SONY

SPU C/C++ 言語拡張

Version 2.1

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

2005 年 10 月 20 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

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**

2005年 10月 20日

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、Synergistic Processor Unit（SPU）用の C/C++言語拡張について説明します。この言語拡張を用いることにより、SPU ハードウエアの特徴を活かしたプログラムの記述が可能となり、SPU プログラムの性能を最大限に引き出すことができます。

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 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. 2.1 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 RFC 00024-3) 「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 RFC 00034-1: CORRECTION NOTICE). `si_stqx` 命令の記述を修正。(TWG RFC 00035-0: CORRECTION NOTICE). 様々なドキュメンテーション関連の間違いを修正。例：「ベクタリテラルの代替形式と説明」の表を修正。(TWG RFC 00036-0: CORRECTION NOTICE, TWG RFC 00041-0: CORRECTION NOTICE, TWG RFC 00045-0: CORRECTION NOTICE). 「Broadband Processor Architecture」を「Cell Broad Engine Architecture」へ、「BPA」を「CBEA」へ変更。(TWG RFC 00037-0: 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 RFC 00040-0: 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 RFC 00043-2).
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 RFC 00032-0: CORRECTION NOTICE による)

バージョン番号および日付	変更点
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 RFC 00031-0: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。
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 RFC 00028-1: 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 RFC 00029-1)
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 RFC 00007-1) 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 RFC 00008-1) SPU 命令「`orx`」を追加。(TWG RFC 00010-0) 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 RFC 00011-0) 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 RFC 00013-0) 「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 RFC 00014-0) 組み込み関数 `si_lqd` および `si_stqd` についての記述を変更。(TWG RFC 00015-1) 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 RFC 00016-1) 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	`xix`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 RFC 00002） ベクタリテラル形式用に新たに仕様を追加。（TWG RFC 00003）
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。

バージョン番号および日付	変更点
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。分離制御チャネル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。
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則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。

バージョン番号および日付	変更点
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。 `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)		
Synergistic Processor Unit 命令セット・アーキテクチャ	1.0	2005 年 8 月
Cell Broadband Engine Architecture	1.0	2005 年 7 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

本ドキュメントの構成

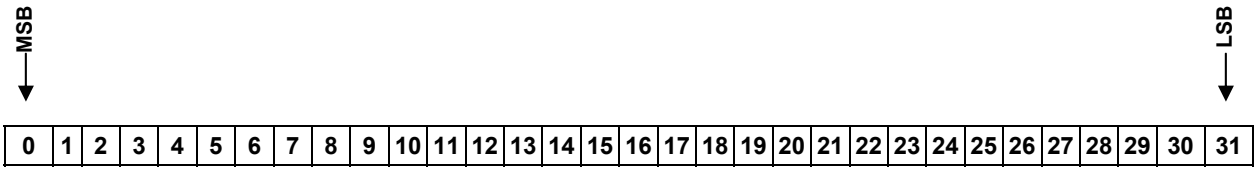
本ドキュメントは、主に以下のセクションを含みます。

1. データ型とプログラム指示文
2. 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. C/C++標準ライブラリ
7. SPU 上の浮動小数点演算

本ドキュメント内で用いられるビット表記および書体の表記上の規則

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’

書体の表記上の規則

本ドキュメントでは、ビット表記に加え、以下の書体の表記上の規則を採用しています。

書体	意味
`courier`	プログラミングコード、処理命令、レジスタ名、データタイプ、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、特に説明文に使用されます。
`courier + イタリック体`	引数、パラメータ、および変数（`const` タイプの変数を含む）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、特に説明文に使用されます。
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

本章では本仕様に必要な基本的データ型、これらのデータ型の操作、また指示文とプログラム制御について説明します。

1.1. データ型

SPU プログラミングモデルは基本的ベクタデータ型一式をC言語へ導入します。ベクタデータ型は全て 128 ビット長で種類により 2 から 16 の要素を含みます。表 1-1にサポートするベクタ型を示します。

表 1-1: ベクタデータ型

ベクタデータ型	内容
vector unsigned char	16 個の 8 ビット符号なし char
vector signed char	16 個の 8 ビット符号付き char
vector unsigned short	8 個の 16 ビット符号なし halfword
vector signed short	8 個の 16 ビット符号付き halfword
vector unsigned int	4 個の 32 ビット符号なし word
vector signed int	4 個の 32 ビット符号付き word
vector unsigned long long	2 個の 64 ビット符号なし doubleword
vector signed long long	2 個の 64 ビット符号付き doubleword
vector float	4 個の 32 ビット単精度 float
vector double	2 個の 64 ビット倍精度 float
qword	quadword (16-byte)

`qword` 型は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、この型については2.1. 個別組み込み関数で説明しています。

コードの移植性を高めるために、ベクタキーワードデータ型用に単一トークンの型定義をヘッダファイル「`spu_intrinsics.h`」で定義しています。これらの型定義を表 1-2に示します。これらの単一トークン型は総称組み込み関数の拡張や Vector Multimedia Extension 組み込み関数および SPU 組み込み関数間のマッピング時にクラス名として使用されます。

表 1-2: 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	単一トークン型定義
vector unsigned char	vec_uchar16
vector signed char	vec_char16
vector unsigned short	vec_ushort8
vector signed short	vec_short8
vector unsigned int	vec_uint4
vector signed int	vec_int4
vector unsigned long long	vec_ullong2
vector signed long long	vec_llong2
vector float	vec_float4
vector double	vec_double2

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;
```

1.2. バイトオーダと要素番号

図 1-1に示すように、バイトオーダ、要素番号、スロット番号は常にビッグエンディアンで表されます。

図 1-1: ベクタ型用のビッグエンディアンによるバイトオーダと要素番号

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

1.3. ベクタ型を使用した演算

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。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。

1.3.1. sizeof() 演算子

ベクタ型を使用しての sizeof()演算は常に 16 を返します。

1.3.2. 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。

1.3.3. アドレス演算子

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

1.3.4. ポインタ演算とポインタ逆参照

ポインタからベクタ型への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（1.6. 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、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);
}
```

1.3.5. 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。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。

あるベクタ型から別のベクタ型へのキャストは通常のＣ言語キャストで提供されています。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た引数のビットパターンと等しくなります。

ベクタ型とスカラ型間のキャストは許されていません。代りに `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。）

1.3.6. ベクタリテラル

表 1-3に示すように、ベクタリテラルは、中括弧で括られた一組の定数式が後に続く、小括弧で括られたベクタ型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。ベクタの要素は対応する定数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の定数として使用できます。

表 1-3: ベクタリテラルの形式と説明

表記	意味
(vector unsigned char) {unsigned int, ...}	16 個の符号なし 8 ビット量 1 セット
(vector signed char) {signed int, ...}	16 個の符号付き 8 ビット量 1 セット
(vector unsigned short) {unsigned short, ...}	8 個の符号なし 16 ビット量 1 セット
(vector signed short) {signed short, ...}	8 個の符号付き 16 ビット量 1 セット
(vector unsigned int) {unsigned int, ...}	4 個の符号なし 32 ビット量 1 セット
(vector signed int) {signed int, ...}	4 個の符号付き 32 ビット量 1 セット

表記	意味
(vector unsigned long long) {unsigned long long, ...}	2 個の符号なし 64 ビット量 1 セット
(vector signed long long) {signed long long, ...}	2 個の符号付き 64 ビット量 1 セット
(vector float) {float, ...}	4 個の 32 ビット 浮動小数点量 1 セット
(vector double) {double, ...}	2 個の 64 ビット 浮動小数点量 1 セット

vector/SIMD multimedia extension との互換性を保つために、表 1-4に示すような括弧で括られた一組の定数式が後に続く括弧で括られたベクタ型の形をとる代替フォーマットもサポートする必要があります。

表 1-4: ベクタリテラルの代替形式と説明

表記	意味
(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
(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
(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
(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
(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
(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
(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
(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
(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
(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
(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
(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
(vector unsigned long long)(unsigned long long)	int 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
(vector unsigned long long)(unsigned long long, unsigned long long)	2 個の int で指定した値をもつ符号なし 64 ビット量 2 個 1 セット
(vector signed long)(signed long long)	int 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
(vector signed long)(signed long long, signed long long)	2 個の int で指定した値をもつ符号付き 64 ビット量 2 個 1 セット
(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
(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

表記	意味
(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
(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

1.4. ヘッダファイル

システムヘッダファイル「spu_intrinsics.h」に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（1ページの表 1-2: 単一トークンベクタデータ型を参照）および MFC チャネルニーモニックの列挙定数の定義（55ページの表 2-86: 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には本仕様書で定義する言語拡張機能をサポートするのに必要な実装固有の定義をすべて含むヘッダファイル「spu_internals.h」をインクルードする必要があります。

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確実に特定のポインタを通して行なえるようにすることによりコンパイラがより良いコードを生成でき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 で修飾されています。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. アラインメント

表 1-5に各種データ型のサイズおよび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	バイト
short	2	ハーフワード
int	4	ワード
long	4	ワード、ダブルワード
long long	8	ダブルワード
float	4	ワード
double	8	ダブルワード
pointer	4	ワード
vector	16	クワッドワード

GCC の aligned 属性を使用することにより、変数あるいは構造体や合併のメンバについてアラインメント制御を行なうことができます。例えば、以下の宣言文において浮動小数点スカラ *factor* はクワッドワード境界にアラインされます。

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

1.6.1. __align_hint

__align_hint 組み込み関数は以下の目的で提供されています。

- ポインタ経由のデータアクセス性を高める。
- 自動ベクトル化のサポートに必要な付加的情報をコンパイラへ与える。

__align_hint は組み込み関数として定義されていますが、具体的コードを生成することはないという意味で、指示文のような挙動を示します。以下に例を示します。

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

__align_hint 組み込み関数はコンパイラへ、ポインタ *ptr* がベースアラインメントを示す *base* および *base* からのオフセットを示す *offset* により指定されるアドレスのデータを指していることを知らせます。ベースアラインメントは 2 のべき乗である必要があります。ベースアドレスが 0 の場合、そのポインタのアラインメントが不明であることを意味します。オフセットは *base* 未満または 0 の値をとる必要があります。

__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を指定することを意図したものではありません。自然なアラインメントになってい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境界を跨いだ状態で配置されてしまいます。プログラマがアラインメントの指定を誤るとプログラムも正しくないものとなる恐れがあります。

プログラミングの注意: コンパイラの実装を仕様に準拠させるためには__align_hint 組み込み関数を提供する必要がありますが、コンパイラはこのようなヒントを無視することができます。

1.7. プログラマの指示による分岐予測

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、標準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、GCC の__builtin_expect 関数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)
```

プログラマはコンパイラへ分岐予測情報を提供するために_builtin_expect を使用することができます。_builtin_expect の返り値は引数 *exp* の値であり、この値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 は同じ値であることを前提としています。

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。このようなケースで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。

1.8. インラインアセンブリ

C/C++言語の定数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。このようなケースにおいて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。

1.9. SPU ターゲット定義

SPU や PowerPC® Processor Unit (PPU)のような複数のターゲットを対象とした条件付コンパイル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 `__SPU__`を定義する必要があります。例えば以下のコードは SPU と PPU の両ターゲットについて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__`の定義は条件付でコードを選択する場合に用います。対象となるプロセッサにより、異なるコードが選択されます。

```
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)));
#else /* PPU */
       qw = vec_perm(qw0, qw1, vec_lvsl(0, ptr));
#endif
       return (qw);
       }
```

2. 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1.3.6. 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 2-6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です。

表 2-6: 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
IRET 命令	iret, iretd, irete

表 2-7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。

- 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
- 限られた状況を除き使用が見込まれない命令

表 2-7: 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	文法	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	文法	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

命令と説明	文法	アセンブリ命令へのマップ
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
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]

命令と説明	文法	アセンブリ命令へのマップ
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
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

命令と説明	文法	アセンブリ命令へのマップ
制御用の組み込み関数		
si_stopd: Stop and Signal with Dependencies SPU の実行を停止し、全てのレジスタ依存性が満たされたときに 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 2-8を参照してください。

表 2-8: キャスト用個別組み込み関数

キャスト用組み込み関数	d	a	説明
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

キャスト用組み込み関数	d	a	説明
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 2-9 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 2-9: 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、その定数領域へ CONST_AREA と offset により特定されるアドレスに存在する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. 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総称組み込み関数がサポートしている入力データ型を指定している。
- 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 のように表現することにより、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 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 2-10: Replicate (Splat) a Scalar across a Vector

d	a	個別組み込み関数	アセンブリ命令へのマップ
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)		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: vector convert 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 2-11: Convert an Integer Vector to a Vector Float

d	a	scale	個別組み込み関数	アセンブリ命令へのマップ
vector float	vector unsigned int	unsigned int	*d* = si_cuflt(*a*, *scale*)	CUFLT d, a, scale
vector float	vector signed int	(7-bit literal)	*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 2-12: Convert a Vector Float to a Signed Integer Vector

d	a	scale	個別組み込み関数	アセンブリ命令へのマップ
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 2-13: Convert a Vector Float to an Unsigned Integer Vector

d	a	scale	個別組み込み関数	アセンブリ命令へのマップ
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符号拡張しベクタ *d* の対応する要素へ代入します。

表 2-14: Sign Extend Vector Elements

d	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-15: Round a Vector Double to a Float

d	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-16: Vector Add

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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			

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-17: Vector Add Extended

d	a	b	c	個別組み込み関数	アセンブリ命令へのマップ
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 2-18: Vector Generate Borrow

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-19: Vector Generate Borrow Extended

d	a	b	c	個別組み込み関数	アセンブリ命令へのマップ
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 2-20: Vector Generate Carry

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-21: Vector Generate Carry Extended

d	a	b	c	個別組み込み関数	アセンブリ命令へのマップ
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 2-22: Vector Multiply and Add

d	a	b	c	個別組み込み関数	アセンブリ命令へのマップ
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 2-23: Vector Multiply High High and Add

d	a	b	c	個別組み込み関数	アセンブリ命令へのマップ
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 2-24: Vector Multiply and Subtract

d	a	b	c	個別組み込み関数	アセンブリ命令へのマップ
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 2-25: Multiply Floating-Point Elements

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-26: Vector Multiply High

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-27: Multiply Four (16-bit) Even-Numbered Integer Elements

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-28: Multiply Four (16-bit) Odd-Numbered Integer Elements

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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。	

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-29: Vector Multiply and Shift Right

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-30: Negative Vector Multiply and Add

d	a	b	c	個別組み込み関数	アセンブリ命令へのマップ
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 2-31: Negative Vector Multiply and Subtract

d	a	b	c	個別組み込み関数	アセンブリ命令へのマップ
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 2-32: Vector Floating-Point Reciprocal Estimate

d	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-33: Vector Reciprocal Square Root Estimate

d	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-34: Vector Subtract

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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		

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-35: Vector Subtract Extended

d	a	b	c	個別組み込み関数	アセンブリ命令へのマップ
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 2-36: Absolute Difference of Sixteen (8-bit) Unsigned Integer Elements

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-37: Average Sixteen (8-bit) Integer Elements

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-38: Sum Sixteen (8-bit) Unsigned Integer Elements

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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仮定に基づき、bisled 命令の対象となる関数 `func` は全ての揮発性レジスタを非揮発性レジスタとして見なす必要があるため、標準の呼び出し規則へ準拠していません。標準の呼び出し規則の詳細については「*SPU Application Binary Interface Specification* 」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 2-39: 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 2-40: Compare Absolute Equal Element by Element

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-41: Compare Absolute Greater Than Element by Element

c	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-42: Compare Equal Element by Element

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-43: Compare Greater Than Element by Element

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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。	
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 2-44: Halt If Compare Equal

a	b	個別組み込み関数	アセンブリ命令へのマップ [1, 2]
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 2-45: Halt If Compare Greater Than

a	b	個別組み込み関数	アセンブリ命令へのマップ [1,2]
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 2-46: Count Ones for Bytes

d	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-47: Count Leading Zero for Words

d	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-48: Gather-bits from a Vector of Bytes, Halfwords, or Words

d	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-49: Form Selection Mask for a Vector of Bytes

d	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-50: Form Selection Mask for Vector of Halfwords

d	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-51: Form Selection Mask for Vector of Words

d	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-52: Select Bits from Vector of Bytes

d	a	b	pattern	個別組み込み関数	アセンブリ命令へのマップ
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 bytes of a vector

```
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 2-53: Shuffle Two Vectors of Bytes

d	a	b	pattern	個別組み込み関数	アセンブリ命令へのマップ
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 2-54: Vector Bit-Wise AND

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-55: Vector Bit-Wise AND with Complement

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-56: Vector Bit-Wise Equivalent

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-57: Vector Bit-Wise Complement of AND

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-58: Vector Bit-Wise Complement of OR

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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: vectorbit-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 2-59: Vector Bit-Wise OR

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-60: Vector Bit-Wise OR with Complement

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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 2-61: OR Word Elements Across

d	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-62: Vector Bit-Wise Exclusive OR

d	a	b	個別組み込み関数	アセンブリ命令へのマップ
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

```
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 2-63: Element-Wise Rotate Vector Left by Bits

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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* の各要素のビット分の論理右シフト（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 2-64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 = (shift & 0x10)? 0: LSR(h,bitshift);
}

For (each word element w in vector a){
    int bitshift = -count & 0x3F;
    w = (shift & 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 2-64: Rotate Left and Mask Vector by Bits

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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* の各要素のビット分の算術右シフト（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 2-65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 = (shift & 0x10)? 0: ASR(h,bitshift);
}

For (each word element w in vector a){
    int bitshift = -count & 0x3F;
    w = (shift & 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 2-65: Element-Wise Rotate Left and Mask Algebraic by Bits

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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			
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 2-66: Rotate Left and Mask Vector by Bits

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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_rot qmbi(*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 2-67: Rotate Left and Mask Quadword by Bytes

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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, b
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, b
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			

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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 2-68: Rotate Left and Mask Quadword by Bytes from Bit Shift Count

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
vector unsigned char	vector unsigned char	int	*d* = si_rotqmbybi(*a*, *count*)	ROTQMBYBI d, a, b
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 quadword left 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 2-69: Rotate Quadword Left by Bits

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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			
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: quadword rotate left 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 2-70: Rotate Left Quadword by Bytes

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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 2-71: Rotate Left Quadword by Bytes from Bit Shift Count

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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 2-72: Element-Wise Shift Left Vector by Bits

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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			

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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 quadword left 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 2-73: Shift Left Quadword by Bits

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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			

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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 2-74: Shift Left Quadword by Bytes

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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			
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			

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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 2-75: Shift Left Quadword by Bytes from Bit Shift Count

d	a	count	個別組み込み関数	アセンブリ命令へのマップ
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 2-76: Disable Interrupts

個別組み込み関数	アセンブリ命令へのマップ
N/A	ILA t, *next_inst* BID t *next_inst:*

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 2-77: Enable Interrupts

個別組み込み関数	アセンブリ命令へのマップ
N/A	ILA t, *next_inst* BIE t *next_inst:*

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 2-78: Move from Floating-Point Status and Control Register

d	個別組み込み関数	アセンブリ命令へのマップ
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 2-79: Move from Special Purpose Register

d	register	個別組み込み関数	アセンブリ命令へのマップ
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 2-80: Move to Floating-Point Status and Control Register

a	個別組み込み関数	アセンブリ命令へのマップ
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 2-81: Move to Special Purpose Register

register	a	個別組み込み関数	アセンブリ命令へのマップ
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。この関数はデータの LS への格納が 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 2-82: 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 2-83: 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、保留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を行ないます。命令ストリームを更新するストア命令の後にはこの命令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 2-84: 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 2-85および表 2-86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 Architecture* 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の順序は決して他のチャネルコマンドや揮発性を持つ LS へのアクセスの順序と入れ替えてはなりません。

表 2-85: 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。
1	SPU_WrEventMask	イベントステータスマスクの書き込み。
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_RdEventStatMask	イベントステータスマスクの読み込み。

チャネル番号	ニーモニック	説明
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 2-86: 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 有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 有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 2-87: Read Word Channel

d	channel	個別組み込み関数	アセンブリ命令へのマップ
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 2-88: Read Quadword Channel

d	channel	個別組み込み関数	アセンブリ命令へのマップ
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 2-89: Read Channel Count

c	channel	個別組み込み関数	アセンブリ命令へのマップ
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 2-90: Write Word Channel

channel	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-91: Write Quadword Channel

channel	a	個別組み込み関数	アセンブリ命令へのマップ
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 2-92: Extract Vector Element from the Specified Element

d	a	element	個別組み込み関数	アセンブリ命令へのマップ [1]
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

d	a	element	個別組み込み関数	アセンブリ命令へのマップ [1]
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 2-93: Insert Scalar into Specified Vector Element

d	a	b	element	個別組み込み関数	アセンブリ命令へのマップ [1]
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)
```

element で指定する要素内でスカラ *a* を値 *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 2-94: Promote Scalar to Vector

d	a	element	個別組み込み関数	アセンブリ命令へのマップ [1]
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 Architecture* を参照してください。

表 3-95: Initiate DMA to/from 32-bit Effective Address

ls	ea	size	tagid	cmd	アセンブリ命令へのマップ
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 Architecture* を参照してください。

表 3-96: Initiate DMA to/from 64-bit Effective Address

ls	eahi	ealow	size	tagid	cmd	アセンブリ命令へのマップ
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、読み出しを実行次第現在の MFC タグステータスを返します。*type* が 1 である場合は読み出しを行なった後に処理が完了していない MFC タグのうち、いずれか MFC タグ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を行なった後にすべての MFC タグの処理が完了するまでブロックします。

表 3-97: Read MFC Tag Status

d	type	アセンブリ命令へのマップ
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   : 15;
  uint64_t size       : 16;
  uint64_t eal        : 32;
} mfc_list_element_t;88
```

`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
```

Example

```
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 Architecture* を参照してください。

表 4-98に MFC DMA コマンドのニーモニックを示します。

表 4-98: 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)
```

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、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_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、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_GETB_CMD))
```

4.4. MFC リスト DMA Commands

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 Architecture* を参照してください。

表 4-99に MFC リスト DMA コマンドのニーモニックを示します。

表 4-99: 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 *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 *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、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_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 *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 *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 *list,
    uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

MFC リストを使用してm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、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_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 *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 Architecture* を参照してください。

表 4-100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 4-100: 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、それをリザーブします。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)
```

`ea` がリザーブされている場合、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)
```

ea がリザーブされているか否かに関わらず、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)
```

ea がリザーブされているか否かに関わらず、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、隣接した未処理の 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 Architecture* を参照してください。

表 4-101に MFC 同期コマンドのニーモニックを示します。

表 4-101: 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、*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。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。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、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_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。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 単体内でのコマンド順位保証のために使用することはお避けください。mfc_eieio はプロセッサおよびデバイス間の同期を保つため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 単体内でのコマンド順位保証のために使用することはお避けください。mfc_eieio はプロセッサおよびデバイス間の同期を保つため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。

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。*mask*（DMA タググループ・クエリマスク）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。

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 4-102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 4-102: 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（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（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 Update Request Status channel

(uint32_t) `mfc_stat_tag_update(void)`

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、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 4-103に示すうちのいずれかとなります。

表 4-103: 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	`mfc_putllc` コマンドの実行に失敗した（DMA リザベーションが消失した）。
2	MFC_PUTLLUC_STATUS	`mfc_putlluc` コマンドの処理が正常に完了している。
4	MFC_GETLLAR_STATUS	`mfc_getllar` コマンドの処理が正常に完了している。

[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 Architecture 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」について、マルチプロセッサ（ユニット）環境における他のプロセッサ（ユニット）からのものを含めたメモリアクセスの順序付けである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)
```

関連付けられた MFC 宛て転送のうち、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 Multisource Synchronization Request 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 Multisource Synchronization Request 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信号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を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 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を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 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 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 Event

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 イベント機構については *Cell Broadband Engine Architecture* を参照してください。

表 4-104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 4-104: 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。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 Architecture* 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)
```

SPU の現在の実行状態を読み出し、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複数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 5-105に示します。

表 5-105: Vector Multimedia Extension 単一トークンデータ型

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プ

5.1.1. データ型

SPU が Vector Multimedia Extension のデータ型をすべてサポートしているとは限りません。Vector Multimedia Extension のデータ型のうち SPU のデータ型へマップされているものを表 5-106に示します。表で網掛けされているのはデータ型が異なる部分です。

表 5-106: Vector Multimedia Extension データ型の SPU データ型へのマップ

Vector Multimedia Extension データ型	SPU データ型へのマップ
vector unsigned char	vector unsigned char
vector unsigned short	vector unsigned short

Vector Multimedia Extension データ型	SPU データ型へのマップ
vector unsigned int	vector unsigned int
vector signed char	vector signed char
vector signed short	vector signed short
vector signed int	vector signed int
vector float	vector float
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` と `vector bool short` は同じデータ型 (`vector unsigned short`)へマップされるため、`vec_unpackh` および `vec_unpackl` の多重定義関数を一意に解決することはできません。

5.1.2. 一対一でマップされた組み込み関数

表 5-107に SPU 組み込み関数へ一対一の関係でマップされた Vector Multimedia Extension 組み込み関数を示します。

表 5-107: SPU 組み込み関数へ一対一の関係でマップされている 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	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

5.1.3. 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 5-108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と考えられます。

表 5-108: 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 5-106で述べたように）ベクタ型 `pixel` および `bool short` を `unsigned short` へマップすると、多重定義関数選択時に競合が発生する。
vec_mfvscr, vec_mtvscr	SPU は IEEE の単精度浮動小数点演算用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、本仕様で定められていない特定のコンパイラをサポートすることが必要になる。

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

5.2.1. データ型

PPU の Vector Multimedia Extension がすべての SPU のデータ型をサポートしているとは限りません。SPU のデータ型のうち PPU の Vector Multimedia Extension データ型へマップされているものを表 5-109に示します。表で網掛けされているのはデータ型が異なる部分です。

表 5-109: SPU データ型の PPU Vector Multimedia Extension データ型へのマップ

SPU データ型	PPU の Vector Multimedia Extension データ型へのマップ
vector unsigned char	vector unsigned char
vector unsigned short	vector unsigned short
vector unsigned int	vector unsigned int
vector signed char	vector signed char
vector signed short	vector signed short
vector signed int	vector signed int
vector float	vector float
vector unsigned long long	vector bool char
vector signed long long	vector bool short
vector double	vector bool int

5.2.2. 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の対応でマップされています。表 5-110にこれらのマッピングを示します。

表 5-110: Vector Multimedia Extension 組み込み関数へ一対一の対応でマップされている 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プ	使用できる型
spu_add	vec_add	vector/vector (スカラは使用不可)
spu_and	vec_and	vector/vector (スカラは使用不可)
spu_andc	vec_andc	全ての型
spu_avg	vec_avg	全ての型
spu_cmpeq	vec_cmpeq	vector/vector (スカラは使用不可)
spu_cmpgt	vec_cmpgt	vector/vector (スカラは使用不可)
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(スカラは使用不可)
spu_nmsub	vec_nmsub	float
spu_nor	vec_nor	全ての型
spu_or	vec_or	vector/vector (スカラは使用不可)
spu_re	vec_re	全ての型
spu_rl	vec_rl	vector/vector (スカラは使用不可)
spu_rsqrte	vec_rsqrte	全ての型
spu_sel	vec_sel	全ての型
spu_sub	vec_sub	vector/vector (スカラは使用不可)
spu_xor	vec_xor	vector/vector (スカラは使用不可)

5.2.3. 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 5-110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と考えられます。

表 5-111: 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は、PPU でサポートすることが容易ではなく、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に基づく命令を要する。
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、 正確なマッピングができるとは限らないこれらの組み込み関数を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. C/C++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てい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。それでもなお、SPU 用に提供する標準ライブラリ関数の実行速度はほとんどの場合において速い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。

6.1. C 標準ライブラリ

本セクションでは、C 標準ライブラリが本仕様に準拠するために最低限満たすべき必要条件について述べます。

6.1.1. 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 6-112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 6-112: 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` マクロはデバッグ関数 `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。表 7-117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	浮動少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“7.1. 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。
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”7.3. 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（要求）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 命令を生かすために、こら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。

ファイル名	説明
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。（“6.1.2. 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。

6.1.2. 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 vector/SIMD multimedia extension 形式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>
                       | <misc-conv>
<char-conv>       ::= 'c'
<str-conv>        ::= 's' | 'P'
<fp-conv>         ::= 'e' | 'E' | 'f' | 'g' | 'G'
<int-conv>        ::= 'd' | 'i' | 'u' | 'p' | 'o' |'x' | 'X'
<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 6-113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 6-113: 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>	'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 f(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 int として印刷されます。
v	<fp-conv>	ベクタは標準 10 進法表記(f or F)または 10 の倍数の累乗指数(e、E、g、G)のいずれかを用いた符号付き 10 進法分数表記での 4 個の 4 バイト要素から成る vector float として印刷されます。
vh or hv	<int-conv>	ベクタは 8 進法(o)、整数(d、i、u)又は 16 進法 f (x、X)表記を用いた 4 個の 4 バイト要素から成る vector unsigned short または vector 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 f (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 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 f (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 long long として印刷されます。
vL or Lv	<fp-conv>	ベクタは標準 10 進法表記(f または F)または 10 の倍数の累乗指数(e、E、g、G)のいずれかを用いた符号付き 10 進法分数表記での 2 個の 8 バイト要素から成る vector double として印刷されます

6.1.3. 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。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。

6.2. C++ライブラリ

本セクションでは、C++標準ライブラリが本仕様に準拠するために最低限満たすべき必要条件について述べます。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 6-114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 6-114: 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	二重連係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。
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 6-115に示します。

表 6-115: 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	浮動少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 6-112: C ライブラリのヘッダファイルを参照してください。

7. SPU 上の浮動小数点演算

C99 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 レジスタへ蓄積します。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、式の計算を含む浮動小数点演算のあらゆる面に影響を与えます。以下のセクションではコンパイラやライブラリにおける相違点が及ぼす個々の影響について説明します。

7.1. 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 7-116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 7-116: 浮動小数点型属性の値

マクロ	値
FLT_DIG	6
FLT_EPSILON	1.19209290E-07
FLT_MANT_DIG	24
FLT_MAX_10_EXP	38
FLT_MAX_EXP	129
FLT_MIN_10_EXP	-37
FLT_MIN_EXP	-125
FLT_MIN	1.17549435E-38
FLT_MAX	6.80564694E+38
DBL_DIG	15
DBL_EPSILON	2.2204460492503131E-016
DBL_MANT_DIG	53
DBL_MAX	1.7976931348623157E+308
DBL_MIN	2.2250738585072014E-308
DBL_MAX_10_EXP	308
DBL_MIN_10_EXP	-307
DBL_MAX_EXP	1024
DBL_MIN_EXP	-1021
FLT_ROUNDS	1 (最近値)に初期化
`FLT_EVAL_METHOD`	0 (拡張は行わない)
FLT_RADIX	2
DECIMAL_DIG	17

7.2. 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有向の丸め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。

7.2.1. 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間において異なる丸めモードを使用することをサポートしています。SPU では単精度用には「0 方向への丸め」モードを用い、倍精度用のデフォルトは「最近値への丸め」モードを用います。

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します。(表 7-116: 浮動小数点型属性の値参照)。

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7.3.2. 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。

7.2.2. 浮動小数点例外

表 7-117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割り込みを起こすことはありません。

表 7-117: 浮動小数点例外用のマクロ

マクロ	コメント
FE_OVERFLOW_SNGL	単精度浮動小数点例外用（定義されている場合）
FE_OVERFLOW_DBL	倍精度浮動小数点例外用
FE_UNDERFLOW_SNGL	単精度浮動小数点例外用（定義されている場合）
FE_UNDERFLOW_DBL	倍精度浮動小数点例外用
FE_INEXACT	ISO/IEC 定義に準拠
FE_INVALID	ISO/IEC 定義に準拠
FE_NC_NAN	不適合の NaN（単精度浮動小数点出力として使用）
FE_NC_DENORM	不適合の非正規化数（単精度浮動小数点出力として使用）
FE_DIFF_SNGL	単精度浮動小数点例外用
FE_ALL_EXCEPT_DBL	上記の全倍精度浮動小数点例外の論理和
FE_ALL_EXCEPT	上記の全浮動小数点例外の論理和

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処理を行ないます。

7.2.3. math.h で定義されているその他の浮動小数点定数

math.h 内にはいくつかの付加的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 7-118にこれらの特別な定数を示します。

表 7-118: 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 へそれぞれ拡張する。
math_errhandling	この定数は int 型および MATH_ERRNO と MATH_ERREXCEPT の値あるいはそれらの論理和をもつ式へ拡張する。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。

7.3. 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。

7.3.1. 浮動小数点演算

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を含む値への変換は、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に収まる。この値の範囲については表 7-116: 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に収まる。
- 整数を浮動小数点数へ倍精度変換すると、DBL_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。

- 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。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収まるものとします。

7.3.2. 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数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。
- 関数の引数として非正規化数を与えた場合は 0 として扱われ、関数はアンダーフローフラグをセットし、+0 を返す。
- 式は「0 方向への丸め」モードで計算される。アルゴリズムへの忠実性を保つために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値が返るため、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 準拠のシステムでは無限大が生成されるようなケースで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) 結果として正負に関わらず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割り込みを発生させ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-math* 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。

- 演算のとして NaN が返される場合は常にクワイエット型の QNaN である。
- 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「最近値への丸め」とする。

- コンパイラは FP_CONTRACT プラグマや *fast-math* コンパイラオプションにより明示的に要求されない限り、Double Floating Multiply and Add (dfma)や Double Floating Multiply and Add (dfma)のような縮約演算を使用しない。縮約演算を使用する場合、ERRNO をセットする必要はない。

7.3.3. 浮動小数点式における特殊なケース

C99 規格にあ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値についてのみ有効であり、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`
  次の二つの倍精度のケースを除き、有効。1） x が SNaN かデフォルト以外の QNaN である場合、結果はデフォルトの QNaN とする。2） `x` が非正規化数である場合、演算では入力された値を適切な符号付きの 0 とする。
- `x/x -> 1.0`
  単精度の場合、`x` が 0 のとき無効。倍精度の場合 `x` が `0`、無限大、 `NaN` のいずれかである場合に無効。
- `x-y -> -(y-x)`
  結果として 0 が生成される場合は常に`+0` であるため、単精度の場合について有効。倍精度の場合、等価的であるとみなすことはできない。DFMS で `x-y` を生成し、DFNMS で`-(y-x)` を生成した結果が NaN ではない場合、その式は有効だが、`x-y` と `y-x` を同じ種類の演算で生成した結果が 0 の場合は互いに異なる符号を持つ可能性があり、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 以内の誤差が生じる可能性がある。
- `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、`-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であるときに無効。倍精度の場合、「最近値への丸め」「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` が成り立つとき、また、`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`
  単精度の場合常に有効。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、結果は非正規化数のオペランド `x` の場合のそれぞれの 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` また `-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` は常に比較不能な値を比較するため、`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` は常に比較不能な値を比較するため、`x==x` が偽となる。

- `x<y -> isless(x,y)`,
  `x<=y -> islessequal(x,y)`,
  `x>y -> isgreater(x,y)`, and
  `x>=y -> isgreaterequal(x,y)`
  倍精度の場合に `x` か `y` が `NaN` であることの影響でフラグがセットされる場合を除いて有効。このような不正なフラグの挙動は FENV_ACCESS プラグマにより変更が可能。

7.3.4. 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別な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。
- `isnanf()` マクロは常に偽を返す。
- 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、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。
- `nextafter(subnormal,y)` 関数がアンダーフローフラグを立てることはない。`nextafter()` 関数や `nexttoward()` 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)`, `atanf(±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)`, `frexpr(±inf,&exp)`, `ldexpf(±inf,ex)`, `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。 acos(|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), and 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明示的でない変換の挙動については7.3.1. 浮動小数点演算 で説明している。

以上