検出できる。
- 先行メモリ同期 (`sync.i`) 操作が、ローカル・プロセッサの命令キャッシュに対して効果を与える。
- `srlz.i` 命令終了後の後続の命令グループのフェッチが再初期化される。

`srlz.i` 命令は、シリアル化される操作がある命令グループより後の命令グループ内になければならない。シリアル化に依存する操作は、`srlz.i` 命令がある命令グループより後の命令グループになければならない。

操作:

```
if (PR[qp]) {
    instruction_serialize();
}
```

ストア (Store)

書式 :

(qp) st*sz*.*sttype*.*sthint* [r_3] = r_2	normal_form, no_base_update_form	M4
(qp) st*sz*.*sttype*.*sthint* [r_3] = r_2, imm_9	normal_form, imm_base_update_form	M5
(qp) st8.spill.*sthint* [r_3] = r_2	spill_form, no_base_update_form	M4
(qp) st8.spill.*sthint* [r_3] = r_2, imm_9	spill_form, imm_base_update_form	M5

説明 :

GR r_2 から最下位から *sz* バイトの値が取り出され、GR r_3 の値によって指定されるアドレスから始まるメモリ位置に書き込まれる。sz コンプリータの値は、7-111 ページの表 7-26 に示してある。*sttype* コンプリータで特殊なストア操作を指定する。それらの操作の説明を表 7-42 に示す。GR r_3 に対応する NaT ビット (normal_form では、GR r_2 に対応する NaT ビット) が 1 である場合は、レジスタ NaT 参照 (Register NaT Consumption) フォルトが発生する。

spill_form では、8 バイト値がストアされ、GR r_2 に対応する NaT ビットが UNAT アプリケーション・レジスタの特定ビットにコピーされる。この命令は、レジスタ /NaT のペアをスピルする場合に使用される。詳細については、4-16 ページの「コントロール・スペキュレーション」を参照のこと。

imm_base_update 形式では、GR r_3 の値が符号付き即値 (imm_9) に加算され、その結果が GR r_3 に戻される。このベース・レジスタの更新はストアの後で行われ、ストア・アドレスにも、ストアされる値 (r_2 と r_1 が同じレジスタを指定している場合) にも影響しない。

表 7-42. ストアのタイプ

sttype コンプリータ	意味	特殊なストア操作
none	通常のストア	
rel	順序付けされたストア	release 語彙付きで順序付けされたストアが実行される。

順序付けされたストアの詳細については、4-30 ページの「メモリ・アクセスの順序」を参照のこと。

物理メモリ・アドレスとアクセス・サイズを使用して ALAT が照会され、すべてのオーバラップ・エントリが無効にされる。

sthint コンプリータの値で、メモリ・アクセスの局所性を指定する。sthint コンプリータの値を表 7-43 に示す。4-26 ページの「メモリ階層の制御と整合性」を参照のこと。

表 7-43. ストアのヒント

sthint コンプリータ	意味
none	時間的局所性、レベル 1
nta	非時間的局所性、全レベル

操作:

```
if (PR[qp]) {
    size = spill_form ? 8 : sz;
    otype = (sttype == 'rel') ? RELEASE : UNORDERED;

    if (imm_base_update_form)
        check_target_register(r3);
    if (GR[r3].nat || (normal_form && GR[r2].nat))
        register_nat_consumption_fault(WRITE);

    paddr = tlb_translate(GR[r3], size, WRITE, PSR.cpl, &mattr,
                                &tmp_unused);
    if (spill_form && GR[r2].nat)
        natd_gr_write(GR[r2], paddr, size, UM.be, mattr, otype, sthint);
    else
        mem_write(GR[r2], paddr, size, UM.be, mattr, otype, sthint);

    if (spill_form) {
        bit_pos = GR[r3]{8:3};
        AR[UNAT]{bit_pos} = GR[r2].nat;
    }

    alat_inval_multiple_entries(paddr, size);

    if (imm_base_update_form) {
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = 0;
    }
}
```

浮動小数点ストア (Floating-Point Store)

書式:

(qp) stf*fsz*.*sthint* [r_3] = f_2	normal_form, no_base_update_form	M9
(qp) stf*fsz*.*sthint* [r_3] = f_2, imm_9	normal_form, imm_base_update_form	M10
(qp) stf8.*sthint* [r_3] = f_2	integer_form, no_base_update_form	M9
(qp) stf8.*sthint* [r_3] = f_2, imm_9	integer_form, imm_base_update_form	M10
(qp) stf.spill.*sthint* [r_3] = f_2	spill_form, no_base_update_form	M9
(qp) stf.spill.*sthint* [r_3] = f_2, imm_9	spill_form, imm_base_update_form	M10

説明:

FR f_2 から *fsz* バイトの値が取り出され、GR r_3 の値によって指定されるアドレスから始まるメモリ位置に書き込まれる。normal_form では、FR f_2 の値がメモリ形式に変換されてからストアされる。integer_form では、FR f_2 の仮数がストアされる。*fsz* コンプリータの値は 7-115 ページの表 7-29 に示してある。normal_form または integer_from では、GR r_3 に対応する NaT ビットが 1 か、あるいは FR f_2 の内容が NaTVal である場合は、レジスタ NaT 参照 (Register NaT Consumption) フォルトが発生する。浮動小数点レジスタ形式からの変換の詳細については、5-1 ページの「データ型および形式」を参照のこと。

spill_form では、FR f_2 から 16 バイトの値が取り出されて、変換されないでストアされる。この命令はレジスタをスピルする場合に使用される。詳細については、4-16 ページの「コントロール・スペキュレーション」を参照のこと。

imm_base_update 形式では、GR r_3 の値が符号付き即値 (imm_9) に加算され、結果が GR *r3* に戻される。このベース・レジスタの更新はストアの後で行われ、ストア・アドレスには影響しない。

物理メモリ・アドレスとアクセス・サイズを使用して ALAT が照会され、すべてのオーバラップ・エントリが無効にされる。

sthint コンプリータの値でメモリ・アクセスの局所性を指定する。*stint* コンプリータの値は、7-181 ページの表 7-43 に示してある。4-26 ページの「メモリ階層の制御と整合性」を参照のこと。

操作:

```
if (PR[qp]) {
    if (imm_base_update_form)
        check_target_register(r3);
    if (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, WRITE);

    if (GR[r3].nat || (!spill_form && (FR[f2] == NATVAL)))
        register_nat_consumption_fault(WRITE);

    size = spill_form ? 16 : (integer_form ? 8 : fsz);

    paddr = tlb_translate(GR[r3], size, WRITE, PSR.cpl, &mattr, &tmp_unused);
    val = fp_fr_to_mem_format(FR[f2], size, integer_form);
    mem_write(val, paddr, size, UM.be, mattr, UNORDERED, sthint);

    alat_inval_multiple_entries(paddr, size);

    if (imm_base_update_form) {
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = 0;
    }
}
```

減算 (Subtract)

書式:

(qp) sub $r_1 = r_2, r_3$	register_form	A1
(qp) sub $r_1 = r_2, r_3$, 1	minus1_form, register_form	A1
(qp) sub $r_1 = imm_8, r_3$	imm8_form	A3

説明: 第 2 ソース・オペランド (および任意設定の定数 1) が第 1 オペランドから減算され、結果が GR r_1 に格納される。レジスタ形式では、第 1 オペランドは GR r_2 であり、即値形式では、第 1 オペランドは符号拡張された imm_8 のエンコーディング・フィールドで与えられる。

minus1_form は、register_form でのみ使用可能である (ただし、即値を調整することにより、即値形式でも同じ結果が得られる)。

操作:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    if (minus1_form)
        GR[r1] = tmp_src - GR[r3] - 1;
    else
        GR[r1] = tmp_src - GR[r3];

    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

ユーザ・マスクの設定 (Set User Mask)

書式: (*qp*) sum imm_{24} M44

説明: imm_{24} オペランドとユーザ・マスク (PSR{5:0}) との論理和が取られ、結果がユーザ・マスクに戻される。

PSR.up は、セキュリティ・パフォーマンス・モニタ・ビット (PSR.sp) がゼロの場合に限りセットできる。そうでない場合は、PSR.up は変更できない。

操作:

```
if (PR[qp]) {
    if (is_reserved_field(PSR_TYPE, PSR_UM, imm24))
        reserved_register_field_fault();

    if (imm24{1})       PSR{1} = 1;
    if (imm24{2} && PSR.sp == 0)          //non-secure perf monitor
                        PSR{2} = 1;
    if (imm24{3})       PSR{3} = 1;
    if (imm24{4})       PSR{4} = 1;
    if (imm24{5})       PSR{5} = 1;
}
```

符号拡張 (Sign Extend)

書式: (*qp*) sxt*xsz* $r_1 = r_3$ I29

説明: GR r_3 の値が、*xsz* によって指定されるビット位置から符号拡張され、結果が GR r_1 に格納される。*xsz* のニーモニック値を表 7-44 に示す。

表 7-44. xsz ニーモニック値

xsz ニーモニック	ビット位置
1	7
2	15
4	31

操作:

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = sign_ext(GR[r3],xsz * 8);
    GR[r1].nat = GR[r3].nat;
}
```

メモリ同期化 (Memory Synchronization)

書式 : (*qp*) sync.i M24

説明 : sync.i 命令によって、ローカル・プロセッサから以前に発行されたキャッシュのフラッシュ (fc) 操作がローカル・データ・メモリの参照において検出可能になった時点で、先行の fc 操作がローカル・プロセッサの命令フェッチ・ストリームからも検出されることが保証される。さらに、fc 操作が事前に起動されていると、それらの操作がリモート・プロセッサ上のデータ・メモリの参照において検出されるようになった時点で、リモート・プロセッサ上の命令メモリ参照においても検出されることが保証される。sync.i は、別のプロセッサから観察されるように、すべてのキャッシュ・フラッシュ操作に対して順序付けされる。sync.i とそれに先行する fc は、別々の命令グループになければならない。語彙上の必要がある場合、sync.i により fc が他のプロセッサ上のデータ・ストリームから検出できるように適切に強制するためには、プラグラマは明示的に順序づけされたデータ参照 (acquire (取得)、release (解放)、fence (フェンス) のいずれかのタイプ) を挿入しなければならない。

sync.i は、ローカル・プロセッサおよびリモート・プロセッサ上の命令キャッシュとデータ・キャッシュとの間の順序付け関係を維持する場合に使用される。命令をシリアル化する操作を使用して、ローカル・プロセッサで sync.iによって起動された同期化操作がプログラム実行中の特定の時点で観察されていたことを確認することができる。

自己修正コード (ローカル・プロセッサ) の例 :

```
     st   [L1] = data  //store into local instruction stream
     fc   L1           //flush stale datum from instruction/data cache
     ;;                //require instruction boundary between fc and sync.i
     sync.i            //ensure local and remote data/inst caches are synchronized
     ;;
     srlz.i            //ensure sync has been observed by the local processor,
     ;;                //ensure subsequent instructions observe modified memory
L1:  target            //instruction modified
```

操作 :

```
if (PR[qp]) {
    instruction_synchronize();
}
```

ビット・テスト (Test Bit)

書式: (*qp*) tbit.*trel*.*ctype* $p_1, p_2 = r_3, pos_6$ I16

説明: 即値 pos_6 によって指定されるビットが GR r_3 から選択される。選択されたビットは、*trel* コンプリータに応じて補数を取られるかそのままで、単一ビットの結果を形成する。この結果は、2 つのプレディケート・レジスタ・デスティネーション p_1 と p_2 に書き込まれる。結果がデスティネーションにどのように書き込まれるかは、*ctype* によって指定される比較タイプによって決まる。Compare 命令と 7-22 ページの表 7-10 を参照のこと。

trel コンプリータの値 .nz および .z は、それぞれ、非ゼロ判定かゼロ判定かを示す。通常タイプおよび unc タイプの比較においては、.z の値だけがハードウェアによって直接サポートされている。つまり、.nz 値は、実際には擬似オペコードである。.nz 値については、アセンブラは単にプレディケート・ターゲット指定子を切り換え、サポートされている関係を利用する。並列タイプについては、両方の関係がハードウェアでサポートされている。

表 7-45. 通常および unc タイプのビット判定での関係

trel	判定関係	擬似オペコード
nz	選択されたビット == 1	z $p_1 \leftrightarrow p_2$
z	選択されたビット == 0	

表 7-46. 並列タイプのビット判定での関係

trel	判定関係
nz	選択されたビット == 1
z	選択されたビット == 0

2 つのプレディケートレジスタ・デスティネーションが同じ (p_1 と p_2 が同一のプレディケート・レジスタを指定) 場合は、修飾プレディケートがセットされているか、比較タイプが unc であれば、この命令は無効操作 (Illegal Operation) フォルトを発生する。

操作：

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    if (trel == 'nz')                                    // 'nz' - test for 1
        tmp_rel = GR[r3]{pos6};
    else                                                 // 'z' - test for 0
        tmp_rel = !GR[r3]{pos6};

    switch (ctype) {
        case 'and':                                      // and-type compare
            if (GR[r3].nat || !tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;
        case 'or':                                       // or-type compare
            if (!GR[r3].nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case 'or.andcm':                                 // or.andcm-type compare
            if (!GR[r3].nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
            }
            break;
        case 'unc':                                      // unc-type compare
        default:                                         // normal compare
            if (GR[r3].nat) {
                PR[p1] = 0;
                PR[p2] = 0;
            } else {
                PR[p1] = tmp_rel;
                PR[p2] = !tmp_rel;
            }
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

Nat テスト (Test NaT)

書式: (*qp*) tnat.*trel*.*ctype* $p_1, p_2 = r_3$ I17

説明: GR r_3 の Nat ビットが、trel コンプリータに応じて補数を取られるかそのままで、単一ビットの結果を形成する。この結果は、プレディケート・レジスタ・デスティネーション p_1 および p_2 に書き込まれる。結果がデスティネーションにどのように書き込まれるかは、*ctype* によって指定される比較タイプによって決まる。Compare 命令と 7-22 ページの表 7-10 を 参照のこと。

trel コンプリータの値 .nz および . z は、それぞれ、、非ゼロ判定かゼロ判定かを示す。通常タイプおよび unc タイプの比較においては、.z の値だけがハードウェアによって直接サポートされている。つまり、.nz 値は、実際には擬似オペコードである。.nz 値については、アセンブラは単にプレディケート・ターゲット指定子を切り換え、サポートされている関係を利用する。並列タイプについては、両方の関係がハードウェアでサポートされる。

表 7-47. 通常および unc タイプの Nat 判定での関係

trel	判定関係	擬似オペコード
nz	選択されたビット == 1	z $p_1 \leftrightarrow p_2$
z	選択されたビット == 0	

表 7-48. 並列タイプの NaT 判定での関係

trel	判定関係
nz	選択されたビット == 1
z	選択されたビット == 0

2 つのプレディケートレジスタ・デスティネーションが同じ (p_1 と p_2 が同一のプレディケート・レジスタを指定) 場合は、修飾プレディケートがセットされているか、比較タイプが unc であれば、この命令は無効操作 (Illegal Operation) フォルトを発生する。

操作：

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    if (trel == ‘nz’)                                       // ‘nz’ - test for 1
        tmp_rel = GR[r3].nat;
    else                                                    // ‘z’ - test for 0
        tmp_rel = !GR[r3].nat;

    switch (ctype) {
        case ‘and’:                                         // and-type compare
            if (!tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;
        case ‘or’:                                          // or-type compare
            if (tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case ‘or.andcm’:                                    // or.andcm-type
compare
            if (tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
            }
            break;
        case ‘unc’:                                         // unc-type compare
        default:                                            // normal compare
            PR[p1] = tmp_rel;
            PR[p2] = !tmp_rel;
            break;
    }
} else {
    if (ctype == ‘unc’) {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

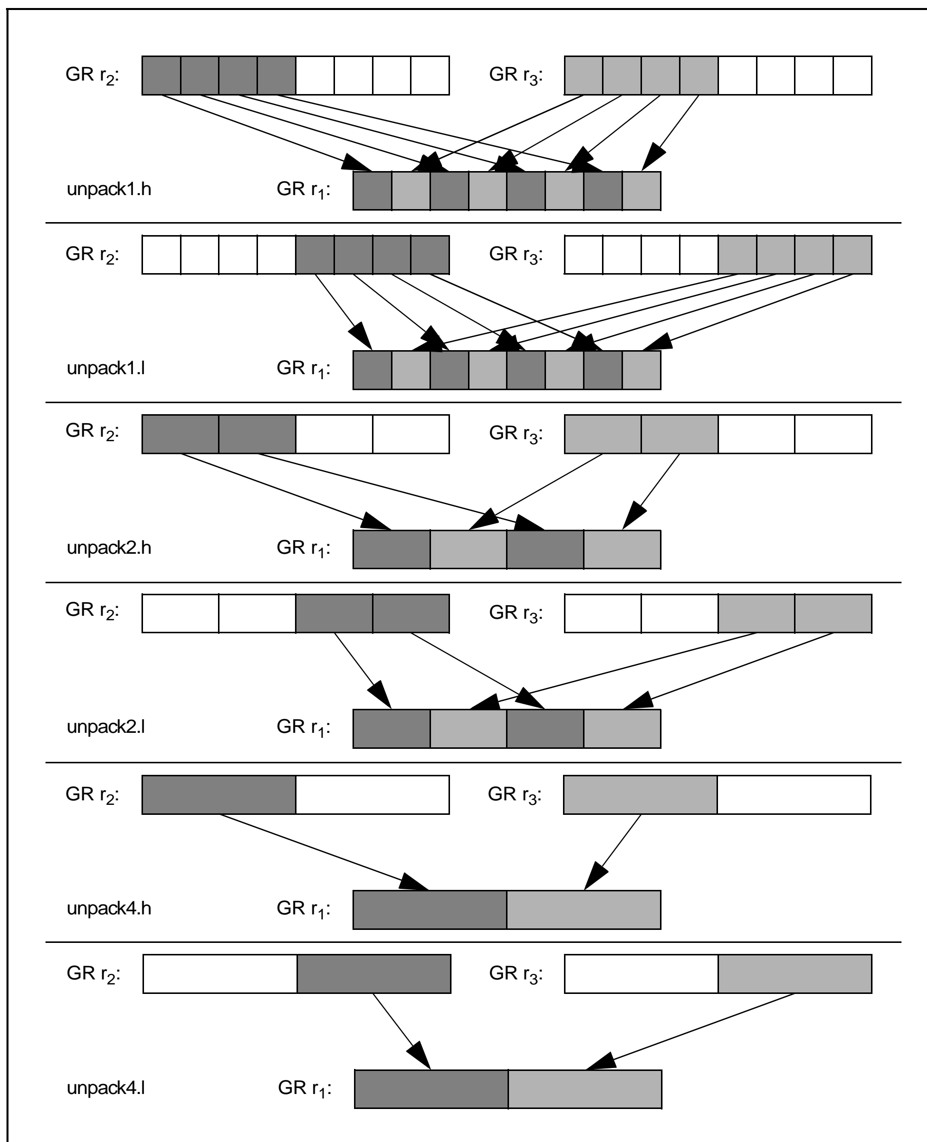
アンパック (Unpack)

書式:

(qp) unpack1.h $r_1 = r_2, r_3$	one_byte_form, high_form	I2
(qp) unpack2.h $r_1 = r_2, r_3$	two_byte_form, high_form	I2
(qp) unpack4.h $r_1 = r_2, r_3$	four_byte_form, high_form	I2
(qp) unpack1.l $r_1 = r_2, r_3$	one_byte_form, low_form	I2
(qp) unpack2.l $r_1 = r_2, r_3$	two_byte_form, low_form	I2
(qp) unpack4.l $r_1 = r_2, r_3$	four_byte_form, low_form	I2

説明: GR r_2 および r_3 の各データ要素がアンパックされ、結果が GR r_1 に格納される。high_form では、各ソース・レジスタの最上位要素が選択され、それに対して low_form では、各ソース・レジスタの最下位要素が選択される。要素はそれぞれのソース・レジスタから交互に選択される。

図 7-43. アンパックの操作



操作:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                    // one-byte elements
        x[0] = GR[r2]{7:0};         y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};        y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};       y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};       y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};       y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};       y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};       y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};       y[7] = GR[r3]{63:56};

        if (high_form)
            GR[r1] = concatenate8(     x[7], y[7], x[6], y[6],
                                       x[5], y[5], x[4], y[4]);
        else
            GR[r1] = concatenate8(     x[3], y[3], x[2], y[2],
                                       x[1], y[1], x[0], y[0]);
    } else if (two_byte_form) {                             // two-byte elements
        x[0] = GR[r2]{15:0};        y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};       y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};       y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};       y[3] = GR[r3]{63:48};

        if (high_form)
            GR[r1] = concatenate4(x[3], y[3], x[2], y[2]);
        else
            GR[r1] = concatenate4(x[1], y[1], x[0], y[0]);
    } else {                                                // four-byte elements
        x[0] = GR[r2]{31:0};        y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};       y[1] = GR[r3]{63:32};

        if (high_form)
            GR[r1] = concatenate2(x[1], y[1]);
        else
            GR[r1] = concatenate2(x[0], y[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

交換 (Exchange)

書式: (*qp*) xchg*sz*.*ldhint* r_1 = [r_3], r_2 M16

説明: *sz* バイトの値が、GR r_3 の値によって指定されるアドレスから始まるメモリ位置から読み込まれる。GR r_2 の値の最下位 *sz* ビットが、GR r_3 の値によって指定されるアドレスから始まるメモリ位置に書き込まれる。次に、メモリから読み込まれた値がゼロ拡張され、GR r_1 に格納され、GR r_1 に対応する NaT ビットがクリアされる。*sz* コンプリータの値を表 7-49 に示す。

IGR r_3 の値によって指定されるアドレスが、メモリでアクセスされるサイズに自然にアライメントされていない場合は、ユーザ・マスク (User Mask) アライメント・チェック・ビット UM.ac (プロセッサ・ステータス・レジスタの PSR.ac) の状態に関わらず、非アライメント・データ参照 (Unaligned Data Reference) フォルトが発生する。

参照先ページに対する読み取りおよび書き込み両方のアクセス特権が必要である。

表 7-49. メモリ交換のサイズ

sz コンプリータ	アクセスされるバイト数
1	1 バイト
2	2 バイト
4	4 バイト
8	8 バイト

このメモリ交換操作は、aquire (取得) の語彙を使用して実行される。つまり、このメモリに対する読み取り / 書き込みは、以降のすべてのデータ・メモリ・アクセスの前に見えるようになる。

メモリに対する読み取り / 書き込みはアトミックな操作であることが保証されている。。

ldhint コンプリータの値でメモリ・アクセスの局所性を指定する。*ldhint* コンプリータの値は 7-113 ページの表 7-28 に示してある。局所性のヒントはプログラムの機能には影響せず、したがってプログラム・コードによって無視することもできる。詳細については、4-26 ページの「メモリ階層の制御と整合性」を参照のこと。

操作:

```
if (PR[qp]) {
    check_target_register(r1, SEMAPHORE);

    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(SEMAPHORE);

    paddr = tlb_translate(GR[r3], sz, SEMAPHORE, PSR.cpl, &mattr, &tmp_unused);

    if (!ma_supports_semaphores(mattr))
        unsupported_data_reference_fault(SEMAPHORE, GR[r3]);

    val = mem_xchg(GR[r2], paddr, sz, UM.be, mattr, ACQUIRE, ldhint);
```

```
    alat_inval_multiple_entries(paddr, sz);

    GR[r1] = zero_ext(val, sz * 8);
    GR[r1].nat = 0;
}
```

固定小数点積和 (Fixed-Point Multiply Add)

書式：

(qp) xma.l $f_1 = f_3, f_4, f_2$	low_form	F2
(qp) xma.lu $f_1 = f_3, f_4, f_2$	pseudo-op of: (qp) xma.l $f_1 = f_3, f_4, f_2$	
(qp) xma.h $f_1 = f_3, f_4, f_2$	high_form	F2
(qp) xma.hu $f_1 = f_3, f_4, f_2$	high_unsigned_form	F2

説明：

2 つのソース・オペランド (FR f_3 および FR f_4) が符号付きまたは符号なしの整数として扱われ、両オペランド間の乗算が行われる。第 3 のソース・オペランド (FR f_2) がゼロ拡張され、得られた積に加算される。結果の上位または下位の 64 ビットが選択され、FR f_1 に格納される。

high_unsigned_form では、FR f_3 と FR f_4 の両仮数フィールドが符号なし整数として扱われ、乗算されて全 128 ビットの符号なし結果が生成される。FR f_2 の仮数フィールドがゼロ拡張され、積に加算される。結果の最上位 64 ビットが FR f_1 の仮数フィールドに格納される。

high_form では、FR f_3 と FR f_4 の両仮数フィールドが符号付き整数として扱われ、乗算されて全 128 ビットの符号付き結果が生成される。FR f_2 の仮数フィールドがゼロ拡張され、積に加算される。結果の最上位 64 ビットが FR f_1 の仮数フィールドに格納される。

low_form では、FR f_3 と FR f_4 の両仮数フィールドが符号付き整数として扱われ、乗算されて全 128 ビットの符号付き結果が生成される。FR f_2 の仮数フィールドがゼロ拡張され、積に加算される。結果の最下位 64 ビットが FR f_1 の仮数フィールドに格納される。

すべての形式で、FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

注：　オペランドとしての f_1 は整数の 1 でなく、レジスタ・ファイル形式の値 1.0 である。

すべての形式で、FR f_3、FR f_4、FR f_2 のどれか 1 つでも NaTVal である場合は、FR f_1 は計算結果ではなく NatVal に設定される。

操作：

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) || fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
    } else {
        if (low_form || high_form)
            tmp_res_128 =
                fp_I64_x_I64_to_I128(FR[f3].significand, FR[f4].significand);
        else // high_unsigned_form
            tmp_res_128 =
                fp_U64_x_U64_to_U128(FR[f3].significand, FR[f4].significand);

        tmp_res_128 =
            fp_U128_add(tmp_res_128, fp_U64_to_U128(FR[f2].significand));

        if (high_form || high_unsigned_form)
            FR[f1].significand = tmp_res_128.hi;
```

```
        else // low_form
            FR[f1].significand = tmp_res_128.lo;

        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

固定小数点乗算 (Fixed-Point Multiply)

書式:

(qp) xmpy.l $f_1 = f_3, f_4$	pseudo-op of: (qp) xma.l $f_1 = f_3, f_4$, f0
(qp) xmpy.lu $f_1 = f_3, f_4$	pseudo-op of: (qp) xma.l $f_1 = f_3, f_4$, f0
(qp) xmpy.h $f_1 = f_3, f_4$	pseudo-op of: (qp) xma.h $f_1 = f_3, f_4$, f0
(qp) xmpy.hu $f_1 = f_3, f_4$	pseudo-op of: (qp) xma.hu $f_1 = f_3, f_4$, f0

説明: 2 つのソース・オペランド (FR f_3 と FR f_3) が符号付きまたは符号なしの整数として扱われ、両オペランド間の乗算が行われる。結果の上位または下位 64 ビットが選択され、FR f_1 に格納される。

high_unsigned_form では、FR f_3 と FR f_4 の両仮数フィールドが符号なし整数として扱われ、乗算されて全 128 ビットの符号なし結果が生成される。結果の最上位 64 ビットが FR f_1 の仮数フィールドに格納される。

high_form では、FR f_3 と FR f_4 の両仮数フィールドが符号付き整数として扱われ、乗算されて全 128 ビットの符号付き結果が生成される。結果の最上位 64 ビットが FR f_1 の仮数フィールドに格納される。

low_form では、FR f_3 と FR f_4 の両仮数フィールドが符号付き整数として扱われ、乗算されて全 128 ビットの符号付き結果が生成される。結果の最下位 64 ビットが FR f_1 の仮数フィールドに格納される。

すべての形式で、FR f_1 の指数フィールドは 2.0^{63} (0x1003E) のバイアス付き指数に設定され、FR f_1 の符号フィールドは正に対応する 0 に設定される。

> ***注：*** オペランドとしての f_1 は整数の 1 でなく、レジスタ・ファイル形式の値 1.0 である。

操作: 7-197 ページの「固定小数点積和 (Fixed-Point Multiply Add)」を参照のこと。

排他的論理和 (Exclusive Or)

書式:

(qp) xor $r_1 = r_2, r_3$	register_form	A1
(qp) xor $r_1 = imm_8, r_3$	imm8_form	A3

説明: 2 つのソース・オペランド間の排他的論理和 (XOR) が取られ、結果が GR r_1 に格納される。register_form では、第 1 オペランドは GR r_2 であり、imm8_form では、第 1 オペランドは imm_8 のエンコーディング・フィールドで与えられる。

操作:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src ^ GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

ゼロ拡張 (Zero Extend)

書式 :　(*qp*) zxt*xsz* $r_1 = r_3$　I29

説明 :　GR r_3 の値が、*xsz* によって指定されるビット位置の上までゼロ拡張され、結果が GR r_1 に格納される。*xsz* のニーモニック値は 7-186 ページの表 7-44 に示してある。

操作 :

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = zero_ext(GR[r3],xsz * 8);
    GR[r1].nat = GR[r3].nat;
}
```

第 II 部： IA-64 最適化ガイド

intel®

IA-64 最適化ガイドについて 8

本書のこの第 2 部では、IA-64 命令セットに関連する最適化手法を詳細に説明する。第 2 部は、IA-64 アプリケーション・アーキテクチャ機能、およびアプリケーションのパフォーマンスに有益な最適化手法について、さらに理解を深めることを意図している。インテルおよび他のベンダーでは、これらの手法を利用するコンパイラの開発を進めている。アプリケーション開発担当の方々には、この第 2 部を IA-64 アセンブリ言語プログラミングのガイドとして利用することはお勧めできない。

注： **手法の実際的な使い方を示すため、本ガイドには、特定の IA-64 プロセッサ向けではなく、仮定のターゲットを対象とするコード例が記載してある。それらのコード例について、ALU 操作には 1 サイクルを要し、ロードには第 1 レベル・キャッシュから戻るのに 2 サイクル要するものとし、かつ、ロード / ストア実行ユニットが 2 つと ALU が 4 つ存在するものとしている。その他のレイテンシおよび実行ユニットの詳細は、必要に応じて文中で説明されている。本ガイドでは、このモデルを「汎用」ターゲットと呼ぶ。**

8.1 IA-64 最適化ガイドの概要

第 9 章「IA-64 プログラミングの概要」。IA-64 アプリケーション・プログラミング環境の概要を示す。

第 10 章「メモリ参照」。コントロールおよびデータのスペキュレーションに関連する機能と最適化について説明する。

第 11 章「プレディケーション、コントロール・フロー、命令ストリーム」。プレディケーション、コントロール・フロー、分岐ヒントに関連する機能および最適化について説明する。

第 12 章「ソフトウェアによるパイプライン化とループのサポート」。ループに対してソフトウェアによるパイプライン処理を使って最適化する方法について詳しく説明する。

第 13 章「浮動小数点アプリケーション」。浮動小数点アプリケーションの現時点でのパフォーマンス上の制約と、これらの制約に関わる IA-64 の機能について説明する。

intel®

IA-64 プログラミングの概要 9

9.1 概要

IA-64 命令セットは、コンパイラからプロセッサに情報を伝達して、命令レイテンシ、発行範囲、機能ユニットの割り当てなどのリソース特性を管理させることができるよう設計されている。このようなリソースは静的にスケジュールできるが、IA-64 では、正しく動作させるために、コードは特定のマイクロアーキテクチャ向けに書く必要はない。

IA-64 は、下記の目的を果たすよう設計された新機能を備えた完全な命令セットを取り入れている。

- 命令レベルの並列性 (ILP) の強化
- メモリ・レイテンシ管理の改善
- 分岐処理および分岐リソース管理の改善
- プロシージャ・コールのオーバヘッドの削減

IA-64 では、さらに、高い浮動小数点パフォーマンスが可能であり、マルチメディア・アプリケーション向けの直接サポートが可能である。

IA-64 命令のシンタックスと語彙の全般については、第 I 部 :「IA-64 アプリケーション・アーキテクチャ・ガイド」で説明している。本章は、アプリケーション・レベルの IA-64 プログラミングの高水準の概説であり、読者には IA-64 アプリケーション・アーキテクチャに関するある程度の知識の他に、これまでにアセンブリ言語プログラミングの経験もあるものと想定している。最適化については、本ガイドの他の章で詳述している。

9.2 レジスタ

IA-64 アーキテクチャは、128 の汎用レジスタ、128 の浮動小数点レジスタ、64 のプレディケート・レジスタ、および最大 128 個の特殊目的レジスタを定義している。IA-64 のアーキテクチャを構成するこれら多数のレジスタによって、中間値データをメモリに頻繁に退避したり復元する必要なく、多数の計算を行うことができる。

整数計算およびマルチメディア計算のためのデータ保持用として、128 の 64 ビット**汎用レジスタ** (`r0` ～ `r127`) がある。これら 128 のレジスタには、それぞれ、レジスタにストアされている値が有効であるかどうかを示すための NaT (ノット・ア・シング) ビットが 1 つずつ付加されている。IA-64 のスペキュレーティブ命令を実行すると、NaT ビットがセットされることがある。レジスタ `r0` は読み取り専用であり、値はゼロである。`r0` に書き込もうとすると、フォルトが発生する。

浮動小数点計算用として、128 の 82 ビット**浮動小数点レジスタ** (`f0` ～ `f127`) がある。最初の 2 つのレジスタ `f0` と `f1` は読み取り専用であり、読み取り値はそれぞれ +0.0 と +1.0 である。`f0` または `f1` に書き込みを行うと、フォルトが発生する。

命令の条件付き実行や条件付き分岐を制御するための 64 の 1 ビットからなる**プレディケート・レジスタ** (`p0` ～ `p63`) がある。最初のレジスタ `p0` は読み取り専用であり、常に真 (1) が読み込まれる。`p0` に書き込んでも、結果は無視される。

間接分岐の分岐先アドレスの指定用として、8 つの 64 ビット**分岐レジスタ** (`b0` ～ `b7`) がある。

各種機能をサポートする**アプリケーション・レジスタ** (`ar0` ～ `ar127`) 用のスペースが 128 個分まである。これらのレジスタ・スロットの多くは、将来用に予約されている。一部のアプリケーション・レジスタにはアセンブラ用の別名が付いている。例えば、`ar66` はエピローグ・カウンタであり、`ar.ec` と呼ばれる。

命令ポインタは、現在実行中の命令バンドルをポイントする 64 ビット・レジスタである。

9.3 IA-64 命令の使用法

IA-64 命令は、3 つの命令からなる 128 ビットのバンドルと呼ばれるグループにまとめられる。各命令はバンドルの第 1、第 2、および第 3 のシラブル (節) を占める。命令の書式、並列表現、およびバンドル指定について、以下で説明する。

9.3.1 書式

基本的な A-64 命令のシンタックスは次の通りである。

[*qp*] *mnemonic*[.*comp*] *dest*=*srcs*

ここで、

qp 修飾プレディケート・レジスタを指定する。修飾プレディケートの値は、命令の結果がハードウェアにコミットされるか、捨てられるかを決定する。プレディケート・レジスタの値が真 (1) のときは、命令が実行され、その結果がコミットされ、例外が発生した場合には通常通りに処理される。値が偽 (0) のときは、結果はコミットされず、例外は発生しない。大部分の IA-64 命令は修飾プレディケートを使用することができる。

mnemonic IA-64 命令を一意に識別する名前を指定する。

comp 1 つ以上の命令コンプリータを指定する。コンプリータは、基本命令ニーモニックに対する任意指定のバリエーションを示す。コンプリータはニーモニックの次にあり、両者の間はピリオドで区切られる。

dest デスティネーション・オペランドを表す。デスティネーション・オペランドは一般的に命令によって生成される結果値である。

srcs ソース・オペランドを表す。大部分の IA-64 命令は最低 2 つの入力ソース・オペランドをもつ。

9.3.2 並列表現

IA-64 では、命令グループと呼ばれる命令のまとまりを、コンパイラまたはアセンブリ・ライタが明示的に指示する必要がある。この命令グループには、レジスタにリード・アフター・ライト (RAW) およびライト・アフター・ライト (WAW) の依存関係がない。命令グループは、アセンブリ・ソース・コードではストップによって区切られる。命令グループには RAW および WAW のレジスタ依存関係がないので、命令間のレジスタ依存関係の有無をハードウェアでチェックする必要はない。下の 2 例は、ストップ (二重セミコロンで示されている) によって区切られた 2 つの命令グループを示している。

```
ld8 r1=[r5] ;; // First group
add r3=r1,r4   // Second group
```

レジスタ・フローに複数の依存関係がある複雑な例を下に示す。

```
ld8 r1=[r5]    // First group
sub r6=r8,r9 ;;// First group
add r3=r1,r4   // Second group
st8 [r6]=r12   // Second group
```

特定の IA-64 プロセッサでは、同一命令グループ内のすべての命令を発行するためのリソースが不足することがあるので、単一の命令グループ内の命令のすべてが必ずしも並列に発行されるとは限らない。

9.3.3 バンドルとテンプレート

アセンブリ・コードでは、128 ビットの各バンドルが括弧で囲まれ、テンプレート指定と 3 つの命令からなっている。したがって、ストップは、任意のバンドルの終わりに指定するか、あるいは、2 つの特殊テンプレート・タイプ (暗黙的にバンドルの途中にストップが入れられる) のいずれかを使用してバンドルの途中に指定することもできる。

バンドル内の各命令は長さが 41 ビットである。他にテンプレート・タイプの指定に 5 ビットが使用される。バンドルのテンプレートによって、IA-64 プロセッサは単純な命令デコード操作により命令をディスパッチすることができ、ストップによって並列処理を明示的に指定することが可能になる。

IA-64 シラブルのタイプは 5 つ (M、I、F、B、および L)、IA-64 命令のタイプは 6 つ (M、L、A、F、B、および L)、基本テンプレートのタイプは 12 (MII、MI_I、MLX、MMI、M_MI、MFI、MMF、MIB、MBB、BBB、MMB、および MFG) ある。各基本テンプレート・タイプには 2 つのバージョンがあり、1 つは 3 番目のシラブルの後にストップがあるもので、もう 1 つはそのストップがないものである。命令は、テンプレート指定に基づいてそれぞれの命令タイプに対応するシラブルに入れなければならない。ただし、A タイプの命令については別で、それらの命令は I、M いずれのシラブルに入れてもよい。例えば、.MII というテンプレート指定は、同一バンドル内の 3 つの命令のうち、最初の命令はメモリ (M) または A タイプの命令であり、次の 2 つの命令は ALU 整数 (I) または A タイプの命令であることを意味する。

```
{ .mii
    ld4  r28=[r8]  //  Load a 4-byte value
    add  r9=2,r1   //  2+r1 and put in r9
    add  r30=1,r1  //  1+r1 and put in r30
}
```

読みやすいように、本書内の大部分のコード例にはテンプレートも中括弧も示してない。

注: 命令グループは任意の数のバンドルにまたがって拡張させることができるので、バンドルの境界と命令グループの境界との間に直接の関係はない。命令グループは、アセンブリ・コード内にストップが設定されているところで開始し、終了する。さらに、分岐が発生したり、ストップが現れた時点で動的に開始し、終了する。

9.4 メモリ・アクセスとスペキュレーション

A-64 では、メモリ・アクセスは、レジスタのロード命令およびストア命令と、特殊なセマフォ命令を介してのみ可能である。IA-64 は、さらに、プログラマ・コントロール下のスペキュレーションを介して、メモリ・レイテンシを隠すための広範囲なサポート機能も提供している。

9.4.1 機能

データと命令は 64 ビット・アドレスで参照される。命令は、リトル・エンディアン・バイト順に、つまり同一メモリ位置の最下位アドレス・バイト位置に最下位バイトが現れる順序でメモリにストアされる。データについては、ビッグ・エンディアンおよびリトル・エンディアンの両方のバイト順のモードがサポートされており、これらのモードはユーザ・マスク・レジスタによってコントロールできる。

整数ロードでは常に各レジスタの全 64 ビットが書き込まれるので、1、2、および 4 バイトの整数ロードはゼロ拡張される。整数ストアでは、指定に応じて、レジスタの 1、2、4、または 8 バイトをメモリに書き込む。

9.4.2 スペキュレーション

コントロール・スペキュレーションでは、プログラマがデータを分割したり、あるいは通常はコードの移動が制限される依存関係をコントロールすることができる。スペキュレーションは 2 種類あり、それぞれコントロール・スペキュレーションとデータ・スペキュレーションと呼ばれている。本項では、IA-64 のスペキュレーションの要約を示す。スペキュレーションによる命令の動作とアプリケーションの詳細については、第 10 章「メモリ参照」を参照のこと。

9.4.3 コントロール・スペキュレーション

コントロール・スペキュレーションでは、ロードやそれに依存するコードを分岐より前に安全に移動させることができる。これに対するサポートは、整数レジスタに関連付けられている特殊な NaT ビットと浮動小数点レジスタの特殊な NaTVal 値によってイネーブルにされる。スペキュレーティブ・ロードによって例外が発生するときは、その例外は即時には発生しない。その代わりに、デスティネーション・レジスタの NaT ビットがセットされる (または浮動小数点レジスタに NaTVal が書かれる)。以降の、NaT ビットがセットされているレジスタを使用するスペキュレーションによる命令では、スペキュレーションによらない命令によって据え置き例外の有無をチェックするか、据え置き例外を発生させるまで、その設定を伝播させてゆく。

例えば、代表的な RISC アーキテクチャ用のコンパイラにおいては、他の情報がない場合は、下に示すシーケンスの分岐より前にロードを安全に移動させることはできない。

```
(p1) br.cond.dptk L1        // Cycle 0
     ld8 r3=[r5] ;;         // Cycle 1
     shr r7=r3,r87          // Cycle 3
```

ロードのレイテンシが 2 サイクルであるとすれば、右シフト (shr) 命令は 1 サイクル分ストールする。ところが、スペキュレーティブ・ロード命令と IA-64 が提供するチェック命令を使用して、上のコードを下に示すように書き換えれば、2 サイクル節減できる。

```
     ld8.s r3=[r5]          // Earlier cycle
     // Other instructions

(p1) br.cond.dptk L1 ;;     // Cycle 0
     chk.s r3,recovery      // Cycle 1
     shr   r7=r3,r87        // Cycle 1
```

このコードは、アクセスされた時点で `r5` がロード可能であり、かつ `ld8.s` から `chk.s` までの間にレイテンシを埋めることができるだけの十分な数の命令があるものと想定している。

9.4.4 データ・スペキュレーション

データ・スペキュレーションでは、競合し合う可能性のあるメモリ参照より前にロードを移動させせることができる。アドバンスド・ロードではデータ・スペキュレーティブ・ロードを排他的に参照する。この IA-64 アセンブリ・シーケンスでロードとストアの順序を検討してみる。

```
st8 [r55]=r45   // Cycle 0
ld8 r3=[r5] ;;  // Cycle 0
shr r7=r3,r87   // Cycle 2
```

IA-64 では、ロードとストアがオーバラップしているメモリ位置を参照するかどうかがわからない場合でも、プログラマはストアより前にロードを移動させるこができる。これは、特殊なアドバンスド・ロード命令とチェック命令を使用することにより実現できる。

```
ld8.a r3=[r5]   // Advanced load
// Other instructions

st8 [r55]=r45   // Cycle 0
ld8.c r3=[r5]   // Cycle 0 - check
shr r7=r3,r87   // Cycle 0
```

注： このスケジュールでの shr 命令は、アドバンスド・ロードと介在するストアとの間に競合がないとすれば、サイクル 0 で発行できる。競合がある場合は、チェック・ロード命令 (`ld8.c`) が競合を検出し、ロードを再発行することになる。

9.5 プレディケーション

プレディケーションとは、修飾プレディケートに基づいて命令を条件付きで実行することである。修飾プレディケートとは、命令によって計算された結果をコミットするかどうかをプロセッサに決定させる値があるプレディケート・レジスタのことである。

プレディケート・レジスタの値は、比較 (cmp) やビット判定 (`tbit`) などの命令の結果によって設定される。特定の命令に関連付けられている修飾プレディケートの値が真 (1) のときは、プロセッサはその命令を実行し、命令の結果がコミットされる。値が偽 (0) のときは、プロセッサは結果を捨て、例外を発生しない。次の C コードで検討してみる。

```
if (a) {
    b = c + d;
}
if (e) {
    h = i + j;
}
```

このコードは、IA-64 では、修飾プレディケートを使用して分岐が削除することができる。下に示す IA-64 の疑似コードは、分岐なしで C 表現を実現している。

```
    cmp.ne  p1,p2=a,r0      // p1 <- a != 0
    cmp.ne  p3,p4=e,r0 ;;   // p3 <- e != 0
(p1)add     b=c,d           // If a != 0 then add
(p3)sub     h=i,j           // If e != 0 then sub
```

プレディケーションの詳細説明については、第 11 章「プレディケーション、コントロール・フロー、命令ストリーム」を参照のこと。プレディケートを使用している命令がそれらの修飾プレディケートにかかわりなくアーキテクチャ上のリソースを読み込んだり、書き込んだりする特殊なケースが少数存在する。

9.6 IA-64 によるプロシージャ・コールのサポート

コールに関する規則では、通常、呼び出し元と呼ばれた側であらかじめレジスタをセーブしておく必要があるが、プロシージャのコールおよびリターン時に大きなオーバヘッドをもたらす可能性がある。この問題に対処するために、IA-64 汎用レジスタの一部が、物理レジスタ・プールから割り当てられた論理的に無限大のスタック・フレーム・セットとして編成される。

9.6.1 スタックされたレジスタ

レジスタ r0 ～ r31 は、グローバル・レジスタまたはスタティック・レジスタと呼ばれ、スタックされたレジスタは含まれない。スタックされたレジスタには、r32 からユーザ設定可能な最高 r127 までの番号が付けられる。

コール先プロシージャは、`alloc` 命令を使用してそのプロシージャ用の新しいスタック・フレームのサイズを指定する。コール先プロシージャは、この命令を使用して、1 フレームにつき最高 96 のレジスタを入力、出力、およびローカルに共用させるよう割り当てることができる。コールが行われると、コール元プロシージャの出力レジスタがコール先プロシージャの入力レジスタとオーバラップされ、したがって、レジスタのコピーもスピルも必要なくパラメータを受け渡しすることができる。

物理レジスタは、スタックされたレジスタが、プロシージャ内で常に `r32` を最初のレジスタとして参照されるようにハードウェアがリネームする。

9.6.2 レジスタ・スタック・エンジン

レジスタ・スタックの管理は、レジスタ・スタック・エンジン (RSE) と呼ばれるハードウェア・メカニズムによって処理される。RSE は、明示的なプログラムの介入なしに、物理レジスタの内容を汎用レジスタ・ファイルとメモリとの間で移動する。それによって、コンパイラからは無限の物理レジスタ・スタックのように見えるプログラミング・モデルが可能になる。ただし、RSE によるレジスタのセーブとリストアはコストが高く付くため、コンパイラはレジスタの使用を最小に抑えるべきである。

9.7 分岐とヒント

分岐はプログラムのパフォーマンスに対して大きな影響をもつので、以下に挙げるように、IA-64 はプログラムのパフォーマンスを向上させるための機能を備えている。

- プレディケーションの使用によるコード内の分岐数の減少。これにより、制御フローの変更が少なくなるので命令のフェッチが改善され、分岐の数が少なくなるので分岐の予測誤りの回数が少なくなり、予測用リソースの競合が小さくなるので分岐予測のヒット率が大きくなる。
- ソフトウェアでの分岐ヒントの提供による、予測およびプリフェッチ用リソース・ハードウェアの利用状況の改善。
- ループに対するソフトウェアによるパイプライン化およびカウント指定ループの終了予測の明示的サポートの提供。

9.7.1 分岐命令

IA-64 における分岐の表現形式は、他のマイクロプロセッサの場合と非常によく似ている。大きな相違は、分岐トリガの制御が、分岐命令でエンコーディングされる条件ではなくプレディケートによって行われる点である。IA-64 では、さらに、分岐予測ストラテジやプリフェッチ、さらにソフトウェアによるパイプライン化に関連するループ、出口、および分岐などの特定の分岐タイプを制御するために、豊富なヒント・セットも備えている。間接分岐の分岐先は、分岐命令に先だって分岐レジスタにロードされる。

9.7.2 ループおよびソフトウェアによるパイプライン化

コンパイラは、アンロールによってループのパフォーマンスを改善する場合がある。ただし、アンロールは、以下の理由から、すべてのループに対して効果があるわけではない。

- アンロールは、利用可能な並列処理を完全には活用しきれない。
- アンロールは、静的に定義されたループ反復回数向けに最適に調整されたものである。
- アンロールによってコード・サイズが大きくなる可能性がある。

これらの制約を克服しながらループのアンロールの利点を維持するため、IA-64 ではアーキテクチャによるソフトウェアによるパイプライン化をサポートしている。ソフトウェアによるパイプライン化により、コンパイラはループをアンロールする必要なく、複数回のループの繰り返し実行をインタリーブさせることができる。IA-64 におけるパイプライン化は下記のものを使用して実現される。

- ループ分岐命令
- `LC` および `EC` アプリケーション・レジスタ
- ローテートするレジスタおよびループ・ステージのプレディケート
- 特殊予測メカニズムを重要な分岐に割り当てるための分岐ヒント

ソフトウェアによってパイプライン化された while ループおよびカウント指定ループの他に、IA-64 は `br.cloop` 命令を使用する単純なカウント指定ループを特別にサポートしている。`cloop` 分岐命令は、修飾プレディケートではなく、64 ビットのループ・カウント (`LC`) アプリケーション・レジスタを使用して分岐終了条件を判定する。

ソフトウェアによるパイプライン化に対する IA-64 のサポートの全般的説明については、第 12 章「ソフトウェアによるパイプライン化とループのサポート」を参照のこと。

9.7.3 レジスタのローテート

レジスタのローテートにより、プレディケーションを使ってのソフトウェアによるパイプライン処理を容易に実現することができる。ローテート・レジスタは、特殊なループ分岐のいずれか 1 つが実行されるたびに、1 レジスタずつローテートされる。したがって、1 回のローテート操作の後には、レジスタ X の内容はレジスタ X + 1 に移り、最大番号のローテート・レジスタの値は `r32` に移っている。汎用レジスタのローテート領域のサイズとしては 8 の倍数が可能であり、`alloc` 命令のフィールドで指定される。プレディケート・レジスタおよび浮動小数点レジスタもローテートできるが、ローテート・レジスタの数はプログラミングすることはできない。ローテートされるのは、プレディケート・レジスタ `p16` ～ `p63` と浮動小数点レジスタ `f32` ～ `f127` である。

9.8 要約

IA-64 は、下記の機能を提供することで、従来のマイクロアーキテクチャでのパフォーマンス障害の影響を低減している。

- 多数のレジスタと命令グループ / バンドルに対するソフトウェア・スケジューリングによる ILP の改善
- プレディケーションによる分岐処理の改善
- レジスタ・スタック・メカニズムによるプロシージャ・コール時のオーバヘッドの軽減
- ループに対するソフトウェアによるパイプライン処理をハードウェアがサポートすることによる、ストリームライン化されたループ処理
- スペキュレーションによるメモリ・レイテンシ隠蔽のサポート

intel®

メモリ参照 10

10.1 概要

メモリ・レイテンシは、整数アプリケーションのパフォーマンスを決定する重要な要因である。メモリ・レイテンシの影響を抑えるために、IA-64 はソフトウェアによるパイプライン化、ラージ・レジスタ・ファイル、およびコンパイラ制御のスペキュレーションをサポートしている。本章では、コンパイラ制御のスペキュレーションに関連する機能と最適化について説明する。ソフトウェアによるパイプライン処理の使用方法に関する詳しい説明については、第 12 章「ソフトウェアによるパイプライン化とループのサポート」を参照のこと。

本章の前半では、IA-64 の非スペキュレーティブなロードとストア、およびデータ依存に関する一般的な概念と用語について説明する。その後、スペキュレーションの概念を紹介し、IA-64 でのスペキュレーションの使われ方に関する説明と例を示す。本章の後半では、メモリ・アクセスと命令スケジューリングに関するいくつかの重要な最適化について説明する。

10.2 非スペキュレーティブなメモリ参照

IA-64 は非スペキュレーティブなロードおよびストアと、明示的なメモリ・ヒント命令をサポートしている。

10.2.1 メモリへのストア

IA-64 の整数ストア命令では 1、2、4、または 8 バイトを、また浮動小数点ストア命令では 4、8、または 10 バイトを書き込むことができる。たとえば、`st4` 命令は、レジスタの最初の 4 バイトをメモリに書き込む。

IA-64 はデフォルトではリトル・エンディアン・メモリのバイト順を使用しているが、ソフトウェアによってユーザ・マスク (UM) のビッグ・エンディアン (be) ビットをセットすることでバイト順を変更できる。

10.2.2 メモリからのロード

IA-64 の整数ロード命令は、発行されたロード命令のタイプに応じて、1、2、4、または 8 バイトをメモリから読み込むことができる。1、2、または 4 バイトのデータをロードすると、ターゲット・レジスタに書き込まれる前に 64 ビットにゼロ拡張される。

ロード命令はさまざまなデータ型に対して用意されているが、IA-64 の基本的なデータ型はクワドワード (8 バイト) である。少数の例外を除き、すべての整数演算はクワドワード・データに対して行われる。これは、符号付き整数、および 32 ビット・アドレスまたは 64 ビット未満の任意のアドレスを扱うときに特に重要となる。

10.2.3 データ・プリフェッチ・ヒント

`lfetch` 命令は、メモリ階層の異なるレベル間でラインを移動するように要求する。IA-64 のすべてのヒント命令と同じように、`lfetch` はプログラムの正確さには影響を与えず、IA-64 のマイクロアーキテクチャでこれを無視するように選択することもできる。

10.3 命令の依存関係

データ依存とコントロール依存は、最適化と命令スケジューリングの基本的な重要要因である。このような依存関係によって、他の命令に依存している命令の配置が制約されるので、コンパイラは、より短いクリティカル・パスとより優れたリソース利用効率を実現できる順序で命令をスケジュールすることができなくなる場合がある。

一般に、メモリ参照は、解消不可能なコントロール依存とデータ依存の主な原因となる。データ依存を無視すると誤った答えが得られ、コントロール依存を無視すると本来ならば発生するはずのないフォルトが発生するからである。本節では以下について説明する。

- メモリ参照の依存関係に関する基本知識
- 依存関係によって、従来のアーキテクチャでのコードのスケジューリングにかかる制約

10.4 節では、コンパイラによって除去できる依存の数を増やすための、IA-64 のメモリ参照機能について説明する。

10.3.1 コントロール依存

分岐するときの方向によって命令が実行されるかどうかが影響を受ける場合に、その命令は分岐上で*コントロールに依存*していると言う。次のコードでは、ロード命令が分岐上でコントロールに依存している。

```
(p1)br.cond some_label
    ld8 r4=[r5]
```

以下の項では、コントロール依存と、その最適化に与える影響について概説する。

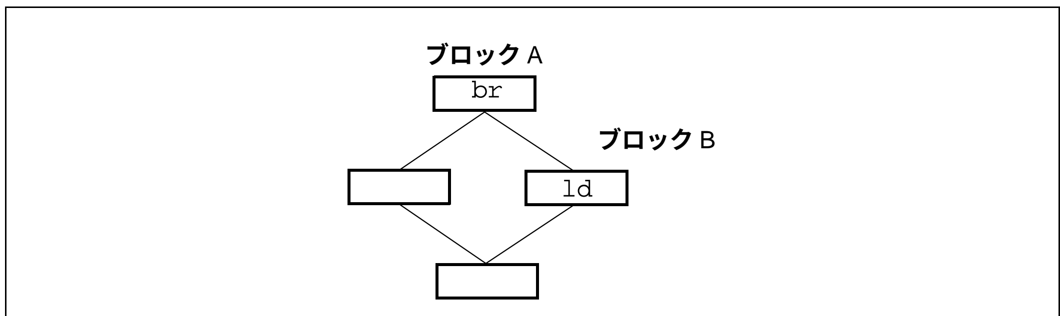
10.3.1.1 命令のスケジューリングとコントロール依存

次のコードは、分岐命令上でコントロール依存を含んでいる。

```
    add r7=r6,1           // Cycle 0
    add r13=r25,r27
    cmp.eq p1,p2=r12,r23
(p1)br.cond  some_label ;;

    ld4 r2=[r3] ;;        // Cycle 1
    sub r4=r2,r11         // Cycle 3
```

コンパイラは、ロード命令を移動したときに、致命的なプログラム・フォルトが発生したり、プログラムの状態が異常を起こしたりしないことを保証できない限り、分岐よりも前にロード命令を安全に移動することはできない。ロード命令を前に移動することはできないので、通常のコード移動を使ってスケジュールを改善することはできない。

つまり、この分岐は、実行が分岐に依存している命令のバリアとなっている。次の図のブロック B のロード命令は、ブロック A の終わりにある条件付き分岐のためにその前に移動することができない。



10.3.2 データ依存

レジスタまたはメモリ位置にアクセスする命令と、同じレジスタまたはメモリ位置を変更する別の命令の間には、データ依存関係が存在する。

10.3.2.1 データ依存の基本事項

以下に、命令間のデータ依存に関連する基本的な用語を示す。

ライト・アフター・ライト (WAW)　同じレジスタまたはメモリ位置に書き込みを行う 2 つの命令の間の依存。

ライト・アフター・リード (WAR)　ある命令がレジスタまたはメモリ位置を読み込み、後続の命令がそのレジスタま

	たはメモリ位置に書き込みを行なう場合の、2 つの命令の間の依存。
リード・アフター・ライト (RAW)	ある命令がレジスタまたはメモリ位置に書き込み、後続の命令がそのレジスタまたはメモリ位置を読み込む場合の、2 つの命令の間の依存。
曖昧なメモリ依存	命令がアクセスするメモリ位置がオーバーラップするかどうかが不定の場合の、ロードとストアの間、または 2 つのストアの間の依存。曖昧なメモリ依存には、WAW、WAR、または RAW の依存が含まれる。
独立したメモリ参照	メモリ・アクセスが競合しないことがわかっている 2 つ以上のメモリ命令による参照。

10.3.2.2 IA-64 におけるデータ依存

IA-64 アーキテクチャでは、プログラマは正しいコード結果を得るために、RAW と WAW のレジスタ依存関係の間にストップを挿入する必要がある。たとえば、次のコードの add 命令は、sub 命令が必要とする r4 の値を計算している。

```
add r4=r5,r6 ;;// Instruction group 1
sub r7=r4,r9   // Instruction group 2
```

add 命令の後にストップを挿入することで、sub 命令が r4 を正しく読み込めるように、1 つの命令グループを終了させている。

一方、IA-64 プロセッサでは、そのアーキテクチャ上の理由から、1 つの命令グループ内でメモリ・ベースの依存関係を守る必要がある。プログラム上で、1 つの命令グループの中にメモリ・ベースのデータ依存命令を含めることができる。この場合でも、命令がシーケンシャルにプログラムの順序で実行されたときと同じ結果がハードウェアによって生成される。次の擬似コードは、ハードウェアによってメモリの依存関係が守られていることを示している。

```
mov r16=1
mov r17=2 ;;
st8 [r15]=r16
st8 [r14]=r17 ;;
```

r14 のアドレスが r15 のアドレスと等しい場合、ユニプロセッサ・ハードウェアでは、このメモリ位置に r17 の値 (2) が必ず格納される。次の RAW の依存関係も、ソフトウェア上では r1 と r2 がオーバーラップするかどうかを判定できない場合でも、同じ命令グループの中で適正に使用することができる。

```
st8 [r1]=x
ld4 y=[r2]
```

10.3.2.3 命令のスケジューリングとデータ依存

正しいコードを生成するためには依存に関する規則だけで十分であるが、効率的なコードを生成するためには、コンパイラは命令のレイテンシを考慮に入れなくてはならない。たとえば汎用のターゲットでは、第 1 レベルのデータ・キャッシュに対して 2 サイクルのレイテンシを持つ。次のコードでは、ストップによって正しい順序が保証されているが、r2 の使用はそのロード命令の 1 サイクル後にスケジュールされる。

```
add r7=r6,1             // Cycle 0
add r13=r25,r27
cmp.eq p1,p2=r12,r23 ;;

add r11=r13,r29         // Cycle 1
ld4 r2=[r3] ;;

sub r4=r2,r11           // Cycle 3
```

ロードのレイテンシは 2 サイクルなので、sub 命令はサイクル 3 までストールされる。ストールを避け、マシンが各サイクルを有効に利用できるようにするために、コンパイラによってロード命令をスケジュールの前方に移動することができる。

```
ld4 r2=[r3]             // Cycle 0
add r7=r6,1
add r13=r25,r27
cmp.eq p1,p2=r12,r23 ;;

add r11=r13,r29 ;;      // Cycle 1

sub r4=r2,r11           // Cycle 2
```

このコードには依存関係のない命令が十分にあるため、ロード命令をスケジュールの前方に移動することで、機能ユニットをより有効に利用し、実行時間を 1 サイクル減らすことができる。

ここで、本来のコード・シーケンスに、ストア命令とロード命令の間に曖昧なメモリの依存関係が含まれているものとする。

```
add r7=r6,1             // Cycle 0
add r13=r25,r27
cmp.ne p1,p2=r12,r23 ;;

st4 [r29]=r13           // Cycle 1
ld4 r2=[r3] ;;

sub r4=r2,r11           // Cycle 3
```

この場合には、メモリの依存関係のために、ロード命令をストア命令よりも前に移動することはできない。ストアがロードや他のストアとの間の曖昧な関係を解消できない場合には、ストアによってデータ依存が発生する。

アーキテクチャ上のサポートがなければ、ストア命令があることによって、ロード命令や他の依存命令を移動することはできない。次の C 言語のステートメントでは、ptr1 と ptr2 が独立したメモリ位置を指しているということが静的にわかっていない限り、順序を変更することはできない。

```
*ptr1 = 6;
x = *ptr2;
```

10.4 依存性を解消するための IA-64 スペキュレーションの使用法

データとコントロールの両方の依存関係によって、プログラム・コードの最適化が制約される。IA-64 では、依存を解消するために使用される 2 つの基本的なテクニックをサポートしている。

データ・スペキュレーション	ロードおよびその値を使用する命令を、曖昧なメモリ書き込みを越えて移動することができる。
コントロール・スペキュレーション	ロードおよびその値を使用する命令を、コントロール依存が生じている分岐を越えて移動することができる。

以下に、ロードのレイテンシを隠し、実行時間を短縮するために使用されるテクニックを示す。

10.4.1 IA-64 スペキュレーション・モデル

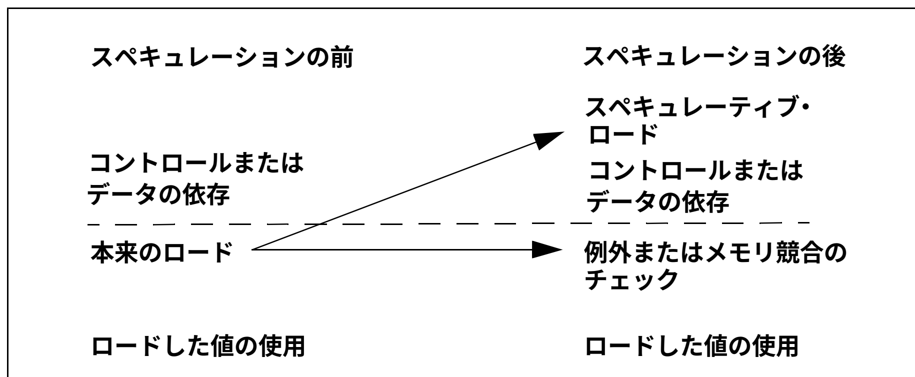
命令スケジューリングに対する依存関係によって引き起こされる制限は、データのロードを、例外処理またはデータ競合の認知から分離することによって解決できる。IA-64 では、これを実現するための特殊なスペキュレーティブな命令をサポートしている。

- コントロール・スペキュレーティブ・ロード命令では例外を据え置く。
- データ・スペキュレーティブ・ロード命令ではアドレス情報を保存する。
- 特殊なチェック命令で例外やデータ競合をチェックする。

IA-64 のスペキュレーティブ・ロードは、次の図に示すように、依存関係のバリア (破線) を越えて移動することができる。

チェックによって、据え置かれた例外や途中にあるストア命令との競合を検出し、失敗したスペキュレーションからリカバリするためのメカニズムを提供する。このサポートにより、スペキュレーティブ・ロード命令とその値を

使用する命令を、非スペキュレーティブ命令よりも前にスケジュールすることができる。その結果、これらのロードによるメモリ・レイテンシは、非スペキュレーティブ・ロードよりも簡単に隠すことができるようになる。



10.4.2 IA-64 のデータ・スペキュレーションの使用法

IA-64 のデータ・スペキュレーションは、アドバンスド・ロード命令と呼ばれる特殊なロード命令 (`ld.a`) と、それに対応するチェック命令 (`chk.a` または `ld.c`) を使用して、データ・スペキュレーションの結果を確認する。

`ld.a` 命令が実行されると、アドバンスド・ロード・アドレス・テーブル (ALAT) と呼ばれるハードウェア構造の中にエントリが割り当てられる。ALAT は物理的なレジスタ番号によってインデックス付けされており、ロード・アドレス、ロードのタイプ、およびロードのサイズを記録する。

アドバンスド・ロード命令の結果を非スペキュレーティブ命令で使用するためには、その前にチェック命令が実行されなくてはならない。チェック命令には、対応するアドバンスド・ロード命令と同じレジスタ番号を指定しなくてはならない。

チェック命令が実行されると、ALAT の中で、同じターゲット物理レジスタ番号と同じタイプを持つエントリが検索される。エントリが見つかると、次の命令から通常どおりに続行される。

マッチするエントリが見つからなかった場合には、スペキュレーティブな結果を再計算する必要がある。

- ロードとその値を使用する命令の一部がスペキュレーションの結果である場合には、`chk.a` 命令を使用する。`chk.a` は、コンパイラが生成したリカバリコードにジャンプして、ロード命令とその依存命令を再実行する。
- ロードした値を使用する命令に対してスペキュレーションが行われていない場合は、`ld.c` 命令を使用する。`ld.c` はロードを再発行する。

以下の条件で、ALAT のエントリは削除される。

- ALAT エントリとオーバーラップするアドレスに書き込みを行うストア。
- ALAT エントリと同じ物理レジスタをターゲットとする別のアドバンスド・ロード。
- 正確さを保持するために必要な、定義されているハードウェアまたはオペレーティング・システム上の条件。
- ALAT のハードウェア上の容量、結合能力、および照合アルゴリズム上の制限。

10.4.2.1 アドバンスド・ロードの例

アドバンスド・ロードを使用することで、命令のシーケンスにおけるクリティカル・パスを短縮することができる。次のコードでは、ロード命令とストア命令が競合するメモリ・アドレスにアクセスしている可能性がある。

```
st8 [r4]=r12        // Cycle 0: ambiguous store
ld8 r6=[r8] ;;      // Cycle 0: load to advance
add r5=r6,r7 ;;     // Cycle 2
st8 [r18]=r5        // Cycle 3
```

上のコードは汎用マシン・モデルでは 4 サイクルで実行されるが、アドバンスド・ロード命令とチェック命令を使うと、次のように書き換えることができる。

```
ld8.a r6=[r8]       // Cycle -2 or earlier

// Other instructions

st8 [r4]=r12        // Cycle 0: ambiguous store
ld8.c r6=[r8]       // Cycle 0: check load
add r5=r6,r7 ;;     // Cycle 0
st8 [r18]=r5        // Cycle 1
```

本来のロード命令はチェック・ロード命令に変更され、アドバンスド・ロード命令が曖昧なストア命令の前にスケジュールされている。スペキュレーションが成功すると、アドバンスド・ロードによるレイテンシが隠されるため、残りの非スペキュレーティブなコードの実行時間が短縮される。

10.4.2.2 リカバリコードの例

前項の非スペキュレーティブなコードを再び例にとる。

```
st8 [r4]=r12        // Cycle 0: ambiguous store
ld8 r6=[r8] ;;      // Cycle 0: load to advance
add r5=r6,r7 ;;     // Cycle 2
st8 [r18]=r5        // Cycle 3
```

コンパイラは、ロード命令だけでなく、そのロード結果を使用する命令も前方に移動することができる。この変換では、アドバンスド・ロード命令の確認のために、ld.c 命令ではなく chk.a 命令を使用する。同じコード・シーケンスの例を使い、ld8 命令だけでなく add 命令も前方に移動させた結果を次に示す。

```
    ld8.a   r6=[r8] ;; // Cycle -3

    // other instructions

    add     r5=r6,r7   // Cycle -1: add that uses r6

    // Other instructions

    st8     [r4]=r12   // Cycle 0
    chk.a   r6,recover // Cycle 0: check
back: // Return point from jump to recover
    st8     [r18]=r5   // Cycle 0
```

リカバリコードも生成する必要がある。

```
recover:
    ld8     r6=[r8] ;; // Reload r6 from [r8]
    add     r5=r6,r7   // Re-execute the add
    br      back       // Jump back to main code
```

スペキュレーションが失敗すると、チェック命令によってラベル recover に分岐し、ここでスペキュレーションが行われたコードが再実行される。スペキュレーションが成功した場合は、変換されたコードの実行時間は、元のコードよりも 3 サイクル短くなる。

10.4.2.3 用語の確認

*アドバンスド・ロード*や*チェック・ロード*などのスペキュレーションに関連する用語は、IA-64 では明確に定義された意味を持っている。以下に、これまでの項で取り上げてきた用語を説明する。

データ・スペキュレーティブ・ロード	依存する 1 つまたは複数のストアよりも前に静的にスケジュールされるスペキュレーティブ・ロード。データ・スペキュレーティブ・ロード命令は ld.a である。
アドバンスド・ロード	データ・スペキュレーティブ・ロード。
チェック・ロード	対応するアドバンスド・ロードを再実行する必要があるかどうかをチェックし、必要があれば再実行を行う命令。チェック・ロード命令は ld.c である。

アドバンスド・ロード・チェック	必要に応じて、レジスタ番号とオフセットをコンパイラが生成した命令セットに渡して、スペキュレーションが行われた命令を再実行する命令。アドバンスド・ロード・チェック命令は chk.a である。
リカバリコード	スペキュレーション・チェックから分岐するプログラム・コード。リカバリコードは、スペキュレーションの失敗からリカバリするために、ロードと一連の依存する命令をやり直す。

10.4.3 IA-64 のコントロール・スペキュレーションの使用法

コントロール・スペキュレーションが成功したかどうかのチェックは、データ・スペキュレーションでのチェックと似ている。

10.4.3.1 NAT ビット

NaT (ノット・ア・シング) ビットは、個々の汎用レジスタが持っている余分な 1 ビットである。レジスタの NaT ビットは、レジスタの内容が有効であるかどうかを示すものである。NaT ビットが 1 にセットされていれば、レジスタには以前のスペキュレーションの失敗を原因とする据え置かれた例外トークンが含まれている。浮動小数点レジスタの場合には、NaTVal と呼ばれる特殊な値によって、例外が据え置かれたことを示す。

コントロール・スペキュレーティブ・ロードの際に、例外が発生し、それが据え置かれた場合には、ロード命令のデスティネーション・レジスタの NaT ビットがセットされる場合がある。例外の据え置き (したがって NaT ビットのセット) を引き起こすイベントと例外の具体的な内容は、部分的にはオペレーティング・システムのポリシーによって異なる。スペキュレーティブ命令が NaT ビットがセットされているソース・レジスタを読み込んだ場合には、その命令のターゲット・レジスタの NaT ビットもセットされる。つまり、NaT ビットは依存する計算の間で伝播していくことになる。

10.4.3.2 コントロール・スペキュレーションの例

コントロール・スペキュレーティブ・ロードがスケジュールされた場合、コンパイラは、スペキュレーティブ・ロードの結果が使用されるすべてのパス上で、スペキュレーティブ・チェック命令の chk.s を挿入しなくてはならない。(chk.s 以外の) 非スペキュレーティブ命令が NaT ビットがセットされているレジスタを読み込むと、Nat 参照フォルトが発生し、オペレーティング・システムによってプログラムが終了する。

次のコード・シーケンスは、IA-64 のコントロール・スペキュレーションの基本的な使用方法を示している。

```
(p1)br.cond some_label // Cycle 0
    ld8 r1=[r5] ;;      // Cycle 1
    add r2=r1,r3        // Cycle 3
```

このコードは、コントロール・スペキュレーティブ・ロード命令とチェック命令を使って書き換えることができる。チェック命令は本来のロード命令と同じ基本ブロックの中に配置できる。

```
    ld8.s r1=[r5] ;;    // Cycle -2

    // Other instructions

(p1)br.cond some_label // Cycle 0
    chk.s r1,recovery   // Cycle 0
    add  r2=r1,r3       // Cycle 0
```

スペキュレーション・チェック命令に動的に到達するまでに、コントロール・スペキュレーティブな命令のチェーンの結果をメモリに格納したり、非スペキュレーティブな命令でアクセスしたりすると、フォルトが発生する。スペキュレーション・チェック命令が実行され、チェックされたレジスタの NaT ビットがセットされていた場合、プロセッサはチェック命令が指すリカバリコードに分岐する。

また、NaT テスト (tnat) および浮動小数点分類 (fclass) 命令を使って、NaT ビットと NaTVals がセットされているかどうかをテストすることができる。

すべてのスペキュレーティブ計算に対してチェックを行う必要があるが、これはすべてのスペキュレーティブ・ロード命令にそれぞれ chk.s が必要であるという意味ではない。10.5.6 項で説明するように、レジスタ間での NaT ビットの伝播を利用して、スペキュレーティブ・チェックの最適化を行うことができる。

10.4.3.3 スピル、フィル、および UNAT レジスタ

NaT ビットがセットされている可能性のあるレジスタに対する保存と復元は、st8.spill 命令および ld8.fill 命令と、ユーザ NaT コレクション (UNAT) アプリケーション・レジスタによって可能になる。

「汎用レジスタおよび NaT のスピル」命令である st8.spill は、汎用レジスタの 8 バイトをメモリに保存し、その NaT ビットを UNAT に書き込む。ストア命令のメモリ・アドレスのビット 8:3 によって、どの UNAT ビットにレジスタの NaT 値が書き込まれているかがわかる。「汎用レジスタのフィル」命令である ld8.fill は、メモリから 8 バイトを汎用レジスタに読み込み、UNAT の値に従ってレジスタの NaT ビットをセットする。NaT ビットのスピルとフィルが正しく行われるためには、ソフトウェアによって UNAT の内容を保存し、復元する必要がある。

対応する浮動小数点命令の stf.spill と ldf.fill は、NaTVal によって例外が表面化することなく、浮動小数点レジスタ形式で浮動小数点レジスタの保存と復元を行う。

10.4.3.4 用語の確認

以下にコントロール・スペキュレーションに関連する用語を示す。

コントロール・スペキュレーティブ・ロード	本来配置されていた制御分岐の前にスケジュールされたされるスペキュレーティブ・ロード。修飾語なしに「スペキュレーティブ・ロード」と言った場合は、一般にデータ・スペキュレーティブ・ロードではなくコントロール・スペキュレーティブ・ロードのことを指す。 ld.s 命令によるロードはコントロール・スペキュレーティブ・ロードである。
スペキュレーション・チェック	スペキュレーティブ命令が例外の据え置きを行ったかどうかをチェックする命令。スペキュレーション・チェック命令には、コンパイラ生成のリカバリコードを指すラベルが含まれる。スペキュレーション・チェック命令は chk.s である。
リカバリコード	スペキュレーションの失敗からリカバリするために実行されるコード。コントロール・スペキュレーティブリカバリコードはデータ・スペキュレーティブリカバリコードに対応するものである。

10.4.4 データおよびコントロールのスペキュレーションの組み合わせ

データ・スペキュレーティブとコントロール・スペキュレーティブの両方の機能を持つロードは、*スペキュレーティブ・アドバンスド・ロード*と呼ばれる。ld.sa 命令は、スペキュレーティブ・ロードと*アドバンスド・ロード*の両方のすべての操作を行う。このタイプのロード命令によって据え置き例外のトークンを生成した場合には ALAT エントリの割り当ては行われないので、アドバンスド・ロード・チェック命令 (chk.a) だけでそれ以降のストアによる干渉と据え置き例外の両方をチェックすることができる。

10.5 メモリ参照の最適化

スペキュレーションは、従来のアーキテクチャよりも多くのコード移動を可能にすることによって、並列処理を増加させ、レイテンシを隠すのに貢献する。スペキュレーションによって、不変コードの移動や共通部分式の削除といった従来のループに対する最適化を多く適用することができる。さらに、IA-64 ではポスト・インクリメント機能を持つロード命令およびストア命令も提供しており、コード・サイズを増やさずに命令スループットを改善することができる、。

メモリ参照の最適化は、以下のものを含めて、いくつかの要因を考慮に入れなくてはならない。

- スペキュレーティブおよび非スペキュレーティブなコードの実行コストの違い
- コード・サイズ
- 干渉の確率と ALAT のプロパティ (データ・スペキュレーションの場合)

本章の後半では、これらの要因と、メモリ・アクセスに関連する最適化について説明する。

10.5.1 スペキュレーションに関する考慮事項

データ・スペキュレーションを使用するときには、コントロール・スペキュレーションを使用する場合よりも注意が必要である。これは、一部には、1 つのコントロール・スペキュレーティブ・ロードが意図せずに他のコントロール・スペキュレーティブ・ロードを失敗させることはないという事実による。一方、ALAT の容量には限度があり、ALAT エントリの置換ポリシーはハードウェアによって異なるため、データ・スペキュレーティブ・ロードではこのような影響が生じることがある。たとえば、アドバンスド・ロード命令が発行され、未使用の ALAT エントリが存在しなかった場合、ハードウェアによって新しいエントリ用のスペースを作るために既存のエントリを無効にする場合がある。

さらに、コントロール・スペキュレーティブ計算に付随する例外は、ページ・フォルトや TLB ミスなどのイベントに関係しているため、正しいコードでは普通は見られない。ただし、過剰なコントロール・スペキュレーションは、付随する命令が発行スロットのスペースを占有するので、高いコストがかかることがある。

プログラムの静的なクリティカル・パスはデータ・スペキュレーションの使用によって軽減される場合もあるが、以下の要因がデータ・スペキュレーションの利点や動的なコストに貢献する。

- 途中にあるストアがアドバンスド・ロードを干渉する確率。
- 失敗したアドバンスド・ロードからリカバリするためのコスト。
- ALAT の具体的なマイクロアーキテクチャ上のサイズ、結合能力、および照合アルゴリズム。

干渉の確率を見定めるのは困難かもしれないが、メモリに対する動的なプロファイリングによって、曖昧なロードとストアがどれほどの頻度で衝突するかを予測することができる。

アドバンスド・ロードを使用するときには、ロードだけを前方に移動させ、`ld.c` 命令を使用するという方法と、ロードとその値を使用する命令の両方を前方に移動させる
(この場合にはより高価なコストになる可能性がある `chk.a` 命令を使用する必要がある) という方法のどちらが良いかをケース・バイ・ケースで検討するべきである。

リカバリコードが実行されない場合でも、そのコードがあることによって。データおよびコントロールのスペキュレーションで使用されるレジスタの期間を延長し、結果的にレジスタのプレッシャーを高め、レジスタ・スタック・エンジン (RSE) によるレジスタ移動のコストが増加する可能性がある。リカバリコードを使用するかどうかを検討する際の情報については、10.5.3 項を参照のこと。

10.5.2 データの干渉

データ・スペキュレーションは、干渉の確率が低く、パスの確率が高いデータ参照で、最も有効な手法である。次に示す擬似コードにおいて、*p1 と *p2 へのストアが var と衝突する確率が独立しているものと仮定する。

```
*p1 =       /* Prob interference = 0.30 */
. . .
*p2 =       /* Prob interference = 0.40 */
. . .
    = var   /* Load to be advanced */
```

コンパイラが var からのロードをポインタ p1 と p2 へのストアの前に移動させると、次のようになる。

```
= 1.0 - (Prob p1 will not interfere with var *
        Prob p2 will not interfere with var)
= 1.0 - (0.70 * 0.60)
= 0.58
```

確率が上記のものと仮定すると、var からのロードを p1 と p2 の前に移動させた場合に、var からのロードが p1 と p2 の少なくともどちらかと干渉する確率は 58% となる。コンパイラはデータ干渉に関する従来のヒューリスティックスと、プロシージャ間のメモリ・アクセス情報を使用して、これらの確率を推定することができる。

ロードを関数コールを越えて前方に移動させるときには、次の点を考慮する必要がある。

- コールされた関数に多数のストアが含まれている場合には、実際の、または別名化された ALAT の衝突が起こる可能性が高い。
- 関数コールの途中でさらにアドバンスド・ロードが実行された場合には、その物理レジスタ番号が、親関数内でのコールで割り当てられた ALAT エントリと同じになるか競合する可能性がある。
- コールされたルーチンによって多数のアドバンスド・ロードが実行されるかどうかが不明な場合には、その ALAT の容量を超過する可能性を考慮する必要がある。

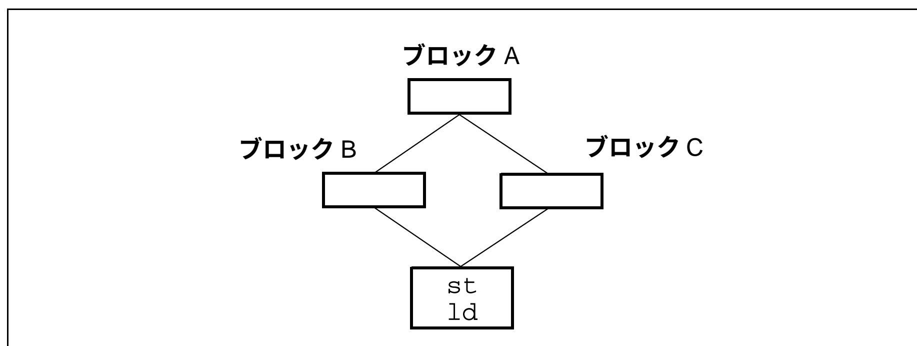
10.5.3 コード・サイズの最適化

いつスペキュレーションを行うかという決定の一環として、コード・サイズの増大を考慮に入れるべきである。*このような検討はスペキュレーションに限ったことではなく、ループのアンロール、プロシージャのインライン化、テールの複製など、コードの重複を引き起こすすべての変換の際に行わなくてはならない。*コード・サイズの増大を最小限に抑えるためのテクニックについては、本項で後に説明する。

一般に、コントロール・スペキュレーションでは、スペキュレーションによる命令が実行されても、その結果が一度も使用されない場合があるため、プログラムの動的なコード・サイズが増大する。コントロール・スペキュレーションに付随するリカバリコードはもっぱらバイナリの静的サイズにのみ影響を与える。これはライン外に配置され、スペキュレーティブな計算が失敗する (コントロール・スペキュレーションではめったに起こらない) まではキャッシュに格納されないためである。

データ・スペキュレーションにおいてもコード・サイズに対して似たような影響を与えるが、ほとんどの非コントロール・スペキュレーティブなデータ・スペキュレーティブ・ロードでは、その結果のチェックが行われるので、一度も使用されない値が計算される可能性は低いという違いがある。また、コントロール・スペキュレーティブ・ロードは、(オペレーティング・システムの設定に応じて) 据え置かれたデータ関連のフォルトなどの異常な状況でのみ失敗するが、データ・スペキュレーティブ・ロードでは ALAT の競合、実メモリの競合、または ALAT 内の別名化などが原因となって失敗することがあるため、アドバンスド・ロードのリカバリコードをどこに配置するかという決定はコントロール・スペキュレーションの場合よりも難しく、各ロードの予想される競合確率に基づいて行われなくてはならない。

一般的な原則として、効率的なコンパイラはスペキュレーションに関連するコードの増大を最小限に抑えようと試みる。たとえば、2 つのパスの結合部分の前にロード命令を移動するためには、それぞれのパスでスペキュレーティブ・コードを複製しなくてはならない場合がある。次のフロー図と説明は、この状況を示している。



コンパイラまたはプログラマが、ロード命令を本来の非スペキュレーティブな位置からブロック B に移動させた場合には、すべてのスペキュレーティブ・コードをブロック B と C の両方に複製しなくてはならない。この複製されたコードによって、すでに存在する NOP スロットを占有する場合もありうる。しかし、コード用の容量が十分でない場合には、ロード命令をブロック A に移動させる方が望ましい。そうすればコピーは 1 つで済むからである。

10.5.4 ポスト・インクリメントのロードおよびストアの使用法

ポスト・インクリメント機能を持つロードおよびストアは、2 つの操作を 1 つの命令に組み合わせることでパフォーマンスを改善することができる。本項ではポスト・インクリメントのロードだけについて説明するが、ほとんどの情報はストアにも当てはまる。

ポスト・インクリメント・ロードは M ユニット上で発行され、そのアドレス・レジスタを即値または汎用レジスタの内容の分だけインクリメントすることができる。次の擬似コードは 2 つのロード命令を実行している。

```
ld8 r2=[r1]
add r1=1,r1 ;;
ld8 r3=[r1]
```

これは、ポスト・インクリメント・ロードを使って次のように書き換えることができる。

```
ld8 r2=[r1],1 ;;
ld8 r3=[r1]
```

ポスト・インクリメント・ロードは依存パスの高さを直接に減らすことはできないかもしれないが、それ以降のロード命令で使用されるアドレスを計算するときに重要である。

- ポスト・インクリメント・ロードは、2 つの命令を 1 つにまとめることによって、コード・サイズの増大を防ぐ。
- 加算は I ユニットと M ユニットのどちらでも発行できる。プログラムが加算をロードと組み合わせると、I ユニットまたは M ユニットのリソースは空いたままとなる。このため、ポスト・インクリメント・ロードを使えば、依存する加算とロードのスループットが 2 倍になる。

ポスト・インクリメント・ロードの短所は、ポスト・インクリメント・ロードと、ポスト・インクリメント値を使用する操作の間に新しい依存関係を作り出すという点にある。ケースによっては、コンパイラは全体的なスケジュールを改善するために、ポスト・インクリメント・ロードをその構成命令に分解する場合もある生じる。一方、コンパイラは命令スケジューリングが終わるまで待ってから、別々のロード命令と加算命令をポスト・インクリメント・ロードに置き換えられるような部分を場当たり式に探すという方法を取ることもできる。

10.5.5 ループの最適化

循環的なコードでは、スペキュレーションは不変コードの移動といった古典的なループに対する最適化の使用を拡張することができる。次の擬似コードを例にとる。

```
while (cond) {
    c = a + b; // Probably loop invariant
    *ptr++ = c;// May point to a or b
}
```

変数 a と b はおそらくループ不変である。しかしコンパイラは、分析の結果としてそうならないことが保証されない限り、*ptr へのストアによって a と b の値が上書きされると仮定しなければならない。アドバンスド・ロード命令とチェック命令を使用することにより、ポインタの曖昧さを解消できない場合でも、不変であると思われるコードをループから削除することができる。

```
    ld4.a r1 = [&a]
    ld4.a r2 = [&b]
    add r3 = r1,r2// Move computation out of loop
    while (cond) {
        chk.a.nc r1, recover1
L1:     chk.a.nc r2, recover2
L2:     *p++ = r3
    }
```

モジュールの終わりに、次のコードを追加する。

```
recover1:       // Recover from failed load of a
    ld4.a r1 = [&a]
    add  r3 = r1, r2
    br.sptk L1  // Unconditional branch

recover2:       // Recover from failed load of b
    ld4.a r2 = [&b]
    add  r3 = r1, r2
    br.sptk L2  // Unconditional branch
```

このループでは、スペキュレーションを使用することで、スペキュレーションされるコードが成功した場合に、c の計算のレイテンシが隠される。

チェック命令にはクリア形式 (clr) と非クリア形式 (nc) の両方があるため、プログラマはどちらを使用するかを決定しなくてはならない。この例は、チェック命令がループの外に移動される場合には、非クリア形式を使用すべきであることを示している。これは、クリア形式 (clr) を使用すると、対応する ALAT エントリが削除されるため、そのレジスタに対する次のチェック命令で失敗するからである。

10.5.6 チェック・コードの最小化

スペキュレーティブ・ロードのチェック命令を組み合わせて、コード・サイズを削減できることがある。スペキュレーティブ命令を介しての NaT ビットと NaTVals の伝播により、複数の中間的なチェック命令をスペキュレーティブな結果に対する 1 つのチェック命令に置き換えることができる。次のコードは、この最適化の可能性を示している。

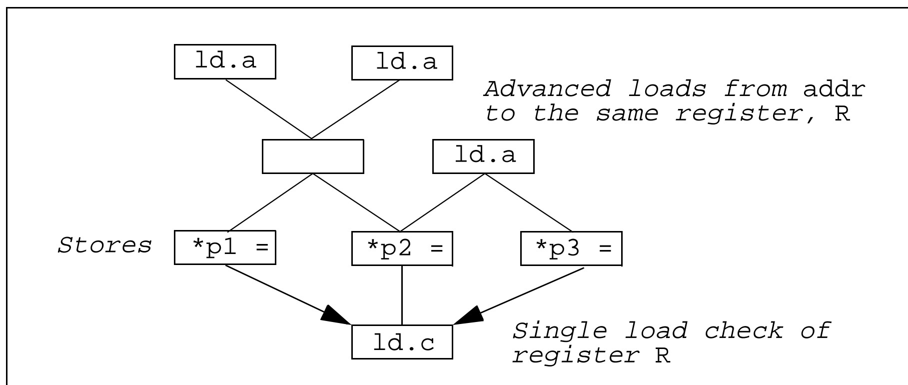
```
ld4.s r1=[r10] // Speculatively load to r1
ld4.s r2=[r20] // Speculatively load to r2
add  r3=r1,r2;;// Add two speculative values

// Other instructions

chk.s r3,imm21  // Check for NaT bit in r3
st4 [r30]=r1    // Store r1
st4 [r40]=r2    // Store r2
st4 [r50]=r3    // Store r3
```

r1、r2、または r3 のストアの前にチェックしなければならないのは、結果レジスタ r3 だけである。r1 または r2 のコントロール・スペキュレーティブ・ロードの時点で NaT ビットがセットされていたとすると、NaT ビットは add 命令を介して r1 または r2 から r3 に伝播しているはずだからである。

チェック・コードの量を減らすもう 1 つの方法は、コントロール・フロー分析を使用して、余分な ld.c 命令または ld.a 命令の発行を避けるというものである。たとえば、コンパイラはアドバンスド・ロードのすべてのコピーが到達することがわかっている位置に 1 つのチェック命令をスケジュールすることができる。次の図のフロー・グラフの一部は、このテクニックの適用例を示している。

次の 2 つの条件が満たされていれば、一番下のブロックにおける 1 つのチェック命令で、すべてのアドバンスド・ロードのチェックを行うことができる。

- 一番下のブロックが、位置 addr からのアドバンスド・ロードを含むすべてのブロックをポスト・ドミネートしている。
- 一番下のブロックが、addr からのアドバンスド・ロードの結果を使用するすべての命令よりも先にある。



10.6 要約

本章の例は、IA-64 が動的プロファイリングや曖昧さの解消といった既存のテクニックを利用できる場合を示している。特殊な IA-64 サポートにより、一般的なシナリオで、通常では不可能なスペキュレーションが可能となる。一方、スペキュレーションによって、より多くのコード移動を可能にすることで ILP を向上させ、ループなどに対する従来の最適化を強化できる。

IA-64 のスペキュレーション・モデルはさまざまな状況に適用することができるが、最適なパフォーマンスを得るためには、慎重にコストおよび利点を分析する必要がある

intel®

プレディケーション、コントロール・フロー、命令ストリーム 11

11.1 概要

本章は、プレディケーション、コントロール・フロー、および分岐ヒントに関連する最適化について説明する 3 つの節から構成されている。

- **プレディケーション**の節では、if 変換、プレディケーションの使用、および分岐の影響を減らすためのコード・スケジューリングについて説明する。
- **コントロール・フローの最適化**の節では、並列比較、マルチウェイ分岐、およびプレディケーションの下での複数レジスタの書き込みを使用してコントロール・フローの縮小と収束を行う最適化について説明する。
- **分岐ヒントとプリフェッチ・ヒント**の節では、ヒントを使って分岐とプリフェッチのパフォーマンスを改善する方法について説明する。

11.2 プレディケーション

プレディケーションにより、コンパイラはコントロール依存をデータ依存に変換することができる。本節では分岐関連のパフォーマンス上の問題の原因をいくつか示し、IA-64 のプレディケーション・メカニズムを要約し、プレディケーションに基づく最適化とテクニックについて説明する。

11.2.1 分岐のパフォーマンス・コスト

分岐は、実行時の予測のためにハードウェア・リソースを消費し、またコンパイル時の命令スケジューリングの自由度を制約し、アプリケーションのパフォーマンスを低下させることがある。

11.2.1.1 予測リソース

分岐予測のリソースには、分岐ターゲット・バッファ、分岐予測テーブル、およびこれらのリソースを制御するために使用されるロジックが含まれる。正確に予測できる分岐の数は、プロセッサ上のバッファのサイズによって制限されるが、この種のバッファはプログラム内で実行される分岐の総数と比べると一般に小さい。

このような制限のために、分岐を多用するコードの実行時間の多くの部分は、予測リソースを巡っての競合に費やされる可能性がある。さらに、分岐予測のパフォーマンスを決定する主要要因はプレディクタのサイズであるとして

も、分岐によっては異なるタイプのプレディクタを使った方が正確に予測を行えることがある。たとえば、分岐には、静的に予測するのに適したものと、動的に予測するのに適したものがある。動的に予測されるものの中にも、ループ分岐のように他のものよりも重要なものがある。

予測の失敗時のコストは一般にパイプラインの長さに比例するので、長い命令パイプラインを持つプロセッサでは優れた分岐予測が必要不可欠である。このため、予測リソースの使用を最適化すれば、アプリケーションの全体的なパフォーマンスが大幅に改善される可能性がある。

たとえば、次のコードの条件が 30% の確率で予測に失敗し、分岐予測の失敗 1 回ごとに 10 サイクルのペナルティがあるとする。平均すると、このコード・シーケンスを 1 回実行するたびに、分岐予測の失敗のために 3 サイクルが費やされることになる (30% * 10 サイクル)。

```
if (r1)
    r2 = r3 + r4;
else
    r7 = r6 - r5;
```

次に、これと同等な、最適化されていない IA-64 のコードを示す。2 つの分岐を含む 5 つの命令があり、予測の失敗の可能性や分岐を行ったために起こるペナルティ・サイクルを計算に入れないと、2 サイクルで実行される。

```
     cmp.eq     p1,p2=r1,r0     // Cycle 0
(p1) br.cond    else_clause     // Cycle 0
     add        r2=r3,r4        // Cycle 1
     br         end_if          // Cycle 1
else_clause:
     sub        r7=r6,r5        // Cycle 1
end_if:
```

前記の情報を使うと、このコードは、クリティカル・パスの長さが 2 サイクルでしかないのにもかかわらず、実行には平均して 5 サイクル必要となる (2 サイクル + (30% * 10 サイクル) = 5)。リソースの競合を減らすか、分岐そのものを削除することによって、分岐予測の失敗のペナルティをなくすことができれば、コード・シーケンスのパフォーマンスは 2 倍も改善される。

11.2.1.2 命令スケジューリング

分岐によって、メモリ状態を変更したり例外を発生する可能性のある命令をコンパイラが移動するのが制限される。プログラム内の命令は、その命令を字句的に囲んでいるすべての分岐に対してコントロール依存の関係にあるからである。コントロール依存に加えて、複合条件の計算には数サイクルかかることがあり、ショートサーキット評価を必要とする C などの言語では、それ自身が中間分岐を必要とすることもある。

IA-64 コンパイラでグローバルなコード移動を行うために使われる主なメカニズムとして、コントロール・スペキュレーションがある。しかし、命令がスペキュレーティブな形式を持っていない場合、あるいは命令がメモリ状態を破壊する可能性がある場合には、コントロール・スペキュレーションだけではコード移動が可能にならないことがある。このため、コード移動の自由度をさらに高め、コンパイラの命令スケジューリングの能力を改善するテクニックが重要となる。

11.2.2 IA-64 におけるプレディケーション

ここまでは、分岐がパフォーマンスに与える影響について説明してきた。本項では、本節で説明する最適化で使用される主要な IA-64 メカニズムであるプレディケーションの概要を説明する。

ほぼすべての IA-64 命令は、保護的なプレディケートでタグ付けを行うことができる。実行時に保護プレディケートの値が偽であれば、プレディケートを持つ命令はアーキテクチャ内で更新が停止され、命令は nop として動作する。プレディケートが真ならば、命令はプレディケートが付いていないものとして動作する。ただし、無条件比較や、浮動小数点の平方根や逆数の近似命令のように、修飾プレディケートがこのように動作しない命令も少数ながら存在する。詳細については、第 I 部 :「IA-64 アプリケーション・アーキテクチャ・ガイド」を参照のこと。

次のシーケンスは、プレディケートの付いた命令のセットを示している。

```
(p1) add   r1=r2,r3
(p2) ld8   r5=[r7]
(p3) chk.s r4,recovery
```

予測レジスタの値を設定するために、IA-64 には次の比較命令およびテスト命令が用意されている。

```
    cmp.eq  p1,p2=r5,r6
    tbit    p3,p4=r6,5
```

また、プレディケートは、その生成命令とその使用を分離するために、ほとんどの場合にストップを必要とする。

```
    cmp.eq  p1,p2=r1,r2 ;;
(p1)add     r1=r2,r3
```

この規則の唯一の例外は、後続の分岐命令の条件として使用されるプレディケートを設定するための整数比較命令またはテスト命令である。

```
    cmp.eq   p1,p2=r1,r2   // No stop required
(p1)br.cond some_target
```

11.2.3 プレディケーションによるプログラム・パフォーマンスの最適化

本項では、プレディケーション関連の最適化、その使用方法、および基本的なパフォーマンス分析テクニックについて説明する。if 変換、予測失敗の削減、オフパス・プレディケーション、コード前方移動、およびコード広報移動などの最適化について説明する。

11.2.3.1 If 変換の適用

プレディケーションによって可能になる最も重要な最適化として、一部のプログラム・シーケンスから分岐を完全に削除できるということである。次の擬似コードは、プレディケーションを使用なければ、if ブロック・コードの前後に条件付きでジャンプする分岐命令が必要となる。

```
    if (r4) {
        add  r1=r2,r3
        ld8  r6=[r5]
    }
```

プレディケーションを使用すると、このシーケンスは分岐なしで書くことができる。

```
    cmp.ne  p1,p0=r4,0 ;;// Set predicate reg
(p1)add     r1=r2,r3
(p1)ld8     r6=[r5]
```

条件付きブロックの中の命令にプレディケートを付けて分岐を削除するプロセスは、*if 変換*と呼ばれる。*if 変換*が行われると、コード移動を制限する分岐の数が減り、発行スロットを巡って競合する分岐の数が少なくなるため、命令のスケジューリングの自由度が高まる。

この変換を行うと、分岐を削除できることに加えて、コントロール・フローが変化する可能性が小さくなるため、動的な命令のフェッチがより効率的になる。より複雑な状況では、複数の分岐を削除できることもある。次に C のコード・シーケンスを示す。

```
    if (r1)
        r2 = r3 + r4;
    else
        r7 = r6 - r5;
```

これは、IA-64 アセンブラでは、分岐なしに次のように書き換えることができる。

```
     cmp.ne p1,p2 = r1,0 ;;
(p1) add    r2 = r3,r4
(p2) sub    r7 = r6,r5
```

それぞれの条件における各命令に相補的なプレディケートが付けられているため、両者は競合しないことが保証される。このため、コンパイラのスケジューリングの自由度が高まり、ハードウェア・リソースを最適に利用できるようになる。また、if 変換の一部としていくつかの分岐とラベルが削除されているため、コンパイラはこれらのステートメントを前後のコードと一緒にスケジュールするよう試みることもできる。

分岐が削除されたため、分岐予測の失敗は起こりえず、分岐が発生した結果としてパイプライン・バブルが生じることもない。このような効果は多くの大規模なアプリケーションで大きなものとなり、これらの変換によって、分岐によって引き起こされるパイプライン内でのストールやフラッシュを大幅に削減することができる。

上記のコードのコストを、そのさらに上のプレディケーションを使用していない場合のコードと比べると、次のようになる。

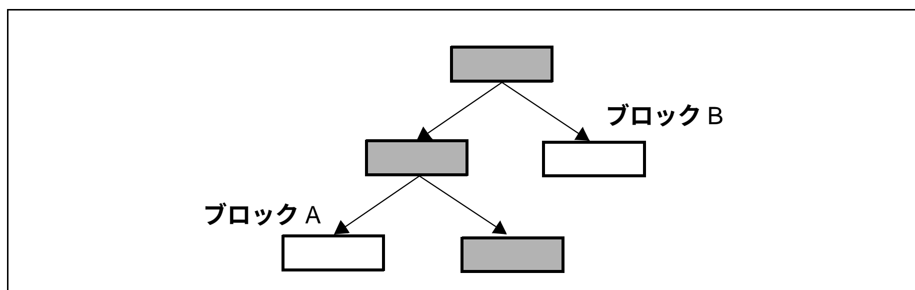
- プレディケーションを使用しないコードの消費サイクル : 2 サイクル + (30% * 10 サイクル) = 5 サイクル
- プレディケーションを使用するコードの消費サイクル : 2 サイクル

この例では、プレディケーションによって平均して 3 サイクル節約できる。

11.2.3.2 オフパス・プレディケーション

コンパイラが動的プロファイル情報を持っている場合には、実行される可能性が最も高いコントロール・フロー・パスに基づいて命令スケジュールを作成することが可能となる。このパスをメイン・トレースと呼ぶ。一部のケースでは、メイン・トレース上にない実行パスも頻繁に実行されるため、それらのクリティカル・パスもプレディケーションを使って最小限にすると効果的である。

次の図では、フローグラフのメイン・トレースを強調して示している。ブロック A とブロック B は、メイン・トレース上にないにもかかわらず、頻繁に実行されると仮定する。



ブロック A またはブロック B の一部の命令を、そのクリティカル・パスを増やすことなくメイン・トレースに含めることができれば、コード前方移動のテクニックを適用して、ブロック A とブロック B を通るクリティカル・パスを減らすことができる。次の項では、プレディケーションを使ってコード前方移動を行う例を示す。

11.2.3.3 コード前方移動

従来のコントロール・スペキュレーションが不適切な場合でも、命令にプレディケートを付け、スケジュール内で上下に移動して依存ツリーの高さを減らすことがある。これが可能なのは、命令にプレディケートを付けることによって、コントロール依存がデータ依存に置き換えられるからである。データ依存がコントロール依存よりも制約が緩ければ、このような変換によって命令スケジュールが改善されることがある。

次の IA-64 のアセンブリ・シーケンスでは、ストア命令を、それを囲む条件命令の前に移動することはできない。これを行うと、アドレス・フォルトや、分岐の方向によってはそれ以外の例外が発生する可能性があるからである。

```
(p1)br.cond some_label     // Cycle 0
    st4     [r34] = r23    // Cycle 1
    ld4     r5 = [r56]     // Cycle 1
    ld4     r6 = [r57]     // Cycle 2:no cycle 1 M's
```

ストア命令を前方に移動するのが望ましい理由の 1 つは、これによってその後のロード命令が前方に移動できるということである。

注: 曖昧なストア操作は、通常のロードを越えて移動させることができないバリアとなる。この例では、ストア命令を移動すると、M ユニット・スロットも解放される。ストア命令が分岐の前に来るようにコードを書き換えるためには、`p2` に `p1` の補数を代入する。

```
(p2)st4     [r34] = r23    // Cycle 0
(p2)ld4     r5 = [r56]     // Cycle 0
(p1)br.cond some_label     // Cycle 0
    ld4     r6 = [r57]     // Cycle 1
```

これでストア命令にプレディケートが付けられたので、分岐の発生時にフォルトや例外が起こる可能性はなく、メモリ状態は、本来のストア命令のホーム・ブロックに入った場合にのみ更新される。ストア命令が移動されると、アドバンスド・ロードやスペキュレーティブ・ロードを使用しなくても、ロード命令を移動できるようになる (ただし、発生した分岐パス上で r5 がライブでないという条件がある)。

11.2.3.4 コード後方移動

コード前方移動と同様に、コード後方移動もストア命令が存在する場合には一般に困難である。次のコードは、コードをラベルを越えて後方に移動する例である。これは、プレディケーションを使用できない場合には一般に安全でない変換である。

```
    ld8     r56 = [r45] ;; // Cycle 0: load
    st4     [r23] = r56 ;; // Cycle 2: store
label_A:
    add ...                // Cycle 3
    add ...
```

```
    add ...
    add ... ;;
```

上のコードで、ロードとストアの間のレイテンシが 2 クロックだとする。ロード命令を他の依存関係のために前方に移動できなかった場合、ロード・レイテンシがカバーされるように命令をスケジュールするには、ストア命令をラベルを越えて後方に移動するしかない。

次のコードは、プレディケーションを使ってコードの後方移動を可能にするという全体的なアイデアを示している。実際のコンパイラが生成するコードでは、この例で明示的に算出されているプレディケートがプレディケート・レジスタにすでに存在していて、余分な命令が不要になる場合もある。

```
    // Point which "dominates" label_A
    cmp.ne p1,p0 = r0,r0   // Initialize p1 to false

    // Other instructions

    cmp.eq  p1,p0 = r0,r0  // Initialize p1 to true
    ld8     r56=[r45] ;;   // Cycle 0
label_A:
    add     ...            // Cycle 1
    add     ...
    add     ...
    add     ... ;;
(p1)st4     [r23]=r56      // Cycle 2
```

ここでは、コード後方移動によって 1 サイクルが節約できる。循環スケジューリングや、ストアによって制約されるコード移動、またはループの外のコードを中に入れるといった高度な例もあるが、ここでは説明しない。

11.2.3.5 キャッシュ・ポリューションの削減

実行時に偽となるプレディケートを持つロードとストアでは、一般に、キャッシュ・ラインの削除、入れ替え、または取り込みを引き起こさない。また、IA-64 のコントロール・スペキュレーションやデータ・スペキュレーションの場合のように、余分な命令やリカバリコードが不要である。このため、プレディケーションを使用したときに、IA-64 のデータ・スペキュレーションやコントロール・スペキュレーションとクリティカル・パスの長さが同じになる場合には、プレディケーションを使用する方が望ましいことがほとんどである。

11.2.4 プレディケーションに関する考慮事項

プレディケーションにはさまざまな利点があるが、プレディケーションを使用すべきかどうかを慎重に検討すべきケースもいくつかある。このようなケースは、一般に、実行パスにおいて総レイテンシのバランスが取れていない場合や、メモリ操作に関連するような特定のリソースを過剰に使用している場合などである。

11.2.4.1 バランスの取れていない実行パス

次に示す単純な条件文では、フロー依存ツリーの高さのバランスが取れていない。このシーケンスに対して述語プレディケートの付いていないアセンブリ言語を使用した場合に if ブロックに 2 クロックが必要で、setf に 8 クロック、getf に 2 クロック、および xma に 6 クロック必要だとすると、合計で 18 クロックとなる。

```
if (r4)     // 2 clocks
    r3 = r2 + r1;
else        // 18 clocks
    r3 = r2 * r1;
f (r3);     // An integer use of r3
```

次に、if 変換を行った IA-64 コードを示す。ここに示したサイクル数は p1 と p2 の値に依存しており、またそれぞれに以下に示すレイテンシがあるものとする。

```
    // Issue cycle if p2 is:TrueFalse
    cmp.ne  p1,p2=r4,r0;;  //  0   0
(p1)add     r3=r2,r1       //  1   1
(p2)setf    f1=r1          //  1   1
(p2)setf    f2=r2 ;;       //  1   1
(p2)xma.l   f3=f1,f2,f0 ;; //  9   2
(p2)getf    r3=f3 ;;       //  15  3
(p2)use of  r3             //  17  4
```

このコードは、p2 が真ならば完了までに 18 サイクル、p2 が偽ならば 5 サイクル必要となる。このようなケースを分析するときには、個々のパスに沿って、実行の重み、分岐予測の失敗の確率、および予測コストを検討する必要がある。

以下に示す 3 つのシナリオでは、分岐予測の失敗時のコストが 10 サイクルであるとする。命令キャッシュや分岐実行のペナルティは考慮しない。

11.2.4.2 ケース 1

if 句が 50% の確率で実行され、分岐予測が絶対に失敗しないものとする。必要なクロック数の平均値は次のようになる。

- プレディケートのないコード : (2 サイクル * 50%) + (18 サイクル * 50%) = 10 クロック
- プレディケート付きのコード : (5 サイクル * 50%) + (18 サイクル * 50%) = 11.5 クロック

このケースでは、if 変換を行うと、コードの実行コストが増える。

11.2.4.3 ケース 2

if 句が 70% の確率で実行され、分岐予測が 10% の確率で失敗し、失敗した場合のコストが 10 クロックだとする。必要なクロック数の平均値は次のようになる。

- プレディケートのないコード : (2 サイクル * 70%) + (18 サイクル * 30%) + (10 サイクル * 10%) = 7.8 クロック
- プレディケート付きのコード : (5 サイクル * 70%) + (18 サイクル * 30%) = 8.9 クロック

やはりこのケースでも、if 変換を行うと、コードの実行コストが増える。

11.2.4.4 ケース 3

if 句が 30% の確率で実行され、分岐予測が 30% の確率で失敗するものとする。必要なクロック数の平均値は次のようになる。

- プレディケートのないコード : (2 サイクル * 30%) + (18 サイクル * 30%) + (10 サイクル * 30%) = 16.2 クロック
- プレディケート付きのコード : (5 サイクル * 30%) + (18 サイクル * 70%) = 14.1 クロック

このケースでは、if 変換によって、実行コストを平均して 2 クロック以上*削減*することになる。

11.2.4.5 リソース使用のオーバーラップ

if 変換を行う前に、プログラマは、フロー依存ツリーの高さに加えて、プレディケート付きのブロックが消費する実行リソースも考慮に入れなくてはならない。命令セットの*リソースのアベイラビリティ*の高さとは、実行に必要な実行リソースだけを考慮に入れたときの最小サイクル数である。

次のコードは if-then-else ステートメントから取ったものである。使用されるのは、2 つのロード／ストア (M) ユニットしか持たない汎用マシン・モデルとする。コンパイラがプレディケートを付け、この 2 つのブロックを組み合わせると、ブロック内のリソース・アベイラビリティの高さは 4 クロックとなる。これは、8 つのメモリ操作を発行するために必要な最小限のクロック数である。

```
then_clause:
    ld  r1=[r21]    // Cycle 0
    ld  r2=[r22]    // Cycle 0
    st  [r32]=r3    // Cycle 1
    st  [r33]=r4 ;;// Cycle 1
    br  end_if
else_clause:
    ld  r3=[r23]    // Cycle 0
    ld  r4=[r24]    // Cycle 0
```

```
    st  [r34]=r5    // Cycle 1
    st  [r35]=r6 ;;// Cycle 1
end_if:
```

前の項の例と同様に、予測失敗の確率と分岐実行のペナルティに応じて、プレディケートを付けるべきかどうかは変わってくる。次に 1 つの例を示す。

11.2.4.6 ケース 1

分岐条件の予測が 10% の確率で失敗し、プレディケート付きのコードの実行に 4 クロック必要だとする。必要なクロック数の平均値は次のようになる。

- プレディケートのないコード : (10 サイクル * 10%) + 2 サイクル = 3 サイクル
- プレディケート付きのコード : 4 サイクル

このコードでは、プレディケートを付けることによって、分岐パスのフロー依存ツリーの高さが等しいにもかかわらず、実行時間が増える。

11.2.5 分岐削除に関するガイドライン

次の if 変換に関するガイドラインは、コードのローカルな動作と実行プロファイルだけがわかっているケースに適用される。

1. プレディケートを付けるかどうかを決める際には、両方のパスのフロー依存ツリーとリソース・アベイラビリティの高さを考慮に入れなくてはならない。
2. if 変換が、本来のコード・シーケンスを通る*いずれかの*コントロール・パスの長さを増大させる場合には、プロファイルまたは予測失敗データを慎重に分析して、変換後のコードの実行時間がプレディケートなしのコードと同等であるか、それ以下であることを確認しなくてはならない。
3. if 変換によってかなりの確率で予測が失敗する分岐を削除できる場合には、その変換は、ブロックのバランスが大きく崩れている場合であっても成果があることが多い。予測の失敗は非常に高コストになるからである。
4. if 変換の対象となるパスのフロー依存ツリーの高さがほぼ等しく、両方のストリームを同時に実行できるだけの十分なリソースがある場合には、一般に if 変換を行うほうが有利である。

これらのガイドラインはコード・セグメントの最適化には有効であるが、プログラムによっては、その動作が、分岐全体の動作、コード・サイズの影響の度合い、分岐予測の失敗に費やされる時間の比率といった非ローカルな効果によって制限される場合がある。このような場合には、if 変換を行うか、その他のスペキュレーティブ変換を行うかという決定はより複雑になる。

11.3 コントロール・フローの最適化

プログラム内でよく起こる現象に、複数のコントロール・フローが 1 つのポイントに収束する、あるいは複数のコントロール・フローが 1 つのポイントから開始されるというものがある。前者の場合、複数のコントロール・フローは同じ変数またはレジスタの値を計算しており、結合ポイントは、プログラムが先に進む前に正しい値を選択しなければならないポイントであることが多い。後者の場合には、いくつかの独立したパスが 1 つの条件セットに基づいて選択されるポイントから、複数のフローが開始されることが多い。

これらの多方向の結合と分岐に加えて、複雑な複合条件の計算には、通常は、複数の条件を 1 つに縮小するためのツリー式の計算が必要となる。IA-64 では、このような条件をより少ないツリー・レベルで計算できる特殊な命令が用意されている。

第 3 のコントロール・フロー関連の最適化では、プレディケーションを使って if 変換による命令のフェッチを改善し、効率的にフェッチできる直線的なシーケンスを生成する。本節の後半では、これらのケースの使用方法と最適化について説明する。

11.3.1 並列比較によるクリティカル・パスの短縮

次に示す複合分岐条件の計算には、特殊な命令を持たないプロセッサ上では、複数の命令が必要となる。

```
    if ( rA || rB || rC || rD ) {
        /* If-block instructions */
    }
    /* after if-block */
```

次の擬似コードは、一連の分岐を使った 1 つの解決法を示している。

```
    cmp.ne  p1,p0 = rA,0
    cmp.ne  p2,p0 = rB,0
(p1)br.cond if_block
(p2)br.cond if_block
    cmp.ne  p3,p0 = rC,0
    cmp.ne  p4,p0 = rD,0
(p3)br.cond if_block
(p4)br.cond if_block
    // after if-block
```

このシーケンスは、すべての条件が偽の場合には実行に少なくとも 2 サイクルを必要とし、1 つまたは複数の分岐予測の失敗のためにさらにサイクル数が必要となる可能性がある。別のシーケンスとして、or ツリーによる縮小が考えられる。

```
    or      r1 = rA,rB
    or      r2 = rC,rD ;;
    or      r3 = r1,r2 ;;
    cmp.ne  p1,p2 = r3,0
(p1)br      if_block
```

この解決法では、分岐条件の計算に 3 サイクルを必要とする。この分岐条件を使って if ブロックへの分岐を行うことができる。

注: また、if ブロックに p1 を使ってプレディケートを付けることで、分岐予測の失敗を避けることもできる

複合条件のコストを削減するために、IA-64 には、and および or 操作を持つ式を最適化するための特殊な並列比較命令が用意されている。これらの比較命令は、複数の and/or 比較命令が、1 つの命令グループの中の同じプレディケートをターゲットとすることができるという点で特殊である。この機能のために、複合条件を 1 つのサイクルで解決できる可能性が生じる。

IA-64 では、この使用モデルを正しく動作させるために、任意のコード実行において、特定のプレディケート・レジスタをターゲットとするすべての命令が次のどちらかの条件を満たすように、プログラマが保証する必要がある。

- 同じ値 (0 または 1) を書き込む
- ターゲット・レジスタに何も書き込まない

つまり、この使用モデルでは、並列比較命令がそのターゲット・レジスタの値を更新できないことがある。したがって、通常の比較とは異なり、並列比較命令で使用されるプレディケートは、その命令の前に初期化されなくてはならない。並列比較の操作の詳細については、第 I 部 :「IA-64 アプリケーション・アーキテクチャ・ガイド」を参照のこと。

初期化コードは、命令グループの中で、並列比較命令の前に置かれなくてはならない。ただし、初期化コードはそれ以前の値への依存関係を持っていないので、コードのクリティカル・パスに関与することなく通常にスケジュールすることができる。

次の命令は、上の例のコードを並列比較を使って生成する方法を示している。

```
    cmp.ne      p1,p0 = r0,r0 ;; // initialize p1 to 0
    cmp.ne.or   p1,p0 = rA,r0
    cmp.ne.or   p1,p0 = rB,r0
    cmp.ne.or   p1,p0 = rC,r0
    cmp.ne.or   p1,p0 = rD,r0
(p1)br.cond     if_block
```

また、p1 を使って if ブロックにインラインでプレディケートを付け、予測の失敗を避けることもできる。さらに、並列比較を使うと、より複雑な条件式も生成できる。

```
if ((rA < 0) && (rB == -15) && (rC > 0))
    /* If-block instructions */
```

次のアセンブリ擬似コードは、上記の C コードのシーケンスの例である。

```
cmp.eq       p1,p0=r0,r0;; // initialize p1 to 1
cmp.ne.and   p1,p0=rB,-15
cmp.ge.and   p1,p0=rA,r0
cmp.le.and   p1,p0=rC,r0
```

正しく使用した場合、and と or を指定した比較では、両方のターゲット・プレディケートに同じ値を書き込むか、あるいはターゲット・プレディケートにまったく書き込みを行わない。並列比較のもう 1 つの使用方法は、複合条件の if と else の両方の部分が必要なケースである。

```
if ( rA == 0 || rB == 10 )
    r1 = r2 + r3;
else
    r4 = r5 -  r6;
```

並列比較には、プレディケートとその補数を同時に計算する andcm という形式がある。

```
    cmp.ne          p1,p2 = r0,r0 ;; // initialize p1,p2
    cmp.eq.or.andcm p1,p2 = rA,r0
    cmp.eq.or.andcm p1,p2 = rB,10 ;;
(p1)add             r1=r2,r3
(p2)sub             r4=r5,r6
```

明らかに、これらの命令を他の命令と組み合わせることで、より複雑な条件を作成することができる。

11.3.2 マルチウェイ分岐によるクリティカル・パスの短縮

IA-64 には、複数の条件と複数のターゲットを持つ分岐をサポートするための特殊な命令は存在しないが、1 つの命令グループ内に複数の連続した B シラブルを配置できるという暗黙のサポートが存在する。

次の例は、4 つの可能な後続ブロックを持つ基本ブロックを示している。次の IA-64 マルチターゲット分岐コードでは BBB バンドル・テンプレートを使用しており、ブロック B、ブロック C、ブロック D のいずれかに分岐するか、あるいはブロック A にフォールスルーできる。

```
label_AA:
        ... // Instructions in block AA
{ .bbb
(p1)br.cond label_B
(p2)br.cond label_C
(p3)br.cond label_D
```

```
}
            // Fall through to A
label_A:
        ... // Instructions in block A
```

すべての分岐が相互に排他的でない限り、プログラムの正確さにとっては分岐の順序が重要である。すべての分岐が相互に排他的であれば、コンパイラは任意の順序を選ぶことができる。

11.3.3 プレディケーションによる、複数の値からの変数またはレジスタの選択

プログラムでは、同じ変数に対して異なる値を計算する複数のパスが結合し、そこから実行が続けられるということがよく行われる。これのバリエーションとして、複数の異なるパスが異なる結果を計算しなければならないが、これらのパスが相補的であることがわかっているため、同じレジスタを使ってもかまわない場合がある。このようなケースでは、プレディケーションを使って最適化を行うことができる。

11.3.3.1 複数の値から 1 つの値を選択する

1 つの変数についてそれぞれ異なる値を計算する複数コントロール・パスが合流する場合には、通常、変数の更新に使用する値を選択するために、一連の条件が必要となる。プレディケーションを使用すると、このコードを分岐なしに効率的に実行することができる。

```
switch (rW)
case 1:
    rA = rB + rC;
    break;
case 2:
    rA = rE + rF;
    break;
case 3:
    rA = rH - rI;
    break;
```

上記の switch ブロック全体は、すべてのプレディケートが事前に計算されていれば、プレディケーションを使って 1 サイクルで実行することができる。rW が 1、2、または 3 に等しく、それぞれの場合に p1、p2、または p3 が真になるものとする。

```
(p1)add  rA=rB,rC
(p2)add  rA=rE,rF
(p3)sub  rA=rH,rI ;;
```

このようなプレディケーションの機能がなければ、これらの値をまとめるためには多数の分岐または条件付き移動操作を行わなくてはならない。

IA-64では、そのクロック内でターゲット・レジスタに書き込みを行う命令のうちの1つだけがtrueのプレディケートを持つという条件が満たされる限り、同じクロック内で複数の命令が同じレジスタをターゲットとすることができる。11.3.1項で説明したように、プレディケート・レジスタの書き込みに関しても似たような機能がある。

11.3.3.2 レジスタ使用の削減

プレディケーションを使用すると、2つの異なる計算で同じレジスタを使用できることがある。このテクニックは、複数の書き込みで同じレジスタに値をストアできるようにするテクニックに似ているが、こちらの方はクリティカル・パスの問題というよりもレジスタ割り当ての最適化である。

if変換を行った後には、命令のシーケンスに相補的なプレディケートが付けられていることがよくある。次の人工的なシーケンスは、`p1`と`p2`によってプレディケートが付けられた命令を示している。コンパイラには`p1`と`p2`が補数であることがわかっている。

```
(p1)add  r1=r2,r3
(p2)sub  r5=r4,r56
(p1)ld8  r7=[r2]
(p2)ld8  r9=[r6] ;;
(p1)a use of r1
(p2)a use of r5
(p1)a use of r7
(p2)a use of r9
```

レジスタ`r1`、`r5`、`r7`、および`r9`がコンパイラ・テンポラリに使用されるものとし、これらはいずれも次に使用されるまでライブであるとすると、上のコード・セグメントは次のように書き換えることができる。

```
(p1)add  r1=r2,r3
(p2)sub  r1=r4,r56  // Reuse r1
(p1)ld8  r7=[r2]
(p2)ld8  r7=[r6] ;;// Reuse r7
(p1)a use of r1
(p2)a use of r1
(p1)a use of r7
(p2)a use of r7
```

この新しいシーケンスでは、使用されるレジスタの数が2つ減っている。IA-64が128個のレジスタを用意していることを思えば、これは大したことではないように思えるかもしれないが、レジスタの使用を減らすことで、命令レベルの並列処理が多用されているコードによく見られる、プログラムおよびレジスタ・スタック・エンジンのスピルとフィルを減らすことができる。

11.3.4 命令ストリームのフェッチの改善

命令は、分岐が発生せずに大きなブロック単位で実行されるときに、パイプライン内を最も効率的に流れる。命令ポインタを変更する必要が生じると、ターゲット予測が行われているために、あるいはパイプラインの後の方までターゲット・アドレスが計算されないために、ハードウェアによって、パイプラインにバブルを挿入しなければならない場合がある。

プレディケーションを使ってコントロール・フローの変更の回数を減らすことで、フェッチの効率は一般に改善される。プレディケーションによって命令キャッシュの効率が下がる唯一のケースは、後からプレディケートが外される命令が多数フェッチされる場合である。このような状況では、必要な結果を計算しない命令のために命令キャッシュのスペースが消費されてしまう。

11.3.4.1 命令ストリームのアラインメント

多くのプロセッサでは、プログラムが新しい位置に分岐すると、命令キャッシュ・ライン上で命令のフェッチが実行される。分岐のターゲットがキャッシュ・ライン境界で始まっていない場合、そのターゲットからのフェッチによってキャッシュ・ライン全体を取り出せない可能性が高い。この問題は、プログラマが、1 バンドルを越える命令グループをキャッシュ・ライン境界をまたがないように配置することによって防ぐことができる。しかし、すべてのラベルに埋め込みを行うと、コード・サイズが許容不可能なほどに増える可能性がある。より現実的なアプローチとして、ループの先頭と頻繁に実行される基本ブロックだけに対して、最初の命令グループが 1 バンドルを越える場合にアラインメントを行う。つまり、ラベル L において次の両方の条件が満たされている場合には、L がキャッシュ・ライン境界に合うように、その前の命令グループに埋め込みを行うことが勧められる。

- キャッシュ・ラインの外からそのラベルに分岐されることが多い。例としては、ループの先頭や、頻繁に実行される else 句がある。
- ラベル L から始まる命令グループが、1 バンドルを越えている。

例として、次のセグメントのラベル L のコードがキャッシュ・アライメントされておらず、2 つのバンドルの間にキャッシュ・ライン境界が存在するものとする。プログラムが L に分岐すると、リソースの予約超過やストップがないにもかかわらず、第 3 の add 命令の後で実行が分割される可能性がある。

```
L:
{ .mii
    add     r1=r2,r3
    add     r4=r5,r6
    add     r7=r8,r9
}
{ .mfb
    ld8     r14=[r56] ;;
    nop.f
    nop.b
}
```

一方、L が偶数バンドル上にアライメントされていれば、L の 4 つの命令すべてを 1 サイクルで発行することができる。

11.4 分岐とプリフェッチ・ヒント

分岐とプリフェッチ・ヒントは、コンパイラまたはアセンブリによる記述がハードウェアに対して追加情報を提供するための手段として、アーキテクチャ内に定義されているものである。コンパイラは、ハードウェアと比べると、より多くの時間を使用して、命令をより広い範囲 (ソースを含む) で検討し、より多くの分析を行う。これらの情報をプロセッサに伝達することで、icache アクセスと分岐予測に関連するペナルティを減らすことができる。

IA-64 アーキテクチャは 2 つのタイプの分岐関連のヒントを定義している。分岐予測ヒントと命令プリフェッチ・ヒントである。分岐予測ヒントでは、コンパイラは特定の分岐を動的に予測するために使用するリソースを推奨することができる (リソースが存在する場合)。プリフェッチ・ヒントでは、コンパイラはデマンド Icache ミスを減らすためにプリフェッチすべきコード部分を指示することができる。

ヒントは、分岐 (br) 命令にコンプリータを付けて指定され、分岐レジスタ (実際のニーモニックは mov br=xx なので、このテキストでは mov2br と略す) に移動する。分岐命令上のヒントは最も簡単に使用できる。命令はすでに存在し、ヒントのコンプリータを指定するだけでよいからである。mov2br 命令は間接分岐に使用される。これらのヒントの具体的な解釈方法はプロセッサによって異なるが、ヒントを指定された場合の一般的な動作は、異なるプロセッサ世代間でも似たようなものになることが予想される。

後からバイナリ書き換えツールを使って、分岐のヒント・フィールドを書き換えることも可能である。これは、プログラムの正確さを損なわずに、静的に、またはプロファイル・データに基づいて実行時に行うことができる。このテクニックを使うと、IA-64 の静的ヒントを、コンパイル時に、またはバイナリの当初の配布時にはよくわかっていなかった使用パターンに応じて修正することができる。

11.5 要約

本章では、プレディケーション、分岐アーキテクチャ、マルチウェイ分岐、並列比較、命令ストリームのアラインメント、および分岐ヒントを含む幅広いトピックを扱った。これらのトピックは、これまでの章でも取り上げてきたが、個々の機能間の相互作用は相互の影響を知ることでよく理解できる。

プレディケーションと、そのスケジューリング範囲の編成との相互作用は、IA-64 のパフォーマンスに大きな影響を与える。残念ながら、この種のコンパイラ・アルゴリズムについての議論は、本書の範囲をはるかに超えている。

ソフトウェアによるパイプライン化とループのサポート 12

12.1 概要

IA-64 は、レジスタ・ローテーション、特殊なループ分岐、アプリケーション・レジスタによって、ループをパイプライン化するための機能を豊富に提供している。これらの機能とプレディケーションおよびスペキュレーションとを組み合わせることで、ソフトウェアによるパイプライン化が可能なループに対してコードの拡大、パスの長さ、および分岐予測の失敗を減らすことができる。

本章の冒頭では、ループに関する基本的な用語と命令を紹介し、アーキテクチャのサポートがない状態でループの最適化を行おうとしたときに生じる問題について説明する。その後、IA-64 固有のループ・サポート機能を紹介する。さらに、IA-64 の機能を使った、さまざまなタイプのループのプログラミングと最適化について説明する。

12.2 ループ関連の用語と基本的なループ・サポート

ループは、カウント指定ループと while ループの 2 つのタイプに分類することができる。カウント指定ループでは、ループ条件はループ・カウンタの値に基づいており、ループを開始する前にトリップ・カウントを計算することができる。while ループでは、ループ条件はより一般的な計算式であり (単純なカウントではない)、トリップ・カウントは不明である。IA-64 はどちらのタイプも直接にサポートしている。

IA-64 は、特殊なカウント指定ループ分岐 (`br.cloop` 命令) とループ・カウント・アプリケーション・レジスタ (`LC`) を提供することで、従来のカウント指定ループのパフォーマンスを改善している。`br.cloop` 命令は分岐プレディケートを持たない。その代わりに、分岐の決定は `LC` レジスタの値に基づいて行われる。`LC` レジスタがゼロよりも大きければ、レジスタの値がデクリメントされ、`br.cloop` 分岐が行われる。

12.3 ループの最適化

多くのループでは、1 回の繰り返しの中に、実行レイテンシを隠し、機能ユニットをフルに活用できるだけの十分な数の独立した命令が存在しない。たとえば、次のループ本体には ILP はほとんどない。

```
L1: ld4     r4 = [r5],4 ;; // Cycle 0  load postinc 4
```

```
    add     r7 = r4,r9 ;;   // Cycle 2
    st4     [r6] = r7,4     // Cycle 3  store postinc 4
    br.cloop L1 ;;          // Cycle 3
```

このコードでは、繰り返し X の命令のすべてが、繰り返し X+1 が開始される前に実行される。繰り返し X のストアと繰り返し X+1 のロードが独立したメモリ参照であれば、独立した命令を繰り返し X+1 から繰り返し X に移動することで機能ユニットの利用効率を高めることができる。これにより、繰り返し X と繰り返し X+1 が実質的にオーバーラップすることになる。

本節では、ループの繰り返しをオーバーラップさせるための一般的な方法を 2 つ説明する。どちらも、従来のアーキテクチャ上ではコードが拡大することになる。コードの拡大の問題は、本章で後に説明する IA-64 のループ・サポート機能によって対処される。これ以降の説明では、上記のループを例として使用する。

12.3.1 ループのアンロール

ループのアンロールとは、ループ本体の複数のコピーを作成し、同時にスケジューリングすることで、命令レベルでの並列性を改善するテクニックである。ループ本体の各コピーのレジスタには、不要な WAW と WAR のデータ依存関係を避けるために、異なる名前が与えられる。次のコードは、ページ 12-1 の例のループを 2 回展開し (オリジナルのループ本体のコピーが 2 つ得られる)、命令スケジューリングを行った結果である。メモリ・ポートは 2 つあり、ロードのサイクル・レイテンシは 2 であると仮定している。例を単純にするために、ループのトリップ・カウントが 2 の乗数である定数 N だとする。このため、ループ本体の最初のコピーの後に終了分岐は必要ない。

```
L1: ld4     r4 = [r5],4 ;;      // Cycle 0
    ld4     r14 = [r5],4 ;;     // Cycle 1
    add     r7 = r4,r9 ;;       // Cycle 2
    add     r17 = r14,r9        // Cycle 3
    st4     [r6] = r7,4 ;;      // Cycle 3
    st4     [r6] = r17,4        // Cycle 4
    br.cloop L1 ;;              // Cycle 4
```

上のコードは最大限の ILP を実現しているわけではない。2 つのロードは、どちらも r5 を使用し、更新するのでシリアル化されている。同じように、2 つのストアはどちらも r6 を使用し、更新する。繰り返しのたびに同じ量だけインクリメント (またはデクリメント) される変数は帰納変数と呼ばれる。1 つの帰納変数 r5(また同様に r6) は、次のコードに示すように 2 つのレジスタに展開できる。

```
    add     r15 = 4,r5
    add     r16 = 4,r6 ;;
L1: ld4     r4 = [r5],8         // Cycle 0
    ld4     r14 = [r15],8 ;;    // Cycle 0
    add     r7 = r4,r9          // Cycle 2
    add     r17 = r14,r9 ;;     // Cycle 2
    st4     [r6] r7,8           // Cycle 3
```

```
    st4     [r16] = r17,8       // Cycle 3
    br.cloop L1 ;;              // Cycle 3
```

ページ 12-1 のオリジナルのループと比べると、機能ユニットの利用効率は 2 倍になり、コード・サイズも 2 倍になっている。しかし、サイクル 1 では命令は発行されておらず、それ以降のサイクルでも機能ユニットは依然として限界までは使用されていない。ループをさらにアンロールすれば利用効率が高まるが、これにはコードの拡大という代償がある。次のループは 4 回アンロールされている (トリップ・カウントが 4 の倍数であるとする)。

```
    add     r15 = 4,r5
    add     r25 = 8,r5
    add     r35 = 12,r5
    add     r16 = 4,r6
    add     r26 = 8,r6
    add     r36 = 12,r6 ;;
L1: ld4     r4 = [r5],16        // Cycle 0
    ld4     r14 = [r15],16 ;;   // Cycle 0
    ld4     r24 = [r25],16      // Cycle 1
    ld4     r34 = [r35],16 ;;   // Cycle 1
    add     r7 = r4,r9          // Cycle 2
    add     r17 = r14,r9 ;;     // Cycle 2
    st4     [r6] = r7,16        // Cycle 3
    st4     [r16] = r17,16      // Cycle 3
    add     r27 = r24,r9        // Cycle 3
    add     r37 = r34,r9 ;;     // Cycle 3
    st4     [r26] = r27,16      // Cycle 4
    st4     [r36] = r37,16      // Cycle 4
    br.cloop L1 ;;              // Cycle 4
```

これで、サイクル 2 を除くすべてのサイクルで、2 つのメモリ・ポートが利用されるようになる。前のコード例のループでは 4 サイクルで 2 回の繰り返しが実行されたのに対し、このコード例では 5 サイクルで 4 回の繰り返しが実行される。

12.3.2 ソフトウェアによるパイプライン化

ソフトウェアによるパイプライン化とは、機能ユニットのハードウェアによるパイプライン化と同じような形で、ループの繰り返しをオーバーラップさせるテクニックである。各繰り返しは、それぞれゼロ個以上の命令を持つ複数のステージに分割される。次に、12-1 ページのループを、パイプライン化された 1 回の繰り返しとして概念的に表したものを示す。各ステージは 1 サイクルである。

```
stage 1:ld4  r4 = [r5],4
stage 2:---                 // empty stage
stage 3:add  r7 = r4,r9
stage 4:st4  [r6] = r7,4
```

次に、パイプライン化された 5 回の繰り返しの概念図を示す。

```
     1   2   3   4   5            Cycle
-------------------------------------------------------
    ld4                            X
        ld4                       X+1
    add     ld4                   X+2
    st4 add     ld4               X+3
        st4 add     ld4           X+4
            st4 add               X+5
                st4 add           X+6
                    st4           X+7
```

隣接する繰り返しの始点の間のサイクル数は、開始インターバル (II) と呼ばれる。上の例の II は 1 である。パイプライン化された繰り返しの各ステージは II サイクルの長さを持つ。本章の例のほとんどはモジュロ・スケジューリングを使用している。これは、II が一定で、ループのすべての繰り返しが同じスケジュールを持つ、ソフトウェアによるパイプライン処理の特殊な形式である。IA-64 のループ・サポート機能を使えば、モジュロ・スケジューリング以外のソフトウェア・パイプライン・アルゴリズムの方が有利なことが多いと思われる。このため、本章の例では、モジュロ・スケジューリングよりもソフトウェアによるパイプライン化の文脈で説明を行っている。

ソフトウェア・パイプライン・ループは、次のように、プロローグ、カーネル、およびエピローグの 3 つのフェーズからなる。

```
     1   2   3   4   5             Phase
-------------------------------------------------------
    ld4
        ld4                        Prolog
    add     ld4
-------------------------------------------------------
    st4 add     ld4                Kernel
        st4 add     ld4
---------------------------------------------------------
            st4 add
                st4 add            Epilog
                    st4
```

プロローグ・フェーズでは、II サイクルごとに (上の例では毎サイクル)、パイプラインをフィルするために新しいループの繰り返しが開始される。プロローグの最初のサイクルでは、最初の繰り返しのステージ 1 が実行される。第 2 のサイクルでは、2 回目の繰り返しのステージ 1 と、1 回目の繰り返しのステージ 2 が実行される (以下同様)。カーネル・フェーズが開始される時点では、パイプラインはいっぱいになっている。この段階では、4 回目の繰り返しのステージ 1、3 回目の繰り返しのステージ 2、2 回目の繰り返しのステージ 3、および 1 回目の繰り返しのステージ 4 が実行される。カーネル・フェーズの中では、II サイクルごとに新しいループが開始され、ループ 1 つが完了する。エピローグ・フェーズでは、新しい繰り返しは開始されないが、進行中の繰り返しが完了し、パイプラインの内容が減っていく。上の例では、繰り返し 3-5 がエピローグ・フェーズで完了する。

ソフトウェア・パイプラインは、オリジナルのソース・コード・ループとは大きく異なるループとしてコーディングされる。ループとループの繰り返しを論じる際の混乱を避けるために、オリジナル・ソース・コードのループについてはソース・ループとソース繰り返しという用語を使用し、ソフトウェア・パイプラインを実現するループについては*カーネル・ループ*と*カーネル繰り返し*という用語を使用することにする。

上の例では、2 回目のソース繰り返しのロードは、1 回目のロードの結果が使用される前に発行される。このように、多くのケースでは、既存のライブな値の上書きを避けるために、ループの隣接する繰り返しのロードでは異なるレジスタをターゲットとしなくてはならない。従来のアーキテクチャでは、このためにはカーネル・ループのアンロールと、ソフトウェアによるレジスタのリネームが必要で、このためにコードの拡大が生じていた。さらに、従来のアーキテクチャでは、プロローグ、カーネル、およびエピローグの各フェーズに対して独立したコード・ブロックが生成されるため、さらにコードが拡大していた。

12.4 IA-64 のループ・サポート機能

従来のアーキテクチャでは、ループの最適化 (ソフトウェアによるパイプライン化やループのアンロールなど) から生じるコードの拡大のために、命令キャッシュ・ミスの数が増え、全体的なパフォーマンスが低下することがある。IA-64 のループ・サポート機能では、一部のループをコードの拡大なしにソフトウェアによってパイプライン化することができる。レジスタをローテートさせることによって、ループのアンロールとソフトウェアによるレジスタのリネームの必要性を軽減するリネーム・メカニズムが可能になる。特殊なソフトウェア・パイプライン・ループ分岐でレジスタのローテーションをサポートし、プレディケートと組み合わせることで、プロローグおよびエピローグ・フェーズ用に独立したコード・ブロックを生成する必要性が軽減される。

12.4.1 レジスタ・ローテーション

レジスタ・ローテーションは、CFM に含まれているローテート・レジスタ・ベース (rrb) レジスタの値にレジスタ番号を加えることで、レジスタをリネームする。rrb レジスタは、各カーネル繰り返しの終わりに一部の特殊なソフトウェア・パイプライン・ループ分岐が実行されたときにデクリメントされる。rrb レジスタがデクリメントされると、レジスタ X の値がレジスタ X+1 に移動したように見える。X が最も大きい番号のローテート・レジスタの場合には、その値は最も小さい番号のローテート・レジスタにラップされる。

プレディケートおよび浮動小数点のレジスタ・ファイルの一定サイズのエリア (`p16` ～ `p63` と `f32` ～ `f127`)、および汎用レジスタ・ファイルのプログラミング可能なサイズのエリアがローテートするように定義される。汎用レジスタ・ファイルの中のローテートするエリアのサイズは、`alloc` 命令の即値によって決定され、ゼロまたは 8 の倍数でなくてはならない。上限は 96 レジスタである。汎用レジスタ・ファイルの中の最も小さい番号のローテート・レジスタは `r32` である。rrb は 3 つのローテート・レジスタ・ファイルのそ

れぞれについて用意されている。汎用レジスタには CFM.rrb.gr、浮動小数点レジスタには CFM.rrb.fr、プレディケート・レジスタには CFM.rrb.pr である。ソフトウェア・パイプライン・ループ分岐はすべての rrb レジスタを同時にデクリメントする。

次に、レジスタ・ローテーションの例を示す。swp_branch 擬似命令は、ソフトウェア・パイプライン・ループ分岐を表している。

```
L1: ld4     r35 = [r4],4        // post increment by 4
    st4     [r5] = r37,4        // post increment by 4
    swp_branch  L1 ;;
```

ロード命令で r35 に書き込む値は、2 回のカーネル繰り返し (および 2 回のローテーション) の後のストア命令によって、r37 として読み出される。その間にも、2 つのインスタンスのロードがさらに実行される。レジスタ・ローテーションによって、これらのインスタンスはその結果を異なるレジスタに書き込むため、ストア命令で必要となる値が変更されることはない。

プレディケート・レジスタのローテーションには 2 つの目的がある。1 つは、まだ必要なプレディケート値の上書きを防ぐことである。2 つ目の目的は、パイプラインのフィルとドレインを制御することである。このために、プログラマはソフトウェア・パイプラインの各ステージにプレディケートを割り当てて、そのステージに含まれる命令の実行を制御する。このプレディケートはステージ・プレディケートと呼ばれる。カウント指定ループでは、アーキテクチャ上、p16 は第 1 ステージのプレディケートとして、p17 は第 2 ステージのプレディケートとして定義されている (以下同様)。次に、12-1 ページのカウント指定ループの例の、パイプライン化されたソース繰り返しの概念図を示す。各ステージは 1 サイクルの長さであり、それぞれのステージ・プレディケートも示している。

```
stage 1:(p16)  ld4 r4 = [r5],4
stage 2:(p17)  ---              // empty stage
stage 3:(p18)  add r7 = r4,r9
stage 4:(p19)  st4 [r6] = r7,4
```

レジスタ・ローテーションは各ステージの終わりに実行される (カーネル・ループの中でソフトウェア・パイプライン・ループ分岐が実行されるとき)。このため、p16 に 1 が書き込まれると第 1 ステージが有効となり、第 1 ステージの終わりにはこれが p17 にローテートされて、同じソース繰り返しの第 2 ステージが実行できるようになる。p16 に 1 が書き込まれると、すべてのステージで新しいソース繰り返しがシーケンシャルに可能になる。次の節で説明するように、この動作により、プロローグ、カーネル、およびエピローグの各フェーズで、パイプライン化されたループのステージの実行を許可または禁止することができる。

intel®

12.4.2 ローテート・プレディケートの初期化に関する注意事項

本章では、ローテート・プレディケートの初期化に `mov pr.rot = immed` 命令を使用している。この命令は CFM.rrb.pr の値を無視する。このため、本章の例は、`mov pr.rot = immed` を使ってプレディケート・レジスタが初期化される前に、CFM.rrb.pr が常にゼロになっているという前提に立って書かれている。

12.4.3 ソフトウェア・パイプライン・ループ分岐

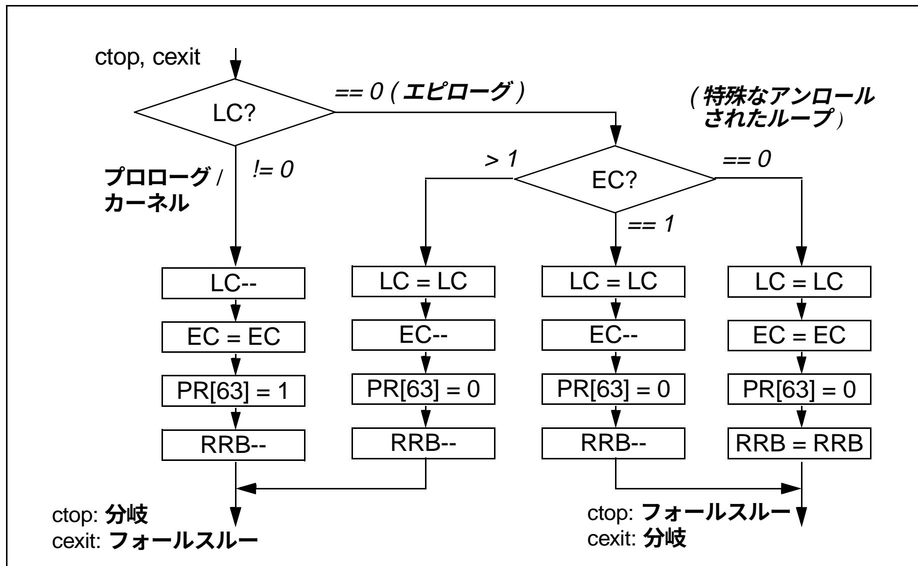
特殊なソフトウェア・パイプライン・ループ分岐により、コンパイラは、レジスタ・ローテーションをサポートし、プロローグおよびエピローグ・フェーズのソフトウェア・パイプラインのフィルとドレインを制御することで、ソフトウェア・パイプライン・ループに対してきわめてコンパクトなコードを生成することができる。一般に、ソフトウェア・パイプライン・ループ分岐が実行されるたびに、以下のアクションが実行される。

1. カーネル・ループの実行を続けるかどうかが決定される。
2. `p16` が、ソフトウェア・パイプラインのステージの実行を制御する値に設定される
   (分岐によって `p63` が書き込まれ、ローテーションの後にこの値は `p16` に含まれている)。
3. レジスタがローテートされる (rrb レジスタがデクリメントされる)。
4. ループ・カウント (LC) またはエピローグ・カウント (EC) アプリケーション・レジスタが選択的にデクリメントされる。

ソフトウェア・パイプライン・ループ分岐には、カウント指定ループと while ループの 2 つのタイプがある。

12.4.3.1 カウント指定ループ分岐

次の図に、カウント指定ループ・タイプのフローチャートを示す。



プロローグおよびカーネル・フェーズで、カーネル・ループの実行を続けるということは、新しいソース繰り返しが開始されるということである。新しいソース繰り返しが、まだパイプラインに残っている以前のソース繰り返しが使用しているレジスタを上書きしないように、レジスタ・ローテーションが実行されなくてはならない。新しいソース繰り返しのステージを有効にするために、p16 が 1 に設定される。残りのソース繰り返しのカウントを更新するために、LC がデクリメントされる。EC は変更されない。

エピローグ・フェーズで、カーネル・ループの実行を続けるということは、ソフトウェア・パイプラインがまだ完全にドレインされておらず、進行中のソース繰り返しの実行を続行しなくてはならないということである。残っているソース繰り返しは依然として結果を書き込んでおり、この結果を使用する命令はローテーションが行われることを期待しているので、レジスタ・ローテーションは続行されなくてはならない。これ以上の新しいソース繰り返しはないので p16 は 0 に設定され、存在しないソース繰り返しに対応する命令は無効にされなくてはならない。EC は、最後のソース繰り返しの中の、残っている実行ステージのカウントを含んでおり、エピローグの際にデクリメントされる。ほとんどのループでは、EC が 1 に等しい状態でソフトウェア・パイプライン化分岐ループが実行されるときには、パイプラインがドレインされており、ループを終了するという決定が行われたことを意味している。特殊なケースとして、展開されたソフトウェア・パイプライン・ループで、cexit タイプの分岐のターゲットが次の順序のバンドルに設定されている場合には、EC が 0 の状態でソフトウェア・パイプライン・ループ分岐が実行される。

カウント指定ループには 2 つのタイプのソフトウェア・パイプライン・ループ分岐がある。br.ctop は、カーネル・ループ実行を続行することが決定されたときに実行され、それ以外の場合は実行されない。これは、ループ実行の決定がループの末尾に置かれているときに使用される。br.cexit は、カーネル・ループ実行を続行することが決定されたときに実行されず、それ以外の場合は実行される。これは、ループ実行の決定がループの末尾以外の場所に置かれているときに使用される。

12.4.3.2 カウント指定ループの例

次に、12-1 ページのカウント指定ループの例の、パイプライン化された繰り返しの概念図を示す。開始インターバル (II) は 1 である。

```
stage 1:(p16)  ld4 r4 = [r5],4
stage 2:(p17)  ---             // empty stage
stage 3:(p18)  add r7 = r4,r9
stage 4:(p19)  st4 [r6] = r7,4
```

効率的なパイプラインを生成するために、コンパイラは、命令のレイテンシと使用可能な機能ユニットを考慮に入れなくてはならない。この例では、ロードのレイテンシは 2 であり、ロードと加算は 2 サイクルだけ離れている。次に示すパイプラインは、メモリ・ポートが 2 つあり、ループ・カウントが 200 であるという前提でコーディングされている。

注: このコードには GR のローテーションが追加されている (その前のコードにはなかった)。また、ポスト・インクリメントされる帰納変数は、レジスタ・ファイルの静的な部分に割り当てられなくてはならない。

```
     mov lc = 199              // LC =loop count - 1
     mov ec = 4                // EC =epilog stages + 1
     mov pr.rot = 1<<16 ;;     // PR16 = 1, rest = 0
L1:
(p16)ld4 r32  = [r5],4         // Cycle 0
(p18)add r35  = r34,r9         // Cycle 0
(p19)st4 [r6] = r36,4          // Cycle 0
     br.ctop L1 ;;             // Cycle 0
```

メモリ・ポートはフルに利用される。次の表は、このループの実行トレースを示している。

サイクル	ポート / 命令				br.ctop の前の状態					
	M	I	M	B	p16	p17	p18	p19	LC	EC
0	ld4			br.ctop	1	0	0	0	199	4
1	ld4			br.ctop	1	1	0	0	198	4
2	ld4	add		br.ctop	1	1	1	0	197	4
3	ld4	add	st4	br.ctop	1	1	1	1	196	4
…	…	…	…	…	…	…	…	…	…	…
100	ld4	add	st4	br.ctop	1	1	1	1	99	4
…	…	…	…	…	…	…	…	…	…	…
199	ld4	add	st4	br.ctop	1	1	1	1	0	4
200		add	st4	br.ctop	0	1	1	1	0	3
201		add	st4	br.ctop	0	0	1	1	0	2
202			st4	br.ctop	0	0	0	1	0	1
...					0	0	0	0	0	0

サイクル 3 でカーネル・フェーズに入り、カーネル・ループの第 4 の繰り返しは、それぞれ第 4、第 2、および第 1 のソース繰り返しの `ld4`、`add`、および `st4` を実行する。サイクル 200 までに 200 個のロードすべてが実行され、エピローグ・フェーズに入る。サイクル 202 で `br.ctop` が実行されるとき、`EC` は 1 になる。`EC` がデクリメントされ、レジスタが最後にローテートされ、実行はカーネル・ループから離れる。

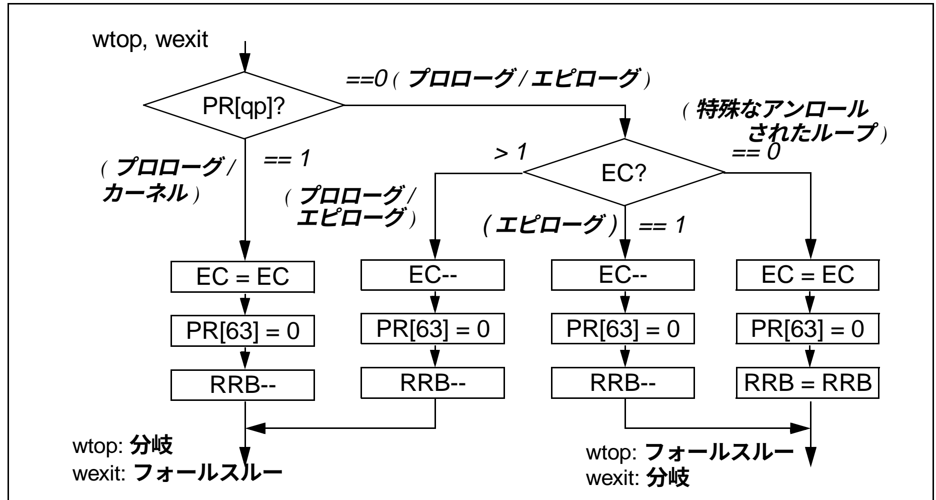
注: この最後のローテーションの後、`EC` とステージ・プレディケート (`p16` ～ `p19`) は 0 になっている。

ループ可変の変数は、可能な限りレジスタ・ファイルのローテートする部分に割り当てることが望ましい。これは、静的な部分のスペースをループ不変の変数のために空けておくためである。

注: ポスト・インクリメントされる帰納変数は、レジスタ・ファイルの静的な部分に割り当てられなくてはならない。

12.4.3.3 while ループ分岐

次の図は、while ループ・タイプの分岐のフローチャートを示している。



while ループ分岐の動作には、カウント指定ループ分岐と比べていくつかの違いがある。while ループ分岐は LC にアクセスしない。その代わりに、この分岐の動作は分岐プレディケートによって決定される。カーネルおよびエピローグ・フェーズでは、分岐プレディケートはそれぞれ 1 と 0 である。プロローグ・フェーズでは、分岐プレディケートは、while ループのプログラミングに使用されたスキームに応じて、0 または 1 のどちらかになる。また、p16 はローテーション後に必ずゼロに設定される。これらの違いは while ループの性質に関係しており、後の節で例とともに詳しく説明する。

12.4.4 用語の確認

以下に、これまでの節で登場した用語を示す。

開始インターバル (II)	ソフトウェア・パイプライン・ループの隣接するソース繰り返しの始点の間のサイクル数。パイプラインの各ステージは II サイクルの長さになる。
プロローグ	ソフトウェア・パイプライン・ループの第 1 フェーズ。パイプラインのフィルが行われる。
カーネル	ソフトウェア・パイプライン・ループの第 2 フェーズ。パイプラインはフルになっている。
エピローグ	ソフトウェア・パイプライン・ループの第 3 フェーズ。パイプラインのドレインが行われる。
ソース繰り返し	オリジナル・ソース・コードのループの繰り返し。
カーネル繰り返し	ソフトウェア・パイプラインをインプリメントするループの繰り返し。

レジスタ・ローテーション	ソフトウェアから検出可能なレジスタ・リネームの 1 形式。レジスタは、デクリメントされるローテート・レジスタ・ベースをもとにリネームされる。
帰納変数	1 回のソース繰り返しごとに同じ数だけインクリメント (またはデクリメント) される値。

12.5 IA-64 におけるループの最適化

レジスタ・ローテーション、プレディケーション、およびソフトウェア・パイプライン・ループ分岐により、コンパクトながら、きわめて並列性の高いコードの生成が可能となる。スペキュレーションにより、ソフトウェア・パイプライン・ループのスループットを制限する依存関係のバリアを取り除くことで、ループのパフォーマンスをさらに高めることができる。レジスタ・ローテーションによって、ソフトウェアによるレジスタのリネームを行うためにはカーネル・ループを展開しなくてはならないという必要条件が解消される。しかし、ケースによっては、ソフトウェアによるパイプライン化の前にソース・ループを転換したり、明示的にプロローグ・ブロックやエピローグ・ブロックを生成することで、パフォーマンスを高められることがある。以降では、ループの最適化について説明する。

12.5.1 while ループ

while ループのプログラミング・スキームは、ループの構造に依存する。本項では、ループ条件がループの終わりに計算される do-while ループについて説明する。最適化コンパイラは、条件判定をループの末尾に移動し、ループ内の分岐の数を減らすために、ループの前に条件判定のコピーを追加することで、while ループ (ループの先頭で条件が計算されるループ) を do-while ループに変換することがある。これ以降では、この種のループを単に while ループと呼ぶ。次に単純な while ループを示す。

```
L1: ld4     r4 = [r5],4 ;;      // Cycle 0
    st4     [r6] = r4,4         // Cycle 2
    cmp.ne  p1,p0 = r4,r0       // Cycle 2
(p1)br      L1 ;;               // Cycle 2
```

このループのパイプライン化された繰り返しの概念図を次に示す。開始インターバル (II) は 1 である。

```
stage 1:    ld4  r4   = [r5],4
stage 2:    ---                 // empty stage
stage 3:    st4  [r6]= r4,4
            cmp.ne.unc  p1,p0 = r4,r0
    (p1)    br   L1
```

次に、ロードとストアが独立したメモリ参照であると仮定したときの、4 つのオーバーラップするソース繰り返しの概念図を示す。ステージ 2 のストア、比較、および分岐の各命令は、擬似命令 scb によって表している。

```
     1   2   3   4     Cycle
-------------------------------------------------------
    ld4                 X
        ld4.s          X+1
    scb     ld4.s      X+2
        scb     ld4.s  X+3
            scb        X+4
                scb    X+5
```

第 2 のソース繰り返しのロード命令が、第 1 のソース繰り返しの比較命令と分岐命令の前に実行されていることに注目してほしい。つまり、このロード命令 (および r5 の更新) はスペキュレーティブである。ループ条件はサイクル X+2 まで計算されないが、リソースの利用効率を最大限に高めるためには、サイクル X+1 で第 2 のソース繰り返しを開始するのが望ましい。IA-64 のコントロール・スペキュレーションのサポートがなければ、第 2 のソース繰り返しはサイクル X+3 まで開始できない。

while ループのループ条件の計算はカウント指定ループとは大きく異なる。カウント指定ループでは、カウント指定ループ分岐を使って 1 サイクルでループ条件を計算することができる。これは、br.cshtop 命令が p16 を設定するときに行う。while ループでは、比較命令によってループ条件を計算し、ステージ・プレディケートを設定しなくてはならない。比較命令が含まれるステージよりも前のステージでは、比較命令によってこれらのステージの実行を完全に制御するのは不可能なため、パイプラインのスペキュレーティブ・ステージと呼ばれる。比較命令によって設定されるステージ・プレディケートは、パイプラインの最初の非スペキュレーティブ・ステージを制御するために (ローテーションの後に) 使用される。

次に 12-12 ページの while ループをパイプライン化したコードを示す。スペキュレーティブ・ロード・チェック命令が追加されている。

```
      mov   ec = 2
      mov   pr.rot = 1 << 16 ;;     // PR16 = 1, rest = 0
L1:
      ld4.s r32 = [r5],4            // Cycle 0
(p18) chk.s r34, recovery           // Cycle 0
(p18) cmp.ne p17,p0 = r34,r0        // Cycle 0
(p18) st4   [r6] = r34,4            // Cycle 0
(p17) br.wtop.sptk L1 ;;            // Cycle 0
L2:
```

カーネル・ループがこのようにプログラミングされている理由を理解するには、次の表のループの実行トレースを参考にするとよい (ソース繰り返しは 200 回と仮定される)。

ロード命令はスペキュレーティブなので、ステージ・プレディケートは割り当てられない。比較命令によって p17 が設定される。これは現在の繰り返しの分岐プレディケートであり、ローテーションの後は、次のソース繰り返しの最初の非スペキュレーティブ・ステージ (ステージ 3) のステージ・プレディケートとなる。プロローグ・ステージでは、比較命令はサイクル 2 まで

最初の有効な結果を生成することができない。プレディケートを初期化することにより、最初のソース繰り返しがサイクル 2 でステージ 2 に達するまで比較命令を無効にするパイプラインが与えられる。その時点で、比較命令により、パイプラインの非スペキュレーティブ・ステージを制御するためのステージ・プレディケートの生成が開始される。比較命令は条件付きであることに注意されたい。これが無非条件の場合には、p17 につねにゼロが書き込まれ、パイプラインは正しく起動されないことになる。

サイクル	ポート / 命令					br.wtop の前の状態			
	M	I	I	M	B	p16	p17	p18	EC
0	ld4.s				br.wtop	1	0	0	2
1	ld4.s				br.wtop	0	1	0	1
2	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
3	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
…	…	…		…	…	…	…	…	…
100	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
…	…	…		…	…	…	…	…	…
199	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
200	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
201	ld4.s	cmp	chk	st4	br.wtop	0	0	1	1
						0	0	0	0

プロローグの最初の 2 サイクルにおける br.wtop 命令の実行は、どのソース繰り返しにも対応していない。これは、最初の有効なループ条件が生成できるようになるまで、単にカーネル・ループを続行するためである。サイクル 1 では、分岐プレディケート p17 は 1 である。このプログラミング・スキームでは、br.wtop の分岐プレディケートは、第 1 のソース繰り返しの最後のスペキュレーティブ・ステージの間はつねに 1 である。それ以前のすべてのステージで、分岐プレディケートはゼロである。分岐プレディケートがゼロのとき、br.wtop は EC が 1 よりも大きい場合にのみカーネル・ループを続行する。また、EC のデクリメントも行う。EC は (エピローグ・ステージの数 + スペキュレーティブ・パイプライン・ステージの数) に初期化されなくてはならない。上の例では、これは 0 + 2 = 2 である。

サイクル 201 では、200 番目のソース繰り返しの比較が実行される。これは最後のソース繰り返しなので、比較の結果はゼロであり、p17 は変更されない。p16 から p17 にローテートされたゼロのために、br.wtop はループの出口までフォールスルーする。EC はデクリメントされ、レジスタは最後のローテートが行われる。

上の例では、エピローグ・ステージは存在しない。分岐プレディケートがゼロになると、カーネル・ループがただちに終了する。

12.5.2 プレディケート付き命令を含むループ

ソース・ループの中の、すでにプレディケートを持っている命令には、ステージ・プレディケートは割り当てられない。ただし、ループ本体の比較命令による制御は行われる。たとえば、次のループはプレディケート付きの命令を含んでいる。

```
L1: ldfs     f4 = [r5],4
    ldfs     f9 = [r8],4 ;;
    fcmp.ge.unc p1,p2 = f4,f9 ;;
(p1)stfs     [r9] = f4, 4
(p2)stfs     [r9] = f9, 4
    br.cloop L1 ;;
```

次に、開始インターバル (II) が 2 のパイプラインの例を示す。浮動小数点ロードのレイテンシが 9 サイクルだと仮定する。

```
stage 1:(p16)  ldfs f4 = [r5],4
        (p16)  ldfs f9 = [r8],4 ;;
               ---                 // empty cycle
stage 2-4:     ---                 // empty stages
stage 5:       ---                 // empty cycle
        (p20)  fcmp.ge.unc p1,p2 = f4,f9 ;;
stage 6:           ---             // empty cycle
        (p1)   stfs [r9] = f4, 4
        (p2)   stfs [r9] = f9, 4
```

次に、このパイプラインを実現するコードを示す。

```
        mov    lc = 199            // LC = loop count - 1
        mov    ec = 6              // EC = epilog stages + 1
        mov    pr.rot=1<<16;;      // PR16 = 1, rest = 0
L1:
(p16)   ldfs   f32 = [r5],4
(p16)   ldfs   f38 = [r8],4 ;;
(p32)   stfs   [r9] = f37, 4
(p20)   fcmp.ge.unc p31,p32 = f36,f42
(p33)   stfs   [r9] = f43, 4
L2:     br.ctop.sptk L1 ;;
```

12.5.3 複数の出口を持つループ

これまで取り上げてきたループの例は、いずれもループの末尾に 1 つの出口を持っていた。次のループは複数の出口を持っている。末尾にあるループ終了分岐の出口と、中程にある早めの出口である。

```
L1:     ld4    r4 = [r5],4 ;;
        ld4    r9 = [r4] ;;
        cmp.eq.unc p1,p0 = r9,r7
```

```
(p1)    br.cond  exit              // early exit
        add     r8 = -1,r8 ;;
        cmp.ge.unc p3,p0 = r8,r0
(p3)    br.cond L1 ;;
```

複数の出口を持つループでは、早めの出口から出たときにパイプラインが正しくドレインされるように特に注意する必要がある。上のループをパイプライン化したコードを生成する方法は 2 つあり、出口が 1 つのループに変換する方法と、複数の出口が明示的に存在する状態でパイプライン化を行う方法である。

12.5.3.1 複数の出口を持つループを 1 つの出口を持つループに変換する

最初の方法は、複数の出口を持つループを、1 つの出口を持つループに変換するというものである。ソース・ループでは、加算、第 2 の比較、および第 2 の分岐が第 1 の分岐によってガードされている。次に示すように、これらの命令の実行をガードするプレディケートを使用し、早めの出口の分岐をループの外に移動することによって、このループを 1 つの出口を持つループに変換することができる。

```
L1:     ld4     r4 = [r5],4 ;;
        ld4     r9 = [r4] ;;
        cmp.eq.unc p1,p2 = r9,r7
        add     r8 = -1,r8 ;;
(p2)    cmp.ge.unc p3,p0 = r8,r0
(p3)    br.cond L1 ;;
(p1)    br.cond exit       // early exit if p1 is 1
```

p3 の計算によって、ソース・ループが終了したかどうかを判定する。p3 がゼロであれば、ループは終了しており、どちらの出口が使われたのかを判定するために p1 が使用される。cmp.eq から add への従属関係によって開始インターバル (II) が制限されないように、加算はスペキュレーティブに実行される (p2 によってガードされていない)。r8 が早めの出口でライブになっていない、あるいは早めの出口のターゲットに補正コードが追加されているという前提がある。次に示すこのループのパイプラインは、ステージ・プレディケートの割り当てが行われているが、その他のローテート・レジスタの割り当ては行われていない。ステージ 4 の終わりの比較命令と分岐命令は、すでにソース・ループに修飾プレディケートを持っているので、ステージ・プレディケートは割り当てられない。

```
stage 1:   ld4.s r4 = [r5],4 ;;   // II = 2
           ---                    // empty cycle
stage 2:   ---                    // empty cycle
           ld4.s r9 = [r4] ;;
stage 3:   ---                    // empty stage
stage 4:
    (p19)  add  r8 = -1,r8
    (p19)  cmp.eq.unc p1,p2 = r9,r7 ;;
    (p2)   cmp.ge.unc p3,p0 = r8,r0
    (p3)   br.cond L1 ;;
```

次に、このパイプラインを実現する、chk 命令を含んだ完全なコードを示す。

```
        mov     ec = 3
        mov     pr.rot = 1 << 16 ;;    // PR16 = 1, rest = 0
L1:     ld4.s   r32 = [r5],4           // Cycle 0
(p19)   chk.s   r36, recovery          // Cycle 0
(p19)   add     r8  = -1,r8            // Cycle 0
(p19)   cmp.eq.unc p31,p32 = r36,r7 ;;// Cycle 0
        ld4.s   r34 = [r33]            // Cycle 1
(p32)   cmp.ge  p18,p0 = r8,r0         // Cycle 1
L2:
(p18)   br.wtop.sptk L1 ;;             // Cycle 1
(p32)   br.cond exit                   // early exit if p32 is 1
```

注: ループが終了すると、p31 の値を p32 にローテートする最後のローテーションが行われる。このため、p32 は早めの出口の分岐の分岐プレディケートとして使用される。

12.5.3.2 明示的な複数の出口を持ったままでのパイプライン化

第 2 の方法は、ループ内の最後の 3 つの命令を br.cloop 命令にまとめ、ループをパイプライン化するというものである。次に、このアプローチを使ったパイプラインを示す。

```
stage 1:ld4.s r4 = [r5],4 ;;      // II = 1
stage 4:ld4.s r9 = [r4] ;;
stage 6:cmp.eq.unc p1,p0 = r9,r7
    (p1)br.cond  exit
        br.cloop L1 ;;
```

このパイプラインには 5 つのスペキュレーティブ・ステージがある。これは、ステージ 6 で br.cond と br.cloop が実行されるまでは、新たなループ繰り返しを開始するための非スペキュレーティブな決定が行えないからである。次に、トリップ・カウントを 200 と仮定して、このパイプラインを実現するコードを示す。

```
        mov     lc = 204
        mov     ec = 1
        mov     pr.rot = 1 << 16 ;;    // PR16 = 1, rest = 0
L1:
        ld4.s   r32 = [r5],4           // Cycle 0
(p21)   chk.s   r38, recovery          // Cycle 0
(p21)   cmp.eq.unc p1,p0 = r38,r7      // Cycle 0
        ld4.s   r36 = [r35]            // Cycle 0
(p1)    br.cond exit                   // Cycle 0
L2:     br.ctop.sptk L1;               // Cycle 0
```

カーネル・ループが br.cond または br.ctop で終了した時点で、最後のソース繰り返しが完了する。これにより、EC は 1 に初期化され、早めの出口のために明示的なエピローグ・ブロックは生成されない。LC レジスタは、スペキュレーティブ・ステージが 5 つあるため、199 よりも 5 だけ大きい値に初期化される。br.ctop の最初の 5 回の実行は、br.cond に対して最初の有効な分岐プレディケートが生成されるまで、単にループを実行するためにだけ行われる。これらの実行の間でも、LC はデクリメントされるため、LC の初期値に 5 を加える必要がある。

第 2 の方法では開始インターバル (II) をより小さくすることができる。このパイプライン化されたコードは、LC が 199 に初期化され、EC が 6 に初期化された場合でも動作する。ただし、早めの出口が使用された場合に、LC がデクリメントされすぎるので、早めの出口のターゲットとして使用する場合には調整を加える必要がある。早めの出口が使用されるときにエピローグがある場合、そのエピローグは明示的に指定しなくてはならない。

12.5.4 ソフトウェアによるパイプライン化に関する考慮事項

ループに対してパイプライン化を行うのが望ましくないケースもある。ソフトウェアによるパイプライン化は繰り返しのスループットを向上させるが、1 回の繰り返しを完了するのに必要な時間を増やす可能性がある。その結果、非常に小さなトリップ・カウントを持つループでは、パイプライン化を行うとパフォーマンスが低下することがある。たとえば、次のループで考えてみる。

```
L1: ld4     r4 = [r5],4            // Cycle 0
    ld4     r7 = [r8],4 ;;         // Cycle 0
    st4     [r6] = r4,4            // Cycle 2
    st4     [r9] = r7,4            // Cycle 2
    br.cloop L1 ;;                 // Cycle 2
```

次に、開始インターバル (II) を 2 とするパイプラインの例を示す。

```
stage 1:ld4  r4 = [r5],4           // Cycle 0
        ld4  r7 = [r8],4 ;;        // Cycle 0
        ---                        // empty cycle
stage 2:---                        // empty cycle
        st4 [r6] = r4,4            // Cycle 3
        st4 [r9] = r7,4 ;;         // Cycle 3
```

ソース・ループでは、3 サイクルごとに 1 回の繰り返しが完了する。ソフトウェアによるパイプライン化が行われたループでは、最初の繰り返しを完了するために 4 サイクルが必要となる。それ以降の繰り返しは 2 サイクルで完了する。トリップ・カウントが 2 であれば、両方のコード例でのループの実行時間は 6 サイクルで等しくなる。ループの平均トリップ・カウントが 2 よりも小さければ、ループに対してソフトウェアでパイプライン化されたコードはソース・ループよりも遅くなることになる。

さらに、関数コールを含んでいる浮動小数点ループは、パイプライン化を行うのが望ましくないことがある。ループで使用する浮動小数点レジスタの数は、ループがパイプライン化されるまで不明である。パイプライン化を行うと、関数コールの前後で、コール側が保存する浮動小数点レジスタの保存と復元に必要な命令のための空のスロットが見つけにくくなる可能性がある。

12.5.5 ソフトウェアによるパイプライン化とアドバンスド・ロード

アドバンスド・ロードを使用すると、不変であると思われるコードをループから取り除き、ループのリソース要件を軽減することができる。また、アドバンスド・ロードにより、繰り返しの中でのクリティカル・パスを短縮し、開始インターバル (II) をより小さくすることができる。アドバンスド・ロードの詳細については、第 10 章「メモリ参照」を参照のこと。ただし、アドバンスド・ロードをレジスタ・ローテーションと組み合わせて使用するときには注意が必要である。ここでの説明では、ALAT が 32 個のエントリを持つものと仮定する。

12.5.5.1 キャパシティの制限

ローテート・レジスタをデスティネーションとするアドバンスド・ロードは、カーネル繰り返しのたびに異なる物理レジスタをターゲットとし、新しい ALAT エントリを割り当てる。たとえば、次の単純なループでは 32 回の繰り返しで 32 個の ALAT エントリを置き換える。

```
L1: (p16)   ld4.a r32 = [r8]
    (p47)   ld4.c r63 = [r8]
            br.ctop L1 ;;
```

不要な ALAT ミスを避けるためには、後のアドバンスド・ロードでチェック対象のエントリを置き換える前に、チェック・ロードまたはアドバンスド・ロード・チェックを実行する必要がある。上の単純なループでは、チェック・ロードがアドバンスド・ロードの 31 回の繰り返しの間に行われるため、不要な ALAT ミスは起こらない。次の例では、アドバンスド・ロードが、対応するチェック・ロードが実行される直前にエントリを置き換えるため、チェック・ロードのたびに ALAT ミスが発生する。

```
L1: (p16)   ld4.a r32 = [r8]
    (p48)   ld4.c r64 = [r8]
            br.ctop L1 ;;
```

12.5.5.2 ALAT 内での競合

アドバンスド・ロードを使って不変と思われるロードをループから取り除くと同時に、ループ内で別のロードを前方に移動させると、後者のロードがローテート・レジスタをターゲットとしている場合にはパフォーマンスが低下する。ローテート・レジスタをターゲットとするアドバンスド・ロードに

よって、ループ不変ロードの ALAT エントリが結局は無効になる。それ以降は、ループ不変ロードに対するチェック・ロードを実行するたびに、ALAT ミスが発生する。

ループ内の複数のアドバンスド・ロードがローテート・レジスタをターゲットとしている場合には、特定のアドバンスド・ロード X に対するチェック・ロードが、ロード X によって割り当てられたエントリを他のアドバンスド・ロードが無効にする前に実行されるように、レジスタの割り当てとレジスタの有効期間を制御しなくてはならない。たとえば、2 つのアドバンスド・ロードを持つ次のループでは、2 つのアドバンスド・ロードとチェック・ロードのペアが、同時に 32 個よりも多くのライブな ALAT エントリを作成することがないため、ALAT ミスなしでローテート・レジスタをターゲットとすることができる。

```
L1: (p16)   ld4.a r32 = [r8]
    (p31)   ld4.c r47 = [r8]
    (p16)   ld4.a r48 = [r9]
    (p31)   ld4.c r63 = [r9]
            br.ctop L1 ;;
```

ALAT ミスを避けるようにコードを配置することができない場合には、アドバンスド・ロードのデスティネーションにスタティック・レジスタを割り当て、ループをアンロールして、必要に応じてアドバンスド・ロードのデスティネーションを明示的にリネームするのがよい方法である。次の例は、ローテート・レジスタを使わないようにループをアンロールする方法を示している。このループの開始インターバル (II) は 1 で、チェック・ロードはアドバンスド・ロードの 1 サイクル (および 1 ローテーション) 後に実行される。

```
L1: (p16)   ld4.a r33 = [r8]
    (p17)   ld4.c r34 = [r8]
            br.ctop L1 ;;
```

ループを 2 回アンロールした場合には、ロードの宛先にスタティック・レジスタを割り当てることができる。

```
L1: (p16)   ld4.a r3 = [r8]
    (p17)   ld4.c r4 = [r8]
            br.cexit L2 ;;
    (p16)   ld4.a r4 = [r8]
    (p17)   ld4.c r3 = [r8]
            br.ctop L1 ;;
L2:         //
```

アドバンスド・ロードによって生成されない値については、ローテート・レジスタを使用することができる。アドバンスド・ロードを検討する際には、この命令キャッシュ・パフォーマンスに対するアンロールの効果をコストの一部として考慮に入れなくてはならない。

intel

12.5.6 ソフトウェアによるパイプライン化の前のループのアンロール

ケースによっては、ソフトウェアによるパイプライン化を行う前にループをアンロールすることで、より高いパフォーマンスを実現することができる。リソースによって制約されているループは、制限となるリソースがフルに利用されるようにアンロールすることで改善できる。次の例で、ターゲット・プロセッサにメモリ・ユニットが 2 つしかないと仮定すると、ループのパフォーマンスはメモリ・ユニットの数によって制約されることになる。

```
L1: ld4  r4 = [r5],4              // Cycle 0
    ld4  r9 = [r8],4 ;;           // Cycle 0
    add  r7 = r4,r9 ;;            // Cycle 2
    st4  [r6] = r7,4              // Cycle 3
    br.cloop L1 ;;                // Cycle 3
```

このループをパイプライン化したコードでは、メモリ命令が 3 つあるが、メモリ・ユニットは 2 つしかないため、少なくとも開始インターバル (II) が 2 でなくてはならない。ストアがロードとは独立であると仮定すると、ソフトウェアによるパイプライン化の前にループを 2 回アンロールすれば、新しいループで II を 3 にすることができる。これはオリジナルのソース・ループの II が実質的に 1.5 になったということである。次に、アンロールされたループのパイプラインの例を示す。

```
stage 1:(p16)  ld4    r4 = [r5],8         // odd iteration
        (p16)  ld4    r9 = [r8],8 ;;      // odd iteration
stage 2:(p16)  ld4    r14 = [r15],8       // even iteration
        (p16)  ld4    r19 = [r18],8 ;;    // even iteration
             // ---   empty cycle
stage 3:(p18)  add    r7 = r4,r9          // odd iteration
        (p17)  add    r17 = r14,r19;;     // even iteration
stage 4:     // ---    empty cycle
        (p19)  st4    [r6]  = r7,8        // odd iteration
        (p18)  st4    [r16] = r17,8 ;;    // even iteration
```

アンロールされたループは、ソース・ループ本体のコピーを 2 つ含んでいる。1 つは奇数回目のソース繰り返しに対応するもので、1 つは偶数回目のソース繰り返しに対応するものである。ステージ・プレディケートを割り当てるときには、この点を念頭に置く必要がある。p16 にシーケンシャルに 1 が割り当てれらることで、新しいソース繰り返しのすべてのステージが有効になる。上のパイプラインのステージ 1 では、奇数回目の繰り返しのステージ・プレディケートは p16 に入っている。偶数回目の繰り返しのステージ・プレディケートはまだ存在しない。上のパイプラインのステージ 2 では、奇数回目の繰り返しのステージ・プレディケートは p17 に入っており、偶数回目の繰り返しの新しいステージ・プレディケートは p16 に入っている。このように、同じパイプライン・ステージで、奇数回目の繰り返しのステージ・プレディケートがプレディケート・レジスタ X にあり、偶数回目の繰り返しのステージ・プレディケートがプレディケート・レジスタ X-1 にある。トリップ・カウントが未知であるとして、このパイプラインを実現するための擬似コードを次に示す。

```
        add     r15 = r5,4
        add     r18 = r8,4
        mov     lc = r2             // LC = loop count - 1
        mov     ec = 4              // EC = epilog stages + 1
        mov     pr.rot=1<<16;;      // PR16 = 1, rest = 0
L1:
(p16)   ld4     r33 = [r5],8        // Cycle 0 odd iteration
(p18)   add     r39 = r35,r38       // Cycle 0 odd iteration
(p17)   add     r38 = r34,r37       // Cycle 0 even iteration
(p16)   ld4     r36 = [r8],8        // Cycle 0 odd iteration
        br.cexit.spnt L3 ;;         // Cycle 0
(p16)   ld4     r33 = [r15],8       // Cycle 1 even iteration
(p16)   ld4     r36 = [r18],8 ;;    // Cycle 1 even iteration
(p19)   st4     [r6]  = r40,8       // Cycle 2 odd iteration
(p18)   st4     [r16] = r39,8       // Cycle 2 even iteration
L2:     br.ctop.sptk L1 ;;          // Cycle 2
L3:
```

ステージの長さが同じでないことに注意されたい。ステージ 1 と 3 の長さは 1 サイクルだが、ステージ 2 と 4 の長さは 2 サイクルである。また、エピローグ・フェーズの長さもトリップ・カウントによって変わる。トリップ・カウントが奇数ならば、エピローグ・ステージの数は 3 で、br.cexit の後から始まり、br.ctop で終わる。トリップ・カウントが偶数ならば、エピローグ・ステージの数は 2 で、br.ctop の後から始まり、br.ctop で終わる。EC は、エピローグ・ステージの数の上限を念頭に置いて設定されなくてはならない。この例では EC は 4 に初期化される。トリップ・カウントが偶数ならば、エピローグ・ステージが 1 つだけ余分に実行され、br.exit L3 が実行される。この余分なエピローグ・ステージで使用されるステージ・プレディケートはすべて 0 なので、何も実行されない。

偶数のトリップ・カウントでは、次に示すように、br.cexit 分岐のターゲットを次の順序のバンドルに設定し、EC を 3 に初期化することで、余分なエピローグ・ステージをなくすことができる。

```
        add     r15 = r5,4
        add     r18 = r8,4
        mov     lc = r2             // LC = loop count - 1
        mov     ec = 3              // EC = epilog stages + 1
        mov     pr.rot=1<<16;;      // PR16 = 1, rest = 0
L1:
(p16)   ld4     r33 = [r5],8        // Cycle 0 odd iteration
(p18)   add     r39 = r35,r38       // Cycle 0 odd iteration
(p17)   add     r38 = r34,r37       // Cycle 0 even iteration
(p16)   ld4     r36 = [r8],8        // Cycle 0 odd iteration
        br.cexit.spnt L4 ;;         // Cycle 0
L4:
(p16)   ld4     r33 = [r15],8       // Cycle 1 even iteration
(p16)   ld4     r36 = [r18],8 ;;    // Cycle 1 even iteration
(p19)   st4     [r6]  = r40,8       // Cycle 2 odd iteration
(p18)   st4     [r16] = r39,8       // Cycle 2 even iteration
```

```
L2:    br.ctop.sptk L1 ;;          // Cycle 2
L3:
```

ループ・トリップが偶数ならば、2 回のエピローグ・ステージが実行され、カーネル・ループは br.ctop で終了する。トリップ・カウントが奇数ならば、最初の 2 つのエピローグ・ステージが実行された後に、br.cexit 分岐が実行される。br.cexit 分岐のターゲットは次の順序のバンドル (L4) なので、カーネル・ループが br.ctop で終了する前に第 3 のエピローグ・ステージが実行される。この最適化により、トリップ・カウントが偶数のときにループの終わりのステージ 1 つ分が節約できるので、トリップ・カウントが小さいループに有効である。

アンロールには有利な点があるが、アンロールとソフトウェアによるパイプライン化を試みる前にいくつかの点を検討する必要がある。アンロールを行うと、パイプラインする側に渡されるループのトリップ・カウントが減るが、トリップ・カウントが小さいループの場合は、ループのパイプライン化を行うのが望ましくないことがある。また、アンロールによってコード・サイズが増えるので、命令キャッシュのパフォーマンスに悪影響を与えることがある。アンロールが最も有利なのは、小さいループに対してである。小さいループでは、リソースがフルに活用されていないためにパフォーマンスが低下している可能性が高く、またアンロールの命令キャッシュ・パフォーマンスに対する影響が大きなループと比べると小さいからである。

12.5.7 リダクションのサポート

次の例では、積の和がレジスタ f7 に累積されている。

```
    mov     f7 = 0 ;;            // initialize sum
L1: ldfs    f4 = [r5],4
    ldfs    f9 = [r8],4 ;;
    fma     f7 = f4,f9,f7 ;;     // accumulate
    br.cloop L1 ;;
```

パフォーマンスは fma 命令のレイテンシによって制約され、ここの例ではこれを 5 サイクルと仮定する。このループをパイプライン化したコードでは、fma のレイテンシが 5 なので、開始インターバル (II) が少なくとも 5 でなくてはならない。このループは、レジスタ・ローテーションを使用することで、次のように変換することができる。

注： このループはまだパイプライン化されていない。ソフトウェアによるパイプライン化を行う前に、レジスタ・ローテーションと特殊なループ分岐を使用しての最適化が行われている。

```
    mov     lc = 199             // LC = loop count - 1
    mov     ec = 1               // Not pipelined, so no epilog
    mov     f33 = 0              // initialize 5 sums
    mov     f34 = 0
    mov     f35 = 0
    mov     f36 = 0
    mov     f37 = 0 ;;
```

```
L1: ldfs    f4 = [r5],4
    ldfs    f9 = [r8],4 ;;
    fma     f32 = f4,f9,f37 ;; // accumulate
    br.ctop L1 ;;

    fadd    f10 = f33,f34      // add sums
    fadd    f11 = f35,f36 ;;
    fadd    f12 = f10,f11 ;;
    fadd    f7 = f12,f37
```

このループはレジスタ f33 〜 f37 に 5 つの独立した和を保持している。繰り返し X における fma 命令は、繰り返し X+5 の fma 命令によって使用される結果を生成する。繰り返し X から X+4 までは互いに独立しているので、開始インターバル (II) を 1 とすることができる。このループをパイプライン化したコードを次に示す。メモリ・ポートは 2 つで、浮動小数点ロードのレイテンシを 9 サイクルと仮定している。

```
    mov     lc = 199           // LC = loop count - 1
    mov     ec = 10            // EC = epilog stages + 1
    mov     pr.rot=1<<16       // PR16 = 1, rest = 0
    mov     f33 = 0            // initialize sums
    mov     f34 = 0
    mov     f35 = 0
    mov     f36 = 0
    mov     f37 = 0
L1:
(p16)ldfs   f50 = [r5],4       // Cycle 0
(p16)ldfs   f60 = [r8],4       // Cycle 0
(p25)fma    f41 = f59,f69,f46  // Cycle 0
    br.ctop.sptk L1 ;;         // Cycle 0
    fadd    f10 = f42,f43      // add sums
    fadd    f11 = f44,f45 ;;
    fadd    f12 = f10,f11 ;;
    fadd    f7 = f12,f46
```

12.5.8 明示的なプロローグとエピローグ

ケースによっては、正しいコードを得るために明示的なプロローグが必要なことがある。これは、スペキュレーティブ命令が、複数のソース繰り返しで有効となる値を生成する場合に生じる。次のループを例にとってみる。

```
    ld4     r3 = [r5] ;;
L1: ld4     r6 = [r8],4        // Cycle 0
    ld4     r5 = [r9],4 ;;     // Cycle 0
    add     r7 = r3,r6 ;;      // Cycle 2
    ld4     r3 = [r5]          // Cycle 3
    and     r10 = 3,r7;;       // Cycle 3
    cmp.ne p1,p0=r10,r11       // Cycle 4
(p1)br.cond L1 ;;              // Cycle 4
```

次に、このループのためのパイプラインの例を示す。

```
stage 1:       ld4.s   r6 = [r8],4    // II = 2
               ld4.s   r5 = [r9],4 ;;
               ---                    // empty cycle
stage 2:       ---                    // empty cycle
               ld4.s   r36 = [r5]
               add     r7 = r37,r6 ;;
stage 3:(p18)  and     r10 = 3,r7 ;;
        (p18)  cmp.ne  p1,p0 = r10,r11
        (p1)   br.wtop L1 ;;
```

注: 上のコードでは、ステージ 2 の ld4 命令および add 命令の順序が変更されている。レジスタ・ローテーションを使用して、add 命令から ld4 命令への WAR のレジスタ依存関係が解消されている。最初の 2 つのステージはスペキュレーティブである。次にパイプラインを実現するコードを示す。

```
    ld4     r36 = [r5]
    mov     ec = 2
    mov     pr.rot = 1 << 16 ;;// PR16 = 1, rest = 0
L1: ld4.s   r32 = [r8],4       // Cycle 0
    ld4.s   r34 = [r9],4       // Cycle 0
(p18)and    r40 = 3,r39 ;;     // Cycle 0
    ld4.s   r36 = [r35]        // Cycle 1
    add     r38 = r37,r33      // Cycle 1
(p18)chk.s r40, recovery       // Cycle 1
(p18)cmp.ne p17,p0 = r40,r11   // Cycle 1
(p17)br.wtop L1 ;;             // Cycle 1
```

このパイプライン化されたループの問題は、ループの前に r36 に書き込まれた値が、add 命令によって使用される前に上書きされるということである。この値は、最初のカーネル繰り返しでの r36 へのロード命令によって上書きされる。このロード命令はパイプラインの第 2 ステージにあるが、スペキュレーティブであり、ステージ・プレディケートを持たないために、最初のカーネル繰り返しの中では制御できない。この問題は、次に示すように、カーネルの 1 回の繰り返しを取り出し、そのコピーからパイプラインの最初のステージにない命令をすべて削除することによって解決できる。

注: 明示的なプロローグの中の命令のデスティネーションレジスタ番号は 1 だけ増やされている。これは、取り出されたカーネル繰り返しの最後にローテーションがないための調整である。

```
    ld4     r37 = [r5]
    mov     ec = 1
    mov     pr.rot = 1<<17 ;;  // PR17 = 1, rest = 0
    ld4     r33 = [r8],4
    ld4     r35 = [r9],4
L1: ld4.s   r32 = [r8],4       // Cycle 0
    ld4.s   r34 = [r9],4       // Cycle 0
(p18)and    r40 = 3,r39;;      // Cycle 0
```

```
    ld4.s   r36 = [r35]         // Cycle 1
    add     r38 = r37,r33       // Cycle 1
(p18)chk.s r40, recovery        // Cycle 1
(p18)cmp.ne p17,p0 = r40,r11    // Cycle 1
(p17)br.wtop L1 ;;              // Cycle 1
```

ケースによっては、プロローグ・フェーズやエピローグ・フェーズの全体または一部のために、独自のコード・ブロックを生成する方がパフォーマンスが高くなることがある。12-9 ページのパイプライン化されたカウント指定ループの実行トレースから明らかなように、プロローグおよびエピローグ・フェーズでは機能ユニットがフルに使用されていない。プロローグとエピローグの一部を取り出し、ループの前の方にあるコードとマージすることができる。次に、そのカウント指定ループに明示的なプロローグとエピローグを付けてパイプライン化したコードを示す。

```
    mov     lc = 196
    mov     ec = 1
prolog:
    ld4     r35 = [r5],4 ;;     // Cycle 0
    ld4     r34 = [r5],4 ;;     // Cycle 1
    ld4     r33 = [r5],4        // Cycle 2
    add     r36 = r35,r9 ;;     // Cycle 2
L1:
    ld4     r32 = [r5],4
    add     r35 = r34,r9
    st4     [r6] = r36,4
L2: br.ctop L1 ;;
epilog:
    add     r35  = r34,r9       // Cycle 0
    st4     [r6] = r36,4 ;;     // Cycle 0
    add     r34 = r33,r9        // Cycle 1
    st4     [r6] = r35,4 ;;     // Cycle 1
    st4     [r6] = r34,4        // Cycle 2
```

プロローグ全体 (カーネル・ループの最初の 3 回の繰り返し) とエピローグ (最後の 3 回の繰り返し) が取り出されている。取り出した命令の再スケジューリングは行っていない。ステージ・プレディケートは、プロローグおよびエピローグ・フェーズの制御には不要なので、命令から削除されている。プロローグからステージ・プレディケートを削除したことにより、プロローグ命令はローテートするプレディケートから独立するので、プロローグ・ステージ間でソフトウェア・パイプライン・ループ分岐が不要となる。これにより、プロローグ全体が、その前にある LC と EC の初期化から独立する。プロローグとエピローグのレジスタ番号は、これらのフェーズの中でステージ間でのローテーションがないということを考慮に入れて調整されている。

注: このコードは、ソース・ループのトリップ・カウントが 4 以上であると仮定している。コンパイル時にトリップ・カウントの最小値が不明な場合には、プロローグの前にトリップ・カウントを実行時にチェックする必要がある。

トリップ・カウントが 4 未満ならば、制御はオリジナル・ループのコピーへと分岐する。

このパイプライン化されたループが外側ループの中にネストされている場合には、さらに最適化を行うことができる。カーネル・ループが先頭に来て、その後に現在の外側ループの繰り返しのエピローグと、次の外側ループの繰り返しのプロローグが来るように外側ループをローテートさせる。さらに、外側ループの直前にもプロローグのコピーを追加する。

注: 前に示したカウント指定ループの実行トレースから、プロローグとエピローグの機能ユニットは相補的であることがわかるので、両者をうまくオーバーラップさせることが可能となる。

明示的なプロローグまたはエピローグを作成することの短所は、コードが拡大されるということである。

12.5.9 ループ内の余分なロードの除去

ループ最適化によって作られたコピー操作を削除するために、ループのアンロールが必要になることがある。次に示すのは、余分なロードを除去する例である。この例では、個々の繰り返しで 2 つの値がロードされるが、そのうちの 1 つは前回のソース繰り返しですでにロードされたものである。

```
    add     r8 = r5,4 ;;
L1: ld4     r4 = [r5],4         // a[i]
    ld4     r9 = [r8],4 ;;      // a[i+1]
    add     r7 = r4,r9 ;;
    st4     [r6] = r7,4
    br.cloop L1 ;;
```

ループの前に最初のロードのコピーを追加し、ロード命令をコピー (mov 命令) に変更することで、余分なロードを除去することができる。

```
    add     r8 = r5,4
    ld4     r9 = [r5],4;;       // a[i]
L1: mov     r4 = r9             // a[i] = previous a[i+1]
    ld4     r9 = [r8],4 ;;      // a[i+1]
    add     r7 = r4,r9 ;;
    st4     [r6] = r7,4
    br.cloop L1 ;;
```

従来のアーキテクチャでは、mov 命令を削除するにはループを 2 回アンロールする必要がある。ループから 1 つの命令を除去するために、コードアンロールを 2 回行う必要がある。IA-64 のレジスタ・ローテーション機能を使うと、ループをアンロールせずに mov 命令を除去できる。

```
    add     r8 = r5,4
    ld4     r33 = [r5],4;;      // a[i]
L1: ld4     r32 = [r8],4 ;;     // a[i+1]
    add     r7 = r33,r32 ;;
```

```
st4     [r6] = r7,4
br.ctop L1 ;;
```

12.6 要約

本章の例は、IA-64 の機能を使って、従来のアーキテクチャでは必要となるコードの拡大なしにループを最適化する方法を示している。レジスタ・ローテーション、プレディケーション、およびソフトウェア・パイプライン・ループ分岐は、いずれもこの機能に貢献する。コントロール・スペキュレーションによって、while ループの繰り返しのオーバーラップが増える。データ・スペキュレーションによって、曖昧さを除去できないロードとストアを含むループの繰り返しのオーバーラップが増える。

浮動小数点アプリケーション 13

13.1 概要

IA-64 浮動小数点アーキテクチャは ANSI/IEEE-754 標準に完全に準拠している。IA-64 は、積和混合命令、(スタティック領域およびローテート領域を持つ) 浮動小数点ラージ・レジスタ・ファイル、拡張範囲レジスタ・ファイル・データ表現、複数の独立した浮動小数点ステータス・フィールド、および高帯域幅のメモリ・アクセス命令といったパフォーマンス強化機能によって、コンパクトで高性能の浮動小数点アプリケーション・コードの作成を可能にしている。

本章の冒頭では、浮動小数点指向のアプリケーション・コードによく見られるパフォーマンスの制限をいくつか紹介する。その後、これらの制限に対処するための IA-64 の機能をコード例付きで説明する。本章の残りの部分では、いくつかのよく使用されるカーネルが IA-64 機能を使って行っている最適化に焦点を当てる。

13.2 FP アプリケーションのパフォーマンスの制限要因

浮動小数点アプリケーションには、ループが多用されるという特徴がある。ループの中には、通常の構造を持つデータに対して複雑な計算を行うもの、1 つの場所から別の場所に単にデータをコピーするもの、さらにはデータの計算と再配置を同時に行う収集 / 分散型の操作を行うものなどがある。以下の項では、パフォーマンスを制限するコード特性と、それらが各種のループに与える影響について説明する。

13.2.1 実行レイテンシ

ループは再帰関係を含んでいることがよくある。例として、Livermore Fortran Kernel スイートの 3 重対角行列の消去カーネルを取り上げる。

```
    DO 5 i = 2, N
   5    X[i] = Z[i] * (Y[i] - X[i-1])
```

`X[i]` と `X[i-1]` の間の依存関係によって、ループの繰り返し時間が減算と乗算のレイテンシの和に制限されている。ループをアンロールし、計算を複製することによって並列性を高めることはできるが、データ依存についての根本的な制限はなくならない。

ループのベクトル化が可能で、ソフトウェアによるパイプライン化が行える場合でも、コードを実行するハードウェアの実行レイテンシによってループの繰り返し時間が制限される場合がある。典型的な例として、単純なベクトル除算を次に示す。

```
    DO 1 I = 1, N
1       X[i] = Y[i] / Z[i]
```

今日の一般的なマイクロプロセッサには非パイプラインの浮動小数点ユニットが組み込まれているので、ループの繰り返し時間は除算のレイテンシ、すなわち数十クロックにもなることがある。

13.2.2 実行帯域幅

十分な ILP が存在し、利用できる場合、パフォーマンスは、使用可能な実行リソース、すなわちマシンの実行帯域幅によって制限される。BLAS3 ライブラリの密な行列乗算カーネルを例に取る。

```
    DO 1 i = 1, N
        DO 1 j = 1, P
            DO 1 k = 1, M
1               C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

ループの交換、ループのアンロール、およびアンロールとジャムといった一般的なテクニックを使って、内側ループの使用可能な ILP を増やすことができる。これを行うと、内側ループには、比較的少数のメモリ操作を含んだ、独立した浮動小数点計算が大量に含まれることになる。この場合、パフォーマンスは、(アキュムレータ `C[i, j]` と中間の計算結果を保持するためのレジスタが十分な数だけあるとして) もっぱらマシンの浮動小数点実行帯域幅によって制約される。

13.2.3 メモリ・レイテンシ

プロセッサとメモリの間のサイクル時間の差は、ほとんどのコードで一般的なメモリ・レイテンシの問題を引き起こすが、浮動小数点コードにはその影響を悪化させる特殊な条件がいくつか存在する。

そのような条件の 1 つが、間接アドレス指定の使用である。次に示すような、一般的な疎 (スパース) の行列ベクトル乗算コードにおける収集 / 分散コードはその良い例である。

```
    DO 1 ROW = 1, N
        R[ROW] = 0.0d0
        DO 1 I = ROWEND(ROW-1)+1, ROWEND(ROW)
1           R[ROW] = R[ROW] + A[I] * X[COL[I]]
```

COL[I] はベクトル X のインデックスに使用されているため、COL[I] のアクセスのメモリ・レイテンシが問題となる。X の要素へのアクセス、積の計算、および R[ROW] の積和の計算は、いずれも COL[I] のアクセスのメモリ・レイテンシに依存する。

浮動小数点コードに見られる、メモリ・レイテンシの影響を悪化させるもう 1 つの一般的な条件が、曖昧なメモリ依存関係の存在である。やはり Livermore Fortran Kernel スイートの不完全 Cholesky 共役勾配抽出カーネルを例に取る。

```
    II       = n
    IPNTP    = 0
222 IPNT     = IPNTP
    IPNTP    = IPNTP + II
    II       = II/2
    I        = IPNTP + 1
cdir$ ivdep
    DO 2 K = IPNT+2, IPNTP, 2
       I     = I+1
   2   X[I]= X[K] - V[K] * X[K-1] - V[K-1] * X[K+1]
    IF (II .GT. 1) GO TO 222
```

DO ループでは、インデックス I で、インデックス K、K+1、K-1 の X を使って X の更新が行われる。コンパイラにこれらのインデックスがオーバーラップするかどうかを判断させるのは難しいので、次の繰り返しの X[K]、X[K+1]、または X[K-1] のロードは、現在の繰り返しの X[I] のストアが行われるまではスケジューリングできない。これにより、これらのオペランドのアクセスのメモリ・レイテンシが問題となる。

13.2.4 メモリ帯域幅

一般に浮動小数点ループは、マシンが計算のオペランドを提供できる速度によって制限される。BLAS1 ライブラリの DAXPY カーネルはその典型的な例である。

```
    DO 1 I = 1, N
   1   Y[I] = Y[I] + A * X[I]
```

この計算では、個々の浮動小数点積和演算において、2 つのオペランド (X[I] と Y[I]) をロードし、1 つの結果をストアする (Y[I]) 必要がある。データ配列 (X と Y) がキャッシュ内になければ、このループのパフォーマンスは、ほとんどのマイクロプロセッサで、マシンの使用可能なメモリ帯域幅によって制限されることになる。

13.3 IA-64 の浮動小数点機能

本節では、13.2 節で説明したパフォーマンスの制限要因の影響を軽減する IA-64 の機能を、例を示しながら説明する。

13.3.1 ラージおよびワイド浮動小数点レジスタ・セット

マシンのサイクル時間が短縮されると、実行ユニットのサイクルのレイテンシは一般に増える。レイテンシが増えるにつれ、複数の演算が同時に行われるときのレジスタに対するプレッシャーも増大する。さらに、複数の実行ユニットが追加されると、さらに多くの命令が同時に実行されることになるので、レジスタのプレッシャーがさらに増大する。

IA-64 は、直接アドレス指定が可能な 128 個の浮動小数点レジスタを提供することで、データの再利用を可能にし、レジスタの数が不十分なときに起こるロード / ストア演算の数を減らしている。ロードとストアの回数が減ることで、計算はメモリ操作 (MOP) ではなく浮動小数点演算 (FLOP) によって制限されるようになるため、パフォーマンスが改善される可能性がある。次の密な行列の乗算コードを例に考える。

```
    DO 1 i = 1, N
        DO 1 j = 1, P
            DO 1 k = 1, M
   1            C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

内側ループ (k) では、個々の積和演算で 2 回のロードが必要である。つまり、MOP:FLOP の比率は 1:1 である。

```
L1: ldfd    f5  = [r5], 8       // Load A[i,k]
    ldfd    f6  = [r6], 8       // Load B[k,j]
    fma.d.s0 f7= f5, f6, f7     // *,+ to C[i,j]
    br.cloop L1
```

ここでは、オペランド (f5、f6) とアキュムレータ (f7) を保持するために 3 つのレジスタが必要である。j が変化するときにそれぞれの B[k、j] に対して A[i、k] が再利用され、i が変化するときにそれぞれの A[i,k] に対して B[k、j] が再利用されることに着目すると、この計算は次のように再構成することができる。

```
    DO 1 i = 1, N
        DO 1 j = 1, P
            DO 1 k = 1, M
                C[i  ,j  ] = C[i  ,j  ]
                            + A[i  ,k]*B[k,j  ]
                C[i+1,j  ] = C[i+1,j  ]
                            + A[i+1,k]*B[k,j  ]
                C[i  ,j+1] = C[i  ,j+1]
                            + A[i  ,k]*B[k,j+1]
   1            C[i+1,j+1] = C[i+1,j+1]
                            + A[i+1,k]*B[k,j+1]
```

これで、4 回のロードごとに 4 回の乗算と加算が実行できるため、MOP:FLOP の比率は 1:2 になる。ただし、このためには 8 個のレジスタが必要である (アキュムレータのために 4 つ、オペランドのために 4 つ)。

```
    add     r6 = r5, 8
    add     r8 = r7, 8
L1: ldfd    f5 = [r5], 16         // Load A[i,k]
    ldfd    f6 = [r6], 16         // Load A[i+1,k]
    ldfd    f7 = [r7], 16         // Load B[k,j]
    ldfd    f8 = [r8], 16         // Load B[k,j+1]
    fma.d.s0 f9 = f5, f7, f9      // *,+ on C[i,j]
    fma.d.s0 f10 = f5, f8, f10    // *,+ on C[i,j+1]
    fma.d.s0 f11 f6, f7, f11      // *,+ on C[i+1,j]
    fma.d.s0 f12 = f6, f7, f12    // *,+ on C[i+1,j+1]
    br.cloop L1
```

使用可能なレジスタは 128 個あるので、i と j の外側ループを 8 回アンロールすることで、16 個のオペランドをロードするだけで 64 個の乗算と加算を実行できるようになる。

浮動小数点レジスタ・ファイルは、スタティックな領域 (f0 ～ f31) とローテートする領域 (f32 ～ f127) の 2 つの領域に分割されている。レジスタ・ローテーションにより、コンパクトなカーネル・オンリーの、ソフトウェアでパイプライン化されたコードを作成するために必要な自動レジスタ・リネーム機能が可能になる。また、レジスタ・ローテーションにより、最もレイテンシが大きい操作よりも短い開始インターバルで、ソフトウェアでパイプライン化されたコードをスケジュールすることができる。たとえば、次の単純なベクトル加算ループを例に取る。

```
    DO 1 i = 1, N
   1    A[i] = B[i] + C[i]
```

基本的な内側ループは次のようになる。

```
L1: ldf     f5  = [r5], 8         // Load B[i]
    ldf     f6  = [r6], 8         // Load C[i]
    fadd    f7  = f5, f6          // Add operands
    stf     [r7]= f7, 8           // Store A[i]
    br.cloop L1
```

浮動小数点ロードの最小レイテンシが 9 クロックで、1 クロックあたり 2 つのメモリ操作を発行できるとすると、上のループはレジスタ・ローテーションがなければ少なくとも 6 回はアンロールしなくてはならない。

```
            add  r8  = r7, 8
L1: (p18)   stf  [r7] = f25, 16    // Cycle 17,26 ...
    (p18)   stf     [r8] = f26, 16     // Cycle 17,26 ...
    (p17)   fadd    f25  = f5, f15     // Cycle 8,17,26 ...
    (p16)   ldf     f5 = [r5], 8       // Cycle 0,9,18 ...
    (p16)   ldf     f15 = [r6], 8      // Cycle 0,9,18 ...
    (p17)   fadd    f26  = f6, f16 ;;  // Cycle 9,18,27 ...
    (p16)   ldf     f6  = [r5], 8      // Cycle 1,10,19 ...
    (p16)   ldf     f16  = [r6], 8     // Cycle 1,10,19 ...
    (p18)   stf     [r7] = f27, 16     // Cycle 20,29 ...
    (p18)   stf     [r8] = f28, 16     // Cycle 20,29 ...
```

```
(p17)  fadd   f27 = f7, f17 ;;    // Cycle 11,20 ...
(p16)  ldf    f7  = [r5], 8       // Cycle 3,12,21 ...
(p16)  ldf    f17  = [r6], 8      // Cycle 3,12,21 ...
(p17)  fadd   f28  = f8, f18 ;;   // Cycle 12,21 ...
(p16)  ldf    f8   = [r5], 8      // Cycle 4,13,22 ...
(p16)  ldf    f18  = [r6], 8      // Cycle 4,13,22 ...
(p18)  stf    [r7] = f29, 16      // Cycle 23,32 ...
(p18)  stf    [r8] = f30, 16      // Cycle 23,32 ...
(p16)  fadd   f29  = f9, f19 ;;   // Cycle 14,23 ...
(p16)  ldf    f9 = [r5], 8        // Cycle 6,15,24 ...
(p16)  ldf    f19  = [r6], 8      // Cycle 6,15,24 ...
(p16)  fadd   f30  = f10, f20 ;;  // Cycle 15,24 ...
(p16)  ldf    f10  = [r5], 8      // Cycle 7,16,25 ...
(p16)  ldf    f20  = [r6], 8      // Cycle 7,16,25 ...
       br.ctop L1 ;;
```

しかし、レジスタ・ローテーションを使用できれば、同じループに対し、アンロールなしで 2 クロックでスケジュールすることができる (2 回アンロールした場合には 1.5 クロック)。

```
L1: (p24)  stf    [r7] = f57, 8       // Cycle 15,17 ...
    (p21)  fadd   f57  = f37, f47     // Cycle 9,11,13 ...
    (p16)  ldf    f32  = [r5], 8      // Cycle 0,2,4,6 ...
    (p16)  ldf    f42  = [r6], 8      // Cycle 0,2,4,6 ...
           br.ctop L1 ;;
```

このように、モジュロ・スケジュールを行った後に (必要ならば) アンロールを行うと有利なことが多い。ソフトウェアによるパイプライン化の詳細と、この変換を使ってループを書き換える方法については、第 12 章を参照のこと。

13.3.1.1 FP の精度に関する注意

浮動小数点レジスタは 82 ビットの長さを持ち、17 ビットが指数範囲、64 ビットが仮数精度、1 ビットが符号ビットである。計算の際に、結果として得られる範囲および精度は、ユーザが選択した計算モデルによって決定される。計算モデルは、命令エンコーディングで静的に指定されるか、浮動小数点ステータス・レジスタの中の精度制御 (PC) ビットおよび最大範囲指数 (WRE) ビットを設定して動的に指定される。ユーザは適切な計算モデルを使用することで、計算における誤差の蓄積を最小限に抑えることができる。上の行列乗算の例では、積和の計算が最大限のレジスタ・ファイル範囲と精度で実行された場合、単精度数の入力に対して、(アキュムレータの中の) 結果は 64 ビットの精度と、最大 17 ビットの指数範囲で保持できる。(単精度の場合には 24 番目ではなく)64 番目の精度ビットで丸めが実行されるので、個々の積和演算ごとに蓄積される誤差は小さくなる。さらに、(単精度の場合には 8 ビットではなく)17 ビットの範囲があるため、オーバーフローやアンダーフローが発生せずに、大きな正または負の積をアキュムレータに加えることができる。この大きな範囲と精度によって、より正確な結果を提供するだけでなく、(誤差の上限によって指定される) 収束に達するまで実行する必要がある反復的な計算のパフォーマンスが改善されることも多い。

13.3.2 積和命令

IA-64 は、基本的な浮動小数点計算として積和混合命令 (fma) を定義している。これは多くの計算 (線型代数、系列展開など) の核をなしており、そのハードウェアにおけるレイテンシが、一般に個々の乗算演算 (丸め付き) のハードウェアと個々の加算演算 (丸め付き) のハードウェアのレイテンシの和よりも小さいためである。

ループの中で依存関係が存続し、その速度がピークの計算速度ではなく浮動小数点計算のレイテンシによって決定されることが多い計算ループでは、一般に積和演算が有利である。Livermore FOTRAN Kernel 9 - General Linear Recurrence Equations を例に取る。

```
    DO 191 k= 1,n
        B5(k+KB5I)= SA(k) + STB5 * SB(k)
        STB5= B5(k+KB5I) - STB5
 191CONTINUE
```

変数 B5(k+KB5I) の 2 つのステートメントの間には真のデータ依存関係があり、変数 STB5 にはループ内の依存関係があるため、ループの繰り返しごとのクロック数は完全に浮動小数点演算のレイテンシによってのみ決定される。fma 命令タイプの操作がなく、個々の積和のレイテンシが 5 クロックで、ロードが 8 サイクルだとすると、ループは次のようになる。

```
L1: (p16)  ldf     f32 = [r5], 8          // Load SA(k)
    (p16)  ldf     f42  = [r6], 8         // Load SB(k)
    (p17)  fmul    f5  = f7, f43;;        // tmp,Clk 0,15 ...
    (p17)  fadd    f6  = f33, f5 ;;       // B5,Clk 5,20 ...
    (p17)  stf     [r7] = f6, 8           // Store B5
    (p17)  fsub    f7  = f6, f7           // STB5,Clk 10,25 ..
           br.ctop L1 ;;
```

fma 命令を使用すると、操作のチェーンの全体的なレイテンシが減る。fma が 5 サイクルだとすると、ループの繰り返し速度は 10 クロックになる (上のループでは 15 クロック)。

```
L1: (p16)  ldf     f32  = [r5], 8         // Load SA(k)
    (p16)  ldf     f42 = [r6], 8          // Load SB(k)
    (p17)  fma     f6  = f7, f43, f33;;   // B5,Clk 0,10 ...
    (p17)  stf     [r7] = f6, 8           // Store B5
    (p17)  fsub    f7  = f6, f7           // STB5,Clk 5,15 ..
           br.ctop L1 ;;
```

また、混合積和演算は、計算のペアに対して丸め誤差が 1 回分しかないという利点を持っている。これは、大きな数値の小さな差を計算するときに便利である。

13.3.3 ソフトウェアによる除算 / 平方根のシーケンス

IA-64 で除算または平方根の演算を行うときには、ソフトウェア・ベースの演算シーケンスが使用される。このシーケンスでは、最初の推測値を取得し (frcpa/frsqrta 命令を使用)、誤差が結果の丸めに影響を与えないほど十分に小さくなるまでニュートン・ラフソン反復法を実行する。次に、レイテンシとスループットの点で最適化された、倍精度の除算と平方根のシーケンスの例を示す。

注: 精度が低くてよい場合には、平方根と除算のシーケンスはもっと少ない命令で完了する。

13.3.3.1 倍精度 - 除算

除算 (最大のスループット) (10 個の命令、8 グループ)	除算 (最小のレイテンシ) (13 個の命令、7 グループ)
frcpa.s0 f8,p6 = f6,f7 ;;	frcpa.s0 f8,p6 = f6,f7 ;;
(p6) fnma.s1 f9 = f7,f8,f1 ;;	(p6) fma.s1 f9 = f6,f8,f0
(p6) fma.s1 f8 = f9,f8,f8	(p6) fnma.s1 f10 = f7,f8,f1 ;;
(p6) fma.s1 f9 = f9,f9,f0 ;;	(p6) fma.s1 f9 = f10,f9,f9
(p6) fma.s1 f8 = f9 ,f8,f8	(p6) fma.s1 f11 = f10,f10,f0
(p6) fma.s1 f9 = f9,f9,f0 ;;	(p6) fma.s1 f8 = f10,f8,f8 ;;
(p6) fma.s1 f8 = f9,f8,f8 ;;	(p6) fma.s1 f9 = f11,f9,f9
(p6) fma.d.s1 f9 = f6,f8,f0 ;;	(p6) fma.s1 f10 = f11,f11,f0
(p6) fnma.d.s1 f6 = f7,f9,f6 ;;	(p6) fma.s1 f8 = f11,f8,f8 ;;
(p6) fma.d.s0 f8 = f6,f8,f9	(p6) fma.d.s1 f9 = f10,f9,f9
	(p6) fma.s1 f8 = f10,f8,f8 ;;
	(p6) fnma.d.s1 f6 = f7,f9,f6 ;;
	(p6) fma.d.s0 f8 = f6,f8,f9

intel®

13.3.3.2 倍精度 - 平方根

平方根 (最大のスループット) (15 個の命令、13 グループ)	平方根 (最小のレイテンシ) (17 個の命令、12 グループ)
`frsqrta.s0 f8,p6 = f6`	`frsqrta.s0 f8,p6 = f6`
`(p6) fma.s1 f10 = f7,f6,f0 ;;`	`(p6) fma.s1 f9 = f7,f6,f0 ;;`
`(p6) fma.s1 f9 = f8,f8,f0 ;;`	`(p6) fma.s1 f10 = f8,f8,f0 ;;`
`(p6) fnma.s1 f9 = f9,f10,f7 ;;`	`(p6) fnma.s1 f10 = f10,f9,f7 ;;`
`(p6) fma.s1 f8 = f9,f8,f8 ;;`	`(p6) fma.s1 f8 = f10,f8,f8 ;;`
`(p6) fma.s1 f9 = f8,f10,f0 ;;`	`(p6) fma.s1 f10 = f8,f9,f0 ;;`
`(p6) fnma.s1 f9 = f9,f8,f7 ;;`	`(p6) fnma.s1 f10 = f10,f8,f7 ;;`
`(p6) fma.s1 f8 = f9,f8,f8 ;;`	`(p6) fma.s1 f8 = f10,f8,f8 ;;`
`(p6) fma.s1 f9 = f8,f10,f0 ;;`	`(p6) fma.s1 f10 = f6,f8,f0`
`(p6) fnma.s1 f9 = f9,f8,f7 ;;`	`(p6) fma.s1 f9 = f8,f9,f0`
`(p6) fma.s1 f8 = f9,f8,f8 ;;`	`(p6) fma.s1 f11 = f7,f8,f0 ;;`
`(p6) fma.d.s1 f9 = f6,f8,f0`	`(p6) fnma.s1 f12 = f10,f10,f6`
`(p6) fma.s1 f8 = f7,f8,f0 ;;`	`(p6) fnma.s1 f7 = f9,f8,f7 ;;`
`(p6) fnma.s1 f6 = f9,f9,f6 ;;`	`(p6) fma.s1 f8 = f12,f11,f10`
`(p6) fma.d.s0 f8 = f6,f8,f9`	`(p6) fma.s1 f7 = f7,f11,f11 ;;`
	`(p6) fnma.s1 f6 = f8,f8,f6 ;;`
	`(p6) fma.d.s0 f8 = f6,f7,f8`

最初の命令 (`frcpa`) では、指定されたニュートン・ラフソン反復法を使って比率 `f6/f7` を得ることができる場合には、`f7` の逆数の近似値 (8 ビットまで有効) を与え、プレディケート (`p6`) を 1 に設定する。しかし、比率 `f6/f7` が特殊な値 (有限数 /0、有限数 / 無限大など) の場合には、`f6/f7` の最終結果は `f8` に与えられ、プレディケート (p6) はクリアされる。特定の境界条件 (オペランド値 (`f6` と `f7`) が単精度、倍精度、さらには拡張倍精度の範囲を大きく越えている場合) では、frcpa はソフトウェア・アシスト・フォルトを発生させ、ソフトウェア・ハンドラが比率 `f6/f7` を生成して、これを f8 に返し、プレディケート (p6) をクリアする。

これらのシーケンスでは、FPSR に用意されている複数のステータス・フィールドが使用される。S0 はメイン (アーキテクチャ) ステータス・フィールドで、最初の操作 (`frcpa`) によってフォルト (V、Z、D) を通知するために、また最後の操作によってトラップを通知するために書き込まれる。すべての中間操作の条件は S1 に書き込まれ、結果として無視される。このように、これらのシーケンスは、(`f8` に)IEEE 754 で定められた正しい結果を得るだけでなく、フラグも (S0 に) 仕様の要件に従って設定される。除算が、S2 をステータス・フィールドとして使用するスペキュレーティブな操作のチェーンの一部である場合には、これらのシーケンスで S0 を S2 に置き換えなくてはならない。この場合でも、S1 は、すべての除算シーケンスの中間操作 (S0、S2、または S3 をターゲットとするもの) によって使用される。これらのフラグはすべて破棄されるためである。

除算と平方根がベクトル化可能なループで使われる場合には、これらの演算をハードウェアではなくソフトウェアで行う方が一般に大幅に有利である。これらの演算は、一般にハードウェアではパイプライン化が不可能なのに対し、ソフトウェアによってパイプライン化を行って、全体的なパフォーマンスを高めることができるからである。

ソフトウェア・ベースの除算 / 平方根の計算のもう 1 つの大きな利点は、結果の精度をユーザが制御し、速度との妥協点を考慮できることである。これは、除算の精度が約 14 ビットで十分で、単精度または倍精度のときよりもシーケンスが短くなるグラフィックス用のコードでよく行われる。

13.3.4 計算モデル

IA-64 では、計算モデルをユーザが完全に制御できる。ユーザは、結果の精度と範囲、丸めモード、および IEEE トラップ応答を選択できる。計算モデルを適切に選択することで、より高い精度とより高いパフォーマンスを満たすコードを生成することができる。

レジスタ・ファイル形式は、単精度、倍精度、および拡張倍精度の 3 つのメモリ・データ型で同じである。すべての計算は (その内容のデータ型にかかわらず) レジスタ上で行われるので、異なる型のオペランドを簡単に組み合わせることができる。また、メモリ型からレジスタ・ファイル形式への変換はロード時に自動的に行われるので、形式変換のために余分な操作を行う必要はない。

C の構文の語彙も簡単にエミュレートすることができる。ロード命令によって、すべての入力オペランドが自動的にレジスタ・ファイル形式に変換される。これにより、レジスタ・ファイル形式に格納された異なる型のデータ・オペランドに対して操作を行い、命令エンコーディングで結果の精度を静的に指定することで、すべての中間結果を強制的に倍精度に変換できる。その後、最終的な結果を導き出す計算で、結果の精度と範囲を指定 (単精度と倍精度の場合は命令エンコーディングで静的に、拡張倍精度の場合はステータス・フィールド・ビットで動的に) すればよい。また、ステータス・フィールド・ビットにより、IA32 FP 計算形式 (範囲 = 拡張、精度 = 単精度 / 倍精度 / 拡張) に準拠することが可能である。

13.3.5 複数のステータス・フィールド

FPSR は、1 つのメイン (アーキテクチャ) ステータス・フィールドと、さらに 3 つの同一のステータス・フィールドに分割される。これらの追加のステータス・フィールドを使って、パフォーマンスを改善することができる。

まず第 1 に、(13.3.3 項で説明した) 除算と平方根のシーケンスは、最終的な結果がそうでなくても、中間結果のオーバーフロー / アンダーフローを引き起こしたり、不正確な結果を出したりする可能性のある操作を含んでいる。正しい IEEE 準拠のフラグ・ステータスを保持するためには、これらの計算のステータス・フラグを破棄する必要がある。これらのフラグの破棄には、追加のステータス・フィールドの 1 つ (通常はステータス・フィールド 1) を使用できる。

第 2 に、スペキュレーションを行う浮動小数点操作では、スペキュレーションされた操作がアーキテクチャ上の状態にコミットされるまで (コミットされる場合)、その操作のステータス・フラグがアーキテクチャ上のステータス・フラグとは別に保持されていなくてはならない。この目的のために、追加のステータス・フィールドの 1 つ (通常はステータス・フィールド 2 と、さらには 3) を使用することができる。

Livermore FORTRAN Kernel 16 - Monte Carlo Search を例に取る。

```
        DO 470 k= 1,n
           k2= k2+1
           j4= j2+k+k
           j5= ZONE(j4)
           IF( j5-n      ) 420,475,450
    415    IF( j5-n+II   ) 430,425,425
    420    IF( j5-n+LB   ) 435,415,415
    425    IF( PLAN(j5)-R) 445,480,440
    430    IF( PLAN(j5)-S) 445,480,440
    435    IF( PLAN(j5)-T) 445,480,440
    440    IF( ZONE(j4-1)) 455,485,470
    445    IF( ZONE(j4-1)) 470,485,455
    450    k3= k3+1
           IF( D(j5)-(D(j5-1)*(T-D(j5-2))**2
      ,        +(S-D(j5-3))**2
      ,           +(R-D(j5-4))**2)) 445,480,440
    455    m= m+1
           IF( m-ZONE(1) ) 465,465,460
    460    m= 1
    465    IF( i1-m) 410,480,410
    470    CONTINUE
    475    CONTINUE
    480    CONTINUE
    485    CONTINUE
```

プロファイリングの結果、ステートメント 450 の後の条件文が最も頻繁に実行されることがわかった。このため、415 〜 445 の条件が評価されている間に、条件文の中の計算をスペキュレーティブに実行するのが望ましい。これは、415 〜 445 の条件の結果、制御が 450 よりも後に移る場合に備えてである。

複数のステータス・フィールドが使用できるおかげで、ユーザは複数の計算環境を維持し、操作ごとにそれらの環境の中から動的にいずれか 1 つを選択することができる。この方法は、結果のインターバルを決定するために、個々のプリミティブ操作を 2 つの異なる丸めモードで計算しなければならないインターバル算術コードでよく使われる。

13.3.6 その他の機能

IA-64 では、さまざまな計算状況のパフォーマンスを強化するために、上記以外にもいくつかのアーキテクチャ上の構成要素を提供している。

13.3.6.1 オペランド・スクリーニング・サポート

一般に、計算の前のステップとして、オペランド・スクリーニングが必須または有効である。オペランドのスクリーニングを行って、それが有効な範囲にあることを確認したり (例 : 平方根の場合は正の有限の値またはゼロ、除算の場合は分母がゼロ以外)、近道を取る (計算の結果を事前に決定できる、または別の方法でより効率的に計算できる) ことができる。fclass 命令を使うと、入力オペランドを特定のクラスのセットの一部あるいはそうでないものとして分類することができる。次に、平方根の計算のための無効なオペランドをスクリーニングするコードを示す。

```
IF (A .LT. 0.0D0 .OR. A .GT. MAXREAL) THEN
    WRITE (*, "INVALID INPUT OPERAND")
ELSE
    WRITE (*, "SQUARE-ROOT = ", SQRT(A))
ENDIF
```

上の条件は、次のように 1 つの fclass 命令で決定することができる。

```
fclass [fill in the details here]
```

結果として得られる相補的なプレディケートを使って、THEN 文と ELSE 文を個別に制御することができる。

13.3.6.2 Min/Max/AMin/AMax

IA-64 は、FORTRAN 組み込み命令 MIN(a、b) またはこれに相当する C のイディオム:a<b ? a : bと、FORTRAN 組み込み命令 MAX(a、b) またはこれに相当する C のイディオム: a<b ? b : a を直接に命令レベルでサポートしている。これらの命令は、FORTRAN では関数コールのオーバーヘッドをなくし、C ではクリティカル・パスの長さを減らすことによって、パフォーマンスを改善することができる。命令は可換でないように設計されているので、ユーザは入力オペランドの順序を適切に選択することで、NaN を無視したり、取得したりすることができる。

配列の中の最小値を探すという問題を考えてみる (これは Livermore FOTRAN kernel 24 に似ている)。

```
   XMIN = X(1)
   DO 24  k= 2,n
24    IF(X(k) .LT. XMIN)  XMIN = X(k)
```

NaN は順序を持たないので、NaN との比較 (LT を含む) を行うと、false が返される。上のコードが次のようにされたとする。

```
    ldf     f5  = r5, 8 ;;
L1: ldf     f6  = r5, 8
    fmin    f5  = f6, f5
    br.cloop L1 ;;
```

配列 (X) の値 (`f6` にロード) が NaN である場合、(`f5` の中の) 新しい最小値は変更されない。これは、NaN が .LT. の比較に失敗し、`fmin` 命令が第 2 引数、この例では `f5` に含まれている以前の最小値を返すからである。

一方、コードが次のようにされたとする。

```
    ldf     f5  = r5, 8 ;;
L1: ldf     f6  = r5, 8
    fmin    f5  = f5, f6
    br.cloop L1 ;;
```

配列 (X) の値 (`f6` にロード) が NaN である場合、(`f5` の中の) 新しい最小値は NaN に設定される。これは、NaN が `.LT.` の比較に失敗し、`fmin` 命令が第 2 引数、この例では `f6` に含まれている NaN を返すからである。

`famin/famax` 命令は、入力オペランドの絶対値に対して比較を実行する (つまり符号ビットを無視する) が、それ以外の点では `fmin/fmax` 命令と同じように動作する (非可換)。

13.3.6.3 整数 / 浮動小数点変換

符号なし整数をそれと等価な値の浮動小数点表現に変換するときには、単に `setf.sig` 命令を使って、整数を浮動小数点レジスタの有効桁フィールドに移動する。結果として得られる浮動小数点値は、(符号なし整数が 263 よりも大きかった場合を除き) 非正規数の表現となる。

符号付き整数から浮動小数点への変換と、浮動小数点から符号付きまたは符号なし整数への変換は、それぞれ `fcvtxf` 命令と `fcvtfx/fcvtfxu` 命令によって行われる。ただし、符号付き整数はその標準の浮動小数点表現に直接変換されるので、変換後に正規化を行う必要はない。

13.3.6.4 FP サブフィールドの処理

ときには浮動小数点値をその構成フィールドを組み合わせて生成するのが便利なことがある。たとえば、浮動小数点値の 2 のべき乗による乗算と除算は、指数部を適切に調節することで簡単に実行することができる。IA-64 では、整数および浮動小数点のレジスタ・ファイルの間で浮動小数点フィールドを移動できる命令を提供している。浮動小数点数の 2.0 による除算は、次のように行われる。

```
    getf.exp   r5  = f5           // Move S+Exp to int
    add        r5  = r5, -1       // Sub 1 from Exp
    setf.exp   f6  = r5           // Move S+Exp to FP
    fmerge.se  f5  = f6, f5       // Merge S+E w/ Mant
```

また、複数の浮動小数点レジスタのフィールドをもとに浮動小数点値を作成することもできる。

13.3.7 メモリ・アクセス制御

メモリ・アクセス・レイテンシが大きくなりつつある傾向と、高い帯域幅のコストを考慮して、IA-64 にはメモリ階層の管理とパフォーマンスの改善に貢献するさまざまなアーキテクチャ上の機能が組み込まれている。13.2 節で説明したように、メモリ・レイテンシと帯域幅は、浮動小数点アプリケーションのパフォーマンスの大きな制限要因である。IA-64 は、この両方の制限に対処するための機能を提供している。

浮動小数点レジスタ・ファイルのコアの帯域幅を拡張するために、IA-64 はペア・ロード命令を定義している。メモリ・レイテンシを減らすために、IA-64 は明示的および暗黙のデータ・プリフェッチを定義している。キャッシュの利用効率を最大にするために、IA-64 はキャッシュ内のデータの割り当て (および割り当て解除) の制御に役立つ局所性の属性をメモリ・アクセス命令の一部として定義している。命令帯域幅がパフォーマンスの制限要因となるような状況のために、IA-64 は対応する命令プリフェッチをトリガするマシン・ヒントを定義している。

13.3.7.1 ペア・ロード命令

浮動小数点ペア・ロード命令では、メモリ内の 2 つの連続した値を、2 つの独立した浮動小数点レジスタにロードすることができる。ターゲット・レジスタは、マシンが 1 つのアクセス・ポートを使ってレジスタを更新できるように、奇数と偶数の物理レジスタでなくてはならない。

注: 奇数 / 偶数のペアという制約は、論理レジスタ番号ではなく物理レジスタ番号に対するものである。この規則に対するプログラミング違反があると、無効操作フォルトが発生する。

たとえば、マシンが毎サイクルに 2 つのペア・ロードを実行できるだけの L1 からの十分な帯域幅を備えているものとする。この場合、データが L1 にあるときには、浮動小数点命令 1 つにつき 2 つのデータ要素 (それぞれ 8 バイト) を必要とするループがピーク速度で動作することができる。このようなケースの典型的な例は、単純な倍精度のドット積、DDOT である。

```
  DO 1 I = 1, N
1     C = C + A(I) * B(I)
```

内側ループは 2 つのロード (A と B)、および 1 つの `fma` 命令 (積を C に累積する) から構成されている。このループは、C に対するリカレンスのために、`fma` 命令のレイテンシで動作する。C に対するリカレンスを避けるには、一般にループを展開し、複数の部分的なアキュムレータを使用する。

```
DO 1 I = 1, N, 8
    C1 = C1 + A[I] * B[I]
    C2 = C2 + A[I+1] * B[I+1]
    C3 = C3 + A[I+2] * B[I+2]
    C3 = C3 + A[I+3] * B[I+3]
    C3 = C3 + A[I+4] * B[I+4]
    C3 = C3 + A[I+5] * B[I+5]
```

```
      C3 = C3 + A[I+6] * B[I+6]
1     C4 = C4 + A[I+7] * B[I+7]
   C = C1 + C2 + C3 + C4
```

通常の (非ダブル・ペアの) ロードが使用された場合、内側ループは 16 のロードと 8 つの fma 命令で構成される。マシンが 2 つのメモリ・ポートを持つとすると、このループは M スロットのアベイラビリティによって制限され、繰り返し 1 回につき 1 クロックのピーク速度で実行される。しかし、このループを 8 つのペア・ロード (A[I]、A[I+1]、および B[I]、B[I+1]、および A[I+2]、A[I+3]、および B[I+2]、B[I+3] など) と 8 つの fmas 命令を使って書き換えると、このループは 2 つの M ユニットだけで、1 クロック当たり繰り返し 2 回 (すなわち繰り返し 1 回につき 0.5 クロック) のピーク速度で実行されるようになる。

13.3.7.2 データ・プリフェッチ

lfetch 命令により、メモリからキャッシュへのデータ・ライン (32 バイト以上として定義される) の先行プリフェッチを行うことができる。そのデータに対するそれ以降のアクセスの局所性の特質を指定し、そのデータをどのキャッシュ・レベルに取り込むべきかを指定する割り当てヒントを使用することができる。

通常のロードでもデータ・プリフェッチの効果は実現できるが、(ロード・ターゲットが決して使用されない場合) lfetch 命令では、プリフェッチされるデータのターゲットとして浮動小数点レジスタを使わずに、メモリ・レイテンシをより効果的に減らすことができる。さらに、lfetch ではデータを複数のレベルのキャッシュにプリフェッチすることができる。

13.3.7.3 割り当て制御

データ・アクセスはさまざまな局所性の属性を持っているため (時間的 / 非時間的、空間的 / 非空間的)、IA-64 ではこれらの属性を反映させてデータ・アクセス (ロードおよびストア) に注釈を付けることができる。これらの注釈に基づき、キャッシュ内のデータ記憶をより効率的に管理することができる。

時間的および非時間的ヒントが定義されている。これらの属性は、各種のキャッシュ・レベルに適用できる (アーキテクチャ上識別されているのは 2 つのキャッシュ・レベルだけである)。非時間的ヒントは、一般にそのキャッシュ・レベルでは再利用が行われないデータに使用される。時間的データは (再利用が行われる) その他のすべてのデータに使用される。

13.4 要約

本章では、多くの科学アプリケーションおよび浮動小数点アプリケーションの制限要因、すなわちメモリ・レイテンシと帯域幅、機能ユニットのレイテンシ、および使用可能な機能ユニット数について説明した。また、IA-64 の、これらのパフォーマンスの制限要因の克服に貢献する、第 12 章「ソフトウェ

アによるパイプライン化とループのサポート」で説明したソフトウェアによるパイプライン化のサポート以外の重要な浮動小数点サポート機能についても説明した。スペキュレーション、丸め、および精度制御のためのアーキテクチャ上のサポートについても説明した。

本章の例には、浮動小数点の除算と平方根、リダクションなどの一般的な科学計算、fma 命令などの機能の使用方法、および各種の Livermore カーネルなどが取り上げられている。

第 III 部： 付録

intel®

命令シーケンスについて A

命令の実行は、次の 4 つのフェーズから構成されている。

1. 命令をメモリから読み込む (*フェッチ*)。
2. 必要ならばアーキテクチャ状態を読み込む (*読み込み*)。
3. 指定された操作を実行する (*実行*)。
4. 必要ならばアーキテクチャ状態を更新する (*更新*)。

命令グループとは、指定されたバンドル・アドレスとスロット番号で始まり、最初のストップまたは発生する分岐で終わる一連の命令シーケンスで、スロット番号とバンドル・アドレスはシーケンシャルに増える。命令グループ内の命令が、明確に定義されたように動作するためには、以下に述べる順序付けと依存関係の要件を満たしていなくてはならない。

命令グループの中の命令がリソースの依存関係の要件を満たしていれば、プログラムは、個々の命令が上に示した各フェーズを順番どおりに通ったように動作する。特定の命令のフェーズと、それに先行する命令の任意のフェーズの間の順序は、以下に示す命令シーケンス規則によって定められる。

- 命令の*フェッチ*と、それ以前に動的に存在する命令の*読み込み*、*実行*、または*更新*の間には、先天的な関係は存在しない。`sync.i` 命令および `srlz.i` 命令を使うと、それ以降のすべての命令の*フェッチ*と、それ以前のすべての命令の*更新*の間にシーケンシャルな関係を強制的に持たせることができる。
- 異なる命令グループの間では、特定の命令グループの中のすべての命令は、その読み込みが、それ以前の命令グループのすべての命令の更新の後に行われたかのように動作する。すべての命令はユニット・レイテンシを持つものと仮定される。ストップの反対側にある命令は、アーキテクチャ上、少なくとも 1 単位のレイテンシによって分離されていると見なされる。

  一部のシステム状態の更新は、ここに示したよりも厳しい要件を持つ。
- 1 つの命令グループの中では、すべての命令が、そのメモリと ALAT 状態の読み込みが、その命令グループ内のそれ以前のすべての命令のメモリと ALAT 状態の更新の後に行われたかのように動作する。
- 1 つの命令グループの中では、すべての命令が、そのレジスタ状態の読み込みが、その命令内の (前後を問わず) 任意の命令によるレジスタ状態の更新の前に行われたかのように動作する。ただし、後に「レジスタの依存関係」と「メモリの依存関係」の項で述べる例外に注意すること。

上記の規則が、レジスタ依存関係の制約、メモリ依存関係の制約、および例外報告の順序のコンテキストを構成している。これらの依存関係の制約は、リソースに対する読み込みと書き込みがプレディケートによって動的に無効にされない命令間にのみ適用される。

- レジスタの依存関係 : 1 つの命令グループの中で、リード・アフター・ライト (RAW) とライト・アフター・ライト (WAW) のレジスタ依存関係は許容されない
(ただし、A-2 ページの「RAW 順序の例外」と A-3 ページの「WAW 順序の例外」を参照)。ライト・アフター・リード (WAR) のレジスタ依存関係は許容される (A-4 ページの「WAR 順序の例外」を参照)。

  これらの依存関係の制約は、(命令のオペランドからの) 明示的なレジスタ・アクセスと、(アプリケーションおよびコントロール・レジスタが特定の命令によって暗黙のうちにアクセスされる場合などの) 暗黙のレジスタ・アクセスの両方に適用される。プレディケート・レジスタ PR0 はこれらのレジスタ依存関係の制約からは除外される。これは、PR0 への書き込みは無視され、読み込みはつねに 1 を返すためである。

- メモリの依存関係 : 1 つの命令グループの中で、RAW、WAW、および WAR のメモリ依存関係と、ALAT 依存関係が許容される。ロードは、同じメモリ・アドレスへの最新のストアの結果を読み出す。同じ命令グループ内で、同じアドレスに対する複数のストアが存在する場合、命令グループの実行後、そのメモリには最新のストアの結果が格納されることになる。同じアドレスに対するロードの後にストアが行われた場合、ロードによってロードされたデータには影響は及ばない。アドバンスド・ロード、チェック・ロード、アドバンスド・ロード・チェック、ストア、およびメモリ・セマフォの命令では、暗黙のうちに ALAT にアクセスする。同じ命令グループ内では RAW、WAW、および WAR の ALAT 依存関係は許容され、メモリの依存関係として説明した内容とおりに動作する。

上に述べた依存関係の制約の最終的な効果は、プロセッサが正しい命令グループの中の命令のすべて (または任意のサブセット) を並列に、または逐次的に実行しても、最終結果は同じになるということである。これらの依存関係の制約が満たされていないと、プログラムの動作は未定義となる。

上に述べた規則の結果として得られる命令シーケンスはシーケンシャルな実行と呼ばれる。

順序規則と依存関係の制約により、正しいシーケンスが実施され、プログラマからシーケンシャルな実行であるように見えていれば、プロセッサは命令の順序を動的に変更し、命令を非ユニット・レイテンシで実行し、さらにはストップまたは実行される分岐の反対側にある命令を同時に実行することができる。

IP は、IP の読み込みと書き込みが、命令ストリームが並列ではなくシリアルに実行されているかのように動作するという点で特殊なリソースである。IP に対する RAW の依存関係は許容され、読み込む側はそれが属しているバンドルの IP を取得する。このため、並列に実行される各バンドルは IP を論理的に読み込み、インクリメントし、書き戻すことになる。WAW も許容される。

無視された AR も、依存関係のチェックという点では例外として扱われない。無視された AR への RAW と WAW の依存関係は許容されない。

A.1 RAW 順序の例外

命令グループ内での RAW のレジスタ依存関係を禁止する規則には、4 つの例外がある。これらの例外は、alloc 命令、チェック・ロード命令、分岐に影響を与える命令、および ld8.fill 命令と st8.spill 命令である。

- alloc 命令は、汎用レジスタ・ファイルのスタックされたサブセットにアクセスするすべての命令が暗黙のうちに読み込む現在のフレーム・マーカ (CFM) に暗黙のうちに書き込みを行う。汎用レジスタ・ファイルのスタックされたサブセットにアクセスする命令が、alloc と同じ命令グループにあると、alloc によって指定されたスタック・フレームを参照する。

注： 一部の命令は、alloc の影響を受ける CFM 以外のリソースに対して RAW または WAW の依存関係を持っており、このために同じ命令グループ内の alloc:flushrs の後に置くことはできない。AR[BSPSTORE] からの移動、AR[RNAT] からの移動、br.cexit、br.ctop、br.wexit、br.wtop、br.call、br.ia、br.ret、clrrrb がその例である。また、alloc は命令グループ内の最初の命令でなくてはならないことにも注意すること。

- チェック・ロード命令は、対応するアドバンスド・ロードに依存しているため、ロードを実行することもしないこともある。チェック・ロードは、ALAT をミスすると、メモリからのロードを実行する。チェック・ロードと、チェック・ロードのターゲットを読み込む後続の命令は、同じ命令グループ内に存在してもかまわない。依存する命令は、チェック・ロードによってロードされた新しい値を取得することになる。
- 分岐命令は分岐レジスタを読み込み、プレディケート・レジスタ、LC、EC、および PFS アプリケーション・レジスタ、および CFM を暗黙のうちに読み込むことがある。LC、EC、およびプレディケート・レジスタを例外として、非分岐命令によるこれらのレジスタへの書き込みは、同じ命令グループの後続の分岐から見えるようになる。非浮動小数点命令によるプレディケート・レジスタへの書き込みは、同じ命令グループの後続の分岐から見えるようになる。同じ命令グループ内での RAW のレジスタ依存関係は、LC と EC では許容されない。プレディケートの書き込む側が浮動小数点命令で、読み込む側が分岐であるような動的な RAW 依存関係も、同じ命令グループ内では許容されない。分岐命令 br.cond、br.call、br.ret、および br.ia は、レジスタの依存関係という点では他の命令と同じように動作する。つまり、修飾プレディケートが 0 ならば、他のリソースの読み込み側または書き込み側としては扱われない。分岐命令 br.cloop、br.cexit、br.ctop、br.wexit、および br.wtop は、修飾プレディケートの値にかかわらず、つねに自分のリソースの読み込み側または書き込み側であるという点で例外的な存在である。
- ld8.fill および st8.spill 命令は、ユーザ NaT コレクション・アプリケーション・レジスタ (UNAT) に暗黙のうちにアクセスする。これらの命令については、UNAT に対して動的な RAW のレジスタ依存関係の制約がビット・レベルで適用される。これらの命令は、UNAT と同じビットにアクセスしない限り、同じ命令グループ内に存在できる。ld8.fill または st8.spill 命令と､UNAT にアクセスする mov ar= ま

たは mov=ar 命令の間の RAW の UNAT 依存関係は、同じ命令グループ内で生じてはならない。

リソースの依存関係の点では、CFM は 1 つのリソースとして扱われる。

A.2 WAW 順序の例外

命令グループ内での WAW のレジスタ依存関係を禁止する規則には、3 つの例外がある。これらは比較タイプの命令、浮動小数点命令、および st8.spill 命令である。

- 比較タイプの命令には、cmp、cmp4、tbit、tnat、fcmp、frsqrta、frcpa、および fclass がある。同じ命令グループ内の比較タイプの命令は、以下の条件が満たされていれば同じプレディケート・レジスタをターゲットにすることができる。
  - — 比較タイプの命令が、すべて AND タイプの比較であるか、OR タイプの比較である (AND タイプの比較は ".and" および ".andcm" コンプリータに対応し、OR タイプの比較は、".or" および ".orcm" コンプリータに対応する)。
  - — 比較タイプの命令はすべて PR 0 をターゲットとしている。PR 0 に対してすべての WAW の依存関係が許容される。比較は任意のタイプでよく、異なるタイプが混在していてもかまわない。

命令グループ内のその他のすべての WAW の依存関係は禁止される。これには、他の書き込み側と同じプレディケート・レジスタにアクセスする、PR への移動命令に対する動的な WAW のレジスタ依存関係が含まれる。

注： PR への移動命令はマスクによって指定された PR だけに書き込みを行うが、PR からの移動命令ではつねにすべてのプレディケート・レジスタを読み込む。

- 浮動小数点命令は、暗黙のうちに浮動小数点ステータス・レジスタ (FPSR) とプロセッサ・ステータス・レジスタ (PSR) に書き込みを行う。FPSR と PSR に対して WAW のレジスタ依存関係の制約は適用されないので、同じ命令グループ内には複数の浮動小数点命令が存在できる。命令グループを実行した後の FPSR と PSR の状態は、すべての書き込みの論理和となる。
- st8.spill 命令は UNAT レジスタに暗黙のうちに書き込みを行う。この命令については、UNAT に対して WAW のレジスタ依存関係の制約がビット・レベルで適用される。UNAT の同じビットに書き込みを行わないのであれば、同じ命令グループ内に複数の st8.spill 命令があってもよい。st8.spill 命令と、UNAT をターゲットとする mov ar= 命令の間の WAW レジスタの依存関係は、同じ命令グループ内では生じてはならない。

無視された AR に対する WAW の依存関係は許容されない。

A.3 WAR 順序の例外

分岐命令による PR63 の読み込みと、同じグループ内のそれ以降のループ終了分岐 (`br.ctop`、`br.cexit`、`br.wtop`、または `br.wexit`) による PR63 への書き込みの間の WAR 依存関係は許容されない。それ以外の WAR 依存関係は許容される。

intel®

IA-64 擬似コード関数 B

表 B-1 に、第 7 章「IA-64 命令リファレンス」で使用しているすべての擬似コード関数を示す。

表 B-1. 擬似コード関数

関数	操作
*xxx*_fault(parameters ...)	3　フォルト関数はいくつか存在する。個々のフォルト関数は、そのフォルトに固有のパラメータを受け付ける (例外コード値、仮想アドレスなど)。フォルトがスペキュレーティブ・ロード例外のために据え置かれた場合、フォルト・ルーチンが据え置きが行われたことの通知とともに返る。それ以外の場合、フォルト・ルーチンは返らず、命令シーケンスを終了させる。
*xxx*_trap(parameters ...)	トラップ関数はいくつか存在する。個々のトラップ関数は、そのトラップに固有のパラメータを受け付ける (トラップ・コード値、仮想アドレスなど)。トラップ・ルーチンは返らない。
acceptance_fence()	キャッシングされていない順序付きシーケンシャル・メモリ・ページに対する先行のデータ・メモリ参照が、後続のデータ・メモリ参照がプロセッサによって実行される前に「受け付けられる」ようにする。
alat_cmp(rtype, raddr)	`rtype` によって指定されたレジスタ・タイプと、`raddr` によって指定されたレジスタ・アドレスに一致する ALAT エントリを見つけた場合に 1 を返す。それ以外の場合はゼロを返す。この関数はプロセッサ固有である。プロセッサは、ALAT 内に一致するエントリが存在している場合でも、(一致するものがなかったことを示す) ゼロを返すように任意に設定できる。これにより、高速な ALAT ルックアップ・サーキットを設計する際のプロセッサの柔軟性が実現されている。
alat_frame_update(delta_bof, delta_sof)	ALAT に対して、フレームの下端やフレームのサイズが変更されたことを通知する。これにより、ALAT のタグ・ビットを管理したり、その他の必要な管理機能を実現することができる。
alat_inval()	ALAT 内のすべてのエントリを無効にする。
alat_inval_multiple_entries(paddr, size)	ALAT に対して、`paddr` によって指定された物理メモリ・アドレスと、`size` によって指定されたアクセス・サイズを使って照会を発行する。一致したすべての ALAT エントリが無効になる。値は返されない。
alat_inval_single_entry(rtype, rega)	ALAT に対して、`rtype` によって指定されたレジスタ・タイプと、`rega` によって指定されたレジスタ・アドレスを使って照会を発行する。一致した ALAT エントリが 1 つだけ無効になる。値は返されない。

表 B-1. 擬似コード関数 (続き)

関数	操作
alat_write(rtype, raddr, paddr, size)	rtype によって指定されたレジスタ・タイプ、raddr によって指定されたレジスタ・アドレス、paddr によって指定された物理メモリ・アドレス、および size によって指定されたアクセス・サイズを使って、新しい ALAT エントリを割り当てる。値は返されない。この関数によって、特定の raddr に対応する ALAT エントリが 1 つしか存在しないことが保証される。ld.c.nc、ldf.c.nc、または ldfp.c.nc 命令の raddr が既存の ALAT エントリのレジスタ・タグに一致したが、命令の size および (または)paddr が既存のエントリとは異なる場合、この関数は既存のエントリを保存するか、それを無効にして、命令で指定された size と paddr を使って新しいエントリを書き込む。
check_target_register(r1)	r1 がフレーム範囲外のスタックされたレジスタ (CFM によって定義) をターゲットにすると、無効操作フォルトが発生し、この関数は返らない。
check_target_register_sof(r1, newsof)	r1 がフレーム範囲外のスタックされたレジスタ (newsof パラメータによって定義) をターゲットにすると、無効操作フォルトが発生し、この関数は返らない。
concatenate2(x1, x2)	2 つの引数の下位 32 ビットを連結し、64 ビットの結果を返す。
concatenate4(x1, x2, x3, x4)	4 つの引数の下位 16 ビットを連結し、64 ビットの結果を返す。
concatenate8(x1, x2, x3, x4, x5, x6, x7, x8)	8 つの引数の下位 8 ビットを連結し、64 ビットの結果を返す。
fadd(fp_dp, fr2)	無限精度の積に浮動小数点レジスタの値を加え、丸めの準備ができた無限精度の和を返す。
fcmp_exception_fault_check(fr2, fr3, frel, sf, *tmp_fp_env)	fcmp 命令のためにすべての浮動小数点フォルト条件をチェックする。
fcvt_fx_exception_fault_check(fr2, trunc, sf *tmp_fp_env)	fcvt.fxおよびfcvt.fx.trunc命令のためにすべての浮動小数点フォルト条件をチェックする。NaN と NaTVal は伝播する。
fcvt_fxu_exception_fault_check(fr2, trunc, sf, *tmp_fp_env)	fcvt.fxu および fcvt.fxu.trunc 命令のためにすべての浮動小数点フォルト条件をチェックする。NaN と NaTVal は伝播する。
fma_exception_fault_check(fr2, fr3, fr4, pc, sf, *tmp_fp_env)	fma 命令のためにすべての浮動小数点フォルト条件をチェックする。NaN、NaTVal、および特殊な IEEE 結果は伝播する。
fminmax_exception_fault_check(fr2, fr3, sf, *tmp_fp_env)	famax、famin、fmax、および fmin 命令のためにすべての浮動小数点フォルト条件をチェックする。
fms_fnma_exception_fault_check(fr2, fr3, fr4, pc, sf, *tmp_fp_env)	fms および fnma 命令のためにすべての浮動小数点フォルト条件をチェックする。NaN、NaTVal、および特殊な IEEE 結果は伝播する。
fmul(fr3, fr4)	2 つの浮動小数点レジスタ値の無限精度の乗算を実行する。
followed_by_stop()	現在の命令の後にストップが続いている場合には TRUE を返す。それ以外の場合は FALSE を返す。

表 B-1. 擬似コード関数 (続き)

関数	操作
fp_check_target_register(f1)	指定された浮動小数点レジスタ識別子が 0 または 1 であれば、この関数は無効操作フォルトを発生させる。
fp_decode_fault(tmp_fp_env)	ISR.code のための浮動小数点例外フォルト・コード値を返す。
fp_decode_traps(tmp_fp_env)	ISR.code のための浮動小数点トラップ・コード値を返す。
fp_is_nan_or_inf(freg)	浮動小数点例外フォルト・チェック関数が、IEEE フォルトがディスエーブルにされたデフォルトの結果または伝播された NaN を返した場合に、true を返す。
fp_equal(fr1, fr2)	IEEE 標準の等値関係テスト。
fp_ieee_recip(num, den)	特殊なオペランドのセットに対する真の商、またはソフトウェア除算アルゴリズムで使用される除数の逆数の近似値を返す。
fp_ieee_recip_sqrt(root)	特殊なオペランドに対する真の平方根の結果、またはソフトウェア平方根アルゴリズムで使用される平方根の逆数の近似値を返す。
fp_is_nan(freg)	浮動小数点レジスタが NaN を含んでいる場合に true を返す。
fp_is_natval(freg)	浮動小数点レジスタが NaTVal を含んでいる場合に true を返す。
fp_is_normal(freg)	浮動小数点レジスタがノーマル数を含んでいる場合に true を返す。
fp_is_pos_inf(freg)	浮動小数点レジスタが正の無限大を含んでいる場合に true を返す。
fp_is_qnan(freg)	浮動小数点レジスタがクワイエット型 NaN を含んでいる場合に true を返す。
fp_is_snan(freg)	浮動小数点レジスタがシグナル型 NaN を含んでいる場合に true を返す。
fp_is_unorm(freg)	浮動小数点レジスタがアンノーマル数を含んでいる場合に true を返す。
fp_is_unsupported(freg)	浮動小数点レジスタがサポートされていない形式を含んでいる場合に true を返す。
fp_less_than(fr1, fr2)	IEEE 標準の「未満」関係テスト。
fp_lesser_or_equal(fr1, fr2)	IEEE 標準の「以下」関係テスト。
fp_normalize(fr1)	アンノーマルの fp 値を正規化する。この関数は、レジスタ・ファイルで表現できないアンノーマル数値をすべてゼロにフラッシュする。
fp_raise_fault(tmp_fp_env)	ローカル命令状態をチェックして、割り込みの発生を必要とするフォルト条件が存在するかどうかを調べる。
fp_raise_traps(tmp_fp_env)	ローカル命令状態をチェックして、割り込みの発生を必要とするトラップ条件が存在するかどうかを調べる。
fp_reg_bank_conflict(f1, f2)	2 つの指定された FR が同じバンクにある場合に true を返す。

表 B-1. 擬似コード関数 (続き)

関数	操作
fp_reg_disabled(f1, f2, f3, f4)	ディスエーブルにされている浮動小数点レジスタ・フォルトが存在するかどうかをチェックする。
fp_reg_read(freg)	FR を読み込み、標準の拡張倍精度のデノーマル数 (および擬似デノーマル数) に、真の数学的指数を与える。これ以外のクラスのオペランドは変更されない。
fp_unordered(fr1, fr2)	IEEE 標準の順序付けなしの関係。
fp_fr_to_mem_format(freg, size)	レジスタ形式の浮動小数点値を、浮動小数点メモリ形式に変換する。レジスタ内の浮動小数点値が、size パラメータに対応する正しい精度に事前に丸められているという前提がある。
frcpa_exception_fault_check(fr2, fr3, sf, *tmp_fp_env)	frcpa 命令のためにすべての浮動小数点フォルト条件をチェックする。NaN、NaTVal、および特殊な IEEE 結果は伝播する。
frsqrta_exception_fault_check(fr3, sf, *tmp_fp_env)	frsqrta 命令のためにすべての浮動小数点フォルト条件をチェックする。NaN、NaTVal、および特殊な IEEE 結果は伝播する。
ignored_field_mask(regclass, reg, value)	指定されたレジスタとレジスタ・タイプの無視ビットに対応するビットを 0 にクリアして値を返すブール関数。
instruction_serialize()	副作用を持つ先行するすべてのレジスタ更新が、それ以降の命令とデータ・メモリ参照が実行される前に行われることが保証される。また、先行の SYNC.i 操作が命令キャッシュによって検出されることが保証される。
instruction_synchronize	キャッシュ・フラッシュ (FC) 操作のために命令とデータ・ストリームを同期させる。この関数により、先行の FC 操作がローカル・データ・キャッシュによって検出されたときには、ローカル命令キャッシュによっても検出されることが保証される。また、先行の FC 操作が別のプロセッサのデータ・キャッシュによって検出されたときには、同じプロセッサの命令キャッシュ内でも検出されることが保証される。
is_finite(freg)	浮動小数点レジスタが有限数を含んでいる場合に true を返す。
is_ignored_reg(regnum)	regnum が無視されたアプリケーション・レジスタである場合に true を返し、それ以外の場合に false を返すブール関数。
is_inf(freg)	浮動小数点レジスタが無限数を含んでいる場合に true を返す。
is_kernel_reg(ar_addr)	ar_addr がカーネル・レジスタ・アプリケーション・レジスタのアドレスである場合に 1 を返す。
is_reserved_field(regclass, arg2, arg3)	指定されたデータが予約済みフィールドに 1 を書き込む場合に true を返す。
is_reserved_reg(regclass, regnum)	regnum が regclass レジスタ・ファイルで予約されている場合に true を返す。
mem_flush(paddr)	物理アドレス paddr によってアドレス指定されているラインは、メモリ階層の中のメモリよりも上のすべてのレベルで無効にされ、メモリとの間に整合性がない場合にはメモリに書き戻される。

表 B-1. 擬似コード関数 (続き)

関数	操作
mem_implicit_prefetch(vaddr, hint)	vaddr によってアドレス指定されているラインを、hint によって指定されたメモリ階層の位置に移動する。この関数はプロセッサに依存しており、無視されることがある。
mem_promote(paddr, mtype, hint)	paddr によってアドレス指定されているラインを、hint によって指定されたアクセス・ヒントの条件に従って、メモリ階層の最高レベルに移動する。この関数はプロセッサに依存しており、無視されることがある。
mem_read(paddr, size, border, mattr, otype, hint)	paddr によって指定された物理メモリ位置から始まる size バイトを、border によって指定されたバイト順序、mattr によって指定されたメモリ属性、および hint によって指定されたアクセス・ヒントで返す。otype はこのアクセスのメモリ・アクセス順序属性を指定するもので、UNORDERED または ACQUIRE でなくてはならない。
fp_mem_to_fr_format(mem, size)	メモリ形式の浮動小数点値を浮動小数点レジスタ形式に変換する。
mem_write(value, paddr, size, border, mattr, otype, hint)	value の下位 size バイトを、parrd で指定された物理メモリ・アドレスから始まるメモリに書き込む。この際には、border によって指定されたバイト順序、mattr によって指定されたメモリ属性、および hint によって指定されたアクセス・ヒントが使用される。otype はこのアクセスのメモリ・アクセス順序属性を指定するもので、UNORDERED または RELEASE でなくてはならない。値は返されない。
mem_xchg(data, paddr, size, byte_order, mattr, otype, hint)	paddr によって指定された物理アドレスから始まる size バイトを返す。この読み込みは hint によって指定された局所性ヒントの条件に従う。読み込みの後、データの下位 size バイトが、メモリ内の paddr によって指定された物理アドレスから始まる size バイトに書き込まれる。読み込みと書き込みはアトミックに実行される。読み込みと書き込みの両方が、mattr によって指定されたメモリ属性の条件に従い、メモリ内のバイト順序は byte_order で指定される。otype はこのアクセスのメモリ・アクセス順序属性を指定するもので、ACQUIRE でなくてはならない。
mem_xchg_add(add_val, paddr, size, byte_order, mattr, otype, hint)	paddr によって指定された物理アドレスから始まる size バイトを返す。この読み込みは hint によって指定された局所性ヒントの条件に従う。その後、メモリから読み込まれた値の和の下位 size バイトと add_val がメモリ内の paddr によって指定された物理アドレスから始まる size バイトに書き込まれる。読み込みと書き込みはアトミックに実行される。読み込みと書き込みの両方が、mattr によって指定されたメモリ属性の条件に従い、メモリ内のバイト順序は byte_order で指定される。otype はこのアクセスのメモリ・アクセス順序属性を指定するもので、ACQUIRE または RELEASE でなくてはならない。

表 B-1. 擬似コード関数 (続き)

関数	操作
mem_xchg_cond(cmp_val, data, paddr, size, byte_order, mattr, otype, hint)	paddr によって指定された物理アドレスから始まる size バイトを返す。この読み込みは hint によって指定された局所性ヒントの条件に従う。メモリから読み込まれた値が cmp_val と等しければ、データの下位 size バイトが、メモリ内の paddr によって指定された物理アドレスから始まる size バイトに書き込まれる。書き込みが実行される場合、読み込みと書き込みはアトミックに実行される。読み込みと書き込みの両方が、mattr によって指定されたメモリ属性の条件に従い、メモリ内のバイト順序は byte_order で指定される。otype はこのアクセスのメモリ・アクセス順序属性を指定するもので、ACQUIRE または RELEASE でなくてはならない。
ordering_fence()	先行のデータ・メモリ参照が、将来のデータ・メモリ参照がプロセッサから見えるようになる前に見えることが保証される。
pr_phys_to_virt(phys_id)	プレディケートの物理レジスタ ID、phys_id から、プレディケートの仮想レジスタ ID を返す
rotate_regs()	レジスタ・リネーム・ベース・レジスタをデクリメントし、結果としてレジスタ・ファイルのローテートを行う。CFM.rrb.gr は CFM.sor がゼロでない場合にのみデクリメントされる。
rse_enable_current_frame_load()	RSE ロード・ポインタ (RSE.BSPLoad) が AR[BSP] よりも大きい場合、RSE による強制ロードによってカレント・フレームのレジスタを復元できることを示す RSE.CFLE ビットが設定される (他のケースでは、RSE はカレント・フレームのレジスタのスピルまたはフィルを行わない)。この関数は必須の RSE によるロードを実行しない。この手順で割り込みは発生しない。
rse_invalidate_non_current_regs()	カレント・フレームの外のすべてのレジスタが無効にされる。
rse_new_frame(current_frame_size, new_frame_size)	レジスタ・リネームをまったく変更せずに、新しいフレームが定義される。新しいフレーム・サイズは new_frame_size パラメータによって完全に定義される (連続したコールは累積しない)。new_frame_size が current_frame_size よりも大きく、無効でクリーンなパーティション内のレジスタの数がフレームの増加したサイズよりも小さい場合には、十分な数のレジスタが使用可能になるまで RSE による強制ストアが発行される。結果として生じる RSE ストアのシーケンスは割り込まれることがある。RSE による強制ストアによって割り込みが発生することがある。rse_store のリストを参照。
rse_preserve_frame(preserved_frame_size)	preserved_frame_size によって指定される数のレジスタが、RSE によって予約されるようにマークされる。レジスタ・リネームにより、GR[32] 以降の preserved_frame_size 個のレジスタが GR[32] にリネームされる。AR[BSP] が、新しい GR[32] が格納されるバッキング・ストア・アドレスを含むように更新される。

表 B-1. 擬似コード関数 (続き)

関数	操作
rse_store(type)	レジスタまたは NaT コレクションをバッキング・ストアに保存する (store_address = AR[BSPSTORE])。store_address{8:3} が 0x3f に等しい場合には、NaT コレクション AR[RNAT] がストアされる。store_address{8:3} が 0x3f に等しくない場合は、レジスタ RSE.StoreReg がストアされ、そのレジスタの NaT ビットが AR[RNAT]{store_address{8:3}} に格納される。ストアが成功すると、AR[BSPSTORE] が 8 だけインクリメントされる。ストアが成功し、レジスタがストアされた場合、RSE.StoreReg が 1 だけインクリメントされる (スタックされたレジスタでのラッピングが行われることがある)。このストアによって、ダーティー・パーティションからクリーン・パーティションにレジスタを移動する。ストアの特権レベルは AR[RSC].pl から取得される。ストアのバイト順序は AR[RSC].be から取得される。RSE による強制ストアでは、タイプは MANDATORY である。RSE ストアは ALAT エントリを無効にしない。
rse_update_internal_stack_pointers(new_store_pointer)	この関数は、AR[BSPSTORE] に新しい値 (new_store_pointer) が与えられたときに、AR[BSP] の新しい値を計算する。この値は、new_store_pointer に、ダーティー・レジスタの数と、その間にある NaT コレクションの数を加えたものに等しい。つまり、ダーティー・パーティションのサイズは、AR[BSPSTORE] の書き込みの前後で同じである。すべてのクリーン・レジスタは無効なパーティションに移動される。
sign_ext(value, pos)	ビット pos-1 から 0 までが value で埋められ、ビット位置 pos から 63 までが value のビット pos-1 で埋められている 64 ビットの数値を返す。pos が 64 以上である場合には、value が返される。
tlb_translate(vaddr, size, type, cpl, *attr, *defer)	変換がイネーブルであるときに、指定された仮想メモリ・アドレス (vaddr) の変換後のデータ物理アドレスを返す。イネーブルでない場合は、vaddr を返す。size でアクセスのサイズを指定し、type でアクセスのタイプを指定する (read、write、advance、spec など)。cpl でアクセス・チェックのための特権レベルを指定する。*attr はマップされた物理メモリ属性を返す。フォルト条件が検出され、据え置かれた場合、tlb_translate は *defer を設定して返る。フォルトが生成されたが、フォルトが据え置かれなかった場合、tlb_translate は返らない。
tlb_translate_nonaccess(vaddr, type)	指定された仮想メモリ・アドレス (vaddr) の変換後のデータ物理アドレスを返す。type でアクセスのタイプを指定する (FC など)。フォルトが生成された場合、tlb_translate_nonaccess は返らない。
unimplemented_physical_address(paddr)	指定された物理アドレスが、このプロセッサ・モデルでサポートされていない場合に TRUE を返す。それ以外の場合には FALSE を返す。この関数はモデル固有である。
impl_undefined_natd_gr_read(paddr, size, be, mattr, otype, ldhint)	NaT のアドレスへのスペキュレーティブ・ロードに対するレジスタの戻りデータを定義する。この関数は他のアドレス空間からのデータを返すことがある。
unimplemented_virtual_address(vaddr)	指定された仮想アドレスが、このプロセッサ・モデルでサポートされていない場合に TRUE を返す。それ以外の場合には FALSE を返す。この関数はモデル固有である。

表 B-1. 擬似コード関数 (続き)

関数	操作
fp_update_fpsr(sf, tmp_fp_env)	浮動小数点命令のローカル状態をグローバル FPSR にコピーする。
zero_ext(value, pos)	ビット pos-1 から 0 までが value で埋められ、ビット位置 pos から 63 までがゼロとなっている 64 ビットの符号なし数値を返す。pos が 64 以上である場合には、value が返される。

IA-64 命令形式 C

個々の IA-64 命令は、6 つのタイプに分類される。各命令タイプは 1 つまたは複数の実行ユニット・タイプ上で実行できる。表 C-1 に、命令タイプと、それらを実行できる実行ユニット・タイプを示す。

表 C-1. 命令タイプと実行ユニット・タイプの関係

命令タイプ	説明	実行ユニット・タイプ
A	整数 ALU	I ユニットまたは M ユニット
I	非 ALU 整数	I ユニット
M	メモリ	M ユニット
F	浮動小数点	F ユニット
B	分岐	B ユニット
L+X	拡張	I ユニット

3 つの命令が、128 ビットのサイズでアラインメントされて、**バンドル**と呼ばれるコンテナにグループ化される。各バンドルは、41 ビットの**命令スロット** 3 つと、5 ビットのテンプレート・フィールド 1 つを含んでいる。バンドルの形式を図 C-1 に示す。

図 C-1. バンドルの形式

テンプレート・フィールドは、カレント・バンドル内のストップと、命令スロットから実行ユニット・タイプへのマッピングの 2 つのプロパティを指定する。この 2 つのプロパティのすべての組み合わせが許容されているわけではない。表 C-2 に定義されている組み合わせを示す。右側の 3 つの列は、バンドル内の 3 つの命令スロットに対応する。個々の列では、テンプレート・フィールドのエンコーディングごとに、その命令スロットによって制御される実行ユニットのタイプが示されている。命令スロットの右側にある二重の線は、カレント・バンドル内のそのポイントでストップが起こることを示している。ストップの定義については、3-14 ページの「命令のエンコーディングの概要」を参照のこと。バンドルの中では、実行順序はスロット 0 からスロット 2 に向かう。使用されていないテンプレート値 (表 C-2 では空のスロットとして示している) は予約されており、無効操作フォルトを発生させる。

long 型の即値整数命令に使用される拡張命令では、2 つの命令スロットを占有する。

表 C-2. テンプレート・フィールドのエンコーディングと命令スロットのマッピング

テンプレート	スロット 0	スロット 1	スロット 2
00	M ユニット	I ユニット	I ユニット
01	M ユニット	I ユニット	I ユニット
02	M ユニット	I ユニット	I ユニット
03	M ユニット	I ユニット	I ユニット
04	M ユニット	L ユニット	X ユニット
05	M ユニット	L ユニット	X ユニット
06			
07			
08	M ユニット	M ユニット	I ユニット
09	M ユニット	M ユニット	I ユニット
0A	M ユニット	M ユニット	I ユニット
0B	M ユニット	M ユニット	I ユニット
0C	M ユニット	F ユニット	I ユニット
0D	M ユニット	F ユニット	I ユニット
0E	M ユニット	M ユニット	F ユニット
0F	M ユニット	M ユニット	F ユニット
10	M ユニット	I ユニット	B ユニット
11	M ユニット	I ユニット	B ユニット
12	M ユニット	B ユニット	B ユニット
13	M ユニット	B ユニット	B ユニット
14			
15			
16	B ユニット	B ユニット	B ユニット
17	B ユニット	B ユニット	B ユニット
18	M ユニット	M ユニット	B ユニット
19	M ユニット	M ユニット	B ユニット
1A			
1B			
1C	M ユニット	F ユニット	B ユニット
1D	M ユニット	F ユニット	B ユニット
1E			
1F			

C.1 形式の要約

命令セットのすべての命令は長さ 41 ビットである。各命令の左側の 4 ビット (40:37) はメジャー・オペコードである。表 C-3 に、ALU(A)、整数 (I)、メモリ (M)、浮動小数点 (F)、および分岐 (B) の 5 つの命令タイプのメジャー・オペコードの割り当てを示す。バンドルのテンプレート・ビットは 4 つの列を区別するために使用されているので、各列の中で同じメジャー・オペコード値が再利用されていることがある。

使用されていないメジャー・オペコード (表 C-3 では空のエントリとして示している) は、次の 3 つのうちのいずれかの動作をする。

- 無視されるメジャー・オペコード (表 C-3 の白のエントリ) は、nop 命令として実行される。
- 予約済みのメジャー・オペコード (表 C-3 の、グレー・スケール・バージョンでは薄灰色、カラー・バージョンでは茶色) は、無効操作フォルトを発生させる。
- PR[qp] が 1 の場合に予約済みとなるメジャー・オペコード (表 C-3 の、グレー・スケール・バージョンでは濃灰色、カラー・バージョンでは紫色) は、命令の qp フィールド (ビット 5:0) で指定されるプレディケート・レジスタが 1 である場合には無効操作フォルトを発生させ、0 の場合は nop 命令として実行される。

表 C-3. メジャー・オペコードの割り当て

メジャー・オペコード (ビット 40:37)	命令タイプ				
	I/A	M/A	F	B	L+X
0	Misc 0	Mem Mgmt 0	FP Misc 0	Misc/Indirect Branch 0	Misc 0
1	1	Mem Mgmt 1	FP Misc 1	Indirect Call 1	1
2	2	2	2	Nop 2	2
3	3	3	3	3	3
4	Deposit 4	Int Ld +Reg/getf 4	FP Compare 4	IP-relative Branch 4	4
5	Shift/Test Bit 5	Int Ld/St +Imm 5	FP Class 5	IP-rel Call 5	5
6	6	FPLd/St+Reg/setf 6	6	6	movl 6
7	MM Mpy/Shift 7	FP Ld/St +Imm 7	7	7	7
8	ALU/MM ALU 8	ALU/MM ALU 8	fma 8	8	8
9	Add Imm_{22} 9	Add Imm_{22} 9	fma 9	9	9
A	A	A	fms A	A	A
B	B	B	fms B	B	B
C	Compare C	Compare C	fnma C	C	C
D	Compare D	Compare D	fnma D	D	D
E	Compare E	Compare E	fselect/xma E	E	E
F	F	F	F	F	F

C-5 ページの表 C-4 に、すべての命令形式の要約を示す。命令フィールドは、C-7 ページの表 C-5 で説明しているように、見やすいように色分けされている。

本章で使用している命令フィールド名については、C-8 ページの表 C-6 で説明している。特殊な表記 (その命令が命令グループの最初の命令でなくてはならない、など) については、C-9 ページの表 C-7 で説明している。これらの表記はオペコード表の「命令」の列に示されている。

即値を含んでいる大部分の命令は、これらの即値を複数の命令フィールドにエンコードしている。たとえば、Add Imm_{14} 命令 (形式 A4) の 14 ビットの即値は、imm_{7b}、imm_{6d}、および s フィールドから作られる。C-92 ページの表 C-65 は、即値を持つ個々の命令について、命令フィールドから即値がどのように生成されるかを示している。

表 C-4. 命令形式の要約

Instruction	Format	Fields (bit 40 → bit 0)
ALU	A1	8 (40–37); (36); x_{2a} (35–34); v_e (33); x_4 (32–29); x_{2b} (28–27); r_3 (26–20); r_2 (19–13); r_1 (12–6); qp (5–0)
Shift L and Add	A2	8; (36); x_{2a}; v_e; x_4; ct_{2d} (28–27); r_3; r_2; r_1; qp
ALU Imm_8	A3	8; s; x_{2a}; v_e; x_4; x_{2b}; r_3; imm_{7b}; r_1; qp
Add Imm_{14}	A4	8; s; x_{2a}; v_e; imm_{6d} (32–27); r_3; imm_{7b}; r_1; qp
Add Imm_{22}	A5	9; s; imm_{9d} (35–27); imm_{5c} (26–22); r_3 (21–20); imm_{7b}; r_1; qp
Compare	A6	C - E; t_b; x_2; t_a; p_2 (32–27); r_3; r_2; c (12); p_1 (11–6); qp
Compare to Zero	A7	C - E; t_b; x_2; t_a; p_2; r_3; 0; c; p_1; qp
Compare Imm_8	A8	C - E; s; x_2; t_a; p_2; r_3; imm_{7b}; c; p_1; qp
MM ALU	A9	8; z_a; x_{2a}; z_b; x_4; x_{2b}; r_3; r_2; r_1; qp
MM Shift and Add	A10	8; z_a; x_{2a}; z_b; x_4; ct_{2d}; r_3; r_2; r_1; qp
MM Multiply Shift	I1	7; z_a; x_{2a}; z_b; v_e; ct_{2d}; x_{2b}; (27); r_3; r_2; r_1; qp
MM Mpy/Mix/Pack	I2	7; z_a; x_{2a}; z_b; v_e; x_{2c}; x_{2b}; (27); r_3; r_2; r_1; qp
MM Mux1	I3	7; z_a; x_{2a}; z_b; v_e; x_{2c}; x_{2b}; (27–24); mbt_{4c} (23–20); r_2; r_1; qp
MM Mux2	I4	7; z_a; x_{2a}; z_b; v_e; x_{2c}; x_{2b}; mht_{8c} (27–20); r_2; r_1; qp
Shift R Variable	I5	7; z_a; x_{2a}; z_b; v_e; x_{2c}; x_{2b}; (27); r_3; r_2; r_1; qp
MM Shift R Fixed	I6	7; z_a; x_{2a}; z_b; v_e; x_{2c}; x_{2b}; (27); r_3; (19); $count_{5b}$ (18–14); (13); r_1; qp
Shift L Variable	I7	7; z_a; x_{2a}; z_b; v_e; x_{2c}; x_{2b}; (27); r_3; r_2; r_1; qp
MM Shift L Fixed	I8	7; z_a; x_{2a}; z_b; v_e; x_{2c}; x_{2b}; (27–25); $ccount_{5c}$ (24–20); r_2; r_1; qp
Popcount	I9	7; z_a; x_{2a}; z_b; v_e; x_{2c}; x_{2b}; (27); r_3; 0; r_1; qp
Shift Right Pair	I10	5; (36); x_2; x; $count_{6d}$ (32–27); r_3; r_2; r_1; qp
Extract	I11	5; (36); x_2; x; len_{6d}; r_3; pos_{6b} (19–14); y (13); r_1; qp
Dep.Z	I12	5; (36); x_2; x; len_{6d}; y (26); $cpos_{6c}$ (25–20); r_2; r_1; qp
Dep.Z Imm_8	I13	5; s; x_2; x; len_{6d}; y; $cpos_{6c}$; imm_{7b}; r_1; qp
Deposit Imm_1	I14	5; s; x_2; x; len_{6d}; r_3; $cpos_{6b}$ (19–14); (13); r_1; qp
Deposit	I15	4; $cpos_{6d}$ (36–31); len_{4d} (30–27); r_3; r_2; r_1; qp
Test Bit	I16	5; t_b; x_2; t_a; p_2; r_3; pos_{6b}; y; c; p_1; qp
Test NaT	I17	5; t_b; x_2; t_a; p_2; r_3; (19–14); y; c; p_1; qp
Break/Nop	I19	0; i; x_3 (35–33); x_6 (32–27); (26); imm_{20a} (25–6); qp
Int Spec Check	I20	0; s; x_3; imm_{13c} (32–20); r_2; imm_{7a}; qp
Move to BR	I21	0; (36); x_3; (32–23); x; (21–20); r_2; (12–9); b_1 (8–6); qp
Move from BR	I22	0; (36); x_3; x_6; (26–16); b_2 (15–13); r_1; qp
Move to Pred	I23	0; s; x_3; (32); $mask_{8c}$ (31–24); (23–20); r_2; $mask_{7a}$; qp
Move to Pred Imm_{44}	I24	0; s; x_3; imm_{27a} (32–6); qp
Move from Pred/IP	I25	0; (36); x_3; x_6; (26–13); r_1; qp
Move to AR	I26	0; (36); x_3; x_6; ar_3; r_2; (12–6); qp
Move to AR Imm_8	I27	0; s; x_3; x_6; ar_3; imm_{7b}; (12–6); qp
Move from AR	I28	0; (36); x_3; x_6; ar_3; (19–13); r_1; qp
Sxt/Zxt/Czx	I29	0; (36); x_3; x_6; r_3; (19–13); r_1; qp

表 C-4. 命令形式の要約 (続き)

命令	形式	40–37	36	35–33	32–31	30	29–28	27	26	25–20	19–16	15	14–13	12–6	5–0
Int Load	M1	4	m	x_6			hint	x	r_3					r_1	qp
Int Load +Reg	M2	4	m	x_6			hint	x	r_3		r_2			r_1	qp
Int Load +Imm	M3	5	s	x_6			hint	i	r_3		imm_{7b}			r_1	qp
Int Store	M4	4	m	x_6			hint	x	r_3		r_2				qp
Int Store +Imm	M5	5	s	x_6			hint	i	r_3		r_2			imm_{7a}	qp
FP Load	M6	6	m	x_6			hint	x	r_3					f_1	qp
FP Load +Reg	M7	6	m	x_6			hint	x	r_3		r_2			f_1	qp
FP Load +Imm	M8	7	s	x_6			hint	i	r_3		imm_{7b}			f_1	qp
FP Store	M9	6	m	x_6			hint	x	r_3		f_2				qp
FP Store +Imm	M10	7	s	x_6			hint	i	r_3		f_2			imm_{7a}	qp
FP Load Pair	M11	6	m	x_6			hint	x	r_3		f_2			f_1	qp
FPLoadPair+Imm	M12	6	m	x_6			hint	x	r_3		f_2			f_1	qp
Line Prefetch	M13	6	m	x_6			hint	x	r_3						qp
LinePrefetch+Reg	M14	6	m	x_6			hint	x	r_3		r_2				qp
LinePrefetch+Imm	M15	7	s	x_6			hint	i	r_3		imm_{7b}				qp
(Cmp &) Exchg	M16	4	m	x_6			hint	x	r_3		r_2			r_1	qp
Fetch & Add	M17	4	m	x_6			hint	x	r_3			s	i_{2b}	r_1	qp
Set FR	M18	6	m	x_6				x			r_2			f_1	qp
Get FR	M19	4	m	x_6				x			f_2			r_1	qp
Int Spec Check	M20	1	s	x_3	imm_{13c}						r_2			imm_{7a}	qp
FP Spec Check	M21	1	s	x_3	imm_{13c}						f_2			imm_{7a}	qp
Int ALAT Check	M22	0	s	x_3	imm_{20b}									r_1	qp
FP ALAT Check	M23	0	s	x_3	imm_{20b}									f_1	qp
Sync/Srlz/ALAT	M24	0		x_3	x_2	x_4									qp
RSE Control	M25	0		x_3	x_2	x_4									0
Int ALAT Inval	M26	0		x_3	x_2	x_4								r_1	qp
FP ALAT Inval	M27	0		x_3	x_2	x_4								f_1	qp
Flush Cache	M28	1		x_3	x_6				r_3						qp
Move to AR	M29	1		x_3	x_6				ar_3		r_2				qp
Move to AR Imm8	M30	0	s	x_3	x_2	x_4			ar_3		imm_{7b}				qp
Move from AR	M31	1		x_3	x_6				ar_3					r_1	qp
Alloc	M34	1		x_3		sor			sol		sof			r_1	0
Move to PSR	M35	1		x_3	x_6						r_2				qp
Move from PSR	M36	1		x_3	x_6									r_1	qp
Break/Nop	M37	0	i	x_3	x_2	x_4				imm_{20a}					qp
Mv from Ind	M43	1		x_3	x_6				r_3					r_1	qp
Set/Reset Mask	M44	0	i	x_3	i_{2d}	x_4			imm_{21a}						qp

表 C-4. 命令形式の要約 (続き)

Bit positions: 40 39 38 37 36 35 34 33 32 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 (fields listed left to right, from bit 40 to bit 0)

Instruction	Format										
IP-Relative Branch	B1	4	s	d	wh	imm_{20b}	p		btype	qp	
Counted Branch	B2	4	s	d	wh	imm_{20b}	p		btype	0	
IP-Relative Call	B3	5	s	d	wh	imm_{20b}	p		b_1	qp	
Indirect Branch	B4	0		d	wh	x_6		b_2	p		btype	qp
Indirect Call	B5	1		d	wh		b_2	p		b_1	qp
Misc	B8	0		x_6		0					
Break/Nop	B9	0/2	i		x_6		imm_{20a}	qp			
FP Arithmetic	F1	8 - D	x	sf	f_4	f_3	f_2	f_1	qp		
Fixed Multiply Add	F2	E	x	x_2	f_4	f_3	f_2	f_1	qp		
FP Select	F3	E	x		f_4	f_3	f_2	f_1	qp		
FP Compare	F4	4	r_b	sf	r_a	p_2	f_3	f_2	t_a	p_1	qp
FP Class	F5	5		fc_2	p_2	$fclass_{7c}$	f_2	t_a	p_1	qp	
FP Recip Approx	F6	0 - 1	q	sf	x	p_2	f_3	f_2	f_1	qp	
FP Recip Sqrt App	F7	0 - 1	q	sf	x	p_2	f_3		f_1	qp	
FP Min/Max/Pcmp	F8	0 - 1		sf	x	x_6	f_3	f_2	f_1	qp	
FP Merge/Logical	F9	0 - 1		x	x_6	f_3	f_2	f_1	qp		
Convert FP to Fixed	F10	0 - 1		sf	x	x_6		f_2	f_1	qp	
Convert Fixed to FP	F11	0		x	x_6		f_2	f_1	qp		
FP Set Controls	F12	0		sf	x	x_6	$omask_{7c}$	$amask_{7b}$		qp	
FP Clear Flags	F13	0		sf	x	x_6		qp			
FP Check Flags	F14	0	s	sf	x	x_6		imm_{20a}	qp		
Break/Nop	F15	0	i		x	x_6		imm_{20a}	qp		
Break/Nop	X1	0	i	x_3	x_6		imm_{20a}	qp	imm_{41}		
Move Imm_{64}	X2	6	i	imm_{9d}	imm_{5c}	i_c	v_c	imm_{7b}	r_1	qp	imm_{41}

表 C-5. 命令フィールドのカラー・キー

フィールドとカラー	
ALU 命令	オペコード拡張
整数命令	オペコード・ヒント拡張
メモリ命令	即値
分岐命令	間接ソース
浮動小数点命令	プレディケート・デスティネーション
整数ソース	整数デスティネーション
メモリ・ソース	メモリのソースとデスティネーション
シフト・ソース	シフト・即値
特殊レジスタ・ソース	特殊レジスタ・デスティネーション
浮動小数点ソース	浮動小数点デスティネーション
分岐ソース	分岐デスティネーション

表 C-5. 命令フィールドのカラー・キー (続き)

フィールドとカラー	
アドレス・ソース	予約済み命令
修飾プレディケート	PR[qp] が 1 の場合の予約済み命令
無視されるフィールド / 命令	

表 C-6. 命令フィールド名

フィールド名	説明
ar_3	アプリケーション・レジスタのソース / ターゲット
b_1, b_2	分岐レジスタのソース / ターゲット
btype	分岐タイプのオペコード拡張
c	補数比較関係のオペコード拡張
$ccount_{5c}$	マルチメディア向け左シフト補数シフトのカウントの即値
$count_{5b}$, $count_{6d}$	マルチメディア向け右シフト / ペア右シフトのカウントの即値
$cpos_x$	デポジットの補数ビット位置の即値
ct_{2d}	マルチメディア向け乗算シフト / シフトおよび加算のシフト・カウントの即値
d	分岐キャッシュ割り当て解除ヒントのオペコード拡張
f_n	浮動小数点レジスタのソース / ターゲット
fc_2, $fclass_{7c}$	浮動小数点クラスの即値
hint	メモリ参照ヒントのオペコード拡張
i, i_{2b}, i_{2d}, imm_x	長さ 1、2、または x の即値
len_{4d}, len_{6d}	抽出 / デポジットの長さの即値
m	メモリ参照での事後変更のオペコード拡張
$mask_x$	プレディケートの即値マスク
mbt_{4c}, mht_{8c}	マルチメディア向け mux1/mux2 の即値
p	シーケンシャル・プリフェッチ・ヒントのオペコード拡張
p_1, p_2	プレディケート・レジスタのターゲット
pos_{6b}	ビット・テスト / 抽出でのビット位置の即値
q	浮動小数点逆数 / 逆数平方根のオペコード拡張
qp	修飾プレディケート・レジスタのソース
r_n	汎用レジスタのソース / ターゲット
s	即値符号ビット
sf	浮動小数点ステータス・フィールドのオペコード拡張
sof, sol, sor	フレームの割り当てサイズ、ローカル領域のサイズ、ローテート領域のサイズの即値
t_a, t_b	比較タイプのオペコード拡張
v_x	予約済みオペコード拡張フィールド
wh	分岐有無ヒントのオペコード拡張
x, x_n	長さ 1 または n のオペコード拡張

表 C-6. 命令フィールド名 (続き)

フィールド名	説明
y	抽出 / デポジット / ビット・テスト / NaT テストのオペコード拡張
z_a, z_b	マルチメディア向けオペランド・サイズのオペコード拡張

表 C-7. 特殊な命令表記

表記	説明
f	命令は命令グループ内の最初の命令でなくてはならない
l	命令は命令グループ内の最後の命令でなくてはならない
t	命令は命令スロット 2 にしか入らない

本章の以降では、すべての命令のエンコーディングを詳しく説明する。まず「A ユニット命令エンコーディング」を示し、その後に、C-10 ページで「A ユニット命令エンコーディング」、C-32 ページで「M ユニット命令エンコーディング」、C-58 ページで「B ユニット命令エンコーディング」、C-64 ページで「F ユニット命令エンコーディング」、C-75 ページで「X ユニット命令エンコーディング」を示す。

個々の節の中では、命令を機能ごとにグループ化し、C-5 ページの表 C-4「命令形式の要約」で示したのと同じ順序でその命令形式を示している。オペコード拡張フィールドについても簡単に説明し、オペコード拡張割り当ても表に示している。使用されていない命令エンコーディング (オペコード拡張の表では空のエントリとして示している) は、次の 3 つのいずれかの動作をする。

- 無視される命令 (表の白のエントリ) は、nop 命令として実行される。
- 予約済みの命令 (表のグレー・スケール・バージョンでは薄灰色、カラー・バージョンでは茶色) は、無効操作フォルトを発生させる。
- PR[qp] が 1 の場合に予約済みとなる命令 (表のグレー・スケール・バージョンでは濃灰色、カラー・バージョンでは紫色) は、命令の qp フィールド (ビット 5:0) で指定されるプレディケート・レジスタが 1 である場合には無効操作フォルトを発生させ、0 の場合は nop 命令として実行される。

命令の定数 0 のフィールドは 0 でなくてはならず、それ以外の場合は未定義の操作が発生する。未定義の操作には、定数フィールドが 0 であるかどうかをチェックし、そうでない場合には無効操作フォルトを発生させるようなものもある。定数 0 のフィールドを持つ命令が修飾プレディケート (qp フィールド) も持っている場合、PR[qp] が 0 ならば、フォルトやその他の未定義の操作は発生しない。命令ビット 5:0(通常、qp に使用される) にある定数 0 のフィールドの場合、フォルトまたはその他の未定義の操作は、これらのビットがアドレス指定する PR に依存することもしないこともある。

命令の無視される (ホワイト・スペース) フィールドは 0 としてコーディングするべきである。アーキテクチャのこのリビジョンでは無視されるが、将来のアーキテクチャでこれらのフィールドをヒント拡張として使用する可能性がある。これらのヒント拡張は、各フィールドの 0 の値がデフォルト設定のヒントに対応するような形で定義される。アセンブラがデフォルトでこれらのフィールドを自動的にゼロに設定することが求められる。

C.2 A ユニット命令エンコーディング

C.2.1 整数 ALU

すべての整数 ALU 命令は、メジャー・オペコード 8 の中にエンコードされる。ビット 35:34(x_{2a}) に 2 ビットのオペコード拡張フィールドを持ち、ほとんどのものは、ビット 28:27(x_{2b}) に第 2 の 2 ビットのオペコード拡張フィールド、ビット 32:29(x_4) に 4 ビットのオペコード拡張フィールド、ビット 33(v_e) に 1 ビットの予約済みオペコード拡張フィールドを持つ。表 C-8 は、2 ビットの x_{2a} と 1 ビットの v_e の割り当てを示しており、表 C-9 は整数 ALU の 4 ビット +2 ビットの割り当てを示しており、C-20 ページの表 C-15 はマルチメディア ALU の 1 ビット +2 ビットの割り当てを示している (これもメジャー・オペコード 8 を共有する)。

表 C-8. 整数 ALU の 2 ビット +1 ビット・オペコード拡張

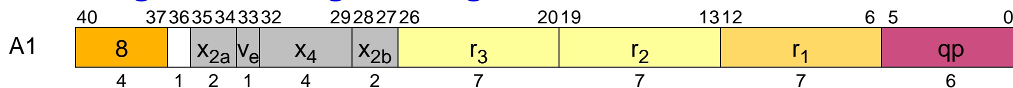
オペコード・ビット 40:377	x_{2a} ビット 35:34	Ve ビット 33	
		0	1
8	0	整数 ALU 4 ビット +2 ビット拡張 (表 C-9)	
	1	マルチメディア ALU 1 ビット +2 ビット拡張 (表 C-12)	
	2	adds – imm_{14} A4	
	3	addp4 – imm_{14} A4	

表 C-9. 整数 ALU の 4 ビット +2 ビット・オペコード拡張

オペコード・ビット 40:377	x_{2a} ビット 35:34	v_e ビット 33	x_4 ビット 32:29	x_{2b} ビット 28:27			
				0	1	2	3
8	0	0	0	add A1	add +1 A1		
			1	sub –1 A1	sub A1		
			2	addp4 A1			
			3	and A1	andcm A1	or A1	xor A1
			4	shladd A2			
			5				
			6	shladdp4 A2			
			7				
			8				
			9		sub – imm_8 A3		
			A				
			B	and – imm_8 A3	andcm – imm_8 A3	or – imm_8 A3	xor – imm_8 A3
			C				
			D				
			E				
			F				

C.2.1.1. Integer ALU - Register-Register

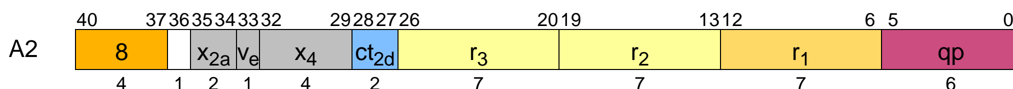
A1



命令	オペランド	オペコード	拡張			
			x_{2a}	v_e	x_4	x_{2b}
add	$r_1 = r_2, r_3$	8	0	0	0	0
	$r_1 = r_2, r_3$, 1					1
sub	$r_1 = r_2, r_3$				1	1
	$r_1 = r_2, r_3$, 1					0
addp4	$r_1 = r_2, r_3$				2	0
and					3	0
andcm						1
or						2
xor						3

C.2.1.2. Shift Left and Add

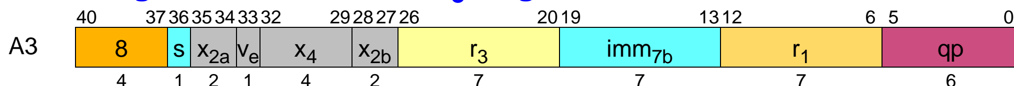
A2



命令	オペランド	オペコード	拡張		
			x_{2a}	v_e	x_4
shladd	$r_1 = r_2, count_2, r_3$	8	0	0	4
shladdp4					6

C.2.1.3. Integer ALU – Immediate$_8$-Register

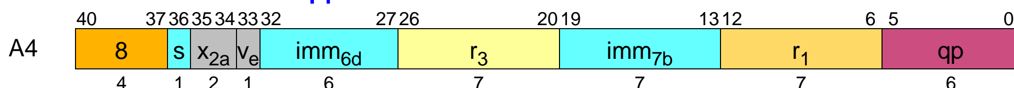
A3



命令	オペランド	オペコード	拡張			
			x_{2a}	v_e	x_4	x_{2b}
sub	$r_1 = imm_8, r_3$	8	0	0	9	1
and					B	0
andcm						1
or						2
xor						3

C.2.1.4. Add Immediate$_{14}$

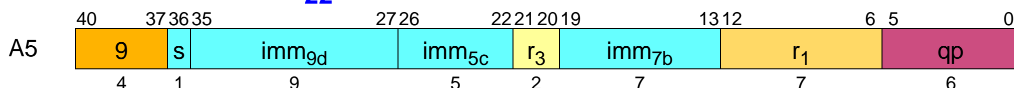
A4



命令	オペランド	オペコード	拡張	
			x_{2a}	v_e
adds	$r_1 = imm_{14}, r_3$	8	2	0
addp4			3	

C.2.1.5. Add Immediate$_{22}$

A5



命令	オペランド	オペコード
addl	$r_1 = imm_{22}, r_3$	9

C.2.2 整数比較

整数比較命令は、メジャー・オペコード C-E の中にエンコードされる。表 C-10 に示すように、ビット 35:34 に 2 ビットのオペコード拡張フィールド (x_2)、ビット 33(t_a)、36(t_b)、および 12(c) に 3 つの 1 ビット・オペコード拡張フィールドを使用する。整数比較即値命令は、メジャー・オペコード C-E の中にエンコードされ、表 C-11 に示すように、ビット 35:34 に 2 ビットのオペコード拡張フィールド (x_2)、ビット 33(t_a) と 12(c) に 2 つの 1 ビット・オペコード拡張フィールドを使用する。

表 C-10. 整数比較オペコード拡張

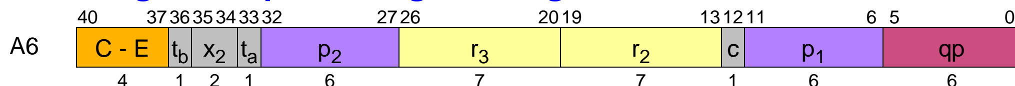
x_2 ビット 35:34	t_b ビット 36	t_a ビット 33	c ビット 12	オペコード・ビット 40:37		
				C	D	E
0	0	0	0	cmp.lt A6	cmp.ltu A6	cmp.eq A6
			1	cmp.lt.unc A6	cmp.ltu.unc A6	cmp.eq.unc A6
		1	0	cmp.eq.and A6	cmp.eq.or A6	cmp.eq.or.andcm A6
			1	cmp.ne.and A6	cmp.ne.or A6	cmp.ne.or.andcm A6
	1	0	0	cmp.gt.and A7	cmp.gt.or A7	cmp.gt.or.andcm A7
			1	cmp.le.and A7	cmp.le.or A7	cmp.le.or.andcm A7
		1	0	cmp.ge.and A7	cmp.ge.or A7	cmp.ge.or.andcm A7
			1	cmp.lt.and A7	cmp.lt.or A7	cmp.lt.or.andcm A7
1	0	0	0	cmp4.lt A6	cmp4.ltu A6	cmp4.eq A6
			1	cmp4.lt.unc A6	cmp4.ltu.unc A6	cmp4.eq.unc A6
		1	0	cmp4.eq.and A6	cmp4.eq.or A6	cmp4.eq.or.andcm A6
			1	cmp4.ne.and A6	cmp4.ne.or A6	cmp4.ne.or.andcm A6
	1	0	0	cmp4.gt.and A7	cmp4.gt.or A7	cmp4.gt.or.andcm A7
			1	cmp4.le.and A7	cmp4.le.or A7	cmp4.le.or.andcm A7
		1	0	cmp4.ge.and A7	cmp4.ge.or A7	cmp4.ge.or.andcm A7
			1	cmp4.lt.and A7	cmp4.lt.or A7	cmp4.lt.or.andcm A7

表 C-11. 整数比較即値オペコード拡張

x_2 ビット 35:34	t_a ビット 36	c ビット 12	オペコード・ビット 40:37		
			C	D	E
2	0	0	cmp.lt – imm$_8$ A8	cmp.ltu – imm$_8$ A8	cmp.eq – imm$_8$ A8
		1	cmp.lt.unc – imm$_8$ A8	cmp.ltu.unc – imm$_8$ A8	cmp.eq.unc – imm$_8$ A8
	1	0	cmp.eq.and – imm$_8$ A8	cmp.eq.or – imm$_8$ A8	cmp.eq.or.andcm – imm$_8$ A8
		1	cmp.ne.and – imm$_8$ A8	cmp.ne.or – imm$_8$ A8	cmp.ne.or.andcm – imm$_8$ A8
3	0	0	cmp4.lt – imm$_8$ A8	cmp4.ltu – imm$_8$ A8	cmp4.eq – imm$_8$ A8
		1	cmp4.lt.unc – imm$_8$ A8	cmp4.ltu.unc – imm$_8$ A8	cmp4.eq.unc – imm$_8$ A8
	1	0	cmp4.eq.and – imm$_8$ A8	cmp4.eq.or – imm$_8$ A8	cmp4.eq.or.andcm – imm$_8$ A8
		1	cmp4.ne.and – imm$_8$ A8	cmp4.ne.or – imm$_8$ A8	cmp4.ne.or.andcm – imm$_8$ A8

C.2.2.1. Integer Compare – Register-Register

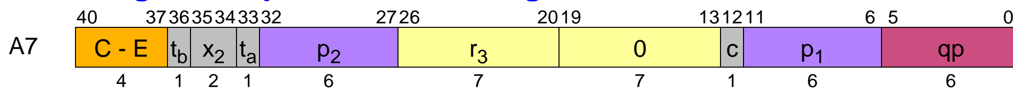
A6



命令	オペランド	オペコード	拡張			
			x_2	t_b	t_a	c
cmp.lt		C				
cmp.ltu		D				0
cmp.eq		E			0	
cmp.lt.unc		C				
cmp.ltu.unc		D				1
cmp.eq.unc		E	0	0		
cmp.eq.and		C				
cmp.eq.or		D				0
cmp.eq.or.andcm		E			1	
cmp.ne.and		C				
cmp.ne.or		D				1
cmp.ne.or.andcm	$p_1, p_2 = r_2, r_3$	E				
cmp4.lt		C				
cmp4.ltu		D				0
cmp4.eq		E			0	
cmp4.lt.unc		C				
cmp4.ltu.unc		D				1
cmp4.eq.unc		E	1	0		
cmp4.eq.and		C				
cmp4.eq.or		D				0
cmp4.eq.or.andcm		E			1	
cmp4.ne.and		C				
cmp4.ne.or		D				1
cmp4.ne.or.andcm		E				

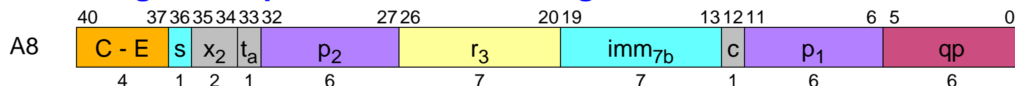
C.2.2.2. Integer Compare to Zero – Register

A7



<table>
<tr><th rowspan="2">命令</th><th rowspan="2">オペランド</th><th rowspan="2">オペコード</th><th colspan="4">拡張</th></tr>
<tr><th>x_2</th><th>t_b</th><th>t_a</th><th>c</th></tr>
<tr><td>cmp.gt.and</td><td rowspan="24">p_1, p_2 = r0, r_3</td><td>C</td><td rowspan="12">0</td><td rowspan="24">1</td><td rowspan="6">0</td><td rowspan="3">0</td></tr>
<tr><td>cmp.gt.or</td><td>D</td></tr>
<tr><td>cmp.gt.or.andcm</td><td>E</td></tr>
<tr><td>cmp.le.and</td><td>C</td><td rowspan="3">1</td></tr>
<tr><td>cmp.le.or</td><td>D</td></tr>
<tr><td>cmp.le.or.andcm</td><td>E</td></tr>
<tr><td>cmp.ge.and</td><td>C</td><td rowspan="6">1</td><td rowspan="3">0</td></tr>
<tr><td>cmp.ge.or</td><td>D</td></tr>
<tr><td>cmp.ge.or.andcm</td><td>E</td></tr>
<tr><td>cmp.lt.and</td><td>C</td><td rowspan="3">1</td></tr>
<tr><td>cmp.lt.or</td><td>D</td></tr>
<tr><td>cmp.lt.or.andcm</td><td>E</td></tr>
<tr><td>cmp4.gt.and</td><td>C</td><td rowspan="12">1</td><td rowspan="6">0</td><td rowspan="3">0</td></tr>
<tr><td>cmp4.gt.or</td><td>D</td></tr>
<tr><td>cmp4.gt.or.andcm</td><td>E</td></tr>
<tr><td>cmp4.le.and</td><td>C</td><td rowspan="3">1</td></tr>
<tr><td>cmp4.le.or</td><td>D</td></tr>
<tr><td>cmp4.le.or.andcm</td><td>E</td></tr>
<tr><td>cmp4.ge.and</td><td>C</td><td rowspan="6">1</td><td rowspan="3">0</td></tr>
<tr><td>cmp4.ge.or</td><td>D</td></tr>
<tr><td>cmp4.ge.or.andcm</td><td>E</td></tr>
<tr><td>cmp4.lt.and</td><td>C</td><td rowspan="3">1</td></tr>
<tr><td>cmp4.lt.or</td><td>D</td></tr>
<tr><td>cmp4.lt.or.andcm</td><td>E</td></tr>
</table>

C.2.2.3. Integer Compare – Immediate-Register



<table>
<tr><th rowspan="2">命令</th><th rowspan="2">オペランド</th><th rowspan="2">オペコード</th><th colspan="3">拡張</th></tr>
<tr><th>x_2</th><th>t_a</th><th>c</th></tr>
<tr><td>cmp.lt</td><td rowspan="24">$p_1, p_2 = imm_8, r_3$</td><td>C</td><td rowspan="12">2</td><td rowspan="6">0</td><td rowspan="3">0</td></tr>
<tr><td>cmp.ltu</td><td>D</td></tr>
<tr><td>cmp.eq</td><td>E</td></tr>
<tr><td>cmp.lt.unc</td><td>C</td><td rowspan="3">1</td></tr>
<tr><td>cmp.ltu.unc</td><td>D</td></tr>
<tr><td>cmp.eq.unc</td><td>E</td></tr>
<tr><td>cmp.eq.and</td><td>C</td><td rowspan="6">1</td><td rowspan="3">0</td></tr>
<tr><td>cmp.eq.or</td><td>D</td></tr>
<tr><td>cmp.eq.or.andcm</td><td>E</td></tr>
<tr><td>cmp.ne.and</td><td>C</td><td rowspan="3">1</td></tr>
<tr><td>cmp.ne.or</td><td>D</td></tr>
<tr><td>cmp.ne.or.andcm</td><td>E</td></tr>
<tr><td>cmp4.lt</td><td>C</td><td rowspan="12">3</td><td rowspan="6">0</td><td rowspan="3">0</td></tr>
<tr><td>cmp4.ltu</td><td>D</td></tr>
<tr><td>cmp4.eq</td><td>E</td></tr>
<tr><td>cmp4.lt.unc</td><td>C</td><td rowspan="3">1</td></tr>
<tr><td>cmp4.ltu.unc</td><td>D</td></tr>
<tr><td>cmp4.eq.unc</td><td>E</td></tr>
<tr><td>cmp4.eq.and</td><td>C</td><td rowspan="6">1</td><td rowspan="3">0</td></tr>
<tr><td>cmp4.eq.or</td><td>D</td></tr>
<tr><td>cmp4.eq.or.andcm</td><td>E</td></tr>
<tr><td>cmp4.ne.and</td><td>C</td><td rowspan="3">1</td></tr>
<tr><td>cmp4.ne.or</td><td>D</td></tr>
<tr><td>cmp4.ne.or.andcm</td><td>E</td></tr>
</table>

C.2.3 マルチメディア

すべてのマルチメディア ALU 命令は、メジャー・オペコード 8 の中にエンコードされる。表 C-12 に示すように、ビット 36(z_a) と 33(z_b) に 2 つの 1 ビット・オペコード拡張フィールド、ビット 35:34(x_{2a}) に 1 つの 2 ビット・オペコード拡張フィールドを使用する。また、マルチメディア ALU 命令は、C-18 ページの表 C-13 に示すように、ビット 32:29(x_4) に 4 ビットのオペコード拡張フィールドを、ビット 28:27(x_{2b}) に 2 ビットのオペコード拡張フィールドを持つ。

表 C-12. マルチメディア ALU の 2 ビット +1 ビット・オペコード拡張

オペコード・ビット 40:37	x_{2a} ビット 35:34	z_a ビット 36	z_b ビット 33	
8	1	0	0	マルチメディア ALU サイズ 1(表 C-13)
			1	マルチメディア ALU サイズ 2(表 C-14)
		1	0	マルチメディア ALU サイズ 3(表 C-15)
			1	

表 C-13. マルチメディア ALU サイズ 1 の 4 ビット +2 ビット・オペコード拡張

オペコード・ビット 40:37	x_{2a} ビット 35:34	z_a ビット 36	z_b ビット 33	x_4 ビット 32:29	x_{2b} ビット 28:27			
					0	1	2	3
8	1	0	0	0	padd1 A9	padd1.sss A9	padd1.uuu A9	padd1.uus A9
				1	psub1 A9	psub1.sss A9	psub1.uuu A9	psub1.uus A9
				2			pavg1 A9	pavg1.raz A9
				3			pavgsub1 A9	
				4				
				5				
				6				
				7				
				8				
				9	pcmp1.eq A9	pcmp1.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

表 C-14. マルチメディア ALU サイズ 2 の 4 ビット +2 ビット・オペコード拡張

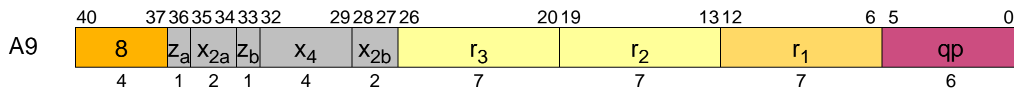
オペコード・ビット 40:37	x_{2a} ビット 35:34	z_a ビット 36	z_b ビット 33	x_4 ビット 32:29	x_{2b} ビット 28:27			
					0	1	2	3
8	1	0	1	0	padd2 A9	padd2.sss A9	padd2.uuu A9	padd2.uus A9
				1	psub2 A9	psub2.sss A9	psub2.uuu A9	psub2.uus A9
				2			pavg2 A9	pavg2.raz A9
				3			pavgsub2 A9	
				4	pshladd2 A10			
				5				
				6	pshradd2 A10			
				7				
				8				
				9	pcmp2.eq A9	pcmp2.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

表 C-15. マルチメディア ALU サイズ 4 の 4 ビット +2 ビット・オペコード拡張

オペコード・ビット 40:37	x_{2a} ビット 35:34	z_a ビット 36	z_b ビット 33	x_4 ビット 32:29	x_{2b} ビット 28:27			
					0	1	2	3
8	1	1	0	0	padd4 A9			
				1	psub4 A9			
				2				
				3				
				4				
				5				
				6				
				7				
				8				
				9	pcmp4.eq A9	pcmp4.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

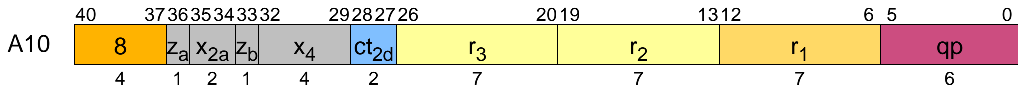
C.2.3.1. Multimedia ALU

A9



命令	オペランド	オペコード	拡張				
			x_{2a}	z_a	z_b	x_4	x_{2b}
padd1	$r_1 = r_2, r_3$	8	1	0	0	0	0
padd2					1		
padd4				1	0		
padd1.sss				0	0		1
padd2.sss					1		
padd1.uuu				0	0		2
padd2.uuu					1		
padd1.uus				0	0		3
padd2.uus					1		
psub1				0	0	1	0
psub2					1		
psub4				1	0		
psub1.sss				0	0		1
psub2.sss					1		
psub1.uuu				0	0		2
psub2.uuu					1		
psub1.uus				0	0		3
psub2.uus					1		
pavg1				0	0	2	2
pavg2					1		
pavg1.raz				0	0		3
pavg2.raz					1		
pavgsub1				0	0	3	2
pavgsub2					1		
pcmp1.eq				0	0	9	0
pcmp2.eq					1		
pcmp4.eq				1	0		
pcmp1.gt				0	0		1
pcmp2.gt					1		
pcmp4.gt				1	0		

C.2.3.2. Multimedia Shift and Add



命令	オペランド	オペコード	拡張			
			x_{2a}	z_a	z_b	x_4
pshladd2	$r_1 = r_2, count_2, r_3$	8	1	0	1	4
pshradd2						6

C.3 ユニット命令エンコーディング

C.3.1 マルチメディアおよび変数シフト

すべてのマルチメディア乗算 / シフト / 最大値 / 最小値 / ミックス / 置換 / パック / アンパック、および変数シフト命令は、メジャー・オペコード 7 の中にエンコーディングされる。表 C-16 に示すように、ビット 36(z_a) と 33(z_b) に 2 つの 1 ビット・オペコード拡張フィールド、ビット 32(v_e) に 1 ビットの予約済みオペコード拡張を使用する。また、表 C-17 に示すように、ビット 35:34(x_{2a}) に 2 ビットのオペコード拡張フィールド、ビット 29:28(x_{2b}) に 2 ビット・フィールドを持ち、ほとんどがビット 31:30(x_{2c}) に 2 ビット・フィールドを持つ。

表 C-16. マルチメディアおよび変数シフトの 1 ビット・オペコード拡張

オペコード・ビット 40:37	z_a ビット 36	z_b ビット 33	v_e ビット 32	
			0	1
7	0	0	マルチメディア・サイズ 1(表 C-17)	
		1	マルチメディア・サイズ 2(表 C-18)	
	1	0	マルチメディア・サイズ 3(表 C-19)	
		1	変数シフト (表 C-20)	

表 C-17. マルチメディア最大値 / 最小値 / ミックス / パック / アンパックのサイズ 1 の 2 ビット・オペコード拡張

<table>
<tr><th rowspan="2">オペコード・ビット 40:37</th><th rowspan="2">z_a ビット 36</th><th rowspan="2">z_b ビット 33</th><th rowspan="2">v_e ビット 32</th><th rowspan="2">x_{2a} ビット 35:34</th><th rowspan="2">x_{2b} ビット 29:28</th><th colspan="4">x_{2c} ビット 31:30</th></tr>
<tr><th>0</th><th>1</th><th>2</th><th>3</th></tr>
<tr><td rowspan="16">7</td><td rowspan="16">0</td><td rowspan="16">0</td><td rowspan="16">0</td><td rowspan="4">0</td><td>0</td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td></td><td></td><td></td><td></td></tr>
<tr><td rowspan="4">1</td><td>0</td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td></td><td></td><td></td><td></td></tr>
<tr><td rowspan="4">2</td><td>0</td><td></td><td>unpack1.h I2</td><td>mix1.r I2</td><td></td></tr>
<tr><td>1</td><td>pmin1.u I2</td><td>pmax1.u I2</td><td></td><td></td></tr>
<tr><td>2</td><td></td><td>unpack1.l I2</td><td>mix1.l I2</td><td></td></tr>
<tr><td>3</td><td></td><td></td><td>psad1 I2</td><td></td></tr>
<tr><td rowspan="4">3</td><td>0</td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td></td><td></td><td>mux1 I3</td><td></td></tr>
<tr><td>3</td><td></td><td></td><td></td><td></td></tr>
</table>

表 C-18. マルチメディア乗算 / シフト / 最大値 / 最小値 / ミックス / パック / アンパックのサイズ 2 の 2 ビット・オペコード拡張

オペコード・ビット 40:37	z_a ビット 36	z_b ビット 33	v_e ビット 32	x_{2a} ビット 35:34	x_{2b} ビット 29:28	x_{2c} ビット 31:30			
						0	**1**	**2**	**3**
7	0	1	0	0	0	pshr2.u – var I5	pshl2 – var I7		
					1	pmpyshr2.u I1			
					2	pshr2 – var I5			
					3	pmpyshr2 I1			
				1	0				
					1	pshr2.u – fixed I6		popcnt I9	
					2				
					3	pshr2 – fixed I6			
				2	0	pack2.uss I2	unpack2.h I2	mix2.r I2	
					1				pmpy2.r I2
					2	pack2.sss I2	unpack2.l I2	mix2.l I2	
					3	pmin2 I2	pmax2 I2		pmpy2.l I2
				3	0				
					1		pshl2 – fixed I8		
					2			mux2 I4	
					3				

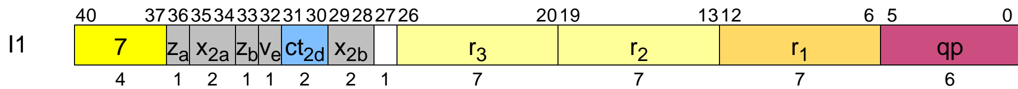
表 C-19. マルチメディア・シフト / ミックス / パック / アンパックのサイズ 4 の 2 ビット・オペコード拡張

オペコード ビット 40:37	z_a ビット 36	z_b ビット 33	v_e ビット 32	x_{2a} ビット 35:34	x_{2b} ビット 29:28	x_{2c} ビット 31:30			
						0	1	2	3
7	1	0	0	0	0	pshr4.u – var I5	pshl4 – var I7		
					1				
					2	pshr4 – var I5			
					3				
				1	0				
					1	pshr4.u – fixed I6			
					2				
					3	pshr4 – fixed I6			
				2	0		unpack4.h I2	mix4.r I2	
					1				
					2	pack4.sss I2	unpack4.l I2	mix4.l I2	
					3				
				3	0				
					1		pshl4 – fixed I8		
					2				
					3				

表 C-20. 変数シフトの 2 ビット・オペコード拡張

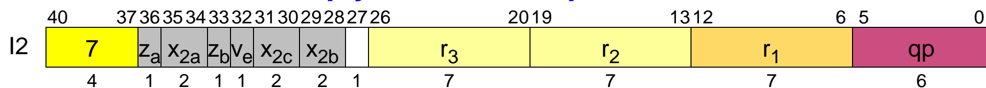
オペコード ビット 40:37	z_a ビット 36	z_b ビット 33	v_e ビット 32	x_{2a} ビット 35:34	x_{2b} ビット 29:28	x_{2c} ビット 31:30			
						0	1	2	3
7	1	1	0	0	0	shr.u – var I5	shl – var I7		
					1				
					2	shr – var I5			
					3				
				1	0				
					1				
					2				
					3				
				2	0				
					1				
					2				
					3				
				3	0				
					1				
					2				
					3				

C.3.1.1. Multimedia Multiply and Shift



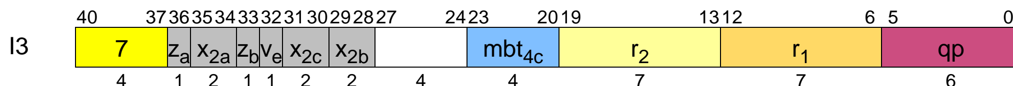
命令	オペランド	オペコード	拡張				
			z_a	z_b	v_e	x_{2a}	x_{2b}
pmpyshr2	$r_1 = r_2, r_3, count_2$	7	0	1	0	0	3
pmpyshr2.u							1

C.3.1.2. Multimedia Multiply/Mix/Pack/Unpack



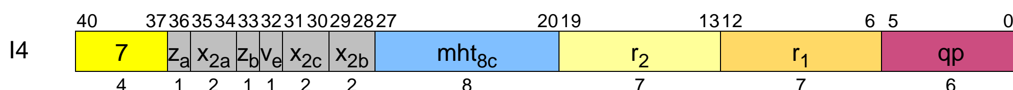
<table>
<tr><th rowspan="2">命令</th><th rowspan="2">オペランド</th><th rowspan="2">オペコード</th><th colspan="6">拡張</th></tr>
<tr><th>z_a</th><th>z_b</th><th>v_e</th><th>x_{2a}</th><th>x_{2b}</th><th>x_{2c}</th></tr>
<tr><td>pmpy2.r</td><td rowspan="22">$r_1 = r_2, r_3$</td><td rowspan="22">7</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="22">0</td><td rowspan="22">2</td><td>1</td><td rowspan="2">3</td></tr>
<tr><td>pmpy2.l</td><td>3</td></tr>
<tr><td>mix1.r</td><td>0</td><td>0</td><td rowspan="3">0</td><td rowspan="6">2</td></tr>
<tr><td>mix2.r</td><td>0</td><td>1</td></tr>
<tr><td>mix4.r</td><td>1</td><td>0</td></tr>
<tr><td>mix1.l</td><td>0</td><td>0</td><td rowspan="3">2</td></tr>
<tr><td>mix2.l</td><td>0</td><td>1</td></tr>
<tr><td>mix4.l</td><td>1</td><td>0</td></tr>
<tr><td>pack2.uss</td><td>0</td><td>1</td><td>0</td><td rowspan="3">0</td></tr>
<tr><td>pack2.sss</td><td>0</td><td>1</td><td rowspan="2">2</td></tr>
<tr><td>pack4.sss</td><td>1</td><td>0</td></tr>
<tr><td>unpack1.h</td><td>0</td><td>0</td><td rowspan="3">0</td><td rowspan="6">1</td></tr>
<tr><td>unpack2.h</td><td>0</td><td>1</td></tr>
<tr><td>unpack4.h</td><td>1</td><td>0</td></tr>
<tr><td>unpack1.l</td><td>0</td><td>0</td><td rowspan="3">2</td></tr>
<tr><td>unpack2.l</td><td>0</td><td>1</td></tr>
<tr><td>unpack4.l</td><td>1</td><td>0</td></tr>
<tr><td>pmin1.u</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td>0</td></tr>
<tr><td>pmax1.u</td><td>1</td></tr>
<tr><td>pmin2</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">3</td><td>0</td></tr>
<tr><td>pmax2</td><td>1</td></tr>
<tr><td>psad1</td><td>0</td><td>0</td><td>3</td><td>2</td></tr>
</table>

C.3.1.3. Multimedia Mux1



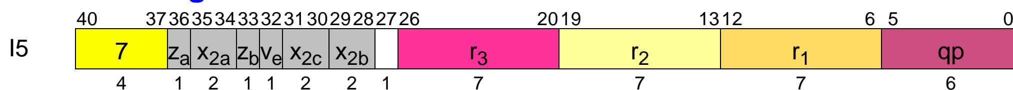
<table>
<tr><th rowspan="2">命令</th><th rowspan="2">オペランド</th><th rowspan="2">オペコード</th><th colspan="6">拡張</th></tr>
<tr><th>z_a</th><th>z_b</th><th>v_e</th><th>x_{2a}</th><th>x_{2b}</th><th>x_{2c}</th></tr>
<tr><td>mux1</td><td>$r_1 = r_2, mbtype_4$</td><td>7</td><td>0</td><td>0</td><td>0</td><td>3</td><td>2</td><td>2</td></tr>
</table>

C.3.1.4. Multimedia Mux2



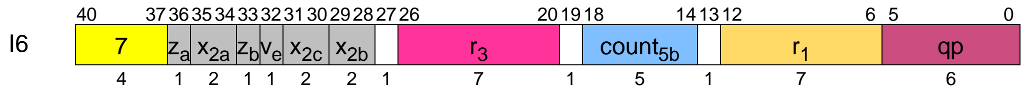
<table>
<tr><th rowspan="2">命令</th><th rowspan="2">オペランド</th><th rowspan="2">オペコード</th><th colspan="6">拡張</th></tr>
<tr><th>z_a</th><th>z_b</th><th>v_e</th><th>x_{2a}</th><th>x_{2b}</th><th>x_{2c}</th></tr>
<tr><td>mux2</td><td>$r_1 = r_2, mhtype_8$</td><td>7</td><td>0</td><td>1</td><td>0</td><td>3</td><td>2</td><td>2</td></tr>
</table>

C.3.1.5. Shift Right – Variable



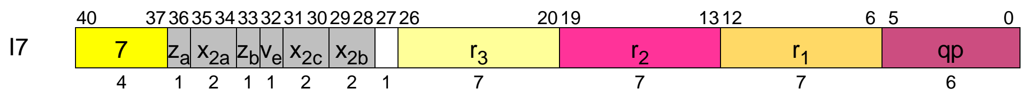
<table>
<tr><th rowspan="2">命令</th><th rowspan="2">オペランド</th><th rowspan="2">オペコード</th><th colspan="6">拡張</th></tr>
<tr><th>z_a</th><th>z_b</th><th>v_e</th><th>x_{2a}</th><th>x_{2b}</th><th>x_{2c}</th></tr>
<tr><td>pshr2</td><td rowspan="6">$r_1 = r_3, r_2$</td><td rowspan="6">7</td><td>0</td><td>1</td><td rowspan="6">0</td><td rowspan="6">0</td><td rowspan="3">2</td><td rowspan="6">0</td></tr>
<tr><td>pshr4</td><td>1</td><td>0</td></tr>
<tr><td>shr</td><td>1</td><td>1</td></tr>
<tr><td>pshr2.u</td><td>0</td><td>1</td><td rowspan="3">0</td></tr>
<tr><td>pshr4.u</td><td>1</td><td>0</td></tr>
<tr><td>shr.u</td><td>1</td><td>1</td></tr>
</table>

C.3.1.6. Multimedia Shift Right – Fixed



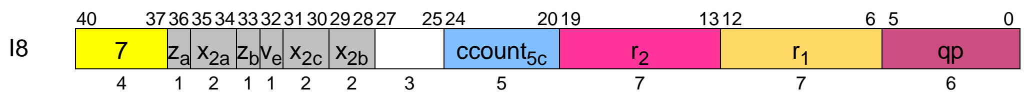
命令	オペランド	オペコード	拡張					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshr2	$r_1 = r_3, count_5$	7	0	1	0	1	3	0
pshr4			1	0				
pshr2.u			0	1			1	
pshr4.u			1	0				

C.3.1.7. Shift Left – Variable



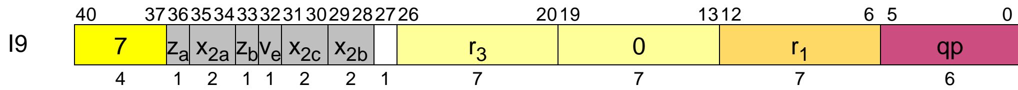
命令	オペランド	オペコード	拡張					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshl2	$r_1 = r_2, r_3$	7	0	1	0	0	0	1
pshl4			1	0				
shl			1	1				

C.3.1.8. Multimedia Shift Left – Fixed



命令	オペランド	オペコード	拡張					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshl2	$r_1 = r_2, count_5$	7	0	1	0	3	1	1
pshl4			1	0				

C.3.1.9. Population Count



命令	オペランド	オペコード	拡張					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
popcnt	$r_1 = r_3$	7	0	1	0	1	1	2

C.3.2 整数シフト

整数シフト、ビット・テスト、および NaT テスト命令は、メジャー・オペコード 5 の中にエンコードされる。ビット 35:34(x_2) に 2 ビットのオペコード拡張フィールド、ビット 33(x) に 1 ビットのオペコード拡張フィールドを使用する。また、抽出およびビット・テスト命令はビット 13(y) に 1 ビットのオペコード拡張フィールドを持つ。表 C-21 に、ビット・テスト、抽出、およびペア右シフトの割り当てを示す。

表 C-21. 整数シフト / ビット・テスト / NaT テストの 2 ビット・オペコード拡張

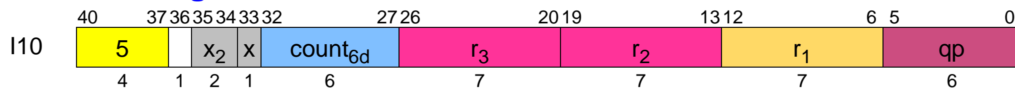
オペコード・ビット 40:37	x_2 ビット 35:34	x ビット 33	y ビット 13	
			0	1
5	0	0	Test Bit (表 C-23)	Test NaT (表 C-23)
	1		extr.u I11	extr I11
	2			
	3		shrp I10	

また、ほとんどのデポジット命令は、ビット 26(y) に 1 ビットのオペコード拡張フィールドを持つ。表 C-22 にこれらの割り当てを示す。

表 C-22. デポジットのオペコード拡張

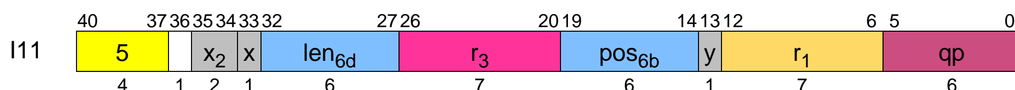
オペコード・ビット 40:37	x_2 ビット 35:34	x ビット 33	y ビット 26	
			0	1
5	0	1	ビット・テスト / テスト NaT(表 C-23)	
	1		dep.z I12	dep.z – imm_8 I13
	2			
	3		dep – imm_1 I14	

C.3.2.1. Shift Right Pair



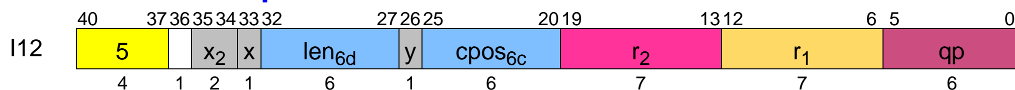
命令	オペランド	オペコード	拡張	
			x_2	x
shrp	$r_1 = r_2, r_3, count_6$	5	3	0

C.3.2.2. Extract



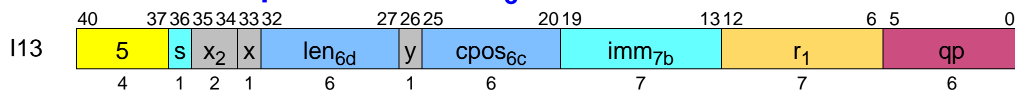
命令	オペランド	オペコード	拡張		
			x_2	x	y
extr.u	$r_1 = r_3, pos_6, len_6$	5	1	0	0
extr					1

C.3.2.3. Zero and Deposit



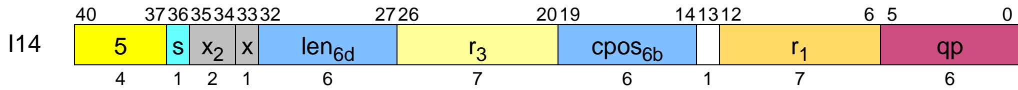
命令	オペランド	オペコード	拡張		
			x_2	x	y
dep.z	$r_1 = r_2, pos_6, len_6$	5	1	1	0

C.3.2.4. Zero and Deposit Immediate$_8$



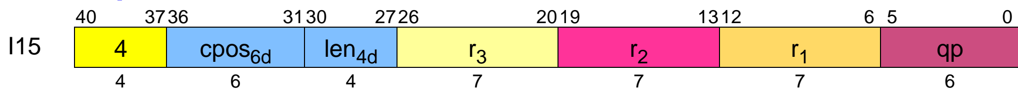
命令	オペランド	オペコード	拡張		
			x_2	x	y
dep.z	$r_1 = imm_8, pos_6, len_6$	5	1	1	1

C.3.2.5. Deposit Immediate$_1$



命令	オペランド	オペコード	拡張	
			x_2	x
dep	$r_1 = imm_1, r_3, pos_6, len_6$	5	3	1

C.3.2.6. Deposit



命令	オペランド	オペコード
dep	$r_1 = r_2, r_3, pos_6, len_4$	4

C.3.3 ビット・テスト

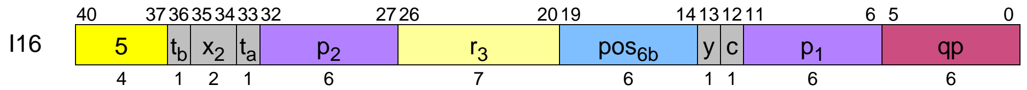
すべてのビット・テスト命令は、メジャー・オペコード 5 の中にエンコードされる。ビット 35:34(x_2) に 2 ビットのオペコード拡張フィールドを使用し、ビット 33(t_a)、36(t_b)、12(c)、および 19(y) に 4 つの 1 ビット・オペコード拡張フィールドを使用する。表 C-23 にこれらの割り当てを要約する。

表 C-23. ビット・テストのオペコード拡張

オペコード ビット 40:37	x_2 ビット 35:34	t_a ビット 33	t_b ビット 36	c ビット 12	y ビット 13	
					0	1
5	0	0	0	0	tbit.z I16	tnat.z I17
				1	tbit.z.unc I16	tnat.z.unc I17
			1	0	tbit.z.and I16	tnat.z.and I17
				1	tbit.nz.and I16	tnat.nz.and I17
		1	0	0	tbit.z.or I16	tnat.z.or I17
				1	tbit.nz.or I16	tnat.nz.or I17
			1	0	tbit.z.or.andcm I16	tnat.z.or.andcm I17
				1	tbit.nz.or.andcm I16	tnat.nz.or.andcm I17

C.3.3.1. Test Bit

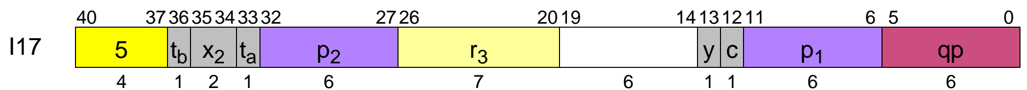
I16



命令	オペランド	オペコード	拡張				
			x_2	t_a	t_b	y	c
tbit.z	$p_1, p_2 = r_3, pos_6$	5	0	0	0	0	0
tbit.z.unc							1
tbit.z.and					1		0
tbit.nz.and							1
tbit.z.or				1	0		0
tbit.nz.or							1
tbit.z.or.andcm					1		0
tbit.nz.or.andcm							1

C.3.3.2. Test NaT

I17



命令	オペランド	オペコード	拡張				
			x_2	t_a	t_b	y	c
tnat.z	$p_1, p_2 = r_3$	5	0	0	0	1	0
tnat.z.unc							1
tnat.z.and					1		0
tnat.nz.and							1
tnat.z.or				1	0		0
tnat.nz.or							1
tnat.z.or.andcm					1		0
tnat.nz.or.andcm							1

C.3.4 その他の I ユニット命令

その他の I ユニット命令は、メジャー・オペコード 0 の中にエンコードされる。ビット 35:33 に、3 ビットのオペコード拡張フィールド (x_3) を使用する。また、一部の命令は、ビット 32:27 に 6 ビットのオペコード拡張フィールド (x_6) を持つ。表 C-24 に 3 ビットの割り当てを、表 C-25 に 6 ビットの割り当てを示す。

表 C-24. その他の I ユニットの 3 ビット・オペコード拡張

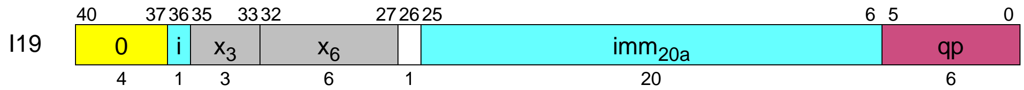
オペコード ビット 40:37	x_3 ビット 35:33	
0	0	6 ビット拡張 (表 C-25)
	1	chk.s.i – int I20
	2	mov to pr.rot – imm_{44} I24
	3	mov to pr I23
	4	
	5	
	6	
	7	mov to b

表 C-25. その他の I ユニットの 6 ビット・オペコード拡張

オペコード ビット 40:37	x_3 ビット 35:33	x_6				
		ビット 30:27	ビット 32:31			
			0	1	2	3
0	0	0	break.i I19	zxt1 I29		mov from ip I25
		1	nop.i I19	zxt2 I29		mov from b I22
		2		zxt4 I29		mov.i from ar I28
		3				mov from pr I25
		4		sxt1 I29		
		5		sxt2 I29		
		6		sxt4 I29		
		7				
		8		czx1.l I29		
		9		czx2.l I29		
		A	mov.i to ar – imm_8 I27		mov.i to ar I26	
		B				
		C		czx1.r I29		
		D		czx2.r I29		
		E				
		F				

C.3.4.1. Break/Nop (I ユニット)

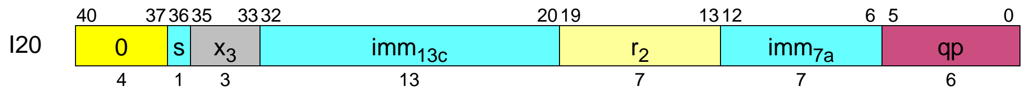
I19



命令	オペランド	オペコード	拡張	
			x_3	x_6
break.i	imm_{21}	0	0	00
nop.i				01

C.3.4.2. Integer Speculation Check (I ユニット)

I20



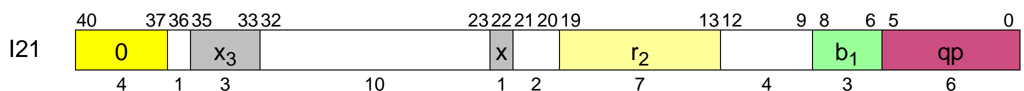
命令	オペランド	オペコード	拡張
			x_3
chk.s.i	r_2, $target_{25}$	0	1

C.3.5 GR/BR 移動

GR/BR 移動命令は、メジャー・オペコード 0 の中にエンコードされる。オペコード拡張の要約については、C-33 ページの「その他の I ユニット命令」を参照のこと。BR への移動命令は、通常の形式を返りの形式から区別するために、ビット 22 に 1 ビットのオペコード拡張フィールド (x) を使用する。

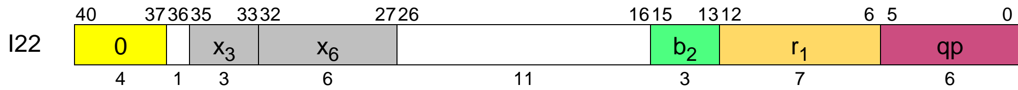
C.3.5.1. Move to BR

I21



命令	オペランド	オペコード	拡張	
			x_3	x
mov	$b_1 = r_2$	0	7	0
mov.ret				1

C.3.5.2. Move from BR

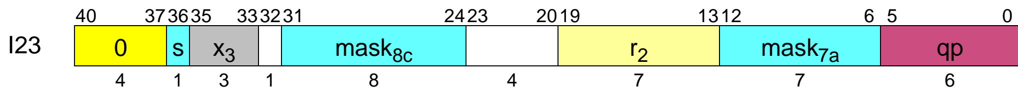


命令	オペランド	オペコード	拡張	
			x_3	x_6
mov	$r_1 = b_2$	0	0	31

C.3.6 GR/ プレディケート /IP 移動

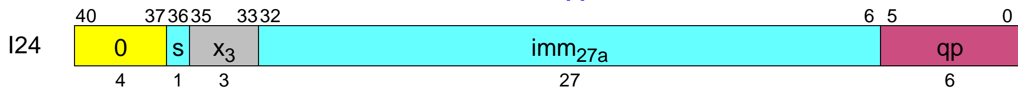
GR/ プレディケート /IP 移動命令は、メジャー・オペコード 0 の中にエンコードされる。オペコード拡張の要約については、C-33 ページの「その他の I ユニット命令」を参照のこと。

C.3.6.1. Move to Predicates – Register



命令	オペランド	オペコード	拡張
			x_3
mov	pr = r_2, $mask_{17}$	0	3

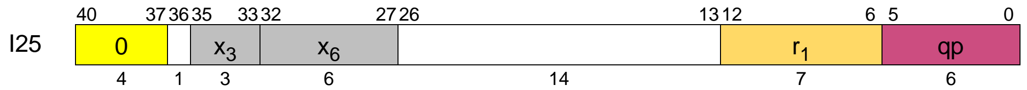
C.3.6.2. Move to Predicates – Immediate$_{44}$



命令	オペランド	オペコード	拡張
			x_3
mov	pr.rot = imm_{44}	0	2

C.3.6.3. Move from Predicates/IP

I25



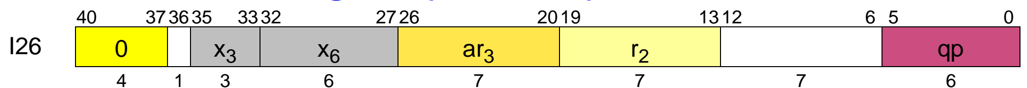
命令	オペランド	オペコード	拡張	
			x_3	x_6
mov	r_1 = ip	0	0	30
	r_1 = pr			33

C.3.7 GR/AR 移動 (I ユニット)

I ユニット GR/AR 移動命令は、メジャー・オペコード 0 の中にエンコードされる (一部の AR は、M ユニットのメモリ管理命令を使ってアクセスされる。C-37 ページの「GR/AR 移動 (I ユニット)」を参照のこと)。I ユニット GR/AR オペコード拡張の要約については、C-27 ページの「その他の I ユニット命令」を参照のこと。

C.3.7.1. Move to AR – Register (I ユニット)

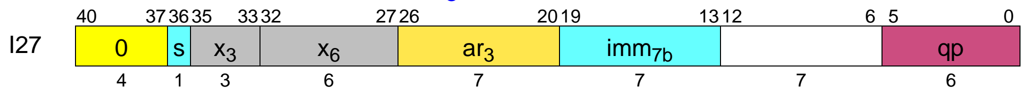
I26



命令	オペランド	オペコード	拡張	
			x_3	x_6
mov.i	$ar_3 = r_2$	0	0	2A

C.3.7.2. Move to AR – Immediate$_8$ (I ユニット)

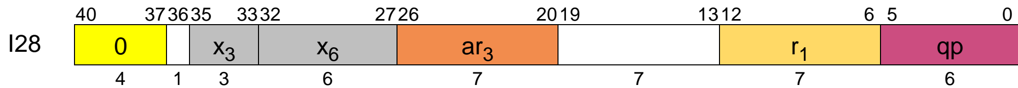
I27



命令	オペランド	オペコード	拡張	
			x_3	x_6
mov.i	$ar_3 = imm_8$	0	0	0A

C.3.7.3. Move from AR (I ユニット)

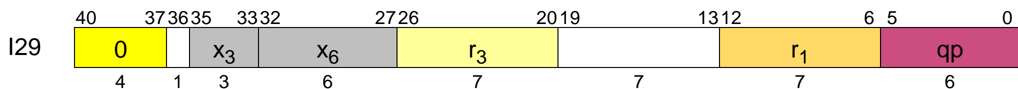
I28



命令	オペランド	オペコード	拡張	
			x_3	x_6
mov.i	$r_1 = ar_3$	0	0	32

C.3.8 符号拡張 / ゼロ拡張 / ゼロ・インデックス計算

I29



命令	オペランド	オペコード	拡張	
			x_3	x_6
zxt1	$r_1 = r_3$	0	0	10
zxt2				11
zxt4				12
sxt1				14
sxt2				15
sxt4				16
czx1.l				18
czx2.l				19
czx1.r				1C
czx2.r				1D

C.4 M ユニット命令エンコーディング

C.4.1 ロードとストア

すべてのロードおよびストア命令は、メジャー・オペコード 4、5、6、および 7 の中にエンコードされる。ビット 35:30(x_6) の 6 ビットのオペコード拡張フィールドが使用される。メジャー・オペコード 4 の命令 (整数ロード / ストア、セマフォ、および FR 取得) は、表 C-26 に示すように、ビット 36(m) とビット 27(x) に 2 つの 1 ビット・オペコード拡張フィールドを使用する。メジャー・オペコード 6 の命令 (浮動小数点ロード / ストア、ペア・ロード、および FR 設定) は、表 C-27 に示すように、ビット 36(m) とビット 27(x) に 2 つの 1 ビット・オペコード拡張フィールドを使用する。

表 C-26. 整数ロード / ストア / セマフォ /FR 取得の 1 ビット・オペコード拡張

オペコード ビット 40:37	m ビット 36	x ビット 27	
4	0	0	ロード / ストア (表 C-28)
	0	1	セマフォ /FR 取得 (表 C-31)
	1	0	ロード +Reg(表 C-29)
	1	1	

表 C-27. 浮動小数点ロード / ストア / ペア・ロード /FR 設定の 1 ビット・オペコード拡張

オペコード ビット 40:37	m ビット 36	x ビット 27	
6	0	0	FP ロード / ストア (表 C-32)
	0	1	FP ペア・ロード /FR 設定 (表 C-35)
	1	0	FP ロード +Reg(表 C-33)
	1	1	FP ペア・ロード +Imm(表 C-35)

整数ロード / ストアのオペコード拡張は、C-40 ページの表 C-28、C-41 ページの表 C-29、および C-42 ページの表 C-30 に要約している。セマフォおよび FR 取得オペコード拡張は、C-43 ページの表 C-31 に要約している。浮動小数点ロード / ストア・オペコード拡張は、C-44 ページの表 C-32、C-45 ページの表 C-33、および C-46 ページの表 C-34 に要約している。浮動小数点ペア・ロードおよび FR 設定オペコード拡張は、C-47 ページの表 C-35 と C-48 ページの表 C-36 に要約している。

表 C-28. 整数ロード / ストアのオペコード拡張

オペコードビット 40:37	m ビット 36	x ビット 27	x_6				
			ビット 35:32	ビット 31:30			
				0	1	2	3
4	0	0	0	ld1 M1	ld2 M1	ld4 M1	ld8 M1
			1	ld1.s M1	ld2.s M1	ld4.s M1	ld8.s M1
			2	ld1.a M1	ld2.a M1	ld4.a M1	ld8.a M1
			3	ld1.sa M1	ld2.sa M1	ld4.sa M1	ld8.sa M1
			4	ld1.bias M1	ld2.bias M1	ld4.bias M1	ld8.bias M1
			5	ld1.acq M1	ld2.acq M1	ld4.acq M1	ld8.acq M1
			6				ld8.fill M1
			7				
			8	ld1.c.clr M1	ld2.c.clr M1	ld4.c.clr M1	ld8.c.clr M1
			9	ld1.c.nc M1	ld2.c.nc M1	ld4.c.nc M1	ld8.c.nc M1
			A	ld1.c.clr.acq M1	ld2.c.clr.acq M1	ld4.c.clr.acq M1	ld8.c.clr.acq M1
			B				
			C	st1 M4	st2 M4	st4 M4	st8 M4
			D	st1.rel M4	st2.rel M4	st4.rel M4	st8.rel M4
			E				st8.spill M4
			F				

表 C-29. 整数ロード +Reg のオペコード拡張

オペコードビット 40:37	m ビット 36	x ビット 27	x_6				
			ビット 35:32	ビット 31:30			
				0	1	2	3
4	1	0	0	ld1 M2	ld2 M2	ld4 M2	ld8 M2
			1	ld1.s M2	ld2.s M2	ld4.s M2	ld8.s M2
			2	ld1.a M2	ld2.a M2	ld4.a M2	ld8.a M2
			3	ld1.sa M2	ld2.sa M2	ld4.sa M2	ld8.sa M2
			4	ld1.bias M2	ld2.bias M2	ld4.bias M2	ld8.bias M2
			5	ld1.acq M2	ld2.acq M2	ld4.acq M2	ld8.acq M2
			6				ld8.fill M2
			7				
			8	ld1.c.clr M2	ld2.c.clr M2	ld4.c.clr M2	ld8.c.clr M2
			9	ld1.c.nc M2	ld2.c.nc M2	ld4.c.nc M2	ld8.c.nc M2
			A	ld1.c.clr.acq M2	ld2.c.clr.acq M2	ld4.c.clr.acq M2	ld8.c.clr.acq M2
			B				
			C				
			D				
			E				
			F				

表 C-30. 整数ロード / ストア +Imm のオペコード拡張

オペコード ビット 40:37	x_6				
	ビット 35:32	ビット 31:30			
		0	1	2	3
5	0	ld1 M3	ld2 M3	ld4 M3	ld8 M3
	1	ld1.s M3	ld2.s M3	ld4.s M3	ld8.s M3
	2	ld1.a M3	ld2.a M3	ld4.a M3	ld8.a M3
	3	ld1.sa M3	ld2.sa M3	ld4.sa M3	ld8.sa M3
	4	ld1.bias M3	ld2.bias M3	ld4.bias M3	ld8.bias M3
	5	ld1.acq M3	ld2.acq M3	ld4.acq M3	ld8.acq M3
	6				ld8.fill M3
	7				
	8	ld1.c.clr M3	ld2.c.clr M3	ld4.c.clr M3	ld8.c.clr M3
	9	ld1.c.nc M3	ld2.c.nc M3	ld4.c.nc M3	ld8.c.nc M3
	A	ld1.c.clr.acq M3	ld2.c.clr.acq M3	ld4.c.clr.acq M3	ld8.c.clr.acq M3
	B				
	C	st1 M5	st2 M5	st4 M5	st8 M5
	D	st1.rel M5	st2.rel M5	st4.rel M5	st8.rel M5
	E				st8.spill M5
	F				

表 C-31. セマフォ /FR 取得のオペコード拡張

オペコードビット 40:37	m ビット 36	x ビット 27	x6				
			ビット 35:32	ビット 31:30			
				0	1	2	3
4	0	1	0	cmpxchg1.acq M16	cmpxchg2.acq M16	cmpxchg4.acq M16	cmpxchg8.acq M16
			1	cmpxchg1.rel M16	cmpxchg2.rel M16	cmpxchg4.rel M16	cmpxchg8.rel M16
			2	xchg1 M16	xchg2 M16	xchg4 M16	xchg8 M16
			3				
			4			fetchadd4.acq M17	fetchadd8.acq M17
			5			fetchadd4.rel M17	fetchadd8.rel M17
			6				
			7	getf.sig M19	getf.exp M19	getf.s M19	getf.d M19
			8				
			9				
			A				
			B				
			C				
			D				
			E				
			F				

表 C-32. 浮動小数点ロード / ストア /Lfetch のオペコード拡張

オペコード ビット 40:37	m ビット 36	x ビット 27	x_6				
			ビット 35:32	ビット 31:30			
				0	1	2	3
6	0	0	0	ldfe M6	ldf8 M6	ldfs M6	ldfd M6
			1	ldfe.s M6	ldf8.s M6	ldfs.s M6	ldfd.s M6
			2	ldfe.a M6	ldf8.a M6	ldfs.a M6	ldfd.a M6
			3	ldfe.sa M6	ldf8.sa M6	ldfs.sa M6	ldfd.sa M6
			4				
			5				
			6				ldf.fill M6
			7				
			8	ldfe.c.clr M6	ldf8.c.clr M6	ldfs.c.clr M6	ldfd.c.clr M6
			9	ldfe.c.nc M6	ldf8.c.nc M6	ldfs.c.nc M6	ldfd.c.nc M6
			A				
			B	lfetch M13	lfetch.excl M13	lfetch.fault M13	lfetch.fault.excl M13
			C	stfe M9	stf8 M9	stfs M9	stfd M9
			D				
			E				stf.spill M9
			F				

表 C-33. 浮動小数点ロード /Lfetch+Reg のオペコード拡張

オペコード ビット 40:37	m ビット 36	x ビット 27	x_6				
			ビット 35:32	ビット 31:30			
				0	1	2	3
6	1	0	0	ldfe M7	ldf8 M7	ldfs M7	ldfd M7
			1	ldfe.s M7	ldf8.s M7	ldfs.s M7	ldfd.s M7
			2	ldfe.a M7	ldf8.a M7	ldfs.a M7	ldfd.a M7
			3	ldfe.sa M7	ldf8.sa M7	ldfs.sa M7	ldfd.sa M7
			4				
			5				
			6				ldf.fill M7
			7				
			8	ldfe.c.clr M7	ldf8.c.clr M7	ldfs.c.clr M7	ldfd.c.clr M7
			9	ldfe.c.nc M7	ldf8.c.nc M7	ldfs.c.nc M7	ldfd.c.nc M7
			A				
			B	lfetch M14	lfetch.excl M14	lfetch.fault M14	lfetch.fault.excl M14
			C				
			D				
			E				
			F				

表 C-34. 浮動小数点ロード / ストア /Lfetch+Imm のオペコード拡張

オペコードビット 40:37	x6				
	ビット 35:32	ビット 31:30			
		0	1	2	3
7	0	ldfe M8	ldf8 M8	ldfs M8	ldfd M8
	1	ldfe.s M8	ldf8.s M8	ldfs.s M8	ldfd.s M8
	2	ldfe.a M8	ldf8.a M8	ldfs.a M8	ldfd.a M8
	3	ldfe.sa M8	ldf8.sa M8	ldfs.sa M8	ldfd.sa M8
	4				
	5				
	6				ldf.fill M8
	7				
	8	ldfe.c.clr M8	ldf8.c.clr M8	ldfs.c.clr M8	ldfd.c.clr M8
	9	ldfe.c.nc M8	ldf8.c.nc M8	ldfs.c.nc M8	ldfd.c.nc M8
	A				
	B	lfetch M15	lfetch.excl M15	lfetch.fault M15	lfetch.fault.excl M15
	C	stfe M10	stf8 M10	stfs M10	stfd M10
	D				
	E				stf.spill M10
	F				

表 C-35. 浮動小数点ペア・ロード /SR 設定のオペコード拡張

オペコードビット 40:37	m ビット 36	x ビット 27	x_6				
			ビット 35:32	ビット 31:30			
				0	1	2	3
6	0	1	0		ldfp8 M11	ldfps M11	ldfpd M11
			1		ldfp8.s M11	ldfps.s M11	ldfpd.s M11
			2		ldfp8.a M11	ldfps.a M11	ldfpd.a M11
			3		ldfp8.sa M11	ldfps.sa M11	ldfpd.sa M11
			4				
			5				
			6				
			7	setf.sig M18	setf.exp M18	setf.s M18	setf.d M18
			8		ldfp8.c.clr M11	ldfps.c.clr M11	ldfpd.c.clr M11
			9		ldfp8.c.nc M11	ldfps.c.nc M11	ldfpd.c.nc M11
			A				
			B				
			C				
			D				
			E				
			F				

表 C-36. 浮動小数点ペア・ロード +Imm のオペコード拡張

オペコード ビット 40:37	m ビット 36	x ビット 27	x6				
			ビット 35:32	ビット 31:30			
				0	1	2	3
6	1	1	0		ldfp8 M12	ldfps M12	ldfpd M12
			1		ldfp8.s M12	ldfps.s M12	ldfpd.s M12
			2		ldfp8.a M12	ldfps.a M12	ldfpd.a M12
			3		ldfp8.sa M12	ldfps.sa M12	ldfpd.sa M12
			4				
			5				
			6				
			7				
			8		ldfp8.c.clr M12	ldfps.c.clr M12	ldfpd.c.clr M12
			9		ldfp8.c.nc M12	ldfps.c.nc M12	ldfpd.c.nc M12
			A				
			B				
			C				
			D				
			E				
			F				

ロードおよびストア命令は、いずれもビット 29:28(hint) に、局所性ヒント情報をエンコードしている 2 ビットのオペコード拡張フィールドを持っている。表 C-37 と表 C-38 にこれらの割り当てを要約する。

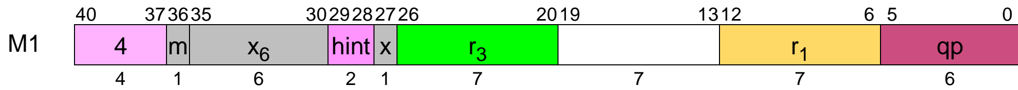
表 C-37. ロード・ヒント・コンプリータ

ヒント・ビット 29:28	*ldhint*
0	*none*
1	.nt1
2	
3	.nta

表 C-38. ストア・ヒント・コンプリータ

ヒント・ビット 29:28	*sthint*
0	*none*
1	
2	
3	.nta

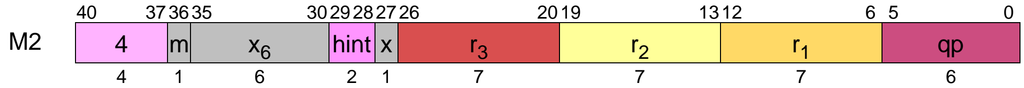
C.4.1.1. Integer Load



命令	オペランド	オペコード	拡張			
			m	x	x_6	ヒント
ld1.*ldhint*	$r_1 = [r_3]$	4	0	0	00	C-49 ページの表 C-37
ld2.*ldhint*					01	
ld4.*ldhint*					02	
ld8.*ldhint*					03	
ld1.s.*ldhint*					04	
ld2.s.*ldhint*					05	
ld4.s.*ldhint*					06	
ld8.s.*ldhint*					07	
ld1.a.*ldhint*					08	
ld2.a.*ldhint*					09	
ld4.a.*ldhint*					0A	
ld8.a.*ldhint*					0B	
ld1.sa.*ldhint*					0C	
ld2.sa.*ldhint*					0D	
ld4.sa.*ldhint*					0E	
ld8.sa.*ldhint*					0F	
ld1.bias.*ldhint*					10	
ld2.bias.*ldhint*					11	
ld4.bias.*ldhint*					12	
ld8.bias.*ldhint*					13	
ld1.acq.*ldhint*					14	
ld2.acq.*ldhint*					15	
ld4.acq.*ldhint*					16	
ld8.acq.*ldhint*					17	
ld8.fill.*ldhint*					1B	
ld1.c.clr.*ldhint*					20	
ld2.c.clr.*ldhint*					21	
ld4.c.clr.*ldhint*					22	
ld8.c.clr.*ldhint*					23	
ld1.c.nc.*ldhint*					24	
ld2.c.nc.*ldhint*					25	
ld4.c.nc.*ldhint*					26	
ld8.c.nc.*ldhint*					27	
ld1.c.clr.acq.*ldhint*					28	
ld2.c.clr.acq.*ldhint*					29	
ld4.c.clr.acq.*ldhint*					2A	
ld8.c.clr.acq.*ldhint*					2B	

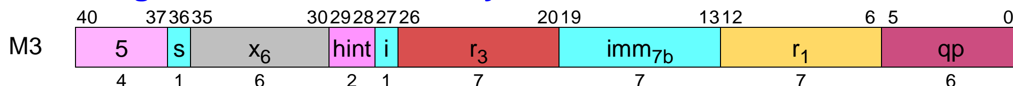
C.4.1.2. Integer Load – Increment by Register

M2



命令	オペランド	オペコード	拡張			
			m	x	x_6	hint
ld1.*ldhint*	r_1 = [r_3], r_2	4	1	0	00	C-49 ページの表 C-37
ld2.*ldhint*					01	
ld4.*ldhint*					02	
ld8.*ldhint*					03	
ld1.s.*ldhint*					04	
ld2.s.*ldhint*					05	
ld4.s.*ldhint*					06	
ld8.s.*ldhint*					07	
ld1.a.*ldhint*					08	
ld2.a.*ldhint*					09	
ld4.a.*ldhint*					0A	
ld8.a.*ldhint*					0B	
ld1.sa.*ldhint*					0C	
ld2.sa.*ldhint*					0D	
ld4.sa.*ldhint*					0E	
ld8.sa.*ldhint*					0F	
ld1.bias.*ldhint*					10	
ld2.bias.*ldhint*					11	
ld4.bias.*ldhint*					12	
ld8.bias.*ldhint*					13	
ld1.acq.*ldhint*					14	
ld2.acq.*ldhint*					15	
ld4.acq.*ldhint*					16	
ld8.acq.*ldhint*					17	
ld8.fill.*ldhint*					1B	
ld1.c.clr.*ldhint*					20	
ld2.c.clr.*ldhint*					21	
ld4.c.clr.*ldhint*					22	
ld8.c.clr.*ldhint*					23	
ld1.c.nc.*ldhint*					24	
ld2.c.nc.*ldhint*					25	
ld4.c.nc.*ldhint*					26	
ld8.c.nc.*ldhint*					27	
ld1.c.clr.acq.*ldhint*					28	
ld2.c.clr.acq.*ldhint*					29	
ld4.c.clr.acq.*ldhint*					2A	
ld8.c.clr.acq.*ldhint*					2B	

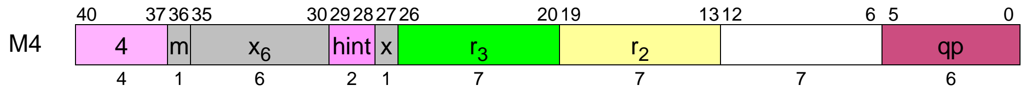
C.4.1.3. Integer Load – Increment by Immediate



命令	オペランド	オペコード	拡張	
			x_6	ヒント
ld1.*ldhint*	$r_1 = [r_3], imm_9$	5	00	C-49 ページの表 C-37
ld2.*ldhint*			01	
ld4.*ldhint*			02	
ld8.*ldhint*			03	
ld1.s.*ldhint*			04	
ld2.s.*ldhint*			05	
ld4.s.*ldhint*			06	
ld8.s.*ldhint*			07	
ld1.a.*ldhint*			08	
ld2.a.*ldhint*			09	
ld4.a.*ldhint*			0A	
ld8.a.*ldhint*			0B	
ld1.sa.*ldhint*			0C	
ld2.sa.*ldhint*			0D	
ld4.sa.*ldhint*			0E	
ld8.sa.*ldhint*			0F	
ld1.bias.*ldhint*			10	
ld2.bias.*ldhint*			11	
ld4.bias.*ldhint*			12	
ld8.bias.*ldhint*			13	
ld1.acq.*ldhint*			14	
ld2.acq.*ldhint*			15	
ld4.acq.*ldhint*			16	
ld8.acq.*ldhint*			17	
ld8.fill.*ldhint*			1B	
ld1.c.clr.*ldhint*			20	
ld2.c.clr.*ldhint*			21	
ld4.c.clr.*ldhint*			22	
ld8.c.clr.*ldhint*			23	
ld1.c.nc.*ldhint*			24	
ld2.c.nc.*ldhint*			25	
ld4.c.nc.*ldhint*			26	
ld8.c.nc.*ldhint*			27	
ld1.c.clr.acq.*ldhint*			28	
ld2.c.clr.acq.*ldhint*			29	
ld4.c.clr.acq.*ldhint*			2A	
ld8.c.clr.acq.*ldhint*			2B	

C.4.1.4. Integer Store

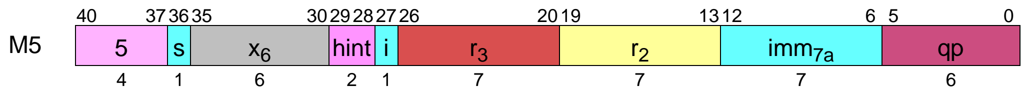
M4



命令	オペランド	オペコード	拡張			
			m	x	x_6	ヒント
st1.*sthint*	$[r_3] = r_2$	4	0	0	30	C-49 ページの表 C-37
st2.*sthint*					31	
st4.*sthint*					32	
st8.*sthint*					33	
st1.rel.*sthint*					34	
st2.rel.*sthint*					35	
st4.rel.*sthint*					36	
st8.rel.*sthint*					37	
st8.spill.*sthint*					3B	

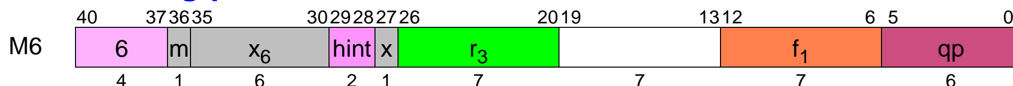
C.4.1.5. Integer Store – Increment by Immediate

M5



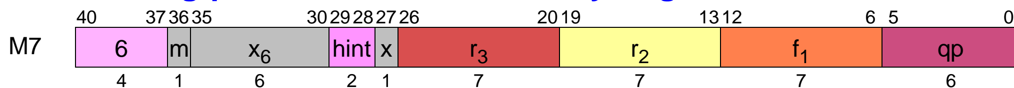
命令	オペランド	オペコード	拡張	
			x_6	ヒント
st1.*sthint*	$[r_3] = r_2, imm_9$	5	30	C-49 ページの表 C-37
st2.*sthint*			31	
st4.*sthint*			32	
st8.*sthint*			33	
st1.rel.*sthint*			34	
st2.rel.*sthint*			35	
st4.rel.*sthint*			36	
st8.rel.*sthint*			37	
st8.spill.*sthint*			3B	

C.4.1.6. Floating-point Load



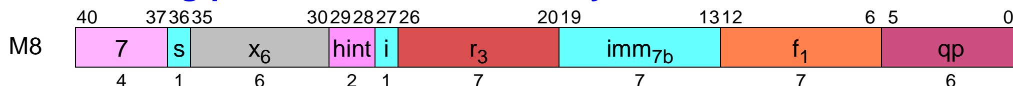
命令	オペランド	オペコード	拡張			
			m	x	x_6	ヒント
ldfs.*ldhint*	$f_1 = [r_3]$	6	0	0	02	C-49 ページの表 C-37
ldfd.*ldhint*					03	
ldf8.*ldhint*					01	
ldfe.*ldhint*					00	
ldfs.s.*ldhint*					06	
ldfd.s.*ldhint*					07	
ldf8.s.*ldhint*					05	
ldfe.s.*ldhint*					04	
ldfs.a.*ldhint*					0A	
ldfd.a.*ldhint*					0B	
ldf8.a.*ldhint*					09	
ldfe.a.*ldhint*					08	
ldfs.sa.*ldhint*					0E	
ldfd.sa.*ldhint*					0F	
ldf8.sa.*ldhint*					0D	
ldfe.sa.*ldhint*					0C	
ldf.fill.*ldhint*					1B	
ldfs.c.clr.*ldhint*					22	
ldfd.c.clr.*ldhint*					23	
ldf8.c.clr.*ldhint*					21	
ldfe.c.clr.*ldhint*					20	
ldfs.c.nc.*ldhint*					26	
ldfd.c.nc.*ldhint*					27	
ldf8.c.nc.*ldhint*					25	
ldfe.c.nc.*ldhint*					24	

C.4.1.7. Floating-point Load – Increment by Register



命令	オペランド	オペコード	拡張			
			m	x	x_6	ヒント
ldfs.*ldhint*	$f_1 = [r_3], r_2$	6	1	0	02	C-49 ページの表 C-37
ldfd.*ldhint*					03	
ldf8.*ldhint*					01	
ldfe.*ldhint*					00	
ldfs.s.*ldhint*					06	
ldfd.s.*ldhint*					07	
ldf8.s.*ldhint*					05	
ldfe.s.*ldhint*					04	
ldfs.a.*ldhint*					0A	
ldfd.a.*ldhint*					0B	
ldf8.a.*ldhint*					09	
ldfe.a.*ldhint*					08	
ldfs.sa.*ldhint*					0E	
ldfd.sa.*ldhint*					0F	
ldf8.sa.*ldhint*					0D	
ldfe.sa.*ldhint*					0C	
ldf.fill.*ldhint*					1B	
ldfs.c.clr.*ldhint*					22	
ldfd.c.clr.*ldhint*					23	
ldf8.c.clr.*ldhint*					21	
ldfe.c.clr.*ldhint*					20	
ldfs.c.nc.*ldhint*					26	
ldfd.c.nc.*ldhint*					27	
ldf8.c.nc.*ldhint*					25	
ldfe.c.nc.*ldhint*					24	

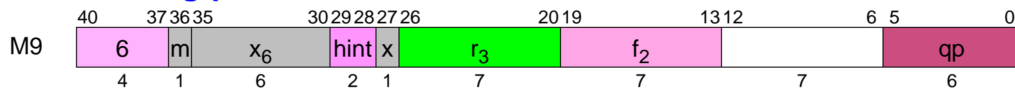
C.4.1.8. Floating-point Load – Increment by Immediate



命令	オペランド	オペコード	拡張	
			x_6	ヒント
ldfs.*ldhint*	f_1 = [r_3], imm_9	7	02	C-49 ページの表 C-37
ldfd.*ldhint*			03	
ldf8.*ldhint*			01	
ldfe.*ldhint*			00	
ldfs.s.*ldhint*			06	
ldfd.s.*ldhint*			07	
ldf8.s.*ldhint*			05	
ldfe.s.*ldhint*			04	
ldfs.a.*ldhint*			0A	
ldfd.a.*ldhint*			0B	
ldf8.a.*ldhint*			09	
ldfe.a.*ldhint*			08	
ldfs.sa.*ldhint*			0E	
ldfd.sa.*ldhint*			0F	
ldf8.sa.*ldhint*			0D	
ldfe.sa.*ldhint*			0C	
ldf.fill.*ldhint*			1B	
ldfs.c.clr.*ldhint*			22	
ldfd.c.clr.*ldhint*			23	
ldf8.c.clr.*ldhint*			21	
ldfe.c.clr.*ldhint*			20	
ldfs.c.nc.*ldhint*			26	
ldfd.c.nc.*ldhint*			27	
ldf8.c.nc.*ldhint*			25	
ldfe.c.nc.*ldhint*			24	

C.4.1.9. Floating-point Store

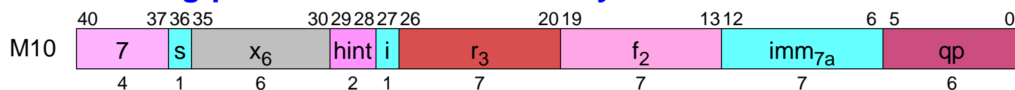
M9



命令	オペランド	オペコード	拡張			
			m	x	x_6	ヒント
stfs.*sthint*					32	
stfd.*sthint*					33	
stf8.*sthint*	$[r_3] = f_2$	6	0	0	31	C-49 ページの 表 C-37
stfe.*sthint*					30	
stf.spill.*sthint*					3B	

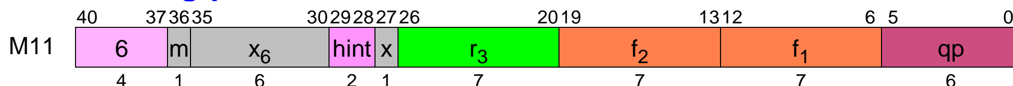
C.4.1.10. Floating-point Store – Increment by Immediate

M10



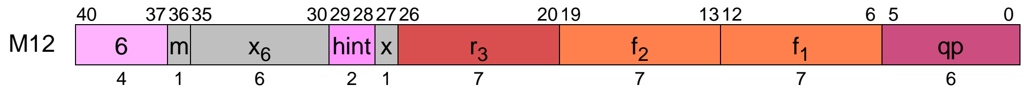
命令	オペランド	オペコード	拡張	
			x_6	ヒント
stfs.*sthint*			32	
stfd.*sthint*			33	
stf8.*sthint*	$[r_3] = f_2, imm_9$	7	31	C-49 ページの 表 C-37
stfe.*sthint*			30	
stf.spill.*sthint*			3B	

C.4.1.11. Floating-point Load Pair



命令	オペランド	オペコード	拡張			
			m	x	x_6	ヒント
ldfps.*ldhint*	$f_1, f_2 = [r_3]$	6	0	1	02	C-49 ページの表 C-37
ldfpd.*ldhint*					03	
ldfp8.*ldhint*					01	
ldfps.s.*ldhint*					06	
ldfpd.s.*ldhint*					07	
ldfp8.s.*ldhint*					05	
ldfps.a.*ldhint*					0A	
ldfpd.a.*ldhint*					0B	
ldfp8.a.*ldhint*					09	
ldfps.sa.*ldhint*					0E	
ldfpd.sa.*ldhint*					0F	
ldfp8.sa.*ldhint*					0D	
ldfps.c.clr.*ldhint*					22	
ldfpd.c.clr.*ldhint*					23	
ldfp8.c.clr.*ldhint*					21	
ldfps.c.nc.*ldhint*					26	
ldfpd.c.nc.*ldhint*					27	
ldfp8.c.nc.*ldhint*					25	

C.4.1.12. Floating-point Load Pair – Increment by Immediate



命令	オペランド	オペコード	拡張			
			m	x	x_6	ヒント
ldfps.*ldhint*	$f_1, f_2 = [r_3]$, 8	6	1	1	02	C-49 ページの表 C-37
ldfpd.*ldhint*	$f_1, f_2 = [r_3]$, 16				03	
ldfp8.*ldhint*					01	
ldfps.s.*ldhint*	$f_1, f_2 = [r_3]$, 8				06	
ldfpd.s.*ldhint*	$f_1, f_2 = [r_3]$, 16				07	
ldfp8.s.*ldhint*					05	
ldfps.a.*ldhint*	$f_1, f_2 = [r_3]$, 8				0A	
ldfpd.a.*ldhint*	$f_1, f_2 = [r_3]$, 16				0B	
ldfp8.a.*ldhint*					09	
ldfps.sa.*ldhint*	$f_1, f_2 = [r_3]$, 8				0E	
ldfpd.sa.*ldhint*	$f_1, f_2 = [r_3]$, 16				0F	
ldfp8.sa.*ldhint*					0D	
ldfps.c.clr.*ldhint*	$f_1, f_2 = [r_3]$, 8				22	
ldfpd.c.clr.*ldhint*	$f_1, f_2 = [r_3]$, 16				23	
ldfp8.c.clr.*ldhint*					21	
ldfps.c.nc.*ldhint*	$f_1, f_2 = [r_3]$, 8				26	
ldfpd.c.nc.*ldhint*	$f_1, f_2 = [r_3]$, 16				27	
ldfp8.c.nc.*ldhint*					25	

C.4.2 ライン・プリフェッチ

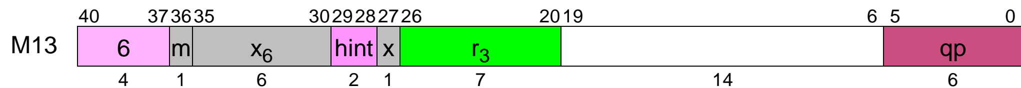
ライン・プリフェッチ命令は、浮動小数点ロード / ストア命令とともに、メジャー・オペコード 6 と 7 の中にエンコードされる。オペコード拡張の要約については、C-32 ページの「ロードとストア」を参照のこと。

すべてのライン・プリフェッチ命令は、表 C-39 に示す局所性ヒント情報をエンコードしている 2 ビットのオペコード拡張フィールドをビット 29:28(ヒント) に持っている。

表 C-39. ライン・プリフェッチ・ヒント・コンプリータ

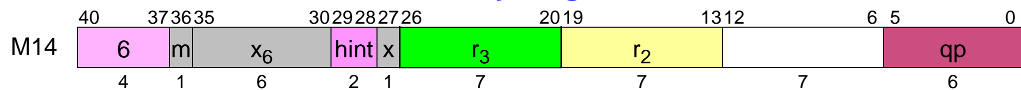
ヒント・ビット 29:28	*lfhint*
0	*none*
1	.nt1
2	.nt2
3	.nta

C.4.2.1. Line Prefetch



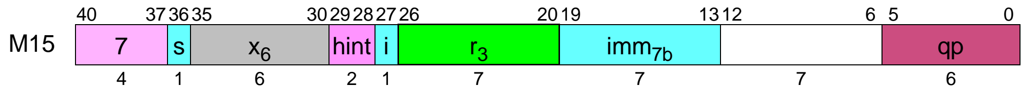
命令	オペランド	オペコード	拡張			
			m	x	x_6	ヒント
lfetch.*lfhint*	$[r_3]$	6	0	0	2C	C-60 ページの表 C-39
lfetch.excl.*lfhint*					2D	
lfetch.fault.*lfhint*					2E	
lfetch.fault.excl.*lfhint*					2F	

C.4.2.2. Line Prefetch – Increment by Register



命令	オペランド	オペコード	拡張			
			m	x	x_6	ヒント
lfetch.*lfhint*	$[r_3]$, r_2	6	1	0	2C	C-60 ページの表 C-39
lfetch.excl.*lfhint*					2D	
lfetch.fault.*lfhint*					2E	
lfetch.fault.excl.*lfhint*					2F	

C.4.2.3. Line Prefetch – Increment by Immediate

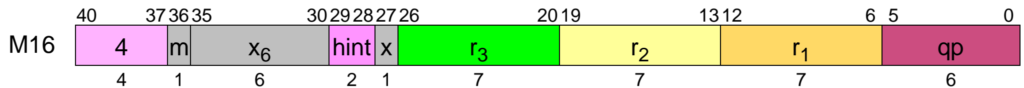


命令	オペランド	オペコード	拡張	
			x_6	ヒント
lfetch.*lfhint*	$[r_3]$, imm_9	7	2C	C-60 ページの表 C-39
lfetch.excl.*lfhint*			2D	
lfetch.fault.*lfhint*			2E	
lfetch.fault.excl.*lfhint*			2F	

C.4.3 セマフォ

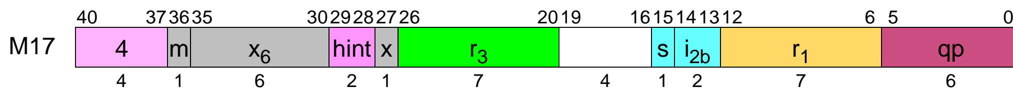
セマフォ命令は、整数ロード / ストア命令とともに、メジャー・オペコード 4 の中にエンコードされる。オペコード拡張の要約については、C-38 ページの「ロードとストア」を参照のこと。

C.4.3.1. Exchange/Compare and Exchange



命令	オペランド	オペコード	拡張			
			m	x	x_6	ヒント
cmpxchg1.acq.*ldhint*	$r_1 = [r_3], r_2$, ar.ccv	4	0	1	00	C-49 ページの表 C-37
cmpxchg2.acq.*ldhint*					01	
cmpxchg4.acq.*ldhint*					02	
cmpxchg8.acq.*ldhint*					03	
cmpxchg1.rel.*ldhint*					04	
cmpxchg2.rel.*ldhint*					05	
cmpxchg4.rel.*ldhint*					06	
cmpxchg8.rel.*ldhint*					07	
xchg1.*ldhint*	$r_1 = [r_3], r_2$				08	
xchg2.*ldhint*					09	
xchg4.*ldhint*					0A	
xchg8.*ldhint*					0B	

C.4.3.2. Fetch and Add – Immediate



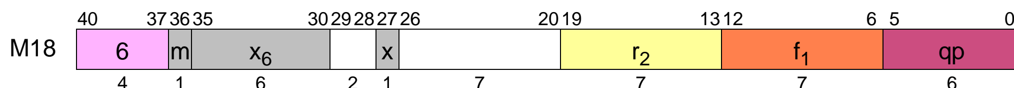
命令	オペランド	オペコード	拡張			
			m	x	x_6	ヒント
fetchadd4.acq.*ldhint*	$r_1 = [r_3], inc_3$	4	0	1	12	C-49 ページの表 C-37
fetchadd8.acq.*ldhint*					13	
fetchadd4.rel.*ldhint*					16	
fetchadd8.rel.*ldhint*					17	

C.4.4 FR 設定 / 取得

FR 設定命令は、浮動小数点ロード / ストア命令とともに、メジャー・オペコード 6 の中にエンコードされる。FR 取得命令は、整数ロード / ストア命令とともに、メジャー・オペコード 4 の中にエンコードされる。オペコード拡張の要約については、C-38 ページの「ロードとストア」を参照のこと。

C.4.4.1. Set FR

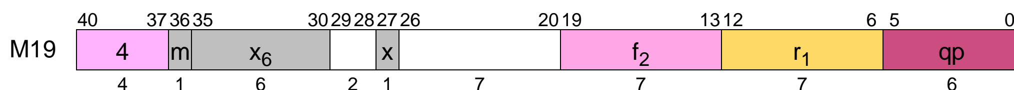
M18



命令	オペランド	オペコード	拡張		
			m	x	x_6
setf.sig	$f_1 = r_2$	6	0	1	1C
setf.exp					1D
setf.s					1E
setf.d					1F

C.4.4.2. Get FR

M19

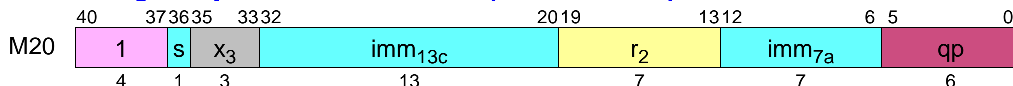


命令	オペランド	オペコード	拡張		
			m	x	x_6
getf.sig	$r_1 = f_2$	4	0	1	1C
getf.exp					1D
getf.s					1E
getf.d					1F

C.4.5 スペキュレーションおよびアドバンスド・ロード・チェック

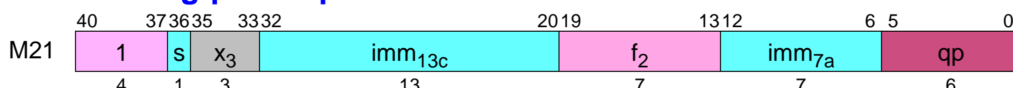
スペキュレーションおよびアドバンスド・ロード・チェック命令は、メモリ管理命令とともに、メジャー・オペコード 0 と 1 の中にエンコードされる。オペコード拡張の要約については、C-68 ページの「メモリ管理」を参照のこと。

C.4.5.1. Integer Speculation Check (M ユニット)



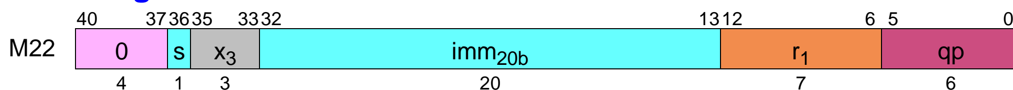
命令	オペランド	オペコード	拡張
			x_3
chk.s.m	r_2, $target_{25}$	1	1

C.4.5.2. Floating-point Speculation Check



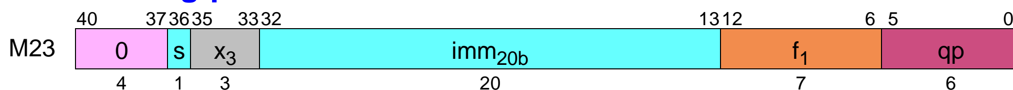
命令	オペランド	オペコード	拡張
			x_3
chk.s	f_2, $target_{25}$	1	3

C.4.5.3. Integer Advanced Load Check



命令	オペランド	オペコード	拡張
			x_3
chk.a.nc	r_1, $target_{25}$	0	4
chk.a.clr			5

C.4.5.4. Floating-point Advanced Load Check

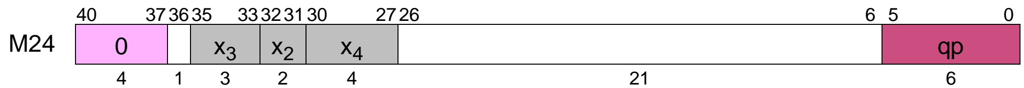


命令	オペランド	オペコード	拡張
			x_3
chk.a.nc	f_1, $target_{25}$	0	6
chk.a.clr			7

C.4.6 キャッシュ / 同期 /RSE/ALAT

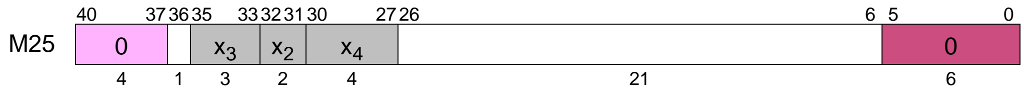
キャッシュ / 同期 /RSE/ALAT 命令は、メモリ管理命令とともに、メジャー・オペコード 0 の中にエンコードされる。オペコード拡張の要約については、C-68 ページの「メモリ管理」を参照のこと。

C.4.6.1. Sync/Fence/Serialize/ALAT Control



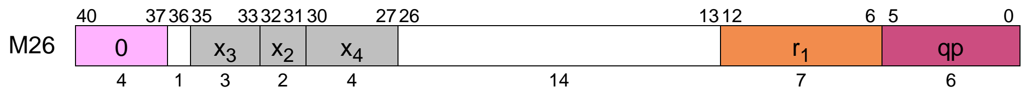
命令	オペコード	拡張		
		x_3	x_4	x_2
invala	0	0	0	1
mf			2	2
mf.a			3	
srlz.i			1	3
sync.i			3	

C.4.6.2. RSE Control



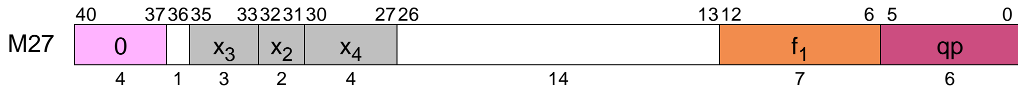
命令	オペコード	拡張		
		x_3	x_4	x_2
flushrs [f]	0	0	C	0

C.4.6.3. Integer ALAT Entry Invalidate



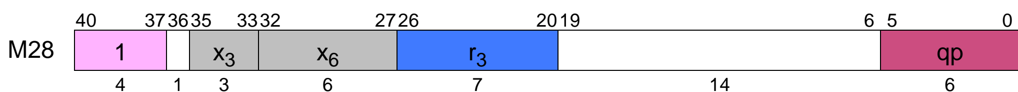
命令	オペランド	オペコード	拡張		
			x_3	x_4	x_2
invala.e	r_1	0	0	2	1

C.4.6.4. Floating-point ALAT Entry Invalidate



命令	オペランド	オペコード	拡張		
			x_3	x_4	x_2
invala.e	f_1	0	0	3	1

C.4.6.5. Flush Cache

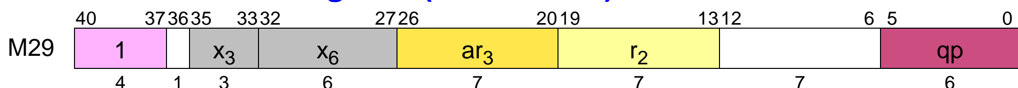


命令	オペランド	オペコード	拡張	
			x_3	x_6
fc	r_3	1	0	30

C.4.7 GR/AR 移動 (M ユニット)

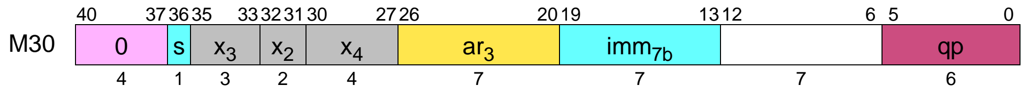
M ユニット GR/AR 移動命令は、メモリ管理命令とともに、メジャー・オペコード 0 の中にエンコードされる (一部の AR は、I ユニット上のシステム制御命令を使ってアクセスされる。C-37 ページの「GR/AR 移動 (I ユニット)」を参照のこと)。M ユニット GR/AR オペコード拡張の要約については、C-68 ページの「メモリ管理」を参照のこと。

C.4.7.1. Move to AR – Register (M ユニット)



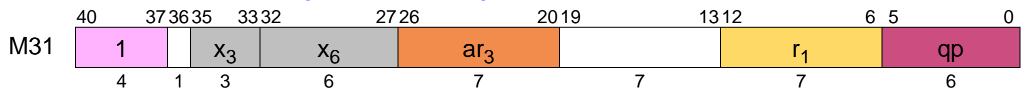
命令	オペランド	オペコード	拡張	
			x_3	x_6
mov.m	$ar_3 = r_2$	1	0	2A

C.4.7.2. Move to AR – Immediate$_8$ (M ユニット)



命令	オペランド	オペコード	拡張		
			x_3	x_4	x_2
mov.m	$ar_3 = imm_8$	0	0	8	2

C.4.7.3. Move from AR (M ユニット)

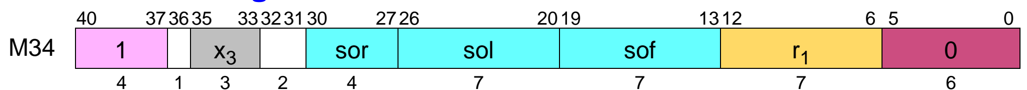


命令	オペランド	オペコード	拡張	
			x_3	x_6
mov.m	$r_1 = ar_3$	1	0	22

C.4.8 その他の M ユニット命令

その他の M ユニット命令は、メモリ管理命令とともに、メジャー・オペコード 0 の中にエンコードされる。オペコード拡張の要約については、C-68 ページの「メモリ管理」を参照のこと。

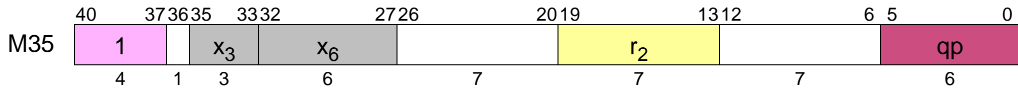
C.4.8.1. Allocate Register Stack Frame



命令	オペランド	オペコード	拡張
			x_3
alloc [f]	r_1 = ar.pfs, i, l, o, r	1	6

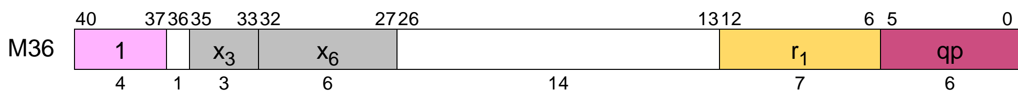
注： 命令エンコーディング内の 3 つの即値は、以下のオペランドから構成される。
sof = $i + l + o$
sol = $i + l$
sor = $r >> 3$

C.4.8.2. Move to PSR



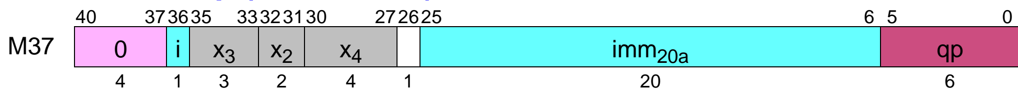
命令	オペランド	オペコード	拡張	
			x_3	x_6
mov	psr.um = r_2	1	0	29

C.4.8.3. Move from PSR



命令	オペランド	オペコード	拡張	
			x_3	x_6
mov	r_1 = psr.um	1	0	21

C.4.8.4. Break/Nop (M ユニット)



命令	オペランド	オペコード	拡張		
			x_3	x_4	x_2
break.m	imm_{21}	0	0	0	0
nop.m				1	

C.4.9 メモリ管理

すべてのメモリ管理命令は、メジャー・オペコード 0 と 1 の中にエンコードされる。ビット 35:33 に 3 ビットのオペコード拡張フィールド (x_3) を使用する。また、一部の命令は、ビット 30:27 に 4 ビットのオペコード拡張フィールド (x_4) か、ビット 32:27 に 6 ビットのオペコード拡張フィールド (x_6) を持つ。4 ビットのオペコード拡張フィールド持つ命令のほとんどは、ビット 32:31 にも 2 ビットの拡張フィールド (x_2) を持つ。表 C-40 はオペコード 0 の 3 ビットの割り当てを示し、表 C-41 はオペコード 0 の 4 ビット +2 ビットの割り当てを要約し、表 C-42 はオペコード 1 の 3 ビットの割り当てを示し、表 C-43 はオペコード 1 の 6 ビットの割り当てを要約している。

表 C-40. オペコード 0 のメモリ管理の 3 ビット・オペコード拡張

オペコード ビット 40:37	x_3 ビット 35:33	
0	0	メモリ管理 4 ビット +2 ビット拡張 (表 C-41)
	1	
	2	
	3	
	4	chk.a.nc – int M22
	5	chk.a.clr – int M22
	6	chk.a.nc – fp M23
	7	chk.a.clr – fp M23

表 C-41. オペコード 0 のメモリ管理の 4 ビット +2 ビット・オペコード拡張

オペコード ビット 40:37	x_3 ビット 35:33	x_4 ビット 30:27	x_2 ビット 32:31			
			0	1	2	3
0	0	0	break.m M37	invala M24		
		1	nop.m M37			srlz.i M24
		2		invala.e – int M26	mf M24	
		3		invala.e – fp M27	mf.a M24	sync.i M24
		4	sum M44			
		5	rum M44			
		6				
		7				
		8			mov.m to ar – imm_8 M30	
		9				
		A				
		B				
		C	flushrs M25			
		D				
		E				
		F				

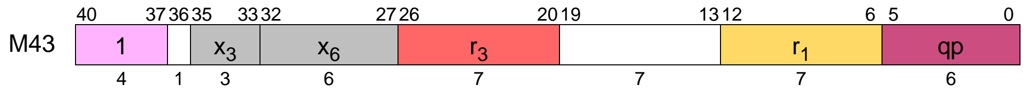
表 C-42. オペコード 1 のメモリ管理の 3 ビット・オペコード拡張

オペコード ビット 40:37	x_3 ビット 35:33	
1	0	メモリ管理 6 ビット拡張 (表 C-43)
	1	chk.s.m – int M20
	2	
	3	chk.s – fp M21
	4	
	5	
	6	alloc M34
	7	

表 C-43. オペコード 1 のメモリ管理の 6 ビット・オペコード拡張

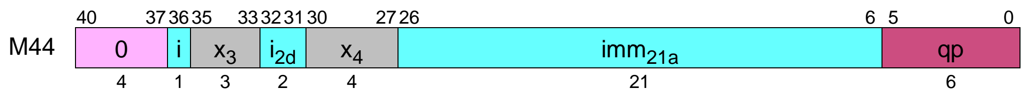
オペコード ビット 40:37	x_3 ビット 35:33	x_6				
		ビット 30:27	ビット 32:31			
			0	1	2	3
1	0	0				fc M28
		1			mov from psr.um M36	
		2			mov.m from ar M31	
		3				
		4				
		5		mov from pmd M43		
		6				
		7		mov from cpuid M43		
		8				
		9			mov to psr.um M35	
		A			mov.m to ar M29	
		B				
		C				
		D				
		E				
		F				

C.4.9.1. Move from Indirect Register



命令	オペランド	オペコード	拡張	
			x_3	x_6
mov	r_1 = pmd[r_3]	1	0	15
	r_1 = cpuid[r_3]			17

C.4.9.2. Set/Reset User Mask



命令	オペランド	オペコード	拡張	
			x_3	x_4
sum	imm_{24}	0	0	4
rum				5

C.5 B ユニット命令エンコーディング

分岐ユニットには、分岐とその他の命令が含まれる。

C.5.1 分岐

間接分岐にはオペコード 0、間接コールにはオペコード 1、IP 相対分岐にはオペコード 4、IP 相対コールにはオペコード 5 が使用される。

メジャー・オペコード 4 の中にエンコードされる IP 相対分岐命令は、表 C-44 に示すように、分岐のタイプを区別するために、ビット 8:6(btype) に 3 ビットのオペコード拡張フィールドを使用する。

表 C-44. IP 相対分岐タイプ

オペコード ビット 40:37	btype ビット 8:6	
4	0	br.cond B1
	1	
	2	br.wexit B1
	3	br.wtop B1
	4	
	5	br.cloop B2
	6	br.cexit B2
	7	br.ctop B2

間接分岐、間接リターン、およびその他の分岐ユニット命令は、メジャー・オペコード 0 の中にエンコードされる。ビット 32:27(x_6) に 6 ビットのオペコード拡張フィールドを使用する。表 C-45 にこれらの割り当てを要約する。

表 C-45. 間接 / その他の分岐のオペコード拡張

オペコード ビット 40:37	x_6				
	ビット 30:27	ビット 32:31			
		0	1	2	3
0	0	break.b B9		間接分岐 (表 C-46)	
	1			間接リターン (表 C-47)	
	2				
	3				
	4	clrrrb B8			
	5	clrrrb.pr B8			
	6				
	7				
	8				
	9				
	A				
	B				
	C				
	D				
	E				
	F				

メジャー・オペコード 0 の中にエンコードされる間接分岐命令は、表 C-46 に示すように、分岐のタイプを区別するために、ビット 8:6(btype) に 3 ビットのオペコード拡張フィールドを使用する。

表 C-46. 間接分岐タイプ

オペコード ビット 40:37	x_6 ビット 32:27	btype ビット 8:6	
0	20	0	br.cond B4
		1	br.ia B4
		2	
		3	
		4	
		5	
		6	
		7	

メジャー・オペコード 0 の中にエンコードされる間接リターン分岐命令は、表 C-47 に示すように、分岐のタイプを区別するために、ビット 8:6(btype) に 3 ビットのオペコード拡張フィールドを使用する。

表 C-47. 間接リターン分岐タイプ

オペコード ビット 40:37	x_6 ビット 32:27	btype ビット 8:6	
0	21	0	
		1	
		2	
		3	
		4	br.ret B4
		5	
		6	
		7	

すべての分岐命令は、シーケンシャル・プリフェッチ・ヒントを提供する 1 ビットのオペコード拡張フィールドをビット 12 に持つ。表 C-48 にこれらの割り当てを要約する。

表 C-48. シーケンシャル・プリフェッチ・ヒント・コンプリータ

p ビット 12	*ph*
0	.few
1	.many

IP 相対および間接分岐命令は、いずれも表 C-49 に示した分岐予測有無ヒント情報をエンコードする 2 ビットのオペコード拡張フィールドをビット 34:33(wh) に持つ。間接コール命令は、表 C-50 に示すように、" 有無 " ヒント情報を提供する 3 ビットのオペコード拡張フィールドをビット 34:32(wh) に持つ

表 C-49. 分岐有無ヒント・コンプリータ

wh ビット 34:33	*bwh*
0	.sptk
1	.spnt
2	.dptk
3	.dpnt

表 C-50. 間接コール有無ヒント・コンプリータ

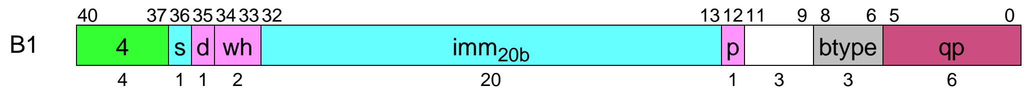
wh ビット 34:32	*bwh*
0	
1	.sptk
2	
3	.spnt
4	
5	.dptk
6	
7	.dpnt

また、分岐命令は、表 C-51 に示すように、分岐キャッシュ割り当て解除ヒントをエンコードする 1 ビットのオペコード拡張フィールドをビット 35(d) に持つ。

表 C-51. 分岐キャッシュ割り当て解除ヒント・コンプリータ

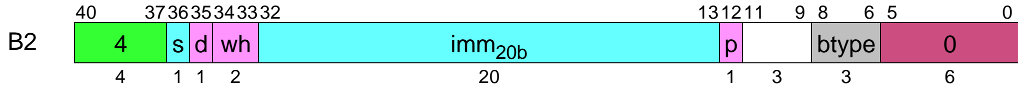
d ビット 35	*dh*
0	*none*
1	.clr

C.5.1.1. IP-Relative Branch



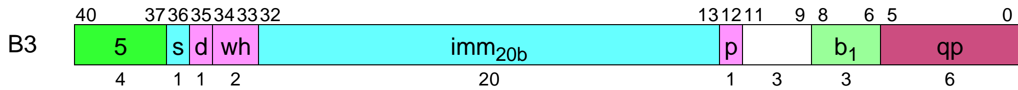
命令	オペランド	オペコード	拡張			
			btype	p	wh	d
br.cond.*bwh*.*ph*.*dh*	$target_{25}$	4	0	C-74 ページの表 C-48	C-74 ページの表 C-49	C-75 ページの表 C-51
br.wexit.*bwh*.*ph*.*dh* t			2			
br.wtop.*bwh*.*ph*.*dh* t			3			

C.5.1.2. IP-Relative Counted Branch



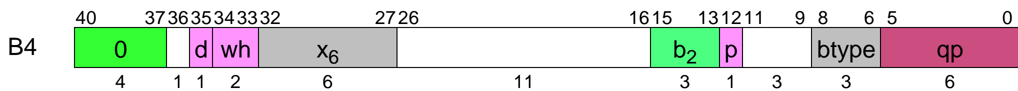
命令	オペランド	オペコード	拡張			
			btype	p	wh	d
br.cloop.*bwh*.*ph*.*dh* [t]	*target*$_{25}$	4	5	C-74 ページの表 C-48	C-74 ページの表 C-49	C-75 ページの表 C-51
br.cexit.*bwh*.*ph*.*dh* [t]			6			
br.ctop.*bwh*.*ph*.*dh* [t]			7			

C.5.1.3. IP-Relative Call



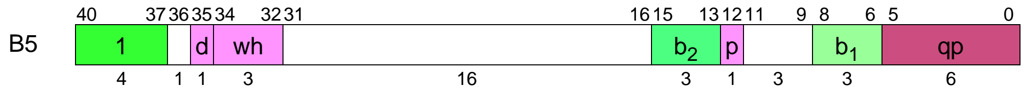
命令	オペランド	オペコード	拡張		
			p	wh	d
br.call.*bwh*.*ph*.*dh*	b_1 = *target*$_{25}$	5	C-74 ページの表 C-48	C-74 ページの表 C-49	C-75 ページの表 C-51

C.5.1.4. Indirect Branch



命令	オペランド	オペコード	拡張				
			x_6	btype	p	wh	d
br.cond.*bwh*.*ph*.*dh*	b_2	0	20	0	C-74 ページの表 C-48	C-74 ページの表 C-49	C-75 ページの表 C-51
br.ia.*bwh*.*ph*.*dh*				1			
br.ret.*bwh*.*ph*.*dh*			21	4			

C.5.1.5. Indirect Call



命令	オペランド	オペコード	拡張		
			p	wh	d
br.call.$bwh.ph.dh$	$b_1 = b_2$	1	C-74 ページの表 C-48	C-74 ページの表 C-49	C-75 ページの表 C-51

C.5.2 Nop

nop 命令はメジャー・オペコード 2 の中にエンコードされる。メジャー・コード 2 の中の nop 命令は、ビット 32:27(x_6) に 6 ビットのオペコード拡張フィールドを使用する。表 C-52 にこれらの割り当てを要約する。

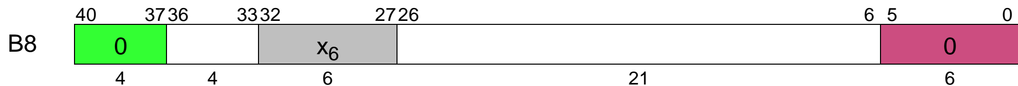
表 C-52. 間接予測 /nop のオペコード拡張

オペコードビット 40:37	x_6				
	ビット 30:27	ビット 32:31			
		0	1	2	3
2	0	nop.b B9			
	1				
	2				
	3				
	4				
	5				
	6				
	7				
	8				
	9				
	A				
	B				
	C				
	D				
	E				
	F				

C.5.3 その他の B ユニット命令

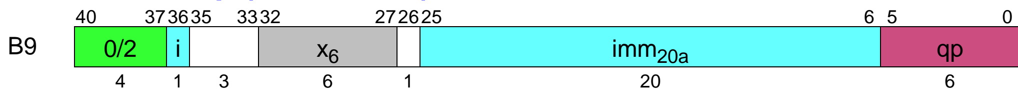
その他の分岐ユニット命令には、C-73 ページの表 C-45 で説明しているように、ビット 32:27(x_6) に 6 ビットのオペコード拡張フィールドを使用する、メジャー・オペコード 0 の中にエンコードされた命令をいくつか含んでいる。

C.5.3.1. Miscellaneous (B ユニット)



命令	オペコード	拡張
		x_6
clrrb [l]	0	04
clrrb.pr [l]		05

C.5.3.2. Break/Nop (B ユニット)



命令	オペランド	オペコード	拡張
			x_6
break.b	*imm₂₁*	0	00
nop.b		2	

C.6 F ユニット命令エンコーディング

浮動小数点命令は、浮動小数点および固定小数点算術演算についてはメジャー・オペコード 8-E、浮動小数点比較についてはオペコード 4、浮動小数点分類についてはオペコード 5、およびその他の浮動小数点命令についてはオペコード 0 と 1 の中にエンコードされる。

その他の浮動小数点命令と浮動小数点逆数近似命令は、メジャー・オペコード 0 と 1 の中にエンコードされる。ビット 33 に 1 ビットのオペコード拡張フィールド (x) を使用し、さらにビット 36(q) に 1 ビットの拡張フィールドか、ビット 32:27 に 6 ビットのオペコード拡張フィールド (x_6) を使用する。表 C-53 に 1 ビットの x の割り当てを示し、表 C-56 に逆数近似命令のための追加の 1 ビットの q の割り当てを示す。表 C-54 と表 C-55 は、6 ビットの x_6 の割り当てを要約している。

表 C-53. その他の浮動小数点の 1 ビット・オペコード拡張

オペコード・ビット 40:37	x ビット 33	
0	0	ビット拡張 (表 C-54)
	1	Reciprocal Approximation (表 C-56) 逆数近似 (表 C-56)
1	0	6 ビット拡張 (表 C-55)
	1	逆数近似 (表 C-58)

表 C-54. オペコード 0 のその他の浮動小数点の 6 ビット・オペコード拡張

オペコード ビット 40:37	x ビット 33	x_6				
		ビット 30:27	ビット 32:31			
			0	1	2	3
0	0	0	break.f F15	fmerge.s F9		
		1	nop.f F15	fmerge.ns F9		
		2		fmerge.se F9		
		3				
		4	fsetc F12	fmin F8		fswap F9
		5	fclrf F13	fmax F8		fswap.nl F9
		6		famin F8		fswap.nr F9
		7		famax F8		
		8	fchkf F14	fcvt.fx F10	fpack F9	
		9		fcvt.fxu F10		fmix.lr F9
		A		fcvt.fx.trunc F10		fmix.r F9
		B		fcvt.fxu.trunc F10		fmix.l F9
		C		fcvt.xf F11	fand F9	fsxt.r F9
		D			fandcm F9	fsxt.l F9
		E			for F9	
		F			fxor F9	

表 C-55. オペコード 1 のその他の浮動小数点の 6 ビット・オペコード拡張

オペコードビット 40:37	x ビット 33	x_6				
		ビット 30:27	ビット 32:31			
			0	1	2	3
1	0	0		fpmerge.s F9		fpcmp.eq F8
		1		fpmerge.ns F9		fpcmp.lt F8
		2		fpmerge.se F9		fpcmp.le F8
		3				fpcmp.unord F8
		4		fpmin F8		fpcmp.neq F8
		5		fpmax F8		fpcmp.nlt F8
		6		fpamin F8		fpcmp.nle F8
		7		fpamax F8		fpcmp.ord F8
		8		fpcvt.fx F10		
		9		fpcvt.fxu F10		
		A		fpcvt.fx.trunc F10		
		B		fpcvt.fxu.trunc F10		
		C				
		D				
		E				
		F				

表 C-56. 逆数近似の 1 ビット・オペコード拡張

オペコード ビット 40:37	x ビット 33	q ビット 36	
0	1	0	frcpa F6
		1	frsqrta F7
1		0	fprcpa F6
		1	fprsqrta F7

ほとんどの浮動小数点命令は、使用する FPSR ステータス・フィールドをエンコードする 2 ビットのオペコード拡張フィールドをビット 35:34(sf) に持つ。表 C-57 にこれらの割り当てを要約する。

表 C-57. 浮動小数点ステータス・フィールド・コンプリータ

sf ビット 35:34	*sf*
0	.s0
1	.s1
2	.s2
3	.s3

C.6.1 算術演算

浮動小数点算術命令は、メジャー・オペコード 8-D の中にエンコードされる。ビット 36 に 1 ビットのオペコード拡張フィールド (x) を、ビット 35:34 に 2 ビットのオペコード拡張フィールド (sf) を使用する。表 C-58 にオペコードと x の割り当てを示す。

表 C-58. 浮動小数点算術の 1 ビット・オペコード拡張

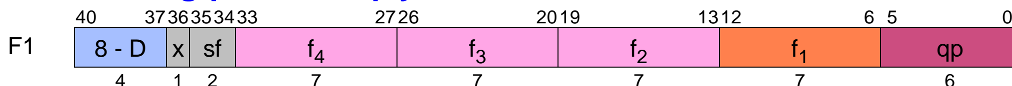
x ビット 36	オペコード ビット 40:37					
	8	9	A	B	C	D
0	fma F1	fma.d F1	fms F1	fms.d F1	fnma F1	fnma.d F1
1	fma.s F1	fpma F1	fms.s F1	fpms F1	fnma.s F1	fpnma F1

固定小数点算術命令と並列浮動小数点選択命令は、ビット 36 に 1 ビットのオペコード拡張フィールド (x) を使用して、メジャー・オペコード E の中にエンコードされる。また、固定小数点算術命令は、ビット 35:34 に 2 ビットのオペコード拡張フィールド (x_2) を持つ。表 C-59 にこれらの割り当てを示す。

表 C-59. 固定小数点の積和および選択のオペコード拡張

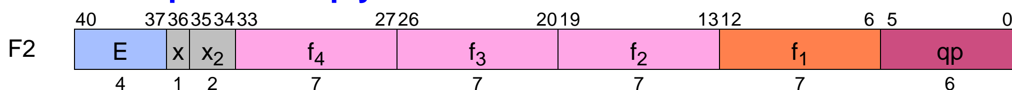
オペコード ビット 40:37	x ビット 36	x_2 ビット 35:34			
		0	1	2	3
E	0	fselect F3			
	1	xma.l F2		xma.hu F2	xma.h F2

C.6.1.1. Floating-point Multiply Add



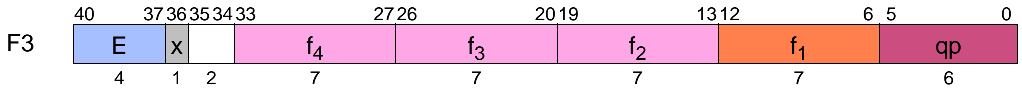
命令	オペランド	オペコード	拡張	
			x	sf
fma.*sf*	$f_1 = f_3, f_4, f_2$	8	0	C-81 ページの表 C-57
fma.s.*sf*			1	
fma.d.*sf*		9	0	
fpma.*sf*			1	
fms.*sf*		A	0	
fms.s.*sf*			1	
fms.d.*sf*		B	0	
fpms.*sf*			1	
fnma.*sf*		C	0	
fnma.s.*sf*			1	
fnma.d.*sf*		D	0	
fpnma.*sf*			1	

C.6.1.2. Fixed-point Multiply Add



命令	オペランド	オペコード	拡張	
			x	x_2
xma.l	$f_1 = f_3, f_4, f_2$	E	1	0
xma.h				3
xma.hu				2

C.6.2 並列浮動小数点 Select



命令	オペランド	オペコード	拡張
			x
fselect	$f_1 = f_3, f_4, f_2$	E	0

C.6.3 比較と分類

プレディケートを設定する浮動小数点比較命令は、メジャー・オペコード 4 の中にエンコードされる。ビット 33(r_a)、36(r_b)、および 12(t_a) に 3 つの 1 ビット・オペコード拡張フィールドを使用し、ビット 35:34 に 2 ビットのオペコード拡張フィールド (sf) を使用する。表 C-60 にオペコード r_a、r_b、および t_a の割り当てを示す。sf の割り当ては、C-81 ページの表 C-57 に示している。

並列浮動小数点比較命令については、C-86 ページで説明する。

表 C-60. 浮動小数点比較のオペコード拡張

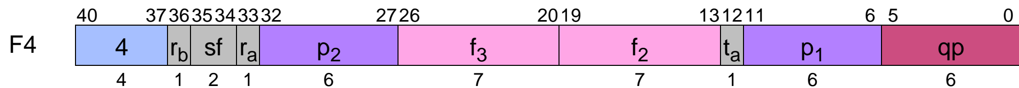
オペコード ビット 40:37	r_a ビット 33	r_b ビット 36	t_a ビット 12	
			0	1
4	0	0	fcmp.eq F4	fcmp.eq.unc F4
		1	fcmp.lt F4	fcmp.lt.unc F4
	1	0	fcmp.le F4	fcmp.le.unc F4
		1	fcmp.unord F4	fcmp.unord.unc F4

浮動小数点分類命令は、メジャー・オペコード 5 の中にエンコードされる。表 C-61 に示すように、ビット 12(t_a) に 1 ビットのオペコード拡張フィールドを使用する。

表 C-61. 浮動小数点分類の 1 ビット・オペコード拡張

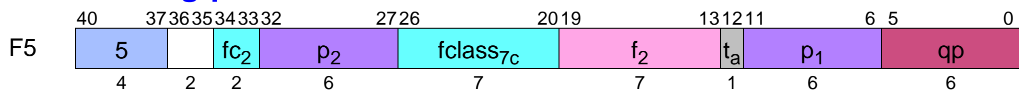
オペコード ビット 40:37	t_a ビット 12	
5	0	fclass.m F5
	1	fclass.m.unc F5

C.6.3.1. Floating-point Compare



命令	オペランド	オペコード	拡張			
			r_a	r_b	t_a	sf
fcmp.eq.*sf*	$p_1, p_2 = f_2, f_3$	4	0	0	0	C-81 ページの表 「浮動小数点ステータス・フィールド・コンプリータ」
fcmp.lt.*sf*				1		
fcmp.le.*sf*			1	0		
fcmp.unord.*sf*				1		
fcmp.eq.unc.*sf*			0	0	1	
fcmp.lt.unc.*sf*				1		
fcmp.le.unc.*sf*			1	0		
fcmp.unord.unc.*sf*				1		

C.6.3.2. Floating-point Class



命令	オペランド	オペコード	拡張
			t_a
fclass.m	$p_1, p_2 = f_2, fclass_9$	5	0
fclass.m.unc			1

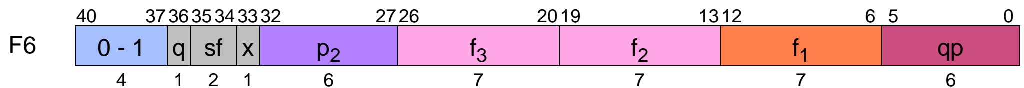
intel®

C.6.4 近似

C.6.4.1. Floating-point Reciprocal Approximation

逆数近似命令は 2 つ存在する。第 1 のメジャー・オペコード 0 に含まれる命令は、フル・レジスタ形式をエンコードする。第 2 のメジャー・オペコード 1 に含まれる命令は、並列形式をエンコードする。

F6

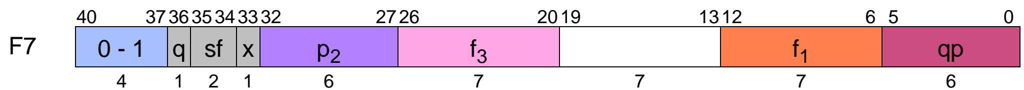


命令	オペランド	オペコード	拡張		
			x	q	sf
frcpa.*sf*	$f_1, p_2 = f_2, f_3$	0	1	0	C-81 ページの表 C-57
fprcpa.*sf*		1			

C.6.4.2. Floating-point Reciprocal Square Root Approximation

平方根逆数近似命令は 2 つ存在する。第 1 のメジャー・オペコード 0 に含まれる命令は、フル・レジスタ形式をエンコードする。第 2 のメジャー・オペコード 1 に含まれる命令は、並列形式をエンコードする。

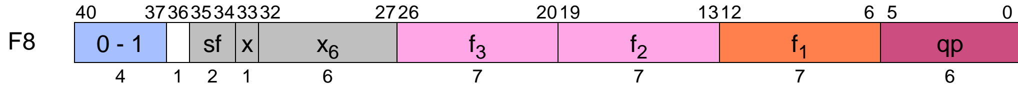
F7



命令	オペランド	オペコード	拡張		
			x	q	sf
frsqrta.*sf*	$f_1, p_2 = f_3$	0	1	1	C-81 ページの表 C-57
fprsqrta.*sf*		1			

C.6.5 最小値 / 最大値と並列比較

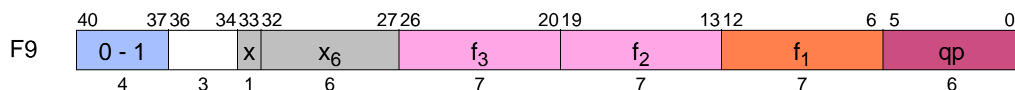
最小値 / 最大値命令には 2 つのグループがある。第 1 のメジャー・オペコード 0 に含まれるグループは、フル・レジスタ形式をエンコードする。第 2 のメジャー・オペコード 1 に含まれるグループは、並列形式をエンコードする。並列比較命令はすべてメジャー・オペコード 1 の中にエンコードされる。



命令	オペランド	オペコード	拡張 x	拡張 x_6	拡張 sf
fmin.*sf*	$f_1 = f_2, f_3$	0	0	14	C-81 ページの表 C-57
fmax.*sf*				15	
famin.*sf*				16	
famax.*sf*				17	
fpmin.*sf*		1		14	
fpmax.*sf*				15	
fpamin.*sf*				16	
fpamax.*sf*				17	
fpcmp.eq.*sf*				30	
fpcmp.lt.*sf*				31	
fpcmp.le.*sf*				32	
fpcmp.unord.*sf*				33	
fpcmp.neq.*sf*				34	
fpcmp.nlt.*sf*				35	
fpcmp.nle.*sf*				36	
fpcmp.ord.*sf*				37	

C.6.6 マージと論理

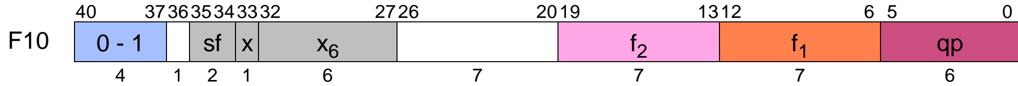
F9



命令	オペランド	オペコード	拡張	
			x	x_6
fmerge.s	$f_1 = f_2, f_3$	0	0	10
fmerge.ns				11
fmerge.se				12
fmix.lr				39
fmix.r				3A
fmix.l				3B
fsxt.r				3C
fsxt.l				3D
fpack				28
fswap				34
fswap.nl				35
fswap.nr				36
fand				2C
fandcm				2D
for				2E
fxor				2F
fpmerge.s		1		10
fpmerge.ns				11
fpmerge.se				12

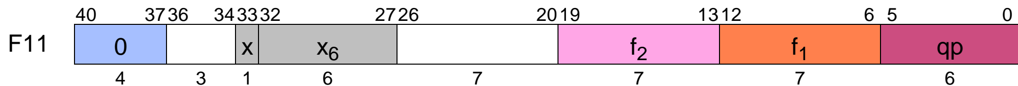
C.6.7 変換

C.6.7.1. Convert Floating-point to Fixed-point



命令	オペランド	オペコード	拡張		
			x	x_6	sf
fcvt.fx.*sf*	$f_1 = f_2$	0	0	18	C-81 ページの表 C-57
fcvt.fxu.*sf*				19	
fcvt.fx.trunc.*sf*				1A	
fcvt.fxu.trunc.*sf*				1B	
fpcvt.fx.*sf*		1		18	
fpcvt.fxu.*sf*				19	
fpcvt.fx.trunc.*sf*				1A	
fpcvt.fxu.trunc.*sf*				1B	

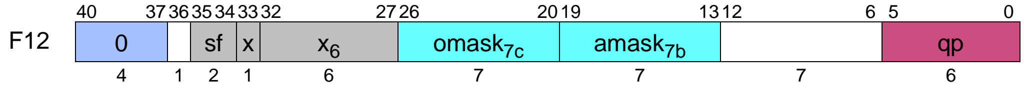
C.6.7.2. Convert Fixed-point to Floating-point



命令	オペランド	オペコード	拡張	
			x	x_6
fcvt.xf	$f_1 = f_2$	0	0	1C

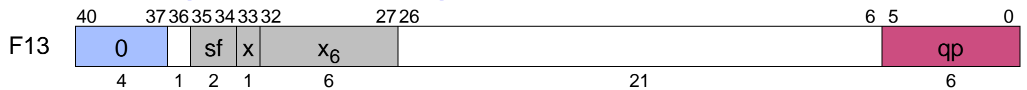
C.6.8 ステータス・フィールド操作

C.6.8.1. Floating-point Set Controls



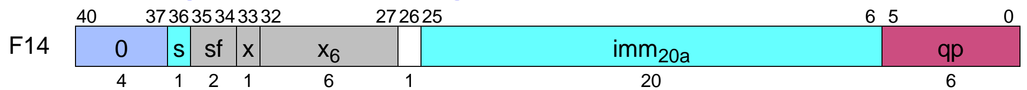
命令	オペランド	オペコード	拡張		
			x	x_6	sf
fsetc.*sf*	$amask_7$, $omask_7$	0	0	04	C-81 ページの表 C-57

C.6.8.2. Floating-point Clear Flags



命令	オペコード	拡張		
		x	x_6	sf
fclrf.*sf*	0	0	05	C-81 ページの表 C-57

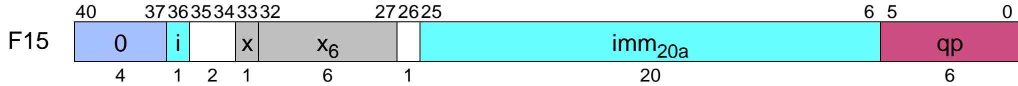
C.6.8.3. Floating-point Check Flags



命令	オペランド	オペコード	拡張		
			x	x_6	sf
fchkf.*sf*	$target_{25}$	0	0	08	C-81 ページの表 C-57

C.6.9 その他の F ユニット命令

C.6.9.1. Break/Nop (F ユニット)



命令	オペランド	オペコード	拡張	
			x	x_6
break.f	imm_{21}	0	0	00
nop.f				01

C.7 ユニット命令エンコーディング

X ユニット命令は、L と X の 2 つの命令スロットを占有する。メジャー・オペコード、オペコード拡張とヒント、qp、および小さな即値・フィールドが X 命令スロットを占有する。movl、break.x、および nop.x では、imm_{41} フィールドが L 命令スロットを占有する。

C.7.1 その他の X ユニット命令

その他の X ユニット命令は、メジャー・オペコード 0 の中にエンコードされる。ビット 35:33 に 3 ビットのオペコード拡張フィールド (x_3) を、ビット 32:27 に 6 ビットのオペコード拡張フィールド (x_6) を使用する。表 C-62 に 3 ビットの割り当てを、表 C-63 に 6 ビットの割り当てを示す。これらの命令は I ユニットによって実行される。

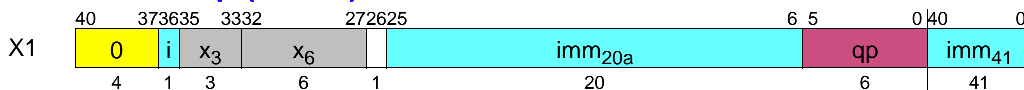
表 C-62. その他の X ユニットの 3 ビット・オペコード拡張

オペコード ビット 40:37	x_3 ビット 35:33	
0	0	6 ビット拡張 (表 C-63)
	1	
	2	
	3	
	4	
	5	
	6	
	7	

表 C-63. その他の X ユニットの 6 ビット・オペコード拡張

オペコード ビット 40:37	x_3 ビット 35:33	x_6				
		ビット 30:27	ビット 32:31			
			0	1	2	3
0	0	0	break.x X1			
		1	nop.x X1			
		2				
		3				
		4				
		5				
		6				
		7				
		8				
		9				
		A				
		B				
		C				
		D				
		E				
		F				

C.7.1.1. Break/Nop (X-Unit)



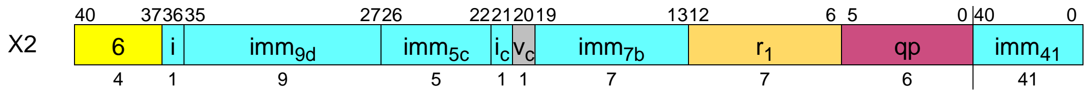
命令	オペランド	オペコード	拡張	
			x_3	x_6
break.x	*imm*$_{62}$	0	0	00
nop.x				01

C.7.2 ロング型即値$_{64}$ 移動

ロング型即値移動命令は、メジャー・オペコード 6 の中にエンコードされる。表 C-64 に示すように、ビット 20(v_c) に 1 ビットの予約済みオペコード拡張を使用する。この命令は I ユニットによって実行される。

表 C-64. ロング型移動の 1 ビット・オペコード拡張

オペコード ビット 40:37	v_c ビット 20	
6	0	movl X2
	1	



命令	オペランド	オペコード	拡張
			v_c
movl	$r_1 = imm_{64}$	6	0

C.8 即値の生成

表 C-65 は、1 つまたは複数の即値を持つ個々の命令について、それらの即値の生成方法を示している。個々の式で、等号の左辺のシンボルはその即値のアセンブリ言語名である。右辺のシンボルは、命令エンコーディングにおけるフィールド名である。

表 C-65. 即値の生成

命令形式	即値の生成
A2	$count_2 = ct_{2d} + 1$
A3 A8 I27 M30	$imm_8 = sign_ext(s << 7 \mid imm_{7b}, 8)$
A4	$imm_{14} = sign_ext(s << 13 \mid imm_{6d} << 7 \mid imm_{7b}, 14)$
A5	$imm_{22} = sign_ext(s << 21 \mid imm_{5c} << 16 \mid imm_{9d} << 7 \mid imm_{7b}, 22)$
A10	$count_2 = (ct_{2d} > 2)$? reservedQP[a] : $ct_{2d} + 1$
I1	$count_2 = (ct_{2d} == 0)$? 0 : $(ct_{2d} == 1)$? 7 : $(ct_{2d} == 2)$? 15 : 16
I3	$mbtype_4 = (mbt_{4c} == 0)$? @brcst : $(mbt_{4c} == 8)$? @mix : $(mbt_{4c} == 9)$? @shuf : $(mbt_{4c} ==$ 0xA) ? @alt : $(mbt_{4c} ==$ 0xB) ? @rev : reservedQP[a]
I4	$mhtype_8 = mht_{8c}$
I6	$count_5 = count_{5b}$
I8	$count_5 = 31 - ccount_{5c}$
I10	$count_6 = count_{6d}$
I11	$len_6 = len_{6d} + 1$ $pos_6 = pos_{6b}$

表 C-65. 即値の生成 (続き)

命令形式	即値の生成
I12	$len_6 = len_{6d} + 1$ $pos_6 = 63 - cpos_{6c}$
I13	$len_6 = len_{6d} + 1$ $pos_6 = 63 - cpos_{6c}$ $imm_8 = sign_ext(s << 7 \vert imm_{7b}, 8)$
I14	$len_6 = len_{6d} + 1$ $pos_6 = 63 - cpos_{6b}$ $imm_1 = sign_ext(s, 1)$
I15	$len_4 = len_{4d} + 1$ $pos_6 = 63 - cpos_{6d}$
I16	$pos_6 = pos_{6b}$
I19 M37	$imm_{21} = i << 20 \vert imm_{20a}$
I23	$mask_{17} = sign_ext(s << 16 \vert mask_{8c} << 8 \vert mask_{7a} << 1, 17)$
I24	$imm_{44} = sign_ext(s << 43 \vert imm_{27a} << 16, 44)$
M3 M8 M15	$imm_9 = sign_ext(s << 8 \vert i << 7 \vert imm_{7b}, 9)$
M5 M10	$imm_9 = sign_ext(s << 8 \vert i << 7 \vert imm_{7a}, 9)$
M17	$inc_3 = sign_ext(((s)\ ?\ -1 : 1) * ((i_{2b} == 3)\ ?\ 1 : 1 << (4 - i_{2b})), 6)$
I20 M20 M21	$target_{25} = IP + (sign_ext(s << 20 \vert imm_{13c} << 7 \vert imm_{7a}, 21) << 4)$
M22 M23	$target_{25} = IP + (sign_ext(s << 20 \vert imm_{20b}, 21) << 4)$
M34	$il = sol$ $o = sof - sol$ $r = sor << 3$
M44	$imm_{24} = i << 23 \vert i_{2d} << 21 \vert imm_{21a}$
B1 B2 B3	$target_{25} = IP + (sign_ext(s << 20 \vert imm_{20b}, 21) << 4)$
B9	$imm_{21} = i << 20 \vert imm_{20a}$
F5	$fclass_9 = fclass_{7c} << 2 \vert fc_2$
F12	$amask_7 = amask_{7b}$ $omask_7 = omask_{7c}$
F14	$target_{25} = IP + (sign_ext(s << 20 \vert imm_{20a}, 21) << 4)$
F15	$imm_{21} = i << 20 \vert imm_{20a}$
X1	$imm_{62} = imm_{41} << 21 \vert i << 20 \vert imm_{20a}$
X2	$imm_{64} = i << 63 \vert imm_{41} << 22 \vert i_c << 21 \vert imm_{5c} << 16 \vert imm_{9d} << 7 \vert imm_{7b}$

a. このエンコーディングは、修飾プレディケートの値が 1 だった場合に、無効操作フォルトを発生させる。