SONY

SONY COMPUTER ENTERTAINMENT

Synergistic Processor Unit

命令セット・アーキテクチャ

Version 1.2

2007年 1月 27日

ソニー株式会社
〒141-0001 東京都品川区北品川 6-7-35
（2007年2月より 〒108-0075 東京都港区港南 1-7-1）

株式会社ソニー・コンピュータエンタテインメント
〒107-0062 東京都港区南青山 2-6-21

ソニーのホームページ　http://www.sony.net
SCEIのホームページ　http://www.scei.co.jp

Cell Broadband Engine のホームページ　http://cell.scei.co.jp

Version 1.2
2007 年 1 月 27 日

目次

図目次

表目次

序文

本ドキュメントは、Cell Broadband Engine Architecture（CBEA）に関連する Synergistic Processor Unit（SPU）命令セット・アーキテクチャ（Instruction Set Architecture — ISA）を記述することを目的とする。

対象とする読者

本ドキュメントは、SPU ISA を使用した製品の開発を意図している設計者を対象とする。本ドキュメントは、13ページの*関連文献*にリストアップするドキュメントと併せてご使用いただきたい。

関連文献

下記のドキュメントは、SPU ISA のための参照資料である。

タイトル	バージョン	発行日
Cell Broadband Engine™ アーキテクチャ	1.01	2006年10月
PowerPC User Instruction Set Architecture, Book I	2.02	2005年1月
PowerPC Virtual Environment Architecture, Book II	2.02	2005年1月
PowerPC Operating Environment Architecture, Book III	2.02	2005年1月

ドキュメント構成

ドキュメント・セクション	説明
前付	タイトル・ページ、著作権および免責事項、目次、図目次、表目次
序文	本ドキュメントの説明、関連ドキュメント、責任、およびその他の全般的な情報
変更履歴	前バージョンから本バージョンへの変更の概略のリスト
セクション 1　はじめに 23ページ	SPU アーキテクチャとその目的を大局的に説明する。
セクション 2　SPU アーキテクチャの概要 25ページ	SPU アーキテクチャの概要を説明する。
セクション 3　メモリ - Load/Store 命令 31ページ	SPU ロード/ストア命令をリストアップし、説明する。
セクション 4　定数生成命令 49ページ	SPU の定数生成命令をリストアップし、説明する。
セクション 5　整数および論理命令 57ページ	SPU の整数および論理命令をリストアップし、説明する。
セクション 6　シフトおよびローテート命令 117ページ	SPU のシフト命令とローテート命令をリストアップし、説明する。
セクション 7　比較、分岐、および停止命令 149ページ	SPU の比較命令、分岐命令、および停止（halt）命令をリストアップし、説明する。
セクション 8　分岐ヒント命令 191ページ	SPU の分岐ヒント命令をリストアップし、説明する。
セクション 9　浮動小数点命令 195ページ	SPU の浮動小数点命令をリストアップし、説明する。

セクション 10　制御命令 237ページ	SPU の制御命令をリストアップし、説明する。
セクション 11　チャネル命令 247ページ	チャネル・インターフェースを介した SPU と外部デバイス間の通信に使用する命令を説明する。
セクション 12　SPU 割り込み機能 251ページ	SPU の割り込み機能を説明する。
セクション 13　同期と順序付け 253ページ	SPU の逐次的に順序付けられたプログラミング・モデルを説明する。
付録 A　命令ニーモニック順命令一覧表 259ページ	ニーモニックでソートして SPU 命令をリストアップする。
付録 B　制御生成命令の詳細 265ページ	制御生成命令によって生成されるマスクの詳細を提供する。

バージョン番号体系

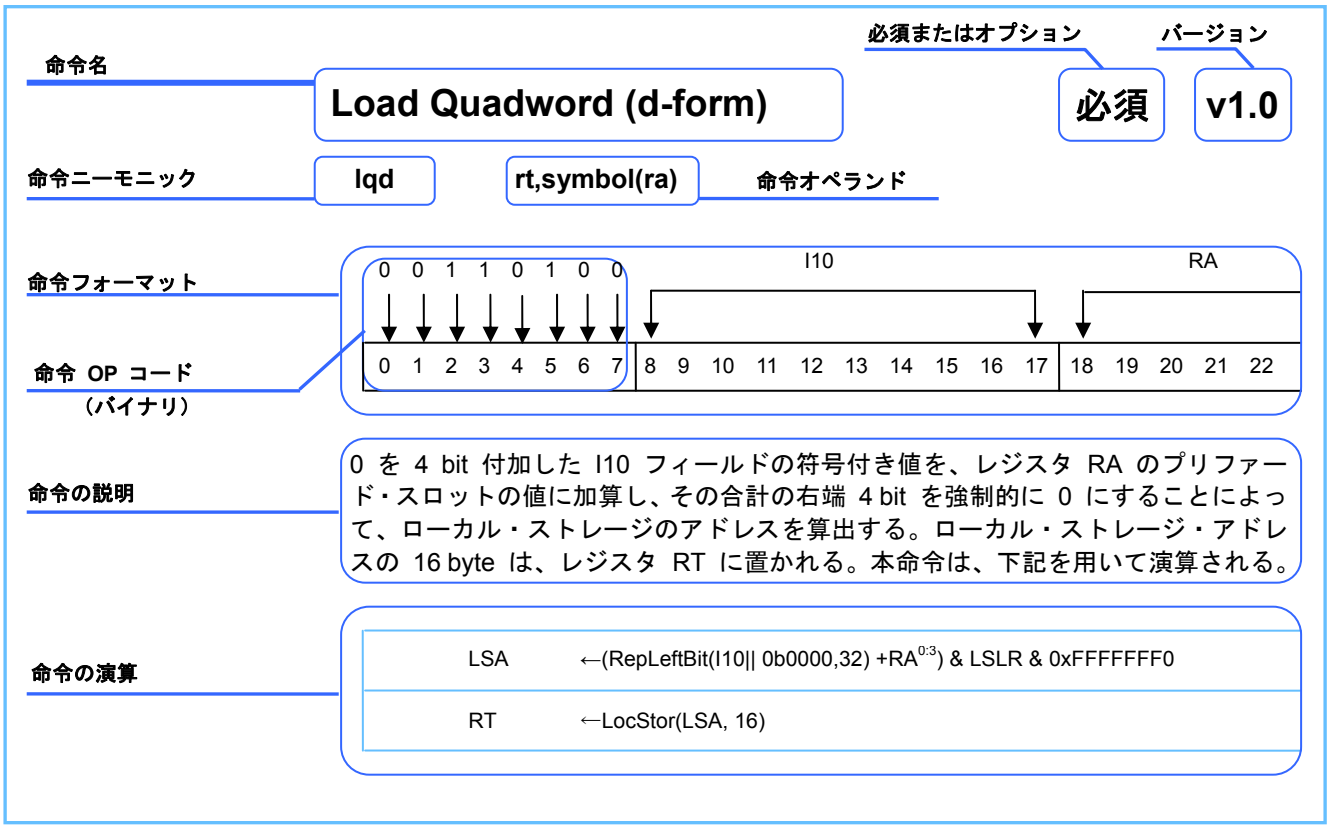
本ドキュメントのバージョン番号は、タイトル・ページおよび各ページのフッタに記されている。バージョン番号の形式は V.xy であり、ここで、

- V はメジャー・バージョン・レベルである。この番号は、アーキテクチャに新しく必須機能が追加された際に増加される。メジャー・レビジョンとマイナー・レビジョンの番号はゼロに設定される。例えば、バージョン 1.12 がバージョン 2.00 になる。
- x はメジャー・レビジョン・レベルである。この番号は、アーキテクチャに新しくオプション機能が追加された際、あるいは、プログラマに影響を及ぼしうる大きな変更が加えられた際に増加される。マイナー・レビジョン・レベルはゼロに設定される。例えば、バージョン 1.12 がバージョン 1.20 になる。
- y はマイナー・レビジョン・レベルである。この番号は、新しい必須機能もオプション機能も含まないようなそれぞれの新リリースについて増加される。例えば、バージョン 1.12 がバージョン 1.13 になる。

命令記述の見方

本ドキュメントに記載されている命令記述の見方を、図 i で説明する。

図 i　　命令記述のフォーマット



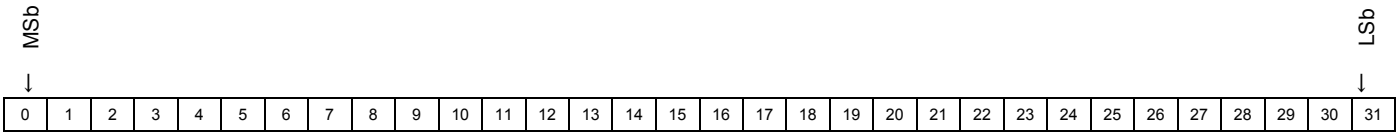
本マニュアルで使用される規約および表記法

バイト順

本ドキュメントを通して、IBM の標準ビッグエンディアン表記法を使用し、バイトは、左から右に昇順で番号を割り当てる。ビッグエンディアンおよびリトルエンディアンのバイト順に関しては、*Cell Broadband Engine™ アーキテクチャ*で解説されている。

ビット順

ビットは、左から右に昇順で番号を割り当て、ビット 0 が最上位ビット（MSb）を、ビット 31 が最下位ビット（LSb）を表わす。



ビットのエンコード

ビットのエンコードの表記は下記のとおりである。

- 16 進数の値は、前に 0x がつけられる。
  例： 0x0A00

- 2 進数の値は、前に 0b がつけられる。
  例： 0b1010

命令、ニーモニック、およびオペランド

本ドキュメントでは、命令、ニーモニック、およびオペランドについて、以下の規約に従う。

- 命令ニーモニックは、**太字**で表記する。例えば、同期命令は、**sync** である。

- 本ドキュメントに記載する各命令記述には、15ページの*図 i* に示すように、命令がオプションと必須のいずれであるか、および、命令が導入されたアーキテクチャのバージョンが示され、ニーモニック、およびオペランドの定形式のリストが含まれる。さらに、アセンブラでサポートされる書式を示すアセンブラ言語のステートメントの例も提供する。

- 変数はイタリック書体で表記する。

レジスタあるいはチャネル、フィールド、およびビット範囲の参照

レジスタおよびチャネルは、フルネーム、またはニーモニックで参照する。ニーモニックは、省略名とも呼ばれる。フィールドは、フィールド名またはビット・ポジションで参照する。

通常、レジスタ・ニーモニックの後ろには、括弧（[ ]）で囲まれたフィールド名やビット・ポジションが続く。例：MSR[R] 。等号の後ろに続く値は、そのフィールドが設定される値を示す。例：MSR[R] = 0 。ビット番号の範囲を参照する場合は、開始および終了ビット番号が括弧で囲まれ、コロンで分けられる。例：[0:34] 。

以下の表では、本ドキュメントにおいてレジスタ、フィールド、およびビット範囲をどのように参照するかを記し、参照の例を与える。

参照のタイプ	フォーマット	例
レジスタの省略名とフィールド名を使用して、特定のレジスタと特定のフィールドを参照する	Register_Short_Name[Field_Name]	MSR[R]
フィールド名を使用して、フィールドを参照する	[Field_Name]	[R]
レジスタの省略名と複数のフィールド名を使用して、特定のレジスタと複数フィールドを参照する	Register_Short_Name[Field_Name1, Field_Name2]	MSR[FE0, FE1]
レジスタの省略名とビット・ポジションを使用して、そのレジスタと複数フィールドを参照する	Register_Short_Name[Bit_Number, Bit_Number]	MSR[52, 55]
レジスタの省略名とビット・ポジションまたはビット範囲を使用して、特定のレジスタとフィールドを参照する	Register_Short_Name[Bit_Number]	MSR[52]
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number]	MSR[39:44]
等号（=）と値が後ろに続くフィールド名は、そのフィールドに対する値を示す	Register_Short_Name[Field_Name]=*n*[1]	MSR[FE0]=0b1 MSR[FE]=0x1
	Register_Short_Name[Bit_Number]=*n*[1]	MSR[52]=0b0 MSR[52]=0x0
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number]=*n*[1]	MSR[39:43]=0b10010 MSR[39:43]=0x11

1. *n* は、括弧の中で指定されたフィールドまたはビットに対する 2 進数または 16 進数の値

レジスタ・トランスファー・ランゲージ（RTL）による命令の定義

本ドキュメントは、通常、PowerPC Architecture™ でのレジスタ・トランスファー・ランゲージ（RTL）の用語および表記法に従う。
RTL の記述は、ほとんどの命令に提供されており、言葉での記述（それが主たる定義である）を明確にすることを意図とする。RTL には下記の規則が適用される。

- **LocStor(*x*,*y*)** は、ローカル・ストレージのロケーション *x* で開始する *y* バイトを参照する。
- **RepLeftBit(*x*,*y*)** は、トータル幅 *y* になるまで左端ビットを複製した *x* の値を返す。
- プログラム・カウンタ（PC）は、オペランドとして使用されるときは実行している命令のアドレスを含み、ターゲットとして使用する場合は次の命令のアドレスを含む。
- RTL 記述で使用されるテンポラリ名は、*表 i* に示す幅を持つ。

表 i　RTL で使用するテンポラリ名およびその幅

テンポラリ名	幅
b, byte, byte1, byte2, c	8 bits
r, s	16 bits
bbbb, EA, QA, t, t0, t1, t2, t3, u, v	32 bits
Q, R, Memdata	128 bits
Rconcat	256 bits
i, j, k, m	Meta（記述専用）

命令フィールド

本ドキュメントに記載されている命令には、*表 ii* に示すフィールドが 1 つ以上含まれることがある。

表 ii　命令フィールド

フィールド	説明
/, *//*, *///*	命令中の予約フィールド。 現在未使用の予約フィールドは、アーキテクチャではチェックしていない個所でも、0 となっている。非互換性を生じずに将来使用できるようにするためである。
I7	7 bit の即値
I8	8 bit の即値
I10	10 bit の即値
I16	16 bit の即値
OP または OPCD	OP コード
RA[18-24]	ソースまたはターゲットとして使用する General Purpose Register（GPR）を指定するためのフィールド
RB[11-17]	ソースまたはターゲットとして使用する GPR を指定するためのフィールド
RC[4-10]	ソースまたはターゲットとして使用する GPR を指定するためのフィールド
RT[25-31]	ターゲットとして使用する GPR を指定するためのフィールド

命令演算表記

本ドキュメントの命令では、*表 iii* に示す表記を使用する。この表では、優先順位の高い順に列記している。表の最初の演算子が最も強く結合される。

表 iii　命令演算表記

表記	説明	注記参照
X_p	レジスタまたは値フィールド *X* のビット *p*	
$X_{p:q}$	レジスタまたは値 *X* のビット *p* から *q* まで	
X^p	レジスタまたは値 *X* のバイト *p*	
$X^{p:q}$	レジスタまたは値 *X* のバイト *p* から *q* まで	
$X_{p::q}$	ビット *p* およびこれに続く合計 *q* ビット分のビット	
$X^{p::q}$	バイト *p* およびこれに続く合計 *q* バイト分のバイト	
$_p0$ および $_p1$	*p* ビットの *0* ストリングおよび *p* ビットの *1* ストリング	1
¬	単項の NOT 演算子	2
, ||	符号付き乗算 符号なし乗算	3
+	2 の補数加算	2
-	2 の補数減算、単項マイナス	2
=, ≠	等しい 等しくない関係	
<, ≤, >, ≥	符号付き比較	
$<^u$, $>^u$	符号なし比較	
&	AND	2
|	OR	2
⊕	Exclusive-OR (a & ¬b | ¬a & b)	2
←	代入	
LSA	Local Storage Address	
LSLR	Local Storage Limit Register	
LocStor(LSA,*width*)	アドレス LSA にある *width* バイトのローカル・ストレージの内容	
if (cond) then... else...	条件実行。else はオプション。then 節や else 節の範囲はインデントで示される。節が単文の場合は、対応する if や else と同じ行内に示される。	
for ... end	For ループ。*to* 節や *by* 節は、反復変数の増分を指定する。*while* 節は、終了条件を与える。	
do ... while (cond)	Do ループ。While 節は終了条件を与える。	
/, //, ///	命令中の予約フィールド。 予約フィールドは、現在は未使用であるが、アーキテクチャではチェックしていなくても、0 としておくこと。非互換性を生じずに将来使用できるようにするためである。	

1. PowerPC の表記法とは異なる。PowerPC では、先行するサブスクリプトではなく、スーパースクリプトを使用する。
2. この演算子の結果は、入力オペランドと同一の幅を持つビットベクトルである。
3. この演算子の結果は、オペランドの幅の合計分の幅を持つビットベクトルである。

変更履歴

本ドキュメントの各リリースは、それ以前にリリースされた全バージョンを置き換える。変更履歴ログは、初版リリース以来、本ドキュメントに対して行なわれた重要な全変更を列挙している。本ドキュメントのこれ以降では、余白のチェンジバーにより、隣接する本文が本ドキュメントの前バージョンから有意に変更されていることを示している。

改訂日付	変更内容	エラータ
2007年 1月 27日	Version 1.2 • 改訂された命令記述形式を図示するために図を更新（15ページ *図 i 命令記述のフォーマット* 参照）。さらに、命令の規約に関する記述も更新（16ページ *命令、ニーモニック、およびオペランド* 参照）。 • Multiply High命令に関するプログラミングの注意を加筆修正してコード例を追加。（77ページ *Multiply High* 参照）。 • IEEE非準拠結果の記述から"非ゼロの"を削除（195ページ *9.1 単精度（拡張範囲モード）* 参照）。 • 全'1'の指数フィールドは無限大ならびに非数(NaN)に使われることを明記（197ページ *表 9-3 倍精度（IEEE モード）の最小値および最大値* 参照）。 • 非正規化数入力の扱いに関する記述を変更（198ページ *9.2.1 単精度フォーマットと倍精度フォーマットの間の変換* 参照）。 • FPSCR[31]の記述から"非ゼロの"を削除（200ページ *9.3 浮動小数点状態および制御レジスタ（Floating-Point Status and Control Register－FPSCR）* 参照）。 • 5個のオプション命令を追加（226ページ *Double Floating Compare Equal*、227ページ *Double Floating Compare Magnitude Equal*、228ページ *Double Floating Compare Greater Than*、229ページ *Double Floating Compare Magnitude Greater Than*、230ページ *Double Floating Test Special Value* 参照）。 • *付録 A* に新命令を追加（259ページ *表 A-1 ニーモニック順命令* 参照）。 • 用語集に編集上の各種変更を実施（267ページ *用語集* 参照）。 • 全体に渡って命令記述形式を改訂。命令の見出しに、命令がオプションか必須か、および、命令がアーキテクチャのどのバージョンで導入されたかを示すようにした。 • 誤記訂正：266ページ *表 B-4 ダブルワード挿入：実効アドレスの右端 4 bit および生成されるマスク。*	
2006年 10月 4日	Version 1.11 • バージョン番号体系を説明（14ページ *バージョン番号体系* 参照）。 • 全体を通じ、16進数と2進数の表記を変更（16ページ *ビットのエンコード* 参照）。 • ビット・エンコーディングの記述と変数表記の規約を変更（16ページ *本マニュアルで使用される規約および表記法* 参照）。 • Select Bits 命令の記述を改訂（115ページ *Select Bits* 参照）。論理右シフトと算術右シフトがどのようにサポートされているかを説明するいくつかの「プログラミングの注意」を改訂（136ページ *Rotate and Mask Halfword* 、137ページ *Rotate and Mask Halfword Immediate* 、138ページ *Rotate and Mask Word* 、139ページ *Rotate and Mask Word Immediate* 参照）。 • インライン・プリフェッチを説明（192ページ *Hint for Branch (r-form)* 参照）。 • 195ページの *9. 浮動小数点命令* の序文を改訂し、浮動小数点命令の結果は実装依存であることを説明する「実装上の注記」を追加。 • 195ページ *表 9-1 単精度（拡張範囲モード）の最小値および最大値*、197ページ *表 9-3 倍精度（IEEE モード）の最小値および最大値*、198ページ *表 9-4 単精度（IEEE モード）の最小値および最大値* を改良。 • 倍精度命令の記述を改良し、各スライスの丸めモードが個別に制御できることを示した（197ページ *9.2* 参照）。 • 非正規化入力がどのように扱われるかの説明を拡張（198ページ *9.2.1* 参照）。 • 浮動小数点状態および制御レジスタにおいて、ビット 20:21 を定義し、ビット 22:23 を再定義（200ページ *9.3* 参照）。 • **lnop** 命令の記述を訂正（240ページ *No Operation (Load)* 参照）。 • 128 ビットの **mtspr** および **mfspr** 命令で 32 ビット値がどのように扱われるかを説明（244ページ *Move from Special-Purpose Register* および 245ページ *Move to Special-Purpose Register* 参照）。 • **rdch** および **wrch** 命令で 32 ビット幅のチャネルがどのように扱われるかを説明（248ページ	YES YES

改訂日付	変更内容	エラータ
	Read Channel および 250ページ *Write Channel* 参照）。 • ローカル・ストレージへの複数アクセスをどのように同期するかをより詳しく説明（256ページ *表 13-3* 参照）。 • ローカル・ストレージの外部アクセスについてより詳しい記述を提供（256ページ *13.6* 参照）。 • いくつかの命令について、RTL記述を簡素化、明瞭化。 • *付録 A プログラミング例* を削除。*付録 B 命令ニーモニック順命令一覧表* が *付録 A* となった。 • 索引を追加（271ページ *索引* 参照）。 • 用語集を追加（267ページ *用語集* 参照）。 • 全体を通じ、「ローカル・ストア」を「ローカル・ストレージ」に変更。 • その他、一貫性、明瞭性のための変更を行なった。 • （日本語版）一部の訳語を改良。 • RTL記述の誤記訂正：47ページ *Generate Controls for Doubleword Insertion (x-form)*、132ページ *Rotate Quadword by Bytes Immediate*、133ページ *Rotate Quadword by Bytes from Bit Shift Count*、135ページ *Rotate Quadword by Bits Immediate*。	
2006年 1月 30日	Version 1.1 • Rotate and Mask Halfword Immediate命令に付随する疑似コードを訂正（137ページ参照）。	
2005年 8月 1日	初版公開	

1. はじめに

ドキュメント「Synergistic Processor Unit（SPU）命令セット・アーキテクチャ（Instruction Set Architecture—ISA）」は、汎用プロセッサと専用ハードウェアの隙間を埋めることを可能とするプロセッサのアーキテクチャについて解説するものである。汎用プロセッサのアーキテクチャの目的は、多種多様なアプリケーションで最高の平均性能を達成することであり、専用ハードウェアの目的は、単一のアプリケーションで最高性能を達成することであるのに対して、本ドキュメントで解説するアーキテクチャは、ゲーム・システム、メディア・システム、およびブロードバンド・システムのための決定的なワークロードにおいて、業界をリードする性能を達成することを目的とする。Synergistic Processor Unit 命令セット・アーキテクチャ（SPU ISA）と Cell Broadband Engine Architecture（CBEA）の目的は、プログラミングの容易さを維持しつつも、エキスパート（リアルタイム）プログラマによる高度な制御を可能にする情報を提供することである。

下記は、SPU のキー・ワークロードである。

- サーフェスのサブディビジョンとレンダリングを含む、グラフィックス・パイプライン
- エンコード、デコード、暗号化、および復号化を含むストリーミング処理
- ゲーム・フィジックス（物理計算）を含むモデリング

同等性能で比較すると、SPU ISA の実装では約半分の消費電力と約半分のチップ面積しか必要としないことから、汎用プロセッサよりも高い費用性能比を実現する。これは、*表 1-1* にリストアップした、SPU ISA アーキテクチャおよび実装の主な特徴によって可能になる。

表 1-1 SPU ISA アーキテクチャおよび実装の主な特徴

特徴	説明
128 bit SIMD 実行ユニットの構成	上述のアプリケーションの多くでは、単一命令/複数データ（SIMD）による並列性が見込まれる。SIMD アーキテクチャにおいては、処理するデータ・エレメントが複数あることで、命令フェッチやデコードのコスト（チップ面積、消費電力）が償われる。128 bit（通常 4 way × 32 bit）の SIMDは、他の汎用プロセッサのアーキテクチャにおける SIMD 処理ユニット、およびそれに対応する既存のコードベースとの共通性を有する。
ソフトウェア管理メモリ	ほとんどのプロセッサは、キャッシュを搭載することでメモリの待ち時間を削減するが、CBEA の SPU では、キャッシュではなく、小さいローカル・メモリを実装している。キャッシュ階層と比較すると、この手法により、バイト当たり必要なチップ面積は約半分になり、アクセス当たりの消費電力も著しく少なくて済む。加えて、リアルタイム・プログラミング向けの高度な制御が提供できる。このアプローチが優位なのは DMA の転送サイズが十分に大きく、十分に予測できる（すなわちデータが必要とされる前に、DMA を実行できる）場合に限られる。理由は、DMA 転送に伴う待ち時間および命令のオーバーヘッドは、キャッシュ・ミスをサービスする待ち時間のオーバーヘッドを超えてしまうからである。
効率的な SRAM 設計をサポートするロード/ストア・アーキテクチャ	SPU ISA マイクロアーキテクチャは、単一ポート（ローカル・ストレージ）メモリを使用した効率的な実装が可能となるように構成されている。
大容量統合レジスタ・ファイル	SPU アーキテクチャの 128 エントリのレジスタ・ファイルによって、レジスタ枯渇を回避するためのレジスタリネーミングなしで、深くパイプライン化した高周波数の実装が可能である。これは特に、ソフトウェア・ループのアンローリング、あるいはその他のインターリーブのテクニックによって待ち時間をカバーする場合に重要になる。現在の高周波数の汎用プロセッサにおいて、リネーム機構は通常、チップ面積および電力のかなりの部分を消費している。
分岐除去の ISA サポート	SPU ISA では、マスクをセットする比較命令を定義しており、そのマスクを用いる 3 オペランド選択命令によって効率良く条件付き代入を実行できる。このような条件付き代入を使用することによって、予測が困難な分岐を回避できる。

予測可能な分岐における、分岐ペナルティ回避のための ISA サポート	分岐を十分早く予測できる場合は、SPU の「分岐ヒント」命令によって、成立した分岐のペナルティをプログラムで回避することが可能になる。このメカニズムは、前回の分岐に関する履歴格納を必要とせず、よってチップ面積と消費電力を節約できる点において、通常の分岐予測のスキームよりも優れている。本 ISA では、分岐命令自体にヒント・ビットを設ける場合には生じる問題を解決している。その問題とは、分岐ターゲットが必要なときにはすでに用意できているように十分早めに分岐を処理するために、命令ストリームの相当な先読み（分岐スキャン）が必要となることである。
グラフィックス指向の単精度（拡張範囲）浮動小数点サポート	ゲーム・アプリケーションに使用されるほとんどのコード基盤は、汎用プロセッサで共通に実装されている IEEE 754 フォーマットとは異なる単精度浮動小数点のフォーマットを仮定している。単精度フォーマットの詳細に関しては、195ページのセクション *9*「*浮動小数点命令*」 を参照のこと。
チャネル・アーキテクチャ	ブロックするチャネルによって SPU 外部である Synergistic Memory Flow Controller（MFC）やシステムのその他部分と通信するという、外部イベントの完了を待つための効率的なメカニズムが提供され、ポーリングや割り込み/待ちループが無くせる。これらは両方とも不必要に電力を消費するものである。
*ユーザ・オンリー*のアーキテクチャ	SPUでは、汎用プロセッサで共通にみられる特定の機能を含まない。具体的には、スーパバイザ・モードをサポートしていない。

2. SPU アーキテクチャの概要

本セクションでは、SPU アーキテクチャの概要を説明する。

SPU アーキテクチャは、128 個の General-Purpose Register（GPR）一式を定義し、各レジスタは 128 bit データを収容できる。レジスタは固定小数点および浮動小数点のデータを保持するために用いられる。命令は、レジスタ中に同じフォーマットの複数オペランドがあるとして、レジスタの全幅に対して演算を行なう。

SPU は、ハーフワード（16 bit）とワード（32 bit）の符号付き整数をサポートし、8 bit の符号なし整数は制限つきでサポートする。数表現は、2 の補数である。

SPU は、IEEE 754 フォーマットの単精度（32 bit）と倍精度（64 bit）の浮動小数点データをサポートしている。しかし、単精度演算については、IEEE 754 完全準拠の実装にはなっていない。

SPU アーキテクチャは、コンディション・コード・レジスタを使用しない。代わりに、比較演算で 0（偽）、あるいは –1（真）のいずれかの結果を設定し、この結果は比較されるオペランドと同一の幅を持つ。これらの結果はビット単位のマスキング、Select 命令、あるいは条件分岐に用いることができる。

SPU のロード/ストアは、ローカル・ストレージと呼ばれるプライベート・メモリにアクセスする。SPU のロード/ストアは、GPR とローカル・ストレージ間でクワッドワードを転送する。実装によって異なったローカル・ストレージのサイズを備えていることがある。しかしながら、ローカル・ストレージ・アドレス空間は 4 GB に制限されている。

SPU ではチャネル・インターフェースを通して、外部デバイスとデータを送受信することができる。SPU チャネル命令は、GPR とチャネル・インターフェース間でクワッドワード（128 bit）を転送する。最大 128 個のチャネルがサポートされる。2 つのチャネルは、Interrupt Return 命令（**iret**）が使用するアドレスを保持している Save-and-Restore Register 0 (SRR0) にアクセスすると定義されている。また、SPU では最大 128 個の Special-Purpose Register（SPR）がサポートされる。Move To Special Purpose Register（**mtspr**）命令と Move From Special Purpose Register（**mfspr**）命令は、GPR と SPR 間で 128 bit データを移動する。

また、SPU では外部条件と呼ばれるステータスシグナルを監視する。Branch Indirect and Set Link If External Data（**bisled**）命令は、外部条件のステータスに基づいて条件分岐する。外部条件が真の場合にアドレス 0 の割り込みハンドラに分岐するように、SPU 割り込み機能を設定することが可能である。

2.1 データ表現

SPU アーキテクチャでは、下記を定義する。

- 8-bit バイト
- 16-bit ハーフワード
- 32-bit ワード
- 64-bit ダブルワード
- 128-bit クワッドワード

バイト順は、ハーフワード、ワード、ダブルワード、およびクワッドワードを形成するバイトの、メモリ中での順番を定義する。SPU は、最上位バイト（MSB）順をサポートする。MSB 順はまた*ビッグエンディアン*とも呼ばれ、最上位バイトは、格納単位の最も若いアドレスのバイト・ポジション（バイト 0）に置かれる。本ドキュメントでの命令の記述は、メモリ中で見えるのと同じく、上位アドレスのバイトが右側に現れるようになっている。

各種の幅の格納単位におけるビットおよびバイトのナンバリングの規約は、*表 2-1* にリストアップした図にて示される。

表 2-1 ビットおよびバイト・ナンバリングの図

示す内容	参照先
ハーフワードのビットおよびバイト・ナンバリング	26ページの *図 2-1*
ワードのビットおよびバイト・ナンバリング	26ページの *図 2-2*
ダブルワードのビットおよびバイト・ナンバリング	26ページの *図 2-3*
クワッドワードのビットおよびバイト・ナンバリング	27ページの *図 2-4*
データ型のレジスタ・レイアウト	28ページの *図 2-5*

これらの規約は、整数データおよび浮動小数点データ（最上位バイトにその符号と、少なくとも指数部の先頭部分を保持する）に適用される。図では、上にバイト番号を、下にビット番号を示す。

図 2-1 ハーフワードのビットおよびバイト・ナンバリング

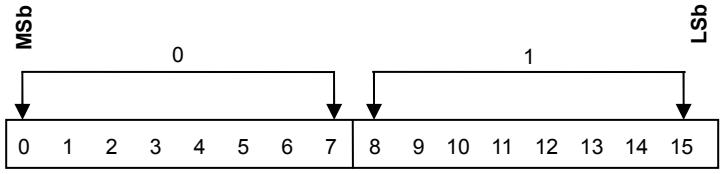


図 2-2 ワードのビットおよびバイト・ナンバリング

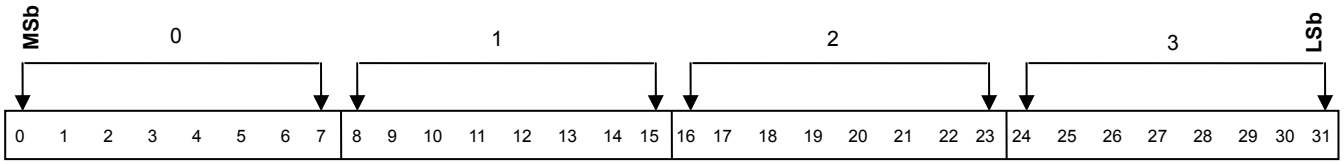


図 2-3 ダブルワードのビットおよびバイト・ナンバリング

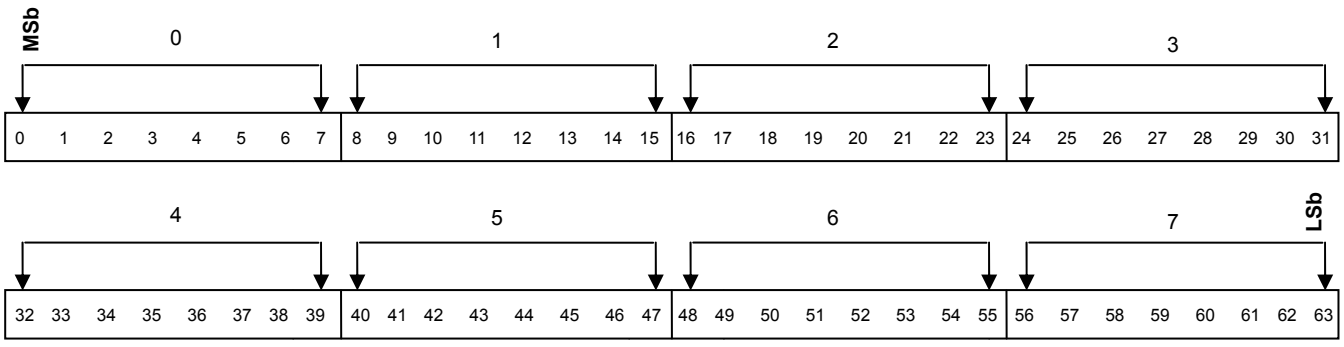
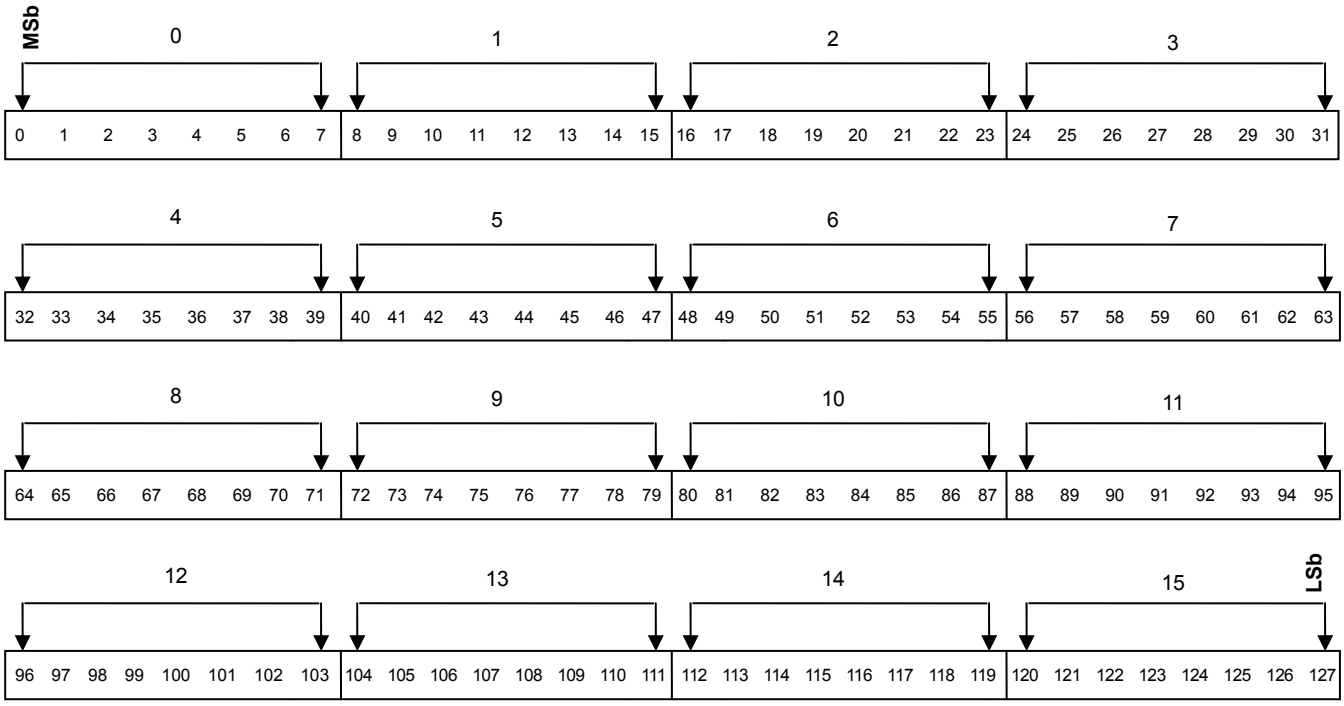


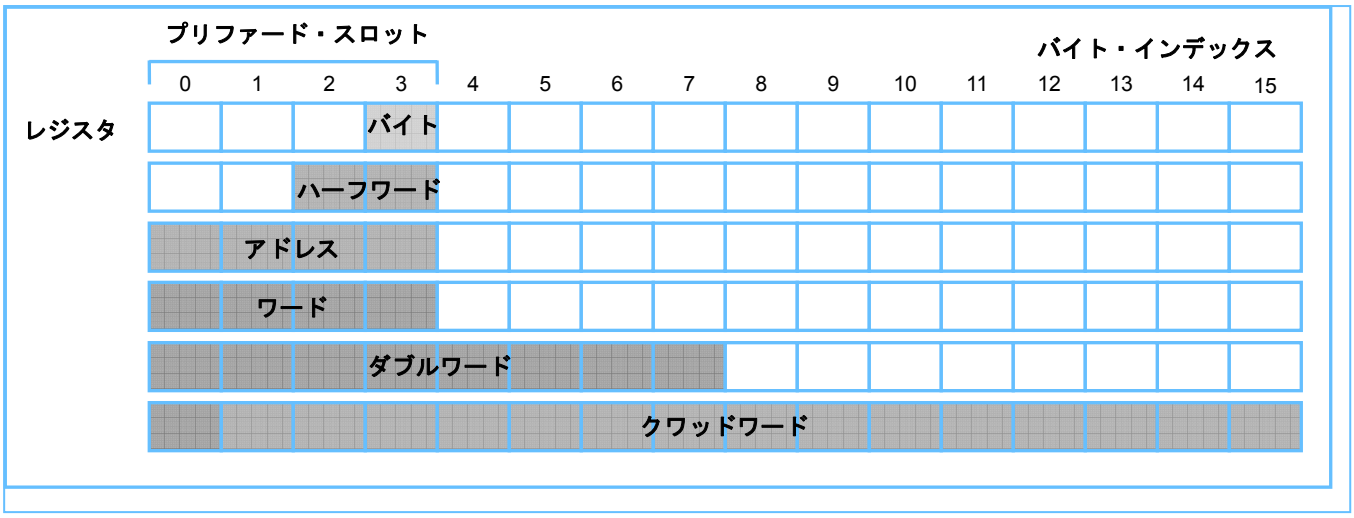
図 2-4　クワッドワードのビットおよびバイト・ナンバリング



2.2 レジスタ内のデータ・レイアウト

すべての GPR は、128 bit 幅である。レジスタの一番左のワード（バイト 0、1、2、および 3）をプリファード・スロット（preferred slot）と呼ぶ。命令がスカラー・オペランドやアドレスを使用あるいは生成するとき、値はプリファード・スロットに入っている。バイト、ハーフワード、ワード、およびダブルワードをストアする際に役立つストア・アシスト命令のセットが用意されている。図 2-5 に、GPR におけるこれらのデータ型のレイアウトを示す。

図 2-5　データ型のレジスタ・レイアウト



2.3 命令フォーマット

命令フォーマットには、6 種類の基本的なフォーマットがある。命令はすべて 32 bit 長である。これらのフォーマットのマイナー・バリエーションも使用される。メモリ中の命令は、ワード境界にアラインする必要がある。これらの命令フォーマットを*図 2-6* から*図 2-11* に示す。
注意：OP コード・フィールドは、本ドキュメントを通して、2 進数で表現される。

*図 2-6　**RR** 命令フォーマット*

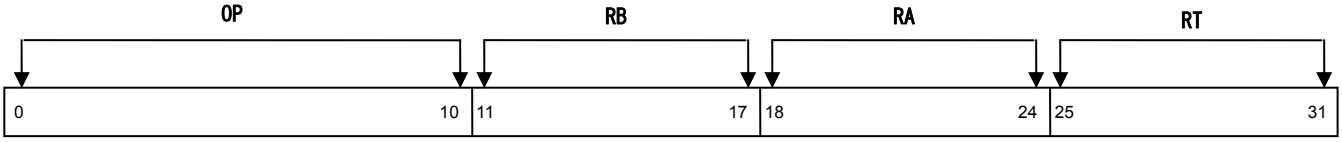


*図 2-7　**RRR** 命令フォーマット*

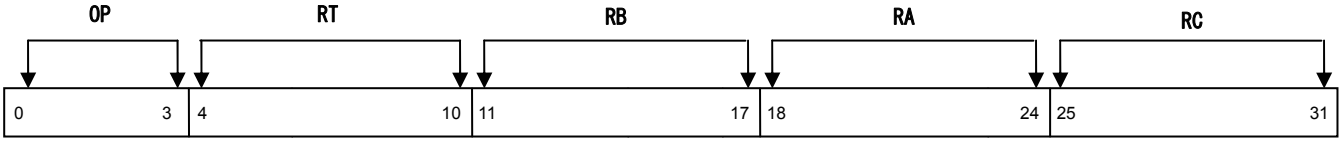


*図 2-8 **RI7** 命令フォーマット*

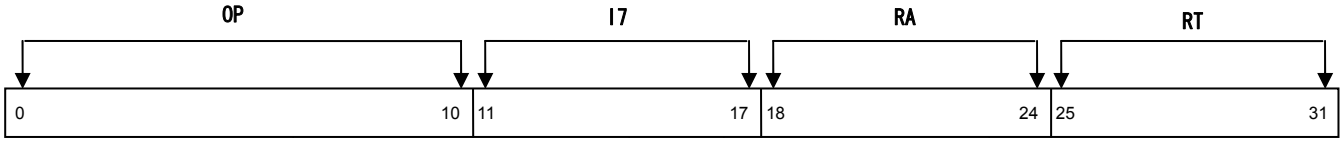


*図 2-9 **RI10** 命令フォーマット*

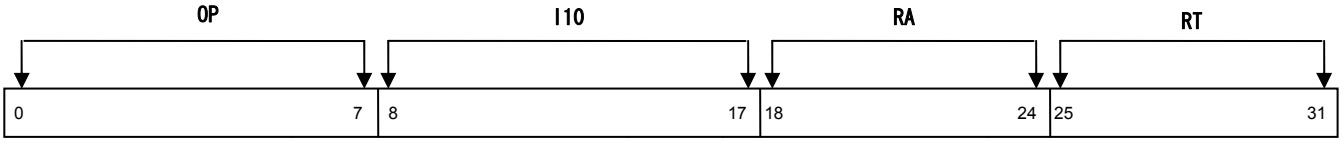


*図 2-10 **RI16** 命令フォーマット*

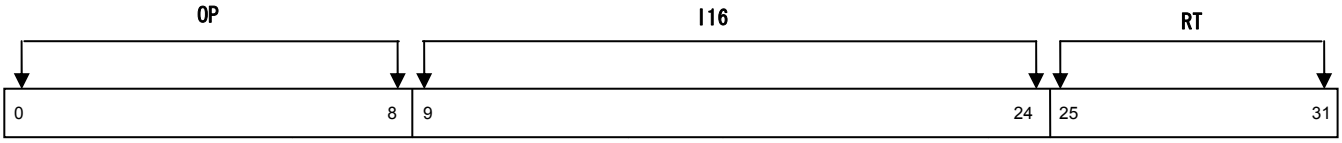
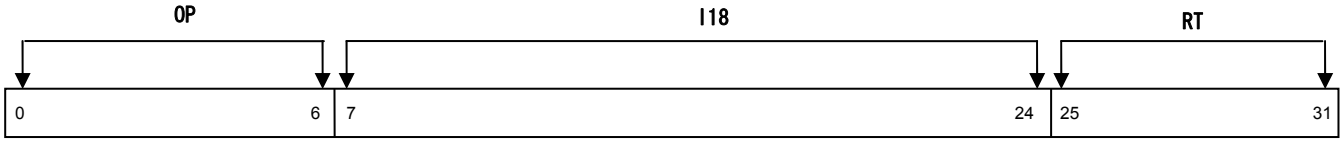


*図 2-11 **RI18** 命令フォーマット*



3. メモリ - ロード/ストア命令

本セクションでは、 SPU のロード/ストア命令をリストアップし、説明する。

SPU アーキテクチャでは、バイト・アドレス指定であるプライベート・メモリ（ローカル・ストレージとも呼ぶ）を定義している。ロード/ストア命令では、1 個か 2 個のレジスタや 1 個の即値からなるオペランドを組み合わせ、メモリ・オペランドの実効アドレスを生成する。アラインされた 16 byte 長のクワッドワードのみロード、ストアできる。よって、実効アドレスの一番右の 4 bit は常に無視され、0 と見なされる。

SPU ローカル・ストレージ・アドレス空間のサイズは 2^{32} byte である。しかし、各実装での実メモリ・サイズは概してこれよりも小さい。メモリの実効サイズは、Local Storage Limit Register（LSLR）によって指定される。実装によっては、LSLR にアクセスする手段を提供している場合がある。しかしながら、これらの手段は SPU Instruction Set Architecture の範囲外である。実装によっては、LSLR の値の変更が許されることがある。しかしながら、SPU が実行中のときに LSLR を変更してはいけない。すべての実効アドレスは、メモリを参照する前に、LSLR と AND 演算される。LSLR は、メモリを実際よりも小さく見せるために使用でき、よって、より小さいメモリ・サイズ向けにコンパイルされたプログラムとの互換性が提供される。LSLR の値はメモリの実効サイズを制御するマスクである。この値は、必ず下記の性質を持たねばならない。

- メモリの実効サイズを、実メモリ・サイズより小さいもしくは等しい値に制限すること。
- 単調（monotonic）であること。すなわち、マスクの最下位 4 ビットが 1 であり、また、最下位から最上位へとビットを見た際、‘1’ から ‘0’ への変化は多くても 1 つ、そして ‘0’ から ‘1’ への変化は存在しない。つまり、値は必ず 2^n-1 であり、このとき n は $\log_2$（実効メモリ・サイズ）である。

この効果として、実効サイズの最後のバイトを越えるメモリへの参照がラップされる。つまり、実効サイズのモジュロと解釈される。この定義により、有効性をチェックする前にアドレスをロードに使用でき、メモリ待ち時間を他の演算とオーバーラップさせることがより容易になる。

クワッドワードよりも小さいサイズのストアは、ロード - 更新 - ストアのシーケンスによって遂行される。このタイプのシーケンスのための「アシスト」命令のグループが用意されている。アシスト命令の名称は、**"Generate Control"** で始まる。本セクションでは、これらの命令についても説明する。例えば、40ページの「*Generate Controls for Byte Insertion (d-form)*」を参照されたい。

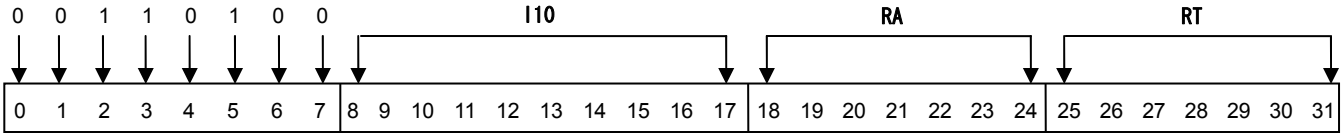
標準的システム構成では、SPU ローカル・ストレージは外部からのアクセスが可能である。よって、SPU プログラムの実行過程において、SPU メモリが非同期に更新される可能性がある。SPU プログラムによるローカル・ストレージへのすべての参照（ロード、ストア）、および SPU メモリへのアラインされた外部からの参照は、アトミックである。アラインされていない参照はアトミックではなく、このような操作の一部分が、SPU で実行しているプログラムから観測される場合がある。LSLR の例とそれに対応するローカル・ストレージ・アドレス空間サイズの関係を*表 3-1* に示す。

表 3-1 LSLR 値および対応するローカル・ストレージ・サイズの例

LSLR	ローカル・ストレージ・サイズ
0x0003FFFF	256 KB
0x0001FFFF	128 KB
0x0000FFFF	64 KB
0x00007FFF	32 KB

Load Quadword (d-form)　必須　v1.0

lqd　　rt,symbol(ra)



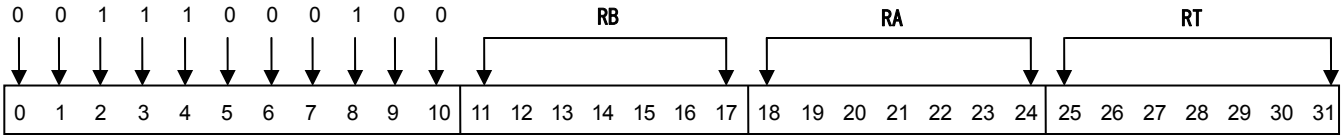
0 を 4 bit 付加した I10 フィールドの符号付き値を、レジスタ RA のプリファード・スロットの値に加算し、その合計の右端 4 bit を強制的に 0 にすることによって、ローカル・ストレージのアドレスを算出する。ローカル・ストレージ・アドレスの 16 byte を、レジスタ RT に置く。本命令は、下記の式を用いて演算される。

LSA	← (RepLeftBit(I10 || 0b0000,32) + $RA^{0:3}$) & LSLR & 0xFFFFFFF0
RT	← LocStor(LSA, 16)

Load Quadword (x-form)

必須　v1.0

lqx　　rt,ra,rb

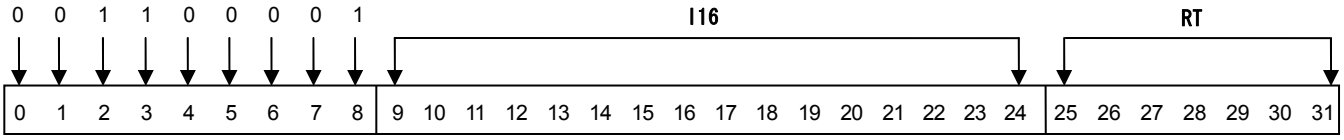


レジスタ RA のプリファード・スロットの値を、レジスタ RB のプリファード・スロットの値に加算し、その合計の右端 4 bit を強制的に 0 にすることによって、ローカル・ストレージのアドレスを算出する。ローカル・ストレージ・アドレスの 16 byte を、レジスタ RT に置く。本命令は、下記の式を用いて演算される。

LSA	← $(RA^{0:3} + RB^{0:3})$ & LSLR & 0xFFFFFFF0
RT	← LocStor(LSA,16)

Load Quadword (a-form)　必須　v1.0

lqa　　rt,symbol

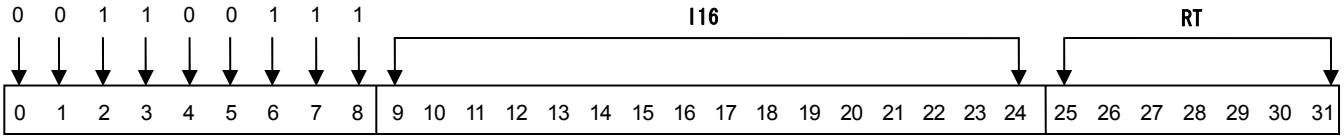


0 を 2 bit 付加した I16 フィールドの値の最上位ビットをコピーして左に拡張したものをローカル・ストレージ・アドレスとして使用する。ローカル・ストレージ・アドレスの 16 byte を、レジスタ RT にロードする。

LSA	← RepLeftBit(I16 || 0b00,32) & LSLR & 0xFFFFFFF0
RT	← LocStor(LSA,16)

Load Quadword Instruction Relative (a-form) 必須 v1.0

lqr rt,symbol

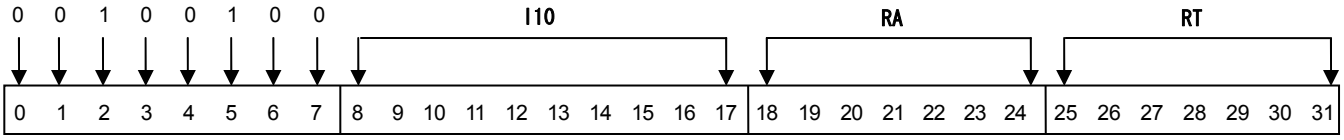


0 を 2 bit 付加した I16 フィールドの値をプログラム・カウンタ（PC）に加算し、ローカル・ストレージ・アドレスを形成する。ローカル・ストレージ・アドレスの 16 byte を、レジスタ RT にロードする。

LSA	← RepLeftBit(I16 || 0b00,32) + PC) & LSLR & 0xFFFFFFF0
RT	← LocStor(LSA,16)

Store Quadword (d-form)　必須　v1.0

stqd　　rt,symbol(ra)



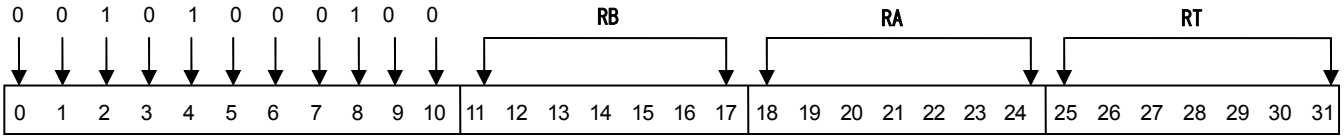
0 を 4 bit 付加した I10 フィールドの符号付き値を、レジスタ RA のプリファード・スロットの値に加算し、その合計の右端 4 bit を強制的に 0 にすることによって、ローカル・ストレージのアドレスを算出する。レジスタ RT の値を、ローカル・ストレージ・アドレスに格納する。

LSA	← (RepLeftBit(I10 || 0b0000,32) + $RA^{0:3}$) & LSLR & 0xFFFFFFF0
LocStor(LSA,16)	← RT

Store Quadword (x-form)

必須　v1.0

stqx　　rt,ra,rb

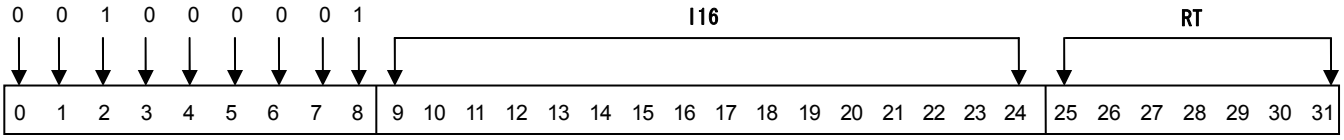


レジスタ RA のプリファード・スロットの値をレジスタ RB のプリファード・スロットの値に加算し、その合計の右端 4 bit を強制的に 0 にすることによって、ローカル・ストレージのアドレスを算出する。レジスタ RT の値を、ローカル・ストレージ・アドレスに格納する。

LSA	← $(RA^{0:3} + RB^{0:3})$ & LSLR & 0xFFFFFFF0
LocStor(LSA,16)	← RT

Store Quadword (a-form)　必須　v1.0

stqa　　rt,symbol

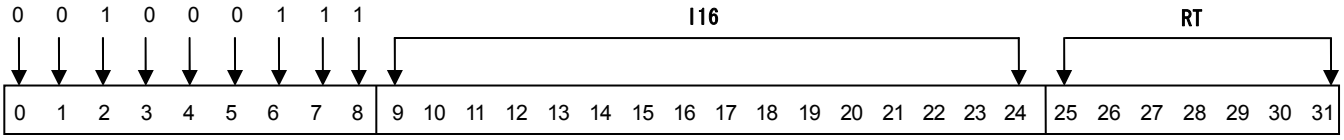


0 を 2 bit 付加した I16 フィールドの値の最上位ビットをコピーして左に拡張したものを、ローカル・ストレージ・アドレスとして使用する。レジスタ RT の値を、ローカル・ストレージ・アドレスで指定されたロケーションに格納する。

LSA	← RepLeftBit(I16 || 0b00,32) & LSLR & 0xFFFFFFF0
LocStor(LSA,16)	← RT

Store Quadword Instruction Relative (a-form)　必須　v1.0

stqr　　rt,symbol

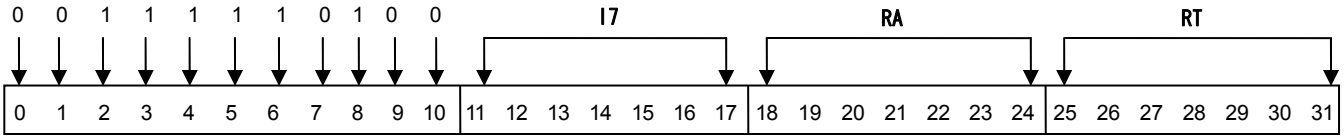


0 を 2 bit 付加した I16 フィールドの値の最上位ビットをコピーして左に拡張したものをプログラム･カウンタ（PC）に加算し、ローカル・ストレージ・アドレスを形成する。レジスタ RT の値を、ローカル・ストレージ・アドレスで指定されたロケーションに格納する。

LSA	← (RepLeftBit(I16 || 0b00,32) + PC) & LSLR & 0xFFFFFFF0
LocStor(LSA,16)	← RT

Generate Controls for Byte Insertion (d-form)　必須　v1.0

cbd　　rt,symbol(ra)



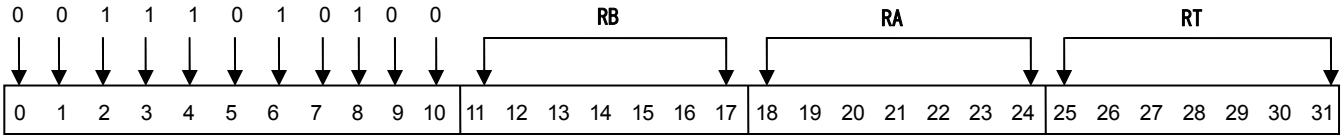
符号付きの I7 フィールドの値をレジスタ RA のプリファード・スロットの値に加算することによって、4 bit のアドレスを算出する。このアドレスは、対象バイトのクワッドワード内でのポジションを決定するために使用される。このポジションを基に、Shuffle Bytes（**shufb**）命令で使用できるマスクを生成する。**shufb** 命令では（事前にロードされた）クワッドワード内の指定ポジションにバイトを挿入する。このバイトは、**shufb** 命令の RA オペランドのプリファード・スロット中の右端のバイト・ポジションから取得する。

生成されるマスクの詳細に関しては、265ページの *付録 B. 制御生成命令の詳細* を参照のこと。

t	← $(RA^{0:3}$ + RepLeftBit(I7,32)) & 0x0000000F
RT	← 0x101112131415161718191A1B1C1D1E1F
RT^t	← 0x03

Generate Controls for Byte Insertion (x-form)　必須　v1.0

cbx　　rt,ra,rb



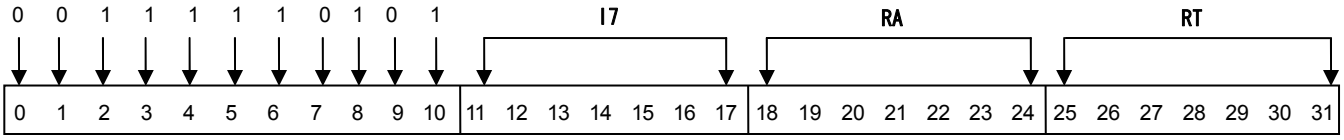
レジスタ RA のプリファード・スロットの値を、レジスタ RB のプリファード・スロットの値に加算することによって、4 bit のアドレスを算出する。このアドレスは、対象バイトのクワッドワード内でのポジションを決定するために使用される。このポジションを基に、**shufb** 命令で使用できるマスクを生成する。**shufb** 命令では（事前にロードされた）クワッドワード内の指定ポジションにバイトを挿入する。このバイトは、**shufb** 命令の RA オペランドのプリファード・スロット中の右端のバイト・ポジションから取得する。

生成されるマスクの詳細に関しては、265ページの *付録 B. 制御生成命令の詳細* を参照のこと。

t	← $(RA^{0:3} + RB^{0:3})$ & 0x0000000F
RT	← 0x101112131415161718191A1B1C1D1E1F
RT^t	← 0x03

Generate Controls for Halfword Insertion (d-form)　必須　v1.0

chd　　rt,symbol(ra)



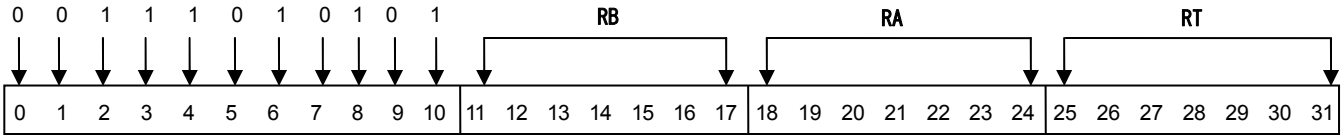
符号付きの I7 フィールドの値をレジスタ RA のプリファード・スロットの値に加算し、最下位ビットを強制的に 0 にすることによって、4 bit のアドレスを算出する。このアドレスは、クワッドワード内のアラインされたハーフワードのポジションを決定するために使用される。このポジションを基に、**shufb** 命令で使用できるマスクを生成する。**shufb** 命令では、クワッドワード内の指定ポジションにハーフワードを挿入する。このハーフワードは、**shufb** 命令の RA オペランドのプリファード・スロット中の右端の 2 byte から取得する。

生成されるマスクの詳細に関しては、265ページの *付録 B. 制御生成命令の詳細* を参照のこと。

t	← $(RA^{0:3}$ + RepLeftBit(I7,32)) & 0x0000000E
RT	← 0x101112131415161718191A1B1C1D1E1F
$RT^{t::2}$	← 0x0203

Generate Controls for Halfword Insertion (x-form)　必須　v1.0

chx　　**rt,ra,rb**



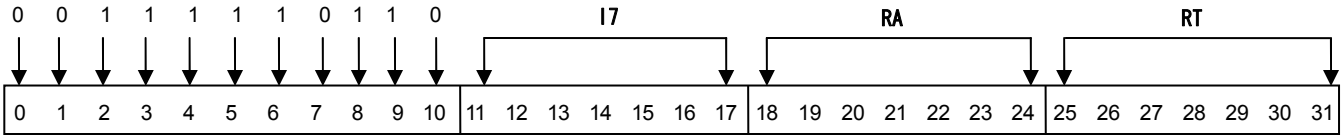
レジスタ RA のプリファード・スロットの値をレジスタ RB のプリファード・スロットの値に加算し、最下位ビットを強制的に 0 にすることによって、4 bit のアドレスを算出する。このアドレスは、クワッドワード内のアラインされたハーフワードのポジションを決定するために使用される。このポジションを基に、**shufb** 命令で使用できるマスクを生成する。**shufb** 命令では、クワッドワード内の指定ポジションにハーフワードを挿入する。このハーフワードは、**shufb** 命令の RA オペランドのプリファード・スロット中の右端の 2 byte から取得する。

生成されるマスクの詳細に関しては、265ページの *付録 B. 制御生成命令の詳細* を参照のこと。

t	← $(RA^{0:3} + RB^{0:3})$ & 0x0000000E
RT	← 0x101112131415161718191A1B1C1D1E1F
$RT^{t::2}$	← 0x0203

Generate Controls for Word Insertion (d-form)　必須　v1.0

cwd　　rt,symbol(ra)



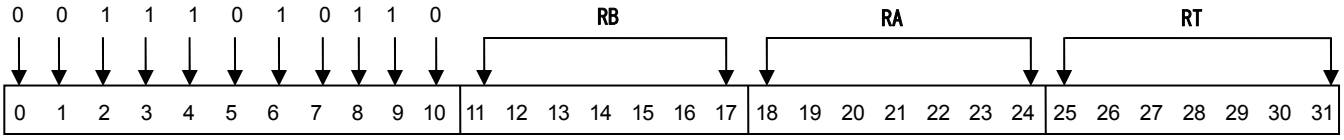
符号付きの I7 フィールドの値をレジスタ RA のプリファード・スロットの値に加算し、最下位の 2 bit を強制的に 0 にすることによって、4 bit のアドレスを算出する。このアドレスは、クワッドワード内のアラインされたワードのポジションを決定するために使用される。このポジションを基に、**shufb** 命令で使用できるマスクを生成する。**shufb** 命令では、クワッドワード内の指定ポジションにワードを挿入する。このワードは、**shufb** 命令の RA オペランドのプリファード・スロットから取得する。

生成されるマスクの詳細に関しては、265ページの *付録 B. 制御生成命令の詳細* を参照のこと。

t	← $(RA^{0:3}$ + RepLeftBit(I7,32)) & 0x0000000C
RT	← 0x101112131415161718191A1B1C1D1E1F
$RT^{t::4}$	← 0x00010203

Generate Controls for Word Insertion (x-form) 必須 v1.0

cwx **rt,ra,rb**



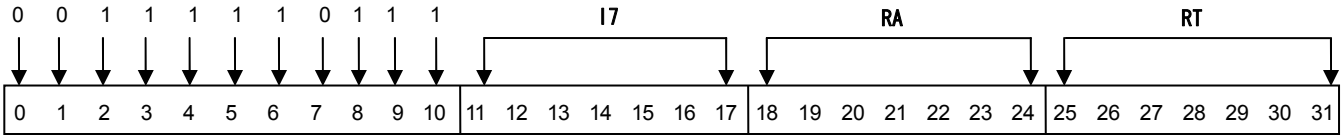
レジスタ RA のプリファード・スロットの値をレジスタ RB のプリファード・スロットの値に加算し、最下位の 2 bit を強制的に 0 にすることによって、4 bit のアドレスを算出する。このアドレスは、クワッドワード内のアラインされたワードのポジションを決定するために使用される。このポジションを基に、**shufb** 命令で使用できるマスクを生成する。**shufb** 命令では、クワッドワード内の指定ポジションにワードを挿入する。このワードは、**shufb** 命令の RA オペランドのプリファード・スロットから取得する。

生成されるマスクの詳細に関しては、265ページの *付録 B. 制御生成命令の詳細* を参照のこと。

t	← $(RA^{0:3} + RB^{0:3})$ & 0x0000000C
RT	← 0x101112131415161718191A1B1C1D1E1F
$RT^{t:4}$	← 0x00010203

Generate Controls for Doubleword Insertion (d-form)　必須　v1.0

cdd　　rt,symbol(ra)



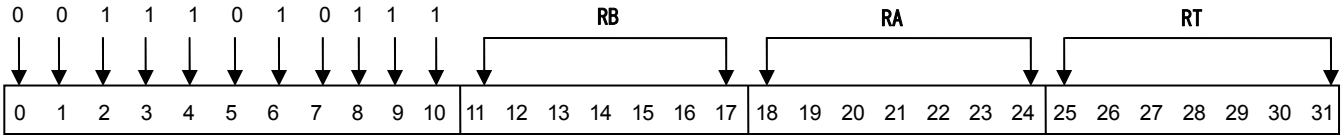
符号付きの I7 フィールドの値をレジスタ RA のプリファード・スロットの値に加算し、最下位の 3 bit を強制的に 0 にすることによって、4 bit のアドレスを算出する。このアドレスは、クワッドワード内のアラインされたダブルワードのポジションを決定するために使用される。このポジションを基に、**shufb** 命令で使用できるマスクを生成する。**shufb** 命令では、クワッドワード内の指定ポジションにダブルワードを挿入する。このダブルワードは、**shufb** 命令の RA オペランドの左端 8 byte から取得する。

生成されるマスクの詳細に関しては、265ページの *付録 B. 制御生成命令の詳細* を参照のこと。

t	← $(RA^{0:3}$ + RepLeftBit(I7,32)) & 0x00000008
RT	← 0x101112131415161718191A1B1C1D1E1F
$RT^{t::8}$	← 0x0001020304050607

Generate Controls for Doubleword Insertion (x-form) 必須 v1.0

cdx **rt,ra,rb**



レジスタ RA のプリファード・スロットの値をレジスタ RB のプリファード・スロットの値に加算し、最下位の 3 bit を強制的に 0 にすることによって、4 bit のアドレスを算出する。このアドレスは、対象ダブルワードのクワッドワード内でのポジションを決定するために使用される。このポジションを基に、**shufb** 命令で使用できるマスクを生成する。**shufb** 命令では、クワッドワード内の指定ポジションにダブルワードを挿入する。このクワッドワードは、**shufb** 命令の RA オペランドの左端 8 byte から取得する。

生成されるマスクの詳細に関しては、265ページの *付録 B. 制御生成命令の詳細* を参照のこと。

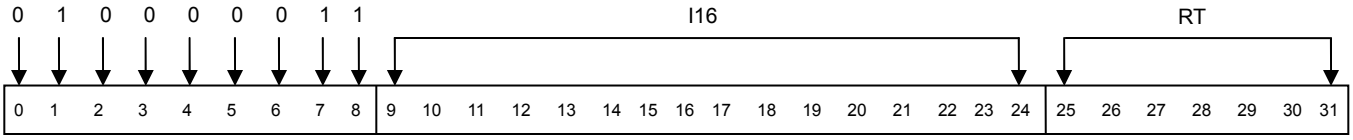
t	← $(RA^{0:3} + RB^{0:3})$ & 0x00000008
RT	← 0x101112131415161718191A1B1C1D1E1F
$RT^{t::8}$	← 0x0001020304050607

4. 定数生成命令

本セクションでは、SPU の定数生成命令をリストアップし、説明する。

Immediate Load Halfword　　必須　v1.0

ilh　　rt,symbol



ハーフワード・スロット 8個それぞれに、下記を行なう。

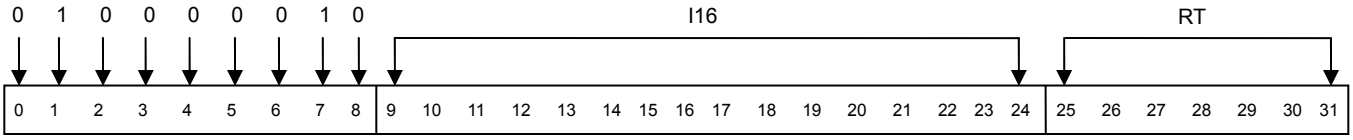
- I16 フィールドの値を、レジスタ RT に置く。

プログラミングの注意：Immediate Load Byte 命令はない。しかし、その命令機能は、**ilh** 命令の I16 フィールドに適切な値を与えれば実現できる。

s	← I16
$RT^{0:1}$	← s
$RT^{2:3}$	← s
$RT^{4:5}$	← s
$RT^{6:7}$	← s
$RT^{8:9}$	← s
$RT^{10:11}$	← s
$RT^{12:13}$	← s
$RT^{14:15}$	← s

Immediate Load Halfword Upper 必須 v1.0

ilhu **rt,symbol**



ワード・スロット 4個それぞれに、下記を行なう。

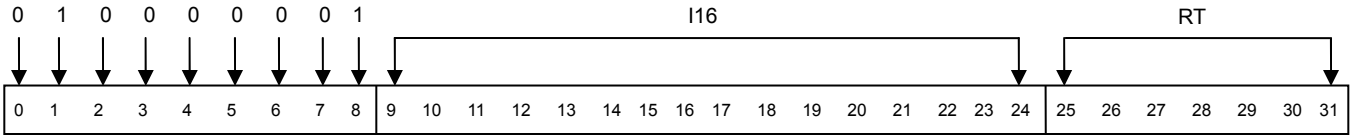
- I16 フィールドの値を、ワードの左端 16 bit に置く。
- ワードの残りのビットを 0 に設定する。

プログラミングの注意：この命令を Immediate Or Halfword Lower（**iohl**）と併せて使用すれば、レジスタのワードの各スロットに任意の 32 bit 値を形成することができる。また、この命令を単独で使用して仮数部が最大 7 bit の有効桁数を持つ即値の浮動小数点定数をロードすることも可能である。

t	← I16 || 0x0000
$RT^{0:3}$	← t
$RT^{4:7}$	← t
$RT^{8:11}$	← t
$RT^{12:15}$	← t

Immediate Load Word 必須 v1.0

il rt,symbol



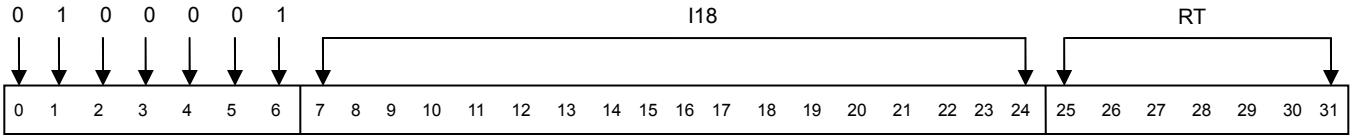
ワード・スロット 4個それぞれに、下記を行なう。

- I16 フィールドの値を、左端のビットを複製することによって、32 bit に拡張する。
- 結果の値を、レジスタ RT に置く。

t	← RepLeftBit(I16,32)
$RT^{0:3}$	← t
$RT^{4:7}$	← t
$RT^{8:11}$	← t
$RT^{12:15}$	← t

Immediate Load Address 必須 v1.0

ila **rt,symbol**



ワード・スロット 4個それぞれに、下記を行なう。

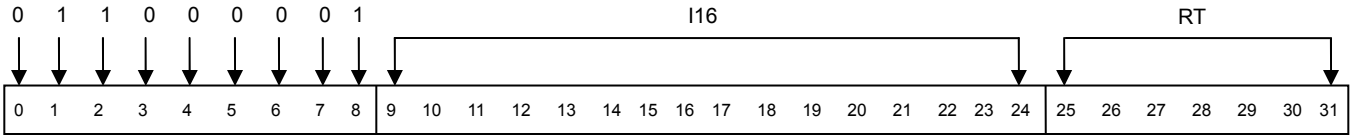
- I18 フィールドの値を、変更せずにレジスタ RT の右端 18 bit に置く。
- レジスタ RT の残りのビットを 0 に設定する。

プログラミングの注意 : Immediate Load Address は、アドレスや小さい定数などの即値を、符号拡張せずにロードするために使用できる。

t	$\leftarrow_{14} 0 \,\|\|\, I18$
$RT^{0:3}$	$\leftarrow t$
$RT^{4:7}$	$\leftarrow t$
$RT^{8:11}$	$\leftarrow t$
$RT^{12:15}$	$\leftarrow t$

Immediate Or Halfword Lower 必須 v1.0

iohl rt,symbol



ワード・スロット 4個それぞれに、下記を行なう。

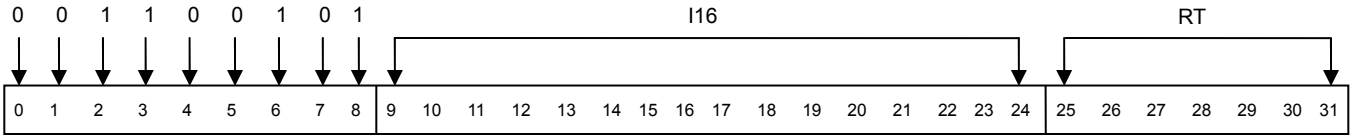
- I16 フィールドの値の前に 0 を付加し、レジスタ RT の値と論理 OR 演算する。
- 結果をレジスタ RT に置く。

プログラミングの注意：Immediate Or Halfword Lower は、Immediate Load Halfword Upper と併せて使用することで 32 bit の即値をロードすることができる。

t	← 0x0000 || I16
$RT^{0:3}$	← $RT^{0:3}$ | t
$RT^{4:7}$	← $RT^{4:7}$ | t
$RT^{8:11}$	← $RT^{8:11}$ | t
$RT^{12:15}$	← $RT^{12:15}$ | t

Form Select Mask for Bytes Immediate 必須 v1.0

fsmbi **rt,symbol**



I16 フィールドの各ビットを 8 回コピーすることによって、レジスタ RT にマスクを生成する。オペランドのビットは、左から右に、結果のバイトに対応する。

プログラミングの注意: この命令は、Select Bits 命令で使用するマスクを生成するために使用できる。また、ハーフワード、ワード、およびダブルワードに対するマスクの生成にも使用できる。

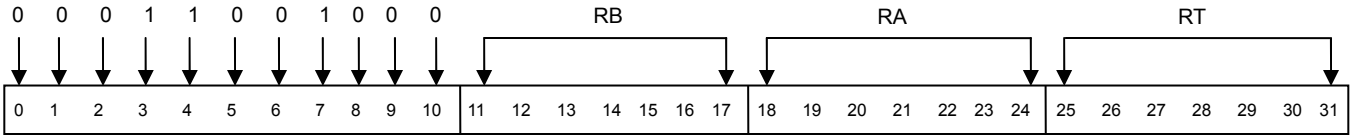
```
s ← I16
for j = 0 to 15
      If s_j = 0 then r^j ← 0x00
      else            r^j ← 0xFF
end
RT ← r
```

5. 整数および論理命令

本セクションでは、SPU の整数および論理命令をリストアップし、説明する。

Add Halfword 必須 v1.0

ah **rt,ra,rb**



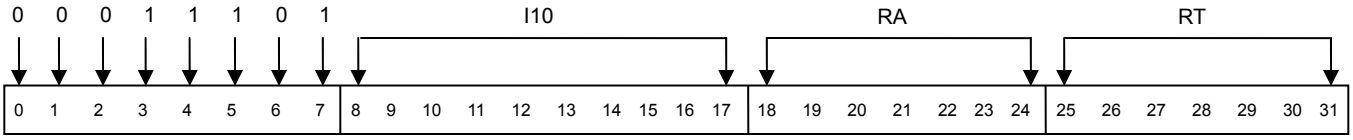
ハーフワード・スロット 8個それぞれに、下記を行なう。

- レジスタ RA のオペランドをレジスタ RB のオペランドに加算する。
- 16 bit の結果を RT に置く。
- オーバーフローとキャリーは、検出しない。

$RT^{0:1}$	$\leftarrow RA^{0:1} + RB^{0:1}$
$RT^{2:3}$	$\leftarrow RA^{2:3} + RB^{2:3}$
$RT^{4:5}$	$\leftarrow RA^{4:5} + RB^{4:5}$
$RT^{6:7}$	$\leftarrow RA^{6:7} + RB^{6:7}$
$RT^{8:9}$	$\leftarrow RA^{8:9} + RB^{8:9}$
$RT^{10:11}$	$\leftarrow RA^{10:11} + RB^{10:11}$
$RT^{12:13}$	$\leftarrow RA^{12:13} + RB^{12:13}$
$RT^{14:15}$	$\leftarrow RA^{14:15} + RB^{14:15}$

Add Halfword Immediate 必須 v1.0

ahi **rt,ra,value**



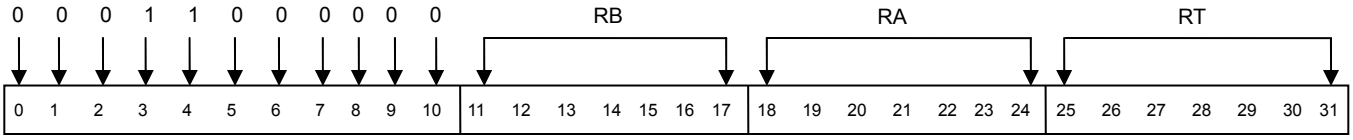
ハーフワード・スロット 8個それぞれに、下記を行なう。

- I10 フィールドの符号付き値を、レジスタ RA の値に加算する。
- 16 bit の結果を RT に置く。
- オーバーフローとキャリーは、検出しない。

s	← RepLeftBit(I10,16)
$RT^{0:1}$	← $RA^{0:1}$ + s
$RT^{2:3}$	← $RA^{2:3}$ + s
$RT^{4:5}$	← $RA^{4:5}$ + s
$RT^{6:7}$	← $RA^{6:7}$ + s
$RT^{8:9}$	← $RA^{8:9}$ + s
$RT^{10:11}$	← $RA^{10:11}$ + s
$RT^{12:13}$	← $RA^{12:13}$ + s
$RT^{14:15}$	← $RA^{14:15}$ + s

Add Word 必須 v1.0

a rt,ra,rb



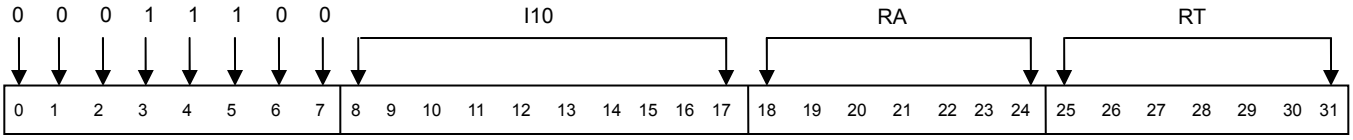
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドに加算する。
- 32 bit の結果をレジスタ RT に置く。
- オーバーフローとキャリーは、検出しない。

$RT^{0:3}$	$\leftarrow RA^{0:3} + RB^{0:3}$
$RT^{4:7}$	$\leftarrow RA^{4:7} + RB^{4:7}$
$RT^{8:11}$	$\leftarrow RA^{8:11} + RB^{8:11}$
$RT^{12:15}$	$\leftarrow RA^{12:15} + RB^{12:15}$

Add Word Immediate 必須 v1.0

ai **rt,ra,value**



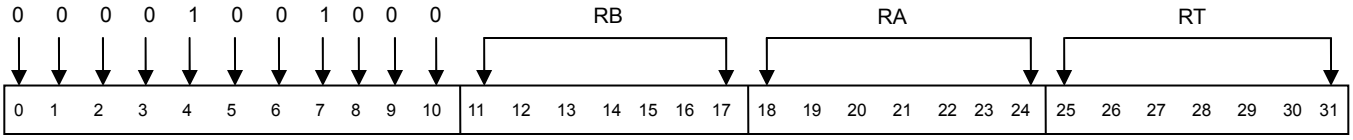
ワード・スロット 4個それぞれに、下記を行なう。

- I10 フィールドの符号付き値を、レジスタ RA のオペランドに加算する。
- 32 bit の結果をレジスタ RT に置く。
- オーバーフローとキャリーは、検出しない。

t	← RepLeftBit(I10,32)
$RT^{0:3}$	← $RA^{0:3}$ + t
$RT^{4:7}$	← $RA^{4:7}$ + t
$RT^{8:11}$	← $RA^{8:11}$ + t
$RT^{12:15}$	← $RA^{12:15}$ + t

Subtract From Halfword 必須 v1.0

sfh **rt,ra,rb**



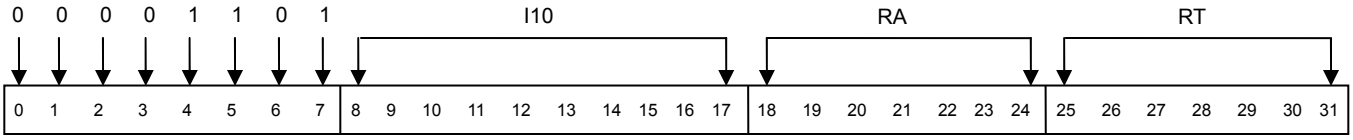
ハーフワード・スロット 8個それぞれに、下記を行なう。

- レジスタ RA の値を RB の値から減算する。
- 16 bit の結果をレジスタ RT に置く。
- オーバーフローとキャリーは、検出しない。

$RT^{0:1}$	$\leftarrow RB^{0:1} + (\neg RA^{0:1}) + 1$
$RT^{2:3}$	$\leftarrow RB^{2:3} + (\neg RA^{2:3}) + 1$
$RT^{4:5}$	$\leftarrow RB^{4:5} + (\neg RA^{4:5}) + 1$
$RT^{6:7}$	$\leftarrow RB^{6:7} + (\neg RA^{6:7}) + 1$
$RT^{8:9}$	$\leftarrow RB^{8:9} + (\neg RA^{8:9}) + 1$
$RT^{10:11}$	$\leftarrow RB^{10:11} + (\neg RA^{10:11}) + 1$
$RT^{12:13}$	$\leftarrow RB^{12:13} + (\neg RA^{12:13}) + 1$
$RT^{14:15}$	$\leftarrow RB^{14:15} + (\neg RA^{14:15}) + 1$

Subtract From Halfword Immediate 必須 v1.0

sfhi **rt,ra,value**



ハーフワード・スロット 8個それぞれに、下記を行なう。

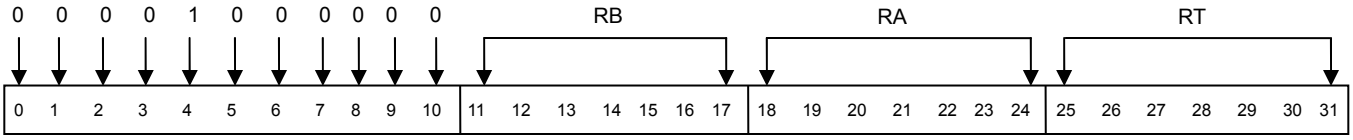
- レジスタ RA の値を、I10 フィールドの符号付き値から減算する。
- 16 bit の結果をレジスタ RT に置く。
- オーバーフローは、検出しない。

プログラミングの注意：Subtract Halfword Immediate 命令はないが、Add Halfword Immediate 命令の即値を負の数にすることで、同一の結果を得ることができる。

t	← RepLeftBit(I10,16)
$RT^{0:1}$	← t + ($\neg RA^{0:1}$) + 1
$RT^{2:3}$	← t + ($\neg RA^{2:3}$) + 1
$RT^{4:5}$	← t + ($\neg RA^{4:5)}$ + 1
$RT^{6:7}$	← t + ($\neg RA^{6:7}$) + 1
$RT^{8:9}$	← t + ($\neg RA^{8:9}$) + 1
$RT^{10:11}$	← t + ($\neg RA^{10:11}$) + 1
$RT^{12:13}$	← t + ($\neg RA^{12:13}$) + 1
$RT^{14:15}$	← t + ($\neg RA^{14:15}$) + 1

Subtract From Word 必須 v1.0

sf **rt,ra,rb**



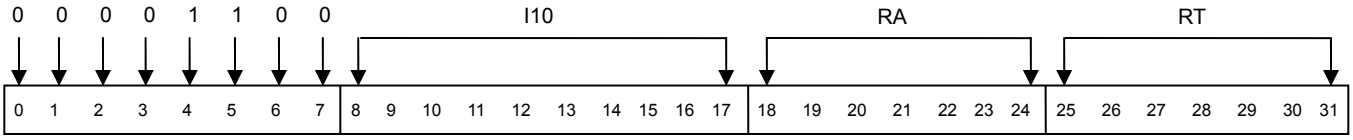
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の値をレジスタ RB の値から減算する。
- 結果をレジスタ RT に置く。
- オーバーフローとキャリーは、検出しない。

$RT^{0:3}$	$\leftarrow RB^{0:3} + (\neg RA^{0:3}) + 1$
$RT^{4:7}$	$\leftarrow RB^{4:7} + (\neg RA^{4:7}) + 1$
$RT^{8:11}$	$\leftarrow RB^{8:11} + (\neg RA^{8:11}) + 1$
$RT^{12:15}$	$\leftarrow RB^{12:15} + (\neg RA^{12:15}) + 1$

Subtract From Word Immediate 必須 v1.0

sfi **rt,ra,value**



ワード・スロット 4個それぞれに、下記を行なう。

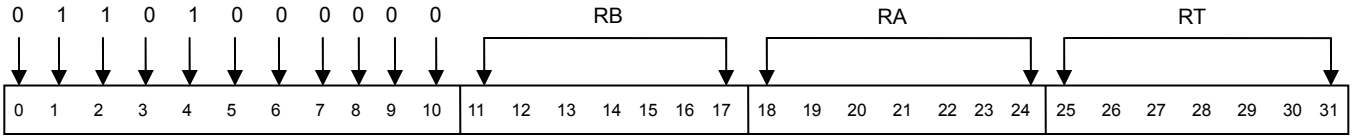
- レジスタ RA の値を、I10 フィールドの値から減算する。
- 結果をレジスタ RT に置く。
- オーバーフローとキャリーは、検出しない。

プログラミングの注意 : Subtract Immediate 命令はないが、Add Immediate 命令の即値を負の数にすることで、同一の結果を得ることができる。

t	← RepLeftBit(I10,32)
$RT^{0:3}$	← t + ($\neg RA^{0:3}$) + 1
$RT^{4:7}$	← t + ($\neg RA^{4:7}$) + 1
$RT^{8:11}$	← t + ($\neg RA^{8:11}$) + 1
$RT^{12:15}$	← t + ($\neg RA^{12:15}$) + 1

Add Extended 必須 v1.0

addx rt,ra,rb



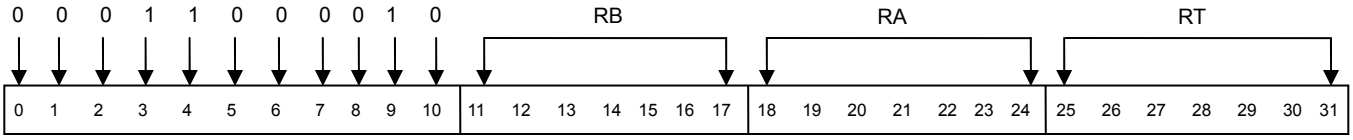
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドおよびレジスタ RT のオペランドの最下位ビットに加算する。
- 32 bit の結果をレジスタ RT に置く。RT 入力のビット 0 から 30 は、予約されているので 0 にすること。

$RT^{0:3}$	$\leftarrow RA^{0:3} + RB^{0:3} + RT_{31}$
$RT^{4:7}$	$\leftarrow RA^{4:7} + RB^{4:7} + RT_{63}$
$RT^{8:11}$	$\leftarrow RA^{8:11} + RB^{8:11} + RT_{95}$
$RT^{12:15}$	$\leftarrow RA^{12:15} + RB^{12:15} + RT_{127}$

Carry Generate 必須 v1.0

cg **rt,ra,rb**



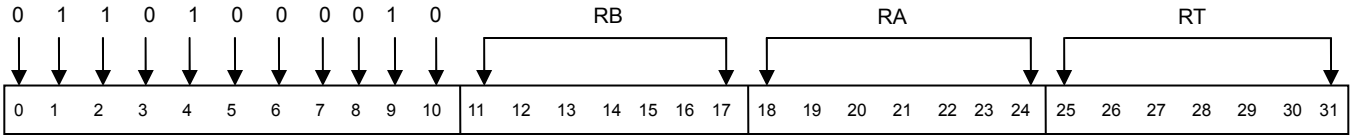
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドに加算する。
- キャリーをレジスタ RT の最下位ビットに置く。
- RT の残りのビットは 0 に設定する。

```
for j = 0 to 15 by 4
        t0:32 = ((0 || RA^(j::4)) + (0 || RB^(j::4)))
        RT^(j::4) ← 31 0 || t0
end
```

Carry Generate Extended 必須 v1.0

cgx rt,ra,rb



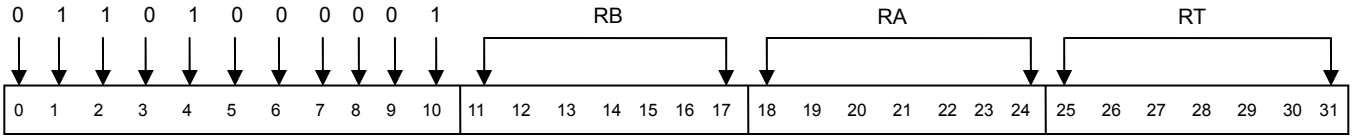
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドおよびレジスタ RT の最下位ビットに加算する。
- キャリーをレジスタ RT の最下位ビットに置く。
- RT の残りのビットは 0 に設定する。RT 入力のビット 0 から 30 は、予約されているので 0 にすること。

```
for j = 0 to 15 by 4
        t0:32 = (0 || RAj::4) + (0 || RBj::4) + (320 || RTj * 8 + 31)
        RTj::4 ← 310 || t0
end
```

Subtract From Extended 必須 v1.0

sfx **rt,ra,rb**



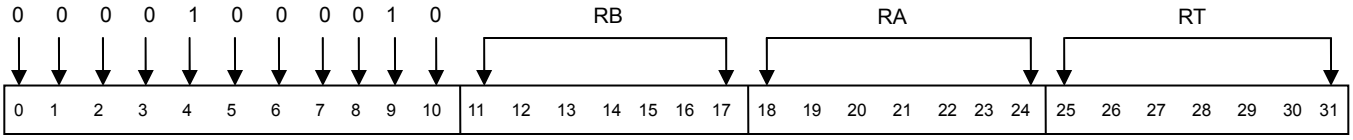
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドから減算する。RT の最下位ビットが ‘0’ である場合は、結果からさらに ‘1’ を減算する。
- 32 bit の結果をレジスタ RT に置く。RT 入力のビット 0 から 30 は、予約されているので 0 にすること。

$RT^{0:3}$	$\leftarrow RB^{0:3} + (\neg RA^{0:3}) + RT_{31}$
$RT^{4:7}$	$\leftarrow RB^{4:7} + (\neg RA^{4:7}) + RT_{63}$
$RT^{8:11}$	$\leftarrow RB^{8:11} + (\neg RA^{8:11}) + RT_{95}$
$RT^{12:15}$	$\leftarrow RB^{12:15} + (\neg RA^{12:15}) + RT_{127}$

Borrow Generate 必須 v1.0

bg rt,ra,rb



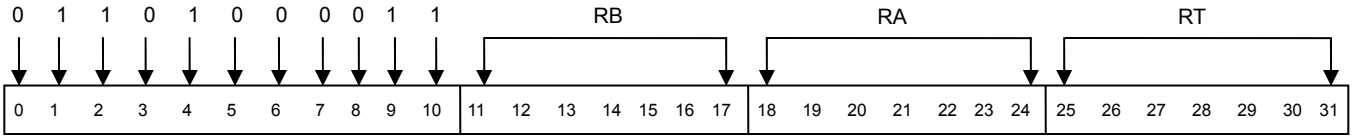
ワード・スロット 4個それぞれに、下記を行なう。

- RA の符号なし値が RB の符号なし値よりも大きい場合は、レジスタ RT に ‘0’ を置く。そうでなければ ‘1’ を置く。

```
for j = 0 to 15 by 4
        if (RB^j::4 ≥^u RA^j::4) then RT^j::4 ← 1
        else                          RT^j::4 ← 0
end
```

Borrow Generate Extended 必須 v1.0

bgx rt,ra,rb



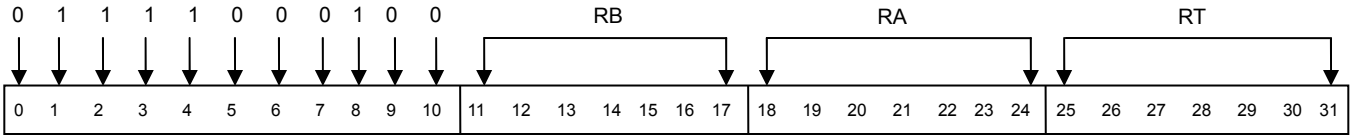
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドから減算する。RT の最下位ビットが ‘0’ の場合は、結果からさらに ‘1’ を減算する。結果が ‘0’ よりも小さい場合は、レジスタ RT に ‘0’ を置く。そうでなければ、レジスタ RT を ‘1’ に設定する。RT 入力のビット 0 から 30 は、予約されているので 0 にすること。

```
for j = 0 to 15 by 4
        if (RT_{j*8+31}) then
                if (RB^{j::4} ≥^u RA^{j::4}) then RT^{j::4} ← 1
                else                            RT^{j::4} ← 0
        else
                if (RB^{j::4} >^u RA^{j::4}) then RT^{j::4} ← 1
                else                            RT^{j::4} ← 0
end
```

Multiply 必須 v1.0

mpy rt,ra,rb



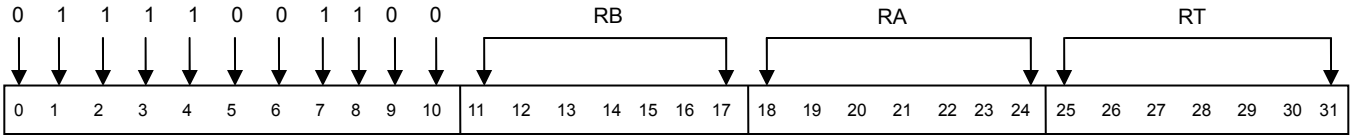
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の右端の 16 bit を、レジスタ RB の右端 16 bit の値に乗算する。
- 32 bit の積をレジスタ RT に置く。
- 各オペランドの左端 16 bit は無視する。

$RT^{0:3}$	$\leftarrow RA^{2:3} * RB^{2:3}$
$RT^{4:7}$	$\leftarrow RA^{6:7} * RB^{6:7}$
$RT^{8:11}$	$\leftarrow RA^{10:11} * RB^{10:11}$
$RT^{12:15}$	$\leftarrow RA^{14:15} * RB^{14:15}$

Multiply Unsigned 必須 v1.0

mpyu **rt,ra,rb**



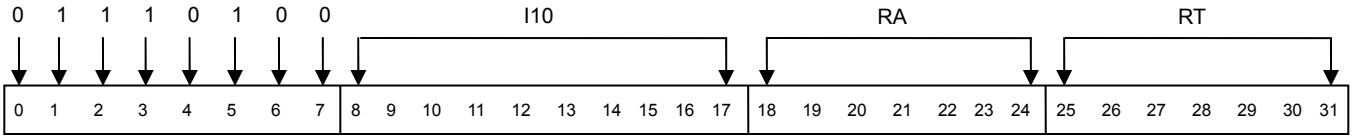
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の右端 16 bit を、レジスタ RB の右端 16 bit で乗算し、その際両方のオペランドを符号なしとして扱う。
- 32 bit の積をレジスタ RT に置く。

$RT^{0:3}$	$\leftarrow RA^{2:3} \mid * \mid RB^{2:3}$
$RT^{4:7}$	$\leftarrow RA^{6:7} \mid * \mid RB^{6:7}$
$RT^{8:11}$	$\leftarrow RA^{10:11} \mid * \mid RB^{10:11}$
$RT^{12:15}$	$\leftarrow RA^{14:15} \mid * \mid RB^{14:15}$

Multiply Immediate 必須 v1.0

mpyi **rt,ra,value**



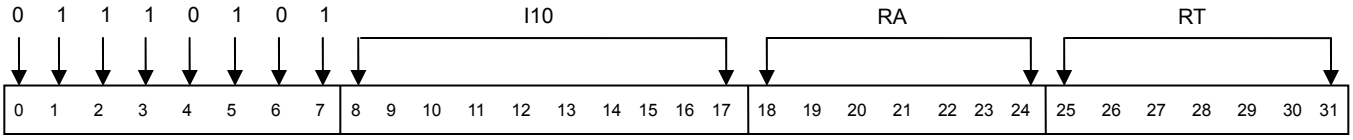
ワード・スロット 4個それぞれに、下記を行なう。

- I10 フィールドの符号付き値を、レジスタ RA の右端 16 bit の値で乗算する。
- 結果の積をレジスタ RT に置く。

t	← RepLeftBit(I10,16)
$RT^{0:3}$	← $RA^{2:3}$ * t
$RT^{4:7}$	← $RA^{6:7}$ * t
$RT^{8:11}$	← $RA^{10:11}$ * t
$RT^{12:15}$	← $RA^{14:15}$ * t

Multiply Unsigned Immediate 必須 v1.0

mpyui rt,ra,value



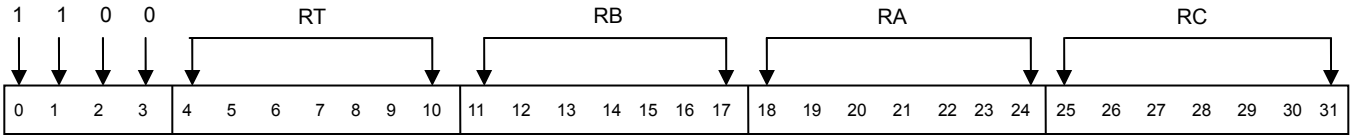
ワード・スロット 4個それぞれに、下記を行なう。

- I10 フィールドの符号付き値を、左端のビットを複製することによって、16 bit に拡張する。結果の値をレジスタ RA の右端 16 bit で乗算し、その際両方のオペランドを符号なしとして扱う。
- 結果の積をレジスタ RT に置く。

t	← RepLeftBit(I10,16)
$RT^{0:3}$	← $RA^{2:3}$ |*| t
$RT^{4:7}$	← $RA^{6:7}$ |*| t
$RT^{8:11}$	← $RA^{10:11}$ |*| t
$RT^{12:15}$	← $RA^{14:15}$ |*| t

Multiply and Add 必須 v1.0

mpya **rt,ra,rb,rc**



ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の値を 16 bit の符号付き整数として扱い、レジスタ RB の 16 bit の符号付き値で乗算する。結果の積をレジスタ RC の値に加算する。
- 結果をレジスタ RT に置く。
- オーバーフローとキャリーは、検出しない。

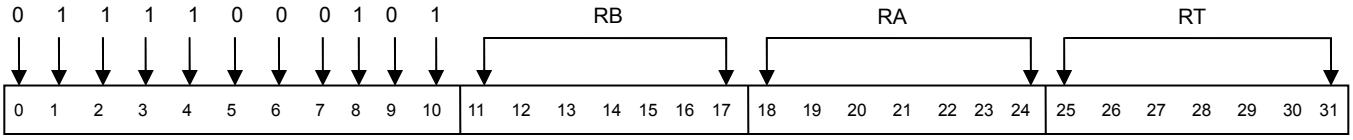
プログラミングの注意：オペランドは、32 bit のフィールド内で右揃えされる。

t0	$\leftarrow RA^{2:3} * RB^{2:3}$
t1	$\leftarrow RA^{6:7} * RB^{6:7}$
t2	$\leftarrow RA^{10:11} * RB^{10:11}$
t3	$\leftarrow RA^{14:15} * RB^{14:15}$
$RT^{0:3}$	$\leftarrow t0 + RC^{0:3}$
$RT^{4:7}$	$\leftarrow t1 + RC^{4:7}$
$RT^{8:11}$	$\leftarrow t2 + RC^{8:11}$
$RT^{12:15}$	$\leftarrow t3 + RC^{12:15}$

Multiply High

必須　v1.0

mpyh　　**rt,ra,rb**



ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA にある値の左端 16 bit を 16 bit 右にシフトし、レジスタ RB の 16 bit の値で乗算する。
- 積は 16 bit 左にシフトし、レジスタ RT に置く。左にシフトアウトされたビットは、破棄する。右端から 0 がシフトインされる。

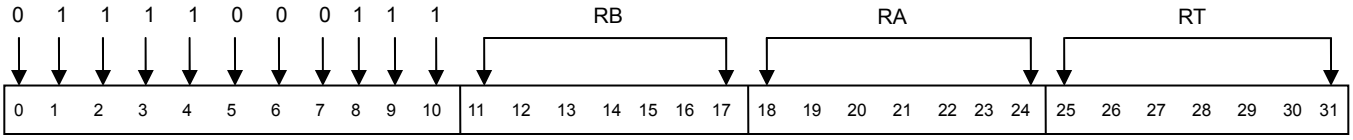
t0	← $RA^{0:1} * RB^{2:3}$
t1	← $RA^{4:5} * RB^{6:7}$
t2	← $RA^{8:9} * RB^{10:11}$
t3	← $RA^{12:13} * RB^{14:15}$
$RT^{0:3}$	← $t0^{2:3}$ || 0x0000
$RT^{4:7}$	← $t1^{2:3}$ || 0x0000
$RT^{8:11}$	← $t2^{2:3}$ || 0x0000
$RT^{12:15}$	← $t3^{2:3}$ || 0x0000

プログラミングの注意：この命令を **mpyu** および Add Word（**a**）と併せて使用すれば、32 bit 乗算を実現できる。32 bit 乗算命令 **mpy32** rt,ra,rb は、以下の命令列によりエミュレートすることができる。

```
mpyh    t1,ra,rb
mpyh    t2,rb,ra
mpyu    t3,ra,rb
a       rt,t1,t2
a       rt,rt,t3
```

Multiply and Shift Right 必須 v1.0

mpys **rt,ra,rb**



ワード・スロット 4個それぞれに、下記を行なう。

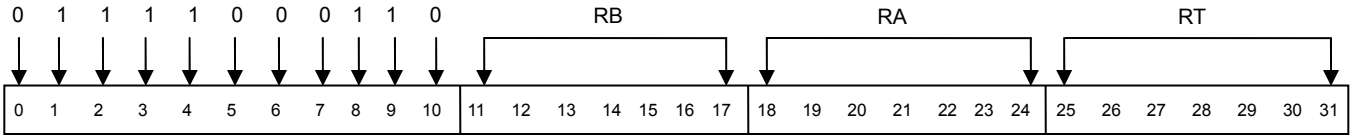
- レジスタ RA の右端 16 bit の値を、レジスタ RB の右端 16 bit の値で乗算する。
- 32 bit の積の左端 16 bit をレジスタ RT の右端 16 bit に置き、符号ビットをレジスタの左 16 bit に複製する。

t0	← $RA^{2:3} * RB^{2:3}$
t1	← $RA^{6:7} * RB^{6:7}$
t2	← $RA^{10:11} * RB^{10:11}$
t3	← $RA^{14:15} * RB^{14:15}$
$RT^{0:3}$	← RepLeftBit($t0^{0:1}$,32)
$RT^{4:7}$	← RepLeftBit($t1^{0:1}$,32)
$RT^{8:11}$	← RepLeftBit($t2^{0:1}$,32)
$RT^{12:15}$	← RepLeftBit($t3^{0:1}$,32)

Multiply High High

必須　v1.0

mpyhh　　rt,ra,rb



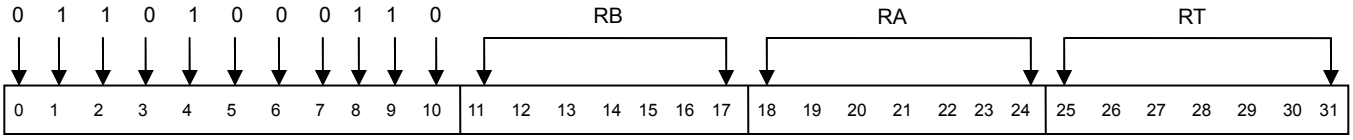
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の左端 16 bit を、レジスタ RB の左端 16 bit で乗算する。
- 32 bit の積をレジスタ RT に置く。

$RT^{0:3}$	$\leftarrow RA^{0:1} * RB^{0:1}$
$RT^{4:7}$	$\leftarrow RA^{4:5} * RB^{4:5}$
$RT^{8:11}$	$\leftarrow RA^{8:9} * RB^{8:9}$
$RT^{12:15}$	$\leftarrow RA^{12:13} * RB^{12:13}$

Multiply High High and Add 必須 v1.0

mpyhha rt,ra,rb



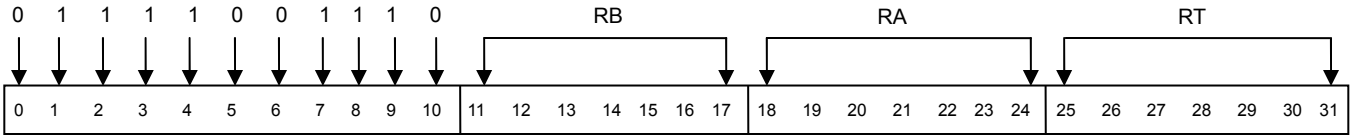
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の左端 16 bit を、レジスタ RB の左端 16 bit で乗算する。積をレジスタ RT の値に加算する。
- その合計をレジスタ RT に置く。

$RT^{0:3}$	$\leftarrow RA^{0:1} * RB^{0:1} + RT^{0:3}$
$RT^{4:7}$	$\leftarrow RA^{4:5} * RB^{4:5} + RT^{4:7}$
$RT^{8:11}$	$\leftarrow RA^{8:9} * RB^{8:9} + RT^{8:11}$
$RT^{12:15}$	$\leftarrow RA^{12:13} * RB^{12:13} + RT^{12:15}$

Multiply High High Unsigned 必須 v1.0

mpyhhu rt,ra,rb



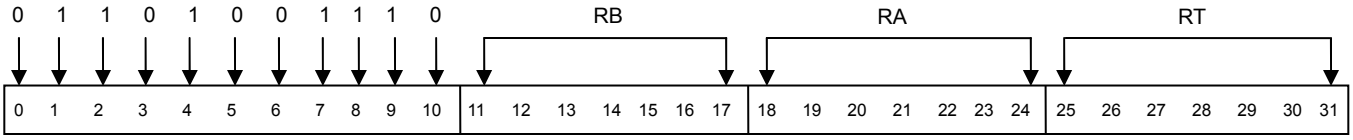
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の左端 16 bit を、レジスタ RB の左端 16 bit で乗算し、その際両方のオペランドを符号なしとして扱う。
- 32 bit の積をレジスタ RT に置く。

$RT^{0:3}$	$\leftarrow RA^{0:1} \mid * \mid RB^{0:1}$
$RT^{4:7}$	$\leftarrow RA^{4:5} \mid * \mid RB^{4:5}$
$RT^{8:11}$	$\leftarrow RA^{8:9} \mid * \mid RB^{8:9}$
$RT^{12:15}$	$\leftarrow RA^{12:13} \mid * \mid RB^{12:13}$

Multiply High High Unsigned and Add 必須 v1.0

mpyhhau rt,ra,rb



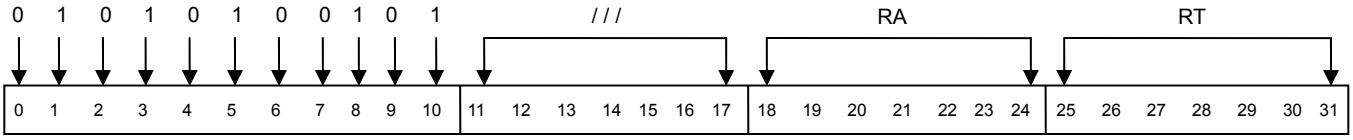
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の左端 16 bit を、レジスタ RB の左端 16 bit で乗算し、その際両方のオペランドを符号なしとして扱う。積をレジスタ RT の値に加算する。
- その合計をレジスタ RT に置く。

$RT^{0:3}$	$\leftarrow RA^{0:1} \mid * \mid RB^{0:1} + RT^{0:3}$
$RT^{4:7}$	$\leftarrow RA^{4:5} \mid * \mid RB^{4:5} + RT^{4:7}$
$RT^{8:11}$	$\leftarrow RA^{8:9} \mid * \mid RB^{8:9} + RT^{8:11}$
$RT^{12:15}$	$\leftarrow RA^{12:13} \mid * \mid RB^{12:13} + RT^{12:15}$

Count Leading Zeros 必須 v1.0

clz **rt,ra**



ワード・スロット 4個それぞれに、下記を行なう。

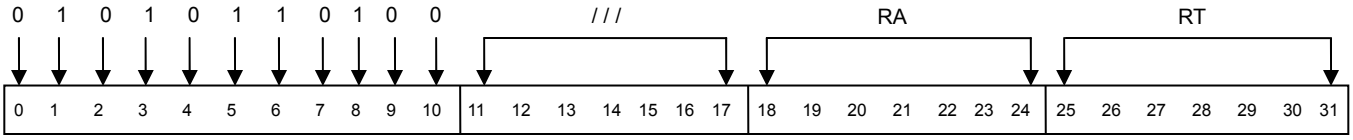
- レジスタ RA のオペランドの最初の ‘1’ のビットの左にある 0 のビット数を算出する。
- 結果をレジスタ RT に置く。レジスタ RA が 0 の場合、結果は 32 である。

プログラミングの注意：レジスタ RT に置く結果は、$0 \le RT \le 32$ を満たす。例えば、RA の対応するスロットが負の整数である場合、レジスタ RT の値は 0 である。レジスタ RA の対応するスロットが 0 である場合、レジスタ RT の値は 32 である。

```
for j = 0 to 15 by 4
        t ← 0
        u ← RA^(j::4)
        For m = 0 to 31
                If u_m = 1 then leave
                t ← t + 1
        end
        RT^(j::4) ← t
end
```

Count Ones in Bytes 必須 v1.0

cntb **rt,ra**



バイト・スロット 16個それぞれに、下記を行なう。

- レジスタ RA の値が ‘1’ であるビットの数を算出する。
- 結果をレジスタ RT に置く。

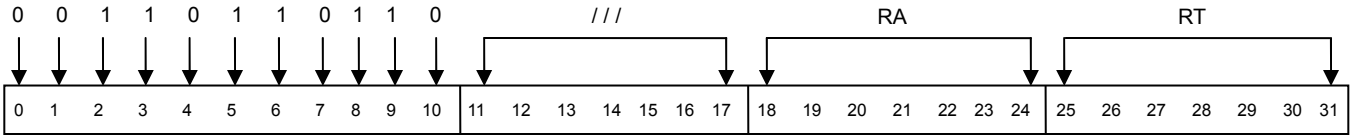
プログラミングの注意：レジスタ RT に格納される結果は、$0 \le RT \le 8$ を満たす。例えば、レジスタ RA の値が 0 である場合、レジスタ RT の値は 0 である。RA の値が -1 である場合、RT の値は 8 である。

```
for j = 0 to 15
        c = 0
        b ← RA^j
        For m = 0 to 7
                If b_m = 1 then c ← c + 1
        end
        RT^j ← c
end
```

（85ページの Form Select Mask for Bytes 命令も参照のこと。）

Form Select Mask for Bytes 必須 v1.0

fsmb rt,ra

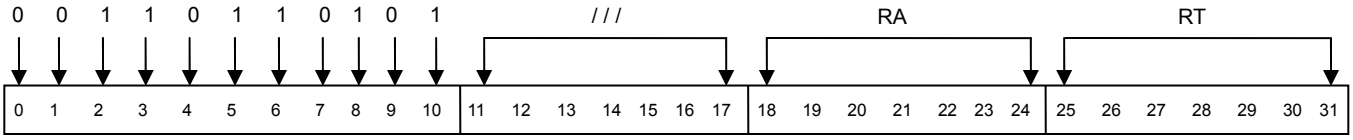


レジスタ RA のプリファード・スロット中の右端 16 bit の各ビットを 8 回複製することによって、レジスタ RT にマスクを生成する。オペランドのビットは、左から右へ、結果のバイトと対応する。

```
s ← RA[2:3]
For j = 0 to 15
        If s_j = 0 then r^j ← 0x00
        else            r^j ← 0xFF
End
RT ← r
```

Form Select Mask for Halfwords 必須 v1.0

fsmh rt,ra

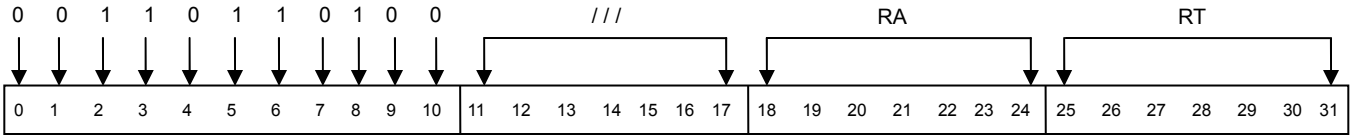


レジスタ RA のプリファード・スロット中の右端 8 bit の各ビットを 16 回複製することによって、レジスタ RT にマスクを生成する。オペランドのビットは、左から右に、結果のハーフワードと対応する。

```
s ← RA^3
k = 0
for j = 0 to 7
        If s_j = 0 then r^k::2 ← 0x0000
        else            r^k::2 ← 0xFFFF
        k = k + 2
end
RT ← r
```

Form Select Mask for Words 必須 v1.0

fsm rt,ra

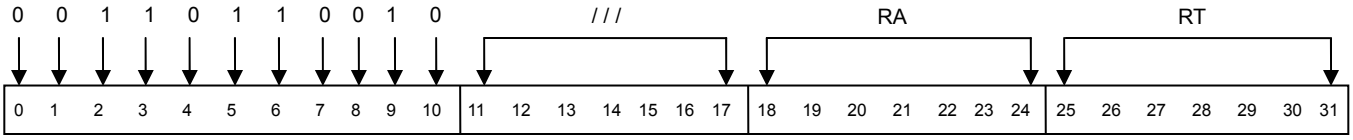


レジスタ RA のプリファード・スロット中の右端 4 bit の各ビットを 32 回複製することによって、レジスタ RT にマスクを生成する。オペランドのビットは、左から右へ、結果のワードに対応する。

```
s ← RA28:31
k = 0
for j = 0 to 3
        If sj = 0 then rk::4 ← 0x00000000
        else           rk::4 ← 0xFFFFFFFF
        k = k + 4
end
RT ← r
```

Gather Bits from Bytes 必須 v1.0

gbb **rt,ra**

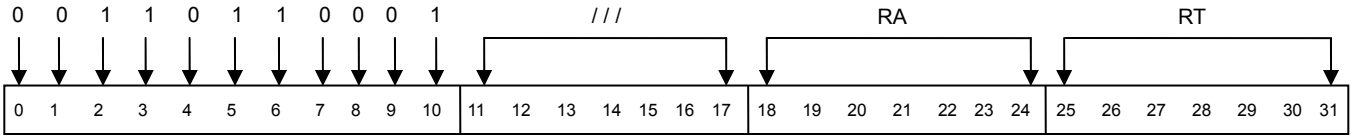


レジスタ RA の各バイトの右端ビットを連結することによって、レジスタ RT のプリファード・スロットの右半分に、16 bit の値を形成する。レジスタ RT の左端 16 bit を、残りのスロットと同様に 0 に設定する。

```
k = 0
s = 0
for j = 7 to 128 by 8
        sk ← RAj
        k = k + 1
end
RT0:3 ← 0x0000 || s
RT4:7 ← 0
RT8:11 ← 0
RT12:15 ← 0
```

Gather Bits from Halfwords 必須 v1.0

gbh **rt,ra**

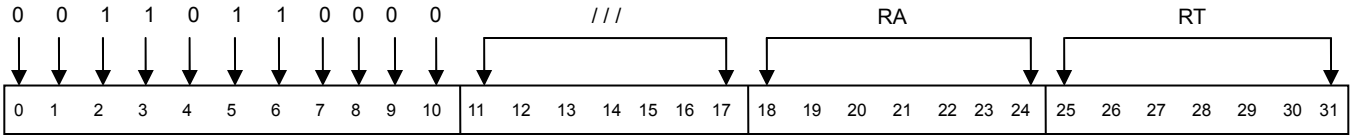


レジスタ RA の各ハーフワードの右端ビットを連結することによって、レジスタ RT のプリファード・スロットの右端のバイトに 8 bit の値を形成する。レジスタ RT のプリファード・スロットの左端 24 bit を、残りのスロットと同様に 0 に設定する。

```
k = 0
s = 0x00
for j = 15 to 128 by 16
        s_k ← RA_j
        k = k + 1
end
RT^0:3 ← 0x000000 || s
RT^4:7 ← 0
RT^8:11 ← 0
RT^12:15 ← 0
```

Gather Bits from Words 必須 v1.0

gb rt,ra

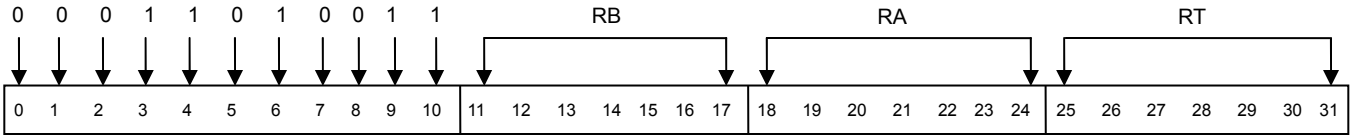


レジスタ RA の各ワードの右端ビットを連結することによって、レジスタ RT の右端 4 bit に、4 bit の値を形成する。レジスタ RT の左端 28 bit を、残りのスロットと同様に 0 に設定する。

```
k = 0
s = 0x0
for j = 31 to 128 by 32
        sk ← RAj
        k ← k + 1
end
RT0:3 ← 0x0000000 || s
RT4:7 ← 0
RT8:11 ← 0
RT12:15 ← 0
```

Average Bytes 必須 v1.0

avgb **rt,ra,rb**



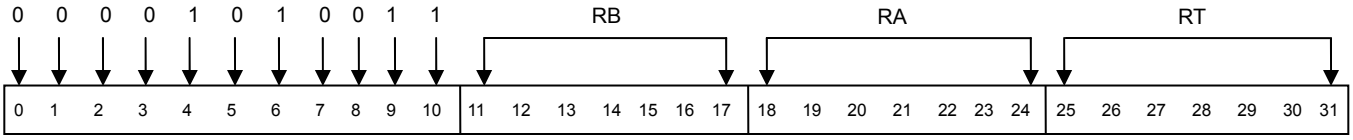
バイト・スロット 16個それぞれに、下記を行なう。

- レジスタ RA のオペランドをレジスタ RB のオペランドに加算し、その結果に ‘1’ を加算する。これらの加算は、精度を落とすことなく実行される。
- 結果を 1 bit 右にシフトし、レジスタ RT に置く。

```
for j = 0 to 15
        RT^j ← ((0x00 || RA^j) + (0x00 || RB^j) + 1)_7:14
end
```

Absolute Differences of Bytes 必須 v1.0

absdb **rt,ra,rb**



バイト・スロット 16個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドから減算する。
- 結果の絶対値をレジスタ RT に置く。

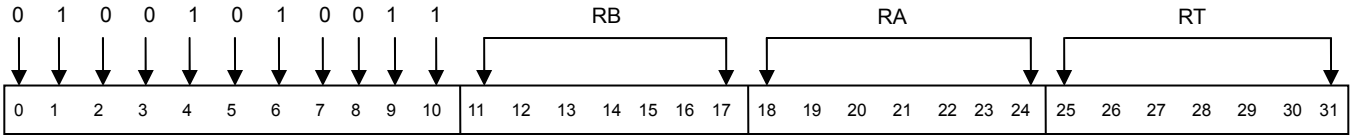
プログラミングの注意：オペランドは符号なしである。

```
for j = 0 to 15
        if (RB^j >^u RA^j) then RT^j ← RB^j - RA^j
        else                    RT^j ← RA^j - RB^j
end
```

Sum Bytes into Halfwords

必須　v1.0

sumb　　**rt,ra,rb**



ワード・スロット 4個それぞれに、下記を行なう。

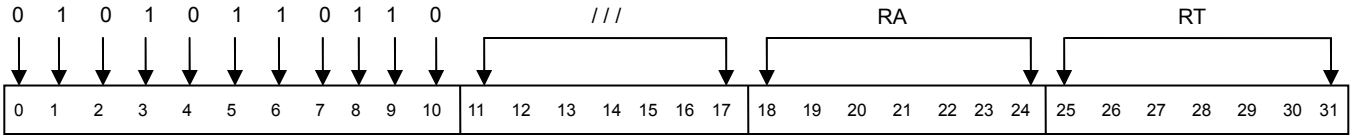
- レジスタ RB の 4 byte を加算し、16 bit の結果をレジスタ RT のバイト 0 および 1 に置く。
- レジスタ RA の 4 byte を加算し、16 bit の結果をレジスタ RT のバイト 2 および 3 に置く。

プログラミングの注意：オペランドは符号なしである。

$RT^{0:1}$	$\leftarrow RB^0 + RB^1 + RB^2 + RB^3$
$RT^{2:3}$	$\leftarrow RA^0 + RA^1 + RA^2 + RA^3$
$RT^{4:5}$	$\leftarrow RB^4 + RB^5 + RB^6 + RB^7$
$RT^{6:7}$	$\leftarrow RA^4 + RA^5 + RA^6 + RA^7$
$RT^{8:9}$	$\leftarrow RB^8 + RB^9 + RB^{10} + RB^{11}$
$RT^{10:11}$	$\leftarrow RA^8 + RA^9 + RA^{10} + RA^{11}$
$RT^{12:13}$	$\leftarrow RB^{12} + RB^{13} + RB^{14} + RB^{15}$
$RT^{14:15}$	$\leftarrow RA^{12} + RA^{13} + RA^{14} + RA^{15}$

Extend Sign Byte to Halfword 必須 v1.0

xsbh **rt,ra**



ハーフワード・スロット 8個それぞれに、下記を行なう。

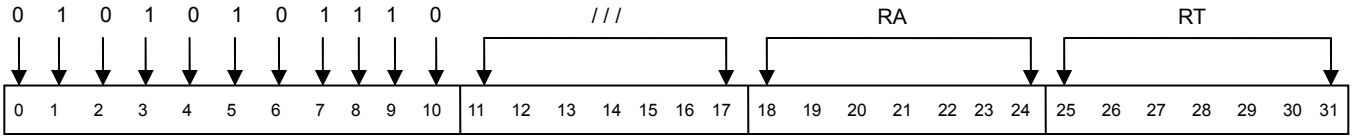
- レジスタ RA にあるオペランドの右のバイトのバイト符号を、左のバイトに拡張する。
- 結果の 16 bit 整数を、レジスタ RT に格納する。

プログラミングの注意：この命令は、バイトを符号付きとして処理する唯一の命令である。

$RT^{0:1}$	← RepLeftBit(RA^{1},16)
$RT^{2:3}$	← RepLeftBit(RA^{3},16)
$RT^{4:5}$	← RepLeftBit(RA^{5},16)
$RT^{6:7}$	← RepLeftBit(RA^{7},16)
$RT^{8:9}$	← RepLeftBit(RA^{9},16)
$RT^{10:11}$	← RepLeftBit(RA^{11},16)
$RT^{12:13}$	← RepLeftBit(RA^{13},16)
$RT^{14:15}$	← RepLeftBit(RA^{15},16)

Extend Sign Halfword to Word 必須 v1.0

xshw rt,ra



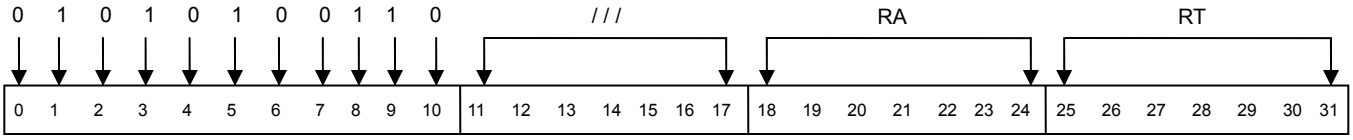
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA にあるオペランドの右半分のハーフワードの符号を、左のハーフワードに拡張する。
- 結果の 32 bit 整数を、レジスタ RT に置く。

$RT^{0:3}$	← RepLeftBit($RA^{2:3}$,32)
$RT^{4:7}$	← RepLeftBit($RA^{6:7}$,32)
$RT^{8:11}$	← RepLeftBit($RA^{10:11}$,32)
$RT^{12:15}$	← RepLeftBit($RA^{14:15}$,32)

Extend Sign Word to Doubleword 必須 v1.0

xswd rt,ra



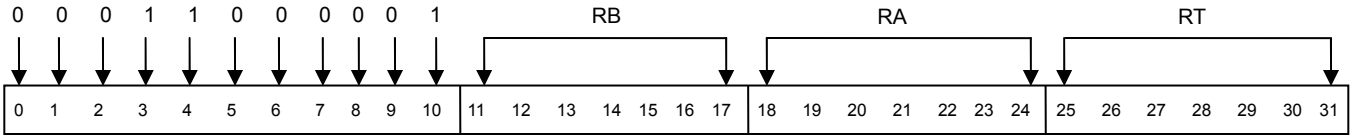
ダブルワード・スロット 2個それぞれに、下記を行なう。

- 右のスロットのワードの符号を、左のワードに拡張する。
- 結果の 64 bit 整数を、レジスタ RT に格納する。

$RT^{0:7}$	← RepLeftBit($RA^{4:7}$,64)
$RT^{8:15}$	← RepLeftBit($RA^{12:15}$,64)

And 必須 v1.0

and **rt,ra,rb**

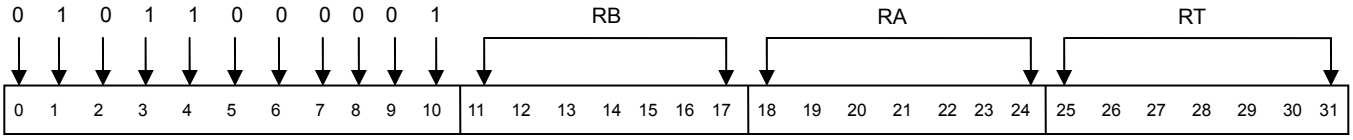


レジスタ RA およびレジスタ RB の値を、論理 AND 演算する。結果をレジスタ RT に置く。

RT0:3	← RA0:3 & RB0:3
RT4:7	← RA4:7 & RB4:7
RT8:11	← RA8:11 & RB8:11
RT12:15	← RA12:15 & RB12:15

And with Complement 必須 v1.0

andc rt,ra,rb

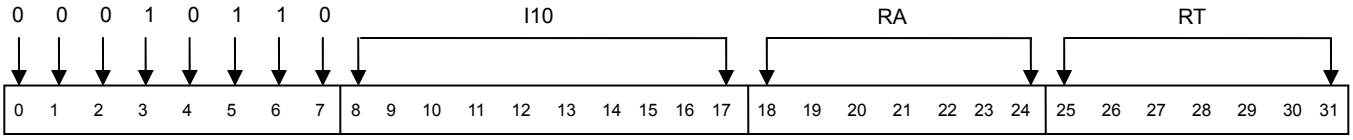


レジスタ RA の値を、レジスタ RB の値の補数と論理 AND 演算する。結果をレジスタ RT に置く。

RT0:3	← RA0:3 & (¬RB0:3)
RT4:7	← RA4:7 & (¬RB4:7)
RT8:11	← RA8:11 & (¬RB8:11)
RT12:15	← RA12:15 & (¬RB12:15)

And Byte Immediate 必須 v1.0

andbi **rt,ra,value**



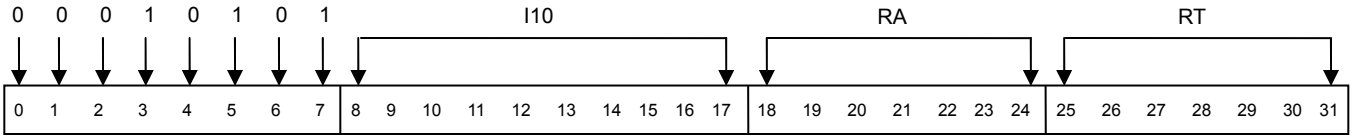
バイト・スロット 16個それぞれに、下記を行なう。

- I10 フィールドの右端 8 bit を、レジスタ RA の値と AND 演算する。
- 結果をレジスタ RT に置く。

b	← I10 & 0x00FF
bbbb	← b || b || b || b
RT0:3	← RA0:3 & bbbb
RT4:7	← RA4:7 & bbbb
RT8:11	← RA8:11 & bbbb
RT12:15	← RA12:15 & bbbb

And Halfword Immediate 必須 v1.0

andhi rt,ra,value



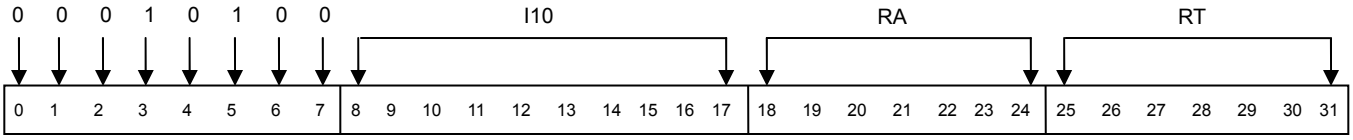
ハーフワード・スロット 8個それぞれに、下記を行なう。

- I10 フィールドを、左端のビットを複製することによって、16 bit に拡張する。結果をレジスタ RA の値と AND 演算する。
- 16 bit の結果をレジスタ RT に置く。

t	← RepLeftBit(I10,16)
RT0:1	← RA0:1 & t
RT2:3	← RA2:3 & t
RT4:5	← RA4:5 & t
RT6:7	← RA6:7 & t
RT8:9	← RA8:9 & t
RT10:11	← RA10:11 & t
RT12:13	← RA12:13 & t
RT14:15	← RA14:15 & t

And Word Immediate 必須 v1.0

andi rt,ra,value



ワード・スロット 4個それぞれに、下記を行なう。

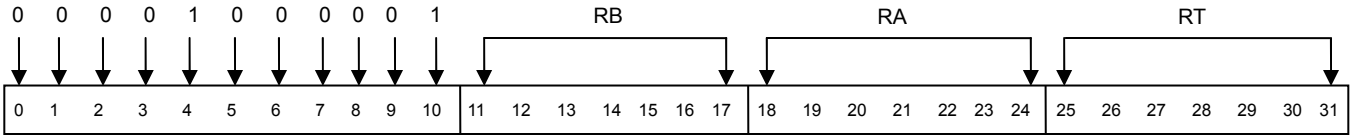
- I10 フィールドの値を、左端のビットを複製することによって、32 bit に拡張する。結果を RA の値と論理 AND 演算する。
- 結果をレジスタ RT に置く。

t	← RepLeftBit(I10,32)
RT0:3	← RA0:3 & t
RT4:7	← RA4:7 & t
RT8:11	← RA8:11 & t
RT12:15	← RA12:15 & t

Or

必須　v1.0

or　　rt,ra,rb



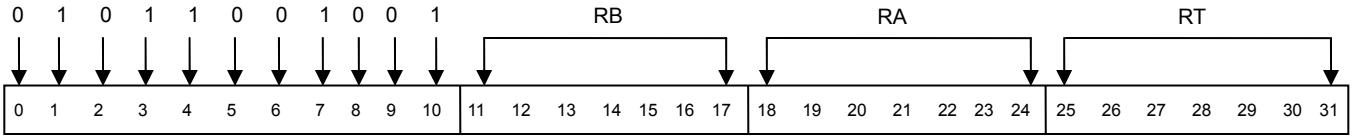
レジスタ RA の値とレジスタ RB の値で論理 OR 演算する。結果をレジスタ RT に置く。

$RT^{0:3}$	$\leftarrow RA^{0:3} \mid RB^{0:3}$
$RT^{4:7}$	$\leftarrow RA^{4:7} \mid RB^{4:7}$
$RT^{8:11}$	$\leftarrow RA^{8:11} \mid RB^{8:11}$
$RT^{12:15}$	$\leftarrow RA^{12:15} \mid RB^{12:15}$

Or with Complement

必須　v1.0

orc　　**rt,ra,rb**

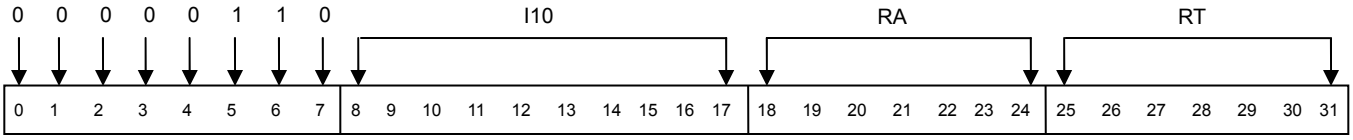


レジスタ RA の値と、レジスタ RB の値の補数で OR 演算する。結果をレジスタ RT に置く。

$RT^{0:3}$	$\leftarrow RA^{0:3} \mid (\neg RB^{0:3})$
$RT^{4:7}$	$\leftarrow RA^{4:7} \mid (\neg RB^{4:7})$
$RT^{8:11}$	$\leftarrow RA^{8:11} \mid (\neg RB^{8:11})$
$RT^{12:15}$	$\leftarrow RA^{12:15} \mid (\neg RB^{12:15})$

Or Byte Immediate 必須 v1.0

orbi **rt,ra,value**



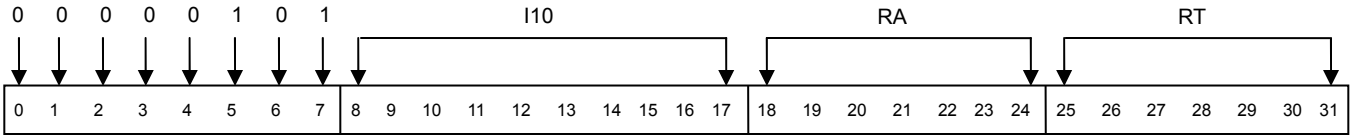
バイト・スロット 16個それぞれに、下記を行なう。

- I10 フィールドの右端 8 bit と、レジスタ RA の値で OR 演算する。
- 結果をレジスタ RT に置く。

b	← I10 & 0x00FF
bbbb	← b || b || b || b
$RT^{0:3}$	← $RA^{0:3}$ | bbbb
$RT^{4:7}$	← $RA^{4:7}$ | bbbb
$RT^{8:11}$	← $RA^{8:11}$ | bbbb
$RT^{12:15}$	← $RA^{12:15}$ | bbbb

Or Halfword Immediate 必須 v1.0

orhi **rt,ra,value**



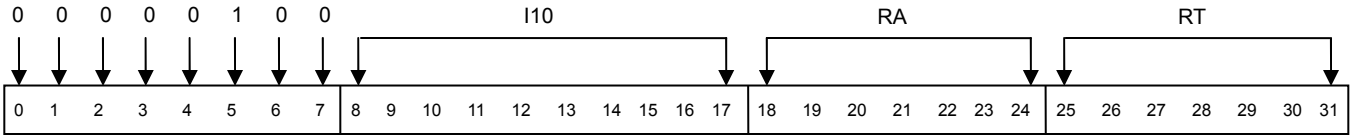
ハーフワード・スロット 8個それぞれに、下記を行なう。

- I10 フィールドの値を、左端ビットを複製することによって、16 bit に拡張する。結果とレジスタ RA の値とで OR 演算する。
- 結果をレジスタ RT に置く。

t	← RepLeftBit(I10,16)
$RT^{0:1}$	← $RA^{0:1}$ | t
$RT^{2:3}$	← $RA^{2:3}$ | t
$RT^{4:5}$	← $RA^{4:5}$ | t
$RT^{6:7}$	← $RA^{6:7}$ | t
$RT^{8:9}$	← $RA^{8:9}$ | t
$RT^{10:11}$	← $RA^{10:11}$ | t
$RT^{12:13}$	← $RA^{12:13}$ | t
$RT^{14:15}$	← $RA^{14:15}$ | t

Or Word Immediate 必須 v1.0

ori **rt,ra,value**



ワード・スロット 4個それぞれに、下記を行なう。

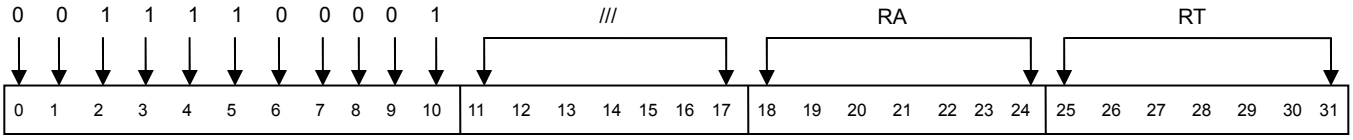
- I10 フィールドを 32 bit に符号拡張し、レジスタ RA の値と OR 演算する。
- 結果をレジスタ RT に置く。

t	← RepLeftBit(I10,32)
RT0:3	← RA0:3 | t
RT4:7	← RA4:7 | t
RT8:11	← RA8:11 | t
RT12:15	← RA12:15 | t

Or Across

必須　v1.0

orx　　**rt,ra**



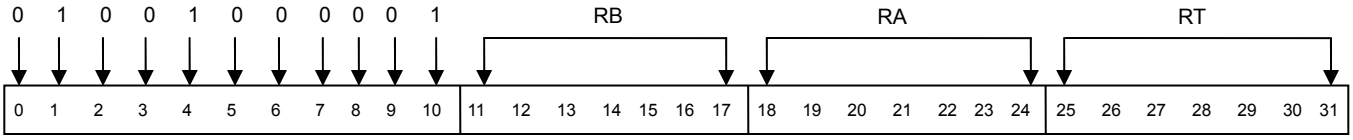
RA の 4 ワードを論理 OR 演算する。結果をレジスタ RT のプリファード・スロットに置く。レジスタの残り 3 つのスロットには、0 が書かれる。

$RT^{0:3}$	$\leftarrow RA^{0:3} \mid RA^{4:7} \mid RA^{8:11} \mid RA^{12:15}$
$RT^{4:15}$	$\leftarrow 0$

Exclusive Or

必須　v1.0

xor　　rt,ra,rb

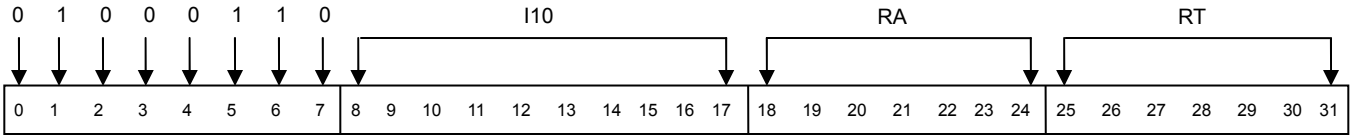


レジスタ RA とレジスタ RB の値で、XOR 演算する。結果をレジスタ RT に置く。

$RT^{0:3}$	$\leftarrow RA^{0:3} \oplus RB^{0:3}$
$RT^{4:7}$	$\leftarrow RA^{4:7} \oplus RB^{4:7}$
$RT^{8:11}$	$\leftarrow RA^{8:11} \oplus RB^{8:11}$
$RT^{12:15}$	$\leftarrow RA^{12:15} \oplus RB^{12:15}$

Exclusive Or Byte Immediate 必須 v1.0

xorbi **rt,ra,value**



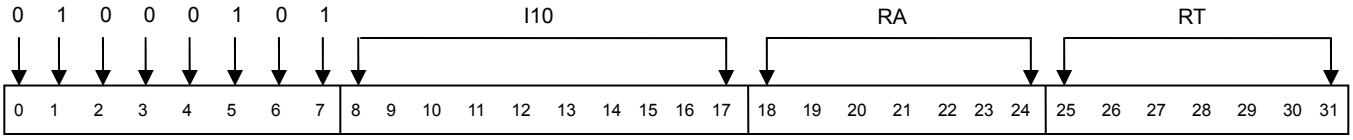
バイト・スロット 16個それぞれに、下記を行なう。

- I10 フィールドの右端 8 bit を、レジスタ RA の値と XOR 演算する。
- 結果をレジスタ RT に置く。

b	← I10 & 0x00FF
bbbb	← b || b || b || b
RT0:3	← RA0:3 ⊕ bbbb
RT4:7	← RA4:7 ⊕ bbbb
RT8:11	← RA8:11 ⊕ bbbb
RT12:15	← RA12:15 ⊕ bbbb

Exclusive Or Halfword Immediate 必須 v1.0

xorhi rt,ra,value



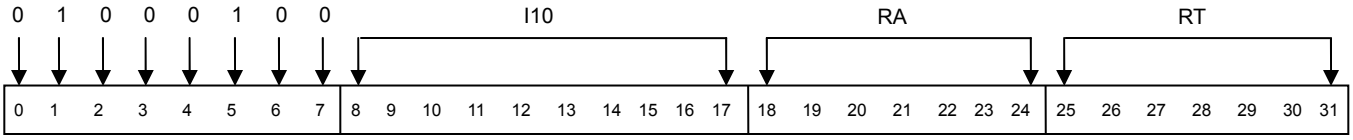
ハーフワード・スロット 8個それぞれに、下記を行なう。

- I10 フィールドの値を、左端のビットを複製することによって、16 bit に拡張する。結果をレジスタ RA の値と XOR 演算する。
- 16 bit の結果をレジスタ RT に置く。

t	← RepLeftBit(I10,16)
$RT^{0:1}$	← $RA^{0:1}$ ⊕ t
$RT^{2:3}$	← $RA^{2:3}$ ⊕ t
$RT^{4:5}$	← $RA^{4:5}$ ⊕ t
$RT^{6:7}$	← $RA^{6:7}$ ⊕ t
$RT^{8:9}$	← $RA^{8:9}$ ⊕ t
$RT^{10:11}$	← $RA^{10:11}$ ⊕ t
$RT^{12:13}$	← $RA^{12:13}$ ⊕ t
$RT^{14:15}$	← $RA^{14:15}$ ⊕ t

Exclusive Or Word Immediate 必須 v1.0

xori **rt,ra,value**



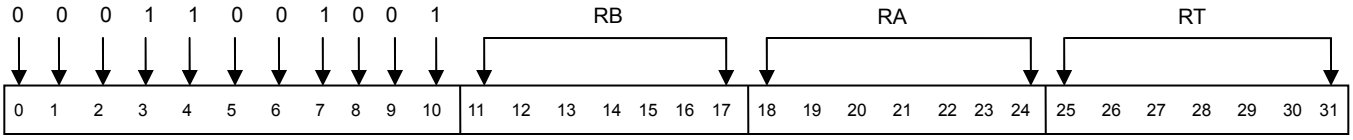
ワード・スロット 4個それぞれに、下記を行なう。

- I10 フィールドを 32 bit に符号拡張し、レジスタ RA の値と XOR 演算する。
- 32 bit の結果をレジスタ RT に置く。

t	← RepLeftBit(I10,32)
$RT^{0:3}$	← $RA^{0:3}$ ⊕ t
$RT^{4:7}$	← $RA^{4:7}$ ⊕ t
$RT^{8:11}$	← $RA^{8:11}$ ⊕ t
$RT^{12:15}$	← $RA^{12:15}$ ⊕ t

Nand 必須 v1.0

nand **rt,ra,rb**



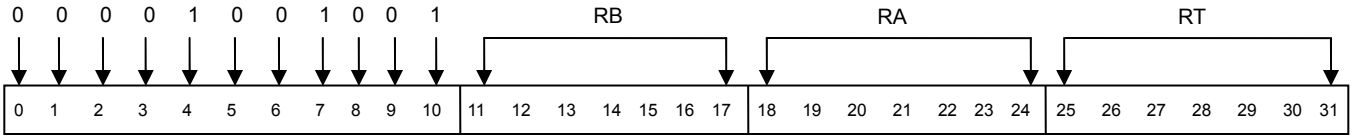
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のビットとレジスタ RB のビットで AND 演算した結果の補数を、レジスタ RT に置く。

$RT^{0:3}$	$\leftarrow \neg(RA^{0:3} \& RB^{0:3})$
$RT^{4:7}$	$\leftarrow \neg(RA^{4:7} \& RB^{4:7})$
$RT^{8:11}$	$\leftarrow \neg(RA^{8:11} \& RB^{8:11})$
$RT^{12:15}$	$\leftarrow \neg(RA^{12:15} \& RB^{12:15})$

Nor 必須 v1.0

nor rt,ra,rb



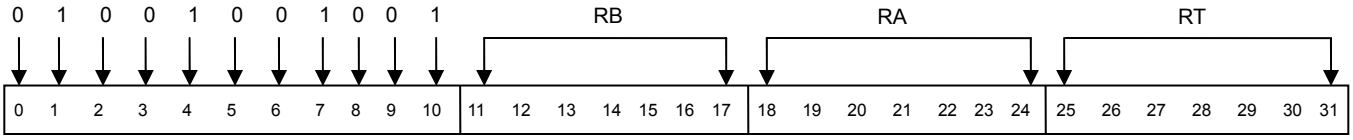
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA とレジスタ RB の値を、論理 OR 演算する。
- 結果の補数をもとめて、レジスタ RT に置く。

$RT^{0:3}$	$\leftarrow \neg(RA^{0:3} \mid RB^{0:3})$
$RT^{4:7}$	$\leftarrow \neg(RA^{4:7} \mid RB^{4:7})$
$RT^{8:11}$	$\leftarrow \neg(RA^{8:11} \mid RB^{8:11})$
$RT^{12:15}$	$\leftarrow \neg(RA^{12:15} \mid RB^{12:15})$

Equivalent 必須 v1.0

eqv **rt,ra,rb**



ワード・スロット 4個それぞれに、下記を行なう。

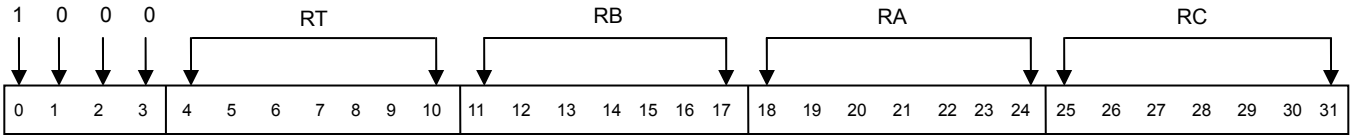
- レジスタ RA とレジスタ RB のビットが同一であれば、結果は '1' である。そうでなければ、'0' である。
- 結果をレジスタ RT に置く。

$RT^{0:3}$	$\leftarrow RA^{0:3} \oplus (\neg RB^{0:3})$
$RT^{4:7}$	$\leftarrow RA^{4:7} \oplus (\neg RB^{4:7})$
$RT^{8:11}$	$\leftarrow RA^{8:11} \oplus (\neg RB^{8:11})$
$RT^{12:15}$	$\leftarrow RA^{12:15} \oplus (\neg RB^{12:15})$

Select Bits

必須　v1.0

selb　　**rt,ra,rb,rc**



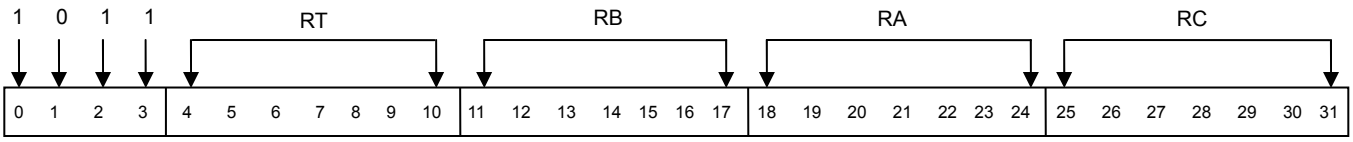
RC のビットを用いて RA または RB いずれかの対応ビットを選び、結果を生成する。

- レジスタ RC のビットが ’0’ であれば、レジスタ RA からビットを選択する。そうでなければ、レジスタ RB からビットを選択する。
- 選択したビットをレジスタ RT に置く。

$$RT^{0:15} \leftarrow RC^{0:15} \,\&\, RB^{0:15} \mid (\neg RC^{0:15}) \,\&\, RA^{0:15}$$

Shuffle Bytes 必須 v1.0

shufb rt,ra,rb,rc



レジスタ RA および RB を、RA の最下位ビットと RB の最上位ビットが隣接するように連結する。結果の値のバイトは、0 から 31 にナンバリングされたものと見なされる。

レジスタ RC および RT の各バイト・スロットについて、下記を行なう。

- レジスタ RC の値を調べて、結果のバイトは*表 5-1* に示すように生成される。
- 結果のバイトをレジスタ RT に挿入する。

表 5-1 レジスタ RC の 2 進値および結果バイト

レジスタ RC の値 （2 進表現）	結果バイト
10xxxxxx	0x00
110xxxxx	0xFF
111xxxxx	0x80
上記以外	レジスタ RC の右端 5 bit でアドレス指定された連結レジスタのバイト

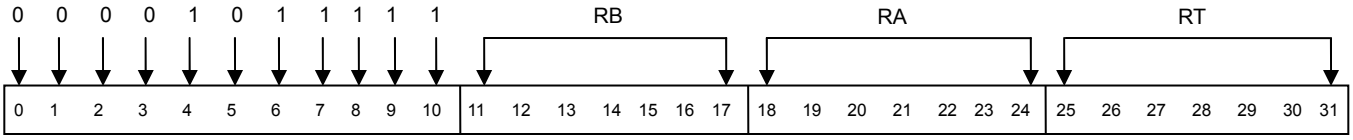
```
Rconcat ← RA || RB
for j = 0 to 15
        b ← RC^j
        If b_0:1 = 0b10 then c ← 0x00
        else If b_0:2 = 0b110 then c ← 0xFF
        else If b_0:2 = 0b111 then c ← 0x80
        else
                b ← b & 0x1F;
                c ← Rconcat^b;
        RT^j ← c
end
```

6. シフトおよびローテート命令

本セクションでは、SPU のシフト命令とローテート命令を説明する。

Shift Left Halfword 必須 v1.0

shlh rt,ra,rb



ハーフワード・スロット 8個それぞれに、下記を行なう。

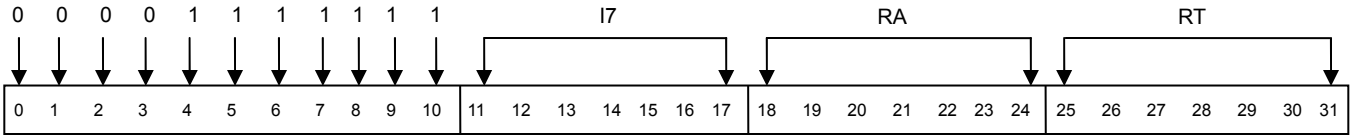
- レジスタ RA の値を、レジスタ RB のビット 11 から 15 のカウントに従い、左にシフトする。
- 結果をレジスタ RT に置く。
- カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。カウントが 15 よりも大きい場合、結果は 0 である。
- ハーフワードの左端からシフトアウトされたビットは破棄し、0 が右側からシフトインする。

注意：ハーフワードの各スロットは、それぞれ独自のシフト量を持つ。

```
for j = 0 to 15 by 2
        s ← RB^(j::2) & 0x001F
        t ← RA^(j::2)
        for b = 0 to 15
                if b + s < 16 then r_b ← t_(b + s)
                else               r_b ← 0
        end
        RT^(j::2) ← r
end
```

Shift Left Halfword Immediate 必須 v1.0

shlhi rt,ra,value



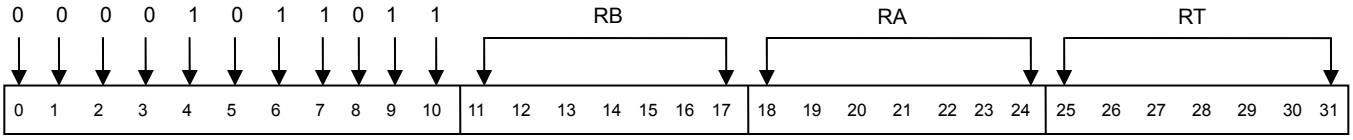
ハーフワード・スロット 8個それぞれに、下記を行なう。

- レジスタ RA の値を、I7 フィールドのビット 13 から 17（右端 5 bit）のカウントに従い、左にシフトする。
- 結果をレジスタ RT に置く。
- カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。カウントが 15 よりも大きい場合、結果は 0 である。
- ハーフワードの左端からシフトアウトされたビットは破棄し、0 が右側からシフトインする。

```
s ← RepLeftBit(I7,16) & 0x001F
for j = 0 to 15 by 2
        t ← RA^(j::2)
        for b = 0 to 15
                if b + s < 16 then r_b ← t_(b + s)
                else               r_b ← 0
        end
        RT^(j::2) ← r
end
```

Shift Left Word 必須 v1.0

shl **rt,ra,rb**



ワード・スロット 4個それぞれに、下記を行なう。

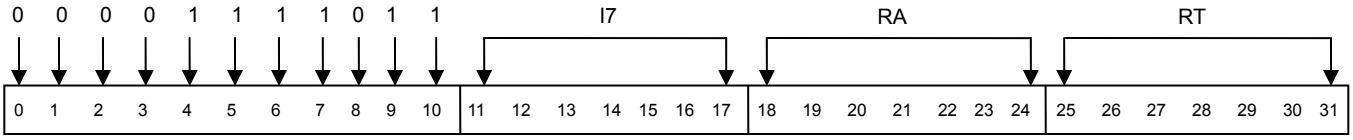
- レジスタ RA の値を、レジスタ RB のビット 26 から 31 のカウントに従い、左にシフトする。
- 結果をレジスタ RT に置く。
- カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。カウントが 31 よりも大きい場合、結果は 0 である。
- ワードの左端からシフトアウトされたビットは破棄し、0 が右側からシフトインする。

注意： ワードの各スロットは、それぞれ独自のシフト量を持つ。

```
for j = 0 to 15 by 4
        s ← RB^(j:4) & 0x0000003F
        t ← RA^(j:4)
        for b = 0 to 31
                if b + s < 32 then r_b ← t_(b+s)
                else               r_b ← 0
        end
        RT^(j:4) ← r
end
```

Shift Left Word Immediate 必須 v1.0

shli rt,ra,value



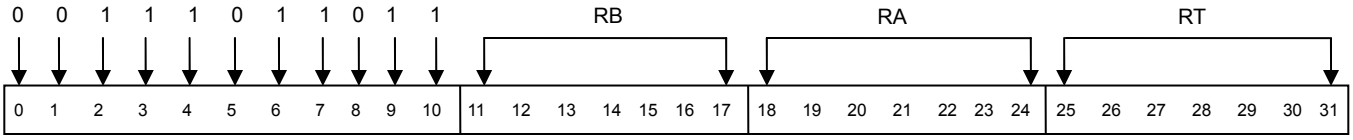
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の値を、I7 フィールドのビット 12 から 17（右端 6 bit）のカウントに従い、左にシフトする。
- 結果をレジスタ RT に置く。
- カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。カウントが 31 よりも大きい場合、結果は 0 である。
- ワードの左端からシフトアウトされたビットは破棄し、0 が右側からシフトインする。

```
s ← RepLeftBit(I7,32) & 0x0000003F
for j = 0 to 15 by 4
        t ← RA^{j::4}
        for b = 0 to 31
                if b + s < 32 then r_b ← t_{b + s}
                else               r_b ← 0
        end
        RT^{j::4} ← r
end
```

Shift Left Quadword by Bits 必須 v1.0

shlqbi rt,ra,rb



レジスタ RA の値を、レジスタ RB のプリファード・スロットのビット 29 から 31 のカウントに従い、左にシフトする。結果をレジスタ RT に置く。最大 7 ビット・ポジションまでのシフトが可能である。

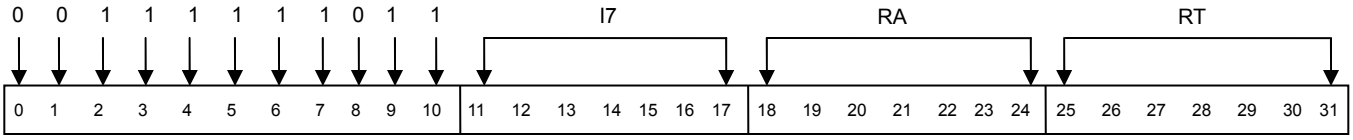
カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。

レジスタの左端からシフトアウトされたビットは破棄し、0 が右側からシフトインする。

```
s ← RB29:31
for b = 0 to 127
        if b + s < 128 then rb ← RAb + s
        else                rb ← 0
end
RT ← r
```

Shift Left Quadword by Bits Immediate 必須 v1.0

shlqbii rt,ra,value



レジスタ RA の値を、I7 フィールドのビット 15 から 17（右端 3 bit）のカウントに従い、左にシフトする。結果をレジスタ RT に置く。シフトは、最大 7 ビット・ポジションまでのシフトが可能である。

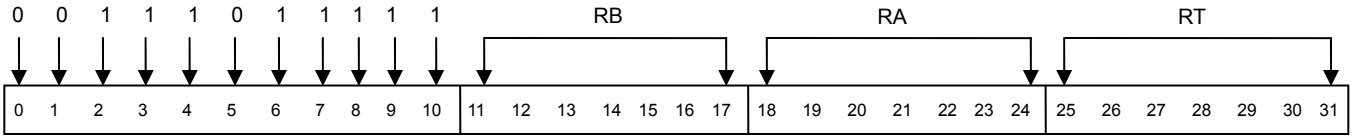
カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。

レジスタの左端からシフトアウトされたビットは破棄し、0 が右側からシフトインする。

```
s ← I7 & 0x07
for b = 0 to 127
        if b + s < 128 then r_b ← RA_(b + s)
        else                r_b ← 0
end
RT ← r
```

Shift Left Quadword by Bytes 必須 v1.0

shlqby rt,ra,rb



レジスタ RA のバイトを、レジスタ RB のプリファード・スロットのビット 27 から 31 のカウントに従い、左にシフトする。結果をレジスタ RT に置く。

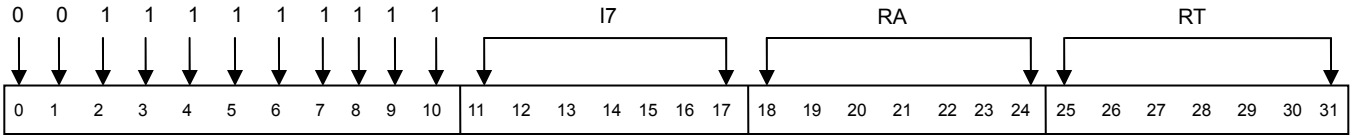
カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。カウントが 15 よりも大きい場合、結果は 0 である。

レジスタの左端からシフトアウトされたバイトは破棄し、ゼロのバイトが右側からシフトインする。

```
s ← RB27:31
for b = 0 to 15
        if b + s < 16 then rb ← RAb + s
        else               rb ← 0
end
RT ← r
```

Shift Left Quadword by Bytes Immediate 必須 v1.0

shlqbyi rt,ra,value



レジスタ RA のバイトを、I7 フィールドのビット 13 から 17（右端 5 bit）のカウントに従い、左にシフトする。結果をレジスタ RT に置く。

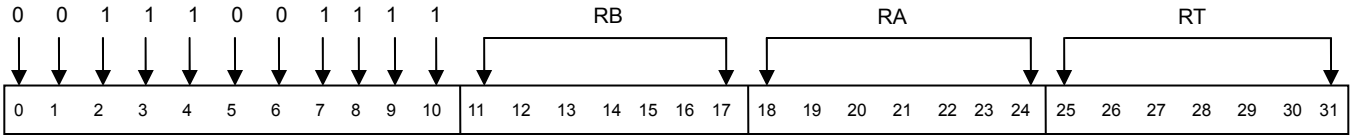
カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。カウントが 15 よりも大きい場合、結果は 0 である。

レジスタの左端からシフトアウトされたバイトは破棄し、ゼロのバイトが右側からシフトインする。

```
s ← I7 & 0x1F
for b = 0 to 15
        if b + s < 16 then r^b ← RA^(b + s)
        else               r^b ← 0
end
RT ← r
```

Shift Left Quadword by Bytes from Bit Shift Count 必須 v1.0

shlqbybi rt,ra,rb



レジスタ RA のバイトを、レジスタ RB のプリファード・スロットのビット 24 から 28 のカウントに従い、左にシフトする。結果をレジスタ RT に置く。

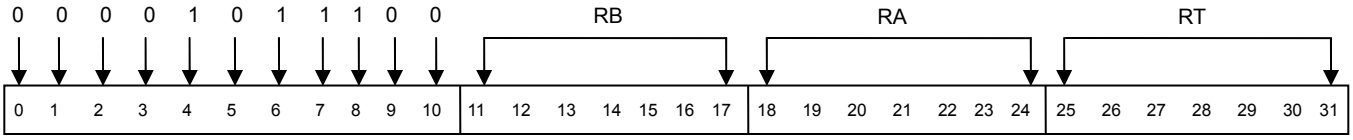
カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。カウントが 15 よりも大きい場合、結果は 0 である。

レジスタの左端からシフトアウトされたバイトは破棄し、ゼロのバイトが右側からシフトインする。

```
s ← RB24:28
for b = 0 to 15
        if b + s < 16 then rb ← RAb + s
        else               rb ← 0x00
end
RT ← r
```

Rotate Halfword 必須 v1.0

roth **rt,ra,rb**



ハーフワード・スロット 8個それぞれに、下記を行なう。

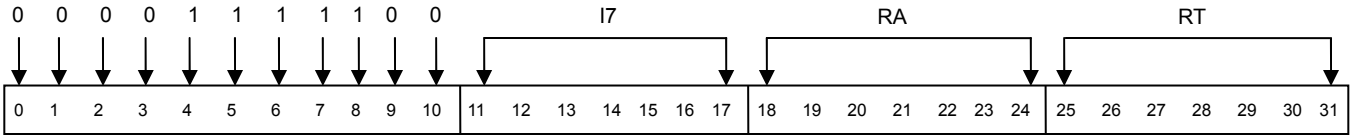
- レジスタ RA の値を、レジスタ RB のビット 12 から 15 のカウントに従い、左にローテートする。
- 結果をレジスタ RT に置く。
- カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。
- ハーフワードの左端からローテートアウトされたビットは、右端にローテートインする。

注意：ハーフワードの各スロットは、それぞれ独自のローテート量を持つ。

```
for j = 0 to 15 by 2
        s ← RB^(j::2) & 0x000F
        t ← RA^(j::2)
        for b = 0 to 15
                if b + s < 16 then r_b ← t_(b + s)
                else               r_b ← t_(b + s - 16)
        end
        RT^(j::2) ← r
end
```

Rotate Halfword Immediate 必須 v1.0

rothi **rt,ra,value**



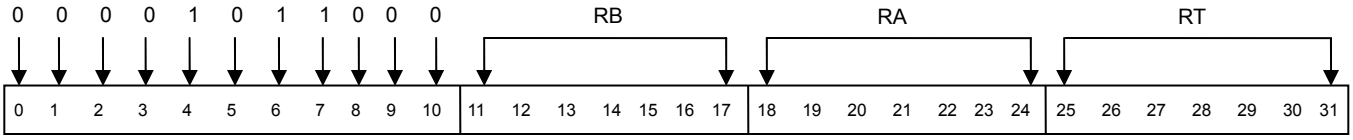
ハーフワード・スロット 8個それぞれに、下記を行なう。

- レジスタ RA の値を、I7 フィールドのビット 14 から 17（右端 4 bit）のカウントに従い、左にローテートする。
- 結果をレジスタ RT に置く。
- カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。
- ハーフワードの左端よりローテートアウトされたビットは、右端にローテートインする。

```
s ← RepLeftBit(I7,16) & 0x000F
for j = 0 to 15 by 2
        t ← RA^(j:2)
        for b = 0 to 15
                if b + s < 16 then r_b ← t_(b + s)
                else               r_b ← t_(b + s - 16)
        end
        RT^(j:2) ← r
end
```

Rotate Word 必須 v1.0

rot **rt,ra,rb**



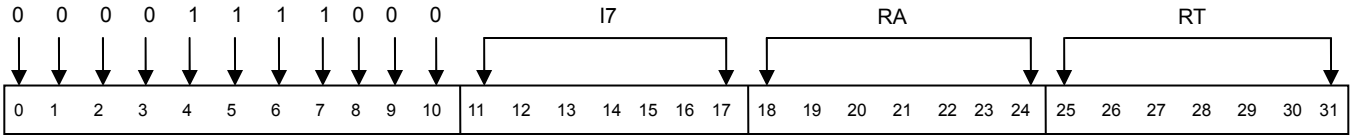
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の値を、レジスタ RB のビット 27 から 31 のカウントに従い、左にローテートする。
- 結果をレジスタ RT に置く。
- カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。
- ワードの左端からローテートアウトされたビットは、右端にローテートインする。

```
for j = 0 to 15 by 4
        s ← RB^(j::4) & 0x0000001F
        t ← RA^(j::4)
        for b = 0 to 31
                if b + s < 32 then r_b ← t_(b + s)
                else               r_b ← t_(b + s - 32)
        end
        RT^(j::4) ← r
end
```

Rotate Word Immediate 必須 v1.0

roti **rt,ra,value**



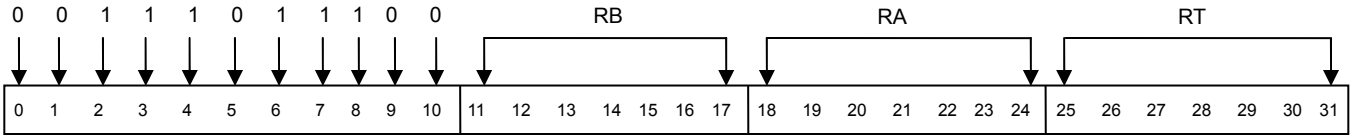
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の値を、I7 フィールドのビット 13 から 17（右端 5 bit）のカウントに従い、左にローテートする。
- 結果をレジスタ RT に置く。
- カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。
- ワードの左端よりローテートアウトされたビットは、右端にローテートインする。

```
s ← RepLeftBit(I7,32) & 0x0000001F
for j = 0 to 15 by 4
        t ← RA^(j::4)
        for b = 0 to 31
                if b + s < 32 then r_b ← t_(b + s)
                else               r_b ← t_(b + s - 32)
        end
        RT^(j::4) ← r
end
```

Rotate Quadword by Bytes 必須 v1.0

rotqby **rt,ra,rb**



レジスタ RA のバイトを、レジスタ RB のプリファード・スロットの右端 4 bit のカウントに従い、左にローテートする。結果をレジスタ RT に置く。最大 15 バイト・ポジションまでのローテーションが可能である。

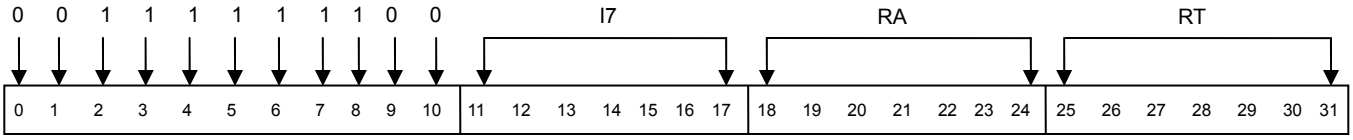
カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。

レジスタの左端よりローテートアウトされたバイトは、右にローテートインする。

```
s ← RB28:31
for b = 0 to 15
        if b + s < 16 then rb ← RAb+s
        else               rb ← RAb+s−16
end
RT ← r
```

Rotate Quadword by Bytes Immediate 必須 v1.0

rotqbyi rt,ra,value



レジスタ RA のバイトを、I7 フィールドの右端 4 bit のカウントに従い、左にローテートする。結果をレジスタ RT に置く。最大 15 バイト・ポジションまでのローテーションが可能である。

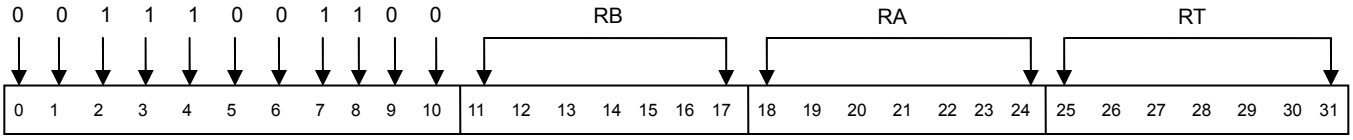
カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。

レジスタの左端からローテートアウトされたバイトは、右にローテートインする。

```
s ← I7 & 0x0F
for i = 0 to 15
        if b + s < 16 then r^b ← RA^(b+s)
        else               r^b ← RA^(b+s−16)
end
RT ← r
```

Rotate Quadword by Bytes from Bit Shift Count 必須 v1.0

rotqbybi rt,ra,rb



レジスタ RA のバイトを、レジスタ RB のプリファード・スロットのビット 25 から 28 のカウントに従い、左にローテートする。結果をレジスタ RT に置く。

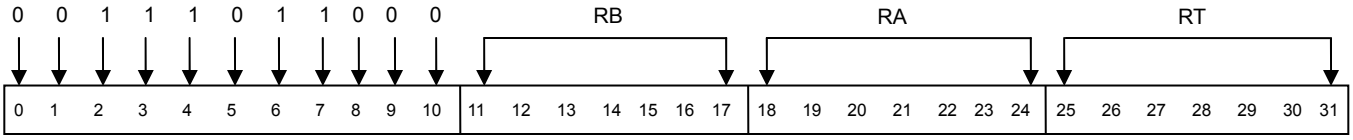
カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。

レジスタの左からローテートアウトされたバイトは、右にローテートインする。

```
s ← RB25:28
for b = 0 to 15
        if b + s < 16 then rb ← RAb + s
        else               rb ← RAb + s - 16
end
RT ← r
```

Rotate Quadword by Bits 必須 v1.0

rotqbi rt,ra,rb



レジスタ RA の値を、レジスタ RB のプリファード・スロットのビット 29 から 31 のカウントに従い、左にローテートする。結果をレジスタ RT に置く。最大 7 ビット・ポジションまでのローテーションが可能である。

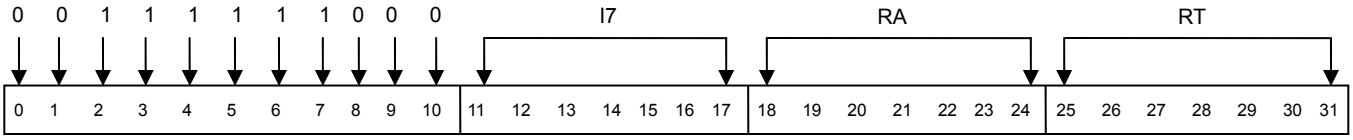
カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。

レジスタの左端からローテートアウトされたビットは、右にローテートインする。

```
s ← RB29:31
for b = 0 to 127
        if b + s < 128 then rb ← RAb + s
        else                rb ← RAb + s - 128
end
RT ← r
```

Rotate Quadword by Bits Immediate 必須 v1.0

rotqbii rt,ra,value



レジスタ RA の値を、I7 フィールドのビット 15 から 17（右端 3 bit）のカウントに従い、左にローテートする。結果をレジスタ RT に置く。最大 7 ビット・ポジションまでのローテーションが可能である。

カウントが 0 である場合、レジスタ RA の値を変更なしにレジスタ RT にコピーする。

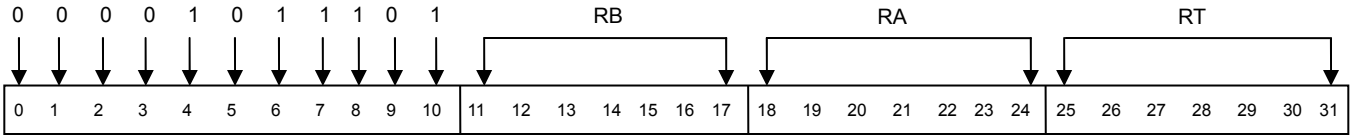
レジスタの左端よりローテートアウトされたビットは、右にローテートインする。

```
s ← I7 & 0x07
for b = 0 to 127
        if b + s < 128 then r_b ← RA_(b + s)
        else                r_b ← RA_(b + s - 128)
end
RT ← r
```

Rotate and Mask Halfword

必須　v1.0

rothm　　**rt,ra,rb**



ハーフワード・スロット 8個それぞれに、下記を行なう。

- shift_count は、(0 - RB) modulo 32 である。
- shift_count が 16 よりも小さい場合、RT を shift_count ビット分右にシフトした RA の値に設定し、左を 0 で満たす。
- そうでなければ、RT を 0 に設定する。

注意：ハーフワードの各スロットは、それぞれ独自のローテート量を持つ。

```
for j = 0 to 15 by 2
        s ← (0-RB^j::2) & 0x001F
        t ← RA^j::2
        for b = 0 to 15
                if b ≥ s then r_b ← t_b - s
                else          r_b ← 0
        end
        RT^j::2 ← r
end
```

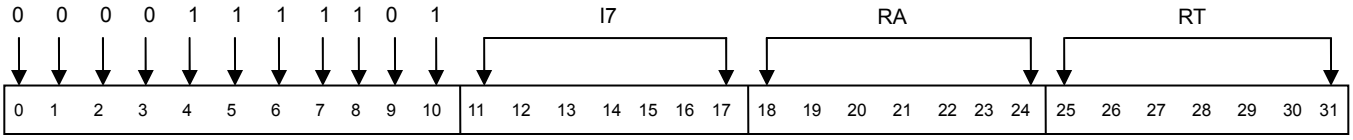
プログラミングの注意：*Rotate and Mask* 命令は論理右シフトを提供し、*Rotate and Mask Algebraic* 命令は算術右シフトを提供する。これらは、命令が受け付けるシフト量が右シフト量の 2 の補数であるという点で、従来の論理右シフトまたは算術右シフトとは異なる。よって、R1 で与えたビット数で、R2 の値を論理的に右シフトするには、下記のシーケンスを使用することができる。

```
sfi     r3,r1,0     Form two's complement
rotm    r4,r2,r3    Rotate, then mask
```

これらの命令の即値形式については、アセンブルやコンパイルにおいて、2 の補数のシフト量を生成することができる。

Rotate and Mask Halfword Immediate 必須 v1.0

rothmi **rt,ra,value**



ハーフワード・スロット 8個それぞれに、下記を行なう。

- shift_count は、(0 - I7) modulo 32 である。
- shift_count が 16 よりも小さい場合、RT を shift_count ビット分右にシフトした RA の値に設定し、左を 0 で満たす。
- そうでなければ、RT を 0 に設定する。

```
s ← (0 - RepLeftBit(I7,32)) & 0x0000001F
for j = 0 to 15 by 2
        t ← RA^(j::2)
        for b = 0 to 15
                if b ≥ s then r_b ← t_(b - s)
                else          r_b ← 0
        end
        RT^(j::2) ← r
end
```

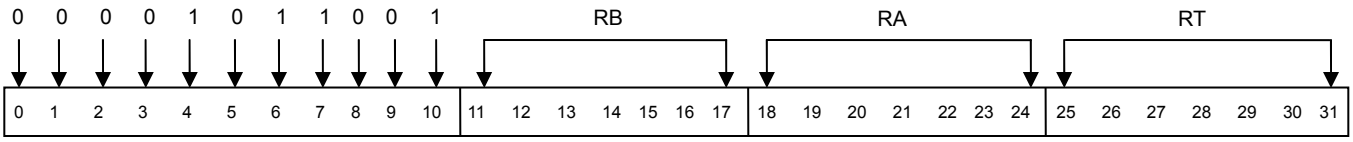
プログラミングの注意：*Rotate and Mask* 命令は論理右シフトを提供し、*Rotate and Mask Algebraic* 命令は算術右シフトを提供する。これらは、命令が受け付けるシフト量が右シフト量の 2 の補数であるという点で、従来の論理右シフトまたは算術右シフトとは異なる。よって、R1 で与えたビット数で、R2 の値を論理的に右シフトするには、下記のシーケンスを使用することができる。

```
sfi     r3,r1,0     Form two's complement
rotm    r4,r2,r3    Rotate, then mask
```

これらの命令の即値形式については、アセンブルやコンパイルにおいて、2 の補数のシフト量を生成することができる。

Rotate and Mask Word 必須 v1.0

rotm **rt,ra,rb**



ワード・スロット 4個それぞれに、下記を行なう。

- shift_count は、(0 - RB) modulo 64 である。
- shift_count が 32 よりも小さい場合、RT を shift_count ビット分右にシフトした RA の値に設定し、左を 0 で満たす。
- そうでなければ、RT を 0 に設定する。

```
for j = 0 to 15 by 4
        s ← (0 - RB^(j:4)) & 0x0000003F
        t ← RA^(j:4)
        for b = 0 to 31
                if b ≥ s then r_b ← t_(b - s)
                else          r_b ← 0
        end
        RT^(j:4) ← r
end
```

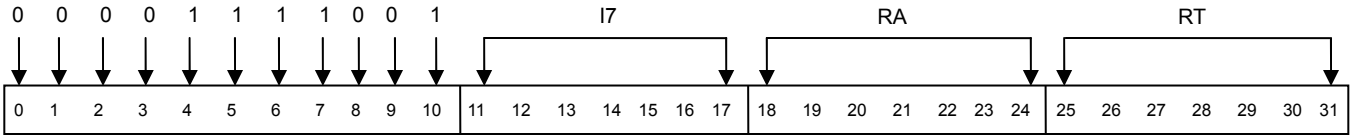
プログラミングの注意：*Rotate and Mask* 命令は論理右シフトを提供し、*Rotate and Mask Algebraic* 命令は算術右シフトを提供する。これらは、命令が受け付けるシフト量が右シフト量の 2 の補数であるという点で、従来の論理右シフトまたは算術右シフトとは異なる。よって、R1 で与えたビット数で、R2 の値を論理的に右シフトするには、下記のシーケンスを使用することができる。

```
sfi     r3,r1,0     Form two's complement
rotm    r4,r2,r3    Rotate, then mask
```

これらの命令の即値形式については、アセンブルやコンパイルにおいて、2 の補数のシフト量を生成することができる。

Rotate and Mask Word Immediate 必須 v1.0

rotmi **rt,ra,value**



ワード・スロット 4個それぞれに、下記を行なう。

- shift_count は、(0 - I7) modulo 64 である。
- shift_count が 32 よりも小さい場合、RT を shift_count ビット分右にシフトした RA の値に設定し、左を 0 で満たす。
- そうでなければ、RT を 0 に設定する。

```
s ← (0 - RepLeftBit(I7,32)) & 0x0000003F
for j = 0 to 15 by 4
        t ← RA^(j:4)
        for b = 0 to 31
                if b ≥ s then r_b ← t_(b - s)
                else          r_b ← 0
        end
        RT^(j:4) ← r
end
```

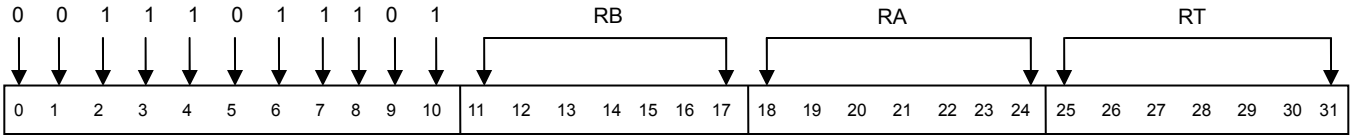
プログラミングの注意：*Rotate and Mask* 命令は論理右シフトを提供し、*Rotate and Mask Algebraic* 命令は算術右シフトを提供する。これらは、命令が受け付けるシフト量が右シフト量の 2 の補数であるという点で、従来の論理右シフトまたは算術右シフトとは異なる。よって、R1 で与えたビット数で、R2 の値を論理的に右シフトするには、下記のシーケンスを使用することができる。

```
sfi     r3,r1,0     Form two's complement
rotm    r4,r2,r3    Rotate, then mask
```

これらの命令の即値形式については、アセンブルやコンパイルにおいて、2 の補数のシフト量を生成することができる。

Rotate and Mask Quadword by Bytes 必須 v1.0

rotqmby rt,ra,rb

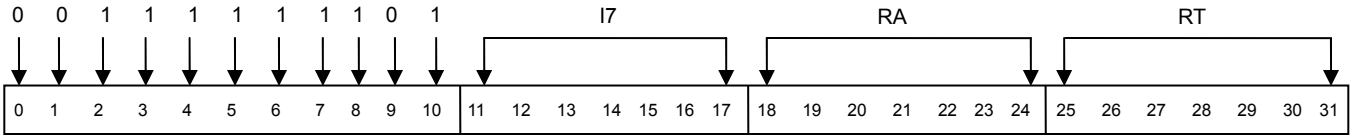


shift_count は、(0 – RB のプリファード・ワード) modulo 32 である。shift_count が 16 よりも小さい場合、RT を shift_count バイト分右にシフトした RA の値に設定し、左を 0x00 のバイトで満たす。そうでなければ、RT を 0 に設定する。

```
s ← (0 - RB27:31) & 0x1F
for b = 0 to 15
        if b ≥ s then rb ← tb - s
        else          rb ← 0x00
end
RT ← r
```

Rotate and Mask Quadword by Bytes Immediate 必須 v1.0

rotqmbyi rt,ra,value

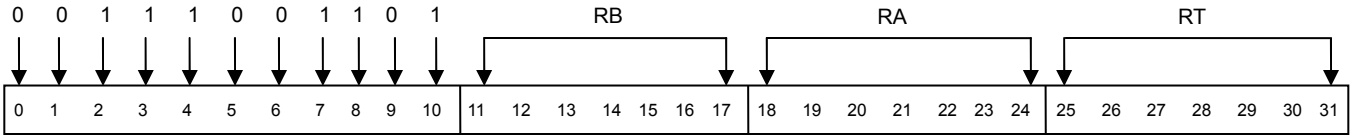


shift_count は、(0 – I7) modulo 32 である。shift_count が 16 よりも小さい場合、RT を shift_count バイト分右にシフトした RA の値に設定し、左を 0x00 のバイトで満たす。そうでなければ、RT のすべてのバイトを 0x00 に設定する。

```
s ← (0 - I7) & 0x1F
for b = 0 to 15
        if b ≥ s then r^b ← t^(b - s)
        else          r^b ← 0x00
end
RT ← r
```

Rotate and Mask Quadword Bytes from Bit Shift Count 必須 v1.0

rotqmbybi rt,ra,rb

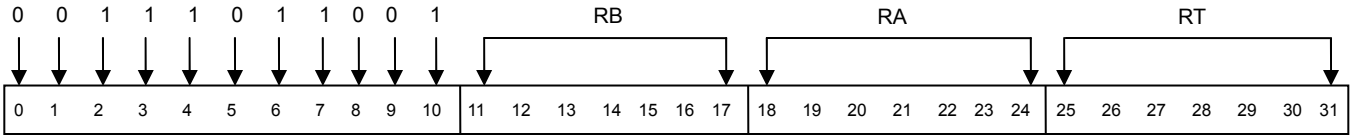


shift_count は、(0 － RB のビット 24 から 28) modulo 32 である。shift_count が 16 よりも小さい場合、RT を shift_count バイト分右にシフトした RA の値に設定し、左を 0x00 のバイトで満たす。そうでなければ、RT のすべてのバイトを 0x00 に設定する。

```
s ← (0 - RB24:28) & 0x1F
for b = 0 to 15
        if b ≥ s then rb ← RAb - s
        else          rb ← 0x00
end
```

Rotate and Mask Quadword by Bits 必須 v1.0

rotqmbi rt,ra,rb

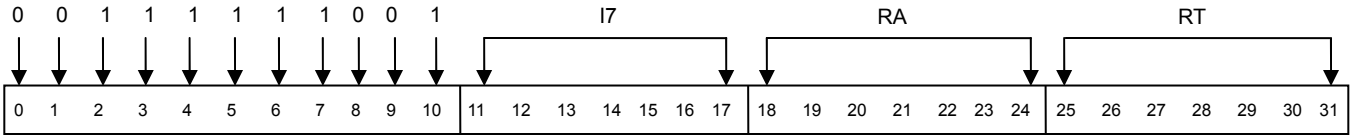


shift_count は、(0 – RB のプリファード・ワード) modulo 8 である。RT を shift_count ビット分右にシフトした RA の値に設定し、左を 0 のビットで満たす。

```
s ← (0 - RB29:31) & 0x07
for b = 0 to 127
        if b ≥ s then rb ← tb - s
        else          rb ← 0
end
RT ← r
```

Rotate and Mask Quadword by Bits Immediate 必須 v1.0

rotqmbii rt,ra,value



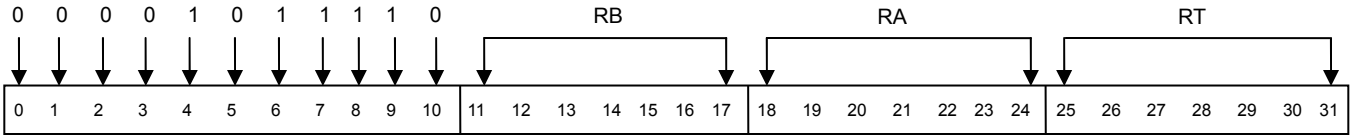
shift_count は、(0 - I7) modulo 8 である。

RT を shift_count ビット分右にシフトした RA の値に設定し、左を 0 のビットで満たす。

```
s ← (0 - I7) & 0x07
for b = 0 to 127
        if b ≥ s then r_b ← t_(b - s)
        else          r_b ← 0
end
RT ← r
```

Rotate and Mask Algebraic Halfword 必須 v1.0

rotmah **rt,ra,rb**



ハーフワード・スロット 8個それぞれに、下記を行なう。

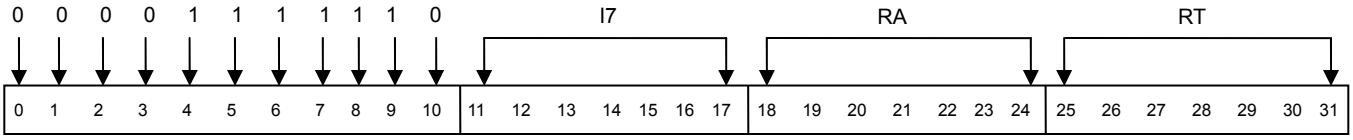
- shift_count は、(0 - RB) modulo 32 である。
- shift_count が 16 よりも小さい場合、RT を shift_count ビット分右にシフトした RA の値に設定し、左には（ハーフワードの）ビット 0 を複製する。
- そうでなければ、RT のこのハーフワードのすべてのビットを、RA のこのハーフワードのビット 0 に設定する。

注意：ハーフワードの各スロットは、それぞれ独自のローテート量を持つ。

```
for j = 0 to 15 by 2
        s ← (0 - RB^(j::2)) & 0x001F
        t ← RA^(j::2)
        for b = 0 to 15
                if b ≥ s then r_b ← t_(b - s)
                else          r_b ← t_0
        end
        RT^(j::2) ← r
end
```

Rotate and Mask Algebraic Halfword Immediate 必須 v1.0

rotmahi rt,ra,value



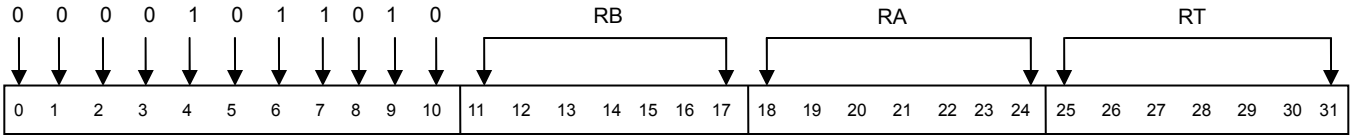
ハーフワード・スロット 8個それぞれに、下記を行なう。

- shift_count は、(0 - I7) modulo 32 である。
- shift_count が 16 よりも小さい場合、RT を shift_count ビット分右にシフトした RA の値に設定し、左には（ハーフワードの）ビット 0 を複製する。
- そうでなければ、RT のこのハーフワードのすべてのビットを、RA のこのハーフワードのビット 0 に設定する。

```
s ← (0 - RepLeftBit(I7,16)) & 0x001F
for j = 0 to 15 by 2
        t ← RA^(j::2)
        for b = 0 to 15
                if b ≥ s then r_b ← t_(b - s)
                else          r_b ← t_0
        end
        RT^(j::2) ← r
end
```

Rotate and Mask Algebraic Word 必須 v1.0

rotma **rt,ra,rb**



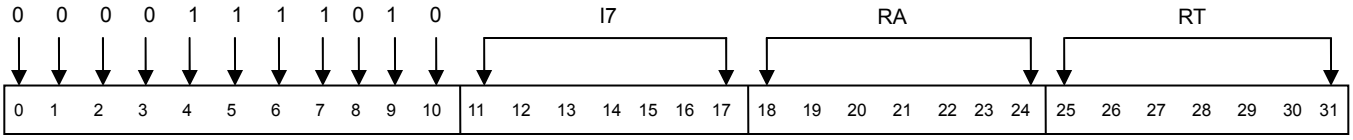
ワード・スロット 4個それぞれに、下記を行なう。

- shift_count は、(0 - RB) modulo 64 である。
- shift_count が 32 よりも小さい場合、RT を shift_count ビット分右にシフトした RA の値に設定し、左には（ワードの）ビット 0 を複製する。
- そうでなければ、RT のこのワードのすべてのビットを、RA のこのワードのビット 0 に設定する。

```
for j = 0 to 15 by 4
        s ← (0 - RB^(j::4)) & 0x0000003F
        t ← RA^(j::4)
        for b = 0 to 31
                if b ≥ s then r_b ← t_(b - s)
                else          r_b ← t_0
        end
        RT^(j::4) ← r
end
```

Rotate and Mask Algebraic Word Immediate 必須 v1.0

rotmai **rt,ra,value**



ワード・スロット 4個それぞれに、下記を行なう。

- shift_count は、(0 - I7) modulo 64 である。
- shift_count が 32 よりも小さい場合、RT を shift_count ビット分右にシフトした RA の値に設定し、左には（ワードの）ビット 0 を複製する。
- そうでなければ、RT のこのワードのすべてのビットを、RA のこのワードのビット 0 に設定する。

```
s ← (0 - RepLeftBit(I7,32)) & 0x0000003F
for j = 0 to 15 by 4
        t ← RA^(j::4)
        for b = 0 to 31
                if b ≥ s then r_b ← t_(b - s)
                else          r_b ← t_0
        end
        RT^(j::4) ← r
end
```

7. 比較、分岐、および停止命令

本セクションでは、SPU の比較命令、分岐命令、および停止（halt）命令をリストアップし、説明する。SPU 割り込み機能の詳細については、251 ページの *セクション 12* を参照すること。

条件分岐命令は、専用のコンディション・コード・レジスタにアクセスするのではなく、レジスタの値を調べて動作する。値は、プリファード・スロットから取得する。通常、値は比較命令によって設定される。

比較命令は、2 つのレジスタ中の値の比較、あるいはレジスタ中の値と即値の比較を行なう。結果は、ターゲットレジスタに、レジスタ・オペランドと同一幅の結果値を設定することによって示される。比較条件が満たされればその値はすべてのビットが 1 であり、そうでなければ値はすべてのビットが 0 である。

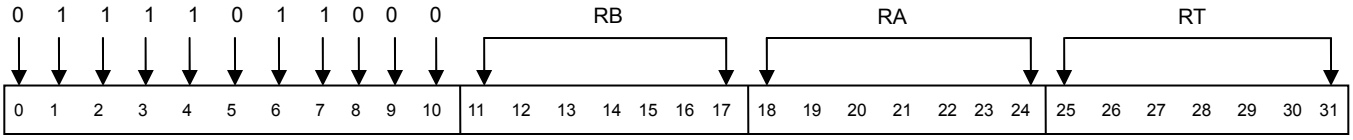
論理比較命令は、オペランドを符号なし整数として扱う。その他の比較命令は、オペランドを 2 の補数の符号付き整数として扱う。

「停止」命令のセットは、テストされた条件が満たされた時に実行を停止する。これらの命令は、例えば、条件を満たせないと重大なエラーとみなされる状況において、アドレスや添字範囲のチェックに使用することを意図している。発生する停止は precise ではない（停止場所が厳密ではない）ので、その結果、停止された実行は一般には再開不可能である。

浮動小数点比較命令は、その他の浮動小数点命令とともに 195 ページの *セクション 9*「*浮動小数点命令*」にリストアップされている。

Halt If Equal 必須 v1.0

heq **ra,rb**



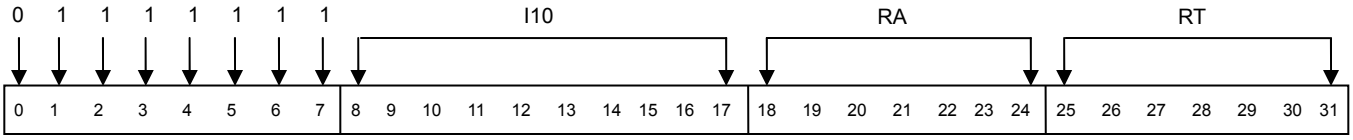
レジスタ RA のプリファード・スロットの値を、レジスタ RB のプリファード・スロットの値と比較する。値が等しければ、当該 halt 命令で、あるいはその後で、プログラムの実行を停止する。

プログラミングの注意：RT は、偽のターゲットである。実装によっては、この命令があたかも RT に格納する値を生成するかのように、命令スケジューリングを行なう場合がある。後続近傍の命令に対するソースデータにならないように、RT をプログラミングすることによって、プログラムは不必要な遅延を避けることができる。偽のターゲットへは何も書き込まれない。

If $RA^{0:3}$ = $RB^{0:3}$ then
Stop after executing zero or more instructions after the halt.

Halt If Equal Immediate 必須 v1.0

heqi **ra,symbol**



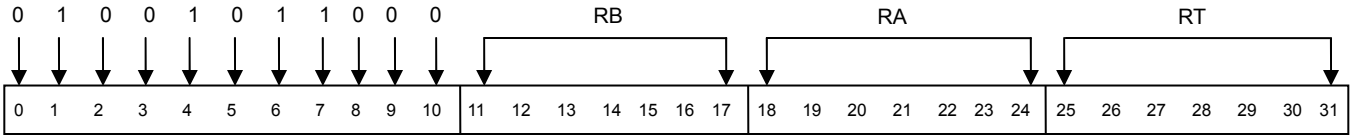
I10 フィールドの値を、左端のビットを複製することによって、32 bit まで拡張する。結果は、レジスタ RA のプリファード・スロットの値と比較される。レジスタ RA の値が即値と等しければ、当該 halt 命令で、あるいはその後で、SPU プログラムの実行を停止する。

プログラミングの注意：RT は、偽のターゲットである。実装によっては、この命令があたかも RT に格納する値を生成するかのように、命令スケジューリングを行なう場合がある。後続近傍の命令に対するソースデータにならないように、RT をプログラミングすることによって、プログラムは不必要な遅延を避けることができる。偽のターゲットへは何も書き込まれない。

If $RA^{0:3}$ = RepLeftBit(I10,32) then
Stop after executing zero or more instructions after the halt.

Halt If Greater Than 必須 v1.0

hgt ra,rb



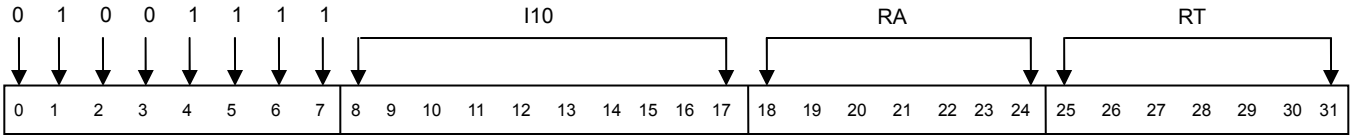
レジスタ RA のプリファード・スロットの値を、レジスタ RB のプリファード・スロットの値と代数的に比較する。レジスタ RA の値が、RB の値よりも大きければ、当該 halt 命令で、あるいはその後で、SPU プログラムの実行を停止する。

プログラミングの注意：RT は、偽のターゲットである。実装によっては、この命令があたかも RT に格納する値を生成するかのように、命令スケジューリングを行なう場合がある。後続近傍の命令に対するソースデータにならないように、RT をプログラミングすることによって、プログラムは不必要な遅延を避けることができる。偽のターゲットへは何も書き込まれない。

If $RA^{0:3} > RB^{0:3}$ then
Stop after executing zero or more instructions after the halt.

Halt If Greater Than Immediate 必須 v1.0

hgti ra,symbol



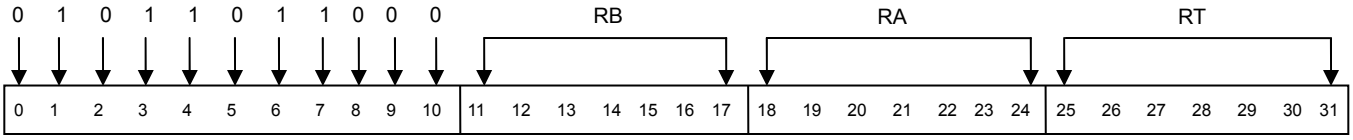
I10 フィールドの値を、左端のビットを複製することによって、32 bit に拡張する。結果は、レジスタ RA のプリファード・スロットの値と代数的に比較される。レジスタ RA の値が、即値よりも大きければ、当該 halt 命令で、あるいはその後で、SPU プログラムの実行を停止する。

プログラミングの注意：RT は、偽のターゲットである。実装によっては、この命令があたかも RT に格納する値を生成するかのように、命令スケジューリングを行なう場合がある。後続近傍の命令に対するソースデータにならないように、RT をプログラミングすることによって、プログラムは不必要な遅延を避けることができる。偽のターゲットへは何も書き込まれない。

```
If RA^0:3 > RepLeftBit(I10,32) then
        Stop after executing zero or more instructions after the halt.
```

Halt If Logically Greater Than 必須 v1.0

hlgt ra,rb



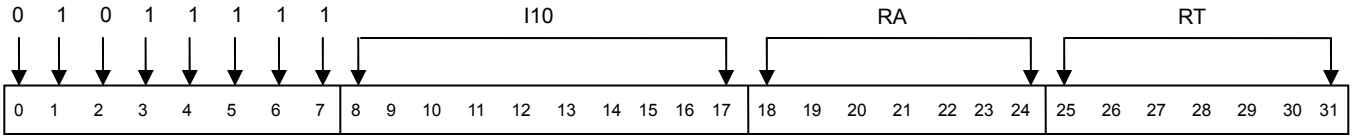
レジスタ RA のプリファード・スロットの値を、レジスタ RB のプリファード・スロットの値と論理的に比較する。レジスタ RA の値が、レジスタ RB の値よりも大きければ、当該 halt 命令で、あるいはその後で、SPU プログラムの実行を停止する。

プログラミングの注意：RT は、偽のターゲットである。実装によっては、この命令があたかも RT に格納する値を生成するかのように、命令スケジューリングを行なう場合がある。後続近傍の命令に対するソースデータにならないように、RT をプログラミングすることによって、プログラムは不必要な遅延を避けることができる。偽のターゲットへは何も書き込まれない。

If $RA^{0:3} >^{u} RB^{0:3}$ then
Stop after executing zero or more instructions after the halt.

Halt If Logically Greater Than Immediate 必須 v1.0

hlgti ra,symbol



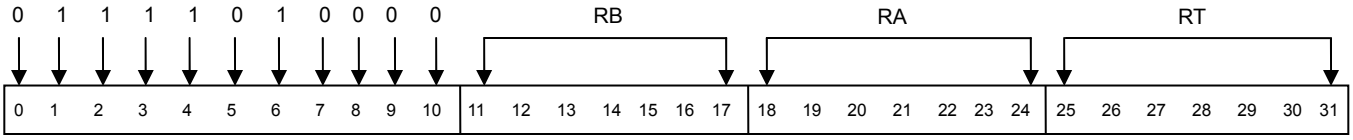
I10 フィールドの値を、左端のビットを複製することによって、32 bit に拡張する。結果は、レジスタ RA のプリファード・スロットの値と論理的に比較される。レジスタ RA の値が、即値よりも論理的に大きければ、当該 halt 命令で、あるいはその後で、SPU プログラムの実行を停止する。

プログラミングの注意：RT は、偽のターゲットである。実装によっては、この命令があたかも RT に格納する値を生成するかのように、命令スケジューリングを行なう場合がある。後続近傍の命令に対するソースデータにならないように、RT をプログラミングすることによって、プログラムは不必要な遅延を避けることができる。偽のターゲットへは何も書き込まれない。

If $RA^{0:3} >^{u}$ RepLeftBit(I10,32) then
Stop after executing zero or more instructions after the halt.

Compare Equal Byte 必須 v1.0

ceqb **rt,ra,rb**



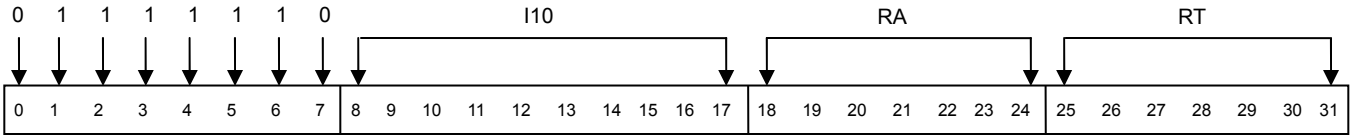
バイト・スロット 16個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドと比較する。オペランドが等しい場合は、すべて 1 のビット（真）の結果が生成される。等しくない場合は、すべて 0 のビット（偽）の結果が生成される。
- 8 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15
        If RA^i = RB^i then   RT^i ← 0xFF
        else                  RT^i ← 0x00
end
```

Compare Equal Byte Immediate 必須 v1.0

ceqbi rt,ra,value



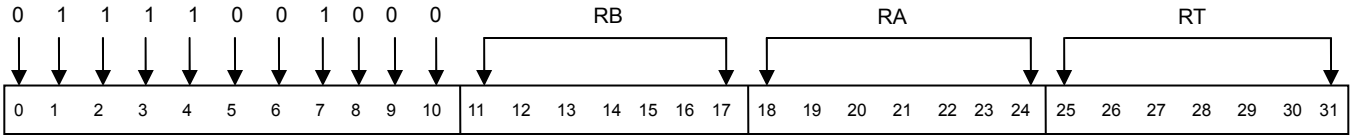
バイト・スロット 16個それぞれに、下記を行なう。

- I10 フィールドの右端 8 bit の値を、レジスタ RA の値と比較する。2 つの値が等しい場合は、すべて 1 のビット（真）の結果が生成される。等しくない場合は、すべて 0 のビット（偽）の結果が生成される。
- 8 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15
        If RA^i = I10_2:9 then RT^i ← 0xFF
        else                   RT^i ← 0x00
end
```

Compare Equal Halfword 必須 v1.0

ceqh **rt,ra,rb**



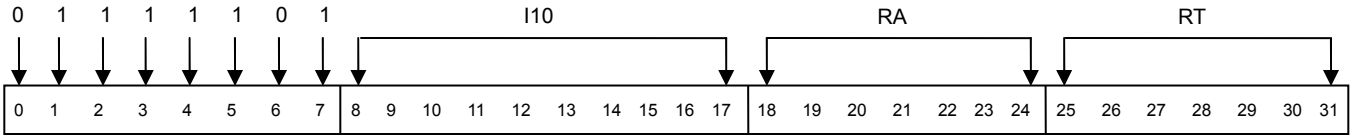
ハーフワード・スロット 8個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドと比較する。2 つのオペランドが等しい場合は、すべて 1 のビット（真）の結果が生成される。等しくない場合は、すべて 0 のビット（偽）の結果が生成される。
- 16 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15 by 2
        If RA^i::2 = RB^i::2 then RT^i::2 ← 0xFFFF
        else                      RT^i::2 ← 0x0000
end
```

Compare Equal Halfword Immediate 必須 v1.0

ceqhi rt,ra,value



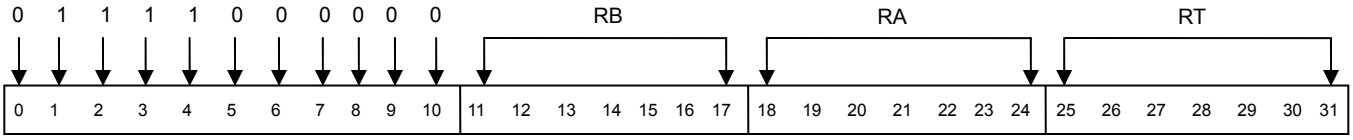
ハーフワード・スロット 8個それぞれに、下記を行なう。

- I10 フィールドの値を、左端のビットを複製することによって、16 bit に拡張し、レジスタ RA の値と比較する。2 つの値が等しい場合は、すべて 1 のビット（真）の結果が生成される。等しくない場合は、すべて 0 のビット（偽）の結果が生成される。
- 16 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15 by 2
        If RA^i::2 = RepLeftBit(I10,16) then RT^i::2 ← 0xFFFF
        else                                RT^i::2 ← 0x0000
end
```

Compare Equal Word 必須 v1.0

ceq **rt,ra,rb**



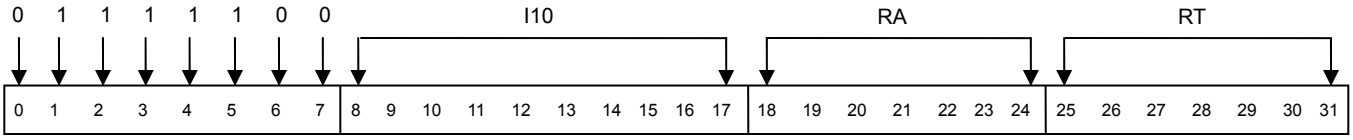
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドと比較する。2 つのオペランドが等しい場合は、すべて 1 のビット（真）の結果が生成される。等しくない場合は、すべて 0 のビット（偽）の結果が生成される。
- 32 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15 by 4
        If RA^i::4 = RB^i::4 then RT^i::4 ← 0xFFFFFFFF
        else                      RT^i::4 ← 0x00000000
end
```

Compare Equal Word Immediate 必須 v1.0

ceqi **rt,ra,value**



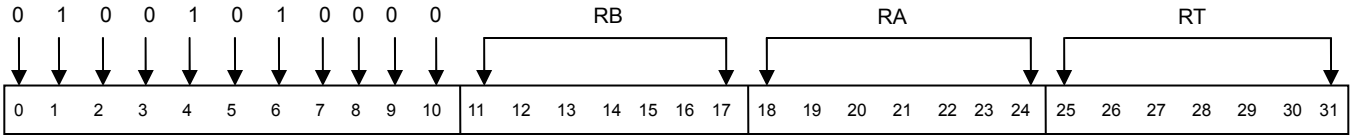
ワード・スロット 4個それぞれに、下記を行なう。

- I10 フィールドの値を、左端のビットを複製することによって、32 bit に拡張し、レジスタ RA の値と比較する。2 つの値が等しい場合は、すべて 1 のビット（真）の結果が生成される。等しくない場合は、すべて 0 のビット（偽）の結果が生成される。
- 32 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15 by 4
        If RA^i::4 = RepLeftBit(I10,32) then RT^i::4 ← 0xFFFFFFFF
        else                                RT^i::4 ← 0x00000000
end
```

Compare Greater Than Byte 必須 v1.0

cgtb **rt,ra,rb**



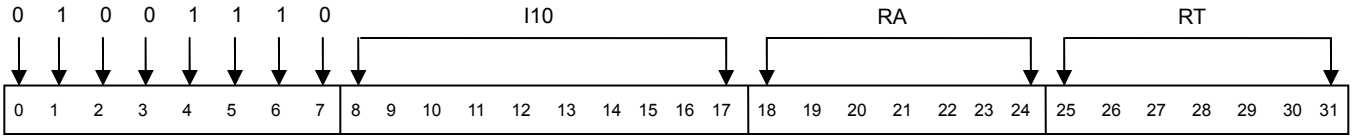
バイト・スロット 16個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドと代数的に比較する。レジスタ RA のオペランドがレジスタ RB のオペランドより大きい場合は、すべて 1 のビット（真）の結果が生成される。そうでなければ、すべて 0 のビット（偽）の結果が生成される。
- 8 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15
        If RA^i > RB^i then   RT^i ← 0xFF
        else                  RT^i ← 0x00
end
```

Compare Greater Than Byte Immediate 必須 v1.0

cgtbi **rt,ra,value**



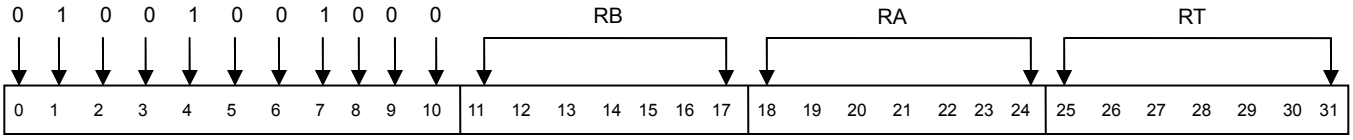
バイト・スロット 16個それぞれに、下記を行なう。

- I10 フィールドの右端の 8 bit を、レジスタ RA の値と代数的に比較する。レジスタ RA の値がより大きい場合は、すべて 1 のビット（真）の結果が生成される。そうでなければ、すべて 0 のビット（偽）の結果が生成される。
- 8 bit の結果をレジスタ RT に置く

```
for i = 0 to 15
        If RA^i > I10_2:9 then RT^i ← 0xFF
        else                   RT^i ← 0x00
end
```

Compare Greater Than Halfword 必須 v1.0

cgth rt,ra,rb



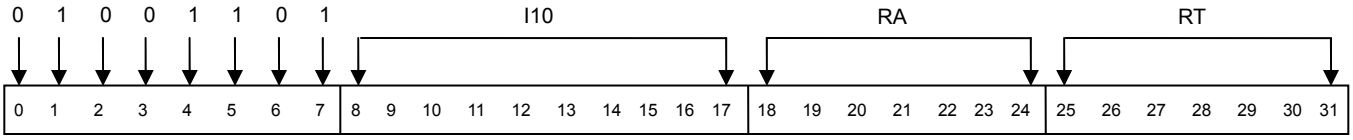
ハーフワード・スロット 8個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドと代数的に比較する。レジスタ RA のオペランドがレジスタ RB のオペランドより大きい場合は、すべて 1 のビット（真）の結果が生成される。そうでなければ、すべて 0 のビット（偽）の結果が生成される。
- 16 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15 by 2
        If RA^(i::2) > RB^(i::2) then RT^(i::2) ← 0xFFFF
        else                          RT^(i::2) ← 0x0000
end
```

Compare Greater Than Halfword Immediate 必須 v1.0

cgthi rt,ra,value



ハーフワード・スロット 8個それぞれに、下記を行なう。

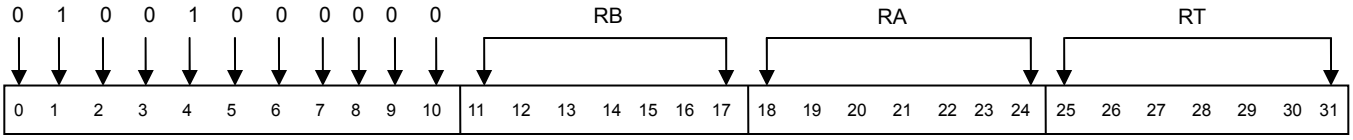
- I10 フィールドの値を 16 bit に拡張し、レジスタ RA の値と代数的に比較する。レジスタ RA の値が I10 フィールドの値より大きい場合は、すべて 1 のビット（真）の結果が生成される。そうでなければ、すべて 0 のビット（偽）の結果が生成される。

- 16 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15 by 2
        If RA^i::2 > RepLeftBit(I10,16) then RT^i::2 ← 0xFFFF
        else                                RT^i::2 ← 0x0000
end
```

Compare Greater Than Word 必須 v1.0

cgt **rt,ra,rb**



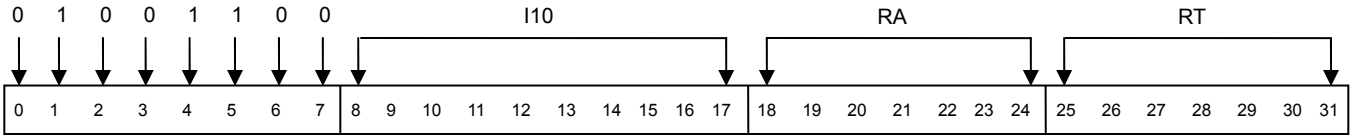
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドと代数的に比較する。レジスタ RA のオペランドがレジスタ RB のオペランドより大きい場合は、すべて 1 のビット（真）の結果が生成される。そうでなければ、すべて 0 のビット（偽）の結果が生成される。
- 32 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15 by 4
        If RA^i::4 > RB^i::4 then RT^i::4 ← 0xFFFFFFFF
        else                      RT^i::4 ← 0x00000000
end
```

Compare Greater Than Word Immediate 必須 v1.0

cgti rt,ra,value



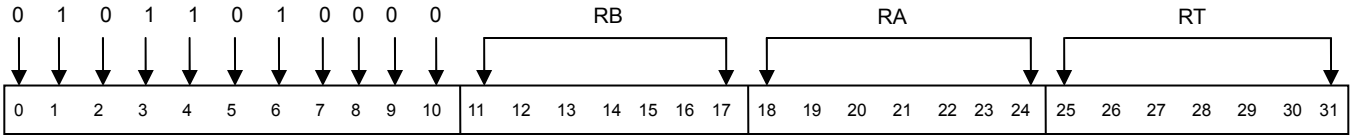
ワード・スロット 4個それぞれに、下記を行なう。

- I10 フィールドの値を、符号拡張によって 32 bit に拡張し、レジスタ RA の値と代数的に比較する。レジスタ RA の値が I10 フィールドの値より大きい場合は、すべて 1 のビット（真）の結果が生成される。そうでなければ、すべて 0 のビット（偽）の結果が生成される。
- 32 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15 by 4
        If RA^i:4 > RepLeftBit(I10,32) then RT^i:4 ← 0xFFFFFFFF
        else                               RT^i:4 ← 0x00000000
end
```

Compare Logical Greater Than Byte 必須 v1.0

clgtb rt,ra,rb



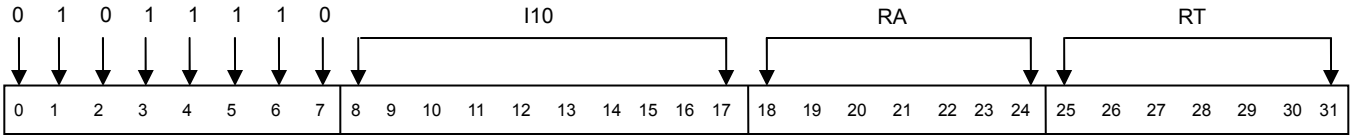
バイト・スロット 16個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドと論理的に比較する。レジスタ RA のオペランドがレジスタ RB のオペランドより論理的に大きい場合は、すべて 1 のビット（真）の結果が生成される。そうでなければ、すべて 0 のビット（偽）の結果が生成される。
- 8 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15
        If RA^i >^u RB^i then   RT^i ← 0xFF
        else                    RT^i ← 0x00
end
```

Compare Logical Greater Than Byte Immediate 必須 v1.0

clgtbi rt,ra,value



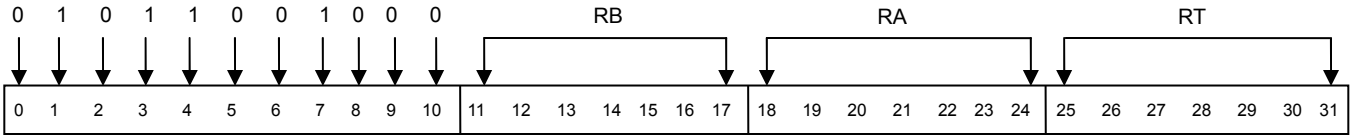
バイト・スロット 16個それぞれに、下記を行なう。

- I10 フィールドの右端 8 bit の値を、レジスタ RA の値と論理的に比較する。レジスタ RA の値の方が論理的に大きい場合は、すべて 1 のビット（真）の結果が生成される。そうでない場合は、すべて 0 のビット（偽）の結果が生成される。
- 8 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15
        If RA^i >^u I10_{2:9} then  RT^i ← 0xFF
        else                        RT^i ← 0x00
end
```

Compare Logical Greater Than Halfword 必須 v1.0

clgth rt,ra,rb



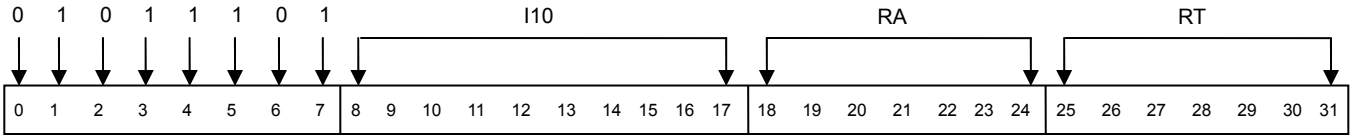
ハーフワード・スロット 8個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドと論理的に比較する。レジスタ RA のオペランドがレジスタ RB のオペランドより論理的に大きい場合は、すべて 1 のビット（真）の結果が生成される。そうでなければ、すべて 0 のビット（偽）の結果が生成される。
- 16 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15 by 2
        If RA^(i::2) >^u RB^(i::2) then RT^(i::2) ← 0xFFFF
        else                           RT^(i::2) ← 0x0000
end
```

Compare Logical Greater Than Halfword Immediate 必須 v1.0

clgthi rt,ra,value



ハーフワード・スロット 8個それぞれに、下記を行なう。

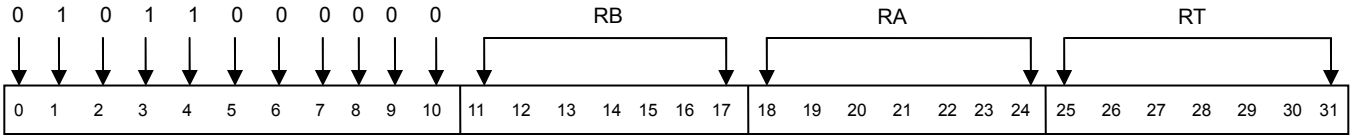
- I10 フィールドの値を、左端ビットを複製することによって、16 bit に拡張し、レジスタ RA の値と論理的に比較する。レジスタ RA の値が I10 フィールドの値より論理的に大きい場合は、すべて 1 のビット（真）の結果が生成される。そうでない場合は、すべて 0 のビット（偽）の結果が生成される。

- 16 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15 by 2
        If RA^(i::2) >^u RepLeftBit(I10,16) then RT^(i::2) ← 0xFFFF
        else                                    RT^(i::2) ← 0x0000
end
```

Compare Logical Greater Than Word 必須 v1.0

clgt rt,ra,rb



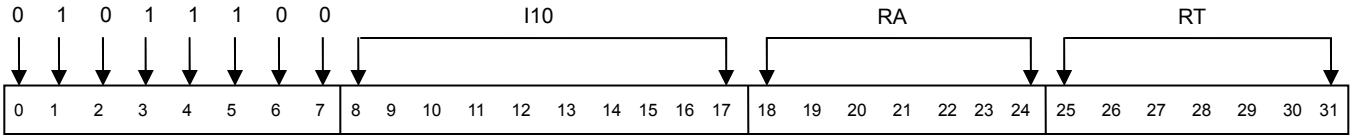
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドと論理的に比較する。レジスタ RA のオペランドがレジスタ RB のオペランドより論理的に大きい場合は、すべて 1 のビット（真）の結果が生成される。そうでない場合は、すべて 0 のビット（偽）の結果が生成される。
- 32 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15 by 4
        If RA^i::4 >^u RB^i::4 then RT^i::4 ← 0xFFFFFFFF
        else                        RT^i::4 ← 0x00000000
end
```

Compare Logical Greater Than Word Immediate 必須 v1.0

clgti **rt,ra,value**



ワード・スロット 4個それぞれに、下記を行なう。

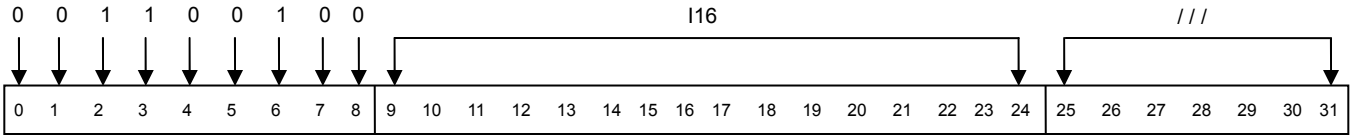
- I10 フィールドの値を、符号拡張によって 32 bit に拡張し、レジスタ RA の値と論理的に比較する。レジスタ RA の値が I10 フィールドの値より論理的に大きい場合は、すべて 1 のビット（真）の結果が生成される。そうでない場合は、すべて 0 のビット（偽）の結果が生成される。

- 32 bit の結果をレジスタ RT に置く。

```
for i = 0 to 15 by 4
        If RA^i::4 >^u RepLeftBit(I10,32) then RT^i::4 ← 0xFFFFFFFF
        else                                   RT^i::4 ← 0x00000000
end
```

Branch Relative 必須 v1.0

br **symbol**



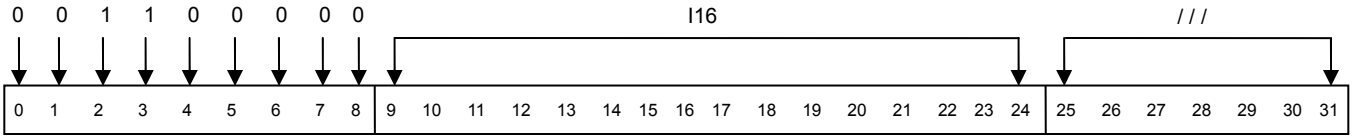
ターゲット命令に実行を移す。ターゲット命令のアドレスは、2 bit の 0 を右側に付け加えた I16 フィールドの値を、符号付き値として扱って、この相対分岐命令のアドレスに加算して算出される。

プログラミングの注意：I16 フィールドの値が 0 の場合は、1 命令無限ループが実行される。

PC	← (PC + RepLeftBit(I16 || 0b00,32)) & LSLR

Branch Absolute 必須 v1.0

bra **symbol**

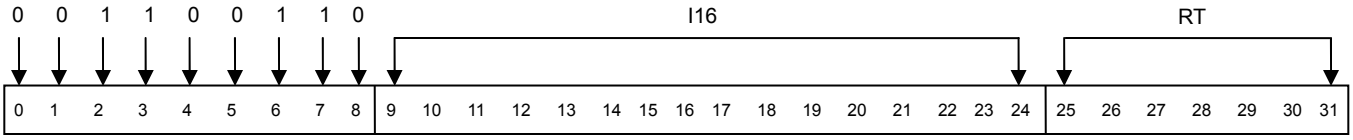


ターゲット命令に実行を移す。ターゲット命令のアドレスは、2 bit の 0 を右側に付け加えた I16 フィールドの値の最上位ビットをコピーして左に拡張したものである。

PC	← RepLeftBit(I16 || 0b00,32) & LSLR

Branch Relative and Set Link 必須 v1.0

brsl **rt,symbol**



ターゲット命令に実行を移す。さらに、リンク・レジスタを設定する。

2 bit の 0 を右側に付け加えた I16 フィールドの値を、符号付き値として扱って、この Branch Relative and Set Link 命令のアドレスに加算してターゲット命令のアドレスを算出する。

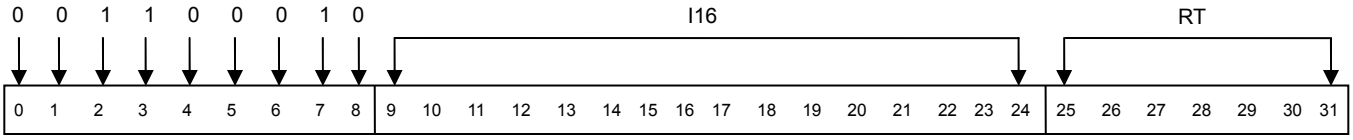
レジスタ RT のプリファード・スロットを、Branch Relative and Set Link 命令に続くバイトのアドレスに設定する。レジスタ RT の残りのスロットは、0 に設定する。

プログラミングの注意：I16 フィールドの値が 0 の場合は、1 命令無限ループが実行される。

$RT^{0:3}$	← (PC + 4) & LSLR
$RT^{4:15}$	← 0
PC	← (PC + RepLeftBit(I16 || 0b00,32)) & LSLR

Branch Absolute and Set Link 必須 v1.0

brasl **rt,symbol**



ターゲット命令に実行を移す。さらに、リンク・レジスタを設定する。

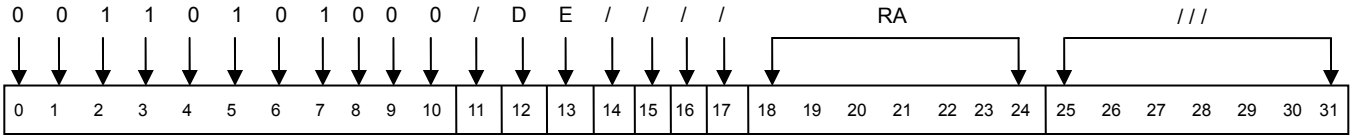
ターゲット命令のアドレスは、2 bit の 0 を右側に付け加えた I16 フィールドの値の最上位ビットをコピーして左に拡張したものである。

レジスタ RT のプリファード・スロットを、Branch Absolute and Set Link 命令に続くバイトのアドレスに設定する。レジスタ RT の残りのスロットは、0 に設定する。

$RT^{0:3}$	← (PC + 4) & LSLR
$RT^{4:15}$	← 0
PC	← RepLeftBit(I16 || 0b00,32) & LSLR

Branch Indirect 必須 v1.0

bi ra



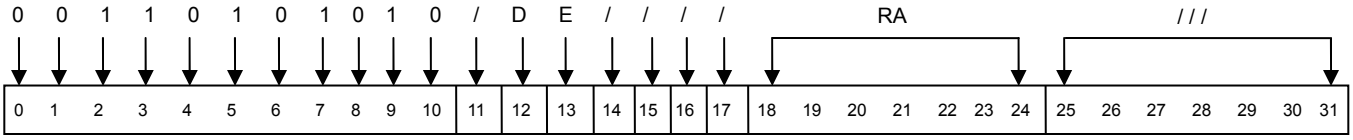
レジスタ RA のプリファード・スロットでアドレス指定された命令に実行を移す。レジスタ RA の値の右端 2 bit は無視され 0 とみなされる。E または D 機能ビットで、割り込みを許可、あるいは禁止にできる（251 ページの *セクション 12「SPU 割り込み機能」を参照*）。

PC ← $RA^{0:3}$ & LSLR & 0xFFFFFFFC

if (E = 0 and D = 0) then interrupt enable status is not modified
else if (E = 1 and D = 0) then enable interrupts at target
else if (E = 0 and D = 1) then disable interrupts at target
else if (E = 1 and D = 1) then reserved

Interrupt Return 必須 v1.0

iret **ra**



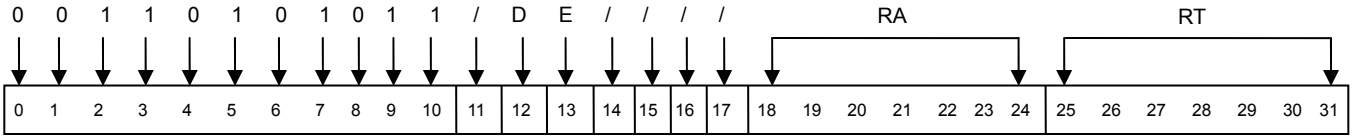
SRR0 でアドレス指定された命令に実行を移す。RA は有効なソースとみなされるが、その値は無視される。E または D 機能ビットで、割り込みを許可、あるいは禁止にできる（251 ページの *セクション 12「SPU 割り込み機能」*を参照）。

PC ← SRR0

if (E = 0 and D = 0) then interrupt enable status is not modified
else if (E = 1 and D = 0) then enable interrupts at target
else if (E = 0 and D = 1) then disable interrupts at target
else if (E = 1 and D = 1) then reserved

Branch Indirect and Set Link if External Data 必須 v1.0

bisled rt,ra



外部条件を調べる。偽の場合は、実行は直後の命令に継続する。真の場合は、次の命令の実効アドレスをレジスタ RA のプリファード・ワード・スロットから取得する。

bisled 命令に続く命令のアドレスを、レジスタ RT のプリファード・ワード・スロットに置く。レジスタ RT の残りは、0 に設定する。

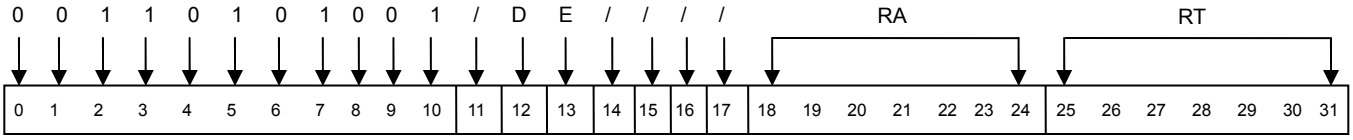
分岐が成立した場合は、E または D 機能ビットによって、割り込みを許可あるいは禁止にできる（251 ページの *セクション 12「SPU 割り込み機能」*を参照）。

```
u ← LSLR & (PC + 4)
t ← RA^0:3 & LSLR & 0xFFFFFFFC
RT^0:3 ← u
RT^4:15 ← 0

if (external event) then
        PC ← t
        if (E = 0 and D = 0) then interrupt enable status is not modified
        else if (E = 1 and D = 0) then enable interrupts at target
        else if (E = 0 and D = 1) then disable interrupts at target
        else if (E = 1 and D = 1) then reserved
else
        PC ← u
```

Branch Indirect and Set Link 必須 v1.0

bisl **rt,ra**



右端 2 bit を 0 とみなしたレジスタ RA のプリファード・ワード・スロットから、次の命令の実効アドレスを取得する。**bisl** 命令に続く命令のアドレスを、レジスタ RT のプリファード・ワード・スロットに置く。レジスタ RT の残りは、0 に設定する。E または D 機能ビットによって、割り込みを許可あるいは禁止にできる（251 ページの *セクション 12「SPU 割り込み機能」を参照*）。

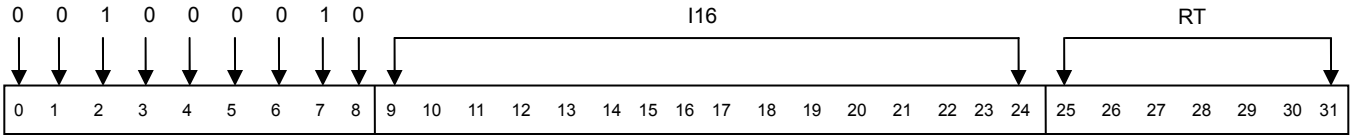
```
t ← RA^0:3 & LSLR & 0xFFFFFFFC
u ← LSLR & (PC + 4)
RT^0:3 ← u
RT^4:15 ← 0x00
PC ← t

if (E = 0 and D = 0) then interrupt enable status is not modified
else if (E = 1 and D = 0) then enable interrupts at target
else if (E = 0 and D = 1) then disable interrupts at target
else if (E = 1 and D = 1) then reserved
```

Branch If Not Zero Word

必須　v1.0

brnz　　rt,symbol



プリファード・スロットを調べる。0 でない場合は、分岐ターゲットに実行を移す。0 の場合は、次の命令へ続行する。

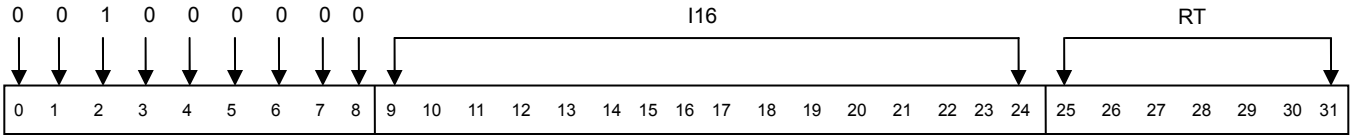
分岐ターゲットのアドレスは、2 bit の 0 を I16 フィールドの値に付加し、その値の最上位ビットをコピーして左に拡張し、さらに命令カウンタの値を加算することによって算出する。

```
If RT0:3 ≠ 0 then
        PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
else
        PC ← (PC + 4) & LSLR
```

Branch If Zero Word

必須　v1.0

brz **rt,symbol**



プリファード・スロットを調べる。0 の場合は、分岐ターゲットに実行を移す。0 でない場合は、次の命令へ続行する。

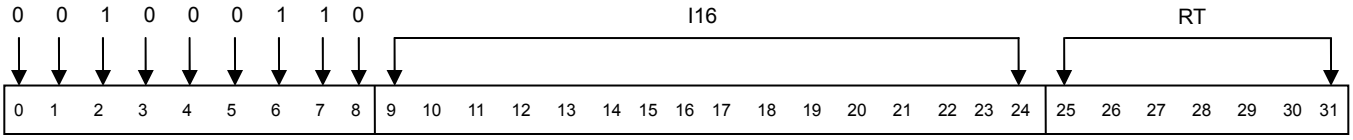
分岐ターゲットのアドレスは、2 bit の 0 を I16 フィールドの値に付加し、その値の最上位ビットをコピーして左に拡張し、さらに命令カウンタの値を加算することによって算出する。

```
If RT0:3 = 0 then
        PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
else
        PC ← (PC + 4) & LSLR
```

Branch If Not Zero Halfword

必須　v1.0

brhnz　　rt,symbol



プリファード・スロットを調べる。右端のハーフワードが 0 でない場合は、分岐ターゲットに実行を移す。0 の場合は、次の命令へ続行する。

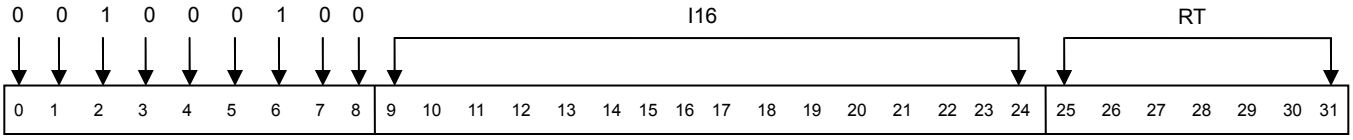
分岐ターゲットのアドレスは、2 bit の 0 を I16 フィールドの値に付加し、その値の最上位ビットをコピーして左に拡張し、さらに命令カウンタの値を加算することによって算出する。

```
If RT[2:3] ≠ 0 then
        PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
else
        PC ← (PC + 4) & LSLR
```

Branch If Zero Halfword

必須　v1.0

brhz **rt,symbol**



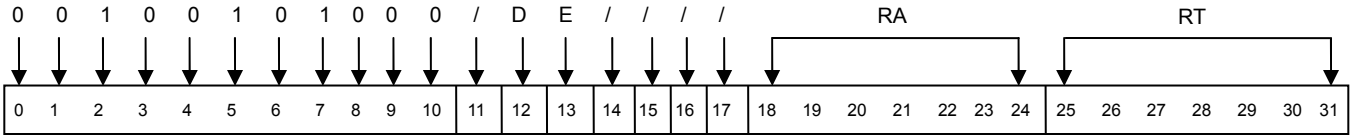
プリファード・スロットを調べる。右端のハーフワードが 0 の場合は、分岐ターゲットに実行を移す。0 でない場合は、次の命令へ続行する。

分岐ターゲットのアドレスは、2 bit の 0 を I16 フィールドの値に付加し、その値の最上位ビットをコピーして左に拡張し、さらに命令カウンタの値を加算することによって算出する。

```
If RT2:3 = 0 then
        PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
else
        PC ← (PC + 4) & LSLR
```

Branch Indirect If Zero 必須 v1.0

biz rt,ra



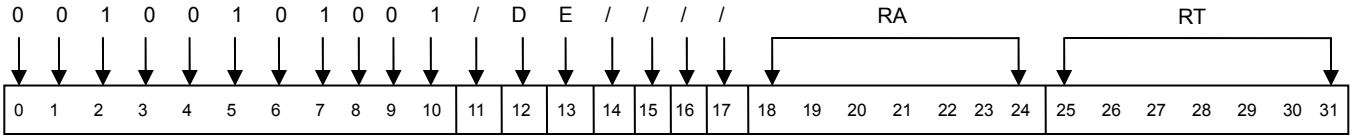
レジスタ RT のプリファード・スロットが 0 でない場合は、実行は直後の命令に継続する。0 の場合は、レジスタ RA のプリファード・スロットで指定されたアドレスへ分岐する。このとき右端 2 bit は 0 として扱われる。分岐する場合は、E または D 機能ビットによって、割り込みを許可あるいは禁止にできる（251 ページの *セクション 12「SPU 割り込み機能」*を参照）。

```
t ← RA0:3 & LSLR & 0xFFFFFFFC
u ← LSLR & (PC + 4)

If RT0:3 = 0 then
        PC ← t & LSLR & 0xFFFFFFFC
        if (E = 0 and D = 0) then interrupt enable status is not modified
        else if (E = 1 and D = 0) then enable interrupts at target
        else if (E = 0 and D = 1) then disable interrupts at target
        else if (E = 1 and D = 1) then reserved
else
        PC ← u
```

Branch Indirect If Not Zero 必須 v1.0

binz rt,ra



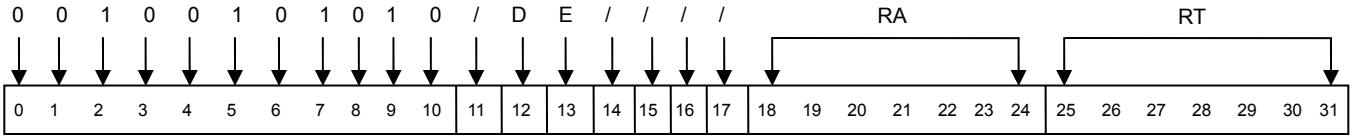
レジスタ RT のプリファード・スロットが 0 の場合は、実行は直後の命令に継続する。0 でない場合は、レジスタ RA のプリファード・スロットで指定されたアドレスへ分岐する。このとき右端 2 bit は 0 として扱われる。分岐する場合は、E または D 機能ビットによって、割り込みを許可あるいは禁止にできる（251 ページの *セクション 12「SPU 割り込み機能」*を参照）。

```
t ← RA0:3 & LSLR & 0xFFFFFFFC
u ← LSLR & (PC + 4)

If RT0:3 ! = 0 then
        PC ← t & LSLR & 0xFFFFFFFC
        if (E = 0 and D = 0) then interrupt enable status is not modified
        else if (E = 1 and D = 0) then enable interrupts at target
        else if (E = 0 and D = 1) then disable interrupts at target
        else if (E = 1 and D = 1) then reserved
else
        PC ← u
```

Branch Indirect If Zero Halfword 必須 v1.0

bihz rt,ra



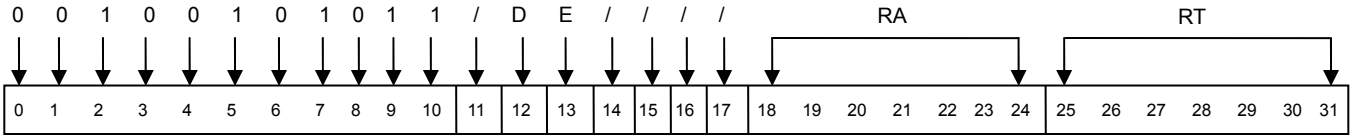
レジスタ RT のプリファード・スロットにある右端のハーフワードが 0 でない場合は、実行は直後の命令に継続する。0 の場合は、レジスタ RA のプリファード・スロットで指定されたアドレスへ分岐する。このとき右端 2 bit は 0 として扱われる。分岐する場合は、E または D 機能ビットによって、割り込みを許可あるいは禁止にできる（251 ページの *セクション 12「SPU 割り込み機能」*を参照）。

```
t ← RA0:3 & LSLR & 0xFFFFFFFC
u ← LSLR & (PC + 4)

If RT2:3 = 0 then do
        PC ← t & LSLR & 0xFFFFFFFC
        if (E = 0 and D = 0) then interrupt enable status is not modified
        else if (E = 1 and D = 0) then enable interrupts at target
        else if (E = 0 and D = 1) then disable interrupts at target
        else if (E = 1 and D = 1) then reserved
else
        PC ← u
```

Branch Indirect If Not Zero Halfword 必須 v1.0

bihnz rt,ra



レジスタ RT のプリファード・スロットにある右端のハーフワードが 0 の場合は、実行は直後の命令に継続する。0 でない場合は、レジスタ RA のプリファード・スロットで指定されたアドレスへ分岐する。このとき右端 2 bit は 0 として扱われる。分岐する場合は、E または D 機能ビットによって、割り込みを許可あるいは禁止にできる（251 ページの *セクション 12「SPU 割り込み機能」*を参照）。

```
t ← RA0:3 & LSLR & 0xFFFFFFFC
u ← LSLR & (PC + 4)

If RT2:3 ! = 0 then
        PC ← t & LSLR & 0xFFFFFFFC
        if (E = 0 and D = 0) then interrupt enable status is not modified
        else if (E = 1 and D = 0) then enable interrupts at target
        else if (E = 0 and D = 1) then disable interrupts at target
        else if (E = 1 and D = 1) then reserved
else
        PC ← u
```

8. 分岐ヒント命令

本セクションでは、SPU の分岐ヒント命令をリストアップし、説明する。

SPU の分岐ヒント命令は、セマンティクスを持たない。これらの命令は、実装に対して将来の分岐命令に関するヒントを与えるものであり、その情報が、分岐ターゲットのプリフェッチあるいはその他の方法による性能向上のために用いられることを意図したものである。

各分岐ヒント命令は、分岐命令のアドレスおよび予期される分岐ターゲットのアドレスを指定する。分岐が成立しないと予期する場合は、ターゲット・アドレスは分岐命令に続く命令のアドレスである。

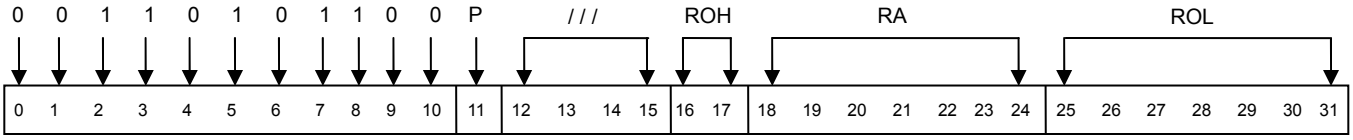
本セクションで説明する命令は、下記の様に定義する **brinst** および **brtarg** 変数を使用する。

- **brinst** = RO
- **brtarg** = I16

Hint for Branch (r-form)

必須　v1.0

hbr　　**brinst,brtarg**



分岐ターゲットのアドレスは、レジスタ RA のプリファード・スロットの値で与える。RO フィールドは、この **hbr** 命令から分岐命令までの符号付きのワードオフセットを与える。

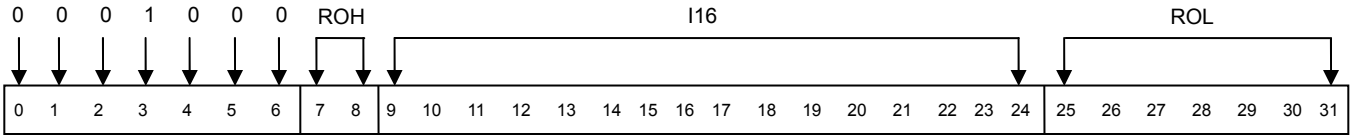
P 機能ビットがセットされている場合は、**hbr** は分岐のヒントを与えるのではない。その代わりに、ここが、インライン・プリフェッチを行なうための実装固有の適切な時期であるというヒントを与える。
インライン・プリフェッチとは、まっすぐにシーケンシャルなプログラムテキストを実行するために必要な命令フェッチ機能である。最適な性能を得るためには、SPU の一部のインプリメンテーションにおいては、プログラムがロードやストアも行なっている場合に、ローカル・ストレージのこれらのインライン・プリフェッチのスケジューリング支援を必要とすることがある。これがどのような時に有益かについては、インプリメンテーション固有の SPU のドキュメントを参照のこと。
P 機能ビットをセットする場合は、ROH（上位）と ROL（下位）を連結することによって形成される RO フィールドを 0 に設定しなければならない。

分岐ターゲット・アドレス ← $RA^{0:3}$ & LSLR & 0xFFFFFFFC
分岐命令アドレス ← (RepLeftBit(ROH || ROL || 0b00,32) + PC) & LSLR

Hint for Branch (a-form)

必須　v1.0

hbra　　**brinst,brtarg**



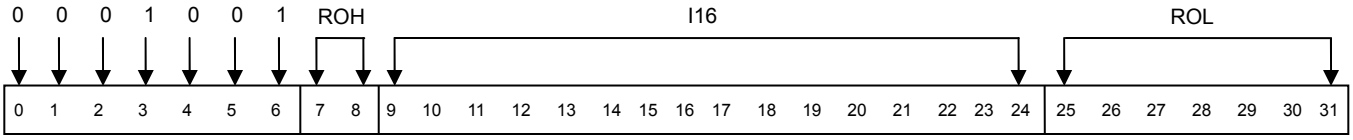
I16 フィールドのアドレスによって、分岐ターゲットのアドレスを指定する。値には、右に 0 を 2 bit 付加してから使用する。

ROH（上位）と ROL（下位）を連結することによって形成される RO フィールドは、この **hbra** 命令から分岐命令までの符号付きのワードオフセットを与える。

```
分岐ターゲット・アドレス← RepLeftBit(I16 || 0b00,32) & LSLR
分岐命令アドレス ← (RepLeftBit(ROH || ROL || 0b00,32) + PC) & LSLR
```

Hint for Branch Relative 必須 v1.0

hbrr **brinst,brtarg**



I16 フィールドで与えられたワードオフセットによって、分岐ターゲットのアドレスを指定する。符号付きの I16 フィールドをこの **hbrr** 命令のアドレスに加算し、分岐ターゲットの絶対アドレスを決定する。

ROH（上位）と ROL（下位）を連結することによって形成される RO フィールドは、この **hbrr** 命令から分岐命令までの符号付きのワードオフセットを与える。

分岐ターゲット・アドレス← (RepLeftBit(I16 || 0b00,32) + PC) & LSLR
分岐命令アドレス← (RepLeftBit(ROH || ROL || 0b00,32) + PC) & LSLR

9. 浮動小数点命令

本セクションでは、SPU 浮動小数点命令を説明する。本セクションでは、SPU の浮動小数点計算と IEEE 規格の浮動小数点計算の違いについても説明する。
単精度浮動小数点命令は *IEEE 754 規格*に準拠した結果の算出を行わないが、SPU で使用する単精度および倍精度浮動小数点数値のデータ・フォーマットは、*IEEE 754* 規格と同一である。

実装上の注記：本アーキテクチャでは、実装によって浮動小数点命令に対して異なる結果を生じることを許容している。ある実装で生成される結果の情報に関しては、実装固有のドキュメントを参照されたい。実装間で同一の結果を達成するには、アーキテクチャに準拠する以上のことが必要となる。

9.1 単精度（拡張範囲モード）

単精度浮動小数点演算については、正規化数の範囲が拡張されている。しかし、規格で定義されているフルレンジは実装されていない。SPU で表現および演算可能な非ゼロの数値の範囲は、*表 9-1* に示す最小値と最大値の間である。*表 9-1* では、レジスタの値から10進数への変換についても示している。

表 9-1 単精度（拡張範囲モード）の最小値および最大値

数値フォーマット	最小の正の値（Smin）			最大の正の値（Smax）			注記
レジスタ数値	0x00800000			0x7FFFFFFF			
ビットフィールド	符号	8-bit バイアス付き指数	仮数（暗黙の[1]と 23 bit）	符号	8-bit バイアス付き指数	仮数（暗黙の[1]と 23 bit）	1
	0	00000001	[1.]000...000	0	11111111	[1.]111...111	
2のべき乗での値	+	$2^{(1 - 127)}$	1	+	$2^{(255 - 127)}$	$2 - 2^{-23}$	2
指数と仮数の組み合わせ	$2^{-126} * (+1)$			$2^{128} * (+[2 - 2^{-23}])$			
レジスタの値の10進表現	$1.2 * 10^{-38}$			$6.8 * 10^{38}$			

注記:
1. 指数フィールドは +127 だけバイアスされている。
2. 数値 $2 - 2^{-23}$ は 2 よりも 1 LSb (least significant bit) だけ小さい。

ゼロには、2 通りの表現形式がある。

- 正のゼロでは、すべてのビットが 0 になる。つまり、符号、指数部、および仮数部が 0 である。
- 負のゼロでは、符号が 1 で、指数部と仮数部は 0 である。

入力では両方のゼロをサポートするが、結果のゼロは常に正のゼロになる。

SPU の単精度の演算には、以下の特徴がある。

- 非数（NaN）を、オペランドとしてサポートせず、結果としても生成しない。
- 無限大（Inf）をサポートしない。対象となる浮動小数点フォーマットで表現できる最大値より大きい絶対値を生成する演算では、代わりに適切な符号、最大のバイアス付き指数、およびすべて 1（2 進数）の絶対値が生成される。SPU においては、Inf の表現形式 (IEEE 規格に準拠) は、SPU 上で使用される最大値よりも小さい数値として解釈されることに注意することが大切である。

- 非正規化数をサポートせず、0 として扱う。よって、IEEE の規則ならば非正規化数を生成する演算では、代わりに正のゼロが生成される。オペランドとして非正規化数を使用すると、0 として扱われる。
- 唯一サポートする丸めモードは、切り捨て（ゼロ方向）である。

単精度の拡張範囲の算術については、オーバーフロー、アンダーフロー、ゼロ除算、および IEEE 非準拠結果の 4 種類の例外条件がテストされる。

- オーバーフロー（OVF）
丸めの前の結果の絶対値が表現可能な最大の正の数値の Smax より大きい場合、オーバーフロー例外が発生する。スライス *k* の演算によりオーバーフローが発生した場合、Floating-Point Status and Control Register（FPSCR）のスライス *k* に対する OVF フラグがセットされ、結果は適切な符号付きで Smax に飽和する。
- アンダーフロー（UNF）
丸めの前の結果の絶対値が表現可能な最小の正の数値の Smin より小さい場合、アンダーフロー例外が発生する。スライス *k* の演算によりアンダーフローが発生した場合、レジスタ FPSCR のスライス *k* に対する UNF フラグがセットされ、結果は正のゼロに飽和する。
- ゼロ除算（Divide by Zero）（DBZ）
estimate 命令の入力にゼロ指数があると、ゼロ除算例外が発生する。スライス *k* の演算によりゼロ除算が発生した場合、レジスタ FPSCR のスライス *k* に対する DBZ フラグがセットされる。
- IEEE 非準拠結果（DIFF）
「IEEE と異なる」（different-from-IEEE）例外は、拡張範囲の算術で生成された結果が IEEE の結果と違っているかもしれないことを示す。この例外は、下記のうちいずれかの条件に当てはまるときに発生する。
  - いずれかの入力または結果が、最大指数を持つ。（IEEE の算術では、このようなオペランドは NaN または無限大として扱われる。拡張範囲の算術では、正規化された値として扱われる。）
  - いずれかの入力が、ゼロ指数かつ非ゼロ仮数を持つ。（IEEE の算術では、このようなオペランドは非正規化数として扱われる。拡張範囲の算術では、ゼロとして扱われる。）
  - アンダーフローが発生する。すなわち、丸めの前の結果がゼロではなく、丸めの後の結果がゼロである。

  スライス *k* の演算で上記の状況が発生した場合、レジスタ FPSCR のスライス *k* に対する DIFF フラグがセットされる。

これらの例外は、拡張範囲の浮動小数点命令によってのみセットされうる。*表 9-2* に、例外をセットしうる命令を示す。

表 9-2 命令と例外設定

命令	OVF 設定	UNF 設定	DBZ 設定	DIFF 設定
fa, fs, fm, fma, fms, fnms, fi	○	○	×	○
frest, frsqest	×	×	○	×
csflt, cuflt	○	○	×	○
cflts, cfltu, fceq, fcneq, fcgt, fcmgt	×	×	×	×

9.2 倍精度

SPU の倍精度命令は、128 ビット値を 2 個の SIMD 倍精度演算として処理する。SIMD スライス 0 はダブルワード 0 を処理し、スライス 1 はダブルワード 1 を処理する。倍精度の場合、通常の IEEE セマンティクスおよび定義が適用される。このフォーマットでサポートされる非ゼロの数値は、*表 9-3* に示す最小値と最大値の間である。*表 9-3* では、レジスタの値から10進数への変換についても示している。

表 9-3 倍精度（IEEE モード）の最小値および最大値

数値フォーマット	最小の正の非正規化値（Dmin）			最大の正の正規化値（Dmax）			注記
レジスタ数値	0x0000000000000001			0x7FEFFFFFFFFFFFFF			
ビットフィールド	符号	11-bit バイアス付き指数	仮数（非正規化数では暗黙の[0] と 52 bit）	符号	11-bit バイアス付き指数	仮数（正規化数では暗黙の[1] と 52 bit）	1
	0	00000000000	[0.]000...001	0	11111111110	[1.]111...111	2
2 のべき乗での値	+	$2^{(0 + 1 - 1023)}$	2^{-52}	+	$2^{(2046 - 1023)}$	$2 - 2^{-52}$	3,4
指数と仮数の組み合わせ	$2^{-1022} * (+2^{-52})$			$2^{1023} * (+[2 - 2^{-52}])$			
レジスタの値の10進表現	$4.9 * 10^{-324}$			$1.8 * 10^{308}$			

注記:
1. 指数は +1023 だけバイアスされている。
2. 全てが 1 の指数フィールドは非数（NaN）および無限大（Inf）に予約されている。
3. 数値 $2 - 2^{-52}$ は 2 よりも 1 LSb だけ小さい。
4. 非正規化数については、指数にさらに 1 が加算される。

SPUの倍精度の演算には、以下の特徴がある。

- ハードウェアでは、IEEE 規格で要求される演算のサブセットのみをサポートする。
- 4 つの丸めモードすべてをサポートする。
- 2 個のスライスの丸めモードは個別に制御可能である。FPSCR の RN0 フィールド（ビット 20-21）がスライス 0 の現在の丸めモードを指定し、FPSCR の RN1 フィールド（ビット 22-23）がスライス 1 の現在の丸めモードを指定する。
- IEEE 例外は、検出されて FPSCR レジスタに累積される。トラップ処理はサポートしていない。
- IEEE 規格には、2 種類の NaN がある。NaN は、バイアス付きの最大指数、および非ゼロの仮数を含んだ値である。符号ビットは無視される。仮数部フィールドの上位ビットが 0b0 である場合は、NaN は Signaling NaN（SNaN）である。そうでなければ、Quiet NaN（QNaN）である。QNaN が NaN 入力を有しない浮動小数演算の結果である場合、その結果は常にデフォルトの QNaN である。すなわち、仮数部フィールドの上位ビットが 1 で、仮数部のその他すべてのビットが 0 であり、さらに符号ビットが 0 である。
- IEEE 規格では、NaN の伝播に関して厳密な規則を持つ。QNaN が、少なくとも 1 個の NaN 入力を有する浮動小数演算の結果である場合、SPU の実装では、デフォルトの QNaN を生じるか、あるいは、入力の NaN 値のうちのひとつを生じるか、いずれかの可能性がある。ある実装が、NaN、QNaN、SNaN の適切な入力を伝播する代わりに QNaN 結果を生成する場合は、非準拠結果の可能性をシグナル通知するために、FPSCR の NaN フラグがセットされる。

- 一部の実装にて、非正規化数は、結果としてのみサポートする場合がある。そのような実装では、非正規化数オペランドは、0 として扱われる（これは、IEEE フラグの設定にも適用される）。オペランドの符号は、保持される。非正規化数のオペランドが強制的にゼロにされる場合は常に、非準拠結果の可能性をシグナル通知するために、FPSCR の DENORM フラグがセットされる。

9.2.1 単精度フォーマットと倍精度フォーマットの間の変換

変換には、倍精度の値を単精度の値に丸めるもの（**frds**）と、単精度の値を倍精度に拡張するもの（**fesd**）の 2 種類がある。両方の操作とも、非正規化数の入力の処理を除いて、IEEE 規格に準拠する。一部の実装では、非正規化数がゼロに強制される場合がある。ある実装が、非正規化入力をゼロに強制する場合には、FPSCR 中のアンダーフロー・フラグではなくて DENORM フラグをセットする。このようにして、これら 2 つの操作においては、NaN、無限大、および非正規化数の結果は、倍精度だけでなく単精度でもサポートしている。非ゼロの IEEE 単精度浮動小数点数の範囲は*表 9-4* に示す最小値と最大値の間である。*表 9-4* では、レジスタの値から10進数への変換についても示している。

表 9-4 単精度（IEEE モード）の最小値および最大値

数値フォーマット	最小の正の非正規化値（Smin）			最大の正の値（Smax）			注記
レジスタ数値	0x00000001			0x7F7FFFFF			
ビットフィールド	符号	8-bit バイアス付き指数	仮数（暗黙の[0] と 23 bit）	符号	8-bit バイアス付き指数	仮数（暗黙の[1] と 23 bit）	1
	0	00000000	[0.]000..001	0	11111110	[1.]111...111	
2のべき乗での値	+	$2^{(0+1-127)}$	2^{-23}	+	$2^{(254-127)}$	$2 - 2^{-23}$	2
指数と仮数の組み合わせ	$2^{-126} * 2^{-23}$			$2^{127} * (2 - 2^{-23})$			
レジスタの値の10進表現	$1.4 * 10^{-45}$			$3.4 * 10^{38}$			

注記:
1. 指数は +127 だけバイアスされている。
2. 数値 $2 - 2^{-23}$ は 2 よりも 1 LSb (least significant bit) だけ小さい。

9.2.2 例外条件

本アーキテクチャでは、トラップによらない例外処理のみをサポートする。すなわち、例外条件を検出し、FPSCR の適切なフィールドにレポートする。これらのフラグは、スティッキーであり、いったんセットされると、FPSCR 書き込み命令によってクリアされるまでセットされたままである。これらの例外フラグは、拡張範囲で実行される単精度演算ではセットされない。倍精度演算は、2 way の SIMD であることから、2 セットの例外フラグがある。

不正確な結果（Inexact Result ― INX）
計算された結果の値が、指数部の範囲と精度の両方を無制限とした場合に算出されるであろうものと異なる場合に、不正確な結果が検出される。

オーバーフロー（Overflow ― OVF）
指数部の範囲を無制限として計算した場合に得られるであろう丸め結果の絶対値が、指定された結果の精度の最大有限値を超えたときにオーバーフローが発生する。

アンダーフロー（Underflow ー UNF）
トラップによらない例外処理では、IEEE 754 規格は、下記のようにアンダーフローを定義している。

UNF = tiny AND loss_of_accuracy

ここで、tiny と loss of accuracy にはそれぞれ 2 つの定義があり、実装には 4 通りのどの組み合わせを選択してもよい。本アーキテクチャでは *tiny-before-rounding* かつ *inexact result (INX)* を実装している。したがって、下記の定義になる。

UNF = tiny_before_rounding AND inexact_result

注意：指数部の範囲を無制限として計算した場合の非ゼロの結果値が、最小の正規化数より絶対値で小さくなるときに tiny before rounding が検出される。

無効演算（Invalid Operation ー INV）
指定された演算に対して、オペランドが無効である場合は、常に無効演算例外が発生する。ハードウェアで実装される演算については、下記の演算において無効演算例外条件が発生する。

- Signaling NaN（SNaN）での浮動小数点演算すべて。
- 加算、減算、および積和演算での無限大の絶対値減算。すなわち、無限大－無限大。
- 無限大とゼロの乗算。

注意：一部の実装においては、非正規化数入力はゼロとして扱い、DENORM フラグと無効演算フラグの両方をセットする場合がある。

伝播しない NaN（Not Propagated NaN ー NaN）
IEEE 規格では、NaN 入力に特別な処理が要求されるが、SPU の実装では、倍精度浮動小数点演算の結果としてデフォルトの QNaN を返す場合がある。少なくとも入力の 1 つが NaN の場合、結果として生じた QNaN は、IEEE 規格完全準拠の設計から渡される結果とは異なっていることがありうる。このことは NaN フィールドのフラグで示される。

非正規化数入力のゼロ強制（Denormal Input Forced to Zero ー DENORM）
SPU の実装では、倍精度演算を行なう前に、ある種の倍精度非正規化数オペランドを強制的にゼロにする場合がある。ある実装においてこれらのオペランドをゼロ強制した場合には、そのゼロには元の非正規化数値の符号が保持される。非正規化数入力をゼロ強制した際には、結果が IEEE 準拠の結果とは異なるかもしれないことをシグナル通知するために、FPSCR に DENORM 例外フラグがセットされる。

プログラミングの注意：IEEE 準拠の倍精度結果が必要なアプリケーションでは、FPSCR の NaN および DENORM フラグを用いて非準拠結果を検出することが出来る。これにより、効率は低いが規格準拠の方式でコードを再実行することが可能である。両方のフラグともにスティッキーであるため、大きなコードブロックをガードすることが可能であり、コードチェックのオーバーヘッドを最小にとどめられる。たとえば、以下のようにする。

FPSCRをクリア
高速コードブロック
if (NaN || DENORM)

{

規格準拠コードブロック

}

CEBA 準拠のプロセッサに内蔵された SPU の場合には、SPU が実行を停止して PPE にシグナルを送り、PPE 側で計算を行なってから SPU を動作再開させるように依頼することも可能である。

表 9-5 に、例外がセットされうる命令をリストアップする。

表 9-5 命令および例外設定

命令	OVF 設定	UNF 設定	INX 設定	INV 設定	NaN 設定	DENORM 設定
dfa, dfs, dfm, dfma, dfms, dfnms, dfnma	○	○	○	○	○	○
fesd	×	×	×	○	○	○
frds	○	○	○	○	○	○

9.3 浮動小数点状態および制御レジスタ（Floating-Point Status and Control Register－FPSCR）

浮動小数点状態および制御レジスタ（Floating-Point Status and Control Register－FPSCR）は、浮動小数点演算がもたらす状態を記録し、倍精度浮動小数点演算の丸めモードを制御する。FPSCR は、FPSCR Read 命令（**fscrrd**）で読み出しされ、FPSCR Write 命令（**fscrwr**）で書き込みされる。ビットの [20:23] は、制御ビットであり、残りのビットは、状態ビット、あるいは未使用のいずれかである。FPSCR にあるすべての状態ビットは、スティッキーである。つまり、一度セットされると、**fscrwr** 命令によってクリアされるまでセットされたままの状態になる。

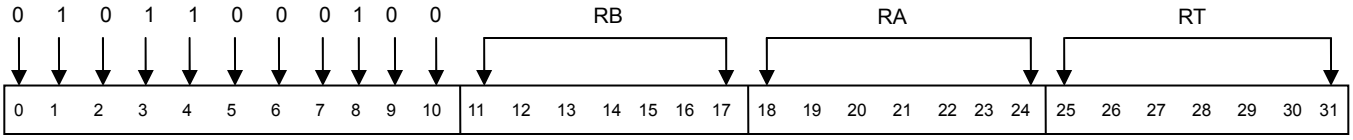
FPSCR のフォーマットは、下記のとおりである。

ビット	説明
0:19	未使用
20:21	2 way SIMD 倍精度演算のスライス 0 に対する丸め制御 (RN0) 00　最も近い偶数への丸め 01　ゼロ方向への丸め（切り捨て） 10　正の無限大方向への丸め 11　負の無限大方向への丸め
22:23	2 way SIMD 倍精度演算のスライス 1 に対する丸め制御 (RN1) 00　最も近い偶数への丸め 01　ゼロ方向への丸め（切り捨て） 10　正の無限大方向への丸め 11　負の無限大方向への丸め
24:28	未使用

29:31	スライス 0 に対する単精度例外フラグ 29　　オーバーフロー（OVF） 30　　アンダーフロー（UNF） 31　　拡張範囲演算で生成された結果は、IEEE 準拠の結果とは異なっているかもしれない。（DIFF）
32:49	未使用
50:55	2 way SIMD 倍精度演算のスライス 0 に対する IEEE 例外フラグ 50　　オーバーフロー（OVF） 51　　アンダーフロー（UNF） 52　　不正確な結果（INX） 53　　無効演算（INV） 54　　QNaN 伝播による非準拠の結果の可能性（NaN） 55　　非正規化数オペランドによる非準拠の結果の可能性（DENORM）
56:60	未使用
61:63	スライス 1 に対する単精度例外フラグ（OVF、UNF、DIFF）
64:81	未使用
82:87	2 way SIMD 倍精度演算のスライス 1 に対する IEEE 例外フラグ（OVF、UNF、INX、INV、NaN、DENORM）
88:92	未使用
93:95	スライス 2 に対する単精度例外フラグ（OVF、UNF、DIFF）
96:115	未使用
116:119	各 4 スライスに対する単精度ゼロ除算フラグ 116　　スライス 0 に対する DBZ 117　　スライス 1 に対する DBZ 118　　スライス 2 に対する DBZ 119　　スライス 3 に対する DBZ
120:124	未使用
125:127	スライス 3 に対する単精度例外フラグ（OVF、UNF、DIFF）

Floating Add 必須 v1.0

fa rt,ra,rb



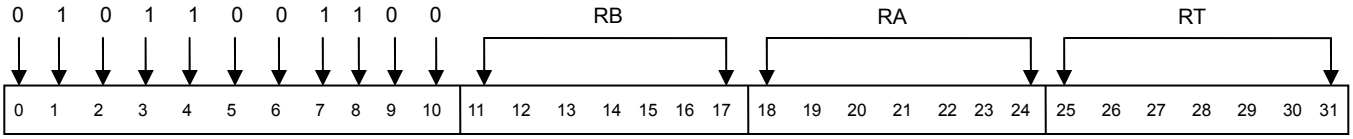
ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドに加算する。
- 結果をレジスタ RT に置く。
- 結果の絶対値が Smax より大きい場合は、（正しい符号が付いた）Smax が結果として生成される。Smin より小さい場合は、0 が生成される。

Double Floating Add

必須　v1.0

dfa　　rt,ra,rb

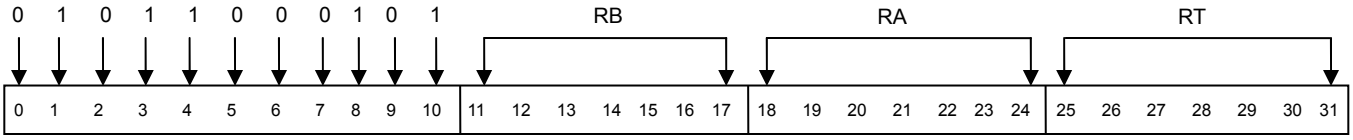


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドに加算する。
- 結果をレジスタ RT に置く。

Floating Subtract 必須 v1.0

fs **rt,ra,rb**

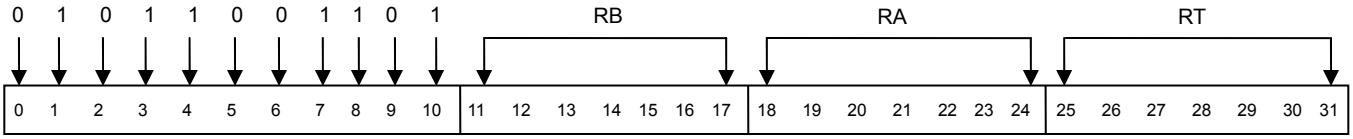


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RB のオペランドを、レジスタ RA のオペランドから減算する。
- 結果をレジスタ RT に置く。
- 結果の絶対値が Smax より大きい場合は、（正しい符号が付いた）Smax が結果として生成される。Smin より小さい場合は、0 が生成される。

Double Floating Subtract 必須 v1.0

dfs **rt,ra,rb**

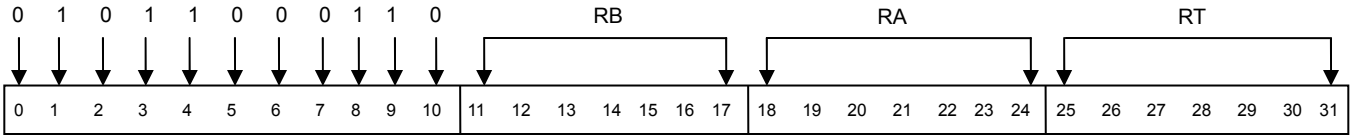


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RB のオペランドを、レジスタ RA のオペランドから減算する。
- 結果をレジスタ RT に置く。

Floating Multiply 必須 v1.0

fm **rt,ra,rb**

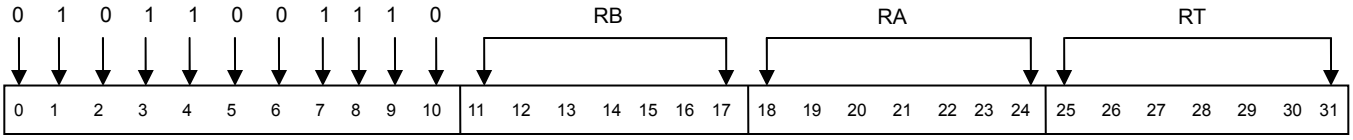


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドで乗算する。
- 結果をレジスタ RT に置く。
- 結果の絶対値が Smax より大きい場合は、（正しい符号が付いた）Smax が結果として生成される。Smin より小さい場合は、0 が生成される。

Double Floating Multiply 必須 v1.0

dfm **rt,ra,rb**

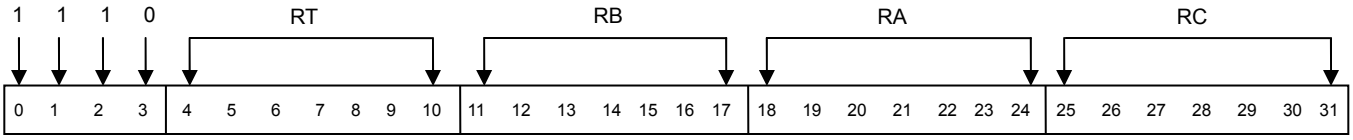


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドで乗算する。
- 結果をレジスタ RT に置く。

Floating Multiply and Add 必須 v1.0

fma **rt,ra,rb,rc**

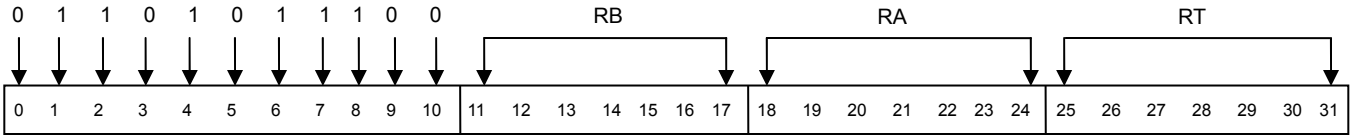


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドをレジスタ RB のオペランドで乗算し、レジスタ RC のオペランドに加算する。中間の乗算は正確（exact）であり、その範囲に制限を受けない。
- 結果をレジスタ RT に置く。
- 加算結果の絶対値が Smax より大きい場合は、（正しい符号が付いた）Smax が結果として生成される。Smin より小さい場合は、0 が生成される。

Double Floating Multiply and Add 必須 v1.0

dfma **rt,ra,rb**

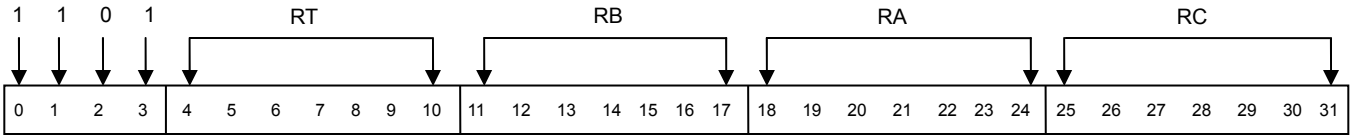


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RA のオペランドをレジスタ RB のオペランドで乗算し、レジスタ RT のオペランドに加算する。中間の乗算は正確（exact）であり、その範囲に制限を受けない。
- 結果をレジスタ RT に置く。

Floating Negative Multiply and Subtract 必須 v1.0

fnms **rt,ra,rb,rc**

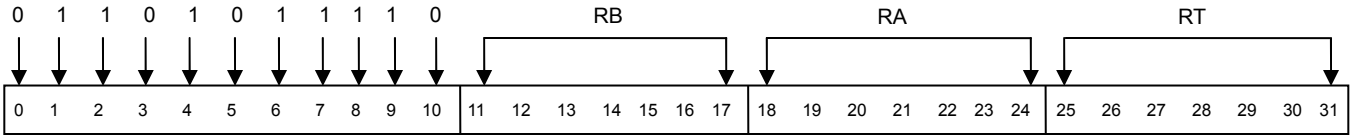


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドをレジスタ RB のオペランドで乗算し、その結果をレジスタ RC のオペランドから減算する。中間の乗算は正確（exact）であり、その範囲に制限を受けない。
- 結果をレジスタ RT に置く。
- 減算結果の絶対値が Smax より大きい場合は、（正しい符号が付いた）Smax が結果として生成される。Smin より小さい場合は、0 が生成される。

Double Floating Negative Multiply and Subtract 必須 v1.0

dfnms **rt,ra,rb**

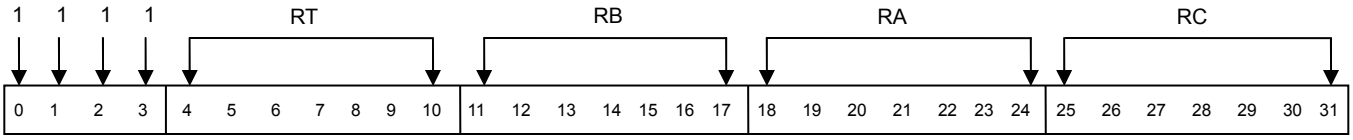


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドで乗算する。その積からレジスタ RT のオペランドを減算する。通常、この積差演算の丸め結果を符号反転することによって結果を取得し、レジスタ RT に置く。一つだけ例外として、結果が QNaN の場合は、結果の符号ビットは 0 である。
- この命令は、Double Floating Multiply and Subtract 命令を使用してから NaN 以外の結果を符号反転することによって得られる結果と同一の結果を生成する。
- 中間の乗算は正確（exact）であり、その範囲に制限を受けない。

Floating Multiply and Subtract 必須 v1.0

fms **rt,rb,ra,rc**

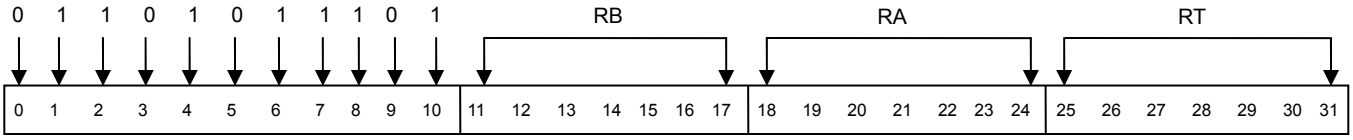


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドで乗算する。中間の乗算は正確（exact）であり、その範囲に制限を受けない。その積からレジスタ RC のオペランドを減算する。
- 結果をレジスタ RT に置く。
- 減算結果の絶対値が Smax より大きい場合は、（正しい符号が付いた）Smax が結果として生成される。Smin より小さい場合は、0 が生成される。

Double Floating Multiply and Subtract 必須 v1.0

dfms **rt,ra,rb**

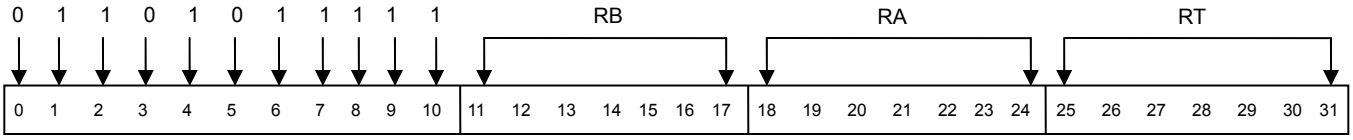


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RA のオペランドを、レジスタ RB のオペランドで乗算する。中間の乗算は正確（exact）であり、その範囲に制限を受けない。その積からレジスタ RT のオペランドを減算する。
- 結果をレジスタ RT に置く。

Double Floating Negative Multiply and Add 必須 v1.0

dfnma **rt,ra,rb**

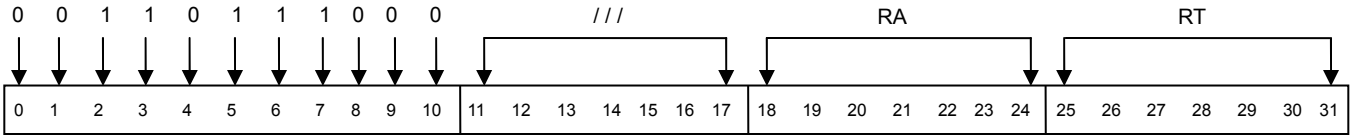


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RA のオペランドをレジスタ RB のオペランドで乗算し、レジスタ RT のオペランドに加算する。中間の乗算は正確（exact）であり、その範囲に制限を受けない。通常、この積和演算の丸め結果を符号反転することによって結果を取得し、レジスタ RT に置く。一つだけ例外として、結果が QNaN の場合は、結果の符号ビットは 0 である。
- この命令は、Double Floating Multiply and Add 命令を使用し、その後 NaN 以外の結果を符号反転することによって得る結果と同一の結果を生成する。

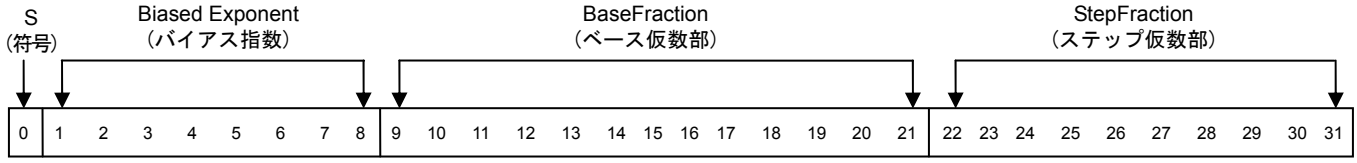
Floating Reciprocal Estimate 必須 v1.0

frest rt,ra



ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを使用して、オペランドの逆数を推定するためのベース（base）とステップ（step）を算出する。下記に示す形式の結果をレジスタ RT に置く。S はベースの結果の符号ビットである。



- ベースの結果は、通常の 23 bit ではなく、13 bit をその仮数部に持つ浮動小数点数として表現される。仮数部の残りの 10 bit は、ステップの絶対値をエンコードするための非正規化数の仮数部として使用される。指数は、ベースの指数と同一である。
- ステップの仮数部は、小数点の前に 0 があるという点、および小数点と仮数部の間に 3 つの 0 ビットがあるという点で、ベースの仮数部（および任意の正規化 IEEE 仮数部）とは異なる。表現される値は、下記のとおりである。

ベース (Base)	S 1.BaseFraction x $2^{BiasedExponent - 127}$
ステップ (Step)	0.000 StepFraction x $2^{BiasedExponent - 127}$

- x をレジスタ RA の初期値とすると、レジスタ RT に置かれた結果は、通常の IEEE の数値として解釈すると非ゼロ x の逆数の推定値を与える。
- レジスタ RA のオペランドがゼロ指数の場合は、ゼロ除算例外のフラグをセットする。

プログラミングの注意：この命令が返す結果は、Floating Interpolate 命令のオペランドとすることを意図したものである。

Floating Reciprocal Estimate 命令が生成する推定値は、補間およびニュートン・ラフソン法の単一ステップの後に、IEEE 単精度浮動小数点逆数の 1 ulp 内の結果を生成するために十分な精度を持つ。下記のコード・シーケンスを考察する。

```
FREST    y0,x              // table-lookup
FI       y1,x,y0           // interpolation
FNMS     t1,x,y1,ONE       // t1= -(x * y1 - 1.0)
FMA      y2,t1,y1,y1       // y2= t1 * y1 + y1
```

下記の 3 つの入力範囲をそれぞれ記述する必要がある。

ゼロ　　1/0 は、拡張範囲の SPU 単精度浮動小数点（sfp）の最大数値を与えると定義される。つまり、y2=0x7FFF FFFF (1.999.. x 2^{128})

大きな値　　$|x| \geq 2^{126}$ であれば、1/x はゼロにアンダーフローする。y2 = 0

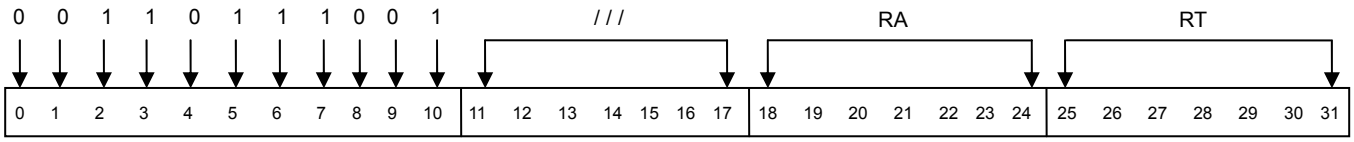
注意：この方式ではアンダーフローするが、IEEE の単精度逆数であればしないような x の特定の値がある。これが問題であれば、下記のコード・シーケンスで IEEE の結果を生成できる。

```
maxnounderflow=0x7e800000
min=0x00800000
msb=0x80000000
FCMEQ selmask,x,maxnounderflow
AND s1,x,msb
OR smin,s1,min
SELB y3,selmask,y2,smin
```

通常ケース　　1/x = Y ここで x * Y < 1.0 かつ x * INC(Y) ≥ 1.0
INC(y) は、y と同一の符号で絶対値が y の直後の大きさである単精度浮動小数点数値を与える。
絶対誤差上限：
$|Y - y2| \leq 1$ ulp　（y2 = Y、または INC(y2) = Y のどちらか）

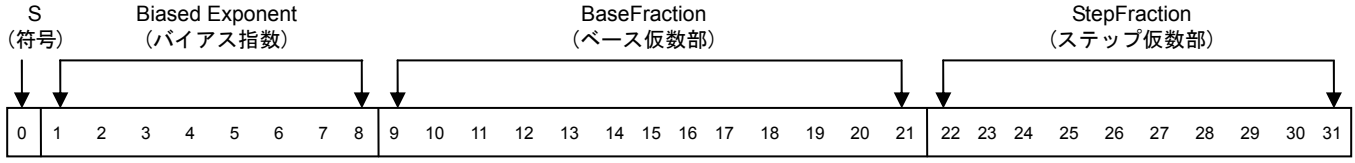
Floating Reciprocal Absolute Square Root Estimate 必須 v1.0

frsqest rt,ra



ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA のオペランドを使用し、オペランドの絶対値の平方根の逆数を推定するためのベースおよびステップを算出する。結果をレジスタ RT に置く。符号ビット（S）は 0 になる。



- x をレジスタ RA の初期値とすると、レジスタ RT に置かれた結果は、通常の IEEE の数値として解釈すると abs(x) の平方根の逆数の推定値を与える。
- レジスタ RA のオペランドがゼロ指数の場合は、ゼロ除算例外のフラグをセットする。

プログラミングの注意：この命令が返す結果は、Floating Interpolate 命令のオペランドとすることを意図したものである。

Floating Reciprocal Absolute Square Root Estimate 命令が生成する推定値は、補間およびニュートン・ラフソン法の単一ステップの後に、IEEE 単精度浮動小数点逆数の 1 ulp 内の結果を生成するために十分な精度を持つ。下記のコード・シーケンスを考察する。

```
mask=0x7fffffff
half=0.5
one=1.0
FRSQEST y0,x          // table-lookup
AND     ax,x,mask     // ax= ABS(x)
FI      y1,ax,y0      // interpolation
FM      t1,ax,y1      // t1= ax * y1
FM      t2,y1,HALF    // t2= y1 * 0.5
FNMS    t1,t1,y1,ONE  // t1= -(t1 * y1 - 1.0)
FMA     y2,t1,t2,y1   // y2= t1 * t2 + y1
```

下記の 3 つの入力範囲をそれぞれ記述する必要がある。

ゼロでかつ：x の仮数部 ≤ 0x000ff53c ならば y2=0x7fffffff (1.999.. x 2^{128})

ゼロでかつ：x の仮数部 > 0x000ff53c ならば y2 ≥ 0x7fc00000

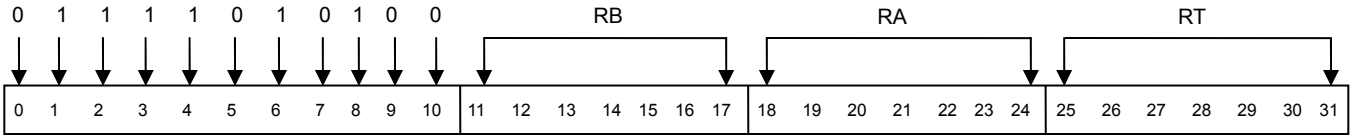
この結果を補正するために、下記のシーケンスを使用することもできる。

```
zero=0.0
mask=0x7fffffff
FCMEQ z,x,zero
AND zmask,z,mask
OR y3,zmask,y2
```

通常ケース　　1/sqrt(x) = Y ここで $x * Y^2 < 1.0$ かつ $x * INC(Y)^2 \geq 1.0$
INC(y) は、y と同一の符号で絶対値がyの直後の大きさである単精度浮動小数点数値を与える。
絶対誤差上限：
$|Y - y2| \leq 1$ ulp （0 と ±1 がすべてありうる）

Floating Interpolate 必須 v1.0

fi **rt,ra,rb**



ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RB のオペランドを分解し、215 ページの *Floating Reciprocal Estimate* で説明したフォーマット、すなわち符号、Biased Exponent、BaseFraction、StepFraction に従い、浮動小数点ベースとステップを生成する。
- レジスタ RA のビット 13 から 31 で、ビット 13 の左に小数点を持つ仮数部 Y を表現する。すなわち $Y \leftarrow 0.RA_{13:31}$

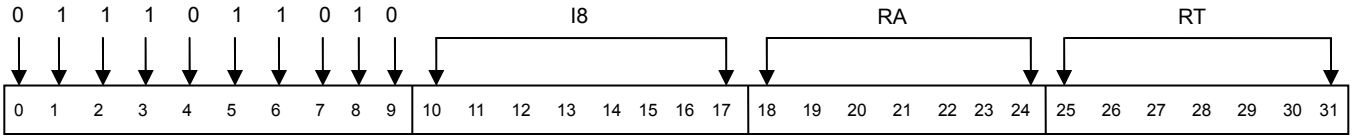
  結果は下記の式で算出される。

  $RT \leftarrow (-1)^{S} * (1.BaseFraction - 0.000StepFraction * Y) * 2^{(BiasedExponent\ -127)}$

プログラミングの注意：レジスタ RB のオペランドが、レジスタ RA をオペランドとした **frest** 命令または **frsqest** 命令の結果である場合、レジスタ RT に置かれた **fi** 命令の結果は、より正確な推定値を与える。

Convert Signed Integer to Floating 必須 v1.0

csflt rt,ra,scale

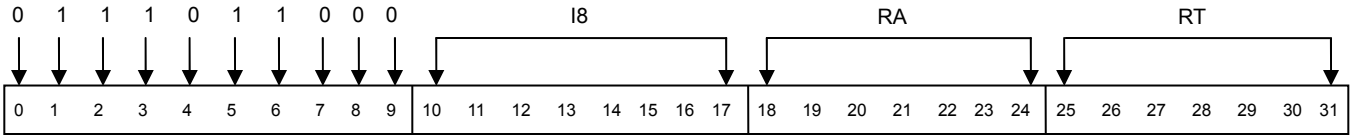


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の符号付き 32 bit 整数値を、拡張範囲の単精度浮動小数点値に変換する。
- 結果を 2^{scale} で除算し、レジスタ RT に置く。スケール因数 scale は、155 から I8 フィールドの符号なし値を減算して得られる 8 bit の符号なし整数である。scale の値が 0 から 127 の範囲になければ、演算結果は未定義である。
- スケール因数は、絶対値の小数点とレジスタ RA の右端との間のビット・ポジションの個数を表わす。ゼロのスケール因数は、レジスタ RA の値がスケールなしの整数であることを意味する。

Convert Floating to Signed Integer 必須 v1.0

cflts rt,ra,scale

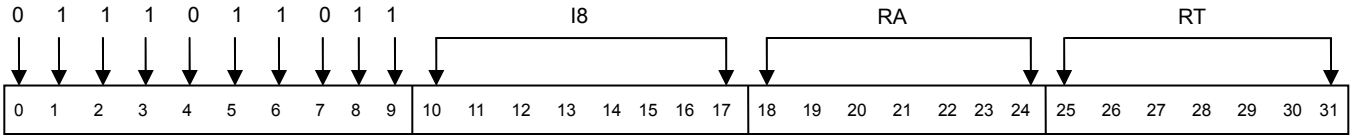


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の拡張範囲の単精度浮動小数点を、2^{scale} で乗算する。スケール因数 scale は、173 から I8 フィールドの符号なし値を減算して得られる 8 bit の符号なし整数である。scale の値が 0 から 127 の範囲になければ、演算結果は未定義である。
- 結果を符号付き 32 bit の整数に変換する。中間結果が、$(2^{31} - 1)$ より大きい場合は、$(2^{31} - 1)$ に飽和演算する。-2^{31} より小さい場合は、-2^{31} に飽和演算する。結果の符号付き整数を、レジスタ RT に置く。
- スケール因数は、結果の小数点の位置をレジスタ RT の右端からのビット・ポジションの個数として表現したものである。ゼロのスケール因数は、レジスタ RA の値がスケールなしの整数であることを意味する。

Convert Unsigned Integer to Floating 必須 v1.0

cuflt rt,ra,scale

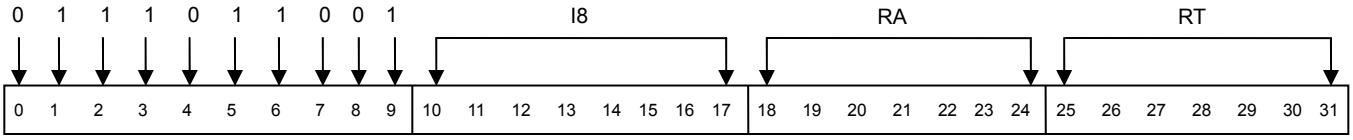


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の 32 bit の符号なし整数値を、拡張範囲の単精度浮動小数点値に変換する。
- 結果を 2^{scale} で除算し、レジスタ RT に置く。スケール因数 scale は、155 から I8 フィールドの符号なし値を減算して得られる 8 bit の符号なし整数である。scale の値が 0 から 127 の範囲になければ、演算結果は未定義である。
- スケール因数は、絶対値の小数点とレジスタ RA の右端との間のビット・ポジションの個数を表わす。ゼロのスケール因数は、レジスタ RA の値がスケールなしの整数であることを意味する。

Convert Floating to Unsigned Integer 必須 v1.0

cfltu rt,ra,scale

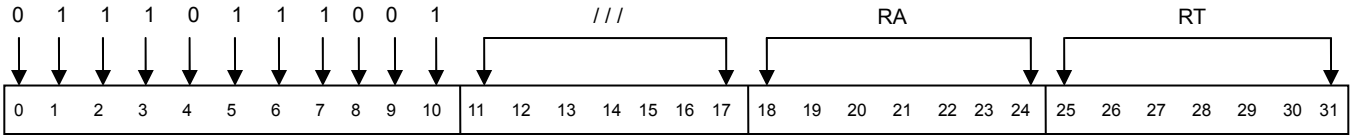


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の拡張範囲の単精度浮動小数点を、2^{scale} で乗算する。スケール因数 scale は、173 から I8 フィールドの符号なし値を減算して得られる 8 bit の符号なし整数である。scale の値が 0 から 127 の範囲になければ、演算結果は未定義である。
- 結果を 32 bit の符号なし整数に変換する。中間結果が、$(2^{32} - 1)$ より大きい場合は、$(2^{32} - 1)$ に飽和演算する。負の場合は、ゼロに飽和演算する。結果の符号なし整数を、レジスタ RT に置く。
- スケール因数は、結果の小数点の位置をレジスタ RT の右端からのビット・ポジションの個数として表現したものである。ゼロのスケール因数は、レジスタ RT の値がスケールなしの整数であることを意味する。

Floating Round Double to Single 必須 v1.0

frds rt,ra

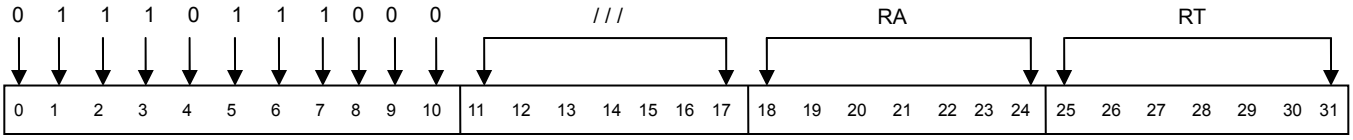


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RA の倍精度浮動小数点値を、単精度浮動小数点値に丸め、左のワード・スロットに置く。変換は *9.2.1 単精度フォーマットと倍精度フォーマットの間の変換*で記述されているように行なわれる。右のワード・スロットにはゼロを置く。
- 丸めは、Floating-Point Status Register で定義された丸めモードに従って行なわれる。倍精度例外を検出し、FPU Status Register に累積する。

Floating Extend Single to Double 必須 v1.0

fesd rt,ra

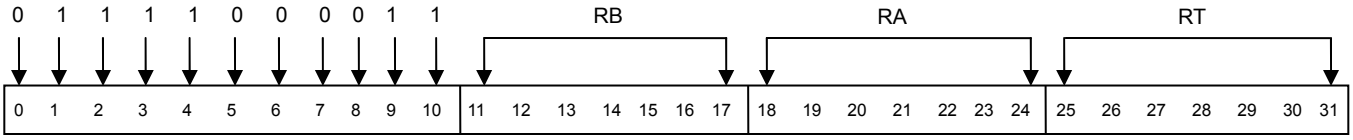


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RA の左スロットにある単精度浮動小数点値を、倍精度浮動小数点値に変換し、レジスタ RT に置く。変換は *9.2.1 単精度フォーマットと倍精度フォーマットの間の変換*で記述されているように行なわれる。右のワード・スロットの値は、無視する。
- 倍精度浮動小数点例外を検出し、FPU Status Register に塁積する。

Double Floating Compare Equal　　オプション　v1.2

dfceq　　rt,ra,rb

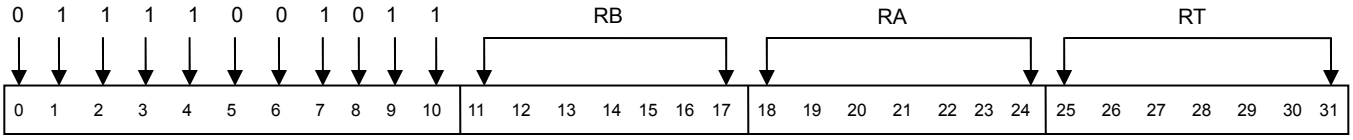


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RA の倍精度浮動小数点値を、レジスタ RB の倍精度浮動小数点値と比較する。これらの値が等しければ、すべて 1（真）の結果をレジスタ RT に生成する。そうでなければ、0（偽）の結果をレジスタ RT に生成する。
- 2つのゼロを比較すると、符号には関係なく、常に等しいという比較結果になる。
- NaN はすべての他オペランドに対して偽の比較結果となる。2つの同一ビットパターンの NaN 同士でさえも、偽の結果を生成する。
- NaN にアクセスすると、FPSCR 中の対応する INV 例外ビットがセットされる。

Double Floating Compare Magnitude Equal　オプション　v1.2

dfcmeq　　rt,ra,rb

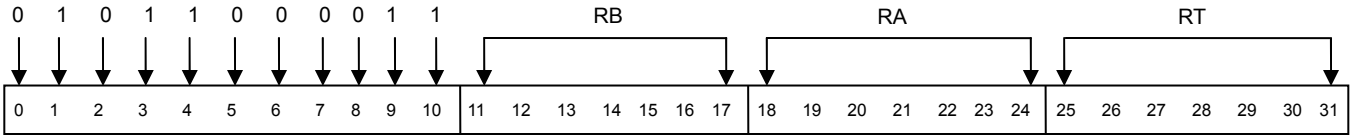


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RA の倍精度浮動小数点数の絶対値を、レジスタ RB の倍精度浮動小数点数の絶対値と比較する。これらの絶対値が等しければ、すべて 1（真）の結果がレジスタ RT に生成される。そうでなければ、0（偽）の結果がレジス RT に生成される。
- 2つのゼロを比較すると、符号には関係なく、常に等しいという比較結果になる。
- NaN はすべての他オペランドに対して偽の比較結果となる。2つの同一ビットパターンの NaN 同士でさえも、偽の結果を生成する。
- NaN にアクセスすると、FPSCR 中の対応する INV 例外ビットがセットされる。

Double Floating Compare Greater Than　オプション　v1.2

dfcgt　　rt,ra,rb

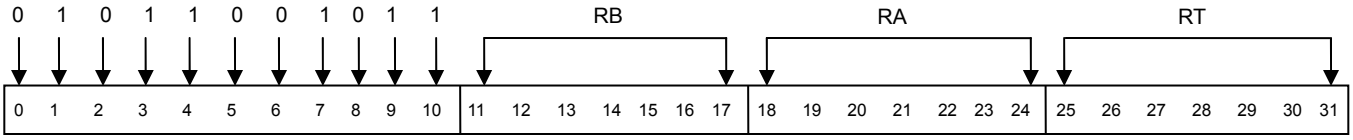


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RA の倍精度浮動小数点値を、レジスタ RB の倍精度浮動小数点値と比較する。RA の値が、RB の値より大きい場合は、すべて 1（真）の結果がレジスタ RT に生成される。そうでなければ、0（偽）の結果がレジスタ RT に生成される。
- 2つのゼロを比較すると、符号ビットには関係なく、決して、より大きいという比較結果にはならない。
- NaN はすべての他オペランドに対して偽の比較結果となる。2つの同一ビットパターンの NaN 同士でさえも、偽の結果を生成する。
- NaN にアクセスすると、FPSCR 中の対応する INV 例外ビットがセットされる。

Double Floating Compare Magnitude Greater Than　　オプション　v1.2

dfcmgt　　rt,ra,rb

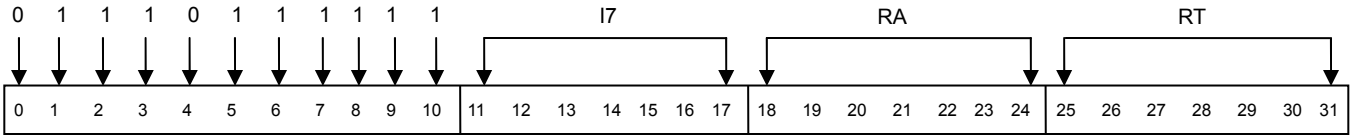


ダブルワード・スロット 2個それぞれに、下記を行なう。

- レジスタ RA の倍精度浮動小数点数の絶対値を、レジスタ RB の倍精度浮動小数点数の絶対値と比較する。レジスタ RA の値の絶対値が、レジスタ RB の値の絶対値より大きい場合は、すべて 1（真）の結果がレジスタ RT に生成される。そうでなければ、0（偽）の結果がレジス RT に生成される。
- 2つのゼロを比較すると、符号には関係なく、決して、より大きいという比較結果にはならない。
- NaN はすべての他オペランドに対して偽の比較結果となる。2つの同一ビットパターンの NaN 同士でさえも、偽の結果を生成する。
- NaN にアクセスすると、FPSCR 中の対応する INV 例外ビットがセットされる。

Double Floating Test Special Value　オプション　v1.2

dftsv　　rt,ra,value



ダブルワード・スロット 2個それぞれに、下記を行なう。

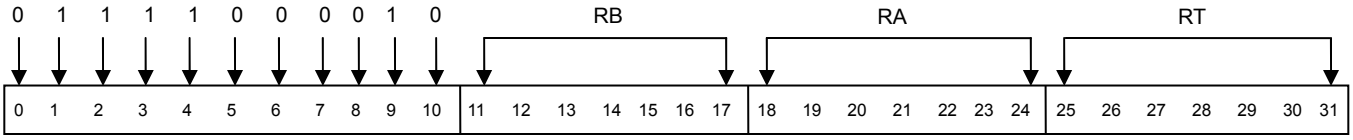
- レジスタ RA の倍精度浮動小数点数に特殊値のテストを行なう。I7 の各ビットによって以下の 7つのチェックを指定する。

I7	RA 値のカテゴリ
1000000	NaN
0100000	+無限大
0010000	–無限大
0001000	+0
0000100	–0
0000010	正の非正規化数
0000001	負の非正規化数

- 指定されたチェックの 1つ以上が真であれば、すべて 1 の結果がレジスタ RT に生成される。指定されたチェックのいずれも該当しなければ、すべて 0 の結果がレジス RT に生成される。

Floating Compare Equal 必須 v1.0

fceq **rt,ra,rb**

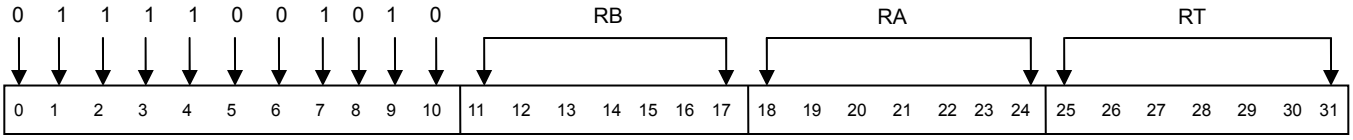


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の浮動小数点値を、レジスタ RB の浮動小数点値と比較する。これらの値が等しければ、すべて 1（真）の結果をレジスタ RT に生成する。そうでなければ、0（偽）の結果をレジスタ RT に生成する。2 つのゼロを比較すると、仮数部や符号には関係なく、常に等しいという比較結果になる。
- この命令は常に、拡張範囲モードで実行され、モード・ビットの設定を無視する。

Floating Compare Magnitude Equal 必須 v1.0

fcmeq **rt,ra,rb**

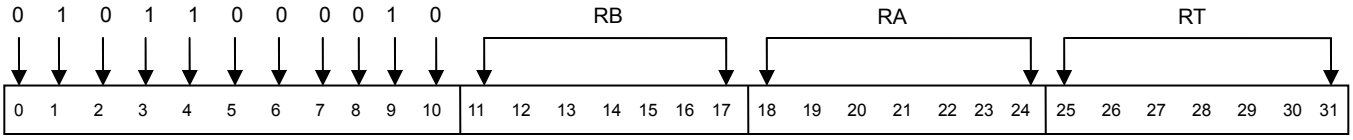


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の浮動小数点数の絶対値を、レジスタ RB の浮動小数点数の絶対値と比較する。これらの絶対値が等しければ、すべて 1（真）の結果がレジスタ RT に生成される。そうでなければ、0（偽）の結果がレジス RT に生成される。2 つのゼロを比較すると、仮数部や符号には関係なく、常に等しいという比較結果になる。
- この命令は常に、拡張範囲モードで実行され、モード・ビットの設定を無視する。

Floating Compare Greater Than 必須 v1.0

fcgt **rt,ra,rb**

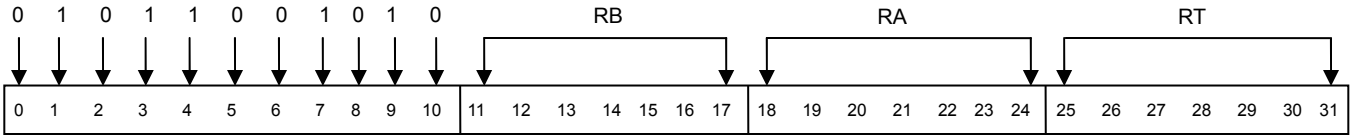


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の浮動小数点値を、レジスタ RB の浮動小数点値と比較する。RA の値が、RB の値より大きい場合は、すべて 1（真）の結果がレジスタ RT に生成される。そうでなければ、0（偽）の結果がレジスタ RT に生成される。2 つのゼロを比較すると、符号ビットや仮数部には関係なく、決して、より大きいという比較結果にはならない。
- この命令は常に、拡張範囲モードで実行され、モード・ビットの設定を無視する。

Floating Compare Magnitude Greater Than 必須 v1.0

fcmgt **rt,ra,rb**

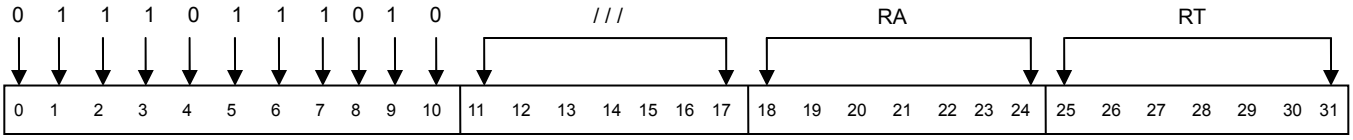


ワード・スロット 4個それぞれに、下記を行なう。

- レジスタ RA の浮動小数点数の絶対値を、レジスタ RB の浮動小数点数の絶対値と比較する。レジスタ RA の値の絶対値が、レジスタ RB の値の絶対値より大きい場合は、すべて 1（真）の結果がレジスタ RT に生成される。そうでなければ、0（偽）の結果がレジス RT に生成される。2 つのゼロを比較すると、仮数部や符号には関係なく、決して、より大きいという比較結果にはならない。
- この命令は常に、拡張範囲モードで実行され、モード・ビットの設定を無視する。

Floating-Point Status and Control Register Write 必須 v1.0

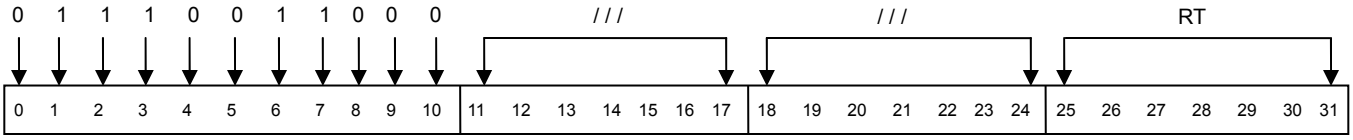
fscrwr ra



レジスタ RA の 128 bit 値を、FPSCR に書き込む。FPSCR の未使用ビットの値は、未定義である。RT は、偽のターゲットである。実装によっては、この命令があたかも RT に格納する値を生成するかのように、命令スケジューリングを行なう場合がある。後続近傍の命令に対するソースデータとならないように RT をプログラミングすることによって、不必要な遅延をプログラムで避けることができる。偽のターゲットへは何も書き込まれない。

Floating-Point Status and Control Register Read 必須 v1.0

fscrrd rt



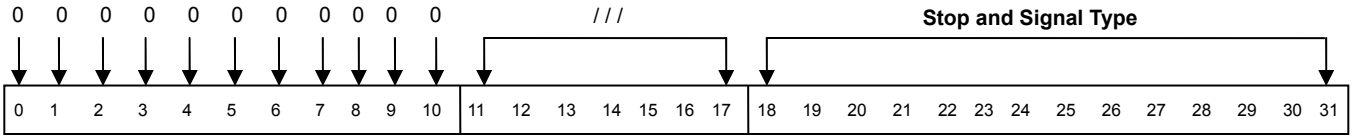
本命令は FPSCR の値を読み出す。結果においては、FPSCR の未使用のビットが強制的に 0 にされる。結果をレジスタ RT に置く。

10. 制御命令

本セクションでは、SPU の制御命令をリストアップし、説明する。

Stop and Signal 必須 v1.0

stop

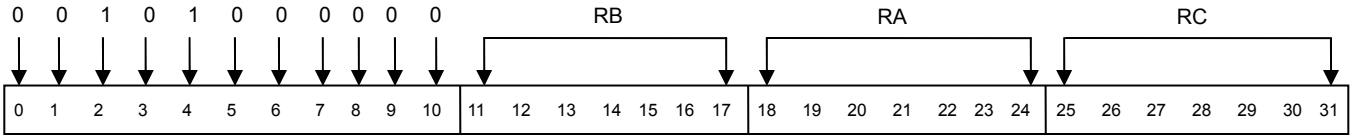


SPU のプログラムの実行を停止し、外部環境にシグナル通知する。それ以降の命令は実行しない。

PC ← PC + 4 & LSLR
precise stop

Stop and Signal with Dependencies 必須 v1.0

stopd



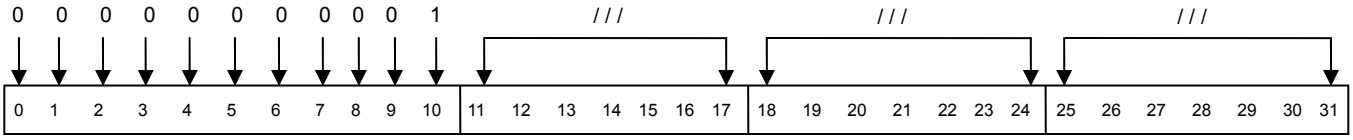
SPU のプログラム実行を停止する。

```
PC ← PC + 4 & LSLR
precise stop
```

プログラミングの注意：依存関係を持つ命令を **stopd** 命令と置き換えて、命令タイミングに影響を与えずにブレークポイントを生成できる点でのみ、この **stopd** 命令は標準的な実装において **stop** 命令と異なる。

No Operation (Load)　　必須　v1.0

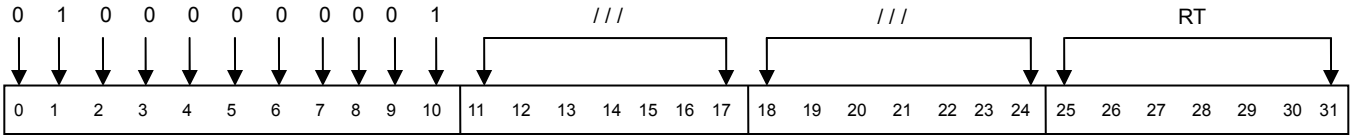
lnop



この命令は、プログラムの実行に全く影響しない。実装定義による命令発行の制御を提供するために存在する命令である。

No Operation (Execute)　　必須　v1.0

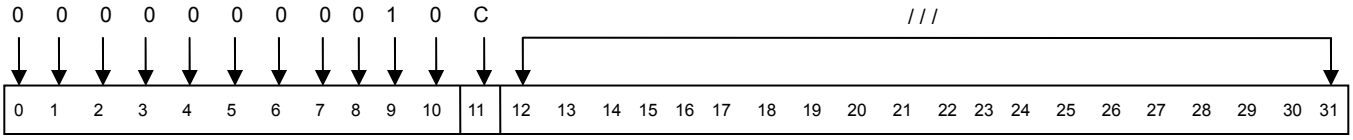
nop



この命令は、プログラムの実行に全く影響しない。実装定義による命令発行の制御を提供するために存在する命令である。RT は、偽のターゲットである。実装によっては、この命令があたかも RT に格納する値を生成するかのように、命令スケジューリングを行なう場合がある。後続近傍の命令に対するソースデータにならないように、RT をプログラミングすることによって、不必要な遅延をプログラムで避けることができる。偽のターゲットへは何も書き込まれない。

Synchronize 必須 v1.0

sync

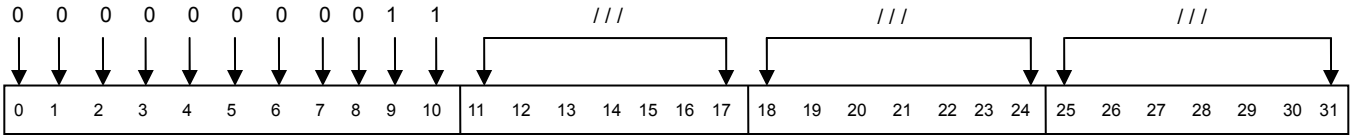


後続の命令をフェッチする前に、保留中のストア命令がすべて完了するまでプロセッサを待たせることを除けば、この命令はプログラムの実行に全く影響しない。命令ストリームを更新するストア命令の後にはこの命令を使用しなければならない。

C 機能ビットをセットすると、命令同期が発生する前に、チャネル同期が発生する。チャネル命令を通じて更新された SPU 状態は、チャネル同期によって実行に影響を及ぼすようになる。同期については、253 ページの *セクション 13「同期と順序付け*」で詳しく論じる。

Synchronize Data 必須 v1.0

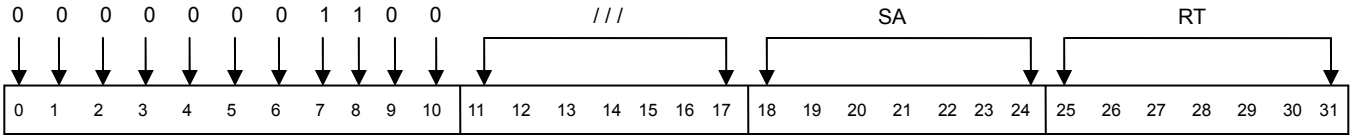
dsync



続行の前に、先行のロード、ストアおよびチャネル命令をすべて強制的に完了させる。先行命令が完了するまで、後続のロード、ストアおよびチャネル命令は開始しない。**dsync** 命令を使用すれば、他の主体から観測された場合でも、SPU ソフトウェアは、ローカル・ストレージ・データの一貫性を保証することができる。この命令は、プロセッサがすでに終えた可能性のある命令プリフェッチには影響しない。同期については、253 ページの *セクション 13「同期と順序付け」* で詳しく論じる。

Move from Special-Purpose Register 必須 v1.0

mfspr rt,sa



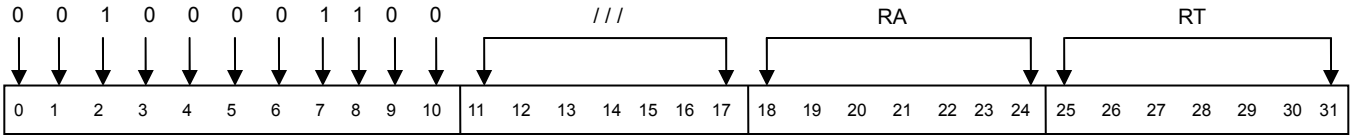
Special-Purpose Register（SPR）SA をレジスタ RT にコピーする。SPR SA が定義されていなければ、0 がコピーされる。

注記： SPU ISA では **mtspr** 命令および **mfspr** 命令を 128 bit 演算として定義する。実装においては、32 bit 幅のレジスタを定義する場合がある。その場合、32 bit 値はプリファード・スロットを占め、その他のスロットにはゼロが返される。

```
if defined(SPR(SA)) then RT ← SPR(SA)
else                     RT ← 0
```

Move to Special-Purpose Register 必須 v1.0

mtspr **sa, rt**



レジスタ RT の値を、Special-Purpose Register（SPR）SA に書き込む。SPR SA が定義されていなければ、何ら操作は行なわれない。

注記： SPU ISA では **mtspr** 命令および **mfspr** 命令を 128 bit 演算として定義する。実装においては、32 bit 幅のレジスタを定義する場合がある。その場合、プリファード・スロットの 32 bit 値が使用され、その他のスロットの値は無視される。

```
if defined(SPR(SA)) then
        SPR(SA) ← RT
else
        do nothing
```

11. チャネル命令

SPU では、メッセージパッシングに基づくチャネル・インターフェースと呼ばれる入出力インターフェースが提供される。本セクションでは、チャネル・インターフェースを介した SPU と外部デバイス間の通信に使用する命令を説明する。

チャネルとは、SPU と外部デバイス間の 128 bit 幅の通信経路である。各チャネルは、1 方向のみの動作であり、SPU がチャネル上で行なえる操作に応じて、「読み出しチャネル」、または「書き込みチャネル」と呼ばれる。SPU プログラムがチャネルからの読み出しあるいはチャネルへの書き込みを行える命令が用意されており、行なう操作はアドレス指定したチャネルのタイプと一致する必要がある。

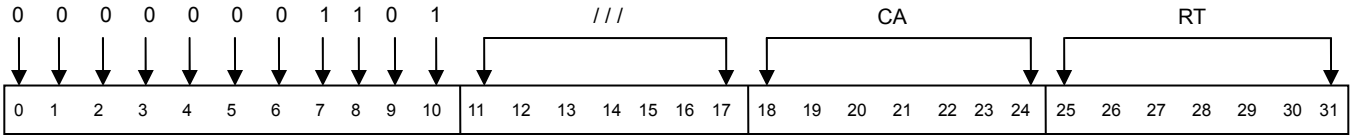
実装におけるチャネル個数は、最大 128 までの任意個である。各チャネルは、0 から 127 の範囲のチャネル・ナンバーを持つ。チャネル・ナンバーには特別な意味はなく、チャネルの方向とチャネル・ナンバーの間に関連性はない。

チャネルや外部デバイスには容量が定義されている。チャネルの容量とは、遅延せずに遂行できる読み出し/書き込みの最小回数である。容量を超えてチャネルにアクセスを試みると、容量が使用可能になってアクセスが完了できるようになるまで、命令処理は中止される。SPU は、チャネルの容量を計測するカウンタを保持し、チャネル容量を読み出す命令を提供している。

容量が使用可能である限り、チャネルや外部デバイスは、SPU に実行の遅延を求めること無しに、連続した SPU アクセスを処理できる。チャネルの容量を超えてチャネルへの書き込みを試みると、外部デバイスがチャネルを空にするまで、SPU はハングする。また、チャネルが空のときにチャネルの読み出しを試みると、デバイスがそのチャネルにデータを挿入するまで SPU はハングする。

Read Channel 必須 v1.0

rdch **rt,ca**



SPU はチャネル CA にあるデータが使用可能（容量が使用可能）になるのを待つ。そのチャネルに対するデータが使用可能であれば、データをチャネルから移動して、レジスタ RT に置く。

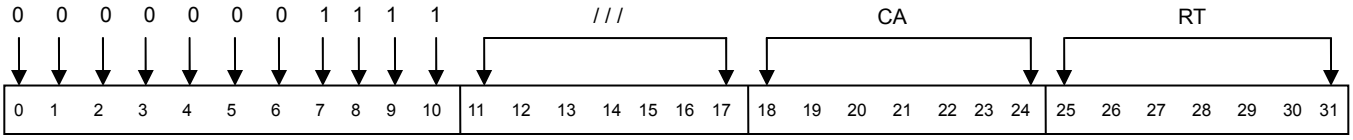
CA フィールドに指定されたチャネルが有効な読み出し可能チャネルでなければ、**rdch** 命令で、あるいはその後で、SPU が停止する。

注記： SPU ISA では **rdch** 命令および **wrch** 命令を 128 bit 演算として定義する。実装においては、32 bit 幅のチャネルを定義する場合がある。その場合、32 bit 値はプリファード・スロットを占め、その他のスロットにはゼロが返される。

```
if readable(Channel(CA)) then
        RT ← Channel(CA)
else
        Stop after executing zero or more instructions after the rdch.
```

Read Channel Count 必須 v1.0

rchcnt rt,ca

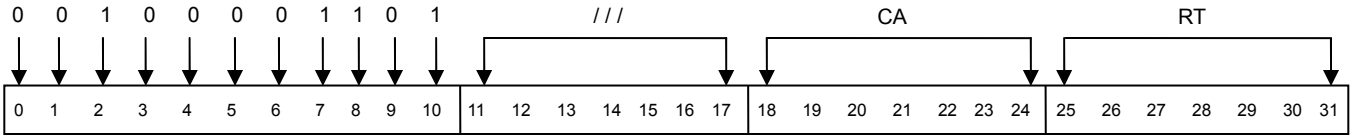


チャネル CA のチャネル容量を、レジスタ RT のプリファード・スロットに置く。未実装チャネルのチャネル容量は 0 である。

$RT^{0:3} \leftarrow$ Channel Capacity(CA)
$RT^{4:15} \leftarrow 0$

Write Channel 必須 v1.0

wrch **ca,rt**



SPU は、**wrch** 命令を実行する前に、チャネル CA の容量が使用可能になるのを待つ。チャネルに使用可能な容量があれば、レジスタ RT の値をチャネル CA に置く。チャネル書き込み命令が、有効な書き込み可能チャネルではないチャネルをターゲットとする場合、**wrch** 命令で、あるいはその後で、SPU が停止する。

注記： SPU ISA では **rdch** 命令および **wrch** 命令を 128 bit 演算として定義する。実装においては、32 bit 幅のチャネルを定義する場合がある。その場合、プリファード・スロットの 32 bit 値が使用され、その他のスロットの値は無視される。

```
if writeable(Channel(CA)) then
        Channel(CA) ← RT
else
        Stop after executing zero or more instructions after the wrch.
```

12. SPU 割り込み機能

本セクションでは、SPU の割り込み機能を説明する。

外部条件は、チャネル・インターフェースを介して制御される外部機能を通じて監視および管理される。外部条件は、下記の機能を通じて SPU 命令シーケンスに影響を及ぼす。

- **bisled** 命令

  bisled 命令は外部条件が存在するかをテストし、存在すればターゲットに分岐する。**bisled** 命令によって SPU ソフトウェアは、外部条件のポーリングを行ない、ハンドラのサブルーチンを呼び出すことができる。ポーリングが望ましくない場合、外部条件が発生したときに、SPU に通常の命令処理を中断させて、ハンドラ・サブルーチンのベクターに飛ぶようにすることも可能である。

- 割り込み機能

下記の間接分岐命令によって、クリティカルなサブルーチンにおいて、ソフトウェアは割り込み機能を許可したり禁止したりすることができる。

- **bi**
- **bisl**
- **bisled**
- **biz**
- **binz**
- **bihz**
- **bihnz**

これらの分岐命令はすべて、[D] および [E] 機能ビットを備えている。これらの分岐のいずれかが成立すると、ターゲット命令が実行される前に、割り込み許可状態が変化する。*表 12-1* で、機能ビットの設定とその結果を説明する。

表 12-1 機能ビット [D] と [E] の設定および結果

機能ビットの設定		結果
[D]	[E]	
0	0	状態は変更されない。
0	1	割り込み処理が許可される。
1	0	割り込み処理が禁止される。
1	1	未定義の動作を引き起こす。

12.1 SPU 割り込みハンドラ

SPU は、単一の割り込みハンドラをサポートしている。このハンドラに対するエントリ・ポイントは、ローカル・ストレージのアドレス 0 である。条件が存在し、割り込みが許可状態の場合、SPU はアドレス 0 に分岐し、割り込み機能を禁止状態にする。次に実行すべき命令のアドレスは、SRR0 レジスタに保存される。**iret** 命令を使用することによって、ハンドラから戻ることができる。**iret** は、SRR0 レジスタに保持されているアドレスに間接分岐する。**iret** は、その他の間接分岐と同様に、割り込みを再度許可状態にできる [E] 機能ビットを持つ。

12.2 SPU 割り込み機能チャネル

割り込み機能は、コンフィギュレーション、状態観測および状態復元のために、いくつかのチャネルを使用する。SRR0 の現在の値は SPU_RdSRR0 チャネルから読み出すことができ、SRR0 への書き込みアクセスが SPU_WrSRR0 チャネルで提供される。SRR0 が **wrch** 14 によって書き込まれた場合は、その新しい値が **iret** 命令にて使用可能であることを確実にするために、同期が必要になる。この同期は、[C] すなわち Channel Sync 機能ビットをセットした **sync** 命令を実行することによって提供される。この同期を実行しないと、**wrch** 14 命令の後に実行する **iret** 命令が、予測不可能なアドレスに分岐する。SPU_RdSRR0 および SPU_WrSRR0 は、ソフトウェアがローカル・ストレージ内の保存領域に SRR0 をセーブ、リストアすることを可能にするので、ネストされた割り込みに対応できる。

13. 同期と順序付け

SPU では逐次的に順序付けされたプログラミング・モデルを提供しているので、一部の例外を除いて、すべての先行する命令は、次の命令が開始される前に完了しているように見える。

SPU 内蔵のシステムでは、ローカル・ストレージへのダイレクト・アクセスを行なう外部デバイスを備えていることが多い。外部デバイスがチャネル・インターフェースを通じても SPU と通信を行なうことができる一般的な構成を、*図 13-1* に示す。 これらのシステムは、メッセージパッシング機能を持った共有メモリ・マルチプロセッサである。

図 13-1 ローカル・ストレージへの複数アクセスを持つシステム

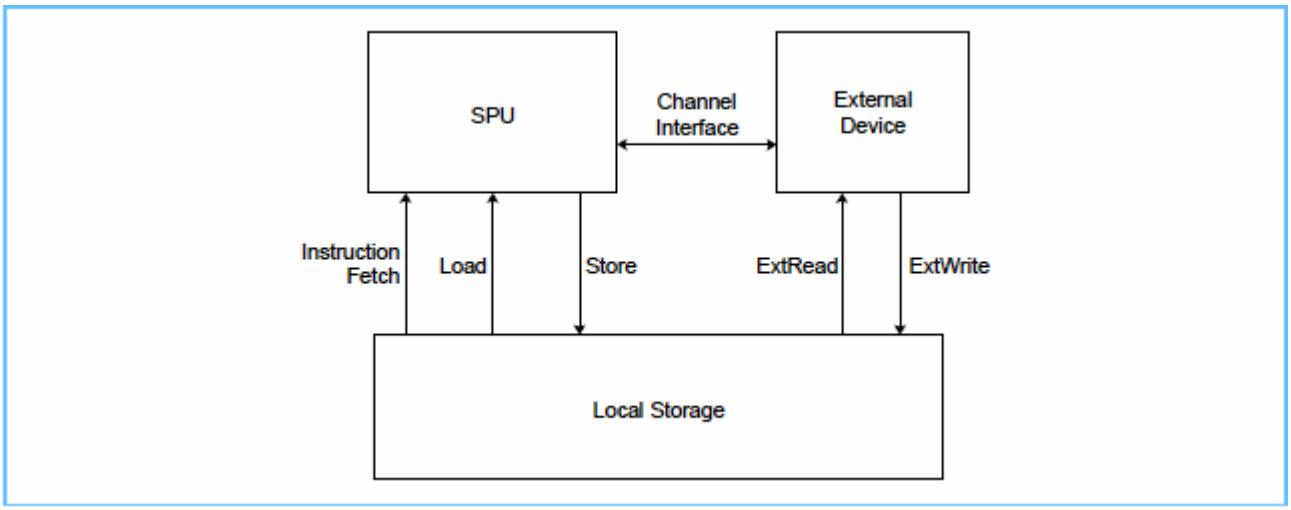


表 13-1 に、ローカル・ストレージが行なう 5 種類のトランザクションを示す。SPU ISA は、外部デバイスのふるまいや、外部デバイスによるローカル・ストレージへのアクセス方法については定義していない。このドキュメントで外部からのローカル・ストレージへの書き込みについて言及するときには、外部デバイスがローカル・ストレージに送ったデータが、後続の SPU Load によってローカル・ストレージから取得できるようになっていることを前提としている。

表 13-1 ローカル・ストレージ・アクセス

名前	説明
Load	SPU の Load 命令がローカル・ストレージの読み出しからデータを取得する。
Store	SPU の Store 命令がローカル・ストレージへの書き込みへデータを送る。
Fetch	SPU の命令フェッチがローカル・ストレージの読み出しからデータを取得する。
ExtWrite	外部デバイスがローカル・ストレージへの書き込みへデータを送る。
ExtRead	外部デバイスがローカル・ストレージの読み出しからデータを取得する。

外部デバイスによるローカル・ストレージ・アクセスと SPU によるローカル・ストレージ・アクセスとの間のやりとりでは、SPU の実装固有の並べ替え、投機的実行（スペキュレーション）、バッファリング、そしてキャッシングの効果があらわになるかもしれない。本セクションでは、一貫性のある結果を得るために、これら一連のトランザクションを順序付けする方法について論じる。

13.1 SPU ローカル・ストレージ・アクセスの投機的実行、並べ替え、キャッシング

SPU ローカル・ストレージのアクセスは weakly consistent である（*PowerPC Virtual Environment Architecture, Book II* を参照）。したがって、ストレージ・アクセスをもたらす命令に適用すると、逐次的な実行モデルは、こうしたアクセスがプログラム順で遂行されているように見えることを、命令を実行している SPU に対してのみ保証する。これらのアクセスは、外部からのローカル・ストレージへのアクセス、あるいは、SPU の命令フェッチに対しては、プログラム順で遂行されているようには見えないかもしれない。すなわち、以下を意味する。外部からのローカル・ストレージへの書き込みが無ければ、任意の特定アドレスからの SPU Load は、そのアドレスに対する最新の SPU Store によって書き込みされたデータを返す。しかしながら、そのアドレスからの命令フェッチが必ずしもそういったデータを返すとは限らない。

SPU では、ローカル・ストレージへのアクセスをキャッシュやバッファしたり、その他に、並べ替えることも許されている。SPU の Load、Store および 命令フェッチは、ローカル・ストレージに実際にアクセスするかもしれないし、しないかもしれない。SPU は投機的なローカル・ストレージの読み出しを行なうことがある。つまり、SPU はプログラムが要求していない命令のためのローカル・ストレージの読み出しを行なうことがある。SPU は投機的なローカル・ストレージへの書き込みは行わない。もし SPU の Store がローカル・ストレージにアクセスする場合は、SPU はプログラムが要求している Store のためのローカル・ストレージへの書き込みのみを行なう。命令フェッチ、Load、Storeのローカル・ストレージへのアクセスは、任意の順で生じうる。

13.2 SPU 内部実行状態

チャネル・インターフェースを使用することで、SPU の内部実行状態を変更できる。内部実行状態とは、チャネル・インターフェースを通じて変更された、命令の順序や実行に影響を及ぼす SPU 内部の（但し、ローカル・ストレージ以外の）任意の状態のことである。例えば、プログラムは SPU_WrSRR0 チャネルに書き込みを行なうことで SRR0 を変更でき、SRR0 は内部実行状態である。チャネル・インターフェースを通じた状態の変更は、 SPU プログラムの実行と同期されているとは限らない。

13.3 同期プリミティブ

SPU では 3 つの同期命令、**dsync**、**sync**、および **sync.c** が提供されている。255 ページの*表 13-2「同期命令」*に示すとおり、これらの命令は、一貫性効果と命令のシリアライズ効果の両方を有する。プログラムでは、これらプリミティブの一貫性効果を利用して、ローカル・ストレージの状態が SPU Load/Store と一貫性を保つように保証することができる。命令のシリアライズ効果によって、SPU プログラムはローカル・ストレージ・アクセスの順序付けを行なうことができる。

dsync 命令は、Load、Store、そしてチャネル・アクセスの順序付けを行なう。ただし、命令フェッチについては行なわない。**dsync** が完了すると、SPU はすべての先行する Load、Store、およびチャネル・アクセスを完了しており、後続する Load、Store、あるいはチャネル・アクセスの実行は開始していないことになる。この時点では、外部からのローカル・ストレージのあるアドレスの読み出しは、そのアドレスに対する最新の SPU Store によって格納されたデータを返す。**dsync** の後の SPU Load は、**dsync** が完了する時点より前に外部から書き込みされたデータを返す。**dsync** 命令は、実際のローカル・ストレージ状態に関しての、SPU 命令の順序付け、および Load/Storeの一貫性にのみ影響を及ぼす。SPU は、ローカル・ストレージにアクセスする外部デバイスに対して **dsync** 通知をブロードキャストはしないので、外部デバイスの状態には影響を与えない。

sync 命令は **dsync** とほぼ同じであるが、命令フェッチの順序付けも行なう。**sync** 命令の後で行なうあるローカル・ストレージ・アドレスからの命令フェッチは、そのアドレスに対する最新の store 命令あるいは外部からの書き込みによって格納されたデータを返す。**sync.c** 命令は **sync** 命令を基にした追加である。**sync.c** 命令は、先行する **wrch** 命令によって生じた内部状態への影響が伝播して後続命令の実行に影響を及ぼすことを保証する。SPU の実行は start イベントで開始され、stop イベントで終了する。start および stopはいずれも **sync.c**

を遂行する。

表 13-2 同期命令

命令	一貫性（コンシステンシ）効果	命令のシリアライズ効果
dsync	後続の外部からの読み出しが、先行する Store によって書き込みされたデータにアクセスすることを保証する。 後続の Load が、外部からの書き込みによって書き込みされたデータにアクセスすることを保証する。	**dsync** より先行する命令によるローカル・ストレージの Load/Store アクセスを **dsync** の完了よりも先に完了させる。 **dsync** より先行する命令によるチャネル読み出し操作を **dsync** の完了よりも先に完了させる。 **dsync** より後の命令によるローカル・ストレージの Load/Store アクセスを **dsync** の完了の後に発生させる。 **dsync** より後の命令によるチャネル読み出し/書き込み操作を **dsync** が完了した後に発生させる。
sync	後続の外部からの読み出しが、先行する Store によって書き込みされたデータにアクセスすることを保証する。 後続の命令フェッチが、先行する Store および外部からの書き込みによって書き込みされたデータにアクセスすることを保証する。 後続の Load が、外部からの書き込みによって書き込みされたデータにアクセスすることを保証する。	**sync** より先行する命令によるすべてのローカル・ストレージおよびチャネルへのアクセスを **sync** の完了よりも先に完了させる。 **sync** より後の命令によるすべてのローカル・ストレージおよびチャネルへのアクセスを **sync** の完了の後に発生させる。
sync.c	後続の外部からの読み出しが、先行する Store によって書き込みされたデータにアクセスすることを保証する。 後続の命令フェッチが、先行する Store および外部からの書き込みによって書き込みされたデータにアクセスすることを保証する。 後続の Load が、外部からの書き込みによって書き込みされたデータにアクセスすることを保証する。 後続命令の処理に、先行する **wrch** 命令によって変更されたすべての内部実行状態の影響が及ぶことを保証する。	**sync.c** より先行する命令によるすべてのローカル・ストレージおよびチャネルへのアクセスを **sync.c** の完了よりも先に完了させる。 **sync.c** より後の命令によるすべてのローカル・ストレージおよびチャネルへのアクセスを **sync.c** の完了の後に発生させる。

表 13-3 は、ローカル・ストレージを変更する動作と他のローカル・ストレージ読み出し/書き込みの間において、どの同期プリミティブが必要かを示している。SPUプログラムでは、自身の load/store命令の間に同期プリミティブを必要とせずに、load命令は、最後の先行 store命令がストアしたデータを得ることができる。

しかし、命令ストリームにストアを行なうプログラムでは、新たにストアした命令に到達する前に **sync**命令を実行しなければならない。**sync**命令は、**sync**命令の前の最後の store命令より後に命令フェッチが命令を読み出すように強制する。**sync**命令無しでは、SPUは新たにストアされた命令を実行するかもしれないし、しないかもしれない。SPUは、最後の syncイベントの時点でローカルストレージにあった命令を実行する可能性がある。

ローカル・ストレージの外部アクセスが生じ、かつ、それが、SPUによる特定のローカル・ストレージ・アクセスより前か後かが明確である場合、SPUと外部デバイスの間でデータを転送させるには同期が必要である。同期無しでは、SPUプログラムのどの実行地点ともつじつまの合わないローカル・ストレージの状態が、外部デバイスには見えてしまう可能性がある。

例えば、SPUプログラムがローカル・ストレージを介して外部リーダにデータを送る際には、データをストアした後、**dsync**命令を実行しなければならない。外部の読み出しが **dsync**命令の後に生じた場合は、ストアされたデータが読み出されることになる。外部ライタがローカル・ストレージに置いたデータを SPUプログラムがロードする際には、load命令を実行する前に、まず **dsync**命令を実行しなければならない。外部書き込みの後に **dsync**命令が実行された場合には、後続の load命令は外部ライタがストアしたデータを読み出せることになる。

表 13-3 ローカル・ストレージへの複数アクセスの同期

ライタ	ローカル・ストレージ書き込みと同期すべきローカル・ストレージ・アクセス				
	Store	Load	Fetch	ExtRead	ExtWrite
Store	不要	不要	**sync**	**dsync**	**dsync**
ExtWrite	**dsync**	**dsync**	**sync**	N/A	N/A

注記： SPU ISAでは、外部リーダ/ライタがそれらのローカル・ストレージへのアクセスを順序付ける方法を定義しない。表 13-3 では、外部リーダ/ライタに関する項目は N/A として示している。

13.4 SPU ローカル・ストレージ・アクセスのキャッシング

SPU の実装では、ローカル・ストレージ・データのキャッシュを命令またはデータのいずれか、あるいは両方に備えている場合がある。同期によってローカル・ストレージの状態の一貫性が求められた場合は、これらのキャッシュは、必ずローカル・ストレージへ/からのデータを反映しなければならない。**dsync** 命令は、変更されたデータが、ローカル・ストレージへアクセスする外部デバイスにとって可視であり、また、これらの外部デバイスにより変更されるデータが、後続の Load および Store にとって可視となることを保証する。**sync** 命令は更に、Store あるいは外部からの Put が、後続の命令フェッチにとって可視となることを保証する。例えば、スヌープしない命令キャッシュは、**sync** が実行されると必ず無効化されねばならず、スヌープしないコピーバック・データキャッシュは、**sync** または **dsync** が実行されると必ずフラッシュされて無効化されねばならない。

13.5 自己書き換えコード

SPU プログラムの中で、命令をローカル・ストレージに Store して、それを実行することができる。もし SPU が、Store に先行して、既にローカル・ストレージから命令の読み出しを行なっていると、新規の命令は SPU の実行にとって見えていないことになる。自己書き換えコードでは、Store されたコードを実行する前に常に **sync** 命令を実行する必要がある。**sync** 命令は、ローカル・ストレージから次の命令がフェッチされる前にすべての Store が完了することを保証する。

13.6 ローカル・ストレージへの外部アクセス

Load/Store は、必ずしもプログラム順にローカル・ストレージへアクセスするわけではない。外部デバイスからのアクセスが、プログラム順と一致しない順でインターリーブされることがある。**dsync** 命令は、すべての先行する Load/Store のローカル・ストレージ・アクセスを先に完了させてから、それ以降の Load/Store の開始を許可する。一方、**sync** は、**sync** 命令実行後に次の命令がフェッチされることを保証する。外部デバイスは、ローカル・ストレージ・アクセスを通じて SPU プログラムと同期できる。

表 13-4 は、SPU プログラムが、ローカル・ストレージのみを通じて同期を行なって外部デバイスに確実にデータを送れる方法を示したものである。この例では、SPU がアドレス C のバッファを通じて外部リーダにデータを送り、その際、ローカル・ストレージのアドレス D のマーカーを用いる。SPU はまず、転送すべきデータをストアする。そして、**dsync** 命令を実行し、マーカーをストアする前にデータがローカル・ストレージに入るようにする。**dsync** 命令はまた、マーカーのストアが並べ替えられてデータのストアの間に混じることを防ぐ。マーカーのストアの後、SPU は再度 **dsync** 命令を実行して、マーカーがローカル・ストレージに入るようにしなければならない。

表 13-5 は、ローカル・ストレージ・ベースの同期を用いて、外部ライタから SPU プログラムにデータを転送

する方法を示したものである。SPU プログラムはまず、データ・レディーを示すマーカーのポーリングを行なう。ポーリングのループには、まず **dsync** 命令があり、後続の load命令が今現在のローカル・ストレージの状態からデータを得るようにする。マーカーが見つかると、SPU プログラムは再度 **dsync** 命令を実行して、データのロードがマーカーのロードより前に遂行されることを防がねばならない。もしそのような並べ替えが生じると、マーカーの書き込みが、並べ替えられたデータのロードと遅らされたマーカーのロードの間に生じることもありうる。その場合、データのロードは古いデータを受け取ることになってしまう。

表 13-4 ローカル・ストレージを通じたデータの送付と同期

外部デバイス		SPU	コメント
		データを C に Store	
		dsync	後続の Store が C への Store の後になるようにする。つまり、マーカーが D に在るがデータがまだ C に無いというローカル・ストレージの見え方が無いようにする。
		マーカーを D に Store	
		dsync	D への Store がローカル・ストレージで外部リーダに見えるようにする。
eloop:	D の読み出し		
	If not marker, goto eloop		
	C の読み出し		

表 13-5 ローカル・ストレージを通じたデータの受け取りと同期

外部デバイス		SPU	コメント
	A にデータ書き込み		これは、外部デバイスがローカル・ストレージを変更する順序であり、この順序付けは SPU ISA で制御されるものではない。
	B にマーカー書き込み		
		loop: **dsync**	後続の Load を実際にローカル・ストレージにアクセスさせる。それにより、B から到着するロードがローカル・ストレージから新しいデータを得るようにする。
		B から Load	
		If not marker, goto loop	A と B の両方がローカル・ストレージに書き込みされたことを確実にする。
		dsync	後続の Load を B からの Load の後に実行させる。**dsync** が無いと、A からの Load が B からの Load より前に遂行されて、A への書き込みの前の内容を得てしまうことがありうる。
		A から Load	A への書き込みのデータが必ず得られるはず。

13.7 チャネル読み出し/チャネル書き込みの投機的実行と並べ替え

SPU は、チャネル読み出しやチャネル書き込みの並べ替えや投機的実行を行わない。チャネル・インターフェースにおけるすべての操作は、プログラム中で現われる順の命令によってもたらされる。

13.8 外部デバイスとのチャネル・インターフェース

チャネル・インターフェースは、チャネル読み出し/チャネル書き込みをプログラム順で SPU のインターフェースに渡すが、Load/Store に関しては順序保証はない。外部デバイスへ送ったメッセージは、外部デバイスが直接ローカル・ストレージにアクセスするトリガーとなりうる。SPU プログラムでは、**sync** または **dsync** 命令のいずれか、あるいは両方を使用して、外部からのアクセスに対して SPU Load/Store を順序付けしたいことがあるかもしれない。SPU プログラムが、チャネル・インターフェースを通じて同期を行なう外部デバイスから確実にデータを送受信する方法を*表 13-6* に示す。

表 13-6 チャネル・インターフェースを通じた同期

外部デバイス	SPU	コメント
SPU はローカル・ストレージのアドレス A を通じてデータを受けとる		
A にデータ書き込み		
チャネル B にメッセージを送信		この順序付けは、SPU ISA の制御によるものではない。
	rdch B	メッセージの待機
	dsync	A からの Load が **rdch** の後で実行され、かつ、ローカル・ストレージのデータにアクセスすることを確実にする。
	A から Load	データが必ず得られるはず
SPU はローカル・ストレージのアドレス C を通じてデータを送る		
	データを C に Store	
	dsync	データがローカル・ストレージにあることを確実にする。
	wrch D	メッセージの送信
チャネル D からメッセージを受信		
C からデータの読み出し		この順序付けは、SPU ISA の制御によるものではない。

注意： SPU アーキテクチャでは、チャネル読み出しあるいはチャネル書き込みに反応して外部デバイスが遂行できる動作を規定しない。SPU は、こうした動作の完了を待たず、また、チャネル操作の前後においてローカル・ストレージの状態の同期を行わない。

13.9 SPU プログラムがチャネル・インターフェースを通じて設定する実行状態

一部の SPU チャネルでは、SPU 実行状態のある側面を制御することができる（例：SRR0）。チャネル書き込みによる状態の変更が、後続命令に影響しないこともありうる。**sync.c** 命令を実行すれば、新しい状態が次の命令に影響を及ぼすことが保証される。

13.10 外部デバイスが設定する実行状態

外部デバイスによる実行状態の変更は、外部からリクエストされた他の状態変更に対しては順序付けされるが、SPU 命令実行に対しては順序付けされない。外部デバイスは、SPU を stop してから、実行状態の変更を生じさせ、SPU を start することで、新しい状態がプログラム実行にとって可視であることを確実に出来る。

付録 A. 命令ニーモニック順命令一覧表

表 A-1 ニーモニック順命令

Mnemonic	Instruction	Page
a	Add Word	60
absdb	Absolute Differences of Bytes	92
addx	Add Extended	66
ah	Add Halfword	58
ahi	Add Halfword Immediate	59
ai	Add Word Immediate	61
and	And	97
andbi	And Byte Immediate	99
andc	And with Complement	98
andhi	And Halfword Immediate	100
andi	And Word Immediate	101
avgb	Average Bytes	91
bg	Borrow Generate	70
bgx	Borrow Generate Extended	71
bi	Branch Indirect	178
bihnz	Branch Indirect If Not Zero Halfword	189
bihz	Branch Indirect If Zero Halfword	188
binz	Branch Indirect If Not Zero	187
bisl	Branch Indirect and Set Link	181
bisled	Branch Indirect and Set Link if External Data	180
biz	Branch Indirect If Zero	186
br	Branch Relative	174
bra	Branch Absolute	175
brasl	Branch Absolute and Set Link	177
brhnz	Branch If Not Zero Halfword	184
brhz	Branch If Zero Halfword	185
brnz	Branch If Not Zero Word	182
brsl	Branch Relative and Set Link	176
brz	Branch If Zero Word	183
cbd	Generate Controls for Byte Insertion (d-form)	40
cbx	Generate Controls for Byte Insertion (x-form)	41
cdd	Generate Controls for Doubleword Insertion (d-form)	46
cdx	Generate Controls for Doubleword Insertion (x-form)	47

Mnemonic	Instruction	Page
ceq	Compare Equal Word	160
ceqb	Compare Equal Byte	156
ceqbi	Compare Equal Byte Immediate	157
ceqh	Compare Equal Halfword	158
ceqhi	Compare Equal Halfword Immediate	159
ceqi	Compare Equal Word Immediate	161
cflts	Convert Floating to Signed Integer	221
cfltu	Convert Floating to Unsigned Integer	223
cg	Carry Generate	67
cgt	Compare Greater Than Word	166
cgtb	Compare Greater Than Byte	162
cgtbi	Compare Greater Than Byte Immediate	163
cgth	Compare Greater Than Halfword	164
cgthi	Compare Greater Than Halfword Immediate	165
cgti	Compare Greater Than Word Immediate	167
cgx	Carry Generate Extended	68
chd	Generate Controls for Halfword Insertion (d-form)	42
chx	Generate Controls for Halfword Insertion (x-form)	43
clgt	Compare Logical Greater Than Word	172
clgtb	Compare Logical Greater Than Byte	168
clgtbi	Compare Logical Greater Than Byte Immediate	169
clgth	Compare Logical Greater Than Halfword	170
clgthi	Compare Logical Greater Than Halfword Immediate	171
clgti	Compare Logical Greater Than Word Immediate	173
clz	Count Leading Zeros	83
cntb	Count Ones in Bytes	84
csflt	Convert Signed Integer to Floating	220
cuflt	Convert Unsigned Integer to Floating	222
cwd	Generate Controls for Word Insertion (d-form)	44
cwx	Generate Controls for Word Insertion (x-form)	45
dfa	Double Floating Add	203
dfceq	Double Floating Compare Equal	226
dfcgt	Double Floating Compare Greater Than	228
dfcmeq	Double Floating Compare Magnitude Equal	227
dfcmgt	Double Floating Compare Magnitude Greater Than	229
dfm	Double Floating Multiply	207

Mnemonic	Instruction	Page
dfma	Double Floating Multiply and Add	209
dfms	Double Floating Multiply and Subtract	213
dfnma	Double Floating Negative Multiply and Add	214
dfnms	Double Floating Multiply and Subtract	211
dfs	Double Floating Subtract	205
dftsv	Double Floating Test Special Value	230
dsync	Synchronize Data	243
eqv	Equivalent	114
fa	Floating Add	202
fceq	Floating Compare Equal	231
fcgt	Floating Compare Greater Than	233
fcmeq	Floating Compare Magnitude Equal	232
fcmgt	Floating Compare Magnitude Greater Than	234
fesd	Floating Extend Single to Double	225
fi	Floating Interpolate	219
fm	Floating Multiply	206
fma	Floating Multiply and Add	208
fms	Floating Multiply and Subtract	212
fnms	Floating Negative Multiply and Subtract	210
frds	Floating Round Double to Single	224
frest	Floating Reciprocal Estimate	215
frsqest	Floating Reciprocal Absolute Square Root Estimate	217
fs	Floating Subtract	204
fscrrd	Floating-Point Status and Control Register Write	236
fscrwr	Floating-Point Status and Control Register Read	235
fsm	Form Select Mask for Words	87
fsmb	Form Select Mask for Bytes	85
fsmbi	Form Select Mask for Bytes Immediate	55
fsmh	Form Select Mask for Halfwords	86
gb	Gather Bits from Words	90
gbb	Gather Bits from Bytes	88
gbh	Gather Bits from Halfwords	89
hbr	Hint for Branch (r-form)	192
hbra	Hint for Branch (a-form)	193
hbrr	Hint for Branch Relative	194
heq	Halt If Equal	150

Mnemonic	Instruction	Page
heqi	Halt If Equal Immediate	151
hgt	Halt If Greater Than	152
hgti	Halt If Greater Than Immediate	153
hlgt	Halt If Logically Greater Than	154
hlgti	Halt If Logically Greater Than Immediate	155
il	Immediate Load Word	52
ila	Immediate Load Address	53
ilh	Immediate Load Halfword	50
ilhu	Immediate Load Halfword Upper	51
iohl	Immediate Or Halfword Lower	54
iret	Interrupt Return	179
lnop	No Operation (Load)	240
lqa	Load Quadword (a-form)	34
lqd	Load Quadword (d-form)	32
lqr	Load Quadword Instruction Relative (a-form)	35
lqx	Load Quadword (x-form)	33
mfspr	Move from Special-Purpose Register	244
mpy	Multiply	72
mpya	Multiply and Add	76
mpyh	Multiply High	77
mpyhh	Multiply High High	79
mpyhha	Multiply High High and Add	80
mpyhhau	Multiply High High Unsigned and Add	82
mpyhhu	Multiply High High Unsigned	81
mpyi	Multiply Immediate	74
mpys	Multiply and Shift Right	78
mpyu	Multiply Unsigned	73
mpyui	Multiply Unsigned Immediate	75
mtspr	Move to Special-Purpose Register	245
nand	Nand	112
nop	No Operation (Execute)	241
nor	Nor	113
or	Or	102
orbi	Or Byte Immediate	104
orc	Or with Complement	103
orhi	Or Halfword Immediate	105

Mnemonic	Instruction	Page
ori	Or Word Immediate	106
orx	Or Across	107
rchcnt	Read Channel Count	249
rdch	Read Channel	248
rot	Rotate Word	129
roth	Rotate Halfword	127
rothi	Rotate Halfword Immediate	128
rothm	Rotate and Mask Halfword	136
rothmi	Rotate and Mask Halfword Immediate	137
roti	Rotate Word Immediate	130
rotm	Rotate and Mask Word	138
rotma	Rotate and Mask Algebraic Word	147
rotmah	Rotate and Mask Algebraic Halfword	145
rotmahi	Rotate and Mask Algebraic Halfword Immediate	146
rotmai	Rotate and Mask Algebraic Word Immediate	148
rotmi	Rotate and Mask Word Immediate	139
rotqbi	Rotate Quadword by Bits	134
rotqbii	Rotate Quadword by Bits Immediate	135
rotqby	Rotate Quadword by Bytes	131
rotqbybi	Rotate Quadword by Bytes from Bit Shift Count	133
rotqbyi	Rotate Quadword by Bytes Immediate	132
rotqmbi	Rotate and Mask Quadword by Bits	143
rotqmbii	Rotate and Mask Quadword by Bits Immediate	144
rotqmby	Rotate and Mask Quadword by Bytes	140
rotqmbybi	Rotate and Mask Quadword Bytes from Bit Shift Count	142
rotqmbyi	Rotate and Mask Quadword by Bytes Immediate	141
selb	Select Bits	115
sf	Subtract From Word	64
sfh	Subtract From Halfword	62
sfhi	Subtract From Halfword Immediate	63
sfi	Subtract From Word Immediate	65
sfx	Subtract From Extended	69
shl	Shift Left Word	120
shlh	Shift Left Halfword	118
shlhi	Shift Left Halfword Immediate	119
shli	Shift Left Word Immediate	121

Mnemonic	Instruction	Page
shlqbi	Shift Left Quadword by Bits	122
shlqbii	Shift Left Quadword by Bits Immediate	123
shlqby	Shift Left Quadword by Bytes	124
shlqbybi	Shift Left Quadword by Bytes from Bit Shift Count	126
shlqbyi	Shift Left Quadword by Bytes Immediate	125
shufb	Shuffle Bytes	116
stop	Stop and Signal	238
stopd	Stop and Signal with Dependencies	239
stqa	Store Quadword (a-form)	38
stqd	Store Quadword (d-form)	36
stqr	Store Quadword Instruction Relative (a-form)	39
stqx	Store Quadword (x-form)	37
sumb	Sum Bytes into Halfwords	93
sync	Synchronize	242
wrch	Write Channel	250
xor	Exclusive Or	108
xorbi	Exclusive Or Byte Immediate	109
xorhi	Exclusive Or Halfword Immediate	110
xori	Exclusive Or Word Immediate	111
xsbh	Extend Sign Byte to Halfword	94
xshw	Extend Sign Halfword to Word	95
xswd	Extend Sign Word to Doubleword	96

付録 B. 制御生成命令の詳細

本セクションに記載する表は、8 種の制御生成命令によって生成されるマスクの詳細を示す。示されるこれらのマスクは、Shuffle Bytes すなわち **shufb** 命令の RC オペランドとして使用することを意図している。表の各行は、実効アドレスの右端 4 bit を示す。表の左側の列中の x は、無視されるビットを示す。「生成されるマスク」の中の空白スペースは、わかりやすくするためにのみ示されるものである。

以下の表の当てはまるものを参照のこと。

- バイト挿入については、*表 B-1 バイト挿入 : 実効アドレスの右端 4 bit および生成されるマスク*（265 ページ）を参照。
- ハーフワード挿入については、表 B-2 *ハーフワード挿入 : 実効アドレスの右端 4 bit および生成されるマスク*（266 ページ）を参照。
- ワード挿入については、*表 B-3 ワード挿入 : 実効アドレスの右端 4 bit および生成されるマスク*（266 ページ）を参照。
- ダブルワード挿入については、*表 B-4 ダブルワード挿入 : 実効アドレスの右端 4 bit および生成されるマスク*（266 ページ）を参照。

表 B-1　バイト挿入 : 実効アドレスの右端 4 bit および生成されるマスク

実効アドレスの右端 4 bit	生成されるマスク
0000	03 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
0001	10 03 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
0010	10 11 03 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
0011	10 11 12 03 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
0100	10 11 12 13 03 15 16 17 18 19 1a 1b 1c 1d 1e 1f
0101	10 11 12 13 14 03 16 17 18 19 1a 1b 1c 1d 1e 1f
0110	10 11 12 13 14 15 03 17 18 19 1a 1b 1c 1d 1e 1f
0111	10 11 12 13 14 15 16 03 18 19 1a 1b 1c 1d 1e 1f
1000	10 11 12 13 14 15 16 17 03 19 1a 1b 1c 1d 1e 1f
1001	10 11 12 13 14 15 16 17 18 03 1a 1b 1c 1d 1e 1f
1010	10 11 12 13 14 15 16 17 18 19 03 1b 1c 1d 1e 1f
1011	10 11 12 13 14 15 16 17 18 19 1a 03 1c 1d 1e 1f
1100	10 11 12 13 14 15 16 17 18 19 1a 1b 03 1d 1e 1f
1101	10 11 12 13 14 15 16 17 18 19 1a 1b 1c 03 1e 1f
1110	10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 03 1f
1111	10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 03

表 B-2　ハーフワード挿入：実効アドレスの右端 4 bit および生成されるマスク

実効アドレスの右端 4 bit	生成されるマスク
000x	0203 1213 1415 1617 1819 1a1b 1c1d 1e1f
001x	1011 0203 1415 1617 1819 1a1b 1c1d 1e1f
010x	1011 1213 0203 1617 1819 1a1b 1c1d 1e1f
011x	1011 1213 1415 0203 1819 1a1b 1c1d 1e1f
100x	1011 1213 1415 1617 0203 1a1b 1c1d 1e1f
101x	1011 1213 1415 1617 1819 0203 1c1d 1e1f
110x	1011 1213 1415 1617 1819 1a1b 0203 1e1f
111x	1011 1213 1415 1617 1819 1a1b 1c1d 0203

表 B-3　ワード挿入：実効アドレスの右端 4 bit および生成されるマスク

実効アドレスの右端 4 bit	生成されるマスク
00xx	00010203 14151617 18191a1b 1c1d1e1f
01xx	10111213 00010203 18191a1b 1c1d1e1f
10xx	10111213 14151617 00010203 1c1d1e1f
11xx	10111213 14151617 18191a1b 00010203

表 B-4　ダブルワード挿入：実効アドレスの右端 4 bit および生成されるマスク

実効アドレスの右端 4 bit	生成されるマスク
0xxx	0001020304050607 18191a1b1c1d1e1f
1xxx	1011121314151617 0001020304050607

用語集

bisled	branch indirect and set link if external data 命令
CBEA	Cell Broadband Engine Architecture 参照。
Cell Broadband Engine アーキテクチャ Cell Broadband Engine Architecture	Cell Broadband Engine アーキテクチャは、疎結合の協調負荷代行プロセッサ群でもって PowerPC 64-bit アーキテクチャを拡張する。Cell Broadband Engine アーキテクチャは、ゲーム、マルチメディア、リアルタイムなどのマーケットセグメントを対象とするマイクロプロセッサの開発のための基礎を提供する。 Cell Broadband Engine は、Cell Broadband Engine アーキテクチャの一実装である。
DBZ	Divide by zero.
DIFF	IEEE 非準拠結果
DMA	Direct Memory Access。専用のコントローラを用いて、メモリや I/O の転送の転送元と転送先のアドレスを生成する技法。
FPU	Floating-Point Unit。浮動小数点ユニット
fscrrd	Floating-Point Status and Control Register read 命令
fscrwr	Floating-Point Status and Control Register write 命令
GPR	汎用レジスタ（General-Purpose Register）参照。
Inf	Infinity.
INV	Invalid operation.
INX	Inexact result.
iohl	Immediate or halfword lower 命令
iret	interrupt return 命令
ISA	Instruction Set Architecture。命令セット・アーキテクチャ
KB	Kilobyte.
LSA	Local Storage Address。SPU の LS 内のアドレスであり、それによって、SPU 内で動作するプログラムや MFC で管理される DMA 転送が LS にアクセスする。
LSb	最下位ビット（least significant bit）参照。
MFC	Memory Flow Controller。SPE の一部分であり、２つの主たる機能を提供する。すなわち、DMA を用いて SPE のローカルストレージ（LS）とメインストレージとの間でデータを転送すること、および、SPU をシステム内の他の処理ユニットと同期させること。
mfspr	Move from special-purpose register 命令
MSb	最上位ビット（most significant bit）参照。
MSB	最上位バイト（most significant byte）参照。
MSR	Machine state register.

mtspr	Move to special-purpose register 命令
NaN	Not a number
OVF	Overflow
PC	program counter.
PowerPC	PowerPC アーキテクチャの（に関する）、あるいは、そのアーキテクチャを実装したマイクロプロセッサの（に関する）、の意。
PowerPC アーキテクチャ PowerPC Architecture	第 3 世代の RISC（Reduced Instruction Set Computer）プロセッサに基くコンピュータアーキテクチャの一つ。 PowerPC アーキテクチャは、Apple、Motorola、IBM により共同開発された。
PPE	PowerPC Processor Element。Cell Broadband Engine 中の汎用プロセッサ。
QNaN	Quiet NaN.
rchcnt	Read channel counter 命令
rdch	Read from channel 命令
RN0	2-way SIMD 倍精度演算のスライス 0 用の丸め制御。
RN1	2-way SIMD 倍精度演算のスライス 1 用の丸め制御。
RO	relative offset
ROH	relative offset high
ROL	relative offset low
RTL	register transfer language
shufb	shuffle bytes 命令
SIMD	Single Instruction, Multiple Data。 単一命令が、ベクトルデータ型を構成する複数データ要素に対して演算を行なう処理。ベクトル処理と呼ばれることもある。 このプログラミングスタイルによりデータレベルの並列性が実現される。
SNaN	Signalling NaN.
SPR	Special-purpose register.
SPU	Synergistic Processor Unit。SPE の一部分であって、ローカルストレージ（LS）からの命令を実行する部分。
SRAM	static random access memory
sync	Synchronize 命令
UNF	Underflow
wrch	Write to channel 命令
アーキテクチャ architecture	プロセッサまたはコンピュータシステムについての要件の詳細仕様。 プロセッサまたはコンピュータシステムをどのように実装しなければならないかの詳細を規定するのではなく、その代わりに、互換な実装のファミリーのための雛型を提供する。

ガーディッド guarded	投機的なロードや命令フェッチへの応答が防止されていること。 オペレーティングシステムは通常、例えば全ての I/O デバイスに対してガーディッド属性を施す。
キャッシュ cache	プロセッサ近傍の高速メモリ。キャッシュには通常、最近アクセスしたデータや命令が入っている。しかし、ある種のキャッシュ制御命令を用いれば、データや命令のキャッシュ状態をロックしたり、追い出したり、その他の変更を加えることができる。
最下位バイト least significant byte	アドレス、レジスタ、データエレメント、命令エンコーディングなどにおける最も小さい値のバイト。
最下位ビット least significant bit	アドレス、レジスタ、データエレメント、命令エンコーディングなどにおける最も小さい値のビット。
最上位バイト most significant byte	アドレス、レジスタ、データエレメント、命令エンコーディングなどにおける最も高位のバイト。
最上位ビット most significant bit	アドレス、レジスタ、データエレメント、命令エンコーディングなどにおける最も高位のビット。
シグナル signal	シグナル通知チャネルで送られる情報。これらチャネルは（SPE への）入力方向のレジスタである。PPE や他のプロセッサが SPE に情報を送るために使用される。各 SPE には、32 bit のシグナル通知レジスタが 2 個あるが、それぞれに対応してメモリマップド I/O（MMIO）レジスタがあって、送り手のプロセッサはシグナル通知データをそこに書き込む。 メールボックスとは異なり、1 対 1 用あるいは多対多用のいずれかに構成設定が可能である。
実効アドレス effective address	プログラムがメモリを参照するために生成あるいは使用されるアドレス。 メモリ管理ユニットは、実行アドレス（EA）を仮想アドレス（VA）に変換し、それをさらにリアルアドレス（RA）に変換し、RA がリアル（物理）メモリをアクセスする。実効アドレス空間の最大サイズは 2^{64} バイトである。
実装 implementation	アーキテクチャに適合する特定のプロセッサ。しかし、それは、アーキテクチャ適合の他の実装とは、例えば、設計、機能セット、オプション機能の実装などが異なっている場合がある。
スヌープ snoop	バス上のアドレスをキャッシュ内のタグと比較し、それにより、メモリコヒーレンシを破るオペレーションを検出すること。
チャネル channel	チャネルは、単方向で機能特有のレジスタあるいはキューである。SPE の SPU と MFC の間の主な通信手段であり、次に MFC は PPE、他の SPE、その他のデバイスとの通信を仲介する。これら他のデバイスは、宛先 SPE 内の MMIO レジスタを用いて、その宛先 SPE のチャネル・インターフェースに情報を転送する。 特定のチャネルには、読み出しまたは書き込みの属性と、ブロッキングまたはノンブロッキングの属性がある。SPU 上のソフトウェアはチャネルコマンドを用いて、DMA コマンドのエンキュー、DMA およびプロセッサの状態の照会、MFC 同期の遂行、デクリメンタ（タイマー）などの補助リソースのアクセス、メールボックスやシグナル通知を介したプロセッサ間通信などを行なう。
同期 synchronization	ストレージ・オペレーションを、発生した順序で完了するように、整える処理。

倍精度 double precision	浮動小数点数値を長形式（コンピュータの 2 ワード）で（内部的に）格納させるようにする仕様。
汎用レジスタ General-Purpose Register	明示的にアドレス可能なレジスタであって、多様な目的（例えば、アキュムレータやインデックスレジスタなど）に使用できるもの。
ビッグエンディアン big-endian	メモリ内のバイトの順番の付け方で、あるワードのアドレス n が最上位バイトに対応するもの。アドレス指定されたワードの中で、各バイトは（左から右に）0, 1, 2, 3 と順番が付けられ、0 が最上位バイトとなる。 リトルエンディアン参照。
フェッチ fetch	キャッシュあるいはシステムメモリのいずれかから命令を取り出してそれを命令キューに置く動作。
浮動小数点 floating point	実数（すなわち分数や小数を伴う数値）を 32 bit あるいは 64 bit で表わす方法。浮動小数点表示は、非常に小さい数や非常に大きい数の記述に有用である。
ベクトル vector	1 次元配列に詰められたデータ要素の組を含んだ命令オペランド。その要素は、固定小数点値あるいは浮動小数点値をとりうる。 大部分の Vector/SIMD Multimedia Extension 命令および SPU SIMD 命令は、ベクトルオペランドに対して演算を行なう。 ベクトルは、SIMD オペランドあるいはパックドオペランドと呼ばれることもある。
マスク mask	ビットのパターンであって、もう一つのデータの組の中のビットパターンを受理あるいは拒絶するために用いる。 ハードウェア割り込みは、ビットの列をセットあるいはクリアすることで許可あるいは禁止され、各割り込みには、マスクレジスタ中でビットポジションが割り当てられている。
命令キャッシュ instruction cache	プログラムの命令をメインメモリから得られるよりも高速にプロセッサに提供するキャッシュ。
リトリエンディアン little-endian	メモリ内のバイトの順番の付け方で、あるワードのアドレス n が最下位バイトに対応するもの。アドレス指定されたワードの中で、各バイトは（左から右に）3, 2, 1, 0 と順番が付けられ、3 が最上位バイトとなる。 ビッグエンディアン参照。
例外 exception	ステータスビットを変更する可能性があり、対応する割り込みが許可されている場合には割り込みを生じることになる、エラー、異常状態、あるいは外部シグナル。
ローカル・ストレージ local storage	各 SPE に付随するストレージ。命令とデータの両方を保持する。
ワード word	4 バイト

索引

D

E

F

G

H

I

L

M

N

O

P

R

S

U

W

X

あ

い

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

と

な

に

ね

は

ひ

ふ

へ

ほ

ま

む

め

も

よ

る

れ

ろ

わ