SONY

Cell Broadband Engine™アーキテクチャ用 C/C++ 言語拡張

Version 2.3

CBEA JSRE Series
Cell Broadband Engine Architecture
Joint Software Reference
Environment Series

2006 年 12 月 4 日

本資料の記載内容は、予告なく変更されることがあります。本資料記載の製品は、不具合により死亡、人身傷害、重大な物損がもたらされ得る、たとえば、体内埋込機器、生命維持装置、その他の危険を伴う用途の応用例に使用することを意図したものではありません。本資料の記載内容は、ソニー株式会社（以下 ソニー）および株式会社ソニー・コンピュータエンタテインメント（以下 SCEI）の製品の仕様もしくは保証に影響を及ぼすものではありません。また、本資料は、知的財産権の使用許諾や権利侵害に対する補償を意味するものではありません。本資料の記載内容は、特定の環境において取得され、説明目的で提示されるものです。動作環境が異なると結果も異なる場合があります。

本資料の記載内容は、現状有姿で提供されるものです。ソニーおよび SCEIは、法令により免責が認められない場合を除き、本資料の記載内容の使用により生じる損害につき一切責任を負いません。

本資料は、英語原文を日本語に翻訳したものです。ソニーおよび SCEIは、翻訳結果の正確性、信頼性に関し、一切保証いたしません。
本資料を使用する際には、最新版であることを確認の上、ご使用願います。最新版は、下記 Cell Broadband Engine のホームページより入手できます。

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**

2006年 12月 4日

SONY

目次

SONY

表目次

図目次

本ドキュメントについて

本ドキュメントは、C/C++などの高級言語からは容易にアクセスできないハードウェア機能をソフトウェア開発者がアクセスできるようにする言語拡張仕様を記述し、それによって、Cell Broadband Engine™ (CBE) の Synergistic Processor Unit (SPU) と Power Processing Unit (PPU) の性能が最大限に引き出せるようにします。また、本ドキュメントは、SPU と PPU 間の通信を容易にする関数の仕様を含んでおり、標準 SPU プログラミング環境の一部として提供しなければならない標準ライブラリの最小セットを列挙しています。

対象者

本ドキュメントは、CBEA に準拠したプロセッサ用の SPU および PPU プログラムを書くシステムおよびアプリケーションのプログラマを対象としています。

変更履歴

本セクションでは、本ドキュメントに関する重要な変更点をバージョンごとにまとめてあります。

バージョン番号および日付	変更点
v. 2.3 2006 年 12 月 4 日	PPU `__stwbrx` 組み込み関数の関数パラメタ順序を訂正。(TWG_RFC00074-0: CORRECTION NOTICE) `signed/unsigned char` のベクタの初期化に用いる要素初期化子の型を訂正。(TWG_RFC00075-0: CORRECTION NOTICE) 倍精度縮約演算の使用は、`FP_CONTRACT` プラグマあるいは no-fast-double コンパイラオプションによって禁止されない限り、デフォルトで許可されている旨、注記を変更。(TWG_RFC00076-0) 1 章に PPU のデータ型とプログラム指示文を追加し、章の表題を変更。(TWG_RFC00077-1) `__fre`, `__frsqrtes`, `__popcntb` 組み込み関数を削除し、 `__frsqrte` 組み込み関数を追加。(TWG_RFC00078-3) 浮動小数点環境のサポートが 2 個の倍精度要素の両方と 4 個の単精度要素のすべてに対して提供されている旨を追記。さらに、FLT_ROUNDS の情報を更新。(TWG_RFC00079-1) 新章「PPU VMX 組み込み関数」を追加し、基盤となる PPU VMX 命令セットを C 言語からアクセス可能にする組み込み関数セットを規定。(TWG_RFC00081-1 and TWG_RFC00092-0) PPU 組み込み関数に 32 ビット ABI サポートを追加。一貫した高レベルインターフェースの提供のために関数引数を変更。いくつかの誤記を訂正。 (TWG_RFC00083-1) `__fctiw` および `__fctiwx` PPU 組み込み関数の戻り値の型を変更。これらおよび類似の変換用組み込み関数の記述名称を変更。`__stfiwx` 組み込み関数を削除。(TWG_RFC00089-1) 廃止された PPU VMX 演算およびその代替に推奨する適切な PPU 組み込み関数を明記。 (TWG_RFC00090-0) 非サポートの言語機能を特定し、C++ 例外処理が SPU で非サポートの旨を規定。(TWG_RFC00091-0) 以下の訂正を適用: TWG_RFC00086-0 and TWG_RFC00087-0
v. 2.2 2006 年 10 月 11 日	以下の要求で行なわれる変更を適用: TWG_RFC00056-0, TWG_RFC00057-0, TWG_RFC00058-2, TWG_RFC00061-1, TWG_RFC00060-1, TWG_RFC00062-0, TWG_RFC00066-2, TWG_RFC00067-2, TWG_RFC00068-0, TWG_RFC00070-1, TWG_RFC00072-0, and TWG_RFC00073-0

バージョン番号および日付	変更点
	内容が SPU に限定されなくなったため、ドキュメント表題を変更。それに伴い「本ドキュメントについて」と「対象者」を変更。TWG_RFC00053-0, TWG_RFC00054-1, TWG_RFC00055-0 を適用。 TWG_RFC00050-1 および TWG_RFC00052-0 に従い、商標名称の使用を *Altivec Technology Programming Interface Manual* への参照に変更。 `spu_sub`（ベクタ減算用の算術組み込み関数）に関連するいくつかのオペランドの誤りを訂正。(TWG_RFC00046-0: CORRECTION NOTICE) 様々な誤記を訂正:（例）longjmp 関数に伴って生じる Stack Pointer Information レジスタのリストア方法を例示するサンプルコードを変更。(TWG_RFC00047-0: CORRECTION NOTICE) ベクタリテラル用の代替形式は必須ではなくオプションである旨を規定。(TWG_RFC00050).
v. 2.1 2005 年 10 月 20 日	「C/C++標準ライブラリ」の章内のセクション「C 標準ライブラリ」下に、サブセクション「malloc()」を追加。このサブセクションの記述はメモリヒープ初期化およびスタック管理の標準プロセスを定義する試みに関するものである。(TWG_RFC00024-3) 「SPU 組み込み関数と Vector Multimedia Extension 組み込み関数」の章内で、組み込み関数のマッピングについて、本仕様で要求されるものと容易な方法が存在しないために要求されないものの区別を明確化し、マッピングが困難な組み込み関数についての補足説明を提供。 (TWG_RFC00034-1: CORRECTION NOTICE). `si_stqx` 命令の記述を修正。(TWG_RFC00035-0: CORRECTION NOTICE). 様々な誤記を修正。例：「ベクタリテラルの代替形式と説明」の表中の記述を変更。(TWG_RFC00036-0: CORRECTION NOTICE, TWG_RFC00041-0: CORRECTION NOTICE, TWG_RFC00045-0: CORRECTION NOTICE). 「Broadband Processor Architecture」を「Cell Broad Engine Architecture」へ、「BPA」を「CBEA」へ変更。(TWG_RFC00037-0: CORRECTION NOTICE). いくつかの BE のバージョン DD1.0 や DD2.0 についての言及を削除。(TWG_RFC00040-0: CORRECTION NOTICE). MFC I/O 組み込み関数について記述した章を追加。これらの組み込み関数は、使用頻度の高いユティリティ関数をまとめて定義することにより MFC のプログラミングに有用。(TWG_RFC00043-2).
v. 2.0 2005 年 7 月 11 日	「本ドキュメントについて」のセクション内のいくつかの項目を削除。 Write Word Channel の表内で「si_wrch(channel,si_to_int(a))」を「si_wrch(channel,si_from_int(a))」へ変更。 ベクタ型指定子はベクタ型のシンタックスでは型指定子として typedef で定義した名称を使用することができないことを説明。 (以上すべて TWG_RFC00032-0: CORRECTION NOTICE による)
v. 1.9 2005 年 6 月 10 日	C/C++ライブラリについて説明する章を追加。 (TWG_RFC00018-5) SPU 浮動小数点演算について説明する章を追加。 (TWG_RFC00027-1) 「Broadband Engine」および「BE」を「Broadband Processor Architecture（BPA）に準拠したプロセッサ」へ変更。「VMX」を「Vector Multimedia Extension」へ変更。「Synergistic Processing Element」を「Synergistic Processor Element」へ変更。「Synergistic Processing Unit」を「Synergistic Processor Unit」へ変更。PPU を初出時あるいは要所において「PowerPC Processor Unit」として定義。文書の参照文を修正。商標所有者を記載するために版権ページを追加。(以上すべて TWG_RFC00031-0:CORRECTION NOTICE による) 「本ドキュメントについて」のセクションにおけるその他の変更。

バージョン番号および日付	変更点
v. 1.8 2005 年 5 月 12 日	マルチソース同期リクエスト用に新たなチャネル番号を追加。 (TWG_RFC00023-1) アラインメントが正しくないベクタのロード例を修正。 「PU」を「PPU」へ、「SPC」を「SPE」へ、「PU-to-SPU」（メールボックス）および「SPU-to-PU」をそれぞれ「inbound」および「outbound」へ変更。(TWG_RFC00028-1: CORRECTION NOTICE) 「spu_mulhh」を「spu_mule」へ名称変更。 (TWG_RFC00021-0) BPA のチャネル名に合わせチャネル名を更新。 (TWG_RFC00029-1)
v. 1.7 2004 年 7 月 16 日	チャネル組み込み関数の順序が他のチャネルコマンドや揮発性 LS アクセスと入れ替わることがないことを明記。(TWG_RFC00007-1) 仕様に準拠したコンパイラが __align_hint 組み込み関数を無視する可能性があることを警告。(TWG_RFC00008-1) SPU 命令「orx」を追加。(TWG_RFC00010-0) イベントマスクやタグマスクの読み込みをサポートするチャネルのニーモニックを追加。(TWG_RFC00011-0) 組み込み関数 spu_ienable および spu_idisable は返り値を持たないことを明記。(TWG_RFC00013-0) 「この組み込み関数は浮動小数点命令に対して揮発性を持つとみなされるため、」で始まる段落を spu_mfspr のセクションより spu_mtfpscr のセクションへ移動。（TWG_RFC00014-0) 組み込み関数 si_lqd および si_stqd についての記述を変更。(TWG_RFC00015-1) 以下の各種ローテート・マスク組み込み関数についての記述を追加。 spu_rlmask、spu_rlmaska、spu_rlmaskqw、spu_rlmaskqwbyte、spu_rlmaskqwbytebc これらの記述には擬似コードを含む。(TWG_RFC00016-1) その他編集上の変更。
v. 1.6 2004 年 3 月 12 日	編集上の変更。
v. 1.5 2004 年 2 月 25 日	xxページに記載されている書体の表記上の規則を反映するためにフォーマットを変更。その他編集上の変更。 spu_mfcdma32 および spu_mfcdma64 について、パラメータの型の一部を変更。（TWG_RFC00002） ベクタリテラル形式用に新たに仕様を追加。（TWG_RFC00003）
v. 1.4 2004 年 1 月 20 日	前付け部も含め文書のフォーマットを変更。その他編集上の変更。
v. 1.3 2003 年 11 月 4 日	割り込みを許可・禁止するための組み込み関数を追加。
v. 1.2 2003 年 9 月 2 日	spu_sel 組み込み関数のパラメータ型を VMX の vec_sel 組み込み関数との互換性のために変更。 si_stopd 個別組み込み関数を追加。 総称組み込み関数 spu_genb および spu_genc の表を修正。
v. 1.1 2003 年 6 月 15 日	RFC24 をサポートするための変更。隔離制御チャネル 64 を追加。 RFC33 をサポートするための変更。spu_addc、spu_addsc、spu_subb、spu_subsb を削除。spu_addx、spu_subx、spu_genc、spu_gencx、spu_genb、spu_genbx を追加。
v. 1.0 2003 年 4 月 28 日	細かな修正。

バージョン番号および日付	変更点
v. 0.9 2003 年 3 月 7 日	新規に追加されたまたは変更のあった命令をサポートするために新たに組み込み関数を追加。追加された関数：fscrrd、 fscrwr、 stop、 dfma、 mpyhhau、 mpyhhu、 rotqmbybi、 iret、 lqr、 stqr また命令 iret、 bisled、 bihnz、 sync の新たに機能が追加されたビットをサポートするための組み込み関数を追加。
v. 0.8 2003 年 1 月 23 日	個別組み込み関数についての記述を改善。パラメータ順序および即値のサイズを完全に定義。 ヘッダファイル spu_intrinsics.h（グローバル）および spu_internals.h（コンパイラ用）を新たに定義。単一トークンのベクタ型およびチャネル列挙子は spu_intrinsics.h 内で宣言されていることを明記。 ポインタキャスト用の個別組み込み関数を追加。 標準化された条件付きコンパイル制御__SPU__を追加。 変換用の個別組み込み関数を総称組み込み関数のようなバイアスなしのパラメータへ変更。揮発性レジスタに対するターゲット関数 bisled の振る舞いが標準呼び出し規約に則っていないことを明記。
v. 0.7 2002 年 11 月 18 日	gcc 形式のインラインアセンブリが必要であることを明記。 __builtin_expect が必要であることを明記。 bisled に対応する個別組み込み関数と総称組み込み関数を追加。 __align_hint 組み込み関数を追加。 restrict 型の修飾子が必要であることを明記。 変換用の総称組み込み関数へ定義域外の換算係数を渡すとエラーが返ることを明記。
v. 0.6 2002 年 9 月 24 日	文書タイトルへ「C++」を追加。 さまざまな説明の追加およびタイプミスの修正。 spu_eqv について入力したものと同じベクタ型を返すように変更。 spu_and、 spu_or、 spu_xor についてパラメータ *a* の要素と同じ型の即値を受け付けるように変更。 キャスト用個別組み込み関数を追加。 個別組み込み関数へ定義域外の即値を渡した場合のデフォルトの動作をエラーの発行へ変更。 __builtin_expect ビルトイン関数についての記述を追加。 「SPU 組み込み関数を VMX 組み込み関数へマップ」のセクションを完成。
v. 0.5 2002 年 8 月 27 日	「VMX 組み込み関数を SPU 組み込み関数へマップ」の記述セクションを編集。 付録部分を削除。 32 ビット読み書きチャネル用組み込み関数のサポートを追加。クワッドワードチャネルの読み書き用関数名を readchqw および writechqw へ変更。
v. 0.4 2002 年 8 月 5 日	spu_promote および spu_extract への命令のマッピングを修正。 総称組み込み関数 spu_re および spu_rsqrte のマッピング先は FI (floating-point interpolate)命令を含むことを明記。 vec_splat との混同を避けるため spu_splat を spu_splats (scalar splat) へ名称を変更。 即値形式組み込み関数のサイズについての記述を追加。 vector signed long を全て vector signed long long へ変更。 spu_sl、spu_slqw、spu_slqwbyte、spu_slqwbytebc について *count* を unsigned へ変更。 spu_rl、spu_rlmask、spu_rlmaska について *count* を signed へ変更。

バージョン番号および日付	変更点
	spu_cntlz の返り値が unsigned 型であることを明記。 spu_gather 組み込み関数の記述を修正。 spu_and、spu_or、spu_xor のスカラについてマッピングの記述を編集。 spu_hcmpeq および spu_hcmpgt のベクタ入力形式を削除。
v. 0.3 2002 年 7 月 16 日	fsmbi をリテラル生成命令の一つとして追加。spu_maskb 組み込み関数へ fsmbi (即値形式)を追加。 比較・停止組み込み関数（spu_hcmpeq、spu_hcmpgt）へベクタ形式のものを追加。 個別組み込み関数が受け付ける唯一のベクタ型として qword データ型を追加。 コード移植用の基本的型としてベクタ型の形定義を追加。 各種 spu_splat 総称組み込み関数を単一の組み込み関数へ統合。 総称組み込み関数のうち spu_load、spu_store、spu_insertctl を削除。
v. 0.2 2002 年 1 月 9 日	Peng からの変更依頼や提案を反映。 vector long 型を全て vector long long へ変更。
v. 0.1 2002 年 1 月 21 日	初版。Tobey コンパイラ組み込み関数の仕様書をベースとして作成。

関連ドキュメント

以下の表は、本ドキュメントの参考文献および資料の一覧です。

ドキュメント名	バージョン	日付
ISO/IEC Standard 9899:1999 (C Standard)		
ISO/IEC Standard14882:1998 (C++ Standard)		
IEEE-754 (Standard for Binary Floating-Point Arithmetic)		
Synergistic Processor Unit 命令セット・アーキテクチャ	1.11	2006 年 10 月
Cell Broadband Engine™ アーキテクチャ	1.01	2006 年 10 月
Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification	1.2	2005 年 5 月
*Tool Interface Standard (TIS), DWARF Debugging I*nformation Format Specification	2.0	2005 年 5 月
PowerPC Architecture Book, Book II: PowerPC Virtual Environment Architecture	2.02	2005 年 1 月

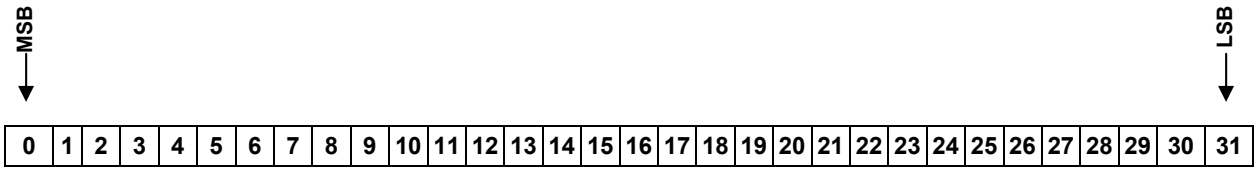
本ドキュメントの構成

本ドキュメントは、主に以下のセクションを含みます。

1. データ型とプログラム指示文
2. SPU低レベル個別・総称組み込み関数
3. 複合組み込み関数
4. MFC入出力のプログラミングサポート
5. SPU組み込み関数とVector Multimedia Extension組み込み関数
6. PPU VMX組み込み関数
7. PPU組み込み関数
8. SPU C/C++標準ライブラリおよび言語サポート
9. SPU上の浮動小数点演算

ビット表記について

本ドキュメントでは、標準ビット表記を採用しています。ビット番号およびバイト番号は、左から右へ昇順に並んでいます。従って 4 バイトワードの場合、以下のようにビット 0 が MSB（最上位ビット）でビット 31 が LSB（最下位ビット）になります。



MSB = Most significant ビット（最上位ビット）

LSB = Least significant ビット（最下位ビット）

ビットエンコーディングの表記法は以下の通りです。

- 16 進数の値の前には、`0x` が付いています。例：`0x0A00`
- 文中に出てくるバイナリ（2 進数）は、引用符（’）で囲んでいます。例：‘1010’

バイト順序と要素の番号付け

図 1-1に示すように、バイト順序と要素の番号付けは常にビッグエンディアン順序で示されます。

図 1-1: ベクタ型のバイトと要素のビッグエンディアン順序

Byte 0 (MSB)	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15 (LSB)
doubleword 0								*doubleword 1*							
word 0				*word 1*				*word 2*				*word 3*			
halfword 0		*halfword 1*		*halfword 2*		*halfword 3*		*halfword 4*		*halfword 5*		*halfword 6*		*halfword 7*	
char 0	*char 1*	*char 2*	*char 3*	*char 4*	*char 5*	*char 6*	*char 7*	*char 8*	*char 9*	*char 10*	*char 11*	*char 12*	*char 13*	*char 14*	*char 15*

書体表記の規約

本ドキュメントでは、ビット表記に加え、以下の書体表記の規約を採用しています。

書体	意味
`courier`	プログラミングコード、処理命令、レジスタ名、データ型、イベント、ファイル名、および他のリテラルを表します。関数名およびマクロ名を表す場合にも使用されます。この書体は、理解に役立つ場合にのみ使用され、特に説明文中で使用されます。
`courier + イタリック体`	引数、パラメータ、および変数（`const` 型の変数を含む）を表します。この書体は、理解に役立つ場合にのみ使用され、特に説明文中で使用されます。

書体	意味
イタリック体（courier なし）	強調を表します。ハイパーリンクの場合を除いて、文献の参照にはイタリック体が使用されます。言葉を初めて定義する場合、イタリック体を使用することが多々あります。
青	ハイパーリンクを表します（カラープリンタまたはオンライン専用）。

1. データ型とプログラム指示文

本章では、PPU の Vector Multimedia eXtension™ (VMX) と SPU とのベクタデータ型、そのデータ型に対する演算、プログラミング指示文、および、あらかじめ定義されたマクロのターゲット定義について述べています。

PPU の Vector Multimedia eXtension (VMX) のデータ型についてここで述べている要件と Altivec Technology Programming Interface Manual とに相違があるならば、それは意図的なものではありません。

1.1. データ型

基本的なベクタデータ型一式をC言語に導入します。表 1-1に、これらのデータ型と、それがPPU、SPU、両方のいずれでサポートされているかを示します。これらのデータ型はすべて 128 ビット長であり、要素のデータ型に応じて 2 個から 16 個の要素を含みます。

表 1-1: ベクタデータ型

ベクタデータ型	内容	SPU/PPU
vector unsigned char	16 個の 8 ビット符号なし char	両方
vector signed char	16 個の 8 ビット符号付き char	両方
vector unsigned short	8 個の 16 ビット符号なし halfword	両方
vector signed short	8 個の 16 ビット符号付き halfword	両方
vector unsigned int	4 個の 32 ビット符号なし word	両方
vector signed int	4 個の 32 ビット符号付き word	両方
vector unsigned long long	2 個の 64 ビット符号なし doubleword	SPU
vector signed long long	2 個の 64 ビット符号付き doubleword	SPU
vector float	4 個の 32 ビット単精度 float	両方
vector double	2 個の 64 ビット倍精度 float	SPU
qword	個別組み込み関数への入出力のみで使用する特殊なクワッドワード（16 バイト）であり、2.1. 個別組み込み関数で説明しています。	SPU
vector bool char	16 個の 8 ビット bool – 0 (false) 255 (true)	PPU
vector bool short	8 個の 16 ビット bool – 0 (false) 65535 (true)	PPU
vector bool int	4 個の 32 ビット bool – 0 (false) $2^{32} - 1$ (true)	PPU
vector pixel	8 個の 16 ビット符号なし halfword, 1/5/5/5 pixel	PPU

ベクタ型のシンタックスでは型の指定子として typedef で定義した名称を使用することはできません。例えば以下のような宣言は認められていません。

```
typedef signed short int16;
vector int16 data;
```

1.1.1. PPUデータ型からSPUデータ型へのマッピング

PPUのベクタデータ型のすべてがSPUでサポートされるわけではありません。PPUのベクタデータ型で同一のSPUデータ型にはマッピングされないものを 表 1-2に示します。

表 1-2: VMX データ型から SPU データ型への同一でないマッピング

VMX データ型	SPU データ型へのマッピング
vector bool char	vector unsigned char
vector bool short	vector unsigned short
vector bool int	vector unsigned int
vector pixel	vector unsigned short[1]

[1] `vector pixel` および `vector bool short` は同一の基本ベクタ型 (`vector unsigned short`) にマッピングされるため、`vec_unpackh` および `vec_unpackl` の多重定義された関数は一意に解決することができません。

1.1.2. SPUデータ型からPPUデータ型へのマッピング

SPUのデータ型のすべてがPPU VMXでサポートされるわけではありません。SPUのデータ型で同一のPPUベクタデータ型にはマッピングされないものを 表 1-3に示します。.

表 1-3: SPU データ型から VMX データ型への同一でないマッピング

SPU Data Type	Maps to VMX Data Type
vector unsigned long long	vector bool char
vector signed long long	vector bool short
vector double	vector bool int

1.2. ヘッダファイル

SPU 用と PPU 用に別個のシステムヘッダファイルが存在しており、本仕様書で定義する言語拡張機能に必要となる型定義やその他の情報を含んでいます。

SPUシステムヘッダファイル`spu_intrinsics.h`では共通の列挙定数および型が定義されており、これらは単一トークンベクタ型の定義（2ページの 表 1-4**エラー! 参照元が見つかりません。**を参照）およびMFCチャネルニーモニックの列挙定数の定義（53ページの 表 2-89: MFCチャネル番号 1を参照）を含みます。これらに加えこのヘッダファイルでは、実装固有の定義があればそれらを含むコンパイラ固有のヘッダファイル`spu_internals.h`がインクルードされます。

PPU のシステムヘッダファイル `altivec.h` は、型および予約語を定義し、さらに、実装固有の定義があればそれらを含んでいます。PPU のシステムヘッダファイル `vec_types.h` は、本仕様書で定義する言語拡張機能に必要となる型定義を行なっています。

1.2.1. 単一トークン型定義

コードの移植性を高めるために、vector予約語のデータ型について、単一トークンの型定義が提供されています。表 1-4に示すこれらの型定義は、SPUでは`spu_intrinsics.h`、PPUでは`vec_types.h`で定義されています。単一トークンの型定義は、型宣言を簡素化するだけでなく、総称組み込み関数の拡張やPPU VMX組み込み関数とSPU組み込み関数の間のマッピングにおけるクラス名としても働きます。

表 1-4: 単一トークンベクタデータ型

vector 予約語データ型	単一トークン型定義	SPU/PPU
vector unsigned char	vec_uchar16	両方
vector signed char	vec_char16	両方
vector unsigned short	vec_ushort8	両方
vector signed short	vec_short8	両方
vector unsigned int	vec_uint4	両方
vector signed int	vec_int4	両方
vector unsigned long long	vec_ullong2	SPU

vector 予約語データ型	単一トークン型定義	SPU/PPU
vector signed long long	vec_llong2	SPU
vector float	vec_float4	両方
vector double	vec_double2	SPU
vector bool char	vec_bchar16	PPU
vector bool short	vec_bshort8	PPU
vector bool int	vec_bint4	PPU
vector pixel	vec_pixel8	PPU

1.3. アラインメント

表 1-5 に各種データ型のサイズとデフォルトのアラインメントを示します。

表 1-5: デフォルトのデータ型アラインメント

データ型	サイズ	アラインメント
char	1	byte
short	2	halfword
int	4	word
long	4	word/doubleword
long long	8	doubleword
float	4	word
double	8	doubleword
pointer	4	word
vector	16	quadword

追加のアラインメント制御は、変数や構造体/共用体のメンバに対して、GCC の aligned 属性を用いて実現できます。例えば、下記の宣言文では、浮動小数点スカラ *factor* をクワッドワード境界にアラインすることができます。

```
float factor __attribute__ ((aligned (16)));
```

1.3.1. __align_hint (SPUのみ)

`__align_hint` 組み込み関数が以下の目的で提供されています。

- ポインタを介したデータアクセスを改善する
- 自動ベクトル化サポートのために必要な追加情報をコンパイラに提供する

この組み込み関数は SPU のみで使用できます。PPU においても有用ですけれども、PPU サポートが必要とはされていません。

`__align_hint` は組み込み関数として定義されていますが、指示文のように振舞います。具体的なコードは全く生成されないからです。例えば、

```
__align_hint(ptr, base, offset)
```

において、`__align_hint` 組み込み関数はコンパイラに対して、ポインタ *ptr* がベースアラインメント *base* と *base* からのオフセット *offset* でもってデータを指していることを通知します。ベースアラインメントは 2 のべき乗でなければなりません。ベースアドレスをゼロにすると、ポインタが既知のアラインメントを持たないことを意味します。アラインメントのオフセットは、*base* より小さいかあるいはゼロでなければなりません。

`__align_hint` 組み込み関数は、自然なアラインメントになっていないポインタの指定を意図したものではありません。自然なアラインメントでないポインタを指定すると、データオブジェクトがクワッドワードをまたぐことになります。正しくないアラインメント指定を行うと、正しくないプログラムが生成されるかもしれません。

プログラミングの注意: 仕様準拠のコンパイラ実装では `__align_hint` 組み込み関数を用意しなければなりませんが、コンパイラはこのヒントを無視する場合があります。

1.4. ベクタ型に対する演算

ほとんどの C/C++演算子また基本演算についてベクタデータ型の演算ができるような拡張はされていませんが、若干のものは拡張されています。拡張されているものには、sizeof()演算子、代入演算子(=)、アドレス演算子(&), ポインタ演算、タイプキャスト演算があります。

1.4.1. sizeof() 演算子

ベクタ型に対する sizeof() 演算は常に 16 を返します。

1.4.2. 代入演算子

ある式の左右どちらかがベクタ型である場合、その式の両側が同一のベクタ型でなければなりません。よって、`a = b` という式は *a* と *b* が同じ型の場合および *a* と *b* のどちらもベクタ型ではない場合についてのみ有効です。それ以外の場合は式は無効となり、その矛盾はコンパイラによりエラーとしてレポートされます。

1.4.3. アドレス演算子

演算`&a` は *a* がベクタ型である場合は有効です。演算結果はベクタ *a* を指すポインタとなります。

1.4.4. ポインタ演算とポインタ逆参照

ベクタ型へのポインタに関する通常のポインタ演算が実行できます。例えば、*p* があるベクタ型へのポインタである場合、*p+1* は *p* の次のベクタへのポインタです。

ベクタポインタ *p*を逆参照するということは *p*の下位 4 ビットをマスクすることによって得られたアドレスからの（アドレスへの）128 ビットベクタのロード（ストア）を意味します。ベクタのアラインメントが正しくない場合、下位 4 ビットの値が 0 になりません。ベクタは 16 バイトアラインされていますが（1.3. アラインメント参照）、それでもなおアラインされていないベクタをロード・ストアしたいことがあるかもしれません。アラインされていないベクタは総称組み込み関数（2.2. 総称組み込み関数とビルトイン関数参照）を用いたいくつかの方法でロードすることができます。以下のコードはSPU上でアラインされていない浮動小数点ベクタをロードする方法の一例です。

```
vector float load_misaligned_vector_float (vector float *ptr)
{
       vector float qw0, qw1;
       int shift;

       qw0 = *ptr;
       qw1 = *(ptr+1);
       shift = (unsigned) ptr & 15;

       return spu_or(
                 spu_slqwbyte(qw0, shift),
                 spu_rlmaskqwbyte(qw1, shift-16));
       }
```

同様に、SPU 上でアラインされていない浮動小数点ベクタへストアする方法の例を以下に示します。

```
void store_misaligned_vector_float (vector float flt, vector float *ptr)
{
       vector float qw0, qw1;
       vector unsigned int mask;
       int shift;

       qw0 = *ptr;
       qw1 = *(ptr+1);
       shift = (unsigned)(ptr) & 15;
       mask = (vector unsigned int)
              spu_rlmaskqwbyte((vector unsigned char)(0xFF), -shift);
```

```
        flt = spu_rlqwbyte(flt, -shift);

        *ptr = spu_sel(qw0, flt, mask);
        *(ptr+1) = spu_sel(flt, qw1, mask);
}
```

1.4.5. 型のキャスト

ベクタ型と非ベクタ型へのポインタは互いにキャストによって型を変換することができます。別名付けの目的では、表 1-6に示すように、ベクタ型は対応する要素型の配列として扱われます。 あるポインタをベクタ型のアドレスへキャストする場合、対象アドレスが確実に 16 バイトアラインされているようにする責任は、プログラマにあります。PPUのみに適用可能なベクタ型については、その基となるスカラ型はありません。

表 1-6: ベクタポインタ型とそれに対応する基本要素のポインタ型

ベクタポインタ型 (vector T*)	基本要素ポインタ型 (T*)	SPU/PPU
vector unsigned char*	unsigned char*	両方
vector signed char*	signed char*	両方
vector unsigned short*	unsigned short*	両方
vector signed short*	signed short*	両方
vector unsigned int*	unsigned int*	両方
vector signed int*	signed int*	両方
vector unsigned long long*	unsigned long long*	SPU
vector signed long long*	signed long long*	SPU
vector float*	float*	両方
vector double*	double*	SPU

あるベクタ型から別のベクタ型へのキャストは明示的でなければならず、通常の C 言語のキャストを用いて行ないます。これらのキャストはいずれもデータ変換を行ないません。よって結果のビットパターンはキャストされる引数のビットパターンと同一です。

ベクタ型とスカラ型間のキャストは許されていません。SPU上では、代わりに`spu_extract`、`spu_insert`、`spu_promote`のいずれかの総称組み込み関数またはキャスト用個別組み込み関数を用いて効率的に同じ結果を得ることができます。（「2.1.1. キャスト用個別組み込み関数」を参照してください。） PPU上では、`vec_lde`および`vec_ste` 組み込み関数を用いてスカラとベクタ型の間のコピーを行なうことができます。

1.4.6. ベクタリテラル

表 1-7に示すように、ベクタリテラルは、小括弧で括られたベクタ型の後に中括弧で括られた定数式の組が続く形として記述されます。ベクタリテラルをマクロへの引数として使う場合は、リテラルを小括弧で括る必要があります。その他の場合はすべて、リテラルは小括弧で括らずに使用できます。ベクタの要素は対応する式を用いて初期化されます。対応する式が指定されない要素はデフォルト値 0 が適用されます。ベクタリテラルは、初期化文の中であるいは実行文中の定数として使用できます。ベクタの初期化とベクタ複合リテラルの構文は、要素指示子がベクタの要素に存在しないことを除き、対応する配列の構文と同じです。初期化子は基となる型のサイズに応じて、要素数が 2, 4, 8 または 16 の配列の役割を果たします。例えば、次の 2 つの初期化は有効かつ同等です。

```
vector signed int v1[] = {{0, 1, 2, 3},{4, 5, 6, 7}};
vector signed int v2[] = {0, 1, 2, 3, 4, 5, 6, 7};
```

次の 2 つの構造体の初期化も有効かつ同等です。

```
struct stypy {
        int i;
        vector signed int t;
} v3 = {1, {0, 1, 2, 3}}, v4 = {1, 0, 1, 2, 3};
```

次の型は SPU 上と PPU 上の両方ともベクタリテラルを用いた初期化はできません: `qword`, `vector bool char`, `vector bool short`, `vector bool int`, `vector pixel`。組み込み関数を用いるか、あるいは、これらのベクタ型にキャストすることにより生成できます。

表 1-7: ベクタリテラルの形式と説明

表記	意味	SPU/PPU
(vector unsigned char) {unsigned char, ...}	16 個の符号なし 8 ビット量 1 セット	両方
(vector signed char) {signed char, ...}	16 個の符号付き 8 ビット量 1 セット	両方
(vector unsigned short) {unsigned short, ...}	8 個の符号なし 16 ビット量 1 セット	両方
(vector signed short) {signed short, ...}	8 個の符号付き 16 ビット量 1 セット	両方
(vector unsigned int) {unsigned int, ...}	4 個の符号なし 32 ビット量 1 セット	両方
(vector signed int) {signed int, ...}	4 個の符号付き 32 ビット量 1 セット	両方
(vector unsigned long long) {unsigned long long, ...}	2 個の符号なし 64 ビット量 1 セット	SPU
(vector signed long long) {signed long long, ...}	2 個の符号付き 64 ビット量 1 セット	SPU
(vector float) {float, ...}	4 個の 32 ビット 浮動小数点量 1 セット	両方
(vector double) {double, ...}	2 個の 64 ビット 浮動小数点量 1 セット	両方

*Altivec Technology Programming Interface Manual*で規定されている構文に対応する代替形式をサポートすることも許されています。この形式は、表 1-8に示すように、括弧で括られたベクタ型の後に括弧で括られた定数式の組が続く形をとります。

表 1-8: ベクタリテラルの代替形式と説明

表記	意味	SPU/PPU
(vector unsigned char)(unsigned int)	int で指定した同じ値をもつ符号なし 8 ビット量 16 個 1 セット	両方
(vector unsigned char)(unsigned int, ..., unsigned int)	16 個の int で指定した値をもつ符号なし 8 ビット量 16 個 1 セット	両方
(vector signed char)(signed int)	int で指定した同じ値をもつ符号付き 8 ビット量 16 個 1 セット	両方
(vector signed char)(signed int, ..., signed int)	16 個の int で指定した値をもつ符号付き 8 ビット量 16 個 1 セット	両方
(vector unsigned short)(unsigned int)	int で指定した同じ値をもつ符号なし 8 ビット量 8 個 1 セット	両方
(vector unsigned short)(unsigned int, ..., unsigned int)	8 個の int で指定した値をもつ符号なし 16 ビット量 8 個 1 セット	両方
(vector signed short)(signed int)	int で指定した同じ値をもつ符号付き 16 ビット量 8 個 1 セット	両方
(vector signed short)(signed int, ..., signed int)	8 個の int で指定した値をもつ符号付き 16 ビット量 8 個 1 セット	両方
(vector unsigned int)(unsigned int)	int で指定した同じ値をもつ符号なし 32 ビット量 4 個 1 セット	両方
(vector unsigned int)(unsigned int, ..., unsigned int)	4 個の int で指定した値をもつ符号なし 32 ビット量 4 個 1 セット	両方
(vector signed int)(signed int)	int で指定した同じ値をもつ符号付き 32 ビット量 4 個 1 セット	両方
(vector signed int)(signed int, ..., signed int)	4 個の int で指定した値をもつ符号付き 32 ビット量 4 個 1 セット	両方
(vector unsigned long long)(unsigned long long)	long long で指定した同じ値をもつ符号なし 64 ビット量 2 個 1 セット	SPU

表記	意味	SPU/PPU
(vector unsigned long long)(unsigned long long, unsigned long long)	2 個の long long で指定した値をもつ符号なし 64 ビット量 2 個 1 セット	SPU
(vector signed long long)(signed long long)	long long で指定した同じ値をもつ符号付き 64 ビット量 2 個 1 セット	SPU
(vector signed long long)(signed long long, signed long long)	2 個の long long で指定した値をもつ符号付き 64 ビット量 2 個 1 セット	SPU
(vector float)(float)	float で指定した同じ値をもつ 32 ビット浮動小数点量 4 個 1 セット	両方
(vector float)(float, float, float, float)	4 個の float で指定した値をもつ 32 ビット浮動小数点量 4 個 1 セット	両方
(vector double)(double)	double で指定した同じ値をもつ 64 ビット浮動小数点量 2 個 1 セット	SPU
(vector double)(double, double)	2 個の double で指定した値をもつ 64 ビット 浮動小数点量 2 個 1 セット	SPU

1.5. Restrict修飾子

「ISO/IEC 9899:1999 -Programming Language C」（略称：C99）で定義されている `restrict` 修飾子は、オブジェクトへのアクセスが全てある特定のポインタを通して行なわれることを保証してコンパイラのより良いコードを生成を手助けすることを意図したものです。ポインタに `restrict` 修飾子がついているとそのポインタは `restrict` 修飾されていることになります。以下に例を挙げます。

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
```

この例ではポインタ *s1* および *s2* の両方が `restrict` 修飾されています。これによりコンパイラはコピー元とコピー先のオブジェクトがオーバーラップしないと安全に仮定することができるため、より効率的な実装が可能となります。

1.6. プログラマの指示によるSPUの分岐予測

分岐予測はフィードバックの指示による最適化を行なうことにより著しく改善することができます。しかしながら、フィードバックの指示による最適化は、典型的なデータセットが存在しない場合においては必ずしも実用的でないことから、その代わりに、SPU においては、GCC の `__builtin_expect` 関数の強化版を使用したプログラマの指示による分岐予測を提供します。

```
int __builtin_expect(int exp, int value)
```

プログラマは `_builtin_expect` を用いてコンパイラに分岐予測情報を与えることができます。`_builtin_expect` の返り値は引数 *exp* の値であり、この引数は整数式である必要があります。動的予測の場合、引数 *value* はコンパイル時の定数か変数のどちらでもかまいません。`_builtin_expect` 関数では、引数 `exp` と `value` は同じ値であると想定されます。

静的予測の例

```
if (__builtin_expect(x, 0)) {
        foo();        /* programmer doesn't expect foo to be called */
}
```

動的予測の例

```
cond2 = ...                 /* predict a value for cond1 */
...
cond1 = ...
if (__builtin_expect(cond1, cond2)) {
        foo();
}
```

```
cond2 = cond1;      /* predict that next branch is the same as the
                    previous */
```

多数の分岐が生成されるのを避けるためにコンパイラは引数 *exp* の複雑さを制限する必要があるかもしれません。このようなケースが生じてプログラムの分岐予測が無視された場合、コンパイラは警告を発行する必要があります。

プログラミングの注意: 上記拡張の実装は PPU には必要とされません。PPU では分岐の静的予測のみをサポートしているからです。

1.7. インラインアセンブリ

C/C++言語の構文や組み込み関数のみを使用して意図した低レベルのプログラミングをすることは、プログラマにとって時として難しいことがあります。このようなケースの対処にインラインアセンブリの使用が求められる場合があるため、インラインアセンブリを提供する必要があります。提供するインラインアセンブリの構文は GCC 実装の AT&T アセンブリ構文に適合していなければなりません。

ハードウェアによる効果的な 2 命令同時発行を可能にするために必要な既知のアラインメントを確実にするために、インラインアセンブリ内で.balignl 指示文を使用することができます。

1.8. ターゲット定義

複数のターゲットを対象として条件付コンパイル可能なコード開発をサポートするために、コンパイラはコードが SPU 用にコンパイルされる場合には __SPU__を、PPU 用にコンパイルされる場合には __PPU__を定義する必要があります。例えば以下のコードはアラインメントのずれたクワッドワードのロードをサポートしています。__SPU__と__PPU__の定義を用いて条件付でどちらのコードを用いるか選択します。対象となるプロセッサに応じて、選択されるコードは異なったものになります。

```
vector unsigned char load_qword_unaligned(vector unsigned char *ptr)
{
       vector unsigned char qw0, qw1, qw;
#ifdef __SPU__
       unsigned int shift;
#endif
       qw0 = *ptr;
       qw1 = *(ptr+1);
#ifdef __SPU__
       shift = (unsigned int)(ptr) & 15;
       qw = spu_or(spu_slqwbyte(qw0, shift),
                   spu_rlmaskqwbyte(qw1, (signed)(shift - 16)));
#elif defined(__PPU__) /* PPU */
       qw = vec_perm(qw0, qw1, vec_lvsl(0, ptr));
#else
# error “This code can only be compiled for PPU or the SPU”
#endif
       return (qw);
}
```

2. SPU低レベル個別・総称組み込み関数

本章ではシステムの下層に位置する命令セットアーキテクチャ（ISA）および Synergistic Processor Element (SPE)ハードウェアを C 言語でアクセスすることを可能にするために最小限必要となる基本的組み込み関数とビルトイン関数について説明します。組み込み関数には以下の三種類があります。

- 個別組み込み関数
- 総称組み込み関数
- ビルトイン関数

組み込み関数はコンパイラ内部に実装することも、マクロとして実装することも可能です。しかしながら、マクロとして実装する場合には引数として渡すベクタリテラルが制限されます。詳細については 1.4.6. ベクタリテラルを参照してください。

2.1. 個別組み込み関数

個別組み込み関数はそれらが SPU アセンブリ命令と一対一の関係にあるという意味において*個別*と表現しています。全ての個別組み込み関数は対応する SPU アセンブリ命令に接頭辞 `si_` を加えた名前がついています。例えばアセンブリ命令 `stop` を実装した個別組み込み関数は `si_stop` と命名されています。

ほとんど全てのアセンブリ命令に対応する個別組み込み関数が存在します。しかしながら、いくつかのアセンブリ命令の機能についてはCやC++言語で提供した方が良いため、それらの命令については個別組み込み関数を提供していません。表 2-9に対応する個別組み込み関数を持たないアセンブリ命令の一覧を示します。

表 2-9: 対応する個別組み込み関数をもたないアセンブリ命令

命令の種類	SPU 命令
分岐命令	br, bra, brsl, brasl, bi, bid, bie, bisl, bisld, bisle, brnz, brz, brhnz, brhz, biz, bizd, bize, binz, binzd, binze, bihz, bihzd, bihze, bihnz, bihnzd, bihnze (bisled, bisledd, bislede を除く)
分岐ヒント命令	hbr, hbrp, hbra, hbrr
割り込み復帰命令	iret, iretd, irete

表 2-10に挙げた個別組み込み関数を除き、全ての個別組み込み関数は総称組み込み関数を介してアクセスできます。アクセスすることができない組み込み関数は以下の 3 つのカテゴリに分類できます。

- 基本的な変数の参照（すなわち、ベクタやスカラのロードやストア）により生成される命令
- 即値ベクタの生成に用いる命令
- 有用性が限定され、まれな状況を除き使用が見込まれない命令

表 2-10: 総称組み込み関数を介したアクセスができない個別組み込み関数

命令と説明	用法	アセンブリ命令へのマップ
クワッドワード未満の挿入を制御するための組み込み関数		
si_cbd: Generate Controls for Byte Insertion (d-form) 符号付き 7 ビット即値 *imm* を *a* のワード要素 0 へ加えることにより実効アドレスを算出する。実効アドレスの下位 4 ビットはクワッドワード内のバイトの位置を決定する際に使用する。決定された位置に基づき、バイト（バイト要素 3）をクワッドワード内の示された位置に挿入するために `si_shufb` と共に使用するパターンを生成する。このパターンをクワッドワード *d* へ代入する。	*d* = si_cbd(*a*, *imm*)	CBD d, imm(a)

命令と説明	用法	アセンブリ命令へのマップ
si_cbx: Generate Controls for Byte Insertion (x-form) *a* のワード要素 0 を *b* のワード要素 0 へ加えることにより実効アドレスを算出する。実効アドレスの下位 4 ビットはクワッドワード内のバイトの位置を決定する際に使用する。決定された位置に基づき、バイト（バイト要素 3）をクワッドワード内の示された位置に挿入するために si_shufb と共に使用するパターンを生成する。このパターンをクワッドワード *d* へ代入する。	*d* = si_cbx(*a*, *b*)	CBX d, a, b
si_cdd: Generate Controls for Doubleword Insertion (d-form) 符号付き 7 ビット即値 *imm* を *a* のワード要素 0 へ加えることにより実効アドレスを算出する。実効アドレスの下位 4 ビットはクワッドワード内のダブルワードの位置を決定する際に使用する。決定された位置に基づき、ダブルワード（ダブルワード要素 0）をクワッドワード内の示された位置に挿入するために si_shufb と共に使用するパターンを生成する。このパターンをクワッドワード *d* へ代入する。	*d* = si_cdd(*a*, *imm*)	CDD d, imm(a)
si_cdx: Generate Controls for Doubleword Insertion (x-form) *a* のワード要素 0 を *b* のワード要素 0 へ加えることにより実効アドレスを算出する。実効アドレスの下位 4 ビットはクワッドワード内のダブルワードの位置を決定する際に使用する。決定された位置に基づき、ダブルワード（ダブルワード要素 3）をクワッドワード内の示された位置に挿入するために si_shufb と共に使用するパターンを生成する。このパターンをクワッドワード *d* へ代入する。	*d* = si_cdx(*a*, *b*)	CDX d, a, b
si_chd: Generate Controls for Halfword Insertion (d-form) 符号付き 7 ビット即値 *imm* を *a* のワード要素 0 へ加えることにより実効アドレスを算出する。実効アドレスの下位 4 ビットはクワッドワード内のハーフワードの位置を決定する際に使用する。決定された位置に基づき、ハーフワード（ハーフワード要素 1）をクワッドワード内の示された位置に挿入するために si_shufb と共に使用するパターンを生成する。このパターンをクワッドワード *d* へ代入する。	*d* = si_chd(*a*, *imm*)	CHD d, imm(a)
si_chx: Generate Controls for Halfword Insertion (x-form) *a* のワード要素 0 を *b* のワード要素 0 へ加えることにより実効アドレスを算出する。実効アドレスの下位 4 ビットはクワッドワード内のハーフワードの位置を決定する際に使用する。決定された位置に基づき、ハーフワード（ハーフワード要素 1）をクワッドワード内の示された位置に挿入するために si_shufb と共に使用するパターンを生成する。このパターンをクワッドワード *d* へ代入する。	*d* = si_chx(*a*, *b*)	CHX d, a, b
si_cwd: Generate Controls for Word Insertion (d-form) 符号付き 7 ビット即値 *imm* を *a* のワード要素 0 へ加えることにより実効アドレスを算出する。実効アドレスの下位 4 ビットはクワッドワード内のワードの位置を決定する際に使用する。決定された位置に基づき、ワード（ワード要素 0）をクワッドワード内の示された位置に挿入するために si_shufb と共に使用するパターンを生成する。このパターンをクワッドワード *d* へ代入する。	*d* = si_cwd(*a*, *imm*)	CWD d, imm(a)
si_cwx: Generate Controls for Word Insertion (x-form) *a* のワード要素 0 を *b* のワード要素 0 へ加えることにより実効アドレスを算出する。実効アドレスの下位 4 ビットはクワッドワード内のワードの位置を決定する際に使用する。決定された位置に基づき、ワード（ワード要素 0）をクワッドワード内の示された位置に挿入するために si_shufb と共に使用するパターンを生成する。このパターンをクワッドワード *d* へ代入する。	*d* = si_cwx(*a*, *b*)	CWX d, a, b

命令と説明	用法	アセンブリ命令へのマップ
定数生成用の組み込み関数		
si_il: Immediate Load Word 符号付き 16 ビット即値 *imm* を 32 ビットへ符号拡張し、クワッドワード *d* の 4 つのワード要素それぞれに代入する。	*d* = si_il(*imm*)	IL d, imm
si_ila: Immediate Load Address 18 ビット即値 *imm* をクワッドワード *d* の 4 つのワード要素それぞれの下位ビットに代入する。各ワードの上位 14 ビットは 0 に設定する。	*d* = si_ila(*imm*)	ILA d, imm
si_ilh: Immediate Load Halfword 符号付き 16 ビット即値 *imm* をクワッドワード *d* の 8 つのハーフワード要素それぞれに代入する。	*d* = si_ilh(*imm*)	ILH d, imm
si_ilhu: Immediate Load Halfword Upper 符号付き 16 ビット即値 *imm* をクワッドワード *d* の 4 つのワード要素それぞれの上位 16 ビットに代入する。各ワードの右から 16 ビットは 0 に設定する。	*d* = si_ilhu(*imm*)	ILHU d, imm
si_iohl: Immediate Or Halfword Lower 16 ビット即値 *imm* の先頭に 0 を付加しクワッドワード *a* の 4 つのワード要素それぞれとの論理和をとる。結果をクワッドワード *d* へ代入する。	*d* = si_iohl(*a*, *imm*)	rt <--- a IOHL rt, imm d <--- rt
ノーオペレーション組み込み関数		
si_lnop: No Operation (load) ロードパイプライン上でノーオペレーション命令を実行する。	si_lnop()	LNOP
si_nop: No Operation (execute) 実行パイプライン上でノーオペレーション命令を実行する。	si_nop()	NOP rt[1]
ロード・ストア用の組み込み関数		
si_lqa: Load Quadword (a-form) 符号拡張された 18 ビット即値 *imm* により下位 4 ビットを強制的に 0 とした実効アドレスを決定する。この実効アドレスのクワッドワードはクワッドワード *d* へ代入する。	*d* = si_lqa(*imm*)	LQA d, imm
si_lqd: Load Quadword (d-form) 符号拡張された 14 ビット即値 *imm* の下位 4 ビットを 0 にし、その値とクワッドワード *a* のワード要素 0 の和の下位 4 ビットを 0 にすることにより実効アドレスを計算する。この実効アドレスのクワッドワードはクワッドワード *d* へ代入する。	*d* = si_lqd(*a*, *imm*)	LQD d, imm(a)
si_lqr: Load Quadword Instruction Relative (a-form) 符号付き 18 ビット即値 *imm* の下位 2 ビットを 0 にし、その値と命令アドレスの和の下位 4 ビットを 0 にすることにより実効アドレスを計算する。この実効アドレスのクワッドワードはクワッドワード *d* へ代入する。	*d* = si_lqr(*imm*)	LQR, d, imm
si_lqx: Load Quadword (x-form) クワッドワード *a* のワード要素 0 とクワッドワード *b* のワード要素 0 の和の下位 4 ビットを 0 にすることにより実効アドレスを計算する。この実効アドレスのクワッドワードはクワッドワード *d* へ代入する。	*d* = si_lqx(*a*, *b*)	LQX d, a, b
si_stqa: Store Quadword (a-form) 符号拡張された 18 ビット即値 *imm* の下位 4 ビットを強制的に 0 とすることにより実効アドレスを決定する。クワッドワード *a* はこの実効アドレスへ格納される。	si_stqa(*a*, *imm*)	STQA a, imm

命令と説明	用法	アセンブリ命令へのマップ
si_stqd: Store Quadword (d-form) 符号付き 14 ビット即値 *imm* の下位 4 ビットを強制的に 0 にし、その値とクワッドワード *b* のワード要素 0 の和の下位 4 ビットを 0 にすることにより実効アドレスを計算する。クワッドワード *a* はこの実効アドレスへ格納される。	si_stqd(*a*, *b*, *imm*)	STQD a, imm(b)
si_stqr: Store Quadword Instruction Relative (a-form) 符号付き 18 ビット即値 *imm* の下位 2 ビットを強制的に 0 にし、その値と命令アドレスの和の下位 4 ビットを 0 にすることにより実効アドレスを計算する。クワッドワード *a* はこの実効アドレスへ格納される。	si_stqr(*a*, *imm*)	STQR, a, imm
si_stqx: Store Quadword (x-form) クワッドワード *b* のワード要素 0 とクワッドワード *c* のワード要素 0 の和の下位 4 ビットを強制的に 0 にすることにより実効アドレスを計算する。クワッドワード *a* はこの実効アドレスへ格納される。	si_stqx(*a*, *b*, *c*)	STQX a, b, c
制御用の組み込み関数		
si_stopd: Stop and Signal with Dependencies 全てのレジスタ依存関係が満たされた後に SPU の実行を停止し、0x3FFF タイプの信号を送出する。この組み込み関数は他の全ての命令に対して揮発性を持つとみなされるため、他の命令と順序が入れ替わることはない。	si_stopd(*a*, *b*, *c*)	STOPD a, b, c

[1] 偽のターゲットであるパラメータ *rt* は近傍命令のレジスタ使用状態に応じて最適な値が選択されます。

個別組み込み関数は以下の型の引数のみ受け付けます。

- 明示的な定数式または記号アドレスとしての即値リテラル
- 列挙型
- qword 型

上記の以外の型をもつ引数は qword へキャストしなければなりません。

個々の命令における詳細については、*Synergistic Processor Unit 命令セット・アーキテクチャ*を参照してください。

2.1.1. キャスト用個別組み込み関数

個別組み込み関数を使用する際、スカラ型から qword 型へ、あるいは qword 型からスカラ型へのキャストが必要な場合があります。ベクタ型同士におけるキャストと同様に、キャスト用個別組み込み関数の実行はレジスタに格納されている引数に何の影響も与えません。キャスト用個別組み込み関数は全て以下の形式をとります。

```
d=casting_intrinsic(a)
```

キャスト用個別組み込み関数の詳細については 表 2-11を参照してください。

表 2-11: キャスト用個別組み込み関数

キャスト用組み込み関数	戻り値／引数の型		説明
	d	a	
si_to_char	signed char	qword	qword *a* のバイト要素 3 を signed char *d* へキャスト
si_to_uchar	unsigned char		qword *a* のバイト要素 3 を unsigned char *d* へキャスト
si_to_short	short		qword *a* のハーフワード要素 1 を short *d* へキャスト
si_to_ushort	unsigned short		qword *a* のハーフワード要素 1 を unsigned short *d* へキャスト
si_to_int	int		qword *a* のワード要素 0 を int *d* へキャスト
si_to_uint	unsigned int		qword *a* のワード要素 0 を unsigned int *d* へキャスト
si_to_ptr	void *		qword *a* のワード要素 0 を void 型ポインタ *d* へキャスト
si_to_llong	long long		qword *a* のダブルワード要素 0 を long long *d* へキャスト
si_to_ullong	unsigned long long		qword *a* のダブルワード要素 0 を unsigned long long *d* へキャスト
si_to_float	float		qword *a* のワード要素 0 を float *d* へキャスト
si_to_double	double		qword *a* のダブルワード要素 0 を double *d* へキャスト
si_from_char	qword	signed char	signed char *a* を qword *d* のバイト要素 3 へキャスト
si_from_uchar		unsigned char	unsigned char *a* を qword *d* のバイト要素 3 へキャスト
si_from_short		short	short *a* を qword *d* のハーフワード要素 1 へキャスト
si_from_ushort		unsigned short	unsigned short *a* を qword *d* のハーフワード要素 1 へキャスト
si_from_int		int	int *a* を qword *d* のワード要素 0 へキャスト
si_from_uint		unsigned int	unsigned int *a* を qword *d* のワード要素 0 へキャスト
si_from_ptr		void *	void ポインタ *a* を qword *d* のワード要素 0 へキャスト
si_from_llong		long long	long long *d* を qword *d* のダブルワード要素 0 へキャスト
si_from_ullong		unsigned long long	unsigned long long *a* を qword *d* のダブルワード要素 0 へキャスト
si_from_float		float	float *a* を qword *d* のワード要素 0 へキャスト
si_from_double		double	double *a* を qword *d* のダブルワード要素 0 へキャスト

キャスト組み込み関数はデータ変換を行なわないため、スカラ型から `qword` 型へキャストした場合のクワッドワード値は部分的に不定となります。

2.2. 総称組み込み関数とビルトイン関数

一つ以上の個別組み込み関数にマップされている演算を行なう関数を総称組み込み関数と呼びます。総称組み込み関数がどの個別組み込み関数へマップされるかは入力引数によります。ビルトイン関数は総称組み込み関数と似ていますが、二つ以上の SPU 命令へマップされている点において総称組み込み関数と異なります。全ての総称組み込み関数およびビルトイン関数の名前には接頭辞 `spu_` を加えた名前がついています。例えばアセンブリ命令 `stop` を実装した総称組み込み関数は `spu_stop` と命名されています。

2.2.1. スカラ型のオペランドをとる組み込み関数のマップ

スカラ型の引数をとる組み込み関数は、即値フィールドをもつ SPU 命令用に導入されました。例えば、組み込み関数 `vector signed int spu_add(vector signed int, int)` はアセンブリ命令 `AI` に変換されます。

即値の長さはアセンブリ命令により異なり、 7、10、16、18 ビットのいずれかになります。定義域外の即値に対してのコンパイラの動作は組み込み関数の種類により異なります。デフォルト設定では、即値形式の個別組み込み関数に定義域外の即値を渡すと、エラーとして通知されます。コンパイラは定義域外の即値を渡された場合に警告を発行し、LSB から指定したビット分のみを使用するというオプションを提供することができます。

総称組み込み関数は、スカラ型のオペランドが命令の即値フィールド内に収まるか否かによらずあらゆる範囲のスカラ型オペランドをサポートします。以下の例について考えてみましょう。

```
d = spu_and (vector unsigned int a, int b);
```

引数 *b* に応じて、以下のような命令が生成されます。

- *b* が複数ある即値形式のいずれかがサポートする定義域内のリテラル定数である場合、当該即値形式の命令が生成される。例えば、*b* が 1 である場合、`ANDI d, a, 1` が生成される。
- *b* がリテラル定数であり、定義域外だが畳み込むことが可能であり代替即値形式の命令を用い実装できる場合、その代替即値形式の命令が生成される。例えば、*b* が `0x30003` である場合、`ANDHI d, a, 3` が生成される。ここで「代替即値形式の命令」とはデータ要素サイズが小さい即値形式の命令を指す。
- *b*が一つまたは二つの即値ロード命令により生成可能なリテラル定数の場合、適切な命令が使用され、その後に非即値形式の命令が続く。即値ロード命令には`IL`, `ILH`, `ILHU`, `ILA`, `IOHL`, `FSMBI`がある。表 2-12に定数 *b*の値と生成が予想される即値ロード命令の対応を示す。

表 2-12: 定数 *b* の値により生成が予想される即値ロード命令

定数 *b*	生成される命令
-6000	IL b, -6000 AND d, a, b
131074 (0x20002)	ILH b, 2 AND d, a, b
131072 (0x20000)	ILHU b, 2 AND d, a, b
134000 (0x20B70)	ILA b, 134000 AND d, a, b
262780 (0x4027C)	ILHU b, 4 IOHL b, 636 AND d, a, b
(0xFFFFFFFF, 0x0, 0x0, 0xFFFFFFFF)	FSMBI b, 0xF00F AND d, a, b

- *b* が変数（非リテラル）の整数である場合、当該整数をベクタ全体に複製し非即値形式版の命令が後くコードを生成する。例えば、*b* が不定整数である場合、定数領域の CONST_AREA と offset により特定されるアドレスにシャッフルパターン (`0x10203, 0x10203, 0x10203, 0x10203`) をロードし、以下の命令を生成する。

```
LQD    pattern, CONST_AREA, offset
SHUFB  b, b, b, pattern
AND    d, a, b
```

2.2.2. 組み込み関数引数の暗黙変換

ベクタ型を持つ引数に対する暗黙の変換はありません。スカラ型の引数は C/C++規格で規定されている規則に従って変換されます。以下の例を考えます。

```
d = spu_insert(a, b, element);
```

スカラ *a* が、*element* 引数で指定されるベクタ *b* の要素に挿入されます。*b* が vector double のとき、*a* は double に、*element* は int に変換され、*d* は vector double でなければなりません。

2.2.3. 表記法と命名規則

以降の総称組み込み関数についての解説では以下の表記法と命名規則を用いています。

- 各総称組み込み関数用の表では、当該関数がサポートしている入力データ型を明記している。
- スカラ型オペランドをとる組み込み関数の場合、即値形式の命令のみを示す。その他の形式の命令については 2.2.1. スカラ型のオペランドをとる組み込み関数のマップで述べる規則により導き出すことができる。
- 総称組み込み関数であるか個別組み込み関数であるかによらず、組み込み関数の中には、入出力用のレジスタを別々に指定する代わりに、入力レジスタを出力レジスタとしても使うアセンブリ命令へマップされるものがある。そのようなアセンブリ命令には、ADDX、DFMS、MPYHHA、SFX などがある。これらの組み込み関数については rt <--- c のように表記することにより、組み込み関数のセマンティックスを満たすためには入出力によってはレジスタからレジスタへのコピー（この場合 *c* から *rt* へのコピー）が必要である可能性があることを示している。入力 *c* が出力 *d* と同じである場合、コピーは行なわれない。
- 個別組み込み関数へマップされない総称組み込み関数については、表中の「個別組み込み関数」欄に「not applicable」を意味する N/A という略語が記されている。

2.3. 定数生成命令に対応する組み込み関数

spu_splats: Splat Scalar to a Vector

```
d = spu_splats(a)
```

単一のスカラ値を同一型のベクタの全ての要素に複製します。結果をベクタ *d* へ代入します。

表 2-13: Splat Scalar to a Vector

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	a		
vector unsigned char	unsigned char	N/A	SHUFB d, a, a, pattern
vector signed char	signed char		
vector unsigned short	unsigned short		
vector signed short	signed short		
vector unsigned int	unsigned int		
vector signed int	signed int		
vector unsigned long long	unsigned long long		
vector signed long long	signed long long		
vector float	float		
vector double	double		
vector unsigned char	unsigned char (リテラル)	N/A	IL d, a *or* ILA d, a *or* ILH d, a&0xFFFF *or* ILHU d, a>>16 *or* ILHU d, a>>16; IOHL d, a *or* FSMBI d, a
vector signed char	signed char (リテラル)		
vector unsigned short	unsigned short (リテラル)		
vector signed short	signed short (リテラル)		
vector unsigned int	unsigned int (リテラル)		
vector signed int	signed int (リテラル)		
vector unsigned long long	unsigned long long (リテラル)		
vector signed long long	signed long long (リテラル)		
vector float	float (リテラル)		
vector double	double (リテラル)		

2.4. 変換命令に対応する組み込み関数

spu_convtf: Convert Vector to Float

```
d = spu_convtf(a, scale)
```

ベクタ *a* の各要素を浮動小数点値に変換し、2^{scale} で除算します。*scale* の許容範囲は 0～127 です。この範囲を超えた値を渡すとエラーとして通知し、コンパイルを停止します。結果をベクタ *d* へ代入します。

表 2-14: Convert an Integer Vector to a Vector Float

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	scale		
vector float	vector unsigned int	unsigned int (7-bit literal)	*d* = si_cuflt(*a*, *scale*)	CUFLT d, a, scale
vector float	vector signed int		*d* = si_csflt(*a*, *scale*)	CSFLT d, a, scale

spu_convts: Convert Floating-Point Vector to Signed Integer Vector

```
d = spu_convts(a, scale)
```

ベクタ *a* の各要素を 2^{scale} で乗算し、結果を符号付き整数に変換します。この中間結果が（2^{31}-1）より大きい場合は (2^{31}-1) に飽和演算し、-2^{31} より小さい場合は、-2^{31} に飽和演算します。*scale* の許容範囲は 0～127 です。この範囲を超えた値を渡すとエラーとして通知しコンパイルを停止します。それ以外の場合結果をベクタ *d* の対応する要素へ代入します。

表 2-15: Convert a Vector Float to a Signed Integer Vector

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	scale		
vector signed int	vector float	unsigned int (7-bit literal)	*d* = si_cflts(*a*, *scale*)	CFLTS d, a, scale

spu_convtu: Convert Floating-Point Vector to Unsigned Integer Vector

```
d = spu_convtu(a, scale)
```

ベクタ *a* を 2^{scale} で乗算し、結果を符号なし整数に変換します。この中間結果が（2^{32}-1）より大きい場合は (2^{32}-1) に飽和演算し、負の値である場合は、0 に飽和演算します。*scale* の許容範囲は 0～127 です。この範囲を超えた値を渡すとエラーとして通知しコンパイルは停止されますが、それ以外の場合結果をベクタ *d* の対応する要素へ代入します。

表 2-16: Convert a Vector Float to an Unsigned Integer Vector

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	scale		
vector unsigned int	vector float	unsigned int (7-bit literal)	*d* = si_cfltu(*a*, *scale*)	CFLTU d, a, scale

spu_extend: Sign Extend Vector

```
d = spu_extend(a)
```

a が固定小数点ベクタである場合、このベクタの各奇数番目の要素を符号拡張しベクタ *d* の対応する要素へ代入します。*a* が浮動小数点ベクタである場合、このベクタの各偶数番目の要素を拡張しベクタ *d* の対応する要素へ代入します。

表 2-17: Sign Extend Vector

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	a		
vector signed short	vector signed char	*d* = si_xsbh(*a*)	XSBH d, a
vector signed int	vector signed short	*d* = si_xshw(*a*)	XSHW d, a
vector signed long long	vector signed int	*d* = si_xswd(*a*)	XSWD d, a
vector double	vector float	*d* = si_fesd(*a*)	FESD d, a

spu_roundtf: Round Vector Double to Vector Float

```
d = spu_roundtf(a)
```

ベクタ a の各ダブルワード要素を単精度浮動小数点値に丸め、ベクタ d の偶数番目の要素に代入します。また、ベクタ d の奇数番目の要素に 0 を代入します。

表 2-18: Round a Vector Double to a Float

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	a		
vector float	vector double	*d* = si_frds(*a*)	FRDS d, a

2.5. 算術演算命令に対応する組み込み関数

spu_add: Vector Add

```
d = spu_add(a, b)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素に加算します。*b* がスカラである場合、そのスカラを各要素に複製し、*a* に加算します。オーバーフローやキャリーは検出せず、飽和演算は行ないません。結果をベクタ *d* の対応する要素へ代入します。

表 2-19: Vector Add

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector signed int	vector signed int	vector signed int	*d* = si_a(*a*, *b*)	A d, a, b
vector unsigned int	vector unsigned int	vector unsigned int		
vector signed short	vector signed short	vector signed short	*d* = si_ah(*a*, *b*)	AH d, a, b
vector unsigned short	vector unsigned short	vector unsigned short		
vector signed int	vector signed int	10-bit signed int (リテラル)	*d* = si_ai(*a*, *b*)	AI d, a, b
vector unsigned int	vector unsigned int			
vector signed int	vector signed int	int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector unsigned int	vector unsigned int	unsigned int		
vector signed short	vector signed short	10-bit signed short (リテラル)	*d* = si_ahi(*a*, *b*)	AHI d, a, b
vector unsigned short	vector unsigned short			
vector signed short	vector signed short	short	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector unsigned short	vector unsigned short	unsigned short		
vector float	vector float	vector float	*d* = si_fa(*a*, *b*)	FA d, a, b
vector double	vector double	vector double	*d* = si_dfa(*a*, *b*)	DFA d, a, b

spu_addx: Vector Add Extended

```
d = spu_addx(a, b, c)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素およびベクタ *c* の対応する要素の LSB へ加算します。結果をベクタ *d* の対応する要素へ代入します。

表 2-20: Vector Add Extended

戻り値／引数の型				個別組み込み関数	アセンブリ命令へのマップ
d	a	b	c		
vector signed int	vector signed int	vector signed int	vector signed int	*d* = si_addx(*a*, *b*, *c*)	rt <--- c ADDX rt, a, b d <--- rt
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int		

spu_genb: Vector Generate Borrow

```
d = spu_genb(a, b)
```

ベクタ *b* の各要素をベクタ *a* の対応する要素から減算します。発生したボローをベクタ *d* の対応する要素の LSB へ代入し、*d* の残りのビットを 0 に設定します。

表 2-21: Vector Generate Borrow

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector signed int	vector signed int	vector signed int	*d* = si_bg(*b*, *a*)	BG rt, b, a
vector unsigned int	vector unsigned int	vector unsigned int		

spu_genbx: Vector Generate Borrow Extended

```
d = spu_genbx(a, b, c)
```

ベクタ *b* の各要素をベクタ *a* の対応する要素から減算します。ベクタ *c* の対応する要素の LSB が 0 である場合は、結果から更に 1 を減算します。結果が 0 よりも小さい場合は、ベクタ *d* の対応する要素へ 1 を代入し、そうでない場合は *d* の対応する要素へ 0 を代入します。

表 2-22: Vector Generate Borrow Extended

戻り値／引数の型				個別組み込み関数	アセンブリ命令へのマップ
d	a	b	c		
vector signed int	vector signed int	vector signed int	vector signed int	*d* = si_bgx(*b*, *a*, *c*)	rt <--- c BGX rt, b, a d <--- rt
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int		

spu_genc: Vector Generate Carry

```
d = spu_genc(a, b)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素へ加算します。発生したキャリーをベクタ *d* の対応する要素の LSB へ代入し、*d* の残りのビットを 0 に設定します。

表 2-23: Vector Generate Carry

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector signed int	vector signed int	vector signed int	*d* = si_cg(*a*, *b*)	CG rt, a, b
vector unsigned int	vector unsigned int	vector unsigned int		

spu_gencx: Vector Generate Carry Extended

```
d = spu_gencx(a, b, c)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素およびベクタ *c* の対応する要素の LSB へ加算します。発生したキャリーをベクタ *d* の対応する要素の LSB へ代入し、*d* の残りのビットを 0 に設定します。

表 2-24: Vector Generate Carry Extended

戻り値／引数の型				個別組み込み関数	アセンブリ命令へのマップ
d	a	b	c		
vector signed int	vector signed int	vector signed int	vector signed int	*d* = si_cgx(*a*, *b*, *c*)	rt <--- c CGX rt, a, b d <--- rt
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int		

spu_madd: Vector Multiply and Add

```
d = spu_madd(a, b, c)
```

ベクタ *a* の各要素をベクタ *b* で乗算し、ベクタ *c* の対応する要素に加算し、ベクタ *d* の対応する要素に代入します。整数の積和演算の場合、乗算を行なう前にベクタ *a* およびベクタ *b* の奇数番目の要素を 32 ビット整数へ符号拡張します。

表 2-25: Vector Multiply and Add

戻り値／引数の型				個別組み込み関数	アセンブリ命令へのマップ
d	a	b	c		
vector signed int	vector signed short	vector signed short	vector signed int	*d* = si_mpya(*a*, *b*, *c*)	MPYA d, a, b, c
vector float	vector float	vector float	vector float	*d* = si_fma(*a*, *b*, *c*)	FMA d, a, b, c
vector double	vector double	vector double	vector double	*d* = si_dfma(*a*, *b*, *c*)	rt <--- c DFMA rt, a, b d <--- rt

spu_mhhadd: Vector Multiply High High and Add

```
d = spu_mhhadd(a, b, c)
```

ベクタ *a* の各偶数番目の要素をベクタ *b* の対応する偶数番目の要素で乗算し、その結果得られる 32 ビット値をベクタ *c* の対応する要素へ加算します。結果をベクタ *d* の対応する要素へ代入します。

表 2-26: Vector Multiply High High and Add

戻り値／引数の型				個別組み込み関数	アセンブリ命令へのマップ
d	a	b	c		
vector signed int	vector signed short	vector signed short	vector signed int	*d* = si_mpyhha (*a*, *b*, *c*)	rt <--- c MPYHHA rt, a, b d <--- rt
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int	*d* = si_mpyhhau (*a*, *b*, *c*)	rt <--- c MPYHHAU rt, a, b d <--- rt

spu_msub: Vector Multiply and Subtract

```
d = spu_msub(a, b, c)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素で乗算し、その積からベクタ *c* の対応する要素を減算します。結果をベクタ *d* の対応する要素へ代入します。

表 2-27: Vector Multiply and Subtract

戻り値／引数の型				個別組み込み関数	アセンブリ命令へのマップ
d	a	b	c		
vector float	vector float	vector float	vector float	*d* = si_fms(*a*, *b*, *c*)	FMS d, a, b, c
vector double	vector double	vector double	vector double	*d* = si_dfms(*a*, *b*, *c*)	rt <--- c DFMS rt, a, b d <--- rt

spu_mul: Vector Multiply

```
d = spu_mul(a, b)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素で乗算し、結果をベクタ *d* の対応する要素へ代入します。

表 2-28: Vector Multiply

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector float	vector float	vector float	*d* = si_fm(*a*, *b*)	FM d, a, b
vector double	vector double	vector double	*d* = si_dfm(*a*, *b*)	DFM d, a, b

spu_mulh: Vector Multiply High

```
d = spu_mulh(a, b)
```

ベクタ *a* の各偶数番目の要素をベクタ *b* における右隣の奇数番目の要素で乗算し、その積を 16 ビット左にシフトし、ベクタ *d* の対応する要素へ代入します。左にシフトアウトされたビットは破棄し、右端から 0 をシフトインします。

表 2-29: Vector Multiply High

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector signed int	vector signed short	vector signed short	*d* = si_mpyh(*a*, *b*)	MPYH d, a, b

spu_mule: Vector Multiply Even

```
d = spu_mule(a, b)
```

ベクタ *a* の各偶数番目の要素をベクタ *b* の対応する偶数番目の要素で乗算し、その結果得られる 32 ビット値をベクタ *d* の対応する要素へ代入します。

表 2-30: Vector Multiply Even

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector signed int	vector signed short	vector signed short	*d* = si_mpyhh(*a*, *b*)	MPYHH d, a, b
vector unsigned int	vector unsigned short	vector unsigned short	*d* = si_mpyhhu(*a*, *b*)	MPYHHU d, a, b

spu_mulo: Vector Multiply Odd

```
d = spu_mulo (a, b)
```

ベクタ *a* の各奇数番目の要素をベクタ *b* の対応する要素で乗算します。*b* がスカラである場合、そのスカラを各要素に複製し、*a* で乗算します。結果をベクタ *d* へ代入します。

表 2-31: Vector Multiply Odd

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector signed int	vector signed short	vector signed short	*d* = si_mpy (*a*, *b*)	MPY d, a, b
		10-bit signed short (リテラル)	*d* = si_mpyi(*a*, *b*)	MPYI d, a, b
		signed short	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned int	vector unsigned short	vector unsigned short	*d* = si_mpyu(*a*, *b*)	MPYU d, a, b
		10-bit signed short (リテラル)	*d* = si_mpyui(*a*, *b*)	MPYUI d, a, b
		unsigned short	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	

spu_mulsr: Vector Multiply and Shift Right

```
d = spu_mulsr(a, b)
```

ベクタ *a* の各奇数番目の要素をベクタ *b* の対応する奇数番目の要素で乗算し、その積である 32 ビット値の左端（上位）16 ビットを符号拡張し、ベクタ *d* の対応する 32 ビット要素へ代入します。

表 2-32: Vector Multiply and Shift Right

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector signed int	vector signed short	vector signed short	*d* = si_mpys(*a*, *b*)	MPYS d, a, b

spu_nmadd: Negative Vector Multiply and Add

```
d = spu_nmadd(a, b, c)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素で乗算し、その積をベクタ *c* の対応する要素へ加算します。結果を符号反転し、ベクタ *d* の対応する要素へ代入します。

表 2-33: Negative Vector Multiply and Add

戻り値／引数の型				個別組み込み関数	アセンブリ命令へのマップ
d	a	b	c		
vector double	vector double	vector double	vector double	*d* = si_dfnma(*a*, *b*, *c*)	rt <-- c DFNMA rt, a, b d <-- rt

spu_nmsub: Negative Vector Multiply and Subtract

```
d = spu_nmsub(a, b, c)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素で乗算し、その結果をベクタ *c* の対応する要素から減算します。結果をベクタ *d* の対応する要素へ代入します。

表 2-34: Negative Vector Multiply and Subtract

戻り値／引数の型				個別組み込み関数	アセンブリ命令へのマップ
d	a	b	c		
vector float	vector float	vector float	vector float	*d* = si_fnms(*a*, *b*, *c*)	FNMS d, a, b, c
vector double	vector double	vector double	vector double	*d* = si_dfnms(*a*, *b*, *c*)	rt <--- c DFNMS rt, a, b d <--- rt

spu_re: Vector Floating-point Reciprocal Estimate

```
d = spu_re(a)
```

ベクタ *a* の各要素についてその浮動小数点逆数の推定値を算出し、結果をベクタ *d* の対応する要素へ代入します。結果として得られる推定値の精度は 12 ビットです。

表 2-35: Vector Floating-Point Reciprocal Estimate

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	a		
vector float	vector float	*t* = si_frest(*a*) *d* = si_fi(*a*, *t*)	FREST d, a FI d, a, d

spu_rsqrte: Vector Floating-Point Reciprocal Square Root Estimate

```
d = spu_rsqrte(a)
```

ベクタ *a* の各要素についてその浮動小数点逆数の平方根の推定値を算出し、結果をベクタ *d* の対応する要素へ代入します。結果として得られる推定値の精度は 12 ビットです。

表 2-36: Vector Floating-Point Reciprocal Square Root Estimate

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	a		
vector float	vector float	*t* = si_frsqest(*a*) *d* = si_fi(*a*, *t*)	FRSQEST d, a FI d, a, d

spu_sub: Vector Subtract

```
d = spu_sub(a, b)
```

ベクタ *b* の各要素をベクタ *a* の対応する要素より減算します。*a* がスカラである場合、そのスカラを *a* の各要素に複製し、*b* を *a* の対応する要素より減算します。オーバーフローやキャリーは検出しません。結果をベクタ *d* の対応する要素へ代入します。

表 2-37: Vector Subtract

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector signed short	vector signed short	vector signed short	*d* = si_sfh(*b*, *a*)	SFH d, b, a
vector unsigned short	vector unsigned short	vector unsigned short		
vector signed int	vector signed int	vector signed int	*d* = si_sf(*b*, *a*)	SF d, b, a
vector unsigned int	vector unsigned int	vector unsigned int		
vector signed int	10-bit signed int (リテラル)	vector signed int	*d* = si_sfi(*b*, *a*)	SFI d, b, a
vector unsigned int		vector unsigned int		
vector signed int	int	vector signed int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector unsigned int	unsigned int	vector unsigned int		
vector signed short	10-bit signed short (リテラル)	vector signed short	*d* = si_sfhi(*b*, *a*)	SFHI d, b, a
vector unsigned short		vector unsigned short		
vector signed short	short	vector signed short	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector unsigned short	unsigned short	vector unsigned short		
vector float	vector float	vector float	*d* = si_fs(*a*, *b*)	FS d, a, b
vector double	vector double	vector double	*d* = si_dfs(*a*, *b*)	DFS d, a, b

spu_subx: Vector Subtract Extended

```
d = spu_subx(a, b, c)
```

ベクタ *b* の各要素をベクタ *a* の対応する要素より減算します。ベクタ *c* の対応する要素の LSB が 0 である場合は減算結果から更に 1 を減算します。結果をベクタ *d* の対応する要素へ代入します。

表 2-38: Vector Subtract Extended

戻り値／引数の型				個別組み込み関数	アセンブリ命令へのマップ
d	a	b	c		
vector signed int	vector signed int	vector signed int	vector signed int	*d* = si_sfx(*b*, *a*, *c*)	rt <--- c SFX rt, b, a d <--- rt
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int		

2.6. バイト演算命令に対応する組み込み関数

spu_absd: Element-Wise Absolute Difference

```
d = spu_absd(a, b)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素より減算し、結果の絶対値をベクタ *d* の対応する要素へ代入します。

表 2-39: Element-Wise Absolute Difference

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned char	vector unsigned char	vector unsigned char	*d* = si_absdb(*a*, *b*)	ABSDB d, a, b

spu_avg: Average of Two Vectors

```
d = spu_avg(a, b)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素に 1 を加算した値へ加算します。結果を 1 ビット右へシフトし、ベクタ *d* の対応する要素へ代入します。

表 2-40: Average of Two Vectors

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned char	vector unsigned char	vector unsigned char	*d* = si_avgb(*a*, *b*)	AVGB d, a, b

spu_sumb: Sum Bytes into Shorts

```
d = spu_sumb(a, b)
```

ベクタ *b* の 4 つの要素の和をとりベクタ *d* の対応する偶数番目の要素へ代入します。また、ベクタ *a* の 4 つの要素の和をとりベクタ *d* の対応する奇数番目の要素へ代入します。

表 2-41: Sum Bytes into Shorts

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned short	vector unsigned char	vector unsigned char	*d* = si_sumb(*a*, *b*)	SUMB d, a, b

2.7. 比較、分岐、および停止命令に対応する組み込み関数

spu_bisled: Branch Indirect and Set Link if External Data

```
(void) spu_bisled(func)
(void) spu_bisled_d(func)
(void) spu_bisled_e(func)
```

チャネル 0 (イベントステータス)のカウントを確認します。カウントが 0 である場合は、次の命令を実行します。0 ではない場合、関数 *func* を呼び出します。パラメータ *func* はパラメータおよび返り値を持たない関数の名前あるいはそのような関数へのポインタです。*func* が呼ばれると spu_bisled_d と spu_bisled_e 形式の組み込み関数は以下のいずれかを行ないます。

- 割り込みを禁止 （spu_bisled_d の場合）
- 割り込みを許可 （spu_bisled_e の場合）

プログラミングの注意: bisled 命令は同期ソフトウェア割り込みとして振舞うとの想定に基づき、bisled 命令の対象となる関数 func では全ての揮発性レジスタを非揮発性レジスタとして扱う必要があるため、標準の呼び出し規約には従っていません。標準の呼び出し規約の詳細については「*SPU Application Binary Interface 仕様書*」を参照してください。

分岐予測については func が呼び出されることはないと想定しています。このため、spu_bisled() 組み込み関数の使用により分岐ヒント命令が挿入されることはありません。

表 2-42: Branch Indirect and Set Link if External Data

総称組み込み関数形式	func	個別組み込み関数	アセンブリ命令へのマップ
spu_bisled	void (*func) ()	si_bisled(*func*)	BISLED $LR, func
spu_bisled_d		si_bisledd(*func*)	BISLEDD $LR, func
spu_bisled_e		si_bislede(*func*)	BISLEDE $LR, func

spu_cmpabseq: Element-Wise Compare Absolute Equal

```
d = spu_cmpabseq(a, b)
```

ベクタ *a* の各要素の絶対値をベクタ *b* の対応する各要素の絶対値と比較します。値が等しい場合はベクタ *d* の対応する要素の全ビットを 1 に設定し、等しくない場合はベクタ *d* の対応する要素の全ビットを 0 に設定します。

表 2-43: Element-Wise Compare Absolute Equal

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned int	vector float	vector float	*d* = si_fcmeq(*a*, *b*)	FCMEQ d, a, b

spu_cmpabsgt: Element-Wise Compare Absolute Greater Than

```
d = spu_cmpabsgt(a, b)
```

ベクタ *a* の各要素の絶対値をベクタ *b* の対応する各要素の絶対値と比較します。ベクタ *a* の要素がベクタ *b* の対応する要素より大きい場合、ベクタ *d* の対応する要素の全ビットを 1 に設定し、そうでない場合はベクタ *d* の対応する要素の全ビットを 0 に設定します。

表 2-44: Element-Wise Compare Absolute Greater Than

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned int	vector float	vector float	*d* = si_fcmgt(*a*, *b*)	FCMGT d, a, b

spu_cmpeq: Element-Wise Compare Equal

```
d = spu_cmpeq(a, b)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素と比較します。*b* がスカラである場合、そのスカラを各要素に複製した後に比較を行ないます。これらのオペランド値が等しい場合はベクタ *d* の対応する要素の全ビットを 1 に設定します。等しくない場合はベクタ *d* の対応する要素の全ビットを 0 に設定します。

表 2-45: Element-Wise Compare Equal

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned char	vector signed char	vector signed char	*d* = si_ceqb(*a*, *b*)	CEQB d, a, b
	vector unsigned char	vector unsigned char		
vector unsigned short	vector signed short	vector signed short	*d* = si_ceqh(*a*, *b*)	CEQH d, a, b
	vector unsigned short	vector unsigned short		
vector unsigned int	vector signed int	vector signed int	*d* = si_ceq(*a*, *b*)	CEQ d, a, b
	vector unsigned int	vector unsigned int		
	vector float	vector float	*d* = si_fceq(*a*, *b*)	FCEQ d, a, b
vector unsigned char	vector signed char	10-bit signed int (リテラル)	*d* = si_ceqbi(*a*, *b*)	CEQBI d, a, b
	vector unsigned char			
	vector signed char	signed char	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
	vector unsigned char	unsigned char		
vector unsigned short	vector signed short	10-bit signed int (リテラル)	*d* = si_ceqhi(*a*, *b*)	CEQHI d, a, b
	vector unsigned short			
	vector signed short	signed short	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
	vector unsigned short	unsigned short		
vector unsigned int	vector signed int	10-bit signed int (リテラル)	*d* = si_ceqi(*a*, *b*)	CEQI d, a, b
	vector unsigned int			
	vector signed int	signed int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
	vector unsigned int	unsigned int		

spu_cmpgt: Element-Wise Compare Greater Than

```
d = spu_cmpgt(a, b)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素と比較します。*b* がスカラである場合、そのスカラを各要素に複製した後に比較を行ないます。ベクタ *a* の要素がベクタ *b* の対応する要素より大きい場合、ベクタ *d* の対応する要素の全ビットを 1 に設定し、そうでない場合はベクタ *d* の対応する要素の全ビットを 0 に設定します。

表 2-46: Element-Wise Compare Greater Than

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned char	vector signed char	vector signed char	*d* = si_cgtb(*a*, *b*)	CGTB d, a, b
		10-bit signed int (リテラル)	*d* = si_cgtbi(*a*, *b*)	CGTBI d, a, b
		signed char	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
	vector unsigned char	vector unsigned char	d = si_clgtb(*a*, *b*)	CLGTB d, a, b
		10-bit signed int (リテラル)	d = si_clgtbi(*a*, *b*)	CLGTBI d, a, b
		unsigned char	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned short	vector signed short	vector signed short	*d* = si_cgth(*a*, *b*)	CGTH d, a, b
		10-bit signed int (リテラル)	*d* = si_cgthi(*a*, *b*)	CGTHI d, a, b
		signed short	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
	vector unsigned short	vector unsigned short	*d* = si_clgth(*a*, *b*)	CLGTH d, a, b
		10-bit signed int (リテラル)	*d* = si_clgthi(*a*, *b*)	CLGTHI d, a, b
		unsigned short	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector unsigned int	vector signed int	vector signed int	*d* = si_cgt(*a*, *b*)	CGT d, a, b
		10-bit signed int (リテラル)	*d* = si_cgti(*a*, *b*)	CGTI d, a, b
		signed int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
	vector unsigned int	vector unsigned int	*d* = si_clgt(*a*, *b*)	CLGT d, a, b
		10-bit signed int (リテラル)	*d* = si_clgti(*a*, *b*)	CLGTI d, a, b
		unsigned int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
	vector float	vector float	*d* = si_fcgt(*a*, *b*)	FCGT d, a, b

spu_hcmpeq: Halt If Compare Equal

```
(void) spu_hcmpeq(a, b)
```

a と *b* の値を比較します。値が等しい場合はプログラムの実行を停止します。

表 2-47: Halt If Compare Equal

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ [1, 2]
a	b		
int	int (非リテラル)	si_heq(*a*, *b*)	HEQ rt, a, b
unsigned int	unsigned int (非リテラル)		
int	10-bit signed int (リテラル)	si_heqi(*a*, *b*)	HEQI rt, a, b
unsigned int			

[1] 10 ビット符号付き値として表現できない即値は 14ページの「2.2.1. スカラ型のオペランドをとる組み込み関数のマップ」に記載されているのと同じ方法で構成されます。

[2] 偽のターゲットであるパラメータ *rt* は近傍命令のレジスタ使用状態に応じて最適な値が選択されます。

spu_hcmpgt: Halt If Compare Greater Than

```
(void) spu_hcmpgt(a, b)
```

a と *b* の値を比較します。*a* が *b* より大きい場合はプログラムの実行を停止します。

表 2-48: Halt If Compare Greater Than

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ [1,2]
a	b		
int	int (非リテラル)	si_hgt(*a*, *b*)	HGT rt, a, b
unsigned int	unsigned int (非リテラル)	si_hlgt(*a*, *b*)	HLGT rt, a, b
int	10-bit signed int (リテラル)	si_hgti(*a*, *b*)	HGTI rt, a, b
unsigned int	10-bit signed int (リテラル)	si_hlgti(*a*, *b*)	HLGTI rt, a, b

[1] 10 ビット符号付き値として表現できない即値は 14ページの「2.2.1. スカラ型のオペランドをとる組み込み関数のマップ」に記載されているのと同じ方法で構成されます。

[2] 偽のターゲットであるパラメータ *rt* は近傍命令のレジスタ使用状態に応じて最適な値が選択されます。

2.8. ビット演算およびマスク演算命令に対応する組み込み関数

spu_cntb: Vector Count Ones for Bytes

```
d = spu_cntb(a)
```

ベクタ *a* の各要素中にある 1 のビット数をカウントし、その値をベクタ *d* の対応する要素へ代入します。

表 2-49: Vector Count Ones for Bytes

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	a		
vector unsigned char	vector unsigned char	si_cntb	CNTB d, a
	vector signed char		

spu_cntlz: Vector Count Leading Zeros

```
d = spu_cntlz(a)
```

ベクタ *a* の各要素中で最初の 1 のビットの左にある 0 のビット数をカウントし、その値をベクタ *d* の対応する要素へ代入します。

表 2-50: Vector Count Leading Zeros

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	a		
vector unsigned int	vector signed int	*d* = si_clz(*a*)	CLZ d, a
	vector unsigned int		
	vector float		

spu_gather: Gather Bits from Elements

```
d = spu_gather(a)
```

ベクタ *a* の各要素の LSB を集めて連結し、ベクタ *d* の要素 0 の下位ビットへ格納して返します。連結されるビットの数は、バイトベクタの場合 16 ビット、ハーフワードベクタの場合 8 ビット、ワードベクタの場合 4 ビットとなります。ベクタ *d* の要素 0 の残りのビットおよびベクタ *d* の他の要素は全て 0 に設定されます。

表 2-51: Gather Bits from Elements

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	a		
vector unsigned int	vector unsigned char	*d* = si_gbb(*a*)	GBB d, a
	vector signed char		
	vector unsigned short	*d* = si_gbh(*a*)	GBH d, a
	vector signed short		
	vector unsigned int	*d* = si_gb(*a*)	GB d, a
	vector signed int		
	vector float		

spu_maskb: Form Select Byte Mask

```
d = spu_maskb(a)
```

a の下位 16 ビットについて各ビットを 8 回複製することにより 128 ビットのベクタマスクを作成しベクタ *d* へ代入します。

表 2-52: Form Select Byte Mask

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	a		
vector unsigned char	unsigned short	*d* = si_fsmb(*a*)	FSMB d, a
	signed short		
	unsigned int		
	signed int		
	16-bit unsigned int (リテラル)	*d* = si_fsmbi(*a*)	FSMBI d, a

spu_maskh: Form Select Halfword Mask

```
d = spu_maskh(a)
```

a の下位 8 ビットについて各ビットを 16 回複製することにより 128 ビットのベクタマスクを作成しベクタ *d* へ代入します。

表 2-53: Form Select Halfword Mask

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	a		
vector unsigned short	unsigned char	*d* = si_fsmh(*a*)	FSMH d, a
	signed char		
	unsigned short		
	signed short		
	unsigned int		
	signed int		

spu_maskw: Form Select Word Mask

```
d = spu_maskw(a)
```

a の下位 4 ビットについて各ビットを 32 回複製することにより 128 ビットのベクタマスクを作成しベクタ *d* へ代入します。

表 2-54: Form Select Word Mask

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	a		
vector unsigned int	unsigned char	*d* = si_fsm(*a*)	FSM d, a
	signed char		
	unsigned short		
	signed short		
	unsigned int		
	signed int		

spu_sel: Select Bits

```
d = spu_sel(a, b, pattern)
```

128 ビットのベクタである *pattern* の各ビットの値に応じてベクタ *a* またはベクタ *b* から対応するビットを選択します。*pattern* のビットが 0 の場合は *a* から対応するビットを選択し、そうでない場合は *b* から選択します。結果をベクタ *d* へ代入します。

表 2-55: Select Bits

戻り値／引数の型				個別組み込み関数	アセンブリ命令へのマップ
d	a	b	pattern		
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	*d* = si_selb(*a*, *b*, *pattern*)	SELB d, a, b, pattern
vector signed char	vector signed char	vector signed char			
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short		
vector signed short	vector signed short	vector signed short			
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int		
vector signed int	vector signed int	vector signed int			
vector float	vector float	vector float			
vector unsigned long long	vector unsigned long long	vector unsigned long long	vector unsigned long long		
vector signed long long	vector signed long long	vector signed long long			
vector double	vector double	vector double			

spu_shuffle: Shuffle Two Vectors of Bytes

```
d = spu_shuffle(a, b, pattern)
```

*pattern*の各バイトの値を調べ、その値に応じて 図 2-2に示すようにバイトを生成し、結果をベクタ *d*の対応するバイトへ代入します。

図 2-2: シャッフルパターン

pattern の各バイトの値 (バイナリ)	生成されるバイト
10xxxxxx	0x00
110xxxxx	0xFF
111xxxxx	0x80
上記以外	*pattern* の下位 5 ビットでアドレス指定された連結値 (a||b)のバイト

表 2-56: Shuffle Two Vectors of Bytes

戻り値／引数の型				個別組み込み関数	アセンブリ命令へのマップ
d	a	b	pattern		
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	*d* = si_shufb(*a*, *b*, *pattern*)	SHUFB d, a, b, pattern
vector signed char	vector signed char	vector signed char			
vector unsigned short	vector unsigned short	vector unsigned short			
vector signed short	vector signed short	vector signed short			
vector unsigned int	vector unsigned int	vector unsigned int			
vector signed int	vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long	vector signed long long			
vector float	vector float	vector float			
vector double	vector double	vector double			

2.9. 論理演算命令に対応する組み込み関数

spu_and: Vector Bit-Wise AND

```
d = spu_and(a, b)
```

ベクタ *a* の各ビットをベクタ *b* の対応するビットと論理 AND 演算します。*b* がスカラである場合、そのスカラを各要素に複写した後に論理 AND 演算を行ないます。結果をベクタ *d* の対応するビットへ代入します。

表 2-57: Vector Bit-Wise AND

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned char	vector unsigned char	vector unsigned char	*d* = si_and(*a*, *b*)	AND d, a, b
vector signed char	vector signed char	vector signed char		
vector unsigned short	vector unsigned short	vector unsigned short		
vector signed short	vector signed short	vector signed short		
vector unsigned int	vector unsigned int	vector unsigned int		
vector signed int	vector signed int	vector signed int		
vector unsigned long long	vector unsigned long long	vector unsigned long long		
vector signed long long	vector signed long long	vector signed long long		
vector float	vector float	vector float		
vector double	vector double	vector double		
vector unsigned char	vector unsigned char	10-bit signed int (リテラル)	*d* = si_andbi(*a*, *b*)	ANDBI d, a, b
vector signed char	vector signed char			
vector unsigned char	vector unsigned char	unsigned char	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed char	vector signed char	signed char		
vector unsigned short	vector unsigned short	10-bit signed int (リテラル)	*d* = si_andhi(*a*, *b*)	ANDHI d, a, b
vector signed short	vector signed short			
vector unsigned short	vector unsigned short	unsigned short	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed short	vector signed short	signed short		
vector unsigned int	vector unsigned int	10-bit signed int (リテラル)	*d* = si_andi(*a*, *b*)	ANDI d, a, b
vector signed int	vector signed int			
vector unsigned int	vector unsigned int	unsigned int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed int	vector signed int	signed int		

spu_andc: Vector Bit-Wise AND with Complement

```
d = spu_andc(a, b)
```

ベクタ *a* の各ビットをベクタ *b* の対応するビットの補数と論理 AND 演算し、その結果をベクタ *d* の対応するビットへ代入します。

表 2-58: Vector Bit-Wise AND with Complement

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned char	vector unsigned char	vector unsigned char	*d* = si_andc(*a*, *b*)	ANDC d, a, b
vector signed char	vector signed char	vector signed char		
vector unsigned short	vector unsigned short	vector unsigned short		
vector signed short	vector signed short	vector signed short		
vector unsigned int	vector unsigned int	vector unsigned int		
vector signed int	vector signed int	vector signed int		
vector unsigned long long	vector unsigned long long	vector unsigned long long		
vector signed long long	vector signed long long	vector signed long long		
vector float	vector float	vector float		
vector double	vector double	vector double		

spu_eqv: Vector Bit-Wise Equivalent

```
d = spu_eqv(a, b)
```

ベクタ *a* の各ビットをベクタ *b* の対応するビットと比較します。*a* と *b* の対応するビット値が等しい場合、ベクタ *d* の対応するビットを 1 に設定し、そうでない場合は 0 に設定します。

表 2-59: Vector Bit-Wise Equivalent

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned char	vector unsigned char	vector unsigned char	*d* = si_eqv(*a*, *b*)	EQV d, a, b
vector signed char	vector signed char	vector signed char		
vector unsigned short	vector unsigned short	vector unsigned short		
vector signed short	vector signed short	vector signed short		
vector unsigned int	vector unsigned int	vector unsigned int		
vector signed int	vector signed int	vector signed int		
vector unsigned long long	vector unsigned long long	vector unsigned long long		
vector signed long long	vector signed long long	vector signed long long		
vector float	vector float	vector float		
vector double	vector double	vector double		

spu_nand: Vector Bit-Wise Complement of AND

```
d = spu_nand(a, b)
```

ベクタ *a* の各ビットをベクタ *b* の対応するビットと論理 AND 演算し、結果の補数をベクタ *d* の対応するビットへ代入します。

表 2-60: Vector Bit-Wise Complement of AND

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned char	vector unsigned char	vector unsigned char	*d* = si_nand(*a*, *b*)	NAND d,a, b
vector signed char	vector signed char	vector signed char		
vector unsigned short	vector unsigned short	vector unsigned short		
vector signed short	vector signed short	vector signed short		
vector unsigned int	vector unsigned int	vector unsigned int		
vector signed int	vector signed int	vector signed int		
vector unsigned long long	vector unsigned long long	vector unsigned long long		
vector signed long long	vector signed long long	vector signed long long		
vector float	vector float	vector float		
vector double	vector double	vector double		

spu_nor: Vector Bit-Wise Complement of OR

```
d = spu_nor(a, b)
```

ベクタ *a* の各ビットをベクタ *b* の対応するビットと論理 OR 演算し、結果の補数をベクタ *d* の対応するビットへ代入します。

表 2-61: Vector Bit-Wise Complement of OR

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned char	vector unsigned char	vector unsigned char	*d* = si_nor(*a*, *b*)	NOR d,a, b
vector signed char	vector signed char	vector signed char		
vector unsigned short	vector unsigned short	vector unsigned short		
vector signed short	vector signed short	vector signed short		
vector unsigned int	vector unsigned int	vector unsigned int		
vector signed int	vector signed int	vector signed int		
vector unsigned long long	vector unsigned long long	vector unsigned long long		
vector signed long long	vector signed long long	vector signed long long		
vector float	vector float	vector float		
vector double	vector double	vector double		

spu_or: Vector Bit-Wise OR

```
d = spu_or(a, b)
```

ベクタ *a* の各ビットをベクタ *b* の対応するビットと論理 OR 演算します。*b* がスカラである場合、そのスカラを各要素に複写した後に論理 OR 演算を行ないます。結果をベクタ *d* の対応するビットへ代入します。

表 2-62: Vector Bit-Wise OR

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned char	vector unsigned char	vector unsigned char	*d* = si_or(*a*, *b*)	OR d, a, b
vector signed char	vector signed char	vector signed char		
vector unsigned short	vector unsigned short	vector unsigned short		
vector signed short	vector signed short	vector signed short		
vector unsigned int	vector unsigned int	vector unsigned int		
vector signed int	vector signed int	vector signed int		
vector unsigned long long	vector unsigned long long	vector unsigned long long		
vector signed long long	vector signed long long	vector signed long long		
vector float	vector float	vector float		
vector double	vector double	vector double		
vector unsigned char	vector unsigned char	10-bit signed int (リテラル)	*d* = si_orbi(*a*, *b*)	ORBI d, a, b
vector signed char	vector signed char			
vector unsigned char	vector unsigned char	unsigned char	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed char	vector signed char	signed char		
vector unsigned short	vector unsigned short	10-bit signed int (リテラル)	*d* = si_orhi(*a*, *b*)	ORHI d, a, b
vector signed short	vector signed short			
vector unsigned short	vector unsigned short	unsigned short	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed short	vector signed short	signed short		
vector unsigned int	vector unsigned int	10-bit signed int (リテラル)	*d* = si_ori(*a*, *b*)	ORI d, a, b
vector signed int	vector signed int			
vector unsigned int	vector unsigned int	unsigned int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed int	vector signed int	signed int		

spu_orc: Vector Bit-Wise OR with Complement

```
d = spu_orc(a, b)
```

ベクタ *a* の各ビットをベクタ *b* の対応するビットの補数と論理 OR 演算し、その結果をベクタ *d* の対応するビットへ代入します。

表 2-63: Vector Bit-Wise OR with Complement

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned char	vector unsigned char	vector unsigned char	*d* = si_orc(*a*, *b*)	ORC d,a, b
vector signed char	vector signed char	vector signed char		
vector unsigned short	vector unsigned short	vector unsigned short		
vector signed short	vector signed short	vector signed short		
vector unsigned int	vector unsigned int	vector unsigned int		
vector signed int	vector signed int	vector signed int		
vector unsigned long long	vector unsigned long long	vector unsigned long long		
vector signed long long	vector signed long long	vector signed long long		
vector float	vector float	vector float		
vector double	vector double	vector double		

spu_orx: OR Word Across

```
d = spu_orx(a)
```

ベクタ *a* の 4 つのワード要素に対して論理 OR 演算を行ない、結果をベクタ *d* のワード要素 0 へ代入します。ベクタ *d* のそれ以外のワード要素(1,2,3)へは 0 を代入します。

表 2-64: OR Word Across

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	a		
vector unsigned int	vector unsigned int	*d* = si_orx(*a*)	ORX d, a
vector signed int	vector signed int		

spu_xor: Vector Bit-Wise Exclusive OR

```
d = spu_xor(a, b)
```

ベクタ *a* の各要素をベクタ *b* の対応する要素と XOR 演算します。*b* がスカラである場合、そのスカラを各要素に複写した後に XOR 演算を行ないます。結果をベクタ *d* の対応するビットへ代入します。

表 2-65: Vector Bit-Wise Exclusive OR

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	b		
vector unsigned char	vector unsigned char	vector unsigned char	d = si_xor(a, b)	XOR d, a, b
vector signed char	vector signed char	vector signed char		
vector unsigned short	vector unsigned short	vector unsigned short		
vector signed short	vector signed short	vector signed short		
vector unsigned int	vector unsigned int	vector unsigned int		
vector signed int	vector signed int	vector signed int		
vector unsigned long long	vector unsigned long long	vector unsigned long long		
vector signed long long	vector signed long long	vector signed long long		
vector float	vector float	vector float		
vector double	vector double	vector double		
vector unsigned char	vector unsigned char	10-bit signed int (リテラル)	d = si_xorbi(a, b)	XORBI d, a, b
vector signed char	vector signed char			
vector unsigned char	vector unsigned char	unsigned char	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed char	vector signed char	signed char		
vector unsigned short	vector unsigned short	10-bit signed int (リテラル)	d = si_xorhi(a, b)	XORHI d, a, b
vector signed short	vector signed short			
vector unsigned short	vector unsigned short	unsigned short	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed short	vector signed short	signed short		
vector unsigned int	vector unsigned int	10-bit signed int (リテラル)	d = si_xori(a, b)	XORI d, a, b
vector signed int	vector signed int			
vector unsigned int	vector unsigned int	unsigned int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed int	vector signed int	signed int		

2.10. シフトおよびローテート命令に対応する組み込み関数

spu_rl: Element-Wise Rotate Left by Bits

```
d = spu_rl(a, count)
```

ベクタ *a* の各要素をベクタ *count* の対応する要素で示されるビット分だけ左にローテートします。要素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します。要素のサイズに応じて、ベクタ *count* のビットのうち一部のみを使用します。ハーフワード要素の場合、ベクタ *count* の下位 4 ビットを使用します。ワード要素の場合、ベクタ *count* の下位 5 ビットを使用します。

結果をベクタ *d* の対応する要素へ代入します。

表 2-66: Element-Wise Rotate Left by Bits

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned short	vector unsigned short	vector signed short	*d* = si_roth(*a*, *count*)	ROTH d, a, count
vector signed short	vector signed short			
vector unsigned int	vector unsigned int	vector signed int	*d* = si_rot(*a*, *count*)	ROT d, a, count
vector signed int	vector signed int			
vector unsigned short	vector unsigned short	7-bit signed int (リテラル)	*d* = si_rothi(*a*, *count*)	ROTHI d, a, count
vector signed short	vector signed short			
vector unsigned short	vector unsigned short	int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed short	vector signed short			
vector unsigned int	vector unsigned int	7-bit signed int (リテラル)	*d* = si_roti(*a*, *count*)	ROTI d, a, count
vector signed int	vector signed int			
vector unsigned int	vector unsigned int	int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed int	vector signed int			

spu_rlmask: Element-Wise Rotate Left and Mask by Bits

```
d = spu_rlmask(a, count)
```

この関数はelement-wise rotate left and mask演算を用いてベクタ*a*の各要素のビット単位の論理右シフト（LSR）を行ないます。*count*へは右シフトさせる量を負にした値を指定します。(*count*パラメータは 表 2-67に示すようにベクタとスカラのどちらもサポートしています。) 例えば、*count*がスカラ値 –5 である場合*a*の各要素を 5 ビット分右シフトします。この関数の作用は以下のコードでより正確に表現されています。

```
For (each halfword element h in vector a){
   int bitshift = -count & 0x1F;
   h = (bitshift & 0x10)? 0: LSR(h,bitshift);
}

For (each word element w in vector a){
   int bitshift = -count & 0x3F;
   w = (bitshift & 0x20)? 0: LSR(w,bitshift);
}
```

結果はベクタ *d* の対応する要素へ代入します。

表 2-67: Element-Wise Rotate Left and Mask by Bits

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned short	vector unsigned short	vector signed short	*d* = si_rothm(*a*, *count*)	ROTHM d, a, count
vector signed short	vector signed short			
vector unsigned int	vector unsigned int	vector signed int	*d* = si_rotm(*a*, *count*)	ROTM d, a, count
vector signed int	vector signed int			
vector unsigned short	vector unsigned short	7-bit signed int (リテラル)	*d* = si_rothmi(*a*, *count*)	ROTHMI d, a, count
vector signed short	vector signed short			
vector unsigned short	vector unsigned short	int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed short	vector signed short			
vector unsigned int	vector unsigned int	7-bit signed int (リテラル)	*d* = si_rotmi(*a*, *count*)	ROTMI d, a, count
vector signed int	vector signed int			
vector unsigned int	vector unsigned int	int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed int	vector signed int			

spu_rlmaska: Element-Wise Rotate Left and Mask Algebraic by Bits

```
d = spu_rlmaska(a, count)
```

この関数はelement-wise rotate left and mask演算を用いてベクタ*a*の各要素のビット単位の算術右シフト（ASR）を行ないます。*count*へは右シフトさせる量を負にした値を指定します。(*count*パラメータは 表 2-68に示すようにベクタとスカラのどちらもサポートしています。) 例えば、*count*がスカラ値 –5 である場合*a*の各要素を 5 ビット分右シフトします。この関数の作用は以下のコードでより正確に表現されています。

```
For (each halfword element h in vector a){
   int bitshift = -count & 0x1F;
   h = (bitshift & 0x10)? 0: ASR(h,bitshift);
}

For (each word element w in vector a){
   int bitshift = -count & 0x3F;
   w = (bitshift & 0x20)? 0: ASR(w,bitshift);
}
```

結果はベクタ *d* の対応する要素へ代入します。

表 2-68: Element-Wise Rotate Left and Mask Algebraic by Bits

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned short	vector unsigned short	vector signed short	*d* = si_rotmah(*a*, *count*)	ROTMAH d, a, count
vector signed short	vector signed short			
vector unsigned int	vector unsigned int	vector signed int	*d* = si_rotma(*a*, *count*)	ROTMA d, a, count
vector signed int	vector signed int			
vector unsigned short	vector unsigned short	7-bit signed int (リテラル)	*d* = si_rotmahi(*a*, *count*)	ROTMAHI d, a, count
vector signed short	vector signed short			
vector unsigned short	vector unsigned short	int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed short	vector signed short			
vector unsigned int	vector unsigned int	7-bit signed int (リテラル)	*d* = si_rotmai(*a*, *count*)	ROTMAI d, a, count
vector signed int	vector signed int			

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned int	vector unsigned int	int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed int	vector signed int			

spu_rlmaskqw: Rotate Left and Mask Quadword by Bits

```
d = spu_rlmaskqw(a, count)
```

この関数は rotate and mask quadword by bits 演算を用いて 7 ビット以下のクワッドワード論理右シフト（LSR）を行ないます。*count* へは右シフトさせる量を負にした値を指定します。例えば *count* が –5 である場合ベクタ *a* を 5 ビット分右シフトします。この関数の作用は以下のコードでより正確に表現されています。

```
qword spu_rlmaskqw(qword a, int count)
{   int bitshift = -count & 0x7;
    return LSR(a,bitshift);
}
```

結果として生じるクワッドワードをベクタ *d* へ代入します。

表 2-69: Rotate Left and Mask Quadword by Bits

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned char	vector unsigned char	int (リテラル)	*d* = si_rotqmbii(*a*, *count*) (*count* = 7 ビット即値)	ROTQMBII d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			
vector unsigned char	vector unsigned char	int (非リテラル)	*d* = si_rotqmbi(*a*, *count*)	ROTQMBI d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			

spu_rlmaskqwbyte: Rotate Left and Mask Quadword by Bytes

```
d = spu_rlmaskqwbyte(a, count)
```

この関数は rotate and mask quadword by bytes 演算を用いてバイト単位でクワッドワード論理右シフト（LSR）を行ないます。*count* へは右シフトする量を負にした値を指定します。例えば *count* が –5 である場合、ベクタ *a* を 5 バイト分右シフトします。この関数の作用は以下のコードでより正確に表現されています。

```
qword spu_rlmaskqwbyte(qword a, int count)
{   int bitshift = (-count << 3) & 0xF8;
    return LSR(a,bitshift);
}
```

結果として生じるクワッドワードをベクタ *d* へ代入します。

表 2-70: Rotate Left and Mask Quadword by Bytes

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned char	vector unsigned char	int (リテラル)	*d* = si_rotqmbyi(*a*, *count*) (*count* = 7 ビット即値)	ROTQMBYI d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			
vector unsigned char	vector unsigned char	int (非リテラル)	*d* = si_rotqmby(*a*, *count*)	ROTQMBY d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			

spu_rlmaskqwbytebc: Rotate Left and Mask Quadword by Bytes from Bit Shift Count

```
d = spu_rlmaskqwbytebc(a, count)
```

この関数は rotate and mask quadword by bytes from bit shift count 演算を用いてバイト単位でクワッドワード論理右シフト（LSR）を行ないます。*count* の 24 から 28 ビット目に右シフトする量を負にした値を指定します。例えば *count* が -10 である場合、ベクタ *a* を 2 バイト分右シフトします。この関数の作用は以下のコードでより正確に表現されています。

```
qword spu_rlmaskqwbytebc(qword a, int count)
{   int bitshift = -(count & 0xF8) & 0xF8;
    return LSR(a,bitshift);
}
```

結果として生じるクワッドワードをベクタ *d* へ代入します。

プログラミングの注意: 以下のコードはこの関数の標準的使用法を示しています。このコードを実行するとベクタ *a* の値を *n* ビット分論理右シフトした値であるベクタ *d* を算出します。

```
d = spu_rlmaskqwbytebc(a,7-n);
d = spu_rlmaskqw(d,-n);
```

表 2-71: Rotate Left and Mask Quadword by Bytes from Bit Shift Count

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned char	vector unsigned char	int	*d* = si_rotqmbybi(*a*, *count*)	ROTQMBYBI d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			

spu_rlqw: Rotate Left Quadword by Bits

```
d = spu_rlqw(a, count)
```

ベクタ *a* を *count* の下位 3 ビットで指定されるビット分だけ左にローテートします。ベクタ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します。結果をベクタ *d* へ代入します。

表 2-72: Rotate Left Quadword by Bits

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned char	vector unsigned char	int (リテラル)	*d* = si_rotqbii(*a*, *count*) (*count* = 7 ビット即値)	ROTQBII d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned char	vector unsigned char	int (非リテラル)	*d* = si_rotqbi(*a*, *count*)	ROTQBI d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			

spu_rlqwbyte: Rotate Left Quadword by Bytes

```
d = spu_rlqwbyte(a, count)
```

ベクタ *a* を *count* の下位 4 ビットで指定されるバイト分だけ左にローテートします。ベクタの左端からローテートアウトされたバイトは、右端にローテートインします。結果をベクタ *d* へ代入します。

表 2-73: Rotate Left Quadword by Bytes

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned char	vector unsigned char	int (リテラル)	*d* = si_rotqbyi(*a*, *count*) (count = 7 ビット即値)	ROTQBYI d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			
vector unsigned char	vector unsigned char	int (非リテラル)	*d* = si_rotqby(*a*, *count*)	ROTQBY d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			

spu_rlqwbytebc: Rotate Left Quadword by Bytes from Bit Shift Count

```
d = spu_rlqwbytebc(a, count)
```

ベクタ *a* を *count* のビット 24 から 28 で指定されるバイト分だけ左にローテートします。ベクタの左端からローテートアウトされたバイトは、右端にローテートインします。結果をベクタ *d* へ代入します。

表 2-74: Rotate Left Quadword by Bytes from Bit Shift Count

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned char	vector unsigned char	int	*d* = si_rotqbybi(*a*, *count*)	ROTQBYBI d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			

spu_sl: Element-Wise Shift Left by Bits

```
d = spu_sl(a, count)
```

ベクタ *a* の各要素をベクタ *count* の対応する要素で示されるビット分だけ左にシフトします。*count* がスカラである場合、そのスカラ値を各要素へ複写した後にシフトを行ないます。

要素の左端からシフトアウトされたビットを破棄し、右端より 0 をシフトインします。要素のサイズに応じて、ベクタ *count* のビットのうち一部のみを使用します。ハーフワード要素の場合、*count* の下位 5 ビットを使用し、ワード要素の場合は下位 6 ビットを使用します。結果をベクタ *d* の対応するビットへ代入します。

表 2-75: Element-Wise Shift Left Vector by Bits

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned short	vector unsigned short	vector unsigned short	*d* = si_shlh(*a*, *count*)	SHLH d, a, count
vector signed short	vector signed short			
vector unsigned int	vector unsigned int	vector unsigned int	*d* = si_shl(*a*, *count*)	SHL d, a, count
vector signed int	vector signed int			
vector unsigned short	vector unsigned short	7-bit unsigned int (リテラル)	*d* = si_shlhi(*a*, *count*)	SHLHI d, a, count
vector signed short	vector signed short			
vector unsigned short	vector unsigned short	unsigned int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed short	vector signed short			
vector unsigned int	vector unsigned int	7-bit unsigned int (リテラル)	*d* = si_shli(*a*, *count*)	SHLI d, a, count
vector signed int	vector signed int			
vector unsigned int	vector unsigned int	unsigned int	“2.2.1. スカラ型のオペランドをとる組み込み関数のマップ”を参照。	
vector signed int	vector signed int			

spu_slqw: Shift Left Quadword by Bits

```
d = spu_slqw(a, count)
```

ベクタ *a* を *count* の下位 3 ビットで指定されるビット分だけ左にシフトします。ベクタの左端からシフトアウトされたビットを破棄し、右端より 0 をシフトインします。結果をベクタ *d* へ代入します。

表 2-76: Shift Left Quadword by Bits

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned char	vector unsigned char	unsigned int (リテラル)	*d* = si_shlqbii(*a*, *count*) (*count* = 7 ビット即値)	SHLQBII d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			
vector unsigned char	vector unsigned char	unsigned int (非リテラル)	*d* = si_shlqbi(*a*, *count*)	SHLQBI d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			

spu_slqwbyte: Shift Left Quadword by Bytes

```
d = spu_slqwbyte(a, count)
```

ベクタ *a* を *count* の下位 5 ビットで指定されるバイト分だけ左にシフトします。ベクタの左端からシフトアウトされたバイトを破棄し、右端より 0 をシフトインします。結果をベクタ *d* へ代入します。

表 2-77: Shift Left Quadword by Bytes

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned char	vector unsigned char	unsigned int (リテラル)	*d* = si_shlqbyi(*a*, *count*) (*count* = 7 ビット即値)	SHLQBYI d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			
vector unsigned char	vector unsigned char	unsigned int (非リテラル)	*d* = si_shlqby(*a*, *count*)	SHLQBY d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			

spu_slqwbytebc: Shift Left Quadword by Bytes from Bit Shift Count

```
d = spu_slqwbytebc(a, count)
```

ベクタ *a* を *count* のビット 24 から 28 で指定されるバイト分だけ左にシフトします。ベクタの左端からシフトアウトされたバイトを破棄し、右端より 0 をシフトインします。結果をベクタ *d* へ代入します。

表 2-78: Shift Left Quadword by Bytes from Bit Shift Count

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ
d	a	count		
vector unsigned char	vector unsigned char	unsigned int	*d* = si_shlqbybi(*a*, *count*)	SHLQBYBI d, a, count
vector signed char	vector signed char			
vector unsigned short	vector unsigned short			
vector signed short	vector signed short			
vector unsigned int	vector unsigned int			
vector signed int	vector signed int			
vector unsigned long long	vector unsigned long long			
vector signed long long	vector signed long long			
vector float	vector float			
vector double	vector double			

2.11. 制御命令に対応する組み込み関数

spu_idisable: Disable Interrupts

```
(void) spu_idisable()
```

非同期割り込みを禁止します。

プログラミングの注意: この組み込み関数は他の全ての命令に対して揮発性を持つとみなされるため、BID 命令と他の命令の順序が入れ替わることはありません。

表 2-79: Disable Interrupts

個別組み込み関数	アセンブリ命令へのマップ
N/A	**位置依存:** ILA t, *next_inst* BID t *next_inst:* **位置独立:** BRSL t, *next_inst* *next_inst:* AI t, t, 8 BID t

spu_ienable: Enable Interrupts

```
(void) spu_ienable()
```

非同期割り込みを許可します。

プログラミングの注意: この組み込み関数は他の全ての命令に対して揮発性を持つとみなされるため、BIE 命令と他の命令の順序が入れ替わることはありません。

表 2-80: Enable Interrupts

個別組み込み関数	アセンブリ命令へのマップ
N/A	**位置依存:** ILA t, *next_inst* BIE t *next_inst:* **位置独立:** BRSL t, *next_inst* *next_inst:* AI t, t, 8 BIE t

spu_mffpscr: Move from Floating-Point Status and Control Register

```
d = spu_mffpscr()
```

浮動小数点状態および制御レジスタ「FPSCR」 の値を読み出し、その値を *d* へ代入します。FPSCR の未使用のビットは強制的に 0 にします。

プログラミングの注意:この組み込み関数は浮動小数点命令に対して揮発性を持つとみなされるため、これらの命令と順序が入れ替わることはありません。浮動小数点命令には以下のものがあります。
`cflts、 cfltu、 csflt、 cuflt、 dfa、 dfm、 dfma、 dfms、 dfnma、 dfnms、 dfs、 fa、 fceq、 fcgt、 fcmeq、 fcmgt、 fesd、 fi、 fm、 fma、 fms、 fnms、 frds、 frest、 frsqest、 fscrwr`

表 2-81: Move from Floating-Point Status and Control Register

戻り値／引数の型	個別組み込み関数	アセンブリ命令へのマップ
d		
vector unsigned int	*d* = si_fscrrd()	FSCRRD d

spu_mfspr: Move from Special Purpose Register

```
d = spu_mfspr(register)
```

列挙定数 *register* で指定した Special Purpose レジスタ（SPR） を読み出し、値を *d* へ代入します。

表 2-82: Move from Special Purpose Register

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	register		
unsigned int	enumeration	*d* = *si_to_uint*(si_mfspr(register))	MFSPR d, register

spu_mtfpscr: Move to Floating-Point Status and Control Register

```
(void) spu_mtfpscr(a)
```

引数 *a* の値を浮動小数点状態および制御レジスタ「FPSCR」へ書き込みます。

プログラミングの注意:この組み込み関数は浮動小数点命令に対して揮発性を持つとみなされるため、これらの命令と順序が入れ替わることはありません。

表 2-83: Move to Floating-Point Status and Control Register

戻り値／引数の型	個別組み込み関数	アセンブリ命令へのマップ
a		
vector unsigned int	si_fscrwr(*a*)	FSCRWR rt[1], a

[1] 偽のターゲットであるパラメータ *rt* は近傍命令のレジスタ使用状態に応じて最適な値が選択されます。

spu_mtspr: Move to Special Purpose Register

```
(void) spu_mtspr(register, a)
```

引数 *a* の値を列挙定数 *register* で指定した Special Purpose レジスタ（SPR）へ書き込みます。

表 2-84: Move to Special Purpose Register

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
register	a		
enumeration	unsigned int	si_mtspr(register, *si_from_uint*(*a*))	MTSPR register, a

spu_dsync: Synchronize Data

```
(void) spu_dsync()
```

後続命令の実行前に先行するストア命令をすべて強制的に完了させます。この関数は LS へのすべてのストアが MFC や PPU から観測可能であることを保証するためのものです。

プログラミングの注意:この組み込み関数はストア命令および MFC 書き込み命令に対して揮発性を持つとみなされるため、これらの命令と順序が入れ替わることはありません。ストア命令および MFC 書き込み命令には以下のものがあります。

`stqa、stqd、 stqr、 stqx、 wrch`

表 2-85: Synchronize Data

個別組み込み関数	アセンブリ命令へのマップ
si_dsync()	DSYNC

spu_stop: Stop and Signal

```
(void) spu_stop(type)
```

SPU プログラムの実行を停止します。`stop` 命令のアドレスを `SPU NPC` レジスタの下位ビットへ格納します。シグナルである *`type`* の値を SPU ステータスレジスタへ書き込み、PPU へ割り込みを発生させます。

プログラミングの注意:この組み込み関数は他の全ての命令に対して揮発性を持つとみなされるため、他の命令と順序が入れ替わることはありません。

表 2-86: Stop and Signal

個別組み込み関数	type	アセンブリ命令へのマップ
si_stop(type)	unsigned int (14 ビット リテラル)	STOP type

spu_sync: Synchronize

```
(void) spu_sync()
(void) spu_sync_c()
```

プロセッサが後続の命令をフェッチする前に、処理途中のストア命令がすべて完了するまで待つようにします。`spu_sync_c` 形式の場合、命令同期の前にチャネル同期も行ないます。命令ストリームを更新するストア命令の後にはこのオペレーションを使用しなければなりません。

プログラミングの注意:これらの同期組み込み関数は他の全ての命令に対して揮発性を持つとみなされるため、他の命令と順序が入れ替わることはありません。

表 2-87: Synchronize

総称組み込み関数 Form	個別組み込み関数	アセンブリ命令へのマップ
spu_sync	si_sync()	SYNC
spu_sync_c	si_syncc()	SYNCC

2.12. チャネル制御命令に対応する組み込み関数

チャネル制御命令に対応する組み込み関数はチャネル番号（*channel*）を入力引数としてとります。チャネル番号は 0 から 127 の範囲内の符号なし整数リテラル値となります。SPUとMFCのチャネル番号およびそれらのニーモニックは 表 2-88および 表 2-89にそれぞれ記載されています。チャネルについての詳細は*Cell Broadband Engine™ アーキテクチャ* を参照してください。

プログラミングの注意:チャネル制御命令に対応する組み込み関数は、決して他のチャネルコマンドや揮発性を持つ LS へのアクセスと順序を入れ替えてはなりません。

表 2-88: SPU チャネル番号 [1]

チャネル番号	ニーモニック	説明
0	SPU_RdEventStat	イベントステータスの読み込み。（マスク適用）
1	SPU_WrEventMask	イベントマスクの書き込み。
2	SPU_WrEventAck	イベント受け付けフラグの書き込み。
3	SPU_RdSigNotify1	シグナル通知 1。
4	SPU_RdSigNotify2	シグナル通知 2。
7	SPU_WrDec	デクリメンタのカウントの書き込み。
8	SPU_RdDec	デクリメンタのカウントの読み込み。
11	SPU_RdEventMask	イベントステータスマスクの読み込み。
13	SPU_RdMachStat	SPU 実行ステータスの読み込み。
14	SPU_WrSRR0	SPU マシン状態の保存／復元レジスタ 0（SRR0）の書き込み。
15	SPU_RdSRR0	SPU マシン状態の保存／復元レジスタ 0（SRR0）の読み込み。
28	SPU_WrOutMbox	Outbound Mailbox の内容の書き込み。
29	SPU_RdInMbox	Inbound Mailbox の内容の読み込み。
30	SPU_WrOutIntrMbox	Outbound Interrupt Mailbox の内容の書き込み（PPU に対する割り込み）。

[1] チャネル列挙子は `spu_intrinsics.h`.で定義されています。

表 2-89: MFC チャネル番号 [1]

チャネル番号	ニーモニック	説明
9	MFC_WrMSSyncReq	マルチソース同期リクエストの書き込み。
12	MFC_RdTagMask	タグマスクの読み込み。
16	MFC_LSA	ローカルメモリアドレス・コマンドパラメータの書き込み。
17	MFC_EAH	上位 DMA 実効アドレス・コマンドパラメータの書き込み。
18	MFC_EAL	下位 DMA 実効アドレス・コマンドパラメータの書き込み。
19	MFC_Size	DMA 転送サイズ・コマンドパラメータの書き込み。
20	MFC_TagID	タグ識別子コマンドパラメータの書き込み。
21	MFC_Cmd	DMA コマンドを関連するクラス ID と共に書き込み、キューに入れる。
22	MFC_WrTagMask	タグマスクの書き込み。
23	MFC_WrTagUpdate	条件付きまたは無条件のタグステータス更新のリクエストの書き込み。
24	MFC_RdTagStat	マスク適用後のタグステータスの読み込み。
25	MFC_RdListStallStat	DMA リストのストール/通知ステータスの読み込み。
26	MFC_WrListStallAck	DMA リストのストール/通知受領応答の書き込み。
27	MFC_RdAtomicStat	最後に完了した即時実行 MFC アトミック更新コマンドの終了ステータスの読み込み。

[1] MFC チャネルは CBEA 準拠のシステムにおける SPU についてのみ使用できます。MFC チャネル列挙子は `spu_intrinsics.h`.で定義されています。

spu_readch: Read Word Channel

```
d = spu_readch(channel)
```

channel で指定したワードチャネルを読み込み、値を *d* へ代入します。指定したチャネルが実装されていない場合は 0 を返します。

表 2-90: Read Word Channel

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	channel		
unsigned int	enumeration	*d* = si_to_uint(si_rdch(*channel*))	RDCH d, channel

spu_readchqw: Read Quadword Channel

```
d = spu_readchqw(channel)
```

channel で指定したクワッドワードチャネルを読み込み、値を *d* へ代入します。指定したチャネルが実装されていない場合は 0 を返します。

表 2-91: Read Quadword Channel

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	channel		
vector unsigned int	enumeration	*d* = si_rdch(*channel*)	RDCH d, channel

spu_readchcnt: Read Channel Count

```
d = spu_readchcnt(channel)
```

channel で指定したチャネルに対して Read Count 演算を行ない、値を *d* へ代入します。指定したチャネルが実装されていない場合は *d* へ 0 を代入します。

表 2-92: Read Channel Count

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
d	channel		
unsigned int	enumeration	*d* = si_rchcnt(*channel*)	RCHCNT d, channel

spu_writech: Write Word Channel

```
(void) spu_writech(channel, a)
```

スカラ *a* の値を列挙定数 *channel* で指定した チャネルへ書き込みます。

表 2-93: Write Word Channel

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
channel	a		
enumeration	int	si_wrch(*channel*, *si_from_int*(*a*))	WRCH channel, a
	unsigned int	si_wrch(*channel*, *si_from_uint*(*a*))	

spu_writechqw: Write Quadword Channel

```
(void) spu_writechqw(channel, a)
```

ベクタ *a* の値を列挙定数 *channel* で指定した チャネルへ書き込みます。

表 2-94: Write Quadword Channel

戻り値／引数の型		個別組み込み関数	アセンブリ命令へのマップ
channel	a		
enumeration	vector unsigned char	si_wrch(*channel*, *a*)	WRCH channel, a
	vector signed char		
	vector unsigned short		
	vector signed short		
	vector unsigned int		
	vector signed int		
	vector unsigned long long		
	vector signed long long		
	vector float		
	vector double		

2.13. スカラ命令に対応する組み込み関数

これまでに説明した組み込み関数はすべてベクタ型のオペランドについての演算を行なうものでした。このセクションではプログラマがスカラとベクタ間の変換を効率的に行なうための特別なユティリティ組み込み関数について説明します。これらのユティリティ組み込み関数を用いることにより、プログラマはスカラ・ベクタ間の変換のためにわざわざアセンブリ言語を用いることなく組み込み関数を使うことができます。このユティリティはシャッフルバイトのように C 言語で表現することが難しい演算を行なう場合に特に有用です。

spu_extract: Extract Vector Element from Vector

```
d = spu_extract(a, element)
```

element で指定する要素をベクタ *a* より抽出し、*d* へ代入します。要素のサイズに応じて、*element* の一部の下位ビットのみを要素のインデックスとして使用します。要素のサイズ 1 バイト、2 バイト、4 バイト、8 バイトに対して、*element* のそれぞれ下位 4 ビット、3 ビット、2 ビット、1 ビット分のみがインデックスとして使用されます。

表 2-95: Extract Vector Element from Vector

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ [1]
d	a	element		
unsigned char	vector unsigned char	int (非リテラル)	N/A	ROTQBY d, a, element ROTMI d, d, -24
signed char	vector signed char		N/A	ROTQBY d, a, element ROTMAI d, d, -24
unsigned short	vector unsigned short		N/A	SHLI t, element, 1 ROTQBY d, a, t ROTMI d, d, -16
signed short	vector signed short		N/A	SHLI t, element, 1 ROTQBY d, a, t ROTMAI d, d, -16
unsigned int	vector unsigned int		N/A	SHLI t, element, 2 ROTQBY d, a, t
signed int	vector signed int		N/A	SHLI t, element, 2 ROTQBY d, a, t
unsigned long long	vector unsigned long long		N/A	SHLI t, element, 3 ROTQBY d, a, t
signed long long	vector signed long long		N/A	SHLI t, element, 3 ROTQBY d, a, t
float	vector float		N/A	SHLI t, element, 2 ROTQBY d, a, t
double	vector double		N/A	SHLI t, element, 3 ROTQBY d, a, t
unsigned char	vector unsigned char	int (リテラル)	N/A	ROTQBYI d, a, element-3
signed char	vector signed char		N/A	
unsigned short	vector unsigned short		N/A	ROTQBYI d, a, 2*(element-1)
signed short	vector signed short		N/A	
unsigned int	vector unsigned int		N/A	ROTQBYI d, a, 4*element
signed int	vector signed int		N/A	
unsigned long long	vector unsigned long long		N/A	ROTQBYI d, a, 8*element
signed long long	vector signed long long		N/A	
float	vector float		N/A	ROTQBYI d, a, 4*element
double	vector double		N/A	ROTQBYI d, a, 8*element

[1] element として指定した値がプリファード（スカラ）要素を指す既知の値（リテラル）である場合、命令は生成されません。1 バイト要素に対するスカラ要素のインデックスは 3 です。2 バイト要素に対するスカラ要素のインデックスは 1 です。4 バイト要素と 8 バイト要素に対するスカラ要素のインデックスは 0 です。次に実行する演算のために結果のスカラ値をさらに大きなデータ型にキャストする必要がある場合、更に符号拡張を行なうことができます。また、この符号拡張は次の演算まで延期することができます。

spu_insert: Insert Scalar into Specified Vector Element

```
d = spu_insert(a, b, element)
```

スカラ *a* を *element* で指定するベクタ *b* の要素へ挿入し、変換後のベクタを *d* へ代入します。ベクタ *b* のその他の要素は変更されません。要素のサイズに応じて、*element* の一部の下位ビットのみを要素のインデックスとして使用します。要素のサイズ 1 バイト、2 バイト、4 バイト、8 バイトに対して、*element* のそれぞれ下位 4 ビット、3 ビット、2 ビット、1 ビット分のみがインデックスとして使用されます。

表 2-96: Insert Scalar into Specified Vector Element

戻り値／引数の型				個別組み込み関数	アセンブリ命令へのマップ [1]
d	a	b	element		
vector unsigned char	unsigned char	vector unsigned char	int (非リテラル)	N/A	CBD t, 0(element) SHUFB d, a, b, t
vector signed char	signed char	vector signed char		N/A	
vector unsigned short	unsigned short	vector unsigned short		N/A	SHLI t, element, 1 CHD t, 0(t) SHUFB d, a, b, t
vector signed short	signed short	vector signed short		N/A	
vector unsigned int	unsigned int	vector unsigned int		N/A	SHLI t, element, 2 CWD t, 0(t) SHUFB d, a, b, t
vector signed int	signed int	vector signed int		N/A	
vector float	float	vector float		N/A	
vector unsigned long long	unsigned long long	vector unsigned long long		N/A	SHLI t, element, 3 CDD t, 0(t) SHUFB d, a, b, t
vector signed long long	signed long long	vector signed long long		N/A	
vector double	double	vector double		N/A	
vector unsigned char	unsigned char	vector unsigned char	int (リテラル)	N/A	LQD pat, CONST_AREA SHUFB d, a, b, pat
vector signed char	signed char	vector signed char		N/A	
vector unsigned short	unsigned short	vector unsigned short		N/A	LQD pat, CONST_AREA SHUFB d, a, b, pat
vector signed short	signed short	vector signed short		N/A	
vector unsigned int	unsigned int	vector unsigned int		N/A	LQD pat, CONST_AREA SHUFB d, a, b, pat
vector signed int	signed int	vector signed int		N/A	
vector float	float	vector float		N/A	
vector unsigned long long	unsigned long long	vector unsigned long long		N/A	LQD pat, CONST_AREA SHUFB d, a, b, pat
vector signed long long	signed long long	vector signed long long		N/A	
vector double	double	vector double		N/A	

[1] element として指定した値が既知の値（リテラル）である場合、定数領域よりシャッフルパターンをロードすることができます。シャッフルパターンの内容は要素のサイズと置換される要素の位置によります。

spu_promote: Promote Scalar to a Vector

```
d = spu_promote(a, element)
```

スカラ *a* を、*element* で指定する要素に値 *a* を含むベクタとして格上げし、変換したベクタを *d* へ代入します。ベクタのその他の要素はすべて不定です。要素とスカラのサイズに応じて、*element* の一部の下位ビットのみをインデックスとして使用します。要素のサイズ 1 バイト、2 バイト、4 バイト、8 バイトに対して、*element* のそれぞれ下位 4 ビット、3 ビット、2 ビット、1 ビット分のみがインデックスとして使用されます。

表 2-97: Promote Scalar to a Vector

戻り値／引数の型			個別組み込み関数	アセンブリ命令へのマップ [1]
d	a	element		
vector unsigned char	unsigned char	int (非リテラル)	N/A	SFI t, element, 3 ROTQBY d, a, t
vector signed char	signed char		N/A	
vector unsigned short	unsigned short		N/A	SFI t, element, 1 SHLI t, t, 1 ROTQBY d, a, t
vector signed short	signed short		N/A	
vector unsigned int	unsigned int		N/A	SFI t, element, 0 SHLI t, t, 2 ROTQBY d, a, t
vector signed int	signed int		N/A	
vector float	float		N/A	
vector unsigned long long	unsigned long long		N/A	SHLI t, element, 3 ROTQBY d, a, t
vector signed long long	signed long long		N/A	
vector double	double		N/A	
vector unsigned char	unsigned char	int (リテラル)	N/A	ROTQBYI d, a, (3-element)
vector signed char	signed char		N/A	
vector unsigned short	unsigned short		N/A	ROTQBYI d, a, 2*(1-element)
vector signed short	signed short		N/A	
vector unsigned int	unsigned int		N/A	ROTQBYI d, a, -4*element
vector signed int	signed int		N/A	
vector float	float		N/A	
vector unsigned long long	unsigned long long		N/A	ROTQBYI d, a, -8*element
vector signed long long	signed long long		N/A	
vector double	double		N/A	

[1] element として指定した値がプリファード（スカラ）要素を指す既知の値（リテラル）である場合、命令は生成されません。1 バイト要素に対するスカラ要素のインデックスは 3 です。2 バイト要素に対するスカラ要素のインデックスは 1 です。4 バイト要素と 8 バイト要素に対するスカラ要素のインデックスは 0 です。

3. 複合組み込み関数

本章ではさまざまな SPU プログラムで利用できる実用的ないくつかの複合組み込み関数について説明します。複合組み込み関数とは一連の低レベル組み込み関数で構成されるものをいいます。ここでの「低レベル組み込み関数」は総称組み込み関数および個別組み込み関数を指しています。オペレーションの複雑さ、使用頻度、スケジューリング制約により特定の処理を複合組み込み関数として提供しています。

複合組み込み関数は DMA 組み込み関数です。DMA 組み込み関数はチャネル制御組み込み関数へ大きく依存しています。

spu_mfcdma32: Initiate DMA to/from 32-bit Effective Address

```
spu_mfcdma32(ls, ea, size, tagid, cmd)
```

size バイトの DMA 転送を（LS からシステムメモリへまたはシステムメモリから LS へ）開始します。実効アドレス *ea* は 32 ビットの仮想メモリアドレスです。LS アドレスはパラメータ *ls* で指定します。DMA 要求は指定した *tagid* を用いて発行されます。DMA のタイプと方向、バンド幅予約、クラス ID は *cmd* パラメータとしてコード化されています。コマンドやサポートされている DMA 操作のサイズにおける制約については、*Cell Broadband Engine™ アーキテクチャ* を参照してください。

表 3-98: Initiate DMA to/from 32-bit Effective Address

戻り値／引数の型					アセンブリ命令へのマップ
ls	ea	size	tagid	cmd	
volatile void *	unsigned int	unsigned int	unsigned int	unsigned int	spu_writech(MFC_LSA, *ls*) spu_writech(MFC_EAL, *ea*) spu_writech(MFC_Size, *size*) spu_writech(MFC_TagID, *tagid*) spu_writech(MFC_Cmd, *cmd*)

spu_mfcdma64: Initiate DMA to/from 64-bit Effective Address

```
spu_mfcdma64(ls, eahi, ealow, size, tagid, cmd)
```

size バイトの DMA 転送を（LS からシステムメモリへまたはシステムメモリから LS へ）開始します。*eahi* と *ealow* の結合として指定される実効アドレスは 64 ビットの仮想メモリアドレスです。 LS アドレスはパラメータ *ls* で指定します。DMA 要求は指定した *tagid* を用いて発行されます。DMA のタイプと方向、バンド幅予約、クラス ID は *cmd* パラメータとしてコード化されています。 コマンドやサポートされている DMA 操作のサイズにおける制約については、*Cell Broadband Engine™ アーキテクチャ* を参照してください。

表 3-99: Initiate DMA to/from 64-bit Effective Address

戻り値／引数の型						アセンブリ命令へのマップ
ls	eahi	ealow	size	tagid	cmd	
volatile void *	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	spu_writech(MFC_LSA, *ls*) spu_writech(MFC_EAH, *eahi*) spu_writech(MFC_EAL, *ealow*) spu_writech(MFC_Size, *size*) spu_writech(MFC_TagID, *tagid*) spu_writech(MFC_CMD, *cmd*)

spu_mfcstat: Read MFC Tag Status

```
d = spu_mfcstat(type)
```

現在の MFC タグステータスを読み出し、その値と現在のタグマスクの値との論理積をとり、結果を *d* に代入します。読み出しのタイプは *type* パラメータにより指定します。*type* が 0 である場合、MFC タグステータスを読み

出して直ちに返します。*type* が 1 である場合は、読み出しは、処理が完了していない MFC タグのうちいずれかの処理が完了するまでブロックし、2 である場合は、読み出しは、すべての MFC タグの処理が完了するまでブロックします。

表 3-100: Read MFC Tag Status

戻り値／引数の型		アセンブリ命令へのマップ
d	type	
unsigned int	unsigned int	spu_writech(MFC_WrTagUpdate, *type*) *d* = spu_readch(MFC_RdTagStat)

4. MFC入出力のプログラミングサポート

本章ではいくつかの MFC ユティリティ関数について説明します。これらの関数はプログラミングの利便性を高めるために提供することが可能ですが、いずれも要件として求められているものではありません。マクロ定義、あるいはコンパイラ中のビルトイン関数のいずれの形でも実装することができます。これらの関数を利用するために、プログラマは `spu_mfcio.h` ヘッダファイルをインクルードする必要があります。

以下のセクションでは各々の関数について、先ず用法を示し、その後に概要説明と実装コードを記述しています。

4.1. 構造体

最も重要な構造体としては、MFC List DMA があります。このリスト中の要素を以下に示します。

mfc_list_element: DMA List Element for MFC List DMA

```
typedef struct mfc_list_element {
  uint64_t notify     : 1;
  uint64_t reserved   : 16;
  uint64_t size       : 15;
  uint64_t eal        : 32;
} mfc_list_element_t;
```

`mfc_list_element` は MFC List DMA 配列の要素です。この構造体は次に挙げるビットフィールドで構成されます。*`notify`* はストール通知ビット、*`reserved`* は予約済みで 0 に設定されます。*`size`* はこのリスト要素の転送サイズ、*`eal`* は 64 ビット実効アドレスの下位ワードです。

4.2. 実効アドレスユティリティ

MFC のプログラミングではしばしば実効アドレスの操作が必要となります。本セクションでは最も頻繁に行なう実効アドレス操作のための関数をいくつか説明します。

mfc_ea2h: Extract Higher 32 Bits from Effective Address

```
(uint32_t) mfc_ea2h(uint64_t ea)
```

64 ビット実効アドレス *`ea`* の上位 32 ビットを抽出します。

実装

```
(uint32_t)((uint64_t)(ea)>>32)
```

mfc_ea2l: Extract Lower 32 Bits from Effective Address

```
(uint32_t) mfc_ea2l(uint64_t ea)
```

64 ビット実効アドレス *`ea`* の下位 32 ビットを抽出します。

実装

```
(uint32_t)(ea)
```

mfc_hl2ea: Concatenate Higher 32 Bits and Lower 32 Bits

```
(uint64_t) mfc_hl2ea(uint32_t high, uint32_t low)
```

64 ビット実効アドレスの上位 32 ビットである *`high`* と下位 32 ビットである *`low`* を連結します。

実装

```
si_to_ullong(si_selb(si_from_uint(high),
    si_rotqbyi(si_from_uint(low), -4), si_fsmbi(0x0f0f)))
```

mfc_ceil128: Round Up Value to Next Multiple of 128

```
(uint32_t) mfc_ceil128(uint32_t value)
(uint64_t) mfc_ceil128(uint64_t value)
(uintptr_t) mfc_ceil128(uintptr_t value)
```

引数 *value* を次の 128 の倍数に切り上げます。

実装

```
(value + 127) & ~127
```

使用例

```
volatile char buf[256];
volatile void *ptr = (volatile void*)mfc_ceil128((uintptr_t)buf);
```

4.3. MFC DMAコマンド

本セクションでは様々な MFC DMA コマンドを実装するための関数について説明します。DMA コマンドについては、サポートされている操作のサイズ制約に関する点も含め、*Cell Broadband Engine™ アーキテクチャ* を参照してください。

表 4-101にMFC DMAコマンドのニーモニックを示します。

表 4-101: MFC DMA コマンドニーモニック [1]

ニーモニック	Opcode	コマンド
MFC_PUT_CMD	0x0020	put
MFC_PUTB_CMD	0x0021	putb
MFC_PUTF_CMD	0x0022	putf
MFC_GET_CMD	0x0040	get
MFC_GETB_CMD	0x0041	getb
MFC_GETF_CMD	0x0042	getf

[1] MFC コマンド列挙子は `spu_mfcio.h` で定義されています。

mfc_put: Move Data from Local Storage to Effective Address

```
(void) mfc_put(volatile void *ls, uint64_t ea, uint32_t size, uint32_t tag,
    uint32_t tid, uint32_t rid)
```

データを LS からシステムメモリへ移動します。この関数の引数は `spu_mfcdma64` コマンドの引数に相当し、*ls* は LS アドレス、*ea* はシステムメモリ内の実効アドレス、`size` は DMA 転送サイズ、*tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea), mfc_ea2l(ea), size, tag,
    ((tid<<24)|(rid<<16)|MFC_PUT_CMD))
```

mfc_putb: Move Data from Local Storage to Effective Address with Barrier

```
(void) mfc_putb(volatile void *ls, uint64_t ea, uint32_t size, uint32_t tag,
    uint32_t tid, uint32_t rid)
```

データを LS からシステムメモリへ移動します。この関数の引数は `spu_mfcdma64` コマンドの引数に相当し、*ls* は LS アドレス、*ea* はシステムメモリ内の実効アドレス、`size` は DMA 転送サイズ、*tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。このコマンドおよび、このコマンドと同一タグ ID を持つ後続のコマンドは、既に発行された同じタググループかつ同じコマンドキューにあるすべてのコマンドに対して、ローカルに順序付けされます。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea), mfc_ea2l(ea), size, tag,
    ((tid<<24)|(rid<<16)|MFC_PUTB_CMD))
```

mfc_putf: Move Data from Local Storage to Effective Address with Fence

```
(void) mfc_putf(volatile void *ls, uint64_t ea, uint32_t size, uint32_t tag,
    uint32_t tid, uint32_t rid)
```

データを LS からシステムメモリへ移動します。この関数の引数は `spu_mfcdma64` コマンドの引数に相当し、*ls* は LS アドレス、*ea* はシステムメモリ内の実効アドレス、*size* は DMA 転送サイズ、*tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。このコマンドは、それより前に発行された同じタググループかつ同じコマンドキューにあるすべてのコマンドに対して、ローカルに順序付けされます。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea), mfc_ea2l(ea), size, tag,
    ((tid<<24)|(rid<<16)|MFC_PUTF_CMD))
```

mfc_get: Move Data from Effective Address to Local Storage

```
(void) mfc_get(volatile void *ls, uint64_t ea, uint32_t size, uint32_t tag,
    uint32_t tid, uint32_t rid)
```

データをシステムメモリから LS へ移動します。この関数の引数は `spu_mfcdma64` コマンドの引数に相当し、*ls* は LS アドレス、*ea* はシステムメモリ内の実効アドレス、*size* は DMA 転送サイズ、*tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea), mfc_ea2l(ea), size, tag,
    ((tid<<24)|(rid<<16)|MFC_GET_CMD))
```

mfc_getf: Move Data from Effective Address to Local Storage with Fence

```
(void) mfc_getf(volatile void *ls, uint64_t ea, uint32_t size, uint32_t tag,
    uint32_t tid, uint32_t rid)
```

データをシステムメモリから LS へ移動します。この関数の引数は `spu_mfcdma64` コマンドの引数に相当し、*ls* は LS アドレス、*ea* はシステムメモリ内の実効アドレス、*size* は DMA 転送サイズ、*tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。このコマンドは、それより前に発行された同じタググループかつ同じコマンドキューにあるすべてのコマンドに対して、ローカルに順序付けされます。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea), mfc_ea2l(ea), size,
    tag,((tid<<24)|(rid<<16)|MFC_GETF_CMD))
```

mfc_getb: Move Data from Effective Address to Local Storage with Barrier

```
(void) mfc_getb (volatile void *ls, uint64_t ea, uint32_t size, uint32_t tag,
    uint32_t tid, uint32_t rid)
```

データをシステムメモリから LS へ移動します。この関数の引数は `spu_mfcdma64` コマンドの引数に相当し、*ls* は LS アドレス、*ea* はシステムメモリ内の実効アドレス、*size* は DMA 転送サイズ、*tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。このコマンドおよび、このコマンドと同一タグ ID を持つ後続のコマンドは、既に発行された同じタググループかつ同じコマンドキューにあるすべてのコマンドに対して、ローカルに順序付けされます。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea), mfc_ea2l(ea), size,tag,
    ((tid<<24)|(rid<<16)|MFC_GETB_CMD))
```

4.4. MFCリストDMAコマンド

本セクションでは MFC リスト DMA コマンドを操作するためのユティリティ関数について説明します。DMA コマンドについては、サポートされている操作のサイズ制約に関する点も含め、*Cell Broadband Engine™ アーキテクチャ* を参照してください。

表 4-102にMFCリストDMAコマンドのニーモニックを示します。

表 4-102: MFC リスト DMA コマンドニーモニック [1]

ニーモニック	Opcode	コマンド
MFC_PUTL_CMD	0x0024	putl
MFC_PUTLB_CMD	0x0025	putlb
MFC_PUTLF_CMD	0x0026	putlf
MFC_GETL_CMD	0x0044	getl
MFC_GETLB_CMD	0x0045	getlb
MFC_GETLF_CMD	0x0046	getlf

[1] MFC コマンド列挙子は `spu_mfcio.h` で定義されています。

mfc_putl: Move Data from Local Storage to Effective Address Using MFC List

```
(void) mfc_putl(volatile void *ls, uint64_t ea, mfc_list_element_t *list,
    uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

MFC リストを使用して、データを LS からシステムメモリへ移動します。この関数の引数は `spu_mfcdma64` コマンドの引数に相当し、`ls` は LS アドレス、`ea` はシステムメモリ内の実効アドレス、`list` は DMA リストのアドレス、`list_size` は DMA リストのサイズ、`tag` は DMA タグ、`tid` は転送クラス ID、`rid` は置換クラス ID です。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea), (unsigned int)(list), list_size, tag,
    ((tid<<24)|(rid<<16)|MFC_PUTL_CMD))
```

mfc_putlb: Move Data from Local Storage to Effective Address Using MFC List with Barrier

```
(void) mfc_putlb(volatile void *ls, uint64_t ea, mfc_list_element_t *list,
    uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

MFC リストを使用して、データを LS からシステムメモリへ移動します。この関数の引数は `spu_mfcdma64` コマンドの引数に相当し、`ls` は LS アドレス、`ea` はシステムメモリ内の実効アドレス、`list` は DMA リストのアドレス、`list_size` は DMA リストのサイズ、`tag` は DMA タグ、`tid` は転送クラス ID、`rid` は置換クラス ID です。このコマンドおよび、このコマンドと同一タグ ID を持つ後続のコマンドは、既に発行された同じタググループかつ同じコマンドキューにあるすべてのコマンドに対して、ローカルに順序付けされます。

実装

```
spu_mfcdma64(ls,mfc_ea2h(ea),(unsigned int)(list), list_size, tag,
    ((tid<<24)|(rid<<16)|MFC_PUTLB_CMD))
```

mfc_putlf: Move Data from Local Storage to Effective Address Using MFC List with Fence

```
(void) mfc_putlf(volatile void *ls, uint64_t ea, mfc_list_element_t *list,
    uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

MFC リストを使用して、データを LS からシステムメモリへ移動します。この関数の引数は `spu_mfcdma64` コマンドの引数に相当し、`ls` は LS アドレス、`ea` はシステムメモリ内の実効アドレス、`list` は DMA リストのアドレス、`list_size` は DMA リストのサイズ、`tag` は DMA タグ、`tid` は転送クラス ID、`rid` は置換クラス ID です。このコマンドは、それより前に発行された同じタググループかつ同じコマンドキューにあるすべてのコマンドに対して、ローカルに順序付けされます。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea),(unsigned int)(list), list_size, tag,
    ((tid<<24)|(rid<<16)|MFC_PUTLF_CMD))
```

mfc_getl: Move Data from Effective Address to Local Storage Using MFC List

```
(void) mfc_getl (volatile void *ls, uint64_t ea, mfc_list_element_t *list,
    uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

MFC リストを使用して、データをシステムメモリから LS へ移動します。この関数の引数は `spu_mfcdma64` コマンドの引数に相当し、*ls* は LS アドレス、*ea* はシステムメモリ内の実効アドレス、*list* は DMA リストのアドレス、*list_size* は DMA リストのサイズ、*tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。

実装

```
spu_mfcdma64(ls,mfc_ea2h(ea),(unsigned int)(list), list_size, tag,
    ((tid<<24)|(rid<<16)|MFC_GETL_CMD))
```

mfc_getlb: Move Data from Effective Address to Local Storage Using MFC List with Barrier

```
(void) mfc_getlb(volatile void *ls, uint64_t ea, mfc_list_element_t *list,
    uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

MFC リストを使用して、データをシステムメモリから LS へ移動します。この関数の引数は `spu_mfcdma64` コマンドの引数に相当し、*ls* は LS アドレス、*ea* はシステムメモリ内の実効アドレス、*list* は DMA リストのアドレス、*list_size* は DMA リストのサイズ、*tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。このコマンドおよび、このコマンドと同一タグ ID を持つ後続のコマンドは、既に発行された同じタググループかつ同じコマンドキューにあるすべてのコマンドに対して、ローカルに順序付けされます。

実装

```
spu_mfcdma64(ls,mfc_ea2h(ea),(unsigned int)(list), list_size, tag,
    ((tid<<24)|(rid<<16)|MFC_GETLB_CMD))
```

mfc_getlf: Move Data from Effective Address to Local Storage Using MFC List with Fence

```
(void) mfc_getlf(volatile void *ls, uint64_t ea, mfc_list_element_t *list,
    uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

MFC リストを使用して、データをシステムメモリから LS へ移動します。この関数の引数は `spu_mfcdma64` コマンドの引数に相当し、*ls* は LS アドレス、*ea* はシステムメモリ内の実効アドレス、*list* は DMA リストのアドレス、*list_size* は DMA リストのサイズ、*tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。このコマンドは、それより前に発行された同じタググループかつ同じコマンドキューにあるすべてのコマンドに対して、ローカルに順序付けされます。

実装

```
spu_mfcdma64(ls,mfc_ea2h(ea),(unsigned int)(list), list_size, tag,
    ((tid<<24)|(rid<<16)|MFC_GETLF_CMD))
```

4.5. MFCアトミック更新コマンド

本セクションでは MFC アトミック DMA コマンドを操作するためのユティリティ関数について説明します。DMA コマンドについては、サポートされている操作のサイズ制約に関する点も含め、*Cell Broadband Engine™ アーキテクチャ* を参照してください。

表 4-103にMFCアトミック更新コマンドのニーモニックを示します。

表 4-103: MFC アトミック更新コマンドニーモニック [1]

ニーモニック	Opcode	コマンド
MFC_GETLLAR_CMD	0x00D0	getllar
MFC_PUTLLC_CMD	0x00B4	putllc
MFC_PUTLLUC_CMD	0x00B0	putlluc
MFC_PUTQLLUC_CMD	0x00B8	putqlluc

[1] MFC コマンド列挙子は spu_mfcio.h 内で定義されています。

mfc_getllar: Get Lock Line and Create Reservation

```
(void) mfc_getllar(volatile void *ls, uint64_t ea, uint32_t tid, uint32_t rid)
```

ロックラインデータを取得し、リザベーションを作成します。この関数の引数は spu_mfcdma64 コマンドの引数に相当し、*ls* は 128 バイトアラインされた LS アドレス、*ea* はシステムメモリ内の実効アドレス、*tid* は転送クラス ID、*rid* は置換クラス ID です。

mfc_getllar コマンドはタグ ID を持たず、MFC により直ちに実行されます。転送サイズは 128 バイトに固定されています。実行の完了を確認するために、本関数呼び出し後に mfc_read_atomic_status() を呼び出す必要があります。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea), mfc_ea2l(ea),128, 0,
    ((tid<<24)|(rid<<16)|MFC_GETLLAR_CMD))
```

mfc_putllc: Put Lock Line if Reservation for Effective Address Exists

```
(void) mfc_putllc(volatile void *ls, uint64_t ea, uint32_t tid, uint32_t rid)
```

実効アドレスに対するリザベーションが存在している場合、ロックラインデータを格納します。この関数の引数は spu_mfcdma64 コマンドの引数に相当し、*ls* は 128 バイトアラインされた LS アドレス、*ea* はシステムメモリ内の実効アドレス、*tid* は転送クラス ID、*rid* は置換クラス ID です。

mfc_putllc コマンドはタグ ID を持たず、MFC により直ちに実行されます。転送サイズは 128 バイトに固定されています。実行の完了を確認するために、本関数呼び出し後に mfc_read_atomic_status() を呼び出す必要があります。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea), mfc_ea2l(ea),128, 0,
    ((tid<<24)|(rid<<16)|MFC_PUTLLC_CMD))
```

mfc_putlluc: Put Lock Line Unconditional

```
(void) mfc_putlluc(volatile void *ls, uint64_t ea, uint32_t tid, uint32_t rid)
```

以前に作成したリザベーションの有無に関わらず、ロックラインデータを格納します。この関数の引数は spu_mfcdma64 コマンドの引数に相当し、*ls* は 128 バイトアラインされた LS アドレス、*ea* はシステムメモリ内の実効アドレス、*tid* は転送クラス ID、*rid* は置換クラス ID です。

このコマンドはタグ ID を持たず、MFC により直ちに実行されます。転送サイズは 128 バイトに固定されています。実行の完了を確認するために、本関数呼び出し後に mfc_read_atomic_status()を呼び出す必要があります。

実装

```
spu_mfcdma64(ls,mfc_ea2h(ea),mfc_ea2l(ea), 128, 0,
    ((tid<<24)|(rid<<16)|MFC_PUTLLUC_CMD))
```

mfc_putqlluc: Put Queued Lock Line Unconditional

```
(void) mfc_putqlluc(volatile void *ls, uint64_t ea, uint32_t tag, uint32_t tid,
    uint32_t rid)
```

以前に作成したリザベーションの有無に関わらず、ロックラインデータをキューに格納します。この関数の引数は spu_mfcdma64 コマンドの引数に相当し、*ls* は 128 バイトアラインされた LS アドレス、*ea* はシステムメモリ内の実効アドレス、*tid* は転送クラス ID、*rid* は置換クラス ID です。

転送サイズは 128 バイトに固定されています。このコマンドの機能は基本的に mfc_putlluc と同じですが、コマンドの実行順序および実行完了の確認方法において異なります。mfc_putlluc が直ちに実行されるのに対し、mfc_putqlluc は他の DMA コマンドと共に DMA コマンドキューへ投入されるため、未処理の即座実行型 mfc_getllar、mfc_putllc、mfc_putlluc コマンドの実行とは関係なく実行されます。プログラムはこのコマンドの実行完了を確認するために、タググループの完了を待つ必要があります。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea), mfc_ea2l(ea), 128, tag,
    ((tid<<24)|(rid<<16)|MFC_PUTQLLUC_CMD))
```

4.6. MFC同期コマンド

本セクションでは MFC 同期コマンド（シグナル通知やストレージの順序付けなど）を実装するための関数について説明します。DMA コマンドについては、サポートされている操作のサイズ制約に関する点も含め、Cell Broadb*and Engine™ アーキテクチャ* を参照してください。

表 4-104にMFC同期コマンドのニーモニックを示します。

表 4-104: MFC 同期コマンドニーモニック [1]

ニーモニック	Opcode	コマンド
MFC_SNDSIG_CMD	0x00A0	sndsig
MFC_SNDSIGB_CMD	0x00A1	sndsigb
MFC_SNDSIGF_CMD	0x00A2	sndsigf
MFC_BARRIER_CMD	0x00C0	barrier
MFC_EIEIO_CMD	0x00C8	mfceieio
MFC_SYNC_CMD	0x00CC	mfcsync

[1] MFC コマンド列挙子は spu_mfcio.h で定義されています。

mfc_sndsig: Send Signal

```
(void) mfc_sndsig(volatile void *ls, uint64_t ea, uint32_t tag, uint32_t tid,
    uint32_t rid)
```

mfc_sndsig コマンドを DMA キューへ投入し、DMA キューに空きがない場合はストールします。この関数の引数は spu_mfcdma64 コマンドの引数に相当し、*ls* は LS アドレス、*ea* はシステムメモリ内の実効アドレス、*tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。転送サイズは 4 バイトに固定されています。

実装

```
spu_mfcdma64(ls,mfc_ea2h(ea),mfc_ea2l(ea),4,tag,
    ((tid<<24)|(rid<<16)|MFC_SNDSIG_CMD))
```

mfc_sndsigb: Send Signal with Barrier

```
(void) mfc_sndsigb(volatile void *ls, uint64_t ea, uint32_t tag, uint32_t tid,
    uint32_t rid)
```

mfc_sndsigb コマンドを DMA キューへ投入し、DMA キューに空きがない場合はストールします。この関数の引数は spu_mfcdma64 コマンドの引数に相当し、*ls* は LS アドレス、*ea* はシステムメモリ内の実効アドレス、*tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。転送サイズは 4 バイトに固定されています。こ

のコマンドおよび、このコマンドと同一タグ ID を持つ後続のコマンドは、既に発行された同じタググループかつ同じコマンドキューにあるすべてのコマンドに対して、ローカルに順序付けされます。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea), mfc_ea2l(ea), 4, tag,
    ((tid<<24)|(rid<<16)|MFC_SNDSIGB_CMD))
```

mfc_sndsigf: Send Signal with Fence

```
(void) mfc_sndsigf(volatile void *ls, uint64_t ea, uint32_t tag, uint32_t tid,
    uint32_t rid)
```

mfc_sndsigf コマンドを DMA キューへ投入し、DMA キューに空きがない場合はストールします。この関数の引数は spu_mfcdma64 コマンドの引数に相当し、*ls* は LS アドレス、*ea* はシステムメモリ内の実効アドレス、*tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。転送サイズは 4 バイトに固定されています。このコマンドは、それより前に発行された同じタググループかつ同じコマンドキューにあるすべてのコマンドに対して、ローカルに順序付けされます。

実装

```
spu_mfcdma64(ls, mfc_ea2h(ea), mfc_ea2l(ea), 4, tag,
    ((tid<<24)|(rid<<16)|MFC_SNDSIGF_CMD))
```

mfc_barrier: Enqueue mfc_barrier Command into DMA Queue or Stall When Queue is Full

```
(void) mfc_barrier(uint32_t tag)
```

mfc_barrier コマンドを DMA キューへ投入し、DMA キューに空きがない場合はストールします。引数 *tag* は DMA タグです。mfc_barrier コマンドは、キュー内で mfc_barrier コマンドに先行する全コマンドを、mfc_barrier コマンドの後続コマンドより先に実行することを、先行あるいは後続の MFC コマンドのタグ設定にかかわらず保証します。

実装

```
spu_mfcdma32(0, 0, 0,  tag, MFC_BARRIER_CMD)
```

mfc_eieio: Enqueue mfc_eieio Command into DMA Queue or Stall When Queue is Full

```
(void) mfc_eieio (uint32_t tag, uint32_t tid, uint32_t rid)
```

mfc_eieio コマンドを DMA キューへ投入し、DMA キューに空きがない場合はストールします。引数 *tag* は DMA タグ、*tid* は転送クラス ID、*rid* は置換クラス ID です。本関数を SPE 単体内だけでのコマンド順序保証のために使用することは避けてください。mfc_eieio はプロセッサ間やデバイス間の同期での使用を意図したコマンドであり、実行するとメモリシステムへ大きな負荷がかかります。

実装

```
spu_mfcdma32(0, 0, 0, tag, ((tid<<24)|(rid<<16)|MFC_EIEIO_CMD))
```

mfc_sync: Enqueue mfc_sync Command into DMA Queue or Stall When Queue is Full

```
(void) mfc_sync (uint32_t tag)
```

mfc_sync コマンドを DMA キューへ投入し、DMA キューに空きがない場合はストールします。引数 *tag* は DMA タグです。本関数を SPE 単体内だけでのコマンド順序保証のために使用することは避けてください。mfc_sync はプロセッサ間やデバイス間の同期での使用を意図したコマンドであり、実行するとメモリシステムへ大きな負荷がかかります。

実装

```
spu_mfcdma32(0, 0, 0, tag, MFC_SYNC_CMD)
```

4.7. MFC DMAステータス

このセクションではMFCコマンドの実行完了やMFC DMAキューエントリのステータスを確認するための関数について説明します。

mfc_stat_cmd_queue: Check the Number of Available Entries in the MFC DMA Queue

```
(uint32_t) mfc_stat_cmd_queue(void)
```

DMAキューの空きスロット数を調べます。この情報は、満杯のDMAキューへコマンドを発行した場合に起こるSPUプログラムの実行停止を回避するために使用されます。満杯のキューは16エントリです。

実装

```
spu_readchcnt(MFC_Cmd)
```

mfc_write_tag_mask: Set Tag Mask to Select MFC Tag Groups to be Included in Query Operation

```
(void) mfc_write_tag_mask (uint32_t mask)
```

クエリ・オペレーションの対象タググループを選択するためのタグマスクを設定します。*mask*はDMAタググループ・クエリマスクです。各ビットはそれぞれ一つのタググループへ対応付けられており、タグ0はLSBにマップされています。

実装

```
spu_writech(MFC_WrTagMask, mask)
```

mfc_read_tag_mask: Read Tag Mask Indicating MFC Tag Groups to be Included in Query Operation

```
(uint32_t) mfc_read_tag_mask(void)
```

クエリ・オペレーションの対象となるタググループを示すタグマスクを読み込みます。マスクの各ビットはそれぞれ一つのタググループへ対応付けられており、タグ0はLSBにマップされています。得られた値はDMAタググループ・クエリマスクを表わしています。

実装

```
spu_readch(MFC_RdTagMask)
```

mfc_write_tag_update: Request That Tag Status be Updated

```
(void) mfc_write_tag_update(uint32_t ts)
```

MFCへタグステータスの更新をリクエストします。引数*ts*により 表 4-105に示すタグステータスの更新条件のいずれかを指定します。

本関数は `mfc_read_tag_status()` を使用してのタグステータス読み込み前に実行される必要があります。タグステータス更新のリクエストは `mfc_write_tag_mask()` を使用してのタググループマスク設定後に行なわれる必要があります。

表 4-105: MFC タグステータス更新条件 [1]

値	ニーモニック	説明
0	MFC_TAG_UPDATE_IMMEDIATE	無条件に直ちに更新。
1	MFC_TAG_UPDATE_ANY	対象となるタググループのいずれかが「未完了のオペレーションが無い」状態にある（状態になった）時に更新。
2	MFC_TAG_UPDATE_ALL	対象となる全てのタググループが「未完了のオペレーションが無い」状態にある（状態になった）時に更新。

[1] 条件列挙子は `spu_mfcio.h` 内で定義されています。

実装

```
spu_writech(MFC_WrTagUpdate, ts)
```

mfc_write_tag_update_immediate: Request That Tag Status be Immediately Updated

```
(void) mfc_write_tag_update_immediate(void)
```

タグステータスの即時更新をリクエストします。

実装

```
spu_writech(MFC_WrTagUpdate, MFC_TAG_UPDATE_IMMEDIATE)
```

mfc_write_tag_update_any: Request That Tag Status be Updated for Any Enabled Completion with No Outstanding Operation

```
(void) mfc_write_tag_update_any(void)
```

対象となる MFC タググループのいずれかが「未完了のオペレーションが無い」状態にある（なった）時にタグステータスを更新するようにリクエストします。

実装

```
spu_writech(MFC_WrTagUpdate, MFC_TAG_UPDATE_ANY)
```

mfc_write_tag_update_all: Request That Tag Status be Updated When All Enabled Tag Groups Have No Outstanding Operation

```
(void) mfc_write_tag_update_all(void)
```

対象となる全てのタググループが「未完了のオペレーションが無い」状態にある（なった）時にタグステータスを更新するようにリクエストします。

実装

```
spu_writech(MFC_WrTagUpdate, MFC_TAG_UPDATE_ALL)
```

mfc_stat_tag_update: Check Availability of Tag Status Update Request Channel

```
(uint32_t) mfc_stat_tag_update(void)
```

タグステータス更新リクエストチャネルが利用可能であるか否かを確認します。結果は以下のいずれかの値として返ります。

- 0: チャネルは未だ利用できる状態にない。
- 1: チャネルは利用できる状態にある。

実装

```
spu_readchcnt(MFC_WrTagUpdate)
```

mfc_read_tag_status: Wait for an Updated Tag Status

```
(uint32_t) mfc_read_tag_status(void)
```

タググループのステータスを読み込みます。タグステータスの更新条件が MFC_TAG_UPDATE_IMMEDIATE に設定されていない場合にこの関数を呼び出すとブロックする可能性があります。返り値の各ビットは各タググループに対応しており、タグ 0 は LSB にマップされています。ビットがセットされていると、対応するタググループに実行の完了していないコマンドがなく（つまりコマンドは完了しており）、かつクエリによりマスクされていないことを意味します。

返り値については、タグステータス更新時にマスクされていないタググループのステータスのみが有効です。タグステータス更新時にマスクされているタググループに対応するビットは 0 となります。

実装

```
spu_readch(MFC_RdTagStat)
```

mfc_read_tag_status_immediate: Wait for the Updated Status of Any Enabled Tag Group

```
(uint32_t) mfc_read_tag_status_immediate(void)
```

タグステータスの即時更新をリクエストします。プロセッサはステータスが更新されるまで待ちます。

実装

```
spu_mfcstat(MFC_TAG_UPDATE_IMMEDIATE)
```

mfc_read_tag_status_any: Wait for No Outstanding Operation of Any Enabled Tag Group

```
(uint32_t) mfc_read_tag_status_any(void)
```

対象となる MFC タググループのいずれかが「未完了のオペレーションが無い」状態にある（なった）時にタグステータスを更新するようにリクエストします。プロセッサはステータスが更新されるまで待ちます。

実装

```
spu_mfcstat(MFC_TAG_UPDATE_ANY)
```

mfc_read_tag_status_all: Wait for No Outstanding Operation of All Enabled Tag Groups

```
(uint32_t)mfc_read_tag_status_all(void)
```

対象となる MFC タググループの全てが「未完了のオペレーションが無い」状態にある（なった）時にタグステータスを更新するようにリクエストします。プロセッサはステータスが更新されるまで待ちます。

実装

```
spu_mfcstat(MFC_TAG_UPDATE_ALL)
```

mfc_stat_tag_status: Check Availability of MFC_RdTagStat Channel

```
(uint32_t)mfc_stat_tag_status(void)
```

MFC_RdTagStat チャネルが使用可能な状態にあるか否かを確認し、以下のいずれかの値を返します。

- 0: タグステータスは未だ読み出せる状態にない。
- 1: タグステータスは読み出せる状態である。

本関数は、タグステータスを読み出すことができる状態ではないときに MFC_RdTagStat チャネルを読み込むことによって起こるチャネルストールを回避するために使用します。

実装

```
spu_readchcnt(MFC_RdTagStat)
```

mfc_read_list_stall_status: Read List DMA Stall-and-Notify Status

```
(uint32_t) mfc_read_list_stall_status(void)
```

List DMA のストール/通知ステータスを読み出して返します。ステータスを読み出せる状態にない場合は、読み出せる状態になるまでストールします。

実装

```
spu_readch(MFC_RdListStallStat)
```

mfc_stat_list_stall_status: Check Availability of List DMA Stall-and-Notify Status

```
(uint32_t) mfc_stat_list_stall_status(void)
```

List DMA のストール/通知ステータスが読み出し可能な状態にあるか否かを確認し、以下のいずれかの値を返します。

- 0: ステータスは未だ読み出せる状態にない。
- 1: ステータスは読み出せる状態である。

実装

```
spu_readchcnt(MFC_RdListStallStat)
```

mfc_write_list_stall_ack: Acknowledge Tag Group Containing Stalled DMA List Commands

```
(void) mfc_write_list_stall_ack(uint32_t tag)
```

前のストール/通知イベントについてアクノリッジを送ります。（mfc_read_list_status および mfc_stat_list_stall_status の項目を参照してください。）引数 *tag* は DMA タグです。

実装

```
spu_writech(MFC_WrListStallAck, tag)
```

mfc_read_atomic_status: Read Atomic Command Status

```
(uint32_t) mfc_read_atomic_status(void)
```

アトミックコマンドのステータスを読み出します。ステータスが読み出せる状態にない場合は、読み出せる状態になるまでストールします。返り値（29 から 31 ビット目のバイナリ値）は 表 4-106に示すうちのいずれかとなります。

表 4-106: Read Atomic Command Status or Stall Until Status Is Available の返り値 [1]

ステータス	ニーモニック	説明
1	MFC_PUTLLC_STATUS	0: `mfc_putllc` コマンドの実行に成功した。 1: `mfc_putllc` コマンドの実行に失敗した（リザベーションが消失した）。
2	MFC_PUTLLUC_STATUS	`mfc_putlluc` コマンドの処理が完了している。
4	MFC_GETLLAR_STATUS	`mfc_getllar` コマンドの処理が完了している。

[1] ステータス列挙子は `spu_mfcio.h` で定義されています

実装

```
spu_readch(MFC_RdAtomicStat)
```

mfc_stat_atomic_status: Check Availability of Atomic Command Status

```
(uint32_t) mfc_stat_atomic_status(void)
```

アトミックコマンドのステータスが読み出せる状態にあるか否かを確認し、以下のいずれかの値を返します。

- 0: アトミック DMA コマンドの処理が完了していない。
- 1: アトミック DMA コマンドの処理が完了しており、ステータスが使用可能である。

実装

```
spu_readchcnt(MFC_RdAtomicStat)
```

4.8. MFC マルチソース同期要求

Cell Broadband Engine™ アーキテクチャ には MFC マルチソース同期機構についての記述があり、その中で「累積的順序付け」とは簡単に言うと、複数のプロセッサ（またはユニット）による、他のプロセッサ（またはユニット）にとってのメモリアクセスの順序付けと定義されています。本セクションでは LS およびメインメモリアドレス領域に渡る累積的順序付けを実現するために利用できるいくつかの関数について説明します。

mfc_write_multi_src_sync_request: Request Multisource Synchronization

```
(void) mfc_write_multi_src_sync_request(void)
```

対応する MFC 宛ての転送で未完了のもののトラッキングを開始するようリクエストします。リクエストされた同期が完了すると、MFC マルチソース同期リクエストチャネルのチャネルカウントが 1 にリセットされます。

実装

```
spu_writech(MFC_WrMSSyncReq,0)
```

mfc_stat_multi_src_sync_request: Check the Status of Multisource Synchronization

```
(uint32_t) mfc_stat_multi_src_sync_request(void)
```

MFC マルチソース同期リクエストチャネルのチャネルカウントを読み出し、以下のいずれかの値を返します。

- 0: 未完了の転送をトラッキング中である。
- 1: `mfc_write_multi_src_sync_request` でリクエストした同期が完了している。

実装

```
spu_readchcnt(MFC_WrMSSyncReq)
```

4.9. SPUシグナル通知

本セクションではシステム内の他のプロセッサやデバイスからのシグナルを読む際に利用できる関数について説明します。

spu_read_signal1: Atomically Read and Clear Signal Notification 1 Channel

```
(uint32_t) spu_read_signal1(void)
```

Signal Notification 1 チャネルを読み出し、セットされているビットがあればアトミックにリセットします。未処理のシグナルがある場合はそのシグナルを返し、未処理のシグナルがない場合、シグナルが発行されるまで SPU をストールさせます。

実装

```
spu_readch(SPU_RdSigNotify1)
```

spu_stat_signal1: Check if Any Pending Signals Exist on Signal Notification 1 Channel

```
(uint32_t) spu_stat_signal1(void)
```

Signal Notification 1 チャネルに未処理のシグナルが存在するか否かを確認し、以下のいずれかの値を返します。

- 0: 未処理のシグナルは存在しない。
- 1: 未処理のシグナルが存在する。

実装

```
spu_readchcnt(SPU_RdSigNotify1)
```

spu_read_signal2: Atomically Read and Clear Signal Notification 2 Channel

```
(uint32_t) spu_read_signal2(void)
```

Signal Notification 2 チャネルを読み出し、セットされているビットがあればアトミックにリセットします。未処理のシグナルがある場合はそのシグナルを返し、未処理のシグナルがない場合、シグナルが発行されるまで SPU をストールさせます。

実装

```
spu_readch(SPU_RdSigNotify2)
```

spu_stat_signal2: Check if Any Pending Signals Exist on Signal Notification 2 Channel

```
(uint32_t) spu_stat_signal2(void)
```

Signal Notification 2 チャネルに未処理のシグナルが存在するか否かを確認し、以下のいずれかの値を返します。

- 0: 未処理のシグナルは存在しない。
- 1: 未処理のシグナルが存在する。

実装

```
spu_readchcnt(SPU_RdSigNotify2)
```

4.10. SPUメールボックス

本セクションでは SPU メールボックスを操作する際に利用できる関数について説明します。

spu_read_in_mbox: Read Next Data Entry in SPU Inbound Mailbox

```
(uint32_t) spu_read_in_mbox(void)
```

SPU Inbound Mailbox キュー内の次のデータエントリを読み出します。キューが空の場合はストールします。返り値は当該アプリケーション固有のメールボックスデータです。各アプリケーション独自のメールボックスデータを定義することができます。

実装

```
spu_readch(SPU_RdInMbox)
```

spu_stat_in_mbox: Get the Number of Data Entries in SPU Inbound Mailbox

```
(uint32_t) spu_stat_in_mbox(void)
```

SPU Inbound Mailbox キュー内のデータエントリ数を取得して返します。返り値が 0 ではない場合、メールボックス内に SPU が読み出していないデータエントリが存在することを意味します。

実装

```
spu_readchcnt(SPU_RdInMbox)
```

spu_write_out_mbox: Send Data to SPU Outbound Mailbox

```
(void) spu_write_out_mbox (uint32_t data)
```

SPU Outbound Mailbox へ *data*（アプリケーション独自に定義されたメールボックスデータ）を送ります。このメールボックスに空がない場合はストールします。

実装

```
spu_writech(SPU_WrOutMbox, data)
```

spu_stat_out_mbox: Get Available Capacity of SPU Outbound Mailbox

```
(uint32_t) spu_stat_out_mbox(void)
```

SPU Outbound Mailbox の現在の空き容量を取得し、その値を返します。返り値が 0 の場合、メールボックスが満杯であることを意味します。

実装

```
spu_readchcnt(SPU_WrOutMbox)
```

spu_write_out_intr_mbox: Send Data to SPU Outbound Interrupt Mailbox

```
(void) spu_write_out_intr_mbox (uint32_t data)
```

SPU Outbound Interrupt Mailbox へ *data*（アプリケーション独自に定義されたメールボックスデータ）を送ります。このメールボックスに空きがない場合はストールします。

実装

```
spu_writech(SPU_WrOutIntrMbox, data)
```

spu_stat_out_intr_mbox: Get Available Capacity of SPU Outbound Interrupt Mailbox

```
(uint32_t) spu_stat_out_intr_mbox(void)
```

SPU Outbound Interrupt Mailbox の現在の空き容量を取得し、その値を返します。返り値が 0 の場合、メールボックスが満杯であることを意味します。

実装

```
spu_readchcnt(SPU_WrOutIntrMbox)
```

4.11. SPUデクリメンタ

本セクションでは SPU の 32 ビットデクリメンタを使用する関数について説明します。

spu_read_decrementer: Read Current Value of Decrementer

```
(uint32_t) spu_read_decrementer(void)
```

デクリメンタの現在の値を読み出して返します。

実装

```
spu_readch(SPU_RdDec)
```

spu_write_decrementer: Load a Value to Decrementer

```
(void) spu_write_decrementer (uint32_t count)
```

`count` の値をデクリメンタへロードします。

実装

```
spu_writech(SPU_WrDec, count)
```

4.12. SPUイベント

本セクションでは SPU イベント監視のために利用できるいくつかの関数について説明します。SPU イベント機構の説明については *Cell Broadband Engine™ アーキテクチャ* を参照してください。

表 4-107にEvent Status、Event Mask、およびEvent Ackのビットフィールドを示します。

表 4-107: MFC イベントビットフィールド [1]

ビット	フィールド名	説明
0x1000	MFC_MULTI_SRC_SYNC_EVENT	マルチソース同期イベント
0x0800	MFC_PRIV_ATTN_EVENT	SPU 特権アテンションイベント
0x0400	MFC_LLR_LOST_EVENT	ロックラインリザベーション消失イベント
0x0200	MFC_SIGNAL_NOTIFY_1_EVENT	SPU シグナル通知 1 利用可能イベント
0x0100	MFC_SIGNAL_NOTIFY_2_EVENT	SPU シグナル通知 2 利用可能イベント
0x0080	MFC_OUT_MBOX_AVAILABLE_EVENT	SPU Outbound Mailbox 利用可能イベント
0x0040	MFC_OUT_INTR_MBOX_AVAILABLE_EVENT	SPU Outbound Interrupt Mailbox 利用可能イベント
0x0020	MFC_DECREMENTER_EVENT	SPU デクリメンタイベント
0x0010	MFC_IN_MBOX_AVAILABLE_EVENT	SPU Inbound Mailbox 利用可能イベント
0x0008	MFC_COMMAND_QUEUE_AVAILABLE_EVENT	MFC SPU コマンドキュー利用可能イベント
0x0002	MFC_LIST_STALL_NOTIFY_EVENT	MFC DMA List コマンドストール/通知イベント
0x0001	MFC_TAG_STATUS_UPDATE_EVENT	MFC タググループステータス更新イベント

[1] ビットフィールド名は `spu_mfcio.h` で定義されています。

spu_read_event_status: Read Event Status or Stall Until Status is Available

```
(uint32_t) spu_read_event_status(void)
```

イベントステータスを読み出し返します。ステータスが読み出せる状態にない場合は読み出せる状態になるまでストールします。通知されてまだアクノリッジされていないイベントはアクノリッジされるまで通知状態に保持されます。

ステータスとして返るのは SPU Read Event Status チャネルの値です。

実装

```
spu_readch(SPU_RdEventStat)
```

spu_stat_event_status: Check Availability of Event Status

```
(uint32_t) spu_stat_event_status(void)
```

未処理の監視対象イベントが存在するか否かを確認し、以下のいずれかの値を返します。

- 0: 監視対象のイベントは発生していない。
- 1: 未処理の監視対象イベントが存在する。

実装

```
spu_readchcnt(SPU_RdEventStat)
```

spu_write_event_mask: Select Events to be Monitored by Event Status

```
(void) spu_write_event_mask (uint32_t mask)
```

イベントステータスの監視対象となるイベントを選択します。引数 *mask* はイベントマスクです。

実装

```
spu_writech(SPU_WrEventMask, mask)
```

spu_write_event_ack: Acknowledge Events

```
(void) spu_write_event_ack (uint32_t ack)
```

当該イベントがソフトウェアにより処理されていることをアクノレッジします。アクノレッジされたイベントは、ステータスがリセットされ、イベントが再サンプリングされます。引数 *ack* はイベントアクノレッジを表します。

実装

```
spu_writech(SPU_WrEventAck, ack)
```

spu_read_event_mask: Read Event Status Mask

```
(uint32_t) spu_read_event_mask(void)
```

イベントマスクの現在の値を読み出して返します。

実装

```
spu_readch(SPU_RdEventMask)
```

4.13. SPU状態管理

本セクションでは割り込み関連の関数について説明します。SPU Machine Status チャネルおよび割り込み関連の SPU チャネルについては *Cell Broadband Engine™ アーキテクチャ* を参照してください。

spu_read_machine_status: Read Current SPU Machine Status

```
(uint32_t) spu_read_machine_status(void)
```

現在の SPU マシンステータスを読み出し、ステータスを返します。

実装

```
spu_readch(SPU_RdMachStat)
```

spu_write_srr0: Write to SPU SRR0

```
(void) spu_write_srr0(uint32_t srr0)
```

srr0 の値を SRR0（SPU state save/restore register 0）レジスタへ書き込みます。

実装

```
spu_writech(SPU_WrSRR0,srr0)
```

spu_read_srr0: Read SPU SRR0

```
(uint32_t) spu_read_srr0(void)
```

SRR0（SPU state save/restore register 0）レジスタの値を読み出して返します。

実装

```
spu_readch(SPU_RdSRR0)
```

5. SPU組み込み関数とVector Multimedia Extension組み込み関数

関数マッピングの手法を用いることにより、SPU 組み込み関数で記述されたソースコードの移植性を高めることができます。組み込み関数のマッピングで重要なものの一つに、SPU と PPU 間のマッピングがあります。本章では SPU 組み込み関数と PPU Vector Multimedia Extension 組み込み関数間で最小限必要なマッピングについて記述します。

多くの組み込み関数にはアーキテクチャ間で効率的な一対一のマッピング方法が存在します。一部の組み込み関数については、効率は劣りますが一対多の命令マッピングを行なうことが可能です。また、マップすることが実用的でないあるいは実現不可能であるという理由により、直接的なマッピングがなされない関数も存在します。本仕様では SPU と PPU 間における一対一のマッピングについてのみ記述します。直接的にマップされない SPU と PPU の組み込み関数については、マッピングが困難であることの理由を記載しています。

SPU 組み込み関数と PPU 組み込み関数間のマッピングは二つのヘッダファイル（vmx2spu.h、spu2vmx.h）に定義されています。前者は Vector Multimedia Extension 組み込み関数の総称 SPU 組み込み関数へのマッピングを、後者は総称 SPU 組み込み関数の Vector Multimedia Extension 組み込み関数へのマッピングをそれぞれ定義しています。これら二つのヘッダファイルで定義された関数は、多重定義インライン関数として実装することもできます。実装を容易にするためにベクタデータ型もマップする必要があります。

ヘッダファイル「vec_types.h」は、Vector Multimedia Extension用の単一トークンベクタデータ型を宣言するため、またSPUとVector Multimedia Extension間のデータ型マッピングを行なうために提供されます。プログラマはベクタデータをこれらの単一トークンデータ型を用いて同様に宣言するする必要があります。Vector Multimedia Extension組み込み関数用の単一トークンベクタデータ型を 表 5-108に示します。

表 5-108: Vector Multimedia Extension 単一トークンベクタデータ型

ベクタキーワードデータ型	単一トークンベクタデータ型定義
vector unsigned char	vec_uchar16
vector signed char	vec_char16
vector bool char	vec_bchar16
vector unsigned short	vec_ushort8
vector signed short	vec_short8
vector bool short	vec_bshort8
vector unsigned int	vec_uint4
vector signed int	vec_int4
vector bool int	vec_bint4
vector float	vec_float4
vector pixel	vec_pixel8

5.1. Vector Multimedia Extension組み込み関数のSPU組み込み関数へのマッピング

本セクションでは Vector Multimedia Extension 組み込み関数の SPU 組み込み関数への一対一マッピングを列挙します。SPU 組み込み関数へのマップが難しい Vector Multimedia Extension 組み込み関数についても列挙します。.

5.1.1. 一対一でマップされた組み込み関数

表 5-109にSPU組み込み関数へ一対一でマップされたVector Multimedia Extension組み込み関数を示します。

表 5-109: SPU 組み込み関数へ一対一でマップされる Vector Multimedia Extension 組み込み関数

総称 Vector Multimedia Extension 組み込み関数	SPU 組み込み関数へのマッピング	使用できる型
vec_add	spu_add	halfword、word、float (byte は使用不可)
vec_addc	spu_genc	全ての型
vec_and	spu_and	全ての型
vec_andc	spu_andc	全ての型
vec_avg	spu_avg	unsigned char
vec_cmpeq	spu_cmpeq	全ての型
vec_cmpgt	spu_cmpgt	全ての型
vec_cmplt	spu_cmpgt	全ての型、パラメータの並び替えが必要
vec_ctf	spu_convtf	全ての型
vec_cts	spu_convts	全ての型
vec_ctu	spu_convtu	全ての型
vec_madd	spu_madd	全ての型
vec_mule	spu_mule	halfword (byte は使用不可)
vec_mulo	spu_mulo	halfword (byte は使用不可)
vec_nmusb	spu_nmsub	全ての型
vec_nor	spu_nor	全ての型
vec_or	spu_or	全ての型
vec_re	spu_re	全ての型
vec_rl	spu_rl	halfword、word (byte は使用不可)
vec_rsqrte	spu_rsqrte	全ての型
vec_sel	spu_sel	全ての型
vec_sub	spu_sub	halfword、word、float
vec_subc	spu_genb	全ての型
vec_xor	spu_xor	全ての型

5.1.2. SPU組み込み関数へマップすることが困難なVector Multimedia Extension組み込み関数

表 5-110に示すVector Multimedia Extension組み込み関数は、総称SPU組み込み関数への簡単なマッピング方法がないため、総称SPU組み込み関数へマップされることはないものと考えられます。

表 5-110: SPU 組み込み関数へマップすることが困難な Vector Multimedia Extension 組み込み関数

総称 Vector Multimedia Extension 組み込み関数	説明
vec_unpackh,vec_unpackl	この関数はSPUデータ型を追加定義することなくマップすることはできない。（表 1-2で述べたように）ベクタ型`pixel`および `bool short` が`unsigned short`へマップされているので、多重定義関数選択時に競合が発生する。
vec_mfvscr, vec_mtvscr	SPU は単精度浮動小数点演算で IEEE の丸めモードをサポートしていないため、VSCR レジスタをサポートすることは困難である。
vec_step	この関数をマップするには、本仕様では定められていない特定のコンパイラでのサポートが必要になる。

5.2. SPU組み込み関数のVector Multimedia Extension組み込み関数へのマッピング

本セクションでは SPU 組み込み関数の Vector Multimedia Extension 組み込み関数への一対一マッピングを列挙します。Vector Multimedia Extension 組み込み関数へのマップが難しい SPU 組み込み関数についても列挙します。

5.2.1. 一対一でマップされた組み込み関数

総称SPU組み込み関数の多くはVector Multimedia Extension組み込み関数へ一対一でマップされています。表 5-111にこれらのマッピングを示します。

表 5-111: Vector Multimedia Extension 組み込み関数へ一対一でマップされる SPU 組み込み関数

総称 SPU 組み込み関数	Vector Multimedia Extension 組み込み関数へのマッピング	使用できる型
spu_add	vec_add	vector/vector (スカラオペランドは不可)
spu_and	vec_and	vector/vector (スカラオペランドは不可)
spu_andc	vec_andc	全ての型
spu_avg	vec_avg	全ての型
spu_cmpeq	vec_cmpeq	vector/vector (スカラオペランドは不可)
spu_cmpgt	vec_cmpgt	vector/vector (スカラオペランドは不可)
spu_convtf	vec_ctf	限られた範囲内の換算係数(5 ビット)
spu_convts	vec_cts	限られた範囲内の換算係数(5 ビット)
spu_convtu	vec_ctu	限られた範囲内の換算係数(5 ビット)
spu_genb	vec_subc	全ての型
spu_genc	vec_addc	全ての型
spu_madd	vec_madd	float
spu_mule	vec_mule	全ての型
spu_mulo	vec_mulo	halfword vector/vector(スカラオペランドは不可)
spu_nmsub	vec_nmsub	float
spu_nor	vec_nor	全ての型
spu_or	vec_or	vector/vector (スカラオペランドは不可)
spu_re	vec_re	全ての型
spu_rl	vec_rl	vector/vector (スカラオペランドは不可)
spu_rsqrte	vec_rsqrte	全ての型
spu_sel	vec_sel	全ての型
spu_sub	vec_sub	vector/vector (スカラオペランドは不可)
spu_xor	vec_xor	vector/vector (スカラオペランドは不可)

5.2.2. Vector Multimedia Extension組み込み関数へマップすることが困難なSPU組み込み関数

表 5-112に示す総称SPU組み込み関数は、Vector Multimedia Extension組み込み関数への簡単なマッピング方法がないため、Vector Multimedia Extension組み込み関数へマップされることはないものと考えられます。

表 5-112: Vector Multimedia Extension 組み込み関数へマップすることが困難な SPU 組み込み関数

総称 SPU 組み込み関数	説明
spu_bisled, spu_bislede, spu_bisledi	SPU のイベント処理や割り込み処理は正確にマップすることができない。
spu_idisable, spu_ienable	
spu_readch, spu_readchqw, spu_readchcnt	特定のチャネル機能について、PPU でサポートすることは容易ではなく、またサポートすることは一般的に望ましくない。チャネルシーケンスの一部はマップできる可能性があるものの、ほとんどの場合プログラマの格別な見識と指示を要する。
spu_writech, spu_writechqw	
spu_mfcdma32, spu_mfcdma64, spu_mfcstat	DMA トランザクションのマッピングは、PPU がメモリへフルアクセスできるため通常必要ない。しかしながら、これらの組み込み関数を、正確なマッピングができるとは限らないメモリ同期のために用いることも可能である。
spu_sync, spu_sync_c	これらの組み込み関数は PPU の同期命令のうちの一つにマップすることが可能ではあるものの、意図した結果が得られない可能性がある。
spu_dsync	
spu_convts, spu_convtu, spu_convtf	換算係数について完全にダイナミックなビット範囲をサポートすることは容易ではない。Vector Multimedia Extension は 5 ビットの換算係数を提供するのに対し、SPU の換算係数は 8 ビットである。実装によっては、同等の組み込み関数を直接マップすることにより提供される 5 ビット範囲のみをサポートするかもしれない。
spu_hcmpeq, spu_hcmpgt	停止命令は exit 関数へのマッピングが可能である場合があるものの、環境によってはこのようなマッピングはできない。
spu_stop, spu_stopd	PPU の実行を停止することが常に適切であるとは限らない。

6. PPU VMX組み込み関数

本章では、C/C++プログラミング言語から下層に位置する PPU VMX 命令セットをアクセスできるようにする組み込み関数について説明します。*AltiVec Technology Programming Interface Manual* の Section 4.4 では PPU VMX 命令セットに対する総称組み込み関数の大部分を定義していますが、いくつかの新命令に対するものを本章で規定します。新しい組み込み関数は 2 種類のカテゴリに属します。ベクタ要素の抽出用の組み込み関数とベクタ要素の挿入用の組み込み関数です。

PPU VMX 組み込み関数はシステムヘッダファイル `altivec.h` で宣言されることになりますが、このヘッダ内でマクロとして定義するか、あるいはコンパイラの内部で実装するか、いずれでもかまいません。

データプリフェッチについては、`__dcbt`、`__dcbtst`、`__dcbt_TH1000`、`__dcbt_TH1010` 組み込み関数を使用してください。これに関連した *AltiVec Technology Programming Interface Manual* に定義されているストリーム制御命令（下記参照）は、PPU では廃止されており、NOP として実行されます。

表 6-113: PPU では廃止されたストリーム制御命令

ストリーム制御命令	説明
vec_dss(a)	Vector Data Stream Stop
vec_dssall()	Vector Stream Stop All
vec_dst(a,b,c)	Vector Stream Touch
vec_dstst(a,b,c)	Vector Data Stream Touch for Store Transient

vec_extract: Extract Vector Element from Vector

```
d = vec_extract(a, element)
```

element で指定される要素をベクタ *a* から抽出しスカラ *d* に返します。要素のサイズに応じて *element* の最下位ビット側の限られたビット数のみをインデックスに使用します。具体的には、1, 2, 4 バイトの要素について、最下位側のそれぞれ 4, 3, 2 ビットのみを使用します。

表 6-114: Extract Vector Element from Vector

戻り値／引数の型			アセンブリ命令へのマップ [1]
d	a	element	
unsigned char	vector unsigned char	int	EA=memaddr + (element&0xF) stvebx a, 0, EA lbzx d, 0, EA
signed char	vector signed char		EA=memaddr + (element&0xF) stvebx a, 0, EA lbzx d, 0, EA extsb d, d
unsigned short	vector unsigned short		EA=memaddr + (element&0x7)<<2 stvehx a, 0, EA lhzx d, 0, EA
signed short	vector signed short		EA=memaddr + (element&0x7)<<2 stvehx a, 0, EA lhzx d, 0, EA extsh d, d
unsigned int	vector unsigned int		EA=memaddr + (element&0x3)<<3 stvewx a, 0, EA lwzx a, 0, EA
signed int	vector signed int		EA=memaddr + (element&0x3)<<3 stvewx a, 0, EA lwzx a, 0, EA extsw d, d [2]
float	vector float		EA=memaddr + (element&0x3)<<3 stvewx a, 0, EA lfsx a, 0, EA

[1] memaddr は 16 バイトアラインの一時メモリロケーションのアドレスです。

[2] プロセッサが 32 ビットモードで動作している場合は、ワードからダブルワードへの符号拡張は省略できます。

vec_insert: Insert Scalar into Specified Vector Element

```
d = vec_insert(a, b, element)
```

スカラ *a* をベクタ *b* の *element* パラメタで指定される要素に挿入し、変更されたベクタを返します。*b* のその他の要素は変更されません。要素のサイズに応じて *element* の最下位ビット側の限られたビット数のみをインデックスに使用します。具体的には、1, 2, 4 バイトの要素について、最下位側のそれぞれ 4, 3, 2 ビットのみを使用します。

表 6-115: Insert Scalar into Specified Vector Element

戻り値／引数の型				アセンブリ命令へのマップ [1]
d	a	b	element	
vector unsigned char	unsigned char	vector unsigned char	int	EA=memaddr + (element&0xF) stbx a, 0, EA lvebx d, 0, EA vperm d, d, a, pattern
vector signed char	signed char	vector signed char		
vector unsigned short	unsigned short	vector unsigned short		EA=memaddr + (element&0x7)<<2 sthx a, 0, EA lvehx d, 0, EA vperm d, d, a, pattern
vector signed short	signed short	vector signed short		
vector unsigned int	unsigned int	vector unsigned int		EA=memaddr + (element&0x3)<<3 stwx a, 0, EA lvewx d, 0, EA vperm d, d, a, pattern
vector signed int	signed int	vector signed int		
vector float	float	vector float		EA=memaddr + (element&0x3)<<3 stfsx a, EA lvewx d, 0, EA vperm d, d, a, pattern

[1] memaddr は 16 バイトアラインの一時メモリロケーションのアドレスです。

vec_lvlx: Load Vector Left Indexed

```
d = vec_lvlx(a, b)
```

EA を *a* の内容と *b* の内容の和にて形成される実効アドレスとし、eb を EA の最下位側 4 ビットの値とします。EA でアドレス指定される (16 - eb) バイトが *d* の左端 (16 - eb) バイトの要素にロードされ、*d* の右端 eb バイトはゼロに設定されます。

表 6-116: Load Vector Left Indexed

戻り値／引数の型			アセンブリ命令へのマップ
d	a	b	
vector unsigned char	any integral type	unsigned char *	lvlx d, a, b
		vector unsigned char *	
vector signed char	any integral type	signed char *	
		vector signed char *	
vector bool char	any integral type	vector bool char *	
vector unsigned short	any integral type	unsigned short *	
		vector unsigned short *	
vector signed short	any integral type	signed short *	
		vector signed short *	
vector bool short	any integral type	vector bool short *	
vector pixel	any integral type	vector pixel *	
vector unsigned int	any integral type	unsigned int *	
		vector unsigned int *	
vector signed int	any integral type	signed int *	
		vector signed int *	
vector bool int	any integral type	vector bool int *	
vector float	any integral type	float *	
		vector float *	

vec_lvlxl: Load Vector Left Indexed Last

```
d = vec_lvlxl(a, b)
```

EA を *a* の内容と *b* の内容の和にて形成される実効アドレスとし、eb を EA の最下位側 4 ビットの値とします。EA でアドレス指定される (16 - eb) バイトが *d* の左端 (16 - eb) バイトの要素にロードされ、*d* の右端 eb バイトはゼロに設定されます。vec_lvlxl は、EA でアドレス指定されるメモリのクワッドワードが近い将来プログラムで再度必要とされることはおそらくないであろうというヒントを与えます。

表 6-117: Load Vector Left Indexed Last

戻り値／引数の型			アセンブリ命令へのマップ
d	a	b	
vector unsigned char	any integral type	unsigned char *	lvlxl d, a, b
		vector unsigned char *	
vector signed char	any integral type	signed char *	
		vector signed char *	
vector bool char	any integral type	vector bool char *	
vector unsigned short	any integral type	unsigned short *	
		vector unsigned short *	
vector signed short	any integral type	signed short *	
		vector signed short *	
vector bool short	any integral type	vector bool short *	
vector pixel	any integral type	vector pixel *	
vector unsigned int	any integral type	unsigned int *	
		vector unsigned int *	
vector signed int	any integral type	signed int *	
		vector signed int *	
vector bool int	any integral type	vector bool int *	
vector float	any integral type	float *	
		vector float *	

vec_lvrx: Load Vector Right Indexed

```
d = vec_lvrx(a, b)
```

EA を *a* の内容と *b* の内容の和にて形成される実効アドレスとし、eb を EA の最下位側 4 ビットの値とします。eb がゼロでなければ（例えば EA がクワッドワードにアラインしていなければ）、(EA - eb) でアドレス指定されるメモリ内の eb バイトが *d* の右端 eb バイトにロードされ、*d* の左端 (16 - eb) バイトはゼロに設定されます。eb がゼロであれば（例えば EA がクワッドワードにアラインしていれば）、*d* の内容はゼロに設定されます。

表 6-118: Load Vector Right Indexed

戻り値／引数の型			アセンブリ命令へのマップ
d	a	b	
vector unsigned char	any integral type	unsigned char *	lvrx d, a, b
		vector unsigned char *	
vector signed char	any integral type	signed char *	
		vector signed char *	
vector bool char	any integral type	vector bool char *	
vector unsigned short	any integral type	unsigned short *	
		vector unsigned short *	
vector signed short	any integral type	signed short *	
		vector signed short *	
vector bool short	any integral type	vector bool short *	
vector pixel	any integral type	vector pixel *	
vector unsigned int	any integral type	unsigned int *	
		vector unsigned int *	
vector signed int	any integral type	signed int *	
		vector signed int *	
vector bool int	any integral type	vector bool int *	
vector float	any integral type	float *	
		vector float *	

vec_lvrxl: Load Vector Right Indexed Last

```
d = vec_lvrxl(a,b)
```

EA を *a* の内容と *b* の内容の和にて形成される実効アドレスとし、eb を EA の最下位側 4 ビットの値とします。eb がゼロでなければ（例えば EA がクワッドワードにアラインしていなければ）、(EA - eb) でアドレス指定されるメモリ内の eb バイトが *d* の右端 eb バイトにロードされ、*d* の左端 (16 - eb) バイトはゼロに設定されます。eb がゼロであれば（例えば EA がクワッドワードにアラインしていれば）、*d* の内容はゼロに設定されます。
vec_lvrxl は、EA でアドレス指定されるメモリのクワッドワードが近い将来プログラムで再度必要とされることはおそらくないであろうというヒントを与えます。

表 6-119: Load Vector Right Indexed Last

戻り値／引数の型			アセンブリ命令へのマップ
d	a	b	
vector unsigned char	any integral type	unsigned char *	lvrxl d, a, b
		vector unsigned char *	
vector signed char	any integral type	signed char *	
		vector signed char *	
vector bool char	any integral type	vector bool char *	
vector unsigned short	any integral type	unsigned short *	
		vector unsigned short *	
vector signed short	any integral type	signed short *	
		vector signed short *	
vector bool short	any integral type	vector bool short *	
vector pixel	any integral type	vector pixel *	
vector unsigned int	any integral type	unsigned int *	
		vector unsigned int *	
vector signed int	any integral type	signed int *	
		vector signed int *	
vector bool int	any integral type	vector bool int *	
vector float	any integral type	float *	
		vector float *	

vec_stvlx: Store Vector Left Indexed

```
(void) vec_stvlx(a, b, c)
```

EA を *b* の内容と *c* の内容の和にて形成される実効アドレスとし、eb を EA の最下位側 4 ビットの値とします。 *a* の左端 (16 - eb) バイトを EA でアドレス指定されるメモリにストアします。

表 6-120: Store Vector Left Indexed

戻り値／引数の型			アセンブリ命令へのマップ
a	b	c	
vector unsigned char	any integral type	unsigned char *	stvlx a, b, c
		vector unsigned char *	
vector signed char	any integral type	signed char *	
		vector signed char *	
vector bool char	any integral type	vector bool char *	
vector unsigned short	any integral type	unsigned short *	
		vector unsigned short *	
vector signed short	any integral type	signed short *	
		vector signed short *	
vector bool short	any integral type	vector bool short *	
vector pixel	any integral type	vector pixel *	
vector unsigned int	any integral type	unsigned int *	
		vector unsigned int *	
vector signed int	any integral type	signed int *	
		vector signed int *	
vector bool int	any integral type	vector bool int *	
vector float	any integral type	float *	
		vector float *	

vec_ stvlxl: Store Vector Left Indexed Last

```
(void) vec_stvlxl(a, b, c)
```

EA を *b* の内容と *c* の内容の和にて形成される実効アドレスとし、eb を EA の最下位側 4 ビットの値とします。 *a* の左端 (16 - eb) バイトを EA でアドレス指定されるメモリにストアします。vec_stvlxl は、EA でアドレス指定されるメモリのクワッドワードが近い将来プログラムで再度必要とされることはおそらくないであろうというヒントを与えます。

表 6-121: Store Vector Left Indexed Last

戻り値／引数の型			アセンブリ命令へのマップ
a	b	c	
vector unsigned char	any integral type	unsigned char *	stvlxl a, b, c
		vector unsigned char *	
vector signed char	any integral type	signed char *	
		vector signed char *	
vector bool char	any integral type	vector bool char *	
vector unsigned short	any integral type	unsigned short *	
		vector unsigned short *	
vector signed short	any integral type	signed short *	
		vector signed short *	
vector bool short	any integral type	vector bool short *	
vector pixel	any integral type	vector pixel *	
vector unsigned int	any integral type	unsigned int *	
		vector unsigned int *	
vector signed int	any integral type	signed int *	
		vector signed int *	
vector bool int	any integral type	vector bool int *	
vector float	any integral type	float *	
		vector float *	

vec_ stvrx: Store Vector Right Indexed

```
(void) vec_ stvrx(a, b, c)
```

EA を *b* の内容と *c* の内容の和にて形成される実効アドレスとし、eb を EA の最下位側 4 ビットの値とします。 *a* の右端 eb バイトを (EA - eb) でアドレス指定されるメモリにストアします。eb がゼロの場合は、EA が 16 バイトにアラインしており、メモリのストアは行われません。

表 6-122: Store Vector Right Indexed

戻り値／引数の型			アセンブリ命令へのマップ
a	b	c	
vector unsigned char	any integral type	unsigned char *	stvrx a, b, c
		vector unsigned char *	
vector signed char	any integral type	signed char *	
		vector signed char *	
vector bool char	any integral type	vector bool char *	
vector unsigned short	any integral type	unsigned short *	
		vector unsigned short *	
vector signed short	any integral type	signed short *	
		vector signed short *	
vector bool short	any integral type	vector bool short *	
vector pixel	any integral type	vector pixel *	
vector unsigned int	any integral type	unsigned int *	
		vector unsigned int *	
vector signed int	any integral type	signed int *	
		vector signed int *	
vector bool int	any integral type	vector bool int *	
vector float	any integral type	float *	
		vector float *	

vec_ stvrxl: Store Vector Right Indexed Last

```
(void) vec_ stvrxl(a, b, c)
```

EA を *b* の内容と *c* の内容の和にて形成される実効アドレスとし、eb を EA の最下位側 4 ビットの値とします。 *a* の右端 eb バイトを (EA - eb) でアドレス指定されるメモリにストアします。eb がゼロの場合は、EA が 16 バイトにアラインしており、メモリのストアは行われません。vec_stvrxl は、EA でアドレス指定されるメモリのクワッドワードが近い将来プログラムで再度必要とされることはおそらくないであろうというヒントを与えます。

表 6-123: Store Vector Right Indexed Last

戻り値／引数の型			アセンブリ命令へのマップ
a	b	c	
vector unsigned char	any integral type	unsigned char *	stvrxl a, b, c
		vector unsigned char *	
vector signed char	any integral type	signed char *	
		vector signed char *	
vector bool char	any integral type	vector bool char *	
vector unsigned short	any integral type	unsigned short *	
		vector unsigned short *	
vector signed short	any integral type	signed short *	
		vector signed short *	
vector bool short	any integral type	vector bool short *	
vector pixel	any integral type	vector pixel *	
vector unsigned int	any integral type	unsigned int *	
		vector unsigned int *	
vector signed int	any integral type	signed int *	
		vector signed int *	
vector bool int	any integral type	vector bool int *	
vector float	any integral type	float *	
		vector float *	

vec_promote: Promote Scalar to a Vector

```
d = vec_promote(a, element)
```

スカラ *a* を、*element* パラメタで指定する要素に *a* を含むベクタとして格上げし、結果をベクタ *d* に返します。ベクタ *d* のその他の要素はすべて不定です。*a* のサイズに応じて *element* の最下位ビット側の限られたビット数のみをインデックスに使用します。具体的には、1, 2, 4 バイトの要素について、最下位側のそれぞれ 4, 3, 2 ビットのみを使用します。

表 6-124: Promote Scalar to a Vector

戻り値／引数の型			アセンブリ命令へのマップ [1]
d	a	element	
vector unsigned char	unsigned char	int	EA=memaddr + (element&0xF) stbx a, 0, EA lvebx d, 0, EA
vector signed char	signed char		
vector unsigned short	unsigned short		EA=memaddr + (element&0x7)<<2 sthx a, 0, EA lvehx d, 0, EA
vector signed short	signed short		
vector unsigned int	unsigned int		EA=memaddr + (element&0x3)<<3 stwx a, 0, EA lvewx d, 0, EA
vector signed int	signed int		
vector float	float		EA=memaddr + (element&0x3)<<3 stfsx a, EA lvewx d, 0, EA

[1] memaddr は 16 バイトアラインの一時メモリロケーションのアドレスです。

vec_splats: Splat Scalar to a Vector

```
d = vec_splats(a)
```

単一のスカラ *a* の値を同じ型のベクタの全要素にわたって複製し、結果をベクタ *d* に返します。

表 6-125: Splat Scalar to a Vector

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
vector unsigned char	unsigned char	a を 16 バイトアラインしたメモリ (EA) にストア lvebx/lvehx/lvewx tmp, 0, EA vspltb/vsplth/vspltw d, tmp, 0
vector signed char	signed char	
vector unsigned short	unsigned short	
vector signed short	signed short	
vector unsigned int	unsigned int	
vector signed int	signed int	
vector float	float	
vector unsigned char	unsigned char (5-bit unsigned literal)	vspltisb d, a *or* vspltish d, a *or* vspltisw d, a *or* vspltisw d, a
vector signed char	signed char (5-bit unsigned literal)	
vector unsigned short	unsigned short (5-bit unsigned literal)	
vector signed short	signed short (5-bit unsigned literal)	
vector unsigned int	unsigned int (5-bit unsigned literal)	
vector signed int	signed int (5-bit unsigned literal)	
vector float	float (5-bit unsigned literal)	

7. PPU組み込み関数

本章では、C プログラミング言語から下層に位置する PPU 命令セットをアクセスできるようにするための、個別組み込み関数の最小セットを規定します。`__setflm` を除き、これらの組み込み関数のそれぞれは、アセンブリ言語に 1 対 1 にマッピングされています。ただし、32 ビット ABI 向けにコンパイルした場合を除きます。その場合は、64 ビットダブルワードの上下半分が別々のレジスタに保持されます。その状況では、対応する 32 ビット組み込み関数は複数命令の列を生成する場合があります。また、他のケースでは、対応する 32 ビット実装はサポートできない場合もあります。

PPU 組み込み関数はシステムヘッダファイル `ppu_intrinsics.h` で宣言されることになりますが、このヘッダ内でマクロとして定義するか、あるいはコンパイラの内部で実装するか、いずれでもかまいません。

組み込み関数の中には、長さ 3, 4, 5, 6, 8, 10 ビットのいずれかのリテラルを使用するものがあります。デフォルトでは、範囲外のリテラルを伴う組み込み関数呼び出しは、コンパイラでエラーと報告されます。範囲外リテラル値の警告を発して範囲外の引数に対して最下位ビット側の規定ビット数のみを用いるようなコンパイラオプションを設けても構いません。

特に記述がない限り、組み込み関数は特定の順序付けを有しません。組み込み関数は通常の演算と同様に、コンパイラで最適化して順序変更することができます。

__cctph: Change Thread Priority to High

```
(void) __cctph()
```

現行（ハードウェア）スレッドのプライオリティを高優先度に変更します。この組み込み関数はコンパイラが順序変更することはありません。

表 7-126: Change Thread Priority to High

戻り値／引数の型	アセンブリ命令へのマップ
none	cctph

__cctpl: Change Thread Priority to Low

```
(void) __cctpl()
```

現行スレッドのプライオリティを低優先度に変更します。この組み込み関数はコンパイラが順序変更することはありません。

表 7-127: Change Thread Priority to Low

戻り値／引数の型	アセンブリ命令へのマップ
none	cctpl

__cctpm: Change Thread Priority to Medium

```
(void) __cctpm()
```

現行スレッドのプライオリティを中優先度に変更します。この組み込み関数はコンパイラが順序変更することはありません。

表 7-128: Change Thread Priority to Medium

戻り値／引数の型	アセンブリ命令へのマップ
none	cctpm

__cntlzd: Count Leading Doubleword Zeros

```
d = __cntlzd(a)
```

ダブルワード *a* 内の先行 0 の個数を *d* に返します。

表 7-129: Count Leading Doubleword Zeros

戻り値／引数の型		アセンブリ命令へのマップ	
d	a	64-bit ABI	32-bit ABI
unsigned int	unsigned long long	cntlzd d, a	cntlzw hi_cnt, a_hi cntlzw lo_cnt, a_lo rlwinm mask, hi_cnt, 26, 0, 5 srawi mask, mask, 31 and lo_cnt, lo_cnt, mask add d, hi_cnt, lo_cnt

__cntlzw: Count Leading Word Zeros

```
d = __cntlzw(a)
```

ワード *a* 内の先行 0 の個数を *d* に返します。

表 7-130: Count Leading Word Zeros

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
unsigned int	unsigned int	cntlzw d, a

__db10cyc: Delay 10 Cycles at Dispatch

```
(void) __db10cyc()
```

現行（ハードウェア）スレッドが命令ディスパッチにおいて 10 サイクルの間ブロックされます。この組み込み関数はコンパイラが順序変更することはありません。

表 7-131: Delay 10 Cycles At Dispatch

戻り値／引数の型	アセンブリ命令へのマップ
none	db10cyc

__db12cyc: Delay 12 Cycles at Dispatch

```
(void) __db12cyc()
```

現行スレッドが命令ディスパッチにおいて 12 サイクルの間ブロックされます。この組み込み関数はコンパイラが順序変更することはありません。

表 7-132: Delay 12 Cycles At Dispatch

戻り値／引数の型	アセンブリ命令へのマップ
none	db12cyc

__db16cyc: Delay 16 Cycles at Dispatch

```
(void) __db16cyc()
```

現行スレッドが命令ディスパッチにおいて 16 サイクルの間ブロックされます。この組み込み関数はコンパイラが順序変更することはありません。

表 7-133: Delay 16 Cycles At Dispatch

戻り値／引数の型	アセンブリ命令へのマップ
none	db16cyc

__db8cyc: Delay 8 Cycles at Dispatch

```
(void) __db8cyc()
```

現行スレッドが命令ディスパッチにおいて 8 サイクルの間ブロックされます。この組み込み関数はコンパイラが順序変更することはありません。

表 7-134: Delay 8 Cycles At Dispatch

戻り値／引数の型	アセンブリ命令へのマップ
none	db8cyc

__dcbf: Data Cache Block Flush

```
(void) __dcbf(pointer)
```

引数 *pointer* を含むキャッシュブロックをフラッシュしてキャッシュから取り除きます。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-135: Data Cache Block Flush

戻り値／引数の型 pointer	アセンブリ命令へのマップ
void*	dcbf base, index

__dcbst: Data Cache Block Store

```
(void) __dcbst(pointer)
```

引数 *pointer* を含むキャッシュブロックをメインメモリに書き込みます。この組み込み関数はコンパイラが順序変更することはありません。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-136: Data Cache Block Store

戻り値／引数の型 pointer	アセンブリ命令へのマップ
void*	dcbst base, index

__dcbt: Data Cache Block Touch

```
(void) __dcbt(pointer)
```

引数の *pointer* を含むキャッシュブロックに対してまもなくロードが行なわれるというヒントをプロセッサに与えます。この組み込み関数はコンパイラが順序変更することはありません。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-137: Data Cache Block Touch

戻り値／引数の型	アセンブリ命令へのマップ
pointer	
void*	dcbt base, index

__dcbt_TH1000: Start Streaming Data

```
(void) __dcbt_TH1000(EATRUNC, D, UG, ID)
```

識別子 *ID* と実効アドレス EATRUNC でもってストリームを開始します。引数 *D* はストリームの進行する方向を表わし、true が前進方向、false が後退方向です。引数 *UG* はストリームが範囲無制限か否かを示します。この組み込み関数はコンパイラが順序変更することはありません。

この組み込み関数用の実効アドレスは以下のようにして計算されます。

```
((unsigned long long) EATRUNC) & ~0x7F) | (((D & 1) << 6) | ((UG & 1) << 5) | (ID & 0xF)
```

アセンブリ命令の引数 *base* と *index* は上記の実効アドレスから算出されます。

表 7-138: Start Streaming Data

戻り値／引数の型				アセンブリ命令へのマップ
EATRUNC	D	UG	ID	
void*	bool	bool	int	dcbt base, index, 8

__dcbt_TH1010: Stop Streaming Data

```
(void) __dcbt_TH1010(G0, S, UNITCNT, T, U, ID)
```

ID で識別されるストリームがもはや必要ではなくなるというヒントをプロセッサに与えます。*G0* がセットされている場合は、プログラムは、完全に記述し終わった初期データストリームのすべてからまもなくロードを行なうことになり、それ以外の初期データストリームからはロードはおそらく行なわないことになります。このケースではその他の引数はすべて無視されます。*S* が ‘10’ ならば、*ID* に対応するストリームが停止することになり、*ID* 以外の引数はすべて無視されます。*S* が ‘11’ ならば、すべてのストリーム ID が停止され、その他の引数はすべて無視されます。*UNITCNT* はデータストリーム中のユニット数を指定します。*T* はデータストリーム中の各ブロックのプログラムでの必要性が一過性である可能性が高いか否かを示します。*U* はデータストリームが無制限であって *UNITCNT* 引数が無視されるのか否かを示します。この組み込み関数はコンパイラが順序変更することはありません。

この組み込み関数用の実効アドレスは以下のようにして計算されます。

```
(((unsigned long long) G0 & 1) << 31)
  | ((S & 0x3) << 29)
  | ((UNITCNT & 0x3FF) << 7)
  | ((T & 1) << 6)
  | ((U & 1) << 5)
  | (ID & 0xF)
```

アセンブリ命令の引数 *base* と *index* は上記の実効アドレスから算出されます。

表 7-139: Stop Streaming Data

戻り値／引数の型						アセンブリ命令へのマップ
G0	S	UNITCNT	T	U	ID	
bool	int	int	bool	bool	int	dcbt base, index, 10

__dcbtst: Data Cache Block Touch for Store

```
(void) __dcbtst(pointer)
```

引数の *pointer* を含むキャッシュブロックに対してまもなくストアが行なわれるというヒントをプロセッサに与えます。この組み込み関数はコンパイラが順序変更することはありません。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-140: Data Cache Block Touch For Store

戻り値／引数の型	アセンブリ命令へのマップ
pointer	
void*	dcbtst base, index

__dcbz: Data Cache Block Set to Zero

```
(void) __dcbz(pointer)
```

引数の *pointer* を含むキャッシュブロックをゼロクリアします。当該アドレスが既にキャッシュにあれば、それを含むキャッシュブロックがゼロクリアされます。当該アドレスを含むキャッシュブロックが無かったら、オールゼロのキャッシュブロックが作成されます。この組み込み関数はコンパイラが順序変更することはありません。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-141: Data Cache Block Set to Zero

戻り値／引数の型	アセンブリ命令へのマップ
pointer	
void*	dcbz base, index

__eieio: Enforce In-Order Execution of I/O

```
(void) __eieio()
```

メモリバリアを作成し、それによって、この `__eieio()` 命令を実行しているプロセッサが実行するロード、ストア、`__dcbz()`、`__eciwx()`、および `__ecowx()` 命令に起因するストレージアクセスに対する順序付け機能を提供します。メモリバリアと順序付け機能については、*PowerPC Architecture Book, Book II: PowerPC Virtual Environment Architecture, Version 2.02* のSection 1.7.1 に説明があります。

表 7-142: Enforce In-Order Execution of I/O

戻り値／引数の型	アセンブリ命令へのマップ
none	eieio

__fabs: Double Absolute Value

```
d = __fabs(a)
```

引数 *a* の絶対値を *d* に符号ビットをゼロにして返します。

表 7-143: Double Absolute Value

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
double	double	fabs d, a

__fabsf: Float Absolute Value

```
d = __fabsf(a)
```

引数 *a* の絶対値を *d* に符号ビットをゼロにして返します。

表 7-144: Float Absolute Value

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
float	float	fabs d, a

__fcfid: Convert Doubleword to Double

```
d = __fcfid(a)
```

a のダブルワードを浮動小数に変換して *d* に返します。

表 7-145: Convert Doubleword to Double

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
double	long long	fcfid d, a

__fctid: Convert Double to Doubleword

```
d = __fctid(a)
```

`double` *a* をダブルワード整数に変換して *d* に返します。この関数では現行の丸めモードを考慮します。

表 7-146: Convert Double to Doubleword

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
long long	double	fctid d, a

__fctidz: Convert Double to Doubleword with Round Towards Zero

```
d = __fctidz(a)
```

`double` *a* をダブルワード整数に変換して *d* に返します。この関数では常にゼロ方向への丸めを行ないます。

表 7-147: Convert Double to Doubleword with Round Towards Zero

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
long long	double	fctidz d, a

__fctiw: Convert Double to Word

```
d = __fctiw(a)
```

double *a* をワード整数に変換して *d* に返します。この関数では現行の丸めモードを考慮します。

表 7-148: Convert Double to Word

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
int	double	fctiw tmp, a stfiwx tmp, r1, tempspace lwzx d, r1, tempspace

__fctiwz: Convert Double to Word with Round Towards Zero

```
d = __fctiwz(a)
```

double *a* をワード整数に変換して *d* に返します。この関数では常にゼロ方向への丸めを行ないます。

表 7-149: Convert Double to Word with Round Towards Zero

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
int	double	fctiwz tmp, a stfiwx tmp, r1, tempspace lwzx d, r1, tempspace

__fmadd: Double Fused Multiply and Add

```
d = __fmadd(a, b, c)
```

引数 *a* に引数 *b* を乗算した積に引数 *c* を加算します。結果の値 (*a*×*b*+*c*) を *d* に返します。

表 7-150: Double Fused Multiply and Add

戻り値／引数の型				アセンブリ命令へのマップ
d	a	b	c	
double	double	double	double	fmadd d, a, b, c

__fmadds: Float Fused Multiply and Add

```
d = __fmadds(a, b, c)
```

引数 *a* に引数 *b* を乗算した積に引数 *c* を加算します。結果の値 (*a*×*b*+*c*) を *d* に返します。

表 7-151: Float Fused Multiply and Add

戻り値／引数の型				アセンブリ命令へのマップ
d	a	b	c	
float	float	float	float	fmadds d, a, b, c

__fmsub: Double Fused Multiply and Subtract

```
d = __fmsub(a, b, c)
```

引数 *a* に引数 *b* を乗算した積から引数 *c* を減算します。結果の値 (*a*×*b*-*c*) を *d* に返します。

表 7-152: Double Fused Multiply and Subtract

戻り値／引数の型				アセンブリ命令へのマップ
d	a	b	c	
double	double	double	double	fmsub d, a, b, c

__fmsubs: Float Fused Multiply and Subtract

```
d = __fmsubs(a, b, c)
```

引数 *a* に引数 *b* を乗算した積から引数 *c* を減算します。結果の値 (*a*×*b*-*c*) を *d* に返します。

表 7-153: Float Fused Multiply and Subtract

戻り値／引数の型				アセンブリ命令へのマップ
d	a	b	c	
float	float	float	float	fmsubs d, a, b, c

__fmul: Double Multiply

```
d = __fmul(a, b)
```

倍精度の *a* と *b* を乗算して積の (*a*×*b*) を *d* に返します。

表 7-154: Double Multiply

戻り値／引数の型			アセンブリ命令へのマップ
d	a	b	
double	double	double	fmul d, a, b

__fmuls: Float Multiply

```
d = __fmuls(a, b)
```

単精度の *a* と *b* を乗算して積の (*a*×*b*) を *d* に返します。

表 7-155: Float Multiply

戻り値／引数の型			アセンブリ命令へのマップ
d	a	b	
float	float	float	fmuls d, a, b

__fnabs: Double Negative

```
d = __fnabs(a)
```

引数 *a* の負の絶対値を *d*.に返します。符号ビットは 1 にセットされます。

表 7-156: Double Negative

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
double	double	fnabs d, a

__fnabsf: Float Negative

```
d = __fnabsf(a)
```

引数 *a* の負の絶対値を *d*.に返します。符号ビットは 1 にセットされます。

表 7-157: Float Negative

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
float	float	fnabs d, a

__fnmadd: Double Fused Negative Multiply and Add

```
d = __fnmadd(a, b, c)
```

引数 *a* と *b* を乗算した積に引数 *c* を加算します。その和の符号を反転した結果の値 -(*a*×*b*+*c*) を *d* に返します。

表 7-158: Double Fused Negative Multiply and Add

戻り値／引数の型				アセンブリ命令へのマップ
d	a	b	c	
double	double	double	double	fnmadd d, a, b, c

__fnmadds: Float Fused Negative Multiply and Add

```
d = __fnmadds(a, b, c)
```

引数 *a* と *b* を乗算した積に引数 *c* を加算します。その和の符号を反転した結果の値 -(*a*×*b*+*c*) を *d* に返します。

表 7-159: Float Fused Negative Multiply and Add

戻り値／引数の型				アセンブリ命令へのマップ
d	a	b	c	
float	float	float	float	fnmadds d, a, b, c

__fnmsub: Double Fused Negative Multiply and Subtract

```
d = __fnmsub(a, b, c)
```

引数 *a* と *b* を乗算した積から引数 *c* を減算します。その差の符号を反転した結果の値 -(*a*×*b*-*c*) を *d* に返します。

表 7-160: Double Fused Negative Multiply and Subtract

戻り値／引数の型				アセンブリ命令へのマップ
d	a	b	c	
double	double	double	double	fnmsub d, a, b, c

__fnmsubs: Float Fused Negative Multiply and Subtract

```
d = __fnmsubs(a, b, c)
```

引数 *a* と *b* を乗算した積から引数 *c* を減算します。その差の符号を反転した結果の値 -(*a*×*b*-*c*) を *d* に返します。

表 7-161: Float Fused Negative Multiply and Subtract

戻り値／引数の型				アセンブリ命令へのマップ
d	a	b	c	
float	float	float	float	fnmsubs d, a, b, c

__fres: Float Reciprocal Estimate

```
d = __fres(a)
```

引数 *a* の逆数の推定値を *d* に返します。推定値は逆数に対して 256 分の 1 までの精度を有しています。

この精度を超えた部分の値は不定であり、本命令の実行結果は、実装間や、同一の実装においても別々の実行の間では異なる場合があります。

表 7-162: Float Reciprocal Estimate

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
float	float	fres d, a

__frsp: Round to Single Precision

```
d = __frsp(a)
```

引数 *a* を単精度に丸めて *d* に返します。

表 7-163: Round to Single Precision

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
float	double	frsp d, a

__frsqrte: Double Reciprocal Square Root Estimate

```
d = __frsqrte(a)
```

引数 *a* の平方根の逆数の推定値を *d* に返します。

推定値は平方根の逆数に対して 32 分の 1 までの精度を有しています。この精度を超えた部分の値は不定であり、本命令の実行結果は、実装間や、同一の実装においても別々の実行の間では異なる場合があります。

表 7-164: Double Reciprocal Square Root Estimate

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
double	double	frsqrte d, a

__fsel: Floating-Point Select of Double

```
d = __fsel(a, b, c)
```

引数 *a* が *0.0* 以上ならば、引数 *b* を *d* に返します。さもなければ、引数 *c* を返します。

表 7-165: Floating-Point Select of Double

戻り値／引数の型				アセンブリ命令へのマップ
d	a	b	c	
double	double	double	double	fsel d, a, b, c

__fsels: Floating-Point Select of Float

```
d = __fsels(a, b, c)
```

引数 *a* が *0.0* 以上ならば、引数 *b* を *d* に返します。さもなければ、引数 *c* を返します。

表 7-166: Floating-Point Select of Float

戻り値／引数の型				アセンブリ命令へのマップ
d	a	b	c	
float	float	float	float	fsel d, a, b, c

__fsqrt: Double Square Root

```
d = __fsqrt(a)
```

引数 *a* の平方根を *d* に返します。

表 7-167: Double Square Root

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
double	double	fsqrt d, a

__fsqrts: Float Square Root

```
d = __fsqrts(a)
```

引数 *a* の平方根を *d* に返します。

表 7-168: Float Square Root

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
float	float	fsqrts d, a

__icbi: Instruction Cache Block Invalidate

```
(void) __icbi(pointer)
```

引数 *pointer* を含む命令キャッシュブロックが命令キャッシュにあれば、それを無効化します。この組み込み関数はコンパイラが順序変更することはありません。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-169: Instruction Cache Block Invalidate

戻り値／引数の型	アセンブリ命令へのマップ
pointer	
void*	icbi base, index

__isync: Instruction Sync

```
(void) __isync()
```

先行するすべての命令が完了するまでプロセッサを待たせます。`__isync()` 関数はすべての `icbi` が遂行されたことを保証します。

表 7-170: Instruction Sync

戻り値／引数の型	アセンブリ命令へのマップ
none	isync

__ldarx: Load Doubleword with Reserved

```
d = __ldarx(pointer)
```

プロセッサのリザベーションアドレスを *pointer* の値に設定します。*pointer* の示すアドレスからのダブルワードを *d* に返します。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

この組み込み関数は 32 ビット ABI 用のコンパイルではサポートされないかもしれません。64 ビットのダブルワードを 2 個の別々のレジスタに保持するからです。

表 7-171: Load Doubleword with Reserved

戻り値／引数の型		アセンブリ命令へのマップ
d	pointer	
unsigned long long	void*	ldarx d, base, index

__ldbrx: Load Reversed Doubleword

```
d = __ldbrx(pointer)
```

pointer の示すアドレスからのダブルワードを逆のエンディアン順序で *d* にロードして返します。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-172: Load Reversed Doubleword

戻り値／引数の型		アセンブリ命令へのマップ	
d	pointer	64-bit ABI	32-bit ABI
unsigned long long	void*	ldbrx d, base, index	lwbrx d_lo, base, index lwbrx d_hi, base, index+4

__lhbrx: Load Reversed Halfword

```
d = __lhbrx(pointer)
```

pointer の示すアドレスからのハーフワードを逆のエンディアン順序で *d* にロードして返します。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-173: Load Reversed Halfword

戻り値／引数の型		アセンブリ命令へのマップ
d	pointer	
unsigned short	void*	lhbrx d, base, index

__lwarx: Load Word with Reserved

```
d = __lwarx(pointer)
```

プロセッサのリザベーションアドレスを *pointer* の値に設定します。*pointer* の示すアドレスからのワードを *d* に返します。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-174: Load Word with Reserved

戻り値／引数の型		アセンブリ命令へのマップ
d	pointer	
unsigned	void*	lwarx d, base, index

__lwbrx: Load Reversed Word

```
d = __lwbrx(pointer)
```

pointer の示すアドレスからのワードを逆のエンディアン順序で *d* にロードして返します。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-175: Load Reversed Word

戻り値／引数の型		アセンブリ命令へのマップ
d	pointer	
unsigned	void*	lwbrx d, base, index

__lwsync: Light Weight Sync

```
(void) __lwsync()
```

メモリバリアを作成し、それによって、この `__lwsync()` 命令を実行しているプロセッサが実行する先行ロード、ストア、および `__dcbz()` 命令に起因するストレージアクセスに対する順序付け機能を提供します。メモリバリアと順序付け機能については、*PowerPC Architecture Book, Book II: PowerPC Virtual Environment Architecture, Version 2.02* のSection 1.7.1 に説明があります。

表 7-176: Light Weight Sync

戻り値／引数の型	アセンブリ命令へのマップ
none	lwsync

__mffs: Move from Floating-Point Status and Control Register

```
d = __mffs()
```

現在の浮動小数点ステータス/制御レジスタを *d* に返します。この組み込み関数はコンパイラが順序変更することはありません。

表 7-177: Move from Floating-Point Status and Control Register

戻り値／引数の型	アセンブリ命令へのマップ
d	
double	mffs d

__mfspr: Move from Special Purpose Register

```
d = __mfspr(spr)
```

spr で指定されるスペシャルパーパスレジスタの内容を *d* に返します。この組み込み関数はコンパイラが順序変更することはありません。

この組み込み関数は 32 ビット ABI 用のコンパイルではサポートされないかもしれません。64 ビットのダブルワードを 2 個の別々のレジスタに保持するからです。

表 7-178: Move from Special Purpose Register

戻り値／引数の型		アセンブリ命令へのマップ
d	spr	
unsigned long long	10-bit literal unsigned int	mfspr d, spr

__mftb: Move from Time Base

```
d = __mftb()
```

タイムベースレジスタを *d* に返します。この組み込み関数はコンパイラが順序変更することはありません。

表 7-179: Move from Time Base

戻り値／引数の型	アセンブリ命令へのマップ	
d	64-bit ABI	32-bit ABI
unsigned long long	mftb d	retry: mftbu d_hi mftb d_lo mftbu tmp cmp d_hi, tmp bne retry

__mtfsb0: Reset Bit of FPSCR

```
(void) __mtfsb0(bt)
```

浮動小数点ステータス/制御レジスタ (FPSCR) のビット *bt* を 0 にセットします。この組み込み関数はコンパイラが順序変更することはありません。さらに、浮動小数点演算に対するバリアも生じることになります。

表 7-180: Reset Bit of FPSCR

戻り値／引数の型	アセンブリ命令へのマップ
bt	
5-bit unsigned int (literal)	mtfsb0 bt

__mtfsb1: Set Bit of FPSCR

```
(void) __mtfsb1(bt)
```

浮動小数点ステータス/制御レジスタ (FPSCR) のビット *bt* を 1 にセットします。この組み込み関数はコンパイラが順序変更することはありません。さらに、浮動小数点演算に対するバリアも生じることになります。

表 7-181: Set Bit of FPSCR

戻り値／引数の型	アセンブリ命令へのマップ
bt	
5-bit unsigned int (literal)	mtfsb1 bt

__mtfsf: Set Fields in FPSCR

```
(void) __mtfsf(flm, b)
```

浮動小数点ステータス/制御レジスタを、各フィールドを引数 *flm* でマスクしながら、*b* の値で設定します。この組み込み関数はコンパイラが順序変更することはありません。さらに、浮動小数点演算に対するバリアも生じることになります。

表 7-182: Set Fields in FPSCR

戻り値／引数の型		アセンブリ命令へのマップ
flm	b	
8-bit unsigned int (literal)	double	mtfsf flm, b

__mtfsfi: Set Field of FPSCR

```
(void) __mtfsfi(bf, u)
```

引数 *u* の値を浮動小数点ステータス/制御レジスタの *bf* フィールドにセットします。この組み込み関数はコンパイラが順序変更することはありません。さらに、浮動小数点演算に対するバリアも生じることになります。

表 7-183: Set Field of FPSCR

戻り値／引数の型		アセンブリ命令へのマップ
bf	u	
3-bit unsigned int (literal)	4-bit unsigned int (literal)	mtfsfi bf, u

__mtspr: Move to Special Purpose Register

```
(void) __mtspr(spr, value)
```

spr で指定されるスペシャルパーパスレジスタに引数 *value* をセットします。この組み込み関数はコンパイラが順序変更することはありません。

この組み込み関数は 32 ビット ABI 用のコンパイルではサポートされないかもしれません。64 ビットのダブルワードを 2 個の別々のレジスタに保持するからです。

表 7-184: Move to Special Purpose Register

戻り値／引数の型		アセンブリ命令へのマップ
spr	value	
10-bit unsigned int (literal)	unsigned long long	mtspr spr, value

__mulhd: Multiply Doubleword, High Part

```
d = __mulhd(a, b)
```

ダブルワードの引数 *a* と *b* の符号付き積の上位部分を *d* に返します。

この組み込み関数は 32 ビット ABI 用のコンパイルではサポートされないかもしれません。64 ビットのダブルワードを 2 個の別々のレジスタに保持するからです。

表 7-185: Multiply Doubleword, High Part

戻り値／引数の型			アセンブリ命令へのマップ
d	a	b	
long long	long long	long long	mulhd d, a, b

__mulhdu: Multiply Double Unsigned Word, High Part

```
d = __mulhdu(a, b)
```

ダブルワードの引数 *a* と *b* の符号無し積の上位部分を *d* に返します。

この組み込み関数は 32 ビット ABI 用のコンパイルではサポートされないかもしれません。64 ビットのダブルワードを 2 個の別々のレジスタに保持するからです。

表 7-186: Multiply Double Unsigned Word, High Part

戻り値／引数の型			アセンブリ命令へのマップ
d	a	b	
unsigned long long	unsigned long long	unsigned long long	mulhdu d, a, b

__mulhw: Multiply Word, High Part

```
d = __mulhw(a, b)
```

ワードの引数 *a* と *b* の符号付き積の上位部分を *d* に返します。

表 7-187: Multiply Word, High Part

戻り値／引数の型			アセンブリ命令へのマップ
d	a	b	
int	int	int	mulhw d, a, b

__mulhwu: Multiply Unsigned Word, High Part

```
d = __mulhwu(a, b)
```

ワードの引数 *a* と *b* の符号無し積の上位部分を *d* に返します。

表 7-188: Multiply Unsigned Word, High Part

戻り値／引数の型			アセンブリ命令へのマップ
d	a	b	
unsigned int	unsigned int	unsigned int	mulhwu d, a, b

__nop: No Operation

```
(void) __nop()
```

好適な nop 命令を生成します。この組み込み関数はコンパイラが順序変更することはありません。

表 7-189: No Operation

戻り値／引数の型	アセンブリ命令へのマップ
none	nop

__rldcl: Rotate Left Doubleword then Clear Left

```
d = __rldcl(a, b, mb)
```

引数 *a* の値を引数 *b* で指定するビット数だけ左方向にローテートします。ビット *mb* からビット 63 までが‘1’でその他が‘0’のマスクを生成します。ローテートしたデータを生成したマスクで AND 演算した結果を *d* に返します。

この組み込み関数は 32 ビット ABI 用のコンパイルではサポートされないかもしれません。64 ビットのダブルワードを 2 個の別々のレジスタに保持するからです。

表 7-190: Rotate Left Doubleword then Clear Left

戻り値／引数の型				アセンブリ命令へのマップ
d	a	b	mb	
unsigned long long	unsigned long long	unsigned long long	6-bit unsigned int (literal)	rldcl d, a, b, mb

__rldcr: Rotate Left Doubleword then Clear Right

```
d = __rldcr(a, b, me)
```

引数 *a* の値を引数 *b* で指定するビット数だけ左方向にローテートします。ビット 0 からビット *me* までが‘1’でその他が‘0’のマスクを生成します。ローテートしたデータを生成したマスクで AND 演算した結果を *d* に返します。

この組み込み関数は 32 ビット ABI 用のコンパイルではサポートされないかもしれません。64 ビットのダブルワードを 2 個の別々のレジスタに保持するからです。

表 7-191: Rotate Left Doubleword then Clear Right

戻り値／引数の型				アセンブリ命令へのマップ
d	a	b	me	
unsigned long long	unsigned long long	unsigned long long	6-bit unsigned int (literal)	rldcr d, a, b, me

__rldic: Rotate Left Doubleword Immediate then Clear

```
d = __rldic(a, sh, mb)
```

引数 *a* の値を引数 *sh* で指定するビット数だけ左方向にローテートします。ビット *mb* からビット *63-sh* までが‘1’でその他が‘0’のマスクを生成します。ローテートしたデータを生成したマスクで AND 演算した結果を *d* に返します。

この組み込み関数は 32 ビット ABI 用のコンパイルではサポートされないかもしれません。64 ビットのダブルワードを 2 個の別々のレジスタに保持するからです。

表 7-192: Rotate Left Doubleword Immediate then Clear

戻り値／引数の型				アセンブリ命令へのマップ
d	a	sh	mb	
unsigned long long	unsigned long long	6-bit unsigned int (literal)	6-bit unsigned int (literal)	rldic d, a, sh, mb

__rldicl: Rotate Left Doubleword Immediate then Clear Left

```
d = __rldicl(a, sh, mb)
```

引数 *a* の値を引数 *sh* で指定するビット数だけ左方向にローテートします。ビット *mb* からビット 63 までが‘1’でその他が‘0’のマスクを生成します。ローテートしたデータを生成したマスクで AND 演算した結果を *d* に返します。

この組み込み関数は 32 ビット ABI 用のコンパイルではサポートされないかもしれません。64 ビットのダブルワードを 2 個の別々のレジスタに保持するからです。

表 7-193: Rotate Left Doubleword Immediate then Clear Left

戻り値／引数の型				アセンブリ命令へのマップ
d	a	sh	mb	
unsigned long long	unsigned long long	6-bit unsigned int (literal)	6-bit unsigned int (literal)	rldicl d, a, sh, mb

__rldicr: Rotate Left Doubleword Immediate then Clear Right

```
d = __rldicr(a, sh, me)
```

引数 *a* の値を引数 *sh* で指定するビット数だけ左方向にローテートします。ビット 0 からビット *me* までが‘1’でその他が‘0’のマスクを生成します。ローテートしたデータを生成したマスクで AND 演算した結果を *d* に返します。

この組み込み関数は 32 ビット ABI 用のコンパイルではサポートされないかもしれません。64 ビットのダブルワードを 2 個の別々のレジスタに保持するからです。

表 7-194: Rotate Left Doubleword Immediate then Clear Right

戻り値／引数の型				アセンブリ命令へのマップ
d	a	sh	me	
unsigned long long	unsigned long long	6-bit unsigned int (literal)	6-bit unsigned int (literal)	rldicr d, a, sh, me

__rldimi: Rotate Left Doubleword Immediate then Mask Insert

```
d = __rldimi(a, b, sh, mb)
```

ビット *mb* からビット 63-*sh* までが‘1’でその他が‘0’のマスクを生成します。*a* の値にこのマスクの補数を AND 演算して、*mb* から 63-*sh* までの範囲内のビットをゼロクリアします。引数 *b* を *sh* ビットだけ左にローテートした結果に前記マスクを AND 演算して、*mb* から 63-*sh* までの範囲の外のビットをゼロクリアします。2 つのマスクされた値を OR 演算して結合し *d* に返します。

この組み込み関数は 32 ビット ABI 用のコンパイルではサポートされないかもしれません。64 ビットのダブルワードを 2 個の別々のレジスタに保持するからです。

表 7-195: Rotate Left Doubleword Immediate then Mask Insert

戻り値／引数の型					アセンブリ命令へのマップ
d	a	b	sh	mb	
unsigned long long	unsigned long long	unsigned long long	6-bit unsigned int (literal)	6-bit unsigned int (literal)	mr d, a rldimi d, b, sh, mb

__rlwimi: Rotate Left Word Immediate then Mask Insert

```
d = __rlwimi(a, b, sh, mb, me)
```

ビット *mb* からビット *me* までが‘1’でその他が‘0’のマスクを生成します。*a* の値にこのマスクの補数を AND 演算して、*mb* から *me* までの範囲内のビットをゼロクリアします。引数 *b* を *sh* ビットだけ左にローテートした結果に前記マスクを AND 演算して、*mb* から *me* までの範囲の外のビットをゼロクリアします。2 つのマスクされた値を OR 演算して結合し *d* に返します。

表 7-196: Rotate Left Word Immediate then Mask Insert

戻り値／引数の型						アセンブリ命令へのマップ
d	a	b	sh	mb	me	
unsigned int	unsigned int	unsigned int	5-bit unsigned int (literal)	5-bit unsigned int (literal)	5-bit unsigned int (literal)	mr d, a rlwimi d, b, sh, mb, me

__rlwinm: Rotate Left Word Immediate then AND With Mask

```
d = __rlwinm(a, sh, mb, me)
```

ビット *mb* からビット *me* までが‘1’でその他が‘0’のマスクを生成します。*a* の値を *sh* ビットだけ左にローテートしてから、このマスクと AND 演算し、*d* に返します。

表 7-197: Rotate Left Word Immediate then AND With Mask

戻り値／引数の型					アセンブリ命令へのマップ
d	a	sh	mb	me	
unsigned int	unsigned int	5-bit unsigned int (literal)	5-bit unsigned int (literal)	5-bit unsigned int (literal)	rlwinm d, a, sh, mb, me

__rlwnm: Rotate Left Word then AND With Mask

```
d = __rlwnm(a, b, mb, me)
```

引数 *a* を引数 *b* で指定するだけ左方向にローテートします。ビット *mb* からビット *me* までが‘1’でその他が‘0’のマスクを生成します。ローテートしたデータに生成したマスクを AND 演算した結果を *d* に返します。

表 7-198: Rotate Left Word then AND With Mask

戻り値／引数の型					アセンブリ命令へのマップ
d	a	b	mb	me	
unsigned int	unsigned int	unsigned int	5-bit unsigned int (literal)	5-bit unsigned int (literal)	rlwnm d, a, b, mb, me

__setflm: Save and Set the FPSCR

```
d = __setflm(a)
```

浮動小数点ステータス/制御レジスタに *a* を設定するとともに、そのレジスタのコンテキスト（元の値）を *d* に返します。この組み込み関数はコンパイラが順序変更することはありません。さらに、浮動小数点演算に対するバリアも生じることになります。

表 7-199: Save and Set the FPSCR

戻り値／引数の型		アセンブリ命令へのマップ
d	a	
double	double	mffs d; mtfsf 0xFF, a

__stdbrx: Store Reversed Doubleword

```
(void) __stdbrx(pointer, b)
```

引数 *b* を、逆のエンディアン順序で、引数 *pointer* が示すロケーションのダブルワードにストアします。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-200: Store Reversed Doubleword

戻り値／引数の型		アセンブリ命令へのマップ	
pointer	b	64-bit ABI	32-bit ABI
void*	unsigned long long	stdbrx b, base, index	stwbrx b_lo, base, index stwbrx b_hi, base, index+4

__stdcx: Store Doubleword Conditional

```
d = __stdcx(pointer, b)
```

プロセッサのリザベーションアドレスが引数 *pointer* の値に等しければ、*b* を引数 *pointer* が示すダブルワードにストアし、*d* に値 1 を返します。さもなければ、ストアは遂行せず、*d* に値 0 を返します。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

この `stdcx.`命令は、値を `cr0.eq` の中、すなわち、コンディションレジスタ 0 の `equals` フィールド中に返します。

この組み込み関数は 32 ビット ABI 用のコンパイルではサポートされないかもしれません。64 ビットのダブルワードを 2 個の別々のレジスタに保持するからです。

表 7-201: Store Doubleword Conditional

戻り値／引数の型			アセンブリ命令へのマップ
d	pointer	b	
bool	void*	unsigned long long	stdcx. b, base, index; d = cr0.eq

__sthbrx: Store Reversed Halfword

```
(void) __sthbrx(pointer, b)
```

引数 *b* を、逆のエンディアン順序で、引数 *pointer* が示すロケーションのハーフワードにストアします。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-202: Store Reversed Halfword

戻り値／引数の型		アセンブリ命令へのマップ
pointer	b	
void*	unsigned short	sthbrx b, base, index

__stwbrx: Store Reversed Word

```
(void) __stwbrx(pointer, b)
```

引数 *b* を、逆のエンディアン順序で、引数 *pointer* が示すロケーションのワードにストアします。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

表 7-203: Store Reversed Word

戻り値／引数の型		アセンブリ命令へのマップ
pointer	b	
void*	unsigned	stwbrx b, base, index

__stwcx: Store Word Conditional

```
d = __stwcx(pointer, b)
```

プロセッサのリザベーションアドレスが引数 *pointer* の値に等しければ、*b* を引数 *pointer* が示すワードにストアし、*d* に値 1 を返します。さもなければ、ストアは遂行せず、*d* に値 0 を返します。

アセンブリ命令の引数 *base* と *index* は *pointer* から算出されます。

この `stwcx.`命令は、値を `cr0.eq` の中、すなわち、コンディションレジスタ 0 の `equals` フィールド中に返します。

表 7-204: Store Word Conditional

戻り値／引数の型			アセンブリ命令へのマップ
d	pointer	b	
bool	void*	unsigned	stwcx. b, base, index; d = cr0.eq

__sync: Sync

```
(void) __sync()
```

メモリバリアを作成し、それによって、同一プロセッサで実行しているすべての命令に対する順序付け機能を提供します。メモリバリアと順序付け機能については、*PowerPC Architecture Book, Book II: PowerPC Virtual Environment Architecture, Version 2.02* のSection 1.7.1 に説明があります。

表 7-205: Sync

戻り値／引数の型	アセンブリ命令へのマップ
none	sync

8. SPU C/C++標準ライブラリおよび言語サポート

本章では、SPU 上の C/C++標準ライブラリの実装と対応する ISO/IEC 規格との差異を述べます。また、共通の言語機能のうち特に SPU 上でサポートされていないものを特定します。

8.1. 標準ライブラリ

SPU に必要な C/C++標準ライブラリは、それぞれ ISO/IEC Standard 9899:1999 で規定している C 標準ライブラリおよび ISO/IEC Standard 14882:1998 で規定している C++標準ライブラリを基に構成されます。しかしながらいずれのライブラリも各々のベースである ISO/IEC 規格に完全に準拠した実装である必要はありません。

ISO/IEC 規格から部分的に逸脱した実装を提案するのは次に述べる二つの理由があります。第一に SPU はほとんどのスタンドアローンのプロセッサが対応しているシステムリソースおよびオペレーティングシステムへのサポートを提供していないことです。二つ目の理由は、SPU のハードウェアは IEEE の浮動小数点演算規格を完全にサポートしているわけではないことです。SPU のオペレーティングシステムへのサポートは限られているため、システムコール、スレッドファシリティ、ファイルの入出力を必要とするライブラリ関数はサポートされない可能性があります。浮動小数点演算での挙動における違いにより、単精度の浮動小数点関数の実行結果はおそらく IEEE で規定されている精度を満たすことはできず、浮動小数点例外の信頼性も劣るでしょう。しかしながら、SPU 用に提供する標準ライブラリ関数はほとんどの場合において高速に実行されることが見込まれます。

以下のセクションでは SPU 用に提供する C/C++ライブラリにおいて最低限必要な機能について述べます。

8.1.1. C標準ライブラリ

本セクションでは、C 標準ライブラリの実装が本仕様に準拠するために最低限満たすべき要件について述べます。

ライブラリ内容

C標準ライブラリに必要な要素は全て 表 8-206に示すヘッダファイル内で宣言・定義されなければなりません。ヘッダファイルの内容でISO標準ライブラリの内容と異なる点は、表中に記載しています。個々の要素の詳細な記述については「関連ドキュメント」のセクションに記載されているISO/IEC規格を参照してください。

表 8-206: C ライブラリのヘッダファイル

ファイル名	説明
assert.h	関数の実行時にアサーションを行なう関数を記述しています。`assert` マクロはデバッグ専用の `printf`（詳細は後述）を用いてアサーション失敗をレポートします。
complex.h	複素数演算を行なう関数を記述しています。
ctype.h	文字を分類する関数を記述しています。このヘッダファイルで定義された関数は“C”ロケールのみを使用します。
errno.h	ライブラリ関数によりレポートされたエラーコードを検査する関数を記述しています。
fenv.h	IEEE形式の浮動小数点演算を制御する関数を記述しています。9.2.2. 浮動小数点例外に単精度および倍精度例外用のマクロの説明を記載しています。
float.h	浮動小数点型の属性を検査するための定義を記述しています。これらの属性については“9.1. 浮動小数点型表示の属性 ”で述べます。
inttypes.h	さまざまな整数型を変換する関数を記述しています。
iso646.h	ISO 646 の変種のキャラクタセットを使用したプログラミングのための関数を記述しています。
limits.h	整数型の属性を検査するための関数を記述しています。マクロ `MB_LEN_MAX` は `1` として定義されています。
locale.h	提供されません。

ファイル名	説明
math.h	一般的な数学計算を実行する関数を記述しています。これらの関数の浮動小数点に関する挙動は”9.3. 浮動小数点演算”に記載した仕様に則ったものとなります。指定（必須）事項としてではありませんが、SPUハードウェアが提供する多くのパフォーマンスの高いSIMD (Single Instruction, Multiple Data) 命令を生かすために、これらの数学関数に対応するベクタ関数を追加することもできます。
setjmp.h	非ローカルの goto ステートメントの実行に用いられる関数を記述しています。
signal.h	提供されません。
stdarg.h	数の決まっていない引数へのアクセスに用いられる関数を記述しています。
stdbool.h	便利な Boolean 型名および定数を定義しています。
stddef.h	いくつかの有用な型とマクロを定義しています。 `wchar_t` の定義は含みません。
stdint.h	さまざまな整数型をサイズ制約とともに定義しています。`SIG_ATOMIC_MAX` や `SIG_ATOMIC_MIN`、また `WCHAR_MAX`、`WCHAR_MIN`、`WINT_MAX`、`WINT_MIN` のいずれの定義も含みません。
stdio.h	デバッグ用途の`printf`を除き提供されません。（“デバッグ用のprintf()”を参照）
stdlib.h	さまざまなオペレーション用の関数を記述しています。関数 `getenv`、`mblen`、`mbstowcs`、`mbtowc`、`system`、`wcstombs`、`wctomb` の定義は含みません。また、型である `wchar_t` およびマクロ `MB_CUR_MAX` の定義も含みません。
string.h	数種類の文字列操作に用いられる関数を記述しています。関数 `strxfrm` は“C”ロケールのみを使用します。
tgmath.h	さまざまな型総称数学関数を記述しています。このヘッダファイルで宣言する単精度関数は math.h 内で宣言されている対応する関数用に記述された仕様に則ったものです。
time.h	提供されません。
wchar.h	提供されません。
wctype.h	提供されません。

デバッグ用の printf()

`printf()` はアプリケーションのデバッグ用に提供されます。この関数の実装は対象となるオペレーティングシステムが提供するサービスに依存します。本関数の詳細な仕様は本書では規定しませんが、本関数がもつ機能の全てを実装することを推奨します。このような実装では C 規格で必要とされる通常の出力書式変換指定子の全てを含むことになるでしょう。これに加え、ベクタ出力書式を扱うために *AltiVec Technology Programming Interface Manual* に記載されている型の変換指定子に対応することを推奨します。出力変換指定は以下のような形式をとります。

```
%[<flags>][<width>][<precision>][<size>]<conversion>
```

引数の内容は以下のとおりです。

```
<flags>          ::= <flag-char> | <flags><flag-char>
<flag-char>       ::= <std-flag-char> | <c-sep>
<std-flag-char>    ::= '-' | '+' | '0' | '#' | ' '
<c-sep>          ::= ',' | ';' | ':' | '_'
<width>          ::= <decimal-integer> | '*'
<precision>       ::= '.' <width> | `.' | `.*'
<size>           ::= `hh' | `h' | 'l' | 'll' | 'L' | <vector-size>
<vector-size>     ::= 'v' | `vhh' | 'vh' | 'vl' | 'vll' | 'vL' | `hhv'
                        | 'hv' | 'lv'| 'llv' | 'Lv'
<conversion>      ::= <char-conv> | <str_conv> | <fp-conv> | <int-conv>
```

```
                    | <byte-conv> | <misc-conv>
<char-conv>         ::= 'c'
<str-conv>          ::= 's'
<fp-conv>           ::= 'e' | 'E' | 'f' | 'F' | 'g' | 'G'
<int-conv>          ::= 'd' | 'i' | 'u' | 'p' | 'o' | 'x' | 'X'
<byte-conv>         ::= 'uc' | 'co' | 'cx' | 'cX'
<misc-conv>         ::=  'n' | '%'
```

ベクタ型においてCの標準出力変換仕様からの拡張部分は太字で示してあります。ベクタ型は 表 8-207に示す変換によりフォーマットされます。文字列変換 (<str-conv>)および雑変換 (<misc-conv>)については、ベクタ用には定義していません。また、'p' 整数変換(<int-conv>)も定義していません。デフォルトの分離文字(<c-sep>)には、分離文字を持たない文字変換(<char-conv>)を除きスペースを用います。

表 8-207: ベクタフォーマット

ベクタサイズ	変換	説明
v	<char-conv>	ベクタは 16 個の 1 バイト要素から成る vector char として印刷されます。'c'変換は連続した ASCII 文字を印刷します。
v	<int-conv> <byte-conv>	'uc'変換の場合、ベクタは 16 個の 1 バイト要素から成る vector unsigned char として印刷されます。同様に、'co'、'cx'、'cX'のいずれかの変換の場合、8 進法または 16 進法表記の vector unsigned char または qword として印刷されます。その他全ての整数変換の場合、ベクタは 8 進法 (o)、整数 (d、 i、 u) 又は 16 進法(x、X)表記の 4 個の 4 バイト要素から成る vector unsigned int または vector signed int として印刷されます。
v	<fp-conv>	ベクタは通常の 10 進法表記(f or F)または 10 の累乗の 10 進指数付き(e、E、g、G)のいずれかを用いた符号付き 10 進法小数表記での 4 個の 4 バイト要素から成る vector float として印刷されます。
vhh or hhv	<int-conv>	ベクタは 8 進法(o)、整数(d、i、u)又は 16 進法(x、X)表記を用いた 16 個の 1 バイト要素から成る vector unsigned char または vector signed char として印刷されます。
vh or hv	<int-conv>	ベクタは 8 進法(o)、整数(d、i、u)又は 16 進法(x、X)表記を用いた 8 個の 2 バイト要素から成る vector unsigned short または vector signed short として印刷されます。
vl or lv	<int-conv>	ベクタは 8 進法(o)、整数(d、i、u)又は 16 進法(x、X)表記を用いた 4 個の 4 バイト要素から成る vector unsigned long または vector signed long として印刷されます。
vll or llv	<int-conv>	ベクタは 8 進法(o)、整数(d、i、u)又は 16 進法(x、X)表記を用いた 2 個の 8 バイト要素から成る vector unsigned long long または vector signed long long として印刷されます。
vL or Lv	<fp-conv>	ベクタは通常の 10 進法表記(f または F)または 10 の累乗の 10 進指数付き(e、E、g、G)のいずれかを用いた符号付き 10 進法小数表記での 2 個の 8 バイト要素から成る vector double として印刷されます

malloc()用ヒープ領域

malloc()で割り当てが可能なヒープ領域は _end からスタックの最後迄であると定義されています。ヒープ領域は、ヒープ領域拡張用の関数により拡張することができます。この関数は現在の Stack Pointer Information レジスタの Available Stack Size 要素および一連の Back Chain クワッドワードで示される保存後の SP（スタックポインタ）レジスタにおける Available Stack Size の値すべてをデクリメントするでしょう。

malloc()用ヒープ領域が拡張されるたびに、拡張後のヒープ領域が現在使用中のスタックに跨らないかを検証するようにコードを記述する必要があります。跨る場合は、拡張操作が失敗するようにするべきです。

setjmp および longjmp の実装もまた、ヒープ拡張関数を使用することの影響を受けます。longjmp 関数を呼び出した結果として Stack Pointer Information レジスタを復元する際、longjmp 関数は setjmp と longjmp の間で Available Stack Size の値が変化した場合にそれを検出し、保存された Stack Pointer Information レジスタの値を修正しなければなりません。例えば以下のようになります。

```
SP.avail_stack_size = SP_set.stack_ptr - SP.stack_ptr +
      SP.avail_stack_size;
```

上記の例で SP は現在の Stack Pointer Information レジスタを、SP_set は最後の setjmp 呼び出し後に保存された Stack Pointer Information レジスタを表しています。

8.1.2. C++標準ライブラリ

本セクションでは、C++標準ライブラリとして必要な最小限の内容について述べます。Cライブラリの場合と同様にC++ライブラリのヘッダファイルにはC++ライブラリの内容が宣言または定義されています。表 8-208はC++標準ライブラリの中核を成すヘッダファイルの一覧であり、ISO標準ライブラリとの内容における相違点も記載されています。

表 8-208: C++ライブラリヘッダファイル

ファイル名	説明
algorithm	便利なアルゴリズムを実装する数々のテンプレートを定義しています。
bitset	ビットのセットを管理するテンプレートクラスを定義しています。
complex	複素数演算をサポートするテンプレートクラスを定義しています。
deque	デキューコンテナを実装するテンプレートクラスを定義しています。
exception	提供されません。
fstream	提供されません。
functional	algorithm および numeric で定義されている数種類のテンプレートのための述語を作成する際に役立つテンプレートを定義しています。
iomanip	提供されません。
ios	提供されません。
iosfwd	提供されません。
iostream	提供されません。
istream	提供されません。
iterator	繰返し子を定義および操作する際に役立つ数種類のテンプレートを定義しています。
limits	数値型の属性を検査します。
list	双方向リンクリストコンテナを実装するテンプレートクラスを定義しています。
locale	提供されません。
map	キーを値にマップする連想コンテナを実装するテンプレートクラスを定義しています。
memory	さまざまなコンテナクラス用のメモリ領域を割り当て・解放する数種類のテンプレートを定義しています。
new	メモリ領域を割り当て・解放する数種類の関数を定義しています。
numeric	便利な数値関数を実装する数種類のテンプレートを定義しています。
ostream	提供されません。
queue	キューコンテナを実装するテンプレートクラスを定義しています。
set	連想コンテナを実装するテンプレートクラスを定義しています。
slist	単方向リストコンテナを実装するテンプレートクラスを定義しています。
sstream	提供されません。
stack	スタックコンテナを実装するテンプレートクラスを定義しています。
stdexcept	提供されません。

ファイル名	説明
streambuf	提供されません。
string	文字列コンテナを実装するテンプレートクラスを定義しています。
strstream	提供されません。
typeinfo	提供されません。
utility	数種類の汎用ユティリティ用テンプレートを定義しています。
valarray	値指向配列をサポートする数種類のクラスおよびテンプレートクラスを定義しています。
vector	ベクタコンテナを実装するテンプレートクラスを定義しています。

C++標準ライブラリには従来の 12 個のCヘッダファイルに加え、それぞれに対応する新しいスタイルのC++ヘッダファイルが含まれています。これらのヘッダファイルを 表 8-209に示します。

表 8-209: C++ライブラリに新たに加えられたヘッダファイルと従来のヘッダファイル

新ヘッダファイル	従来のヘッダファイル	説明
cassert	assert.h	関数の実行時にアサーションを行なう関数を記述しています。[1]
cctype	ctype.h	文字を分類する関数を記述しています。[1]
cerrno	errno.h	ライブラリ関数によりレポートされたエラーコードを検査する関数を記述しています。[1]
cfloat	float.h	浮動小数点型の属性を検査するための定義を記述しています。
ciso646	iso646.h	ISO 646 の変種のキャラクタセットを使用したプログラミングのための関数を記述しています。
climits	limits.h	整数型の属性を検査するための関数を記述しています。[1]
clocale	locale.h	提供されません。
cmath	math.h	一般的な数学計算を実行する関数を記述しています。[1]
csetjmp	setjmp.h	非ローカルの goto ステートメントの実行に用いられる関数を記述しています。
csignal	signal.h	提供されません。
cstdarg	stdarg.h	数の決まっていない引数へのアクセスに用いられる関数を記述しています。
cstddef	stddef.h	いくつかの有用な型とマクロを定義しています。[1]
cstdio	stdio.h	提供されません。
cstdlib	stdlib.h	さまざまなオペレーション用の関数を記述しています。[1]
cstring	string.h	いく種類かの文字列の操作に用いられる関数を記述しています。[1]
ctime	time.h	提供されません。
cwchar	wchar.h	提供されません。
cwctype	wctype.h	提供されません。

[1] 個別の実装上の制限については 表 8-206: C ライブラリのヘッダファイルを参照してください。

8.2. 非サポートの言語仕様

C および C++の実装は、各々の ISO/IEC 規格に規定されている言語機能を可能な限り満たすべきです。しかしながら、SPU アーキテクチャの制限のために、特に特定の機能がサポートされていません。以下に非サポート機能を挙げます。

- C++ 例外処理

9. SPU上の浮動小数点演算

C99 規格の Annex F では IEC 60559 の浮動小数点規格のサポートを規定しています。本章では Annex F および ISO/IEC Standard 60559 の規定と異なる形で SPU コンパイラとライブラリに適用される仕様について説明します。

浮動小数点数の挙動は基本的に SPU ハードウェアにより決定されます。単精度の場合、SPU ハードウェアは拡張単精度の値範囲をサポートします。非正規化数の引数は 0 として扱い、NaN や無限大はサポートしていません。丸めモードでサポートしているのは切り捨てのみです（「0 方向への丸め」モード。特定の拡張範囲浮動小数点命令についてのみ例外が適用される）。倍精度の場合、SPU は IEEE 標準で規定されている値範囲をサポートしますが、これについても非正規化数の引数は 0 として扱います。IEEE 定義の例外を検知し、FPSCR レジスタへ累積します。本アーキテクチャでは NaN の伝播についての IEEE 規定は実装されていません。（詳細については *Synergistic Processor Unit 命令セット・アーキテクチャ*を参照してください。）これらの相違点に加え他の IEEE 標準との相違点も、データ型特性、丸めモード、例外ステータス、エラー通知、式の評価を含む浮動小数点演算のほとんどあらゆる面に影響を与えます。以下のセクションでは、これらの相違点がコンパイラやライブラリに及ぼす個々の影響について説明します。

9.1. 浮動小数点型表示の属性

浮動小数点型表示の属性は`float.h`内のマクロとして宣言されます。表 9-210にこれらのマクロとそれぞれに対応するSPUに適用可能な値を示します。

表 9-210: 浮動小数点型属性の値

マクロ	値
FLT_DIG	6
FLT_EPSILON	0x1p-23f (1.19209290E-07f)
FLT_MANT_DIG	24
FLT_MAX_10_EXP	38
FLT_MAX_EXP	129
FLT_MIN_10_EXP	-37
FLT_MIN_EXP	-125
FLT_MAX	0x1.FFFFFEp128f (6.80564694E+38f)
FLT_MIN	0x1p-126f (1.17549436E-38f)
FLT_ROUNDS	16 に初期化（両要素とも最近接値への丸め）
FLT_EVAL_METHOD	0（格上げは生じない）
FLT_RADIX	2
DBL_DIG	15
DBL_EPSILON	0x1p-52 (2.2204460492503131E-016)
DBL_MANT_DIG	53
DBL_MAX_10_EXP	308
DBL_MAX_EXP	1024
DBL_MIN_10_EXP	-307
DBL_MIN_EXP	-1021
DECIMAL_DIG	17

9.2. 浮動小数点環境

fenv.h 内で定義されているマクロは、浮動小数点演算における向き指定丸めの制御モードと浮動小数点例外ステータスフラグを制御します。

9.2.1. 丸めモード

C 言語仕様では全ての浮動小数点型について同一の丸めモードを使用することが規定されていますが、SPU ハードウェアは単精度と倍精度の演算で異なる丸めモードをサポートしています。SPU では単精度用の丸めモードは「0 方向への丸め」であり、倍精度用のデフォルト丸めモードは「最近接値への丸め」となっています。

C99 規格では、浮動小数点加算の丸めモードはFLT_ROUNDSの実装で定義された値により決定すると定めています。SPUの場合、このマクロは倍精度用のみに用い、単精度の丸めモードは常に切り捨てとなります。(表 9-210: 浮動小数点型属性の値参照)。

FLT_ROUNDSは 5 ビットの値を返し、それは倍精度の両要素についての丸めモードを表わしています。最上位ビットは常に 1 となっています。それに続く 2 ビットは要素 0 の丸めモードであり、最下位 2 ビットが要素 1 の丸めモードです。表 9-211に各要素に対する 2 ビットが表わす丸めモードを挙げます。

表 9-211: FLT_ROUNDS の 2 ビットが表わす丸めモード

2 ビットの値	丸めモード
00	最近接偶数への丸め
01	0 方向への丸め（切り捨て）
10	+無限大方向への丸め
11	–無限大方向への丸め

SPUハードウェアでは単精度用のモードとして「0 方向への丸め」モードのみをサポートしているため、単精度数学関数の一部は必然的にC99 規格を逸脱することになります。標準ライブラリにおいて規格を逸脱した数学関数とマクロについては 9.3.2. C演算子および標準ライブラリ数学関数の全般的な挙動にて後述します。

倍精度の丸めモードを要素 0 および要素 1 について設定するために使用できるマクロを 表 9-212に列挙します。要素 0 と要素 1 に対するマクロは、併用してビット毎ORを行なって両要素の丸めモードを設定することができます。あるいは、個別に用いてその要素のみの丸めモードを設定することもできます。

表 9-212: 倍精度丸めモード用マクロ

マクロ	コメント
FE_TONEAREST	要素 0 を最近接偶数への丸めに設定
FE_TOWARDZERO	要素 0 を 0 方向への丸めに設定
FE_UPWARD	要素 0 を+無限大方向への丸めに設定
FE_DOWNWARD	要素 0 を–無限大方向への丸めに設定
FE_TONEAREST_1	要素 1 を最近接偶数への丸めに設定
FE_TOWARDZERO_1	要素 1 を 0 方向への丸めに設定
FE_UPWARD_1	要素 1 を+無限大方向への丸めに設定
FE_DOWNWARD_1	要素 1 を–無限大方向への丸めに設定

9.2.2. 浮動小数点例外

表 9-213および 表 9-214にfenv.h内で定義する浮動小数点例外用マクロを示します。SPU浮動小数点ハードウェアの挙動における制限により、単精度のライブラリ関数はこれらの例外フラグに対して未定義の影響を及ぼす可能性があります。更に、いかなる例外の発生もハードウェアトラップを起こすことはありません。

表 9-213: 単精度浮動小数点例外用マクロ

マクロ	コメント
FE_OVERFLOW_SNGL	要素 0 のオーバフロー例外
FE_UNDERFLOW_SNGL	要素 0 のアンダフロー例外
FE_DIFF_SNGL	要素 0 の IEEE からの相違例外
FE_DIVBYZERO_SNGL	要素 0 の 0 除算例外
FE_OVERFLOW_SNGL_1	要素 1 のオーバフロー例外
FE_UNDERFLOW_SNGL_1	要素 1 のアンダフロー例外
FE_DIFF_SNGL_1	要素 1 の IEEE からの相違例外
FE_DIVBYZERO_SNGL_1	要素 1 の 0 除算例外
FE_OVERFLOW_SNGL_2	要素 2 のオーバフロー例外
FE_UNDERFLOW_SNGL_2	要素 2 のアンダフロー例外
FE_DIFF_SNGL_2	要素 2 の IEEE からの相違例外
FE_DIVBYZERO_SNGL_2	要素 2 の 0 除算例外
FE_OVERFLOW_SNGL_3	要素 3 のオーバフロー例外
FE_UNDERFLOW_SNGL_3	要素 3 のアンダフロー例外
FE_DIFF_SNGL_3	要素 3 の IEEE からの相違例外
FE_DIVBYZERO_SNGL_3	要素 3 の 0 除算例外
FE_ALL_EXCEPT_SNGL	要素 0 用の全マクロのビット毎 OR
FE_ALL_EXCEPT_SNGL_1	要素 1 用の全マクロのビット毎 OR
FE_ALL_EXCEPT_SNGL_2	要素 2 用の全マクロのビット毎 OR
FE_ALL_EXCEPT_SNGL_3	要素 3 用の全マクロのビット毎 OR

表 9-214: 倍精度浮動小数点例外用マクロ

マクロ	コメント
FE_OVERFLOW_DBL	要素 0 のオーバフロー例外
FE_UNDERFLOW_DBL	要素 0 のアンダフロー例外
FE_INEXACT_DBL	要素 0 の ISO/IEC「不正確」
FE_INVALID_DBL	要素 0 の ISO/IEC「無効」
FE_NC_NAN_DBL	要素 0 の非準拠 NaN の可能性
FE_NC_DENORM_DBL	要素 0 の非準拠非正規化数の可能性
FE_OVERFLOW_DBL_1	要素 1 のオーバフロー例外
FE_UNDERFLOW_DBL_1	要素 1 のアンダフロー例外
FE_INEXACT_DBL_1	要素 1 の ISO/IEC「不正確」
FE_INVALID_DBL_1	要素 1 の ISO/IEC「無効」
FE_NC_NAN_DBL_1	要素 1 の非準拠 NaN の可能性
FE_NC_DENORM_DBL_1	要素 1 の非準拠非正規化数の可能性
FE_ALL_EXCEPT_DBL	要素 0 用の全マクロのビット毎 OR
FE_ALL_EXCEPT_DBL_1	要素 1 用の全マクロのビット毎 OR
FE_ALL_EXCEPT	この表の全マクロのビット毎 OR

C99 規格で定められている浮動小数点環境変数は倍精度浮動小数点例外のみに適用されます。

`FENV_ACCESS` プラグマはプログラムが浮動小数点ステータスを制御および検査する意図が有るか否かについての情報をコンパイラへ与えるために使用します。このプラグマが有効である場合、コンパイラはコード変換後も本仕様で定めた挙動を確実に保つための適切な処理を行ないます。

9.2.3. math.hで定義されているその他の浮動小数点定数

math.h内にはその他のいくつかの浮動小数点定数が定義されています。これらの定数は関数がさまざまな定義域エラーや値域エラーを通知するために用いられます。これらの多くはSPU用に標準とは異なる定義を持ちます。表 9-215にこれらの特別な定数を示します。

表 9-215: 浮動小数点定数

マクロ	説明
HUGE_VAL	無限大
HUGE_VALF	FLT_MAX
HUGE_VALL	無限大
INFINITY NAN	倍精度演算は IEEE の定義に準拠。これらのマクロは単精度演算には用いない。
FP_INFINITE FP_NAN FP_NORMAL FP_SUBNORMAL FP_ZERO	単精度演算の場合、fpclassify() は FP_NORMAL および FP_ZERO クラスのみを返し、FP_NAN、FP_INFINITE、FP_SUBNORMAL が生成されることは決してない。
FP_FAST_FMA FP_FAST_FMAF FP_FAST_FMAL	これらは fma 関数が float と double のオペランドを乗算・加算するより速く実行することを示すために定義されている。
FP_ILOGB0 FP_ILOGBNAN	FP_ILOGB0 は x が 0 であるか非正規化値である場合に ilogb(x) と ilogbf(x) が返す値で、この値は INT_MIN である。 FP_ILOGBNAN は x が NaN である場合に ilogb(x) が返す値で、この値は INT_MAX である。単精度の ilogbf についてはこの値は適用されない。
MATH_ERRNO MATH_ERREXCEPT	これらの定数は整数定数 1 と 2 へそれぞれ展開する。
math_errhandling	この定数は int 型で MATH_ERRNO か MATH_ERREXCEPT か、あるいは両者の論理和の値をもつ式へ展開する。math_errhandling の値はプログラムが実行されている間は一定の値に保たれる。

9.3. 浮動小数点演算

本セクションでは浮動小数点データ変換について規定し、C の演算子および標準ライブラリ関数の全般的な挙動や浮動小数点演算の結果が IEEE 規格を逸脱する可能性をもつ特殊なケースについて説明します。また、セクションの終わりの部分でいくつかの数学関数における特殊な挙動について述べます。

9.3.1. 浮動小数点変換

本セクションでは次に挙げる 4 種類の浮動小数点データ変換について規定します。1 種類目は整数から浮動小数点数への変換、2 種類目は浮動小数点数から整数への変換、3 種類目は浮動小数点精度間の変換、そして 4 種類目は浮動小数点数と文字列間の変換です。

整数から浮動小数点数への変換

整数から浮動小数点数への変換は、以下の規則に則ったものとします。

- 整数を浮動小数点数へ単精度変換すると、拡張単精度浮動小数点数域内の結果が生じる。この値の範囲については 表 9-210: 浮動小数点型属性の値を参照。
- 整数を浮動小数点数へ単精度変換すると、0 方向への丸めが行なわれる。
- 整数を浮動小数点数へ倍精度変換すると、C99 の浮動小数点数域内の結果が生じる。
- 整数を浮動小数点数へ倍精度変換すると、FLT_ROUNDS の値が示す丸めモードでの丸めが行なわれる。

浮動小数点数から整数への変換

浮動小数点数から整数への変換では以下のような挙動を示します。

- 単精度浮動小数点数を整数へ変換すると、オーバーフロー例外、アンダーフロー例外、および IEEE 不適合結果例外が発生する。
- 倍精度浮動小数点数を整数へ変換するとオーバーフロー例外とアンダーフロー例外が発生する。倍精度浮動小数点数が無限大あるいは NaN である場合や浮動小数点数の整数部が整数型の値の範囲を超えている場合、「無効演算（invalid）」浮動小数点数例外が発生し、結果は不定である。整数型の範囲外だが整数の値をもつ浮動小数点数を変換した場合、ハードウェアによる「不正確（inexact）」浮動小数点数例外が発生する。

浮動小数点精度間の変換

コンパイラは最大の性能を出すために IEEE 規格で定める範囲内のみで `float` から `double` および `double` から `float` への変換を行ないます。非正規化数を入力した場合（結果は強制的に 0 となります）を除き、これらの変換は IEEE 規格へ準拠したものとなります。IEEE 規格で定める範囲外の値の変換については規定しません。結果が NaN、無限大、あるいは非正規化数となる変換についても規定していません。

浮動小数点数と文字列間の変換

浮動小数点数と文字列間の変換は拡張単精度浮動小数点範囲および IEEE 規格の倍精度浮動小数点範囲に則るものとします。

9.3.2. C演算子および標準ライブラリ数学関数の全般的な挙動

ライブラリ関数とコンパイラは丸めおよびオーバーフローに関して同一の一般的規則に従います。しかしながら、これらの規則はコードが単精度であるか倍精度であるかにより異なります。

単精度コードの場合

単精度の場合、C 演算子(+ -, *, /)および標準ライブラリ数学関数は以下のような挙動を示します。

- 演算結果の絶対値が表現可能な正の最大拡張精度数より大きい場合、結果は適切な符号の付いた FLT_MAX となり、オーバーフローフラグがセットされる。
- 符号反転演算子と `fabsf()`、`copysignf()` 関数を除くすべての演算子と標準関数について、非正規化数の引数は`+0.0` として扱われる。
- 符号反転演算子と `fabsf()`、`copysignf()` 関数を除くすべての演算子と標準関数は、非正規化数または–0.0 を返すことはない。
- 符号反転演算子と `fabsf()`、`copysignf()` 関数は、符号ビットだけが変更されるように実装しなければならない。
- 式は「0 方向への丸め」モードで評価される。アルゴリズムの正確性が別の丸め方向に依存するような実装をすると正しくない結果を生じるため、このような実装のプログラムは使用できない。
- 絶対値が大き過ぎる値の代わりに FLT_MAX が返る場合、オーバーフローフラグがセットされる。単精度の無限大は定義されていないため、IEEE754 準拠のシステムならば無限大が生成されるようなケースにおいては、無限大であることを通知するために FLT_MAX を用いるものとする。この変更により一般的な三角関数恒等式を機能させることができる。
- NaN はサポートしておらず、入力パラメータからのコピーは一切必要ない。
- デフォルトでは、コンパイラは次の仮定のもとに単精度演算の最適化を行なうことができる。1) 引数に NaN が渡されることはない。2) 結果として正負の無限大が生成されることはない。
- コンパイラは浮動小数点演算がゼロ除算、オーバーフロー、アンダーフローのようにユーザが認識可能なトラップを発生させることはないとみなすことができる。
- コンパイル時に計算される定数式の結果は、実行時に計算した場合に得られるであろう結果と同じものとなる。例えば、以下の式は FLT_MAX として計算される。

```
float x = 6.0e38f * 8.1e30f;
```

- コンパイラは FP_CONTRACT プラグマや *no-fast-float* コンパイラオプションにより明示的に禁止されない限り、Floating Reciprocal Absolute Square Root Estimate (frsqest)や Floating Multiply and Add (fma)のような単精度縮約演算を使用できる。縮約演算を使用する場合、`errno` を設定する必要はない。

倍精度コードの場合

倍精度浮動小数点演算の場合、C 演算子や標準ライブラリ数学関数は以下の点を除いて IEEE 規格に準拠するものとします。

- 演算の結果として NaN が返される場合は常に QNaN である。
- 符号反転演算子と `fabs()`、`copysign()` 関数を除き、非正規化数は結果としてのみサポートする。非正規化数のオペランドは 0 にそのオペランドと同一の符号を付けた値として扱う。
- 倍精度におけるデフォルトの丸めモードは「最近接値への丸め」とする。
- コンパイラは FP_CONTRACT プラグマや *no-fast-double* コンパイラオプションにより明示的に禁止されない限り、Double Floating Multiply and Add (dfma)のような倍精度縮約演算を使用できる。縮約演算を使用する場合、`errno` をセットする必要はない。

9.3.3. 浮動小数点式における特殊なケース

C99 規格に記載されている標準式変換の中には、以下のように SPU 上では必要とされる効果を得ることができない可能性があるものが含まれています。

- `x/2 -> x*0.5`
  値が完全な 2 のべき乗であるこの特定の値については有効であるが、一般的(例えば、 `x/10 != x*0.1`)には浮動小数点定数が有限の 2 進浮動小数点表記法で完全に表現することができないため無効。
- `x*1 -> x and x/1 -> x`
  以下の場合には無効。1） x が SNaN かデフォルト以外の QNaN である場合（倍精度のみ）。2） `x` が非正規化数である場合。3） `x` が `-0.0` である場合（単精度のみ）。
- `x/x -> 1.0`
  単精度の場合、`x` が 0 または非正規化数のとき無効。倍精度の場合 `x` が `0`、非正規化数、無限大、 `NaN` のいずれかである場合に無効。
- `x-y -> -(y-x)`
  結果が `0` の場合は互いに異なる符号を持つ可能性があるため無効。また、倍精度の場合は、正または負の無限大へ丸めた場合の 0 以外の結果には 1 ulp の誤差が生じる可能性があるため無効。
- `x-x -> 0.0`
  単精度の場合は常に有効。倍精度の場合、`x` が `NaN` または無限大であるとき、この等価演算は無効である。負の無限大方向へ丸める倍精度演算の場合も無効であり、この場合結果は`-0.0` となる。
- `0*x -> 0.0`
  単精度の場合は常に有効。倍精度の場合 `x` が `NaN`、無限大、負の数、`-0` のいずれかである場合、この等価演算は無効である。
- `x+0 -> x`
  単精度の場合、`x` が非正規化数のオペランドまたは `-0` であるときに無効。倍精度の場合、「最近接値への丸め」「正の無限大への丸め」、「切り捨て」のいずれかのモードにおいて `x=-0` のときに無効。また、倍精度で `x` が SNaN、デフォルト以外の QNaN、非正規化数のいずれかであるとき（最後に挙げた条件の場合、x+0 は適切な符号付きの 0 となる）に無効。
- `x-0 -> x`
  単精度の場合、`x` が非正規化数のオペランドまたは `-0` であるケースを除き有効。倍精度の場合、`x` が SNaN、デフォルト以外の QNaN、非正規化数のいずれかであるときに無効であり、`x` が `+0` かつ丸めモードが「負の無限大への丸め」であるとき（この場合 `x-0 = +0-0 =-0`）も無効である。オペランドが正規化数であるときの結果は、丸めモードが「負の無限大への丸め」であっても有効。
- `-x -> 0-x`
  単精度の場合 `x` が`+0` または非正規化数であるときに無効。倍精度の場合、次の条件にあてはまるとき無効。1)NaN の場合、-x の値は不定であり、結果はすべての NaN について異なる。2) `x` が `+0` かつ丸めモードが「偶数の最近接値への丸め」、「正の無限大への丸め」、「切り捨て」のいずれかであるとき、`0-x = +0` かつ `-x = -0` となる。

- `x!=x -> false`
  単精度の場合に常に有効。倍精度の場合、`x=NaN` の比較は常に比較不能となるため、`x!=x` が真となる。
- `x==x -> true`
  単精度の場合に常に有効。倍精度の場合、`x=NaN` の比較は常に比較不能となるため、`x==x` が偽となる。
- `x<y -> isless(x,y)`,
  `x<=y -> islessequal(x,y)`,
  `x>y -> isgreater(x,y)`,
  `x>=y -> isgreaterequal(x,y)`
  倍精度の場合に `x` か `y` が `NaN` であることの副作用でフラグがセットされる場合を除いて有効。このような無効なフラグの挙動は FENV_ACCESS プラグマにより変更が可能。

9.3.4. 標準数学関数における特殊な挙動

本セクションでは `math.h` で宣言されているさまざまな浮動小数点関数がもつ特有の挙動について説明します。前述のとおり SPU ハードウェアは浮動小数点関数の挙動に直接的影響を持ちます。厳密な IEEE 仕様の挙動とハードウェアの挙動とでは多くの相違点があるため、標準数学関数は例外を発生させるようなケースあるいは値の範囲を超えている状態に対して厳密なチェックを行なう必要がありません。従って、多くの関数の結果を定義しなおしています。以下に相違点を挙げます。

- `nanf()` 関数は `0` を返す。
- `isnan()` マクロは単精度の場合は常に偽を返す。
- C99 標準仕様とは異なり、`nearbyint`, `lrint`, `llrint`, `fma` の単精度演算はいずれも 0 方向へ丸められる。
- 三角関数、双曲線関数、指数関数、対数関数、ガンマ関数において値が丸められる場合に inexact フラグをセットする必要はない。
- 単精度の場合における `frexp(NaN,exp)` や `modf(NaN,iptr)` の境界例については、これらの関数が NaN を伝播して返すことから定義していない。
- `nextafterf(非正規化数,y)` 関数がアンダーフローフラグを立てることはない。`nextafterf()` 関数や `nexttowardf()` 関数は IEEE の定める最大 float 値を超えた値に増加する場合も成功する。
- 単精度の場合について以下に挙げる境界例は無限大の引数は無効であるためサポートしない。
  `atanf(±inf)`, `atan2f(±y, ±inf)`, `atan2f(±inf,x)`, `atan2f(±inf,±inf)`, `acoshf(+inf)`, `asinhf(±inf)`, `atanhf(±1)`, `atanhf(±inf)`, `coshf(±inf)`, `sinhf(±inf)`, `tanhf(±inf)`, `expf(±inf)`, `exp2f(±inf)`, `expm1f(±inf)`, `frexpf(±inf,&exp)`, `ldexpf(±inf,exp)`, `logf(+inf)`, `log10f(+inf)`, `log1pf(+inf)`, `log2f(+inf)`, `logbf(±inf)`, `modff(±inf,iptr)`, `scalbnf(±inf,n)`, `cbrtf(±inf)`, `fabsf(±inf)`, `hypotf(±inf,y)`, `powf(-1,±inf)`, `powf(x,±inf)`, `powf(±inf,y)`, `sqrtf(±inf)`, `erff(±inf)`, `erfcf(±inf)`, `lgammaf(±inf)`, `tgammaf(+inf)`, `ceilf(±inf)`, `floorf(±inf)`, `nearbyintf(±inf)`, `roundf(±inf)`, `rintf(±inf)`, `lrintf(±inf)`, `llrintf(±inf)`, `lroundf(±inf)`, `llroundf(±inf)`, `truncf(±inf)`, `fmodf(x,±inf)`, `remainderf(±inf)`, `remquof(±inf)`, `copysignf(±inf)`
- 単精度の場合について以下に挙げる境界例は IEEE に準拠していない値を返す。 acosf(|x|>1), asinf(|x|>1), acoshf(x<1.0), atanhf(|x|>1), tgammaf(x<0), fmodf(x,0), ldexpf(x,BIG_INT), `logf(±0)`, `logf(x<0)`, `log10f(±0)`, `log10f(x<0)`, `log1pf(-1)`, `log1pf(x<-1)`, `log2f(±0)`, `log2f(x<0)`, `logbf(±0)`, `powf(±0,y)`, `tgammaf(±0)`
- 単精度の場合について以下に挙げる境界例は NaN を返さない。
  cosf(±inf), sinf(±inf), tanf(±inf), tgammaf(-inf), fmodf(±inf,y), nextafterf(x,±inf), fmaf(±inf|0,0|±inf,z), fmaf(±inf,0,-+inf).
- 倍精度の関数の引数として単精度の値を与えた場合および単精度の変数へ倍精度の関数の結果を代入した場合の暗黙の変換の挙動については 9.3.1. 浮動小数点変換 で説明している。

索引

P

R

S

あ

い

え

く

た

て

ひ

ふ

へ

ほ

ま

以上