SONY

SPU Application Binary Interface 仕様書

Version 1.6

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

2006 年 12 月 4 日

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

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

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

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

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

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

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

2006年 12月 4日

SONY

目次

SONY

表目次

図目次

本ドキュメントに関して

本ドキュメントは、Synergistic Processor Unit（SPU）の Application Binary Interface（ABI）を定義する。

対象とする読者

本ドキュメントは、Cell Broadband Engine™ Architecture (CBEA) に準拠するプロセッサの SPU に対する言語処理系およびその他ソフトウェアを開発するシステム・プログラマおよびアプリケーション・プログラマを対象とする。

変更履歴

本セクションでは、このドキュメントの各バージョンで SPU ABI に実施された重要な変更を説明する。

バージョン番号および日付	変更点
v. 1.6 2006 年 12 月 4 日	レジスタ保存関数サンプル中のラベルの配置を修正し、レジスタ・リストア関数サンプル中の欠落ラベルを追加（「TWG_RFC00084-0」による。） stdarg.h で宣言される並び中の va_start マクロの綴りを訂正、va_copy を追加（「TWG_RFC00085-0」による。） 再配置を実際の実装と合致するように変更し、2 つの新たな再配置を追加（「TWG_RFC00088-0」による。）
v. 1.5 (corrected) 2006 年 10 月 11 日	「TWG_RFC00069-1」を適用。 SPUNAME 記述子のサイズを固定長（32 バイト）から 4 の倍数の可変長に変更（「TWG_RFC00048-0」による。） `e_machine` SPU ELF ヘッダ・フィールドに対するコメントを改訂し、23 が値として正式に受理された事実を反映。 編集上の各種変更を実施。 以下のリクエストにて行われる変更を適用：TWG_RFC00063-2、TWG_RFC00064-0、TWG_RFC00065-1、TWG_RFC00070-1、TWG_RFC00080-0。
v.1.4 2005 年 10 月 20 日	メモリ・ヒープの初期化およびスタック管理の標準プロセスを定義。（「TWG RFC 00024-3」による。） スタック・フレームに適用されるルールを説明しているセクションを変更。（「TWG RFC 00030-0」による。） 「Broadband Processor Architecture」を「Cell Broadband Engine Architecture」へ、「BPA」を「CBEA」へ変更。（「TWG RFC 00037-0: CORRECTION NOTICE」による。） ローカル・ストレージにロードされる、割り当て可能な ELF セクションに適用される制限事項を追加。（「TWG RFC 00038-2」および「TWG RFC 00044-0」の改訂による。） 環境ポインタを使用する言語において R2 レジスタが環境ポインタとして使用されることを記述。（「TWG RFC 00039-0」による。） 一部の記述誤りを訂正。（「TWG RFC 00041-0: CORRECTION NOTICE」および「TWG RFC 00045-0: CORRECTION NOTICE」による。）
v.1.3 2005 年 8 月 1 日	「本ドキュメントに関して」の一部セクションを削除。 「再配置型」の表において、「`R_SPU_ADDR7`」のフィールドエントリを「`I7*`」から「`I7`」へ変更するなど、一部の記述誤りを訂正。 （以上すべての変更は、「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.1.2 2005年6月10日	「Broadband Engine」または「BE」を「Broadband Processor Architecture に準拠するプロセッサ」または「BPA に準拠するプロセッサ」に変更し、「Synergistic Processing Unit」を「Synergistic Processor Unit」に変更。主な例に関して、「PPU」を「PowerPC Processor Unit」として定義。いくつかの参照文献を訂正し、商標権の所有者を明確にするため、コピーライトのページを追加。（すべての変更は、「TWG RFC 00031-0: CORRECTION NOTICE」による。） 「本ドキュメントに関して」のセクションに対するその他の変更。
v.1.1 2005 年 5 月 9 日	PU を PPU に変更（TWG RFC 00028-0: CORRECTION NOTICE）。
v.1.0 2004 年 11 月 17 日	すべての SPU ELF 実行可能プログラムに、PT_NOTE セクションを追加（TWG RFC 00019-0）。 スタック・レイアウトを修正して、パラメータ・リスト領域の最小スペースに対する要求を除去し、引数の受け渡しおよび戻り値に使用するレジスタ数を増やした（TWG RFC 00020-0）。
v. 0.9 2004 年 6 月 16 日	`.bss` SPU 特別セクションの記述を変更。この記述は現在、プログラム・ローダーによる `.bss` の初期化を指定する（TWG RFC 00001-2）。 汎用レジスタ規約を変更し、揮発性および非揮発性レジスタの間の再割り当てを反映する。具体的には、非揮発性レジスタ数を減少させた。この変更により、何点かの図（TWG RFC 00004-5）も変更。 その他編集上の変更。
v. 0.8 2004 年 3 月 12 日	TWG RFC 00006 で要求されるように、グローバル・データ型は常に 16 バイトの境界でアラインする必要があることを追記。 その他編集上の変更。
v. 0.7 2004 年 2 月 25 日	ixページに記載されるタイポグラフィック規約を反映するため、ドキュメント・フォーマットを変更。その他編集上の変更。
v. 0.6 2004 年 1 月 23 日	前付を含め、ドキュメントを新しいフォーマットに変更。 その他編集上の変更。
v. 0.5 2003 年 9 月 15 日	`R_SPU_ADDR10I` 再配置型を追加。SPU プラグインをサポートするため、SPUNAME セクションのメモを追加。ELF ヘッダー・フィールド `e_type` の記述を追加。
v. 0.4 2003 年 6 月 15 日	スタック初期化およびオーバーフロー検出のメカニズムを変更。追加の再配置型 `R_SPU_ADDR7`、`R_SPU_REL9`、および `R_SPU_REL9I` を追加。SPU オブジェクトの PU/PPC オブジェクトへの組み込みに使用する SPU 環境メモを記述するプログラム・ヘッダー・セクションを追加。
v. 0.3 2003 年 3 月 7 日	構造体のパディング・サンプルを提供。揮発性に関する規約を追加。TOC レジスタとその使用方法の記述を削除。レジスタの保存/リストアのサンプル機能を編集。規約をマングルするシンボル名を明記。
v. 0.2 2002 年 11 月 21 日	レジスタ・レイアウト図を追加。モジュール外関数の呼び出しシーケンスを追加。パラメータ・リスト領域は、少なくとも 8 クワッドワード分に等しくあるべきことを明示。パラメータ受け渡し規約のサンプルを提供。割り込み処理のサポート追加を開始。レジスタの保存/リストアおよび関数呼び出しのコードのサンプルを編集。
v. 0.1 2002 年 9 月 30 日	本ドキュメントの初リリース。

関連文献

下記の表に、本ドキュメントのための参照資料と補足資料のリストを示す。

ドキュメント・タイトル	バージョン	リリース日
Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification	1.2	1995 年 5 月
Tool Interface Standard (TIS), DWARF Debugging Information Format Specification	2.0	1995 年 5 月

ドキュメント構成

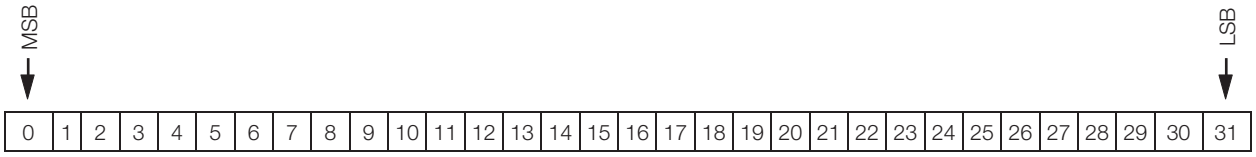
本ドキュメントは、下記の主セクションで構成される。

1. はじめに
2. 低水準システムの情報
3. オブジェクト・ファイル
4. プログラム・ロードおよび動的リンク

本ドキュメントで使用するビット表記法およびタイポグラフィック規約

ビット表記法

本ドキュメントを通して、標準のビット表記法を使用する。ビットおよびバイトは、左から右に昇順してナンバリングする。よって、下記図に示す通り、4 バイト・ワードの場合は、ビット 0 が最上位ビット（MSB）であり、ビット 31 が最下位ビット（LSB）である。



MSB = Most significant bit（最上位ビット）

LSB = Least significant bit（最下位ビット）

ビット・エンコードに対する表記は、下記の通りである。

- 16 進値は、`0x` が前に付く。例えば、`0x0A00` と表記される。
- 文章中の２進値は、シングル・クオテーションで囲まれる。例えば、‘1010’ と表記される。

その他のタイポグラフィック規約

ビット表記の他にも、本ドキュメントを通して下記のタイポグラフィック規約が使用される。

規約	意味
`courier`	プログラミング・コード、処理命令、レジスタ名、データ型、イベント、ファイル名、およびその他リテラルを示す。また、関数名およびマクロ名を示す。これらの規約は、特に文章による説明において、理解を容易にするためにのみ使用する。
`courier` + 斜字体	`const` 型の変数を含む、引数、パラメータおよび変数を示す。これらの規約は、特に文章による説明において、理解を容易にするためにのみ使用する。
斜字体 *（courier なし）*	強調を示す。ハイパーリンクされているものを除き、参照文献は斜字体で表記する。用語を初めて定義する場合に、その用語を斜字体で表現する。

規約	意味
青	ハイパーリンクを示す（カラープリントまたはオンラインのみ）。

1. はじめに

SPU Application Binary Interface は、コンパイル済みのアプリケーション・プログラムに対するシステム・インターフェースを定義する。本インターフェースは、CBEA 準拠システムの Synergistic Processer Unit 上で、これらのプログラムを再コンパイルまたは再コーディングすることなしに実行することを可能にする。本ドキュメントの目的は、バイナリ・インターフェース仕様のセットを標準化して可搬性を高めることである。

本ドキュメントは、低水準の言語バインディング規約を定義する。これらの規約は、C プログラミング言語を使用して説明されるが、その他の言語の使用を除外するものではない。

2. 低水準システム情報

本章では、言語処理系が従うべきルールを説明する。これらのルールに従うことによって、言語処理系は下記の目的を達成することができる。

- 関数呼び出しシーケンスのための規格準拠コードを生成する。引数の受け渡し、値の戻し、およびレジスタの使用も含む。
- 異なるソース言語で記述されたコード・モジュールから、プログラムのグローバル・データへのアクセスを許可する。（共通のデータ型のルールのみを定義する。）
- 異なるベンダーの言語処理系で生成されたオブジェクト・モジュールを混合する。

2.1. データ表現

2.1.1. バイト順

SPU アーキテクチャでは、下記のマシン・データ型を定義する。

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

バイト順は、ハーフワード、ワード、ダブルワード、およびクワッドワードを形成するバイトの、メモリにおける順番を定義する。SPU は、最上位バイト（MSB）順をサポートする。MSB 順は、「ビッグ・エンディアン」とも呼ばれ、最上位バイトは、ストレージ・ユニットの最も若いアドレスのバイト位置（byte 0）に置かれる。

図 2-1から 図 2-4に、各種の幅のストレージ・ユニットにおける、ビットおよびバイトのナンバリングの規約を示す。これらの規約は、整数データおよび浮動小数点データ（最上位バイトにその符号と、最低でも指数部の最初の部分を保持する）の両方に適用される。以下の図では、上にバイト番号を、下にビット番号を示す。

図 2-1：ハーフワードのビットおよびバイト番号順

0 *msb*	*1* *lsb*
0 7	8 15

図 2-2：ワードのビットおよびバイト番号順

0 *msb*	*1*	*2*	*3* *lsb*
0 7	8 15	16 23	24 31

図 2-3：ダブルワードのビットおよびバイト番号順

0 *msb*	*1*	*2*	*3*	*4*	*5*	*6*	*7* *lsb*
0 7	8 15	16 23	24 31	32 39	40 47	48 55	56 63

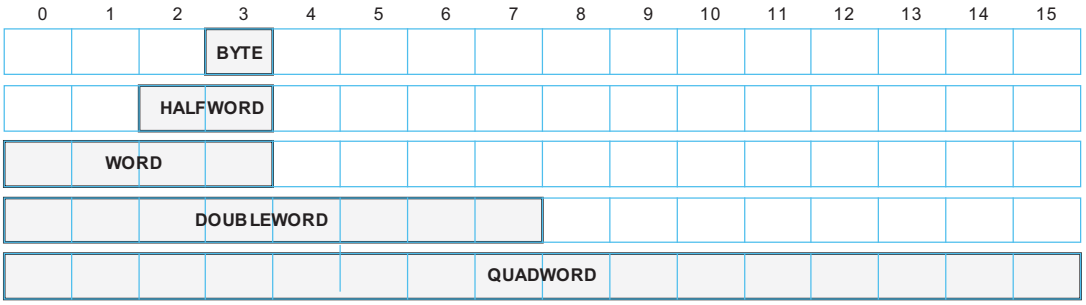
図 2-4：クワッドワードのビットおよびバイト番号順

0 *msb*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15* *lsb*
0 7	8 15	16 23	24 31	32 39	40 47	48 55	56 63	64 71	72 79	80 87	88 95	96 103	104 111	112 119	120 127

2.1.2. レジスタ・レイアウト

汎用レジスタは、128 ビット幅である。128 ビットよりも小さいデータ型は、「プリファード・スロット」と呼ぶ、レジスタ内の指定された位置に置く。図 2-5は、汎用レジスタにおけるデータ型のレイアウトを図示したものである。

図 2-5：データ型のレジスタ・レイアウト



2.1.3. 基本型

表 2-1に、標準 C データ型、そのサイズとアラインメント、および対応する SPU マシン・データ型を示す。グローバル変数は、データ型に関わらず、常に 16 バイトの境界でアラインする必要がある。グローバル変数を 16 バイトにアラインすると、使用するデータ・メモリを増大させるが、16 バイトにアラインすることによって、より少ない命令でより速くデータにアクセスすることが可能になる。従って、増大するデータ・メモリの使用量に対して埋め合わせできる。

表 2-1：基本データ型

型	C 型	サイズ	アラインメント（バイト）	SPU マシン・データ型
文字	char unsigned char	1	1	符号なしバイト
	signed char	1	1	符号付きバイト
	short signed short	2	2	符号付きハーフワード
	unsigned short	2	2	符号なしハーフワード
整数	_Bool	1	1	符号なしバイト
	int signed int long int signed long enum	4	4	符号付きワード
	unsigned int unsigned long	4	4	符号なしワード
	long long signed long long	8	8	符号付きダブルワード
	unsigned long long	8	8	符号なしダブルワード
ポインタ	any type * any type (*) ( )	4	4	符号なしワード
浮動小数点	float	4	4	単精度
	double	8	8	倍精度
	long double	8	8	倍精度
ベクトル	any type	16	16	クワッドワード

注意：この ABI では、IEEE 754 拡張倍精度（128 ビット）浮動小数点は規定しない。この IEEE 標準フォーマットを使用するプログラムは、ABI との整合性を持たない。また、この ABI を実装するプラットフォームはこれらのプログラムをサポートする必要はない。拡張倍精度の浮動小数点をサポートするプラットフォームの場合は、符号ビット、16383 のバイアスを持つ 15 ビットの指数、および「暗黙」ビットを先頭に持つ 112 の小数部ビットを実装する必要がある。アラインメントは 16 バイトでなければならない。

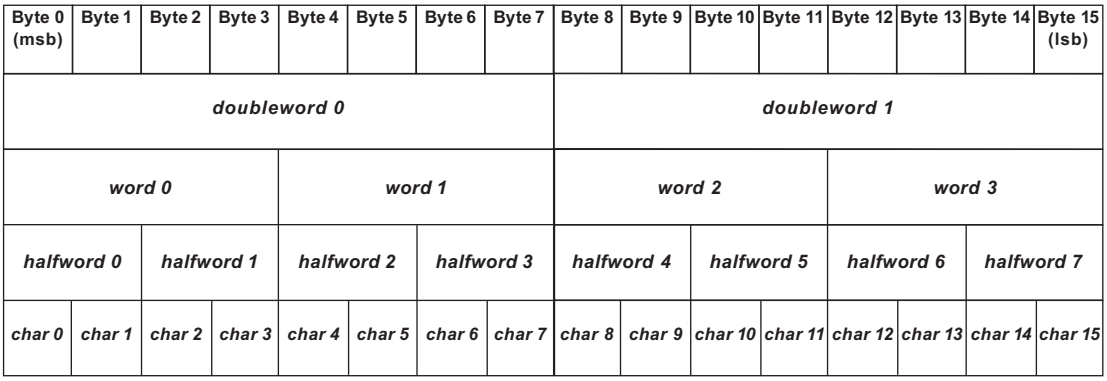
SPU は、いくつかのベクトル・データ型をサポートする。すべてのベクトル型は、128 ビットであり、複数のスカラー要素を持つ。表 2-2で、対応するベクトル型を説明する。

表 2-2：ベクトル型

ベクトル・データ型	説明
qword	未指定型の 128 ビット・クワッドワード・ベクトル
vector unsigned char	8 ビット符号なし整数文字（バイト）が 16 個
vector signed char	8 ビット符号付き整数文字（バイト）が 16 個
vector unsigned short	16 ビット符号なし整数ハーフワードが 8 個
vector signed short	16 ビット符号付き整数ハーフワードが 8 個
vector unsigned int	32 ビット符号なし整数ワードが 4 個
vector signed int	32 ビット符号付き整数ワードが 4 個
vector unsigned long long	64 ビット符号なし整数ダブルワードが 2 個
vector signed long long	64 ビット符号付き整数ダブルワードが 2 個
vector float	32 ビット単精度浮動小数点が 4 個
vector double	64 ビット倍精度浮動小数点が 2 個

図 2-6に示す通り、ベクトルおよびベクトル要素は、MSB 順を使用する。

図 2-6：ベクトル・データ型のバイト順および要素の番号付け



2.1.4. 集成体および共用体

集成体（構造体か配列かに関わらず）と共用体は、最も厳格にアラインされるコンポーネント（最大のアラインメントを持つコンポーネント）のアラインメントを仮定する。集成体と共用体を含む、すべてのオブジェクトのサイズは、常にそのオブジェクトのアラインメントの倍数になる。

配列は、自己の要素と同一のアラインメントを使用する。構造体および共用体オブジェクトは、下記の基準に従い、サイズとアラインメントの制限を満たすためにパディングが要求される場合がある。

- 構造体全体あるいは共用体オブジェクト全体は、最も厳格にアラインされるメンバと同一の境界でアラインする。
- 各メンバは、適切なアラインメントを持つ、最小の使用可能なオフセットを割り当てられる。前のメンバによっては、内部パディングが要求される場合もある。

- 必要であれば、構造体のサイズを増やして、そのアラインメントの倍数にする。この場合は、最後のメンバによっては、末尾のパディングを必要とする場合もある。

構造体アクセスの効率を向上させるため、コンパイラで一番外側の構造体に対し更に制限を設け、クワッドワードのアラインメントを達成してもよい。

図 2-7から 図 2-11は、上記の各アラインメント規約を示すものである。

図 2-7：ワードよりも小さい構造体

図 2-8：パディングなしの構造体

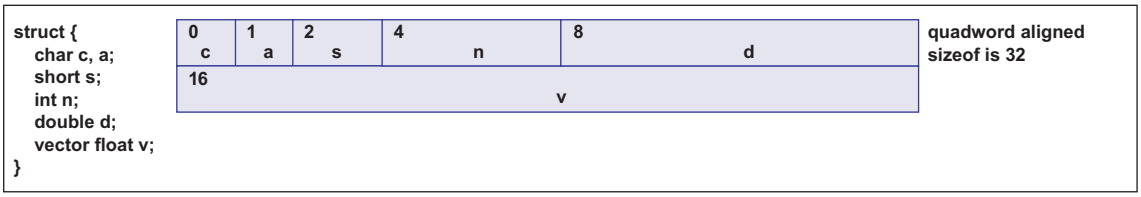


図 2-9：内部パディングを持つ構造体

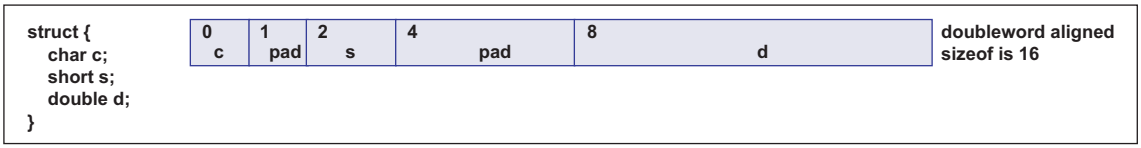


図 2-10：内部および末尾にパディングを持つ構造体

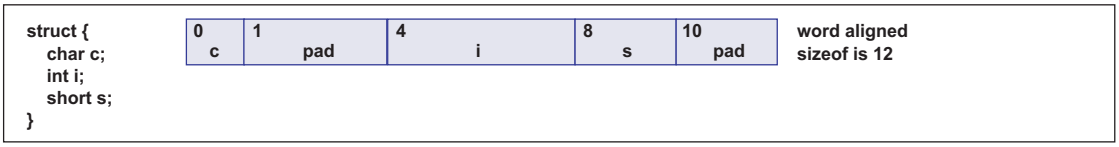
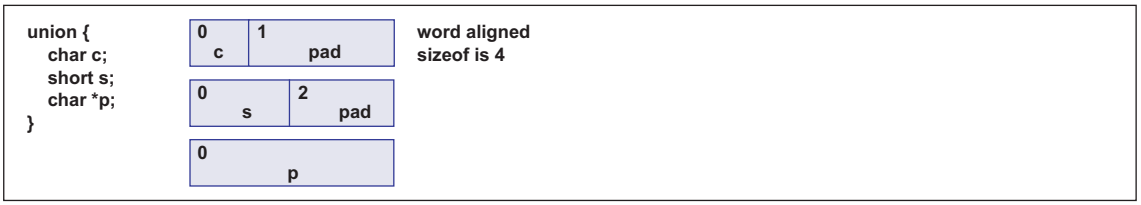


図 2-11：共用体割り当て



2.1.5. ビット・フィールド

C の構造体または共用体は、「ビット・フィールド」を持つ場合がある。「ビット・フィールド」は、指定された数のビットを持つ整数オブジェクトを定義する。「プレーン」のビット・フィールド（符号付きでも符号なしでもない）は、常に非負値を持つ。ビット・フィールドの型は、（負の値も持てる型である）`short`、`int`、`long`、または `long long` にしてよいが、これらの型のビット・フィールドは、同一のサイズを持った対応する符号なし型のビット・フィールドと同じ範囲を持つ。

その他の構造体および共用体メンバと同一のサイズとアラインメント・ルールを使用し、さらに以下の項目を追加する。

- ビット・フィールドは、左から右に、つまり最上位ビットから最下位ビットに割り当てられる。

- ビット・フィールドは、宣言されたデータ型に対して適切なストレージ・ユニットに完全に置かれる必要がある。よって、ビット・フィールドは自身のユニット境界を越えない。
- ストレージ・ユニット内に十分なスペースがある時かつその時に限り、ビット・フィールドはその他の構造体および共用体メンバ（ビット・フィールドまたはそれ以外）とそのストレージ・ユニットを共用しなければならない。
- 名前なしのビット・フィールドのデータ型は、各ビット・フィールドのメンバ・オフセットがアラインメントの制限を遵守するにもかかわらず、構造体または共用体のアラインメントに影響しない。名前なしの 0 幅のビット・フィールドが使われたら、自身のデータ型に対応するストレージ・ユニットに他のメンバ（ビット・フィールドかそれ以外の種類のメンバかに関わらず）が置かれることを回避する必要がある。

表 2-3に、対応する各ビット・フィールドのデータ型の幅と範囲を示す。

表 2-3：ビット・フィールド範囲

ビット・フィールドのデータ型	幅（w）	範囲
signed char	1 ～ 8	-2^{w-1} ～ $2^{w-1}-1$
char		0 ～ $2^{w}-1$
unsigned char		0 ～ $2^{w}-1$
signed short	1 ～ 16	-2^{w-1} ～ $2^{w-1}-1$
short		0 ～ $2^{w}-1$
unsigned short		0 ～ $2^{w}-1$
signed int	1 ～ 32	-2^{w-1} ～ $2^{w-1}-1$
int		0 ～ $2^{w}-1$
enum		0 ～ $2^{w}-1$
unsigned int		0 ～ $2^{w}-1$
signed long		-2^{w-1} ～ $2^{w-1}-1$
long		0 ～ $2^{w}-1$
unsigned long		0 ～ $2^{w}-1$
signed long long	1 ～ 64	-2^{w-1} ～ $2^{w-1}-1$
long long		0 ～ $2^{w}-1$
unsigned long long		0 ～ $2^{w}-1$

2.1.6. 揮発性

SPU プロセッサは、クワッドワードのデータ・アクセスのみをサポートする。正しい揮発性セマンティクスを達成するため、揮発性（volatile）の修飾子付き変数を単独でクワッドワードに置く必要がある。このセマンティクスを遵守するのは、プログラマの責任である。ABI では、揮発性の修飾子付き変数のアラインメントまたは割り当てに対してその他特定のルールを規定しない。

2.2. 関数呼び出しシーケンス

本セクションでは、スタック・フレームのレイアウト、レジスタの用途、および引数の受け渡しを含む、標準の関数呼び出しシーケンスを説明する。

注意：標準の関数呼び出しシーケンス要件は、グローバル関数にのみ適用される。別のコンパイル単位から呼び出せないローカル関数には、異なる規約を使用してもよい。しかし、標準でない関数呼び出しシーケンスの使用は推奨しない。

2.2.1. レジスタ

SPU は、128 個の汎用レジスタを持つ。これらのレジスタは、それぞれ 128 ビットである。表 2-4で、これらのレジスタの種類および用途を説明する。

表 2-4：汎用レジスタ・ルール

レジスタ	種類	用途
R0（LR）	専用	戻りアドレス / リンク・レジスタ。このレジスタは、呼び出された関数が通常戻るアドレスを持つ。関数呼び出しにわたって揮発性であり、非リーフ関数はこれを保存しなければならない。
R1（SP）	専用	スタック・ポインタ情報。SPレジスタのワード要素 0 は、現在のスタック・ポインタを持つ。スタック・ポインタは常に、16 バイトでアラインされなければならない。また、割り当てられた有効なスタック・フレームで最下位のものを指し示し、低いアドレスに向かって伸びていかなければならない。スタック・フレーム・アドレスにあるワードの内容は常に、前回割り当てられたスタック・フレームを指し示す。SPレジスタのワード要素 1 は「利用可能スタック空間」のバイト数を持つ。詳細に関しては、「2.2.2. スタック・フレーム」を参照のこと。
R2	揮発性	環境ポインタ。このレジスタは環境ポインタを必要とする言語において環境ポインタとして使用される。
R3 ～ R74	揮発性	関数の引数リストと戻り値の最初の 72 クワッドワード。
R75 ～ R79	揮発性	作業用レジスタ
R80 ～ R127	非揮発性	ローカル変数レジスタ。これらのレジスタは、関数呼び出しにわたって保存されなければならない。

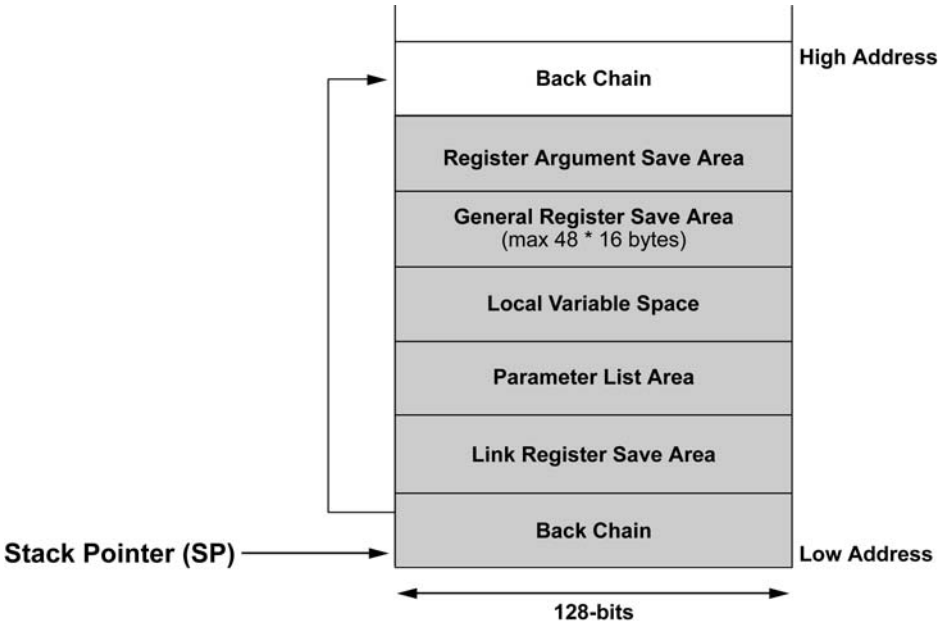
R0、R2、および R3 から R79 のレジスタは、揮発性である。つまり、これらのレジスタ中の値は、関数呼び出しにわたって保存されない。R0、および R75 から R79 のレジスタ中の値は、関数呼び出しシーケンスの期間中でさえも保存されない可能性がある。よって、関数においては、これらのレジスタに呼び出し側で入れたものと同じ値があることを期待してはならない。

R1 および R80 から R127 のレジスタは、非揮発性である。呼び出された関数は、これらのレジスタの値を変更する前に保存する必要がある。また、戻る前に前回の値をこれらのレジスタにリストアしなければならない。

2.2.2. スタック・フレーム

レジスタの使用に加えて、各関数呼び出しは、ランタイム・スタック上にスタック・フレームを持つ場合がある。ランタイム・スタックは、高いアドレスから下に向かって伸びていく。図 2-12に、スタック・フレームの構成を示す。この図で *SP* は、呼び出された関数のスタック・ポインタ（汎用レジスタ `R1` のワード要素 `0`）を意味する。このポインタは、関数がスタック・フレームを生成するコードを実行した後のものである。

図 2-12：標準スタック・フレーム



以下の要件を、スタック・フレームに適用する。

- スタック・ポインタは、16 バイト（クワッドワード）のアラインメントを維持しなければならない。
- スタック・ポインタは、割り当てられた最下位のスタック・フレームの最初のワード、すなわち「Back Chain」を指し示さなければならない。スタックは、下に向かって（低いアドレスの方向に）伸びていかなければならない。スタック・フレームの最初のワードは常に、前回割り当てられたスタック・フレーム（高いアドレスの方向）を指し示さなければならない。ただし、最初のスタック・フレームでは、0（NULL）の Back Chain ポインタを持たなければならない。
- スタック・ポインタが必要な場合、呼び出された関数で「スタック・ポインタ情報」レジスタ（SP）の全ワード要素をディクリメントし、戻る前にリストアしなければならない。
- スタック・ポインタとオフセットを用いてメモリへの格納を行う場合には、-2000（-125 * 16）より小さいオフセットを使用してはならない。これによって割り込みハンドラは最初にスタックポインタに-2000 を加算して、アプリケーションスタックを利用できるようになる。
- スタック・フレームが割り当てられた時に、ディクリメントされたスタック・ポインタ情報レジスタの「利用可能スタック空間」のワード（`R1` のワード要素 1）を評価することによって、スタックオーバーフローをテストすることができる。「利用可能スタック空間」のワードが負の場合には、オーバーフローが検出され、プログラム実行が停止される。
- 呼び出し側が 72 クワッドワードを超える引数を渡す必要がある場合、この呼び出し側は「パラメータ・リスト領域」を割り当てる必要がある。「2.2.3. 引数の受け渡し」を参照のこと。「パラメータ・リスト領域」を必要とする場合、この領域はレジスタにて渡されないすべての引数を包含するのに十分な大きさでなければならない。パラメータ・リスト領域の内容は、関数呼び出しにわたって保存されない。
- 関数がいずれかの非揮発性レジスタの値を変更する場合、その関数は事前に 128 ビット・レジスタ全体の値を「汎用レジスタ保存領域」のクワッドワードに保存する必要がある。

- その他の領域は、コンパイラとコンパイルされているコードによって異なる。標準の関数呼び出しシーケンスでは、スタック・フレームの最大サイズを定義しない。最小のスタック・フレームは、下記に説明するように、最初の 2 個のクワッドワードで構成される。この呼び出しシーケンスでは、標準スタック・フレームの「ローカル変数空間」の言語による使用方法、およびその大きさに関して制限しない。

スタック・フレームのヘッダは、「Back Chain」のクワッド・ワードと「リンク・レジスタ保存領域」のクワッド・ワードの両方で構成される。これらの 128 ビット・クワッドワードの最上位 32 ビットにはそれぞれ、「Back Chain」へのポインタとリターン・アドレスが格納される。各クワッドワードの残りの 96 ビットはツールチェーンによる使用のために予約されている。

ある関数が別の関数を呼び出す場合、呼び出し側の関数は事前に下記項目を実行する必要がある。

- 関数に入った時のリンク・レジスタの 128 ビットの内容を、その関数の呼び出し側のスタック・フレームの「リンク・レジスタ保存領域」に保存する。
- 自身のスタック・フレームを生成する。

スタック・フレームのヘッダを除いて、関数は使用しない領域に対してスペースを割り当てる必要は無い。別の関数を呼び出さず、またスタック・フレームのその他のいずれのパートも必要としない場合、関数はスタック・フレームを生成する必要は無い。

フレーム全体のいずれのパディングも「ローカル変数空間」内でなければならない。「パラメータ・リスト領域」はスタック・フレーム・ヘッダの直後に置かれる必要があり、また「レジスタ保存領域」はパディングを含んではならない。

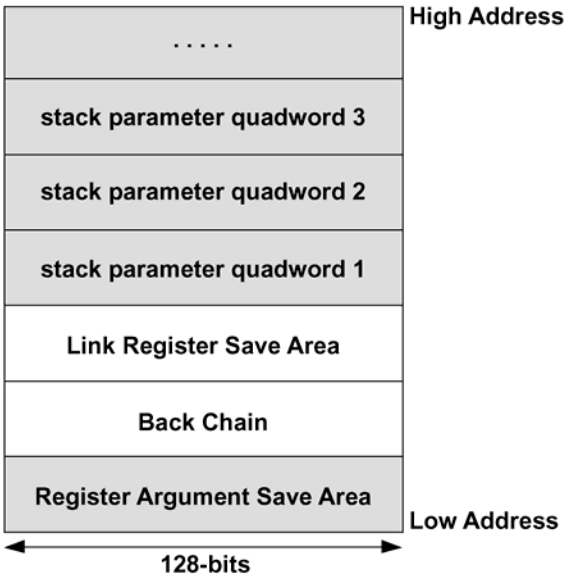
2.2.3. 引数の受け渡し

引数リストをストレージ内に生成する、あるいは引数をスタックにプッシュするよりも、関数引数をレジスタで渡した方がより効率的である。これには、２つの理由がある。1）すべての演算はレジスタ内で実行しなければならない、また 2）呼び出し側がレジスタに引数を算出し、呼び出される関数に同一のレジスタでこれらの引数を渡す場合は、メモリ・トラフィックを排除することができるからである。２番目のケースでは、呼び出された関数は同一のレジスタを使用してその後の演算を行うこともできる。

SPU の場合は、最高 72 個のクワッドワードが汎用レジスタに入れて渡され、レジスタ `R3` から `R74` に順次ロードされる。72 個より少ない引数が要求される場合、不必要なレジスタはロードされず、呼び出される関数に入る時、これらのレジスタに含まれる値はいずれも未定義である。

呼び出される関数に渡される引数がこれら 72 個のレジスタに適合しない場合、呼び出し側の関数は自身の「パラメータ・リスト領域」に、これらの引数に対する追加の空間を 図 2-13に示すように割り当てる必要がある。

図 2-13：パラメータ・リスト領域の配置



下記のアルゴリズムは、C 言語の場合の引数データの受け渡し場所を指定するものである。説明の便宜上、引数は左（最初の引数）から右に順序づけられているものと見なす。実際には、引数の評価順序は未指定である。

- 次のように初期化を行う：*reg* = 3、*stack_arg* = パラメータ・クワッドワード 1 のアドレス
- 各引数について、引数型を決定し、下記ルールに従って格納する。
  - 単純な引数（スカラー、ベクトル、またはオブジェクトへのポインタ）の場合、*reg* が 74 よりも小さいかあるいは等しい場合は、レジスタ *reg* に引数をコピーし、*reg* をインクリメントする。
  - *構造体*または*共用体*の場合、構造体全体が残りの引数レジスタに適合する場合は、構造体のアラインメントに揃った引数のメモリ・イメージを、一度に 16 バイトずつ、引数すべてがコピーされるまで、レジスタに置いていく。適合しない場合は、下記に説明するように、構造体全体をスタックに置く。（「2.1.4. 集成体および共用体」を参照のこと。）
  - 単純でない引数、または 74 よりも大きい *reg* を持つ単純な引数（つまり、上記で処理されない引数）を、呼び出し側のスタック・フレームのパラメータ・クワッドワードに入れて渡す。スタックで渡される値は、レジスタに置かれていた値と同一である。よって、スタックには、レジスタ・イメージが含まれることになる。このスタックの割り当ては、下記を実行して行う。

    (a) *stack_arg* をクワッドワード・アラインメントまでパディングして進め、*stack_arg*, ..., *stack_arg+size-1* に引数をバイト対応で（1 番低いアドレスに置かれたバイトから開始して）コピーする。 ここで *size* は、対象の引数に含まれるバイト数である。

    (b) *stack_arg* を *stack_arg+size* に設定する。
  - 単純な引数を、「2.1.2. レジスタ・レイアウト」で説明したクワッドワードの「プリファード・スロット」に置く。

上記のアライン・アルゴリズムで飛ばされたレジスタおよびワードの内容は、未定義である。

引数受け渡しの例

```
struct {
       int i;
       double d;
       vector unsigned int v[36];
} s, t;
int a, b;
float x, y, z;


x = func(a, x, y, z, s, t, b);
```

表 2-5で示すように、この例では、パラメータはレジスタおよびスタックで渡される。

表 2-5：レジスタ割り当ての例

パラメータ	レジスタ	パラメータ・リスト領域オフセット
a	R3	格納されない
x	R4	格納されない
y	R5	格納されない
z	R6	格納されない
s	R7 - R43	格納されない
t	-	0 - 591
b	-	592 - 607

2.2.4. 可変引数リスト

ANSI C 仕様では、可変引数リストを持つ関数を宣言するときは、末尾の省略記号 (...) を含むプロトタイプの使用が要求される。

通常は移植可能な C プログラムの中には、ある特定の引数受け渡しスキームに依存するものがある。このようなプログラムでは、すべての引数はスタックに渡され、引数はスタック上で昇順に現れることを前提としている。このような前提事項を持つプログラムは、今まで多くの実装で正しく動作していたとしても、やはり真に移植可能なものではない。いずれにせよ、引数のいくつかはレジスタで渡されるので、これらのプログラムは SPU 準拠のコンパイラでは正しく動作しない。

移植可能な C プログラムでは、可変引数リストを管理するため、`va_start`、`va_arg`、`va_end`、および `va_copy` のマクロと、`va_list` の型を使用する。これらのマクロはコンパイラによって定義され、ヘッダ・ファイルの `stdarg.h` に提供される。

可変引数関数の引数は、固定引数関数の引数と同一の方法で受け渡される。「2.2.3. 引数の受け渡し」で説明したように、引数はレジスタ `R3` から最高 `R74` までに、また必要であれば「パラメータ・リスト領域」に置かれる。可変引数関数である呼び出される側の関数は、引数レジスタを自身の「レジスタ引数保存領域」にコピーする。「レジスタ引数保存領域」の相対位置は 図 2-12 中に示されている。

スカラー型の可変引数は、データ型を持たない引数と同一の方法で、呼び出し側の関数によって暗黙的にプロモートされる。文字または short のデータ型を持つ引数は整数にプロモートされ、単精度浮動小数点は倍精度浮動小数点にプロモートされる。その他すべてのデータ型はプロモートされない。

`va_list` 型と、`stdarg.h` で宣言される 可変引数マクロを 図 2-14に示す。

図 2-14：`stdarg.h` の内容

```
/* Aligning the fields makes accessing them faster. */
typedef struct __va_list {
   char *next_arg    __attribute__ ((__aligned__ (16)));
   char *caller_stack    __attribute__ ((__aligned__ (16)));
} va_list;



#define va_start(v,l)      __builtin_va_start(v,l)
#define va_end(v)            /* nothing */
#define va_arg(v,l)         __builtin_va_arg(v,l)
#define va_copy(d,s)       (d) = (s)
```

図 2-14で示す `__builtin_va_start` および`__builtin_va_arg` 関数は、コンパイラ内で実装され、図 2-15で示す擬似コードに従って動作する。

図 2-15：可変引数リストマクロの擬似コード実装

```
__builtin_va_start (AP, LAST)
{
   int paddedsize = (sizeof (LAST) + 15) & -16;

   AP.next_arg = (unsigned char *) &LAST;

   /* get caller's stack pointer */
   AP.caller_stack = __builtin_frame_address (1);
```

```
    if (AP.next_arg + paddedsize > AP.caller_stack      && AP.next_arg <= AP.caller_stack)
      AP.next_arg = AP.caller_stack + 32;
    else
      AP.next_arg += padded_size;
}

TYPE __builtin_va_arg(AP, TYPE)
{
    int padded_size = (sizeof(TYPE) + 15) & -16;
    char *argp;

    /* If this arg overlaps with AP.caller_stack, the
        whole argument must start at the beginning of the caller's
        arguments. */

    if (AP.next_arg + paddedsize > AP.caller_stack
          && AP.next_arg <= AP.caller_stack)
      argp = AP.caller_stack + 32;
    else
      argp = AP.next_arg;
    AP.next_arg = argp + paddedsize;
    return *(TYPE *)argp;
}
```

2.2.5. 戻り値

関数は、スカラー、ベクトル、および集成体と共用体を、レジスタ `r3` から始まるレジスタで戻す必要がある。（スカラーとは、ポインタ、あるいは `char`、`short`、`int`、`enum`、`long int`、`long long`、`float`、または `double` のデータ型のいずれかである。集成体とは、構造体と配列である。）集成体は、レジスタの最上位バイト側、つまり左端のバイトを占有する。

1152 バイトより大きい集成体あるいは共用体は、呼び出し側によって割り当てられたストレージ・バッファに戻さなくてはならない。バッファのアドレスは、隠し引数として `r3` に渡される。このアドレスは、１番目の引数として渡され、引数受け渡しアルゴリズムの *`reg`* は 3 に初期化されずに、代わりに 4 に初期化される。

2.2.6. モジュール外関数呼び出し

原則的に SPU プログラムは静的バインディングである。理由は、すべてのシンボルがリンク・エディタによって完全に解決されるからである。しかし SPU ABI では、「プラグイン」動的バインディングと呼ばれる、ある限られたフォームの動的バインディングが可能である。下記に挙げる特徴は、プラグイン動的バインディングを定義するものである。

- プラグイン・モジュールは、動的な外部参照を持たず、単一のエントリ・ポイントを持つ。
- プラグイン・モジュールは、SPU によってロードされる。プラグイン・モジュールのエントリ・ポイントは、SPU プラグイン・ローダーから関数ポインタとして返される。
- 複数のプラグイン・モジュールが共存してもよい。SPU プログラムの責任で、プラグイン・ストレージ管理を実行する。
- 呼び出し側とプラグイン・コールバック関数間におけるデータ共有は、相互の合意により、プラグインで受け渡しされてもよい。本 ABI では、特定のメカニズムは強制しない。

プラグインを呼び出すと、ポインタによって関数が呼び出される。「2.3.6. ポインタによる関数呼び出し」を参照のこと。

2.3. コーディング例

本セクションでは、関数呼び出し、静的オブジェクトへのアクセス、およびプログラムのあるパートから別のパートへの制御の転送など、基本的な操作に対するコード・シーケンスの例を説明する。これまでのセクションでは、プログラムによるシステムの使用方法、および実行環境に関するプログラムの前提を説明した。本セクションでは、これまでのセクションとは異なり、どのような方法で操作を実行する必要があるかというよりも、どのような方法で操作が実行され得るかを説明する。

本セクションで挙げる例では、ANSI C 言語の規約を使用する。その他のプログラミング言語でも、これらと同一の規約を使用できる。しかしながら、これらの規約を使用しなくても、プログラムを ABI に準拠させることは可能である。

SPU コードは通常、位置に依存しない。つまり、コードは特定のロード・アドレスに依存せず、またローカル・ストレージにおいて多様な位置で適切に実行できる。位置依存のコードを書くことも可能であるが、以下の例では位置に依存しないコードのみを示す。

2.3.1. コード・モデルの概要

SPU プロセッサは、汎用プロセッサと専用プロセッサ間の隙間を埋め、その特有のアーキテクチャの特徴は最も効率的にプログラミングするための技法を促すものである。使用され得る技法には以下がある。1）プラグイン・オブジェクトを使用して、限られたローカル・ストレージ内で大きなプログラムをサポートする、2）コルーチンを使用して、プリエンプティブまたは非プリエンプティブのコンテキスト・スイッチングのオーバーヘッドを負わずにマルチスレッドの同時実行をサポートする。

プラグインなどの技法は、コンパイラが位置に依存しないコードを生成する能力に依存する。位置に依存しないコードには、下記項目が不可欠である。

- 現在のアドレスから相対的なアドレスを保持する、または、転送アドレスを保持するレジスタを使用する制御転送命令。（相対分岐は、現在のアドレスに対して宛先アドレスを算出するのであり、絶対アドレスに対して算出するのではない。）
- 命令に絶対アドレスを埋め込む代わりに実行中に絶対アドレスを算出すること。

これらの条件は、相対的とレジスタ・ベースの両方の、分岐およびロード/ストア命令を提供する、SPU アーキテクチャによって満たされる。

2.3.2. 関数のプロローグおよびエピローグ

本セクションでは、関数のプロローグ・コードおよびエピローグ・コードを説明する。関数プロローグは、必要であれば、スタック・フレームを生成し、また関数が使用する非揮発性レジスタを保存する。関数エピローグは、プロローグ・コードで保存されたレジスタをリストアし、前回のスタック・フレームをリストアし、そして呼び出し側に戻る。

下記のルールを除いて、本 ABI では関数プロローグおよびエピローグの既定のコード・シーケンスを強制しない。それでもやはり、高信頼性コールチェーン・バックトラックを可能にする下記のルールに従う必要がある。

1. 関数が非揮発性の汎用レジスタを使用する場合、その関数は「汎用レジスタ保存領域」にレジスタを保存しなければならない。呼び出し側のスタック・フレームからの負のオフセットを使用することによって、新しいスタック・フレームを生成する前にレジスタを保存することが可能である。スタック・オーバーフローのテストは、「汎用レジスタ保存領域」の非揮発性レジスタへの保存が行われるよりも前に行われなければならない。オーバーフローのテストでは、使用スタックの長さも考慮されなければならない。
2. 関数が別の関数を呼び出す前に、その関数は自身のスタック・フレームを生成する必要がある。このスタック・フレームは、16 バイトの倍数でなければならない。また、この関数に入った時の「リンク・レジスタ（R0）」を、呼び出し側のスタック・フレームの「リンク・レジスタ保存領域」に保存する必要がある。
3. 新規のスタック・フレームを生成すると、必要に応じた負の変位で SP レジスタ（R1）の全ワード要素が調整される。スタック・フレーム・オーバーフローのテストを行ってもよい（このテストは必須ではない）。スタック・ポインタの調整によって、新規のスタック・フレームが生成される前に、実行が停止される。

4. スタック・ポインタ情報レジスタ（R1）のワード要素 1 を検証することによって、新規のスタック・ポインタのスタック・オーバーフローのテストを行ってもよい。オーバーフローが検出された場合、すなわちワード要素 1 が負の値であった場合には、プログラムの実行が停止される。
5. 関数が自身のスタック・フレームを割り当て開放する場合は、(a)「Back Chain」のクワッドワード値を「スタック・ポインタ情報」レジスタ（R1）にロードするか、あるいは(b)ディクリメントされた分スタック・ポインタ情報レジスタの全ワード要素をインクリメントする。

インライン・コードを使用して、関数が使用する非揮発性レジスタの保存、リストアをしてもよい。しかし、保存またはリストアするレジスタが数多くある場合は、「2.3.3. レジスタ保存およびリストア関数」で述べるように、保存およびリストアのサブルーチンを提供し使用する方が効率的な場合もある。

SPU プログラムに入るときには、標準外のプロローグを使用してもよい。また、その時非揮発性レジスタを保存する必要はない。

2.3.3. レジスタ保存およびリストア関数

本セクションでは、レジスタの保存およびリストアのための関数を説明する。これらの関数は、ABI の一部ではない標準外の呼び出し規約を使用する。しかしながら、コンパイラ間での統一を促すために、これらの関数を本ドキュメントに収める。

レジスタ保存およびリストア関数は、レジスタ 127 からレジスタ *r* までの連続する汎用レジスタを保存/リストアする。*r* は、80 から 127 までの 1 個の値を表す。各関数は、影響されるレジスタの数を除いて、同一の動作をする 48 個のサブ関数のファミリを表す。

効率性を向上させるためには、これらの関数に分岐ヒントとノーオペレーション（NOP）を適切に挿入して命令フェッチの長期の停止を避けることも可能である。以下のアルゴリズムで、分岐ヒントを置くのに十分な数の命令が、呼び出し側関数と保存/リストア関数で確実に利用可能になる。

- 保存/リストアされるレジスタ数がある数 *n* よりも少ない場合は、保存およびリストア関数をインラインする。
- 保存/リストアされるレジスタの数が *n* を超える場合は、最初の *n* 個のレジスタをインラインに保存/リストアし、その後保存/リストア関数を呼び出し残りのレジスタを保存/リストアする。

レジスタ保存およびリストアには、２個の関数がある。１つはレジスタを保存するためのサブ関数のファミリで、もう１つはレジスタをリストアするためのサブ関数のファミリである。

- レジスタ保存関数である `_savegpr_n` は、`n` から 127 までのレジスタを保存し、戻る。これらの関数は、`LR` に戻りアドレス、`R75` には調整されたスタック・ポインタ、`SP` にはレジスタ保存領域の上端のアドレスが入っていることを必要とする。スタックオーバーフローをテストするコードを挿入してもよい。
- レジスタ・リストア関数である `_restoregpr_n` は、`n` から 127 までのレジスタをリストアし、戻る。これらの関数は、128 ビットの `LR` が既にリロードされていて、R75 には調整済みのスタック・ポインタが、SP にはレジスタ保存領域の上端のアドレスが入っていることを必要とする。

図 2-16および 図 2-17に、サンプルのプロローグとエピローグからそれぞれ呼び出される、保存関数およびリストア関数の使用方法を示す。

図 2-16：レジスタ保存関数のサンプル

```
# サンプル・プロローグ（レジスタ 94 から 127 を保存する）

                il      $75, <frame_size>
                hbrr    prologue_branch, _savegpr_110
                sf      $75, $75, $SP
                stqd    $LR, 16($SP)
                stqd    $94, -544($SP)
                stqd    $95, -528($SP)
                stqd    $96, -512($SP)
                . . .
                stqd    $108, -320($SP)
```

```
                    stqd    $109, -304($SP)
prologue_branch:    brsl    $LR, _savegpr_110


#   保存関数

_savegpr_80:        stqd    $80, -768($SP)
_savegpr_81:        stqd    $81, -752($SP)
_savegpr_82:        stqd    $82, -736($SP)
                    . . .

_savegpr_110:       stqd    $110, -288($SP)
_savegpr_111:       stqd    $111, -272($SP)
_savegpr_112:       hbr     _save_branch, $LR
                    stqd    $112, -256($SP)
                    stqd    $113, -240($SP)
                    stqd    $114, -224($SP)
                    stqd    $115, -208($SP)
                    . . .

                    stqd    $125, -48($SP)
                    stqd    $126, -32($SP)
                    stqd    $127, -16($SP)
                    lr      $SP, $75
_save_branch:       bi      $LR
```

図 2-17：レジスタ・リストア関数のサンプル

```
# サンプル・エピローグ（レジスタ 94 から 127 をリストアする）

                    il      $75, <frame_size>
                    hbrr    epilogue_branch, _restoregpr_110
                    a       $75, $SP, $75
                    lr      $SP, $75
                    lqd     $94, -544($SP)
                    lqd     $95, -528($SP)
                    lqd     $96, -512($SP)
                    . . .
                    lqd     $108, -320($SP)
                    lqd     $109, -304($SP)
                    lqd     $LR, 16($SP)
epilogue_branch:    br      _restoregpr_110


#   リストア関数

_restoregpr_80:     lqd     $80, -768($SP)
_restoregpr_81:     lqd     $81, -752($SP)
_restoregpr_82:     lqd     $82, -736($SP)
                    . . .
_restoregpr_110:    lqd     $110, -288($SP)
_restoregpr_111:    lqd     $111, -272($SP)
_restoregpr_112:    hbr     _restore_branch,$LR
                    lqd     $112, -256($SP)
                    lqd     $113, -240($SP)
```

```
                        lqd     $114, -224($SP)
                        . . .
                        lqd     $125, -48($SP)
                        lqd     $126, -32($SP)
                        lqd     $127, -16($SP)
                        lr      $SP, $75
_restore_branch:        bi      $LR
```

2.3.4. データ・オブジェクト

本セクションでは、静的なストレージ存続期間を持つオブジェクトを説明する。ここではスタックに常駐するオブジェクトは除外する。なぜなら、プログラムは常にスタック・ポインタまたはフレーム・ポインタ相対で、そのアドレスを算出するからである。

SPU アーキテクチャでは、ロード命令およびストア命令のみがメモリにアクセスする。位置に依存しないコードを維持するために、データ・オブジェクトは`lqr` および`stqr` の相対ロード命令およびストア命令を使用してアクセスされる必要がある。位置に依存しないロードおよびストアを 図 2-18に示す。

図 2-18：位置に依存しないロードおよびストア

C	アセンブリ
extern vector unsigned int src; extern vector unsigned int dst; extern vector unsigned int *ptr;	.extern src .extern dst .extern ptr
	.text
dst = src;	lqr $5, src stqr $5, dst
ptr = &dst;	ila $2, base brsl $3, base base: ila $5, dst sf $3, $2, $3 a $5, $5, $3 stqr $5, ptr
*ptr = src;	lqr $5, ptr lqr $6, src stqd $6, 0($5)

2.3.5. 名前による関数呼び出し

名前付き関数は静的にバインドされる必要があることから、これらの関数の呼び出しアドレスはリンク・エディットの間に解決される。位置に依存しない状態を維持するためには、表 2-6に示すように相対分岐命令を使用する。生成される命令は、相対分岐の距離によって異なる。

表 2-6：相対アドレス指定の名前付き関数呼び出し

距離（バイト）	命令
-128K to 128K-1	brsl $LR, `relative_func_addr`

-128K より小さいか、あるいは 128K-1 より大きい相対アドレス指定の関数呼び出しは、SPU プロセッサの相対アドレス指定可能な範囲内にある「トランポリン」を使用することによってサポートされる。

位置に依存するコードは、表 2-7に示すように、絶対アドレッシングを使用してもよい。生成される命令は、呼び出されている関数のアドレスによって異なる。

表 2-7：絶対アドレス指定の名前付き関数呼び出し

アドレス	命令
0x00000000 to 0x0001FFFF 0xFFFE0000 to 0xFFFFFFFF	brasl $LR, `func_addr`
0x00020000 to 0xFFFDFFFF	ilhu $3, `func_addr`@h iohl $3, `func_addr`@l bisl $LR, $3

関数呼び出し分岐の再配置修正に関しては、「3.5. 再配置」を参照のこと。`func_addr@h` および `func_addr@l` 表記は、関数アドレスの高位と低位を意味する。

2.3.6. ポインタによる関数呼び出し

呼び出されている関数がモジュール外関数、あるいはモジュール内関数かに関わらず、ポインタによる関数呼び出しをサポートするために生成されるコードは同一である。図 2-19は、ポインタによる関数呼び出しの例を示す。

図 2-19：ポインタによる関数呼び出し

```
lqr     $11, func_ptr          # 関数エントリへのポインタをレジスタ 11 にロードする
bisl    $LR, $11               # モジュール外関数を呼び出す
        . . .
```

2.3.7. 動的なスタック空間割り当て

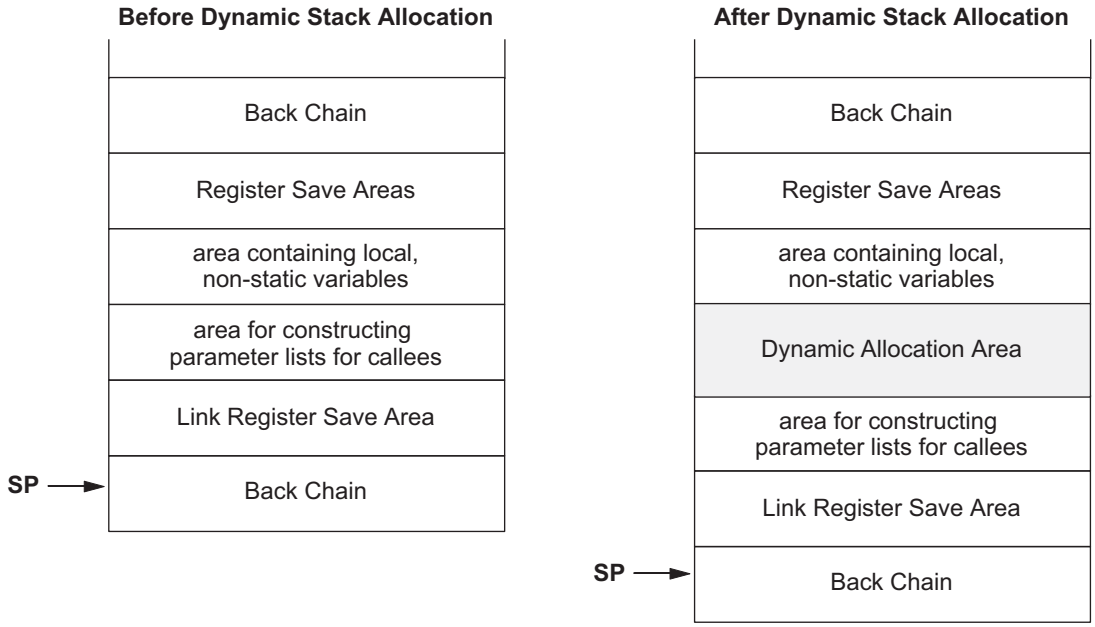
フレームは、プログラム実行中に、プログラム・スタック上に動的に割り当てられる。通常は、個々のスタック・フレームは静的なサイズを持つが、SPU アーキテクチャにおいては、alloca 関数をサポートするため、動的な割り当ての機能が提供される。

動的空間を割り当てるためのメカニズムは、関数の中に完全に組み込まれる。このメカニズムは、標準の呼び出しシーケンスに影響しない。動的なスタック割り当ては、スタックを「パラメータ・リスト領域」のすぐ上で（高いほうのアドレスで）「広げる」ことによって達成される。下記ステップで、詳細なプロセスを説明する。

1. 新たにスタック・フレームを取得した後、また最初に動的空間を割り当てる前に、新しいレジスタ（フレーム・ポインタ）をスタック・ポインタの値に設定する。フレーム・ポインタは、関数の非静的ローカル変数への参照に使用される。
2. 16 バイトのスタック・アラインメントを維持するため、割り当てられる動的空間の量は 16 バイトの倍数に丸められる。
3. スタック・ポインタは丸められたバイト数でディクリメントされ、前回のスタック・フレーム（Back Chain）のアドレスは新しいスタック・ポインタによってアドレス指定されたワードに格納される。

図 2-20に、動的スタック割り当ての前後のスタック・フレームの構成を示す。

図 2-20：動的なスタック空間の割り当て



上記のプロセスは、関数起動 1 回の範囲内で、必要な回数だけ繰り返すことができる。戻り時になると、スタック・ポインタは「Back Chain」の値に設定され、したがって、動的に割り当てられたすべてのスタック空間はスタック・フレームの残りの部分とともに取り除かれる。動的に割り当てられたスタック領域が開放された後は、プログラムはその領域を参照してはならない。

2.4. デバッグ・フォーマット

SPU をターゲットとするオブジェクトで使用されるデバッグ・フォーマットは、「Debug with Arbitrary Record Format（DWARF）」でもよい。本 ABI では特定のデバッグ・フォーマットを指定しないが、DWARF を実装するすべてのシステムは、以下のセクションで説明する定義を使用しなければならない。

2.4.1. DWARF レジスタ・ナンバー・マッピング

SPU レジスタに対して、レジスタ・ナンバー・マッピングを指定する必要がある。表 2-8は、SPU プロセッサに対するレジスタ・ナンバー・マッピングを示す。汎用レジスタのいくつかは、特殊な目的にため予約されており、よっていくつかの省略形を使用してのアクセスが可能である。

表 2-8：SPU レジスタ・ナンバー・マッピング

レジスタ名	ナンバー	省略形
汎用レジスタ 0-127	0 - 127	R0 ～ R127
リンク・レジスタ	0	LR
スタック・ポインタ	1	SP
浮動小数点状態および制御レジスタ	128	FPSCR

2.4.2. アドレス・クラス・コード

DWARF version 2 の仕様では、プロセッサ固有のアドレス・クラス・コードの定義も要求される。表 2-9に示すように、SPU プロセッサはアドレス・クラス・コードを定義する。

表 2-9：SPU アドレス・クラス・コード

コード	値	意味
ADDR_none	0	クラスは指定されない

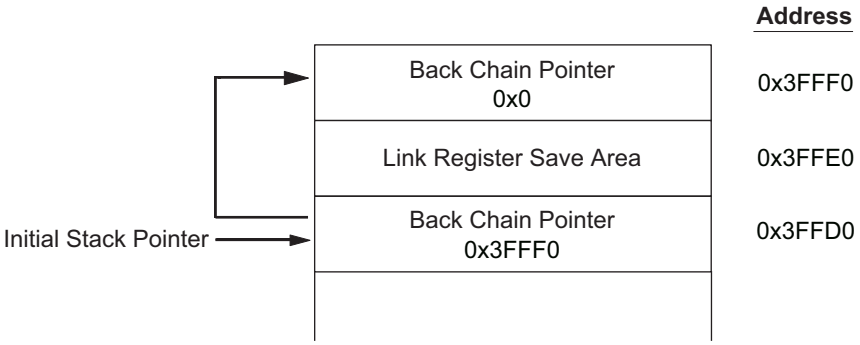
2.5. オペレーティング・システム・インターフェース

SPU は原則的にオペレーティング・システムを実行しないことから、制御している PowerPC® Processor Unit（PPU）によって提供されるオペレーティング・システム・サービスに依存する。PPU と SPU 間のオペレーティング・システム・インターフェースは、それぞれのオペレーティング・システムに対する CBEA Application Binary Interface 仕様によって指定される。

2.5.1. プログラム初期化

SPU プログラムが初めて実行開始されると、レジスタ `r1`（*SP*）の内容がスタックの一番上に初期化される。通常は、スタックの一番上は最大のクワッドワード・アドレスに置かれる最小限のスタックである。図 2-21に示すように、256 Kバイトのローカル・ストレージを持つシステムはスタック・ポインタを `0x3FFD0` に初期化する。このアドレスは、`0x3FFF0` への Back Chain へのポインタを含む。`0x3FFF0` のBack Chain へのポインタは、NULL（0）ポインタを含む。最初に実行される関数が「リンク・レジスタ」を保存するための空間（アドレス 0x3FFE0）が割り当てられる。その他すべてのレジスタの内容は、未指定である。よって、レジスタが指定された値を持つことを要求するプログラムは、明示的にその値を設定する必要がある。

図 2-21：メモリ・スタック



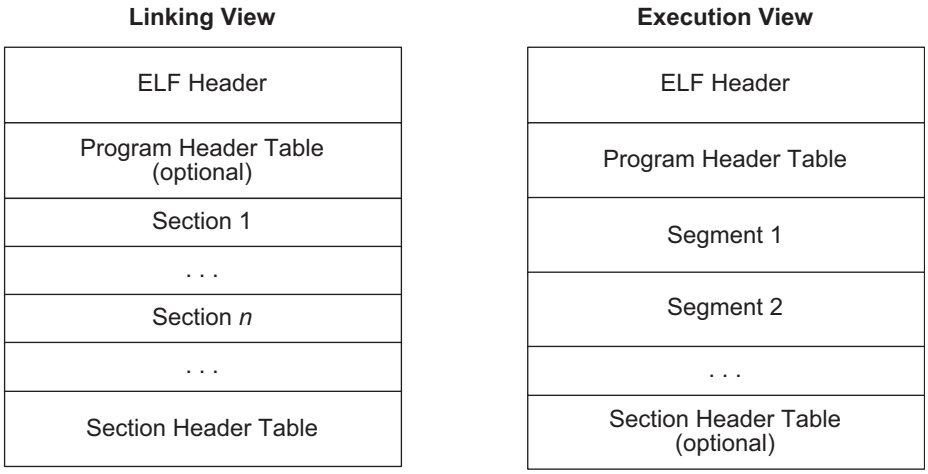
3. オブジェクト・ファイル

SPU オブジェクトのファイル・フォーマットは、「Executable and Linking Format（ELF）」でなければならない。本ドキュメントでは、ELF 標準を完全には規定しないが、代わりに ELF の概要を提供し、ソフトウェア・ツール間でのオブジェクト・ファイルの可搬性を確実なものにするために必要なセクションおよびフィールドを規定する。

3.1. ファイル・フォーマット

オブジェクト・ファイルは２つのアクティビティに関わる。１つは、プログラム・リンク（プログラムを作成すること）で、もう１つはプログラム実行（プログラムを走らせること）である。便宜上、および効率性を考慮し、オブジェクト・ファイル・フォーマットはファイル内容の並列した見方を提供し、これら２つのアクティビティが必要とするものの違いを反映する。図 3-22は、これら２つの見方を示す。

図 3-22：オブジェクト・ファイル・フォーマット



3.2. ELF ヘッダ

ELF ヘッダには、マシン固有の情報が含まれる。表 3-10に、SPU オブジェクトの固有情報を示す。

表 3-10：SPU ELF ヘッダ・フィールド

フィールド	値	コメント
e_ident[EI_CLASS]	ELFCLASS32	32 ビット・インプリメント
e_ident[EI_DATA]	ELFDATA2MSB	ビッグ・エンディアンのデータ・エンコード
e_type	ET_NONE	ファイル型無し
	ET_REL	再配置可能ファイル。実行可能ファイルまたはプラグイン・ファイルを生成するための、オブジェクト・リンクに適切なコードとデータを保持する再配置可能なファイル。
	ET_EXEC	実行可能ファイル。実行に適切なプログラムを保持する実行可能なファイル。
	ET_DYN	プラグイン・ファイル。プラグイン・ファイルには、名前付けされた各プラグインに、SPUNAME メモ・セクションが含まれる必要がある。追加情報に関しては、「2.2.6. モジュール外関数呼び出し」を参照のこと。
e_machine	EM_SPU	SPU プロセッサ ID。定義値は、23 である。
e_flags	0	現在、フラグは定義されていない。よって、このメンバには 0 が含まれる必要がある。

3.3. シンボル

コンパイラによって作成されるグローバル・シンボルを、「マングル」してはならない。つまり、これらのシンボルに先行文字を付加してはならない。

3.4. セクション

表 3-11に、プログラム・データとプログラム・コードを保持する ELF セクションを示す。

表 3-11：SPU 特別セクション

名称	型	属性	セクション内容
.bss	SHT_NOBITS	SHF_ALLOC SHF_WRITE	プログラムのメモリ・イメージの構築に使われる未初期化のデータ。定義により、プログラム実行の開始時に、プログラム・ローダーはデータをゼロで初期化する。
.data	SHT_PROGBITS	SHF_ALLOC SHF_WRITE	プログラムのメモリ・イメージの構築に使われる初期化されたデータ。
.text	SHT_PROGBITS	SHF_ALLOC SHF_EXECINSTR	プログラムの *テキスト*、すなわち実行可能な命令。

ローカル・ストレージにロードされる割り当て可能な ELF セクションには、以下の制限が適用される。

- セクションの下位境界は、16 バイトアラインのアドレスで開始しなければならない。
- セクションのサイズは 16 バイトの倍数でなければならない。セクション内容の合計サイズが 16 バイトの倍数でない場合には、16 バイトの倍数になるようにセクションが拡張される。拡張された領域の各バイトには 0 が設定される。

このような制限があるため、(1)すべてのロード可能な ELF セグメントは 16 バイト境界で開始する、(2)メモリサイズおよびファイルサイズは 16 バイトの倍数とする。

この仕様は最低要件を定義しているにすぎない。CBEA の実装によっては、より大きいセグメント-アラインの制約を課して DMA 転送を効率化し、パフォーマンスの向上を図る、といったことを行ってよい。

注意：ローカル・ストレージにロードされていない ELF セクションについては、これらの制限が適用される必要はない。

3.5. 再配置

3.5.1. 再配置型

再配置エントリは、命令およびデータ再配置フィールドの変更方法を記述する。再配置は、ワード、あるいはワードのサブセット上で操作される。表 3-13に示す計算は、アクションが再配置可能ファイルを実行可能ファイルに変換することを前提とする。概念的には、リンク・エディタが 1 つ以上の再配置可能なファイルをマージし、出力ファイルを形成する。このプロセスの一部として、リンク・エディタはまず入力ファイルの組み合わせ方法および配置方法を決定する。次に、シンボル値を更新し、その後必要な再配置を操作する。表 3-12に、再配置フィールドおよびその説明を示す。

表 3-12：再配置フィールド

フィールド	説明
word32	4 バイトを占有する 32 ビットのフィールドを指定する。4 バイトのアラインメントである
I7	4 バイトでアラインされるワードのビット 11～17 内に含まれる 7 ビットのフィールドを指定する。ワードのその他のビットは、変更されない。
I9	4 バイトでアラインされるワードのビット 7～8 および 25～31 内に含まれる 9 ビットのフィールドを指定する。ワードのその他のビットは、変更されない。
I9I	4 バイトでアラインされるワードのビット 16～17 および 25～31 内に含まれる 9 ビットのフィールドを指定する。ワードのその他のビットは、変更されない。
I10	4 バイトでアラインされるワードのビット 8～17 内に含まれる 10 ビットのフィールドを指定する。ワードのその他のビットは、変更されない。
I16	4 バイトでアラインされるワードのビット 9～24 内に含まれる 16 ビットのフィールドを指定する。ワードのその他のビットは、変更されない。
I18	4 バイトでアラインされるワードのビット 7～24 内に含まれる 18 ビットのフィールドを指定する。ワードのその他のビットは、変更されない。

表 3-13に、再配置型を示す。（表中で使用される表記法の説明に関しては、表 3-13に続く注釈を参照のこと。）

表 3-13：再配置型

名称	値	フィールド [1]	計算 [2]	コード生成例
R_SPU_NONE	0	無し	無し	-
R_SPU_ADDR10	1	I10*	(S + A) >> 4	lqd $3, symbol($4)
R_SPU_ADDR16	2	I16*	(S + A) >> 2	brasl $LR, function
R_SPU_ADDR16_HI	3	I16	#hi(S + A)	ilhu $3, symbol@h
R_SPU_ADDR16_LO	4	I16	#lo(S + A)	iohl $3, symbol@l
R_SPU_ADDR18	5	I18*	S + A	ila $3, symbol
R_SPU_ADDR32	6	word32	S + A	.word symbol
R_SPU_REL16	7	I16*	(S + A - P) >> 2	brsl $LR, function
R_SPU_ADDR7	8	I7	S + A	cwd $3, symbol($4)
R_SPU_REL9	9	I9*	(S + A - P) >> 2	hbra function, -100
R_SPU_REL9I	10	I9I*	(S + A - P) >> 2	hbr function, $3
R_SPU_ADDR10I	11	I10*	S + A	ai $3, $3, symbol
R_SPU_ADDR16I	12	I16*	S + A	il $3, symbol
R_SPU_REL32	13	word32	S + A - P	.word symbol
R_SPU_ADDR16X	14	I16*	S + A	ilh $3, symbol

[1] 表中の「フィールド」エントリにアスタリスクを持つ再配置型は、再配置の値が割り当てられたビットに収まらない場合、失敗する場合もある。

[2] 表中の「計算」エントリは、下記の表記を使用して説明される。

- *A*、*P*、および *S* は、それぞれ以下を表現する。

  A: 再配置可能なフィールドの値を算出するために使用する加数。

  P: 再配置されているストレージ・ユニットの場所（セクション・オフセットまたはアドレス）。`r_offset` を使用して算出される。

  S: インデックスが再配置エントリに置かれるシンボルの値。

- 「+」シンボルと「-」シンボルは、それぞれ順番に、32 ビット・モジュロの加算と減算である。「>>」は、右オペランドに与えられたビット数で、左オペランドの値を算術右シフト（符号をコピーしてシフト）することを示す。

- ワードのサブセットを更新する再配置型の場合、上位ビットはシフトされた前と同一でなければならない。シフト操作する再配置型の場合、シフトされる桁数分の最下位ビットは 0 でなければならない。

- `#hi(value)` と `#lo(value)` は、それぞれ順番に、示された値の最上位 16 ビットと最下位 16 ビットを示す。つまり、`#lo(x) = (x & 0xFFFF)`、`#hi(x) = ((x >> 16) & 0xFFFF` である。

4. プログラム・ロードおよび動的リンク

本章では、プログラム実行に関連するオブジェクト・ファイルを説明する。本章は、「3. オブジェクト・ファイル」と併せて読むこととする。

4.1. プログラム・ ヘッダ

プログラム・ヘッダ・テーブルは、主要なデータ構造である。プログラム・ヘッダには、ファイル内のセグメント・イメージの位置と、プログラムのメモリ・イメージを生成するために必要なその他の情報が含まれる。

4.1.1. SPU 環境メモ

SPU オブジェクトには、SPU プログラムの属性およびランタイム環境を定義する型 `PT_NOTE` のプログラム・ヘッダ要素を持つ、型 `SHT_NOTE` のセクションが含まれている場合がある。表 4-14および 表 4-15に、SPU 環境メモについての詳細を提供する。

表 4-14：SPU 環境メモ

フィールド	サイズ（バイト）	値
namesz	4	8
descsz	4	sizeof(spu_env)
type	4	1
name	8	“IBM SPU”
desc	sizeof(spu_env)	spu_env 構造体

SPU 環境メモには、`spu_env` 構造体のインスタンスが含まれる。`spu_env` 構造体のエントリを、表 4-15に示す。

表 4-15：spu_env 構造体

型	名称	説明
Elf32_Word	revision	構造体リビジョン・ナンバー。初期の構造体ナンバーは、1である。この構造体に将来追加されるものは、末尾に付加され、リビジョン・ナンバーがインクリメントされる。
Elf32_Word	ls_size	プログラムを実行するターゲットとなる SPU ローカル・ストレージのサイズ。要求される AMR（Address Memory Range －アドレス・メモリ範囲）レジスタ設定を指定する。サイズ 0 は、AMR レジスタを有効なアドレス範囲全体に設定する必要があることを示す。
Elf32_Word	stack_size	ランタイム SPU スタック・サイズ。「利用可能スタック空間」（レジスタ `R1` のワード要素 1）の確立に使用される。SPU 環境が指定されない場合、または stack_size に 0 が指定された場合には、「利用可能スタック空間」の値は `<top_of_stack> - _end` に初期化される。そうでない場合には、「利用可能スタック空間」の値は stack_size に初期化される。
Elf32_Word	flags	ELF_SPU_ENCRYPTED（bit 31）。SPU ELF プログラムの暗号化を指定し、実行前に暗号化解除および認証が必要であることを示す。

4.1.2. SPU ネーム・メモ

SPU オブジェクトは、ルックアップ・ネーム・ストリングで特定される必要があり、この名前は型 PT_NOTE のプログラム・ヘッダ要素を持つ SHT_NOTE に含まれていなければならない。

表 4-16に、SPU ネーム・メモ内にあるフィールドのサイズおよび値を示す。

表 4-16 : SPU ネーム・メモ

フィールド	サイズ（バイト）	値
namesz	4	8
descsz	4	desc フィールド中のバイト数。この値は 4 の倍数でなければならない。
type	4	1
name	8	“SPUNAME”
desc	(descsz 参照)	オブジェクトのパス名を特定する null 終了のルックアップ・ストリング。

以上