**IBM**

# C/C++ Language Extensions for Cell Broadband Engine Architecture

## Version 2.5

**CBEA JSRE Series**

Cell Broadband Engine Architecture
Joint Software Reference Environment
Series

September 14, 2007

**IBM**®

**IBM**

# Table of Contents

## List of Tables

IBM

## List of Figures

# About This Document

This document describes language extension specifications that allow software developers to access hardware features that are not easily accessible from a high level language, such as C or C++, in order to obtain the best performance from a Synergistic Processor Unit (SPU) and a PowerPC® Processor Unit (PPU) of the Cell Broadband Engine™. This document also includes function specifications to facilitate communication between SPUs and PPU, and it lists a minimal set of standard library functions that must be provided as part of a standard SPU programming environment.

## Audience

This document is intended for system and application programmers who want to write SPU and PPU programs for a CBEA-compliant processor.

## Version History

This section describes significant changes made to each version of this document.

| Version Number & Date | Changes |
|---|---|
| v. 2.5<br>September 14, 2007 | Corrected miscellaneous documentation errors (TWG_RFC00102-1: CORRECTION NOTICE). |
| | Added six new PPU intrinsics to simplify streaming data prefetch (TWG_RFC00103-0 as amended by TWG_RFC00103-1). |
| | Described special behaviors for some of the missing classification macros (TWG_RFC00104-0). |
| | Changed the return/argument types of several PPU intrinsics (TWG_RFC00105-0). |
| | Changed the descriptive names of the SPU rotate and shift intrinsics (TWG_RFC00106-2). |
| | Changed the descriptive names of several intrinsics (TWG_RFC00107-2: CORRECTION NOTICE). |
| | Added a section describing the MFC tag mananger (TWG_RFC00109-2). |
| | Eliminated unnecessary spaces from several headings (TWG_RFC00111-0: CORRECTION NOTICE). |
| | Specified the SPU "fastest minimum-width integer" typedefs in a way that conforms with the implementations for both spu-gcc and spuxlc (TWG_RFC00117-0). |
| | Clarified the mapping of intrinsics between SPU and VMX (TWG_RFC00118-1). |
| | Corrected the implementation specification of the `mfc_hl2ea` function so that it matches the implementation in `spu_mfcio.h`. (TWG_RFC00119-0: CORRECTION NOTICE). |
| | Made miscellaneous editorial changes. |
| v. 2.4<br>March 8, 2007 | Added support for enhanced double precision SPU instructions (TWG_RFC00071-0). |
| | Specified use of vector data types with standard C/C++ operators (TWG_RFC00082-1). |
| | Made it explicit that the vector keyword n the SPU is the same as the vector keyword on the PPU (TWG_RFC00096-0). |
| | Provided a predefined macro for use by compilers that are targeted to a processor that supports the SPU's optional enhanced double precision instructions (TWG_RFC00097-0). |
| | Attached "volatile" with *dmalist* arguments in intrinsics (TWG_RFC00100-0). |

| Version Number & Date | Changes |
|---|---|
| | Corrected various organizational, grammatical, and spelling issues (TWG_RFC00093-0: CORRECTION NOTICE and TWG_RFC00094-0: CORRECTION NOTICE). |
| | Specified the kinds of variables to which the aligned attribute applies (TWG_RFC00098-0). |
| | Corrected the specification of `isnan()` so that it applies only to single precision (TWG_RFC00099-0: CORRECTION NOTICE). |
| | Corrected various minor errors (TWG_RFC00101-0: CORRECTION NOTICE). |
| v. 2.3<br>December 4, 2006 | Corrected the function parameter ordering of the PPU `__stwbrx` instrinsic (TWG_RFC00074-0: CORRECTION NOTICE) |
| | Corrected the type of element initializers used to initialize a vector of `signed`/`unsigned char` (TWG_RFC00075-0: CORRECTION NOTICE) |
| | Changed to note that the use of double-precision contracted operations is permitted by default unless prohibited by the `FP_CONTRACT` pragma or the no-fast-double compiler option (TWG_RFC00076-0). |
| | Added PPU data types and programming directives to Chapter 1, and changed title from "SPU Data Types and Program Directives" to "Data Types and Programming Directives" (TWG_RFC00077-1). |
| | Removed the `__fre`, `__frsqrtes`, and `__popcntb` intrinsics, and added the `__frsqrte` intrinsic (TWG_RFC00078-3). |
| | Added that support is provided in the floating-point environment for both double-precision elements and all four single-precision elements. Also, updated information for `FLT_ROUNDS` (TWG_RFC00079-1). |
| | Added a new chapter, "PPU VMX Intrinsics", that specifies a set of intrinsic functions making the underlying PPU VMX instruction set accessible from the C programming language (TWG_RFC00081-1 and TWG_RFC00092-0). |
| | Added 32-bit ABI support to the PPU intrinsic functions, changed function arguments to provide a consistent high-level interface, and corrected several typographical errors (TWG_RFC00083-1). |
| | Changed the return type of the `__fctiw` and `__fctiwx` PPU intrinsic functions , changed the descriptive names of these and other similar conversion intrinsics, and removed the `__stfiwx` intrinsic function (TWG_RFC00089-1). |
| | Identified deprecated PPU VMX operations and recommendations for suitable PPU intrinsic function alternatives (TWG_RFC00090-0). |
| | Identified non-supported language features and specified that C++ exception handling is not supported on the SPU (TWG_RFC00091-0). |
| v. 2.2<br>October 11, 2006 | Applied the changes made in the following requests: TWG_RFC00056-0, TWG_RFC00057-0, TWG_RFC00058-2, TWG_RFC00061-1, TWG_RFC00060-1, TWG_RFC00062-0, TWG_RFC00066-2, TWG_RFC00067-2, TWG_RFC00068-0, TWG_RFC00070-1, TWG_RFC00072-0, and TWG_RFC00073-0. |
| | Changed document title because its contents are no longer limited to the SPU. Changed the sections "About this Document" and "Audience" accordingly. Applied TWG_RFC00053-0, TWG_RFC00054-1, and TWG_RFC00055-0. |
| | Replaced uses of a protected name by references to the document *AltiVec™ Technology Programming Interface Manual* per TWG_RFC00050-1 and TWG_RFC00052-0. |
| | Corrected several operand errors related to `spu_sub`, which is the arithmetic intrinsic for vector subtraction (TWG_RFC00046-0: CORRECTION NOTICE). |
| | Corrected various documentation errors; for example, changed sample code demonstrating how to restore the Stack Pointer Information register as a result of invoking the longjmp function (TWG_RFC00047-0: CORRECTION NOTICE). |
| | Specified that alternate vector syntax for vector literals is optional rather than |

IBM

| Version Number & Date | Changes |
|---|---|
| | mandatory (TWG_RFC00050). |
| v. 2.1<br>October 20, 2005 | Added a sub-section called "Malloc Heap" to the C library section of the "C and C++ Standard Libraries" chapter. This section is related to an attempt to define a standard process for memory heap initialization and stack management (TWG_RFC00024-3). |
| | In the "SPU and Vector Multimedia Extension Intrinsics" chapter, clarified which intrinsic mappings are required according to this specification and which are not because a straightforward mapping does not exist. Provided additional explanations regarding the intrinsics that are difficult to map (TWG_RFC00034-1: CORRECTION NOTICE). |
| | Corrected the description of the `si_stqx` instruction (TWG_RFC00035-0: CORRECTION NOTICE). |
| | Corrected various documentation errors; for example, changed several descriptions in the "Alternate Vector Literal Format and Description" table. |
| | (TWG_RFC00036-0: CORRECTION NOTICE, TWG_RFC00041-0: CORRECTION NOTICE, TWG_RFC00045-0: CORRECTION NOTICE). |
| | Changed "Broadband Processor Architecture" to "Cell Broadband Engine Architecture", and changed "BPA" to "CBEA" (TWG_RFC00037-0: CORRECTION NOTICE). |
| | Deleted several references to BE revisions DD1.0 and DD2.0 (TWG_RFC00040-0: CORRECTION NOTICE). |
| | Added a new chapter describing MFC I/O intrinsics; these intrinsics facilitate MFC programming by defining a common set of utility functions (TWG_RFC00043-2). |
| v. 2.0<br>July 11, 2005 | Deleted several sections in the "About This Document" chapter. Changed two entries in the Write Word Channel table from `si_wrch(channel, si_to_int(a))` to `si_wrch(channel, si_from_int(a))`. Clarified that the syntax for vector type specifiers does not allow the use of a typedef name as a type specifier. (All changes per TWG_RFC00032-0: CORRECTION NOTICE.) |
| v. 1.9<br>June 10, 2005 | Added new chapter describing C and C++ Libraries (TWG_RFC00018-5). |
| | Added new chapter describing SPU floating-point arithmetic (TWG_RFC00027-1). |
| | Changed "Broadband Engine" or "BE" to "a processor compliant with the Broadband Processor Architecture" or "a processor compliant with BPA"; changed VMX to Vector Multimedia Extension; changed Synergistic Processing Element to Synergistic Processor Element; and changed Synergistic Processing Unit to Synergistic Processor Unit. Defined a PPU as a PowerPC Processor Unit on first major instance. Corrected several book references and changed copyright page so that trademark owners were specified. (All changes per TWG_RFC00031-0: CORRECTION NOTICE.) |
| | Made miscellaneous changes to the "About This Document" section. |
| v. 1.8<br>May 12, 2005 | Added new channel number for multisource synchronization requests (TWG_RFC00023-1). |
| | Corrected example describing loading of misaligned vectors. |
| | Changed PU to PPU and SPC to SPE; changed "PU-to-SPU" (mailboxes) and "SPU-to-PU" to "inbound" and "outbound" respectively (TWG_RFC00028-1: CORRECTION NOTICE). |
| | Changed the name of `spu_mulhh` to `spu_mule` (TWG_RFC00021-0). |
| | Updated channel names to coincide with BPA channel names (TWG_RFC00029-1). |
| v. 1.7<br>July 16, 2004 | Clarified that channel intrinsics must not be reordered with respect to other channel commands or volatile local-storage memory accesses (TWG_RFC00007-1). |
| | Warned that compliant compilers may ignore `__align_hint` intrinsics (TWG_RFC00008-1). |
| | Added an additional SPU instruction, `orx` (TWG_RFC00010-0). |

| Version Number & Date | Changes |
|---|---|
| | Added mnemonics for channels that support reading the event mask and tag mask (TWG_RFC00011-0). |
| | Specified that `spu_ienable` and `spu_idisable` intrinics do not have return values (TWG_RFC00013-0). |
| | Moved paragraph beginning "This intrinsic is considered volatile..." from `spu_mfspr` intrinsic to `spu_mtf`pscr (TWG_RFC00014-0). |
| | Changed the descriptions for `si_lqd` and `si_stqd` intrinics (TWG_RFC00015-1). |
| | Provided new descriptions of various rotation-and-mask intrinsics, specifically: `spu_rlmask`, `spu_rlmaska`, `spu_rlmaskqw`, `spu_rlmaskqwbyte`, and `spu_rlmaskqwbytebc`. These descriptions include pseudo-code examples (TWG_RFC00016-1). |
| | Made miscellaneous editorial changes. |
| v. 1.6<br>March 12, 2004 | Made miscellaneous editorial changes. |
| v. 1.5<br>February 25, 2004 | Changed formatting of document so that it reflects the typographic conventions mentioned in the "About This Document" section. Made miscellaneous editorial changes. |
| | Changed some of the parameter types for `spu_mfcdma32` and `spu_mfcdma64`, as requested in TWG_RFC00002. |
| | Inserted new specifications for the vector literal format, as requested in TWG_RFC00003. |
| v. 1.4<br>January 20, 2004 | Changed document to new format, including front matter. Made miscellaneous editorial changes. |
| v. 1.3<br>November 4, 2003 | Added enable/disable interrupt intrinsics. |
| v. 1.2<br>September 2, 2003 | Changed parameter types of `spu_sel` intrinsic to be compatible with Vector Multimedia Extension's `vec_sel`. |
| | Added `si_stopd` specific intrinsic. |
| | Corrected tables for `spu_genb` and `spu_genc` generic intrinsics. |
| v. 1.1<br>June 15, 2003 | Made changes to support RFC 24. Added isolation control channel 64. |
| | Made changes to support RFC 33. Removed `spu_addc`, `spu_addsc`, `spu_subb`, and `spu_subsb`. Added `spu_addx`, `spu_subx`, `spu_genc`, `spu_gencx`, `spu_genb`, and `spu_genbx`. |
| v. 1.0<br>April 28, 2003 | Made minor corrections. |
| v. 0.9<br>March 7, 2003 | Added new intrinsics to support new or modified instructions. These include: `fscrrd`, `fscrwr`, `stop`, `dfma`, `mpyhhau`, `mpyhhu`, `rotqmbybi`, `iret`, `lqr`, and `stqr`. Also added intrinsics to support new feature bits for `iret`, `bisled`, `bihnz`, and `sync`. |
| v. 0.8<br>January 23, 2003 | Improved documentation of specific intrinsics. Completely defined parameter ordering and immediate sizes. |
| | Defined new global (`spu_intrinsics.h`) and compiler specific (`spu_internals.h`) header files. Specified that single token vector types and channel enumerants are declared in `spu_intrinsics.h`. |
| | Added specific pointer casting intrinsics. |
| | Added standardized `__SPU__` conditional compilation control. |
| | Changed specific convert intrinsics to unbiased scale parameters, such as generic intrinsics. |
| | Specified that the bisled target function does not observe the standard calling convention with respect to volatile registers. |
| v. 0.7 | Specified that gcc-style inline assembly is required. |

IBM

| Version Number & Date | Changes |
|---|---|
| November 18, 2002 | Specified that `__builtin_expect` is required. |
| | Added `bisled` specific and generic intrinsics. |
| | Added `__align_hint` intrinsic. |
| | Specified that the `restrict` type qualifier is required. |
| | Specified that out-of-range scale factors on generic conversion intrinsics return an error. |
| v. 0.6 September 24, 2002 | Changed document title to include C++. |
| | Made miscellaneous clarifications and typing corrections. |
| | Changed `spu_eqv` to return the same vector type as its inputs. |
| | Changed `spu_and`, `spu_or`, and `spu_xor` to accept immediate values of the same type as the elements of parameter *a*. |
| | Added specific casting intrinsics. |
| | Changed default action on out-of-range immediate values for specific intrinsics to issuing an error. |
| | Added documentation of the `__builtin_expect` builtin. |
| | Completed SPU-to-Vector Multimedia Extension intrinsic mapping section. |
| v. 0.5 August 27, 2002 | Edited discussion of Vector Multimedia Extension-to-SPU intrinsic mapping. |
| | Removed appendices. |
| | Added support for 32-bit read and write channel intrinsics. Renamed quadword channel read and write to `readchqw` and `writechqw`. |
| v. 0.4 August 5, 2002 | Corrected the instruction mapping for `spu_promote` and `spu_extract`. |
| | Specified that instruction mapping for generic intrinsics `spu_re` and `spu_rsqrte` include the `FI` (floating-point interpolate) instruction. |
| | Renamed `spu_splat` to `spu_splats` (scalar splat) to avoid confusion with `vec_splat`. |
| | Added documentation about the size of the immediate intrinsic forms. |
| | Changed all `vector signed long` to `vector signed long long`. |
| | Changed *count* to unsigned for `spu_sl`, `spu_slqw`, `spu_slqwbyte`, and `spu_slqwbytebc`. |
| | Changed *count* to signed for `spu_rl`, `spu_rlmask` and `spu_rlmaska`. |
| | Specified that the return value of `spu_cntlz` is an unsigned value. |
| | Corrected description of `spu_gather` intrinsic. |
| | Edited mapping documentation of scalars for `spu_and`, `spu_or`, and `spu_xor`. |
| | Removed vector input forms of `spu_hcmpeq` and `spu_hcmpgt`. |
| v. 0.3 July 16, 2002 | Added `fsmbi` to literal constructor instructions. Added `fsmbi` (immediate form) to `spu_maskb` intrinsic. |
| | Added vector forms to compare and halt (`spu_hcmpeq` and `spu_hcmpgt`) intrinsics. |
| | Added `qword` data type as the only vector type accepted by specific intrinsics. |
| | Added typedefs for the vector types as the basic types used for code portability. |
| | Merged all `spu_splat` generic intrinsics into a single intrinsic. |
| | Dropped `spu_load`, `spu_store`, and `spu_insertctl` generic intrinsics. |
| v. 0.2 July 9, 2002 | Incorporated changes and suggestions from Peng. |
| | Changed `vector long` types to `vector long long`. |
| v. 0.1 June 21, 2002 | First version of the language extension specification. Initial specification based on the Tobey compiler intrinsics specification. |

## Related Documentation

The following table provides a list of references and supporting materials for this document:

| Document Title | Version | Date |
| --- | --- | --- |
| *ISO/IEC Standard 9899:1999 (C Standard)* | | |
| *ISO/IEC Standard 14882:1998 (C++ Standard)* | | |
| *IEEE-754 (Standard for Binary Floating-Point Arithmetic)* | | |
| *Synergistic Processor Unit Instruction Set Architecture* | 1.2 | January 2007 |
| *Cell Broadband Engine Architecture* | 1.01 | October 2006 |
| *Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification* | 1.2 | May 1995 |
| *Tool Interface Standard (TIS), DWARF Debugging Information Format Specification* | 2.0 | May 1995 |
| *PowerPC Architecture Book, Book II: PowerPC Virtual Environment Architecture* | 2.02 | January 2005 |
| *AltiVec™ Technology Programming Interface Manual* | | June 1999 |

## Conventions Used in This Document

### Bit Notation

Standard bit notation is used throughout this document. Bits and bytes are numbered in ascending order from left to right. Thus, for a 4-byte word, bit 0 is the most significant bit and bit 31 is the least significant bit, as shown in the following figure:
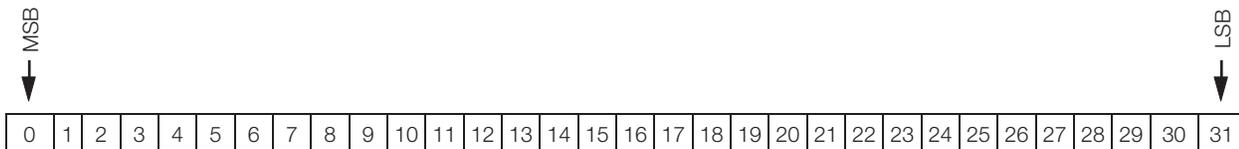
MSB = Most significant bit

LSB = Least significant bit

Notation for bit encoding is as follows:

- Hexadecimal values are preceded by `0x`. For example: `0x0A00`.
- Binary values in sentences appear in single quotation marks. For example: '1010'.

XX About This Document

**Byte Ordering and Element Numbering**

Byte ordering and element numbering are always displayed in big-endian order, as shown in Figure 1-1.

Figure 1-1: Big-Endian Byte/Element Ordering for Vector Types

| 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 |

**Other Conventions**

The following typographic conventions are used throughout this document:

| Convention | Meaning |
|---|---|
| `courier` | Indicates programming code and literals, such as processing instructions, register names, data types, events, and file names. Also indicates function and macro names. This convention is only used where it facilitates comprehension, especially in narrative descriptions. |
| `courier + italics` | Indicates arguments, parameters, and variables. This convention is only used where it facilitates comprehension, especially in narrative descriptions. |
| *italics (without courier)* | Indicates emphasis. Except when hyperlinked, book references are in italics. When a term is first defined, it is often in italics. |
| blue | Indicates a hyperlink (color printers or online only). |

C/C++ Language Extensions for Cell Broadband Engine Architecture, Version 2.5

# 1. Data Types and Programming Directives

This chapter specifies PPU Vector Multimedia Extension (VMX) and SPU vector data types, operations on these data types, programming directives, and predefined macro target definitions.

Any conflct between the requirements described here for PPU VMX data types and the *AltiVec™ Technology Programming Interface Manual* is unintentional.

The `vector` keyword and the `__vector` keyword have the same properties, defined in the *AltiVec™ Technology Programming Interface Manual*. The `__vector` keyword is preferred for code portability because it is always defined.

## 1.1. Data Types

In this section, a set of fundamental vector data types are introduced to the C language, and several mappings are described that relate PPU and SPU data types to one another.

### 1.1.1. Fundamental Data Types

The fundamental vector data types that are supported by the PPU and SPU are shown in Table 1-1. All of these data types are 128-bits long and contain from 2 to 16 elements, depending on the corresponding element data type.

Table 1-1: Vector Data Types

| Vector Data Type | Content | SPU/PPU |
|---|---|---|
| vector unsigned char | 16 8-bit unsigned chars | Both |
| vector signed char | 16 8-bit signed chars | Both |
| vector unsigned short | 8 16-bit unsigned halfwords | Both |
| vector signed short | 8 16-bit signed halfwords | Both |
| vector unsigned int | 4 32-bit unsigned words | Both |
| vector signed int | 4 32-bit signed words | Both |
| vector unsigned long long | 2 64-bit unsigned doublewords | SPU |
| vector signed long long | 2 64-bit signed doublewords | SPU |
| vector float | 4 32-bit single-precision floats | Both |
| vector double | 2 64-bit double-precision floats | SPU |
| qword | quadword (16-byte), used exclusively as an input/output to a specific intrinsic function. See section "2.1. Specific Intrinsics" | SPU |
| vector bool char | 16 8-bit bools – 0 (false) 255 (true) | PPU |
| vector bool short | 8 16-bit bools  – 0 (false) 65535 (true) | PPU |
| vector bool int | 4 32-bit bools  – 0 (false) $2^{32}$ – 1 (true) | PPU |
| vector pixel | 8 16-bit unsigned halfword, 1/5/5/5 pixel | PPU |

The syntax for vector type specifiers does not allow the use of a typedef name as a type specifier. For example, the following declaration is not allowed:

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

### 1.1.2. Mapping of PPU Data Types to SPU Data Types

Not all PPU vector data types are supported on the SPU. The PPU vector data types that do not map identically to SPU data types are shown in Table 1-2.

Table 1-2: Non-identical Mapping of PPU VMX Data Types to SPU Data Types

| PPU VMX Data Type | Maps to SPU Data Type |
|---|---|
| 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] Because `vector pixel` and `vector bool short` are mapped to the same base vector type (`vector unsigned short`), the overloaded functions for `vec_unpackh` and `vec_unpackl` cannot be uniquely resolved.

### 1.1.3. Mapping of SPU Data Types to PPU Data Types

Not all SPU data types are supported by the PPU VMX. The SPU data types that do not map identically to PPU vector data types are shown in Table 1-3.

Table 1-3: Non-identical Mapping of SPU Data Types to PPU VMX Data Types

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

## 1.2. Header Files

There are separate system header files for the SPU and PPU that include typedefs and other information required by this specification.

### 1.2.1. Header File Contents

The SPU system header file, `spu_intrinsics.h`, defines common enumerations and typedefs. These include the single token vector types and MFC channel mnemonic enumerations (see Table 1-4 and Table 2-91, respectively). In addition, `spu_intrinsics.h` will include a compiler-specific header file, `spu_internals.h`, that contains any implementation-specific definitions.

The PPU system header file, `altivec.h`, defines typedefs and keywords and also includes any implementation-specific definitions. The PPU system header file, `vec_types.h`, defines typedefs required by the language extension features defined in this specification.

### 1.2.2. Single Token Typedefs

To improve code portability, single token typedefs are provided for the vector keyword data types. These typedefs, which are shown in Table 1-4 are defined in `spu_intrinsics.h` on the SPU and in `vec_types.h` on the PPU. Besides simplifying type declarations, the single token types serve as class names for extending generic intrinsics or for mapping between PPU VMX intrinsics and/or SPU intrinsics.

Table 1-4: Single Token Vector Data Types

| Vector Keyword Data Type | Single Token Typedef | SPU/PPU |
|---|---|---|
| vector unsigned char | vec_uchar16 | Both |
| vector signed char | vec_char16 | Both |
| vector unsigned short | vec_ushort8 | Both |
| vector signed short | vec_short8 | Both |
| vector unsigned int | vec_uint4 | Both |
| vector signed int | vec_int4 | Both |
| vector unsigned long long | vec_ullong2 | SPU |

| Vector Keyword Data Type | Single Token Typedef | SPU/PPU |
|---|---|---|
| vector signed long long | vec_llong2 | SPU |
| vector float | vec_float4 | Both |
| vector double | vec_double2 | SPU |
| vector bool char | vec_bchar16 | PPU |
| vector bool short | vec_bshort8 | PPU |
| vector bool int | vec_bint4 | PPU |
| vector pixel | vec_pixel8 | PPU |

## 1.3. Alignment

### 1.3.1. Default Data Type Alignments

Table 1-5 shows the size and default alignment of the various data types.

Table 1-5: Default Data Type Alignments

| Data Type | Size | Alignment |
|---|---|---|
| char | 1 | byte |
| short | 2 | halfword |
| int | 4 | word |
| long | 4 | word/doubleword |
| long long | 8 | doubleword |
| float | 4 | word |
| double | 8 | doubleword |
| pointer | 4 | word |
| vector | 16 | quadword |

The aligned attribute will be provided by implementations to align static, global, and local variables, as well as static and non-static data members. The aligned attribute will not guarantee alignment of variables allocated using malloc or operator new. Implementations will support at least 128-byte alignment.

In the following declaration statement, the floating-point scalar $factor$ will be aligned on a quadword boundary:

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

### 1.3.2. __align_hint

The `__align_hint` intrinsic is provided to improve data access through pointers and to provide compilers the additional information that is needed to support auto-vectorization. This built-in function is available only on the SPU.

Although `__align_hint` is defined as an intrinsic, it behaves like a directive, because no code is ever specifically generated. For example:

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

The `__align_hint` intrinsic informs the compiler that the pointer $ptr$ points to data with a base alignment of $base$ and with an offset from $base$ of $offset$. The base alignment has to be a power of 2. A base address of zero implies that the pointer has no known alignment. The alignment offset has to be less than $base$ or zero.

The `__align_hint` intrinsic is not intended to specify pointers that are not naturally aligned. Specifying pointers that are not naturally aligned results in data objects straddling quadword boundaries. If a programmer specifies alignment incorrectly, incorrect programs might result.

**Programming Note:** Although compliant compiler implementations must provide the `__align_hint` intrinsic, compilers may ignore these hints.

## 1.4. Operating on Vector Types

This section describes the C/C++ operators and operations that are required to act on vector data types. These operators are the `sizeof()` operator, the assignment operator (`=`), and the address operator (`&`). Many other standard C/C++ operators are also extended for vector data types. The overloading of these operators for vector data types is described in "10. Operator Overloading for Vector Data Types".

The operations on vector data types are pointer operations and type casting operations.

### 1.4.1. sizeof() Operator

The operation `sizeof()` on a vector type always returns 16.

### 1.4.2. Assignment Operator

If either the left or right side of an expression has a vector type, both sides of the expression has to be of the same vector type. Thus, the expression `a = b` is valid and represents assignment if `a` and `b` are of the same type or if neither variable is a vector type. Otherwise, the expression is invalid, and the compiler reports the inconsistency as an error.

### 1.4.3. Address Operator

The operation `&a` is valid when `a` is a vector type. The result of the operation is a pointer to vector `a.`

### 1.4.4. Pointer Arithmetic and Pointer Dereferencing

The usual pointer arithmetic involving a pointer to a vector type can be performed. For example, assuming `p` is a pointer to a vector type, `p+1` is the pointer to the next vector following `p`.

Dereferencing the vector pointer `p` implies a 128-bit vector load from or store to the address obtained by masking the 4 least significant bits of `p`. When a vector is misaligned, the 4 least significant bits of its address are nonzero. Although vectors are 16-byte aligned (see section "1.3. Alignment"), it nevertheless might be desirable to load or store a vector that is misaligned. A misaligned vector can be loaded in several ways using generic intrinsics (see section "2.2. Generic Intrinsics and Built-ins").

The following code shows one example of how to load a misaligned floating-point vector on the SPU:

```
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));
}
```

Similarly, this next example shows how to store to a misaligned floating-point vector on the SPU.

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

```
        int shift;

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

        flt = spu_rlqwbyte(flt, -shift);

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

### 1.4.5. Type Casting

Pointers to vector types and non-vector types may be cast back and forth to each other. For the purpose of aliasing, a vector type is treated as an array of its corresponding element type, as shown in Table 1-6. If a pointer is cast to the address of a vector type, it is the programmer's responsibility to ensure that the address is 16-byte aligned. Vector types that are applicable only on the PPU do not have an underlying scalar type.

Table 1-6: Vector Pointer Types and Matching Base Element Pointer Types

| Vector Pointer Type (vector T*) | Base Element Pointer Type (T*) | SPU/PPU |
|---|---|---|
| vector unsigned char* | unsigned char* | Both |
| vector signed char* | signed char* | Both |
| vector unsigned short* | unsigned short* | Both |
| vector signed short* | signed short* | Both |
| vector unsigned int* | unsigned int* | Both |
| vector signed int* | signed int* | Both |
| vector unsigned long long* | unsigned long long* | SPU |
| vector signed long long* | signed long long* | SPU |
| vector float* | float* | Both |
| vector double* | double* | SPU |

Casts from one vector type to another vector type has to be explicit and are done using normal C-language casts. None of these casts performs any data conversion. Thus, the bit pattern of the result is the same as the bit pattern of the argument that is cast.

Casts between vector types and scalar types are illegal. On the SPU, the `spu_extract`, `spu_insert`, and `spu_promote` generic intrinsics or the specific casting intrinsics may be used to efficiently achieve the same results (see section "2.1.1. Specific Casting Intrinsics"). On the PPU, the `vec_lde` and `vec_ste` intrinsics may be used to copy between scalar and vector types.

### 1.4.6. Vector Literals

As shown in Table 1-7, a vector literal is written as a parenthesized vector type followed by a curly braced set of constant expressions. If a vector literal is used as an argument to a macro, the literal has to be enclosed in parentheses. In all other cases, the literal can be used without enclosing parentheses. The elements of the vector are initialized to the corresponding expression. Elements for which no expressions are specified default to 0. Vector literals may be used either in initialization statements or as constants in executable statements. The syntax for vector initialization and for vector compound literals is the same as the corresponding array syntax except designators which do not exist for vector elements. The initializer should act as an array of either 2, 4, 8, or 16 elements depending on the size of the underlying type. For example the following two initializations are valid and equivalent:

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

The following two struct initializers are also valid and equivalent:

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

The following types on both the SPU and PPU cannot be initialized using a vector literal: `qword`, `vector bool char`, `vector bool short`, `vector bool int`, and `vector pixel`. They can be created by using the intrinsics or by casting to these vector types.

Table 1-7: Vector Literal Format and Description

| Notation | Represents | SPU/PPU |
|---|---|---|
| (vector unsigned char) {unsigned char, ...} | A set of 16 unsigned 8-bit quantities. | Both |
| (vector signed char) {signed char, ...} | A set of 16 signed 8-bit quantities. | Both |
| (vector unsigned short) {unsigned short, ...} | A set of 8 unsigned 16-bit quantities. | Both |
| (vector signed short) {signed short, ...} | A set of 8 signed 16-bit quantities. | Both |
| (vector unsigned int) {unsigned int, ...} | A set of 4 unsigned 32-bit quantities. | Both |
| (vector signed int) {signed int, ...} | A set of 4 signed 32-bit quantities. | Both |
| (vector unsigned long long) {unsigned long long, ...} | A set of 2 unsigned 64-bit quantities. | SPU |
| (vector signed long long) {signed long long, ...} | A set of 2 signed 64-bit quantities. | SPU |
| (vector float) {float, ...} | A set of 4 32-bit floating-point quantities. | Both |
| (vector double) {double, ...} | A set of 2 64-bit floating-point quantities. | SPU |

An alternate format may also be supported which corresponds to the syntax specified in the *AltiVec™ Technology Programming Interface Manual*. This format consists of a parenthesized vector type followed by a parenthesized set of constant expressions. See Table 1-8.

Table 1-8: Alternate Vector Literal Format and Description

| Notation | Represents | SPU/PPU |
|---|---|---|
| (vector unsigned char)(unsigned int) | A set of 16 unsigned 8-bit quantities that all have the value specified by the integer. | Both |
| (vector unsigned char)(unsigned int, ..., unsigned int) | A set of 16 unsigned 8-bit quantities specified by the 16 integers. | Both |
| (vector signed char)(signed int) | A set of 16 signed 8-bit quantities that all have the value specified by the integer. | Both |
| (vector signed char)(signed int, ..., signed int) | A set of 16 signed 8-bit quantities specified by the 16 integers. | Both |
| (vector unsigned short)(unsigned int) | A set of 8 unsigned 16-bit quantities that all have the value specified by the integer. | Both |
| (vector unsigned short)(unsigned int, ..., unsigned int) | A set of 8 unsigned 16-bit quantities specified by the 8 integers. | Both |
| (vector signed short)(signed int) | A set of 8 signed 16-bit quantities that all have the value specified by the integer. | Both |
| (vector signed short)(signed int, ..., signed int) | A set of 8 signed 16-bit quantities specified by the 8 integers. | Both |
| (vector unsigned int)(unsigned int) | A set of 4 unsigned 32-bit quantities that all have the value specified by the integer. | Both |
| (vector unsigned int)(unsigned int, ..., unsigned int) | A set of 4 unsigned 32-bit quantities specified by the 4 integers. | Both |
| (vector signed int)(signed int) | A set of 4 signed 32-bit quantities that all have the value specified by the integer. | Both |

| Notation | Represents | SPU/PPU |
|---|---|---|
| (vector signed int)(signed int, ..., signed int) | A set of 4 signed 32-bit quantities specified by the 4 integers. | Both |
| (vector unsigned long long)(unsigned long long) | A set of 2 unsigned 64-bit quantities that all have the value specified by the long integer. | SPU |
| (vector unsigned long long)(unsigned long long, unsigned long long) | A set of 2 unsigned 64-bit quantities specified by the 2 long integers. | SPU |
| (vector signed long long)(signed long long) | A set of 2 signed 64-bit quantities that all have the value specified by the long integer. | SPU |
| (vector signed long long)(signed long long, signed long long) | A set of 2 signed 64-bit quantities specified by the 2 long integers. | SPU |
| (vector float)(float) | A set of 4 32-bit floating-point quantities that all have the value specified by the float. | Both |
| (vector float)(float, float, float, float) | A set of 4 32-bit floating-point quantities specified by the 4 floats. | Both |
| (vector double)(double) | A set of 2 64-bit double-precision quantities that all have the value specified by the double. | SPU |
| (vector double)(double, double) | A set of 2 64-bit quantities specified by the 2 doubles. | SPU |

## 1.5. Restrict Type Qualifier

The `restrict` type qualifier, which is specified in the C99 language specification, is intended to help the compiler generate better code by ensuring that all access to a given object is obtained through a particular pointer. When a pointer uses the `restrict` type qualifier, the pointer is `restrict`-qualified. For example:

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

In the above prototype, both pointers, *s1* and *s2*, are `restrict`-qualified. Therefore, the compiler can safely assume that the source and destination objects will not overlap, allowing for a more efficient implementation.

## 1.6. SPU Programmer Directed Branch Prediction

Branch prediction can be significantly improved by using feedback-directed optimization. However, feedback-directed optimization is not always practical in situations where typical data sets do not exist. Instead, on the SPU, programmer-directed branch prediction is provided using an enhanced version of GCC's `__builtin_expect` function.

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

Programmers can use `__builtin_expect` to provide the compiler with branch prediction information. The return value of `__builtin_expect` is the value of the *exp* argument, which has to be an integral expression. For dynamic prediction, the *value* argument can be either a compile-time constant or a variable. The `__builtin_expect` function assumes that *exp* equals *value*.

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

Dynamic Prediction Example
        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 */
```

Compilers may require limiting the complexity of the expression argument because multiple branches could be generated. When this situation occurs, the compiler has to issue a warning if the program's branch expectations are ignored.

Implementation of this extension is not required for the PPU because the PPU only supports static prediction for branches

## 1.7. Inline Assembly

Occasionally, a programmer might not be able to achieve the desired low-level programming result by using only C/C++ language constructs and intrinsic functions. To handle these situations, the use of inline assembly might be necessary, and therefore, it has to be provided. The inline assembly syntax have to match the AT&T assembly syntax implemented by GCC.

The `.balignl` directive may be used within the inline assembly to ensure the known alignment that is needed to achieve effective dual-issue by the hardware.

## 1.8. Target Definitions

Compilers must define `__SPU__` when code is being compiled for the SPU, and `__PPU__` when code is being compiled for the PPU. The availability of these definitions enables the development of code that can be conditionally compiled for either target.

As an example, the following code supports misaligned quadword loads. The `__SPU__` and `__PPU__` defines are used to conditionally select which code to use. The code that is selected will be different depending on the processor target.

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

When compiling for an SPU implementation that supports the optional enhanced double-precision instructions, `__SPU_EDP__` will also be defined. The enhanced double-precision instructions include `DFCEQ`, `DFCGT`, `DFMCEQ`, `DFMCGT`, and `DFTSV`.

# 2. SPU Low-Level Specific and Generic Intrinsics

This chapter describes the minimal set of basic intrinsics and built-ins that make the underlying Instruction Set Architecture (ISA) and Synergistic Processor Element (SPE) hardware accessible from the C programming language. There are three types of intrinsics:

- Specific
- Generic
- Built-ins

Intrinsics may be implemented either internally within the compiler or as macros. However, if an intrinsic is implemented as a macro, restrictions apply with respect to vector literals being passed as arguments. For more details, see section "1.4.6. Vector Literals".

The instruction set may vary among SPU implementations. If an instruction is not supported by the SPU implementation for which the intrinsic is being compiled, special handling shall occur. For specific intrinsics, an error is generated if the targeted SPU does not support the corresponding instruction. For generic intrinsics, an alternate instruction mapping will be generated that achieves an equivalent operation.

Throughout this section, intrinsics which may generate special handling are indicated by a dagger ([†]).

## 2.1. Specific Intrinsics

Specific intrinsics are *specific* in the sense that they have a one-to-one mapping with a single SPU assembly instruction. All specific intrinsics are named using the SPU assembly instruction prefixed by the string si_. For example, the specific intrinsic that implements the stop assembly instruction is named si_stop.

A specific intrinsic exists for nearly every assembly instruction. However, the functionality provided by several of the assembly instructions is better provided by the C/C++ language; therefore, for these instructions no specific intrinsic has been provided. Table 2-9 describes the assembly instructions that have no corresponding specific intrinsic.

Table 2-9: Assembly Instructions for which No Specific Intrinsic Exists

| Instruction Type | SPU Instructions |
|---|---|
| Branch Instructions | 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, and bihnze (excluding bisled, bisledd, bislede) |
| Branch Hint Instructions | hbr, hbrp, hbra, and hbrr |
| Interrupt Return Instructions | iret, iretd, and irete |

All specific intrinsics are accessible through generic intrinsics, except for the specific intrinsics shown in Table 2-10. The intrinsics that are not accessible fall into three categories:

- Instructions that are generated using basic variable referencing (that is, using vector and scalar loads and stores)
- Instructions that are used for immediate vector construction
- Instructions that have limited usefulness and are not expected to be used except in rare conditions

Table 2-10: Specific Intrinsics Not Accessible Through Generic Intrinsics

| Instruction/Description | Usage | Assembly Mapping |
|---|---|---|
| Generate Controls for Sub-Quadword Insertion | | |
| *si_cbd: Generate Controls for Byte Insertion (d-form)*<br><br>An effective address is computed by adding the value in the signed 7-bit immediate *imm* to word element 0 of *a*. The rightmost 4 bits of the effective address are used to determine the position of the addressed byte within a quadword. Based on the position, a pattern is generated that can be used with the si_shufb intrinsic to insert a byte (byte element 3) at the indicated position within a quadword. The pattern is returned in quadword *d*. | *d* = si_cbd(*a*, *imm*) | CBD d, imm(a) |
| *si_cbx: Generate Controls for Byte Insertion (x-form)*<br><br>An effective address is computed by adding the value of word element 0 of *a* to word element 0 of *b*. The rightmost 4 bits of the effective address are used to determine the position of the addressed byte within a quadword. Based on the position, a pattern is generated that can be used with the si_shufb intrinsic to insert a byte (byte element 3) at the indicated position within a quadword. The pattern is returned in quadword *d*. | *d* = si_cbx(*a*, *b*) | CBX d, a, b |
| *si_cdd: Generate Controls for Doubleword Insertion (d-form)*<br><br>An effective address is computed by adding the value in the signed 7-bit immediate *imm* to word element 0 of *a*. The rightmost 4 bits of the effective address are used to determine the position of the addressed doubleword within a quadword. Based on the position, a pattern is generated that can be used with the si_shufb intrinsic to insert a doubleword (doubleword element 0) at the indicated position within a quadword. The pattern is returned in quadword *d*. | *d* = si_cdd(*a*, *imm*) | CDD d, imm(a) |
| *si_cdx: Generate Controls for Doubleword Insertion (x-form)*<br><br>An effective address is computed by adding the value of word element 0 of *a* to word element 0 of *b*. The rightmost 4 bits of the effective address are used to determine the position of the addressed doubleword within a quadword. Based on the position, a pattern is generated that can be used with the si_shufb intrinsic to insert a doubleword (doubleword element 3) at the indicated position within a quadword. The pattern is returned in quadword *d*. | *d* = si_cdx(*a*, *b*) | CDX d, a, b |
| *si_chd: Generate Controls for Halfword Insertion (d-form)*<br><br>An effective address is computed by adding the value in the signed 7-bit immediate *imm* to word element 0 of *a*. The rightmost 4 bits of the effective address are used to determine the position of the addressed halfword within a quadword. Based on the position, a pattern is generated that can be used with the si_shufb intrinsic to insert a halfword (halfword element 1) at the indicated position within a quadword. The pattern is returned in quadword *d*. | *d* = si_chd(*a*, *imm*) | CHD d, imm(a) |
| *si_chx: Generate Controls for Halfword Insertion (x-form)*<br><br>An effective address is computed by adding the value of word element 0 of *a* to word element 0 of *b*. The rightmost 4 bits of the effective address are used to determine the position of the addressed halfword within a quadword. Based on the position, a pattern is generated that can be used with the si_shufb intrinsic to insert a halfword (halfword element 1) at the indicated position within a quadword. The pattern is returned in quadword *d*. | *d* = si_chx(*a*, *b*) | CHX d, a, b |

| Instruction/Description | Usage | Assembly Mapping |
|---|---|---|
| *si_cwd: Generate Controls for Word Insertion (d-form)* <br><br> An effective address is computed by adding the value in the signed 7-bit immediate *imm* to word element 0 of *a*. The rightmost 4 bits of the effective address are used to determine the position of the addressed word within a quadword. Based on the position, a pattern is generated that can be used with the si_shufb intrinsic to insert a word (word element 0) at the indicated position within a quadword. The pattern is returned in quadword *d*. | *d* = si_cwd(*a*, *imm*) | CWD d, imm(a) |
| *si_cwx: Generate Controls for Word Insertion (x-form)* <br><br> An effective address is computed by adding the value of word element 0 of *a* to word element 0 of *b*. The rightmost 4 bits of the effective address are used to determine the position of the addressed word within a quadword. Based on the position, a pattern is generated that can be used with the si_shufb intrinsic to insert a word (element 0) at the indicated position within a quadword. The pattern is returned in quadword *d*. | *d* = si_cwx(*a*, *b*) | CWX d, a, b |
| Constant Formation Intrinsics | | |
| *si_il: Immediate Load Word* <br><br> The 16-bit signed immediate value *imm* is sign-extended to 32 bits and placed into each of the 4 word elements of quadword *d*. | *d* = si_il(*imm*) | IL d, imm |
| *si_ila: Immediate Load Address* <br><br> The 18-bit immediate value *imm* is placed in the rightmost bits of each of the 4 word elements of quadword *d*. The upper 14 bits of each word is set to 0. | *d* = si_ila(*imm*) | ILA d, imm |
| *si_ilh: Immediate Load Halfword* <br><br> The 16-bit signed immediate value *imm* is placed in each of the 8 halfword elements of quadword *d*. | *d* = si_ilh(*imm*) | ILH d, imm |
| *si_ilhu: Immediate Load Halfword Upper* <br><br> The 16-bit signed immediate value *imm* is placed into the leftmost 16 bits each of the 4 word elements of quadword *d*. The rightmost 16 bits are set to 0. | *d* = si_ilhu(*imm*) | ILHU d, imm |
| *si_iohl: Immediate or Halfword Lower* <br><br> The 16-bit immediate value *imm* is prepended with zeros and ORed with each of the 4 word elements of quadword *a*. The result is returned in quadword *d*. | *d* = si_iohl(*a*, *imm*) | rt <--- a <br> IOHL rt, imm <br> d <--- rt |
| No Operation Intrinsics | | |
| *si_lnop: No Operation (load)* <br><br> A no-operation is performed on the load pipeline. | si_lnop() | LNOP |
| *si_nop: No Operation (execute)* <br><br> A no-operation is performed on the execute pipeline. | si_nop() | NOP rt[1] |
| Memory Load and Store Intrinsics | | |
| *si_lqa: Load Quadword (a-form)* <br><br> An effective address is determined by the sign-extended 18-bit value *imm*, with the 4 least significant bits forced to zero. The quadword at this effective address is returned in quadword *d*. | *d* = si_lqa(*imm*) | LQA d, imm |
| *si_lqd: Load Quadword (d-form)* <br><br> An effective address is computed by zeroing the 4 least significant bits of the sign-extended 14-bit immediate value *imm*, adding *imm* to word element 0 of quadword *a*, and forcing the 4 least significant bits of the result to zero. The quadword at this effective address is then returned in quadword *d*. | *d* = si_lqd(*a*, *imm*) | LQD d, imm(a) |

| Instruction/Description | Usage | Assembly Mapping |
|---|---|---|
| *si_lqr: Load Quadword Instruction Relative (a-form)*<br><br>An effective address is computed by forcing the 2 least significant bits of the signed 18-bit immediate value *imm* to zero, adding this value to the address of the instruction, and forcing the 4 least significant bits of the result to zero. The quadword at this effective address is then returned in quadword *d*. | *d* = si_lqr(*imm*) | LQR, d, imm |
| *si_lqx: Load Quadword (x-form)*<br><br>An effective address is computed by adding word element 0 of quadword *a* to word element 0 of quadword *b* and forcing the 4 least significant bits to zero. The quadword at this effective address is then returned in quadword *d*. | *d* = si_lqx(*a*, *b*) | LQX d, a, b |
| *si_stqa: Store Quadword (a-form)*<br><br>An effective address is determined by the sign-extended 18-bit value *imm*, with the 4 least significant bits forced to zero. The quadword *a* is stored at this effective address. | si_stqa(*a*, *imm*) | STQA a, imm |
| *si_stqd: Store Quadword (d-form)*<br><br>An effective address is computed by zeroing the 4 least significant bits of the sign-extended 14-bit immediate value *imm*, adding *imm* to word element 0 of quadword *b*, and forcing the 4 least significant bits to zero. The quadword *a* is then stored at this effective address. | si_stqd(*a*, *b*, *imm*) | STQD a, imm(b) |
| *si_stqr: Store Quadword Instruction Relative (a-form)*<br><br>An effective address is computed by forcing the 2 least significant bits of the signed 18-bit immediate value *imm* to zero, adding this value to the address of the instruction, and forcing the 4 least significant bits of the result to zero. The quadword *a* is then stored at this effective address. | si_stqr(*a*, *imm*) | STQR, a, imm |
| *si_stqx: Store Quadword (x-form)*<br><br>An effective address is computed by adding word element 0 of quadword *b* to word element 0 of quadword *c* and forcing the 4 least significant bits to zero. The quadword *a* is then stored at this effective address. | si_stqx(*a*, *b*, *c*) | STQX a, b, c |
| Control Intrinsics | | |
| *si_stopd: Stop and Signal with Dependencies*<br><br>Execution of the SPU is stopped and a signal type of `0x3FFF` is delivered after all register dependencies are met. This intrinsic is considered volatile with respect to all instructions and will not be reordered with any other instructions. | si_stopd(*a*, *b*, *c*) | STOPD a, b, c |

[1] The false target parameter `rt` is optimally chosen depending on the register usage of neighboring instructions.

Specific intrinsics accept only the following types of arguments:

- Immediate literals, as an explicit constant expression or as a symbolic address
- Enumerations
- `qword` arguments

Arguments of other types must be cast to `qword`.

For complete details on the specific instructions, see the *Synergistic Processor Unit Instruction Set Architecture*.

### 2.1.1. Specific Casting Intrinsics

When using specific intrinsics, it may be necessary to cast from scalar types to the `qword` data type, or from the `qword` data type to scalar types. Similar to casting between vector data types, specific cast intrinsics have no effect on an argument that is stored in a register. All specific casting intrinsics are of the following form:

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

See Table 2-11 for additional details about the specific casting intrinsics.

Table 2-11: Specific Casting Intrinsics

| Casting Intrinsic | Return/Argument Types | | Description |
| --- | --- | --- | --- |
| | d | a | |
| si_to_char | signed char | | Cast byte element 3 of qword $a$ to signed char $d$. |
| si_to_uchar | unsigned char | | Cast byte element 3 of qword $a$ to unsigned char $d$. |
| si_to_short | short | | Cast halfword element 1 of qword $a$ to short $d$. |
| si_to_ushort | unsigned short | | Cast halfword element 1 of qword $a$ to unsigned short $d$. |
| si_to_int | int | | Cast word element 0 of qword $a$ to int $d$. |
| si_to_uint | unsigned int | qword | Cast word element 0 of qword $a$ to unsigned int $d$. |
| si_to_ptr | void * | | Cast word element 0 of qword $a$ to a void pointer $d$. |
| si_to_llong | long long | | Cast doubleword element 0 of qword $a$ to long long $d$. |
| si_to_ullong | unsigned long long | | Cast doubleword element 0 of qword $a$ to unsigned long long $d$. |
| si_to_float | float | | Cast word element 0 of qword $a$ to float $d$. |
| si_to_double | double | | Cast doubleword element 0 of qword $a$ to double $d$. |
| si_from_char | | signed char | Cast signed char $a$ to byte element 3 of qword $d$. |
| si_from_uchar | | unsigned char | Cast unsigned char $a$ to byte element 3 of qword $d$. |
| si_from_short | | short | Cast short $a$ to halfword element 1 of qword $d$. |
| si_from_ushort | | unsigned short | Cast unsigned short $a$ to halfword element 1 of qword $d$. |
| si_from_int | | int | Cast int $a$ to word element 0 of qword $d$. |
| si_from_uint | qword | unsigned int | Cast unsigned int $a$ to word element 0 of qword $d$. |
| si_from_ptr | | void * | Cast void pointer $a$ to word element 0 of qword $d$. |
| si_from_llong | | long long | Cast long long $a$ to doubleword element 0 of qword $d$. |
| si_from_ullong | | unsigned long long | Cast unsigned long long $a$ to doubleword element 0 of |
| si_from_float | | float | Cast float $a$ to word element 0 of qword $d$. |
| si_from_double | | double | Cast double $a$ to doubleword element 0 of qword $d$. |

Because the casting intrinsics do not perform data conversion, casting from a scalar type to a *qword* type results in portions of the quadword being undefined.

## 2.2. Generic Intrinsics and Built-ins

Generic intrinsics are operations that map to one or more specific intrinsics. The mapping of a generic intrinsic to a specific intrinsic depends on the input arguments to the intrinsic. Built-ins are similar to generic intrinsics; however, unlike generic intrinsics, built-ins map to more than one SPU instruction. All generic intrinsics and built-ins are prefixed by the string spu_. For example, the generic intrinsic that implements the stop assembly instruction is named spu_stop.

### 2.2.1. Mapping Intrinsics with Scalar Operands

Intrinsics with scalar arguments are introduced for SPU instructions with immediate fields. For example, the intrinsic function vector signed int spu_add(vector signed int, int) will translate to an AI assembly instruction.

Depending on the assembly instruction, immediate values are either 7, 10, 16, or 18 bits in length. The action performed for out-of-range immediate values depends on the type of intrinsic. By default, immediate-form specific intrinsics with an out-of-range immediate value are flagged as an error. Compilers may provide an option to issue a warning for out-of-range immediate values and use only the specified number of least significant bits for the out-of-range argument.

Generic intrinsics support a full range of scalar operands. This support is not dependent on whether the scalar operand can be represented within the instruction's immediate field. Consider the following example:

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

Depending on argument *b*, different instructions are generated:

- If *b* is a literal constant within the range supported by one of the immediate forms, the immediate instruction form is generated. For example, if *b* equals 1, then `ANDI d, a, 1` is generated.

- If *b* is a literal constant and is out-of-range but can be folded and implemented using an alternate immediate instruction form, the alternate immediate instruction is generated. For example, if *b* equals `0x30003`, then `ANDHI d, a, 3` is generated. In this context, "alternate immediate instruction form" means an immediate instruction form having a smaller data element size.

- If *b* is a literal constant that can be constructed using one or two immediate load instructions followed by the non-immediate form of the instruction, the appropriate instructions will be used. Immediate load instructions include `IL`, `ILH`, `ILHU`, `ILA`, `IOHL`, and `FSMBI`. Table 2-12 shows possible uses of the immediate load instructions for various constants *b*.

Table 2-12: Possible Uses of Immediate Load Instructions for Various Values of Constant b

| Constant *b* | Generates Instructions |
|---|---|
| -6000 | IL  b, -6000<br>AND   d, a, b |
| 131074  (0x20002) | ILH   b, 2<br>AND   d, a, b |
| 131072  (0x20000) | ILHU   b, 2<br>AND   d, a, b |
| 134000  (0x20B70) | ILA   b, 134000<br>AND   d, a, b |
| 262780  (0x4027C) | ILHU   b, 4<br>IOHL   b, 636<br>AND   d, a, b |
| (0xFFFFFFFF, 0x0, 0x0, 0xFFFFFFFF) | FSMBI b, 0xF00F<br>AND    d, a, b |

- If *b* is a variable (non-literal) integer, code to splat the integer across the entire vector is generated followed by the non-immediate form of the instruction. For example, if *b* is an integer of unknown value, the constant area is loaded with the shuffle pattern (`0x10203, 0x10203, 0x10203, 0x10203`) at "`CONST_AREA, offset`" and the following instructions are generated:

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

### 2.2.2. Implicit Conversion of Arguments of Intrinsics

There is no implicit conversion of arguments that have a vector type. Arguments of scalar type are converted according to the rules specified in the C/C++ standards. Consider, for example,

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

Scalar *a* is inserted into the element of vector *b* that is specified by the *element* parameter. When *b* is a `vector double`, *a* must be converted to `double`, *element* must be converted to `int`, and *d* must be a `vector double`.

### 2.2.3. Notations and Conventions

The remaining documentation describing the generic intrinsics uses the following rules and naming conventions:

- The table associated with each generic intrinsic specifies the supported input types.

- For intrinsics with scalar operands, only the immediate form of the instruction is shown. The other forms can be deduced in accordance with the rules discussed in section "2.2.1. Mapping Intrinsics with Scalar Operands".

- Some intrinsics, whether specific or generic, map to assembly instructions that do not uniquely specify all input and output registers. Instead, an input register also serves as the output register. Examples of these assembly instructions include `ADDX`, `DFMS`, `MPYHHA`, and `SFX`. For these intrinsics, the notation $rt <--- c$ is used to imply that a register-to-register copy (copy $c$ to $rt$) might be required to satisfy the semantics of the intrinsic, depending on the inputs and outputs. No copies will be generated if input $c$ is the same as output $d$.

- Generic intrinsics that do not map to specific intrinsics are identified by the acronym "N/A" (not applicable) in the Specific Intrinsics column of the respective table.

## 2.3. Constant Formation Intrinsics

### spu_splats: Splat Scalar to Vector

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

A single scalar value is replicated across all elements of a vector of the same type. The result is returned in vector $d$.

Table 2-13: Splat Scalar to Vector

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | a | | |
| vector unsigned char | unsigned char | N/A | SHUFB d, a, a, pattern |
| vector signed char | signed char | | |
| vector unsigned short | unsigned short | | |
| vector signed short | signed short | | |
| vector unsigned int | unsigned int | | |
| vector signed int | signed int | | |
| vector unsigned long long | unsigned long long | | |
| vector signed long long | signed long long | | |
| vector float | float | | |
| vector double | double | | |
| vector unsigned char | unsigned char (literal) | N/A | IL d, a<br>   or<br>ILA d, a<br>   or<br>ILH d, a&0xFFFF<br>   or<br>ILHU d, a>>16<br>   or<br>ILHU d, a>>16;<br>IOHL d, a<br>   or<br>FSMBI  d, a |
| vector signed char | signed char (literal) | | |
| vector unsigned short | unsigned short (literal) | | |
| vector signed short | signed short (literal) | | |
| vector unsigned int | unsigned int (literal) | | |
| vector signed int | signed int (literal) | | |
| vector unsigned long long | unsigned long long (literal) | | |
| vector signed long long | signed long long (literal) | | |
| vector float | float (literal) | | |
| vector double | double (literal) | | |

## 2.4. Conversion Intrinsics

**spu_convtf: Convert Integer Vector to Vector Float**

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

Each element of vector $a$ is converted to a floating-point value and divided by $2^{scale}$. The allowable range for $scale$ is 0 to 127. Values outside this range are flagged as an error and compilation is terminated. The result is returned in vector $d$.

Table 2-14: Convert Integer Vector to Vector Float

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | scale | | |
| vector float | vector unsigned int | unsigned int (7-bit literal) | $d$ = si_cuflt($a$, $scale$) | CUFLT d, a, scale |
| vector float | vector signed int | | $d$ = si_csflt($a$, $scale$) | CSFLT d, a, scale |

**spu_convts: Convert Vector Float to Signed Integer Vector**

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

Each element of vector $a$ is scaled by $2^{scale}$, and the result is converted to a signed integer. If the intermediate result is greater than $2^{31}-1$, the result saturates to $2^{31}-1$. If the intermediate value is less than $-2^{31}$, the result saturates to $-2^{31}$. The allowable range for $scale$ is 0 to 127. Values outside this range are flagged as an error and compilation is terminated. The results are returned in the corresponding elements of vector $d$.

Table 2-15: Convert Vector Float to Signed Integer Vector

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | scale | | |
| vector signed int | vector float | unsigned int (7-bit literal) | $d$ = si_cflts($a$, $scale$) | CFLTS d, a, scale |

**spu_convtu: Convert Vector Float to Unsigned Integer Vector**

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

Each element of vector $a$ is scaled by $2^{scale}$ and the result is converted to an unsigned integer. If the intermediate result is greater than $2^{32}-1$, the result saturates to $2^{32}-1$. If the intermediate value is negative, the result saturates to zero. The allowable range for $scale$ is 0 to 127. Values outside this range are flagged as an error and compilation is terminated. The results are returned in the corresponding elements of vector $d$.

Table 2-16: Convert Vector Float to Unsigned Integer Vector

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | scale | | |
| vector unsigned int | vector float | unsigned int (7-bit | $d$ = si_cfltu($a$, $scale$) | CFLTU d, a, scale |

**spu_extend: Extend Vector**

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

For a fixed-point vector $a$, each odd element of vector $a$ is extended to a double and returned in the corresponding element of vector $d$. For a floating-point vector, each even element of $a$ is sign-extended and returned in the corresponding element of $d$.

Table 2-17: Extend Vector

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | a | | |
| vector signed short | vector signed char | $d$ = si_xsbh($a$) | XSBH d, a |
| vector signed int | vector signed short | $d$ = si_xshw($a$) | XSHW d, a |
| vector signed long long | vector signed int | $d$ = si_xswd($a$) | XSWD d, a |
| vector double | vector float | $d$ = si_fesd($a$) | FESD d, a |

**spu_roundtf: Round Vector Double to Vector Float**

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

Each doubleword element of vector $a$ is rounded to a single-precision floating-point value and placed in the even element of vector $d$. Zeros are placed in the odd elements of $d$.

Table 2-18: Round Vector Double to Vector Float

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | a | | |
| vector float | vector double | $d$ = si_frds($a$) | FRDS d, a |

## 2.5. Arithmetic Intrinsics

**spu_add: Vector Add**

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

Each element of vector $a$ is added to the corresponding element of vector $b$. If $b$ is a scalar, the scalar value is replicated for each element and then added to $a$. Overflows and carries are not detected, and no saturation is performed. The results are returned in the corresponding elements of vector $d$.

Table 2-19: Vector Add

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector signed int | vector signed int | vector signed int | $d$ = si_a($a$, $b$) | A d, a, b |
| vector unsigned int | vector unsigned int | vector unsigned int | | |
| vector signed short | vector signed short | vector signed short | $d$ = si_ah($a$, $b$) | AH d, a, b |
| vector unsigned short | vector unsigned short | vector unsigned short | | |
| vector signed int | vector signed int | 10-bit signed int (literal) | $d$ = si_ai($a$, $b$) | AI d, a, b |
| vector unsigned int | vector unsigned int | | | |
| vector signed int | vector signed int | int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector unsigned int | vector unsigned int | unsigned int | | |
| vector signed short | vector signed short | 10-bit signed short (literal) | $d$ = si_ahi($a$, $b$) | AHI d, a, b |
| vector unsigned short | vector unsigned short | | | |

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector signed short | vector signed short | short | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| 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)
```

Each element of vector $a$ is added to the corresponding element of vector $b$ and to the least significant bit of the corresponding element of vector $c$. The result is returned in the corresponding element of vector $d$.

Table 2-20: Vector Add Extended

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | c | | |
| vector signed int | vector signed int | vector signed int | vector signed int | $d$ = si_addx( $a$, $b$, $c$) | rt <--- c<br>ADDX rt, a, b<br>d <--- rt |
| vector unsigned int | vector unsigned int | vector unsigned int | vector unsigned int | | |

### spu_genb: Vector Generate Borrow

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

Each element of vector $b$ is subtracted from the corresponding element of vector $a$. The resulting borrow out is placed in the least significant bit of the corresponding element of vector $d$. The remaining bits of d are set to 0.

Table 2-21: Vector Generate Borrow

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector signed int | vector signed int | vector signed int | $d$ = si_bg($b$, $a$) | BG rt, b, a |
| vector unsigned int | vector unsigned int | vector unsigned int | | |

### spu_genbx: Vector Generate Borrow Extended

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

Each element of vector $b$ is subtracted from the corresponding element of vector $b$. An additional 1 is subtracted from the result if the least significant bit of the corresponding element of vector $c$ is 0. If the result is less than 0, a 1 is placed in the corresponding element of vector d; otherwise, a 0 is placed in the corresponding element of $d$.

Table 2-22: Vector Generate Borrow Extended

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | c | | |
| vector signed int | vector signed int | vector signed int | vector signed int | $d$ = si_bgx( $b$, $a$, $c$) | rt <--- c<br>BGX rt, b, a<br>d <--- rt |
| vector unsigned int | vector unsigned int | vector unsigned int | vector unsigned int | | |

**spu_genc: Vector Generate Carry**

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

Each element of vector $a$ is added to the corresponding element of vector $b$. The resulting carry out is placed in the least significant bit of the corresponding element of vector $d$. The remaining bits of $d$ are set to 0.

Table 2-23: Vector Generate Carry

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector signed int | vector signed int | vector signed int | $d$ = si_cg($a$, $b$) | CG rt, a, b |
| vector unsigned int | vector unsigned int | vector unsigned int | | |

**spu_gencx: Vector Generate Carry Extended**

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

Each element of vector $a$ is added to the corresponding element of vector $b$ and the least significant bit of the corresponding element of vector $c$. The resulting carry out is placed in the least significant bit of the corresponding element of vector $d$. The remaining bits of $d$ are set to 0.

Table 2-24: Vector Generate Carry Extended

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | c | | |
| vector signed int | vector signed int | vector signed int | vector signed int | $d$ = si_cgx( $a$, $b$, $c$) | rt <--- c<br>CGX rt, a, b<br>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)
```

Each element of vector $a$ is multiplied by vector $b$ and added to the corresponding element of vector $c$. The result is returned to the corresponding element of vector $d$. For integer multiply-and-adds, the odd elements of vectors $a$ and $b$ are sign-extended to 32-bit integers prior to multiplication.

Table 2-25: Vector Multiply and Add

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | c | | |
| vector signed int | vector signed short | vector signed short | vector signed int | $d$ = si_mpya( $a$, $b$, $c$) | MPYA d, a, b, c |
| vector float | vector float | vector float | vector float | $d$ = si_fma( $a$, $b$, $c$) | FMA d, a, b, c |
| vector double | vector double | vector double | vector double | $d$ = si_dfma( $a$, $b$, $c$) | rt <--- c<br>DFMA rt, a, b<br>d <--- rt |

**spu_mhhadd: Vector Multiply High High and Add**

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

Each even element of vector $a$ is multiplied by the corresponding even element of vector $b$, the 32-bit result is added to the corresponding element of vector $c$, and the result is returned in the corresponding element of vector $d$.

Table 2-26: Vector Multiply High High and Add

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | c | | |
| vector signed int | vector signed short | vector signed short | vector signed int | $d$ = si_mpyhha($a, b, c$) | rt <--- c<br>MPYHHA rt, a, b<br>d <--- rt |
| vector unsigned int | vector unsigned short | vector unsigned short | vector unsigned int | $d$ = si_mpyhhau($a, b, c$) | rt <--- c<br>MPYHHAU rt, a, b<br>d <--- rt |

**spu_msub: Vector Multiply and Subtract**

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

Each element of vector $a$ is multiplied by the corresponding element of vector $b$, and the corresponding element of vector $c$ is subtracted from the product. The result is returned in the corresponding element of vector $d$.

Table 2-27: Vector Multiply and Subtract

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | c | | |
| vector float | vector float | vector float | vector float | $d$ = si_fms($a, b, c$) | FMS d, a, b, c |
| vector double | vector double | vector double | vector double | $d$ = si_dfms($a, b, c$) | rt <--- c<br>DFMS rt, a, b<br>d <--- rt |

**spu_mul: Vector Multiply**

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

Each element of vector $a$ is multiplied by the corresponding element of vector $b$ and returned in the corresponding element of vector $d$.

Table 2-28: Vector Multiply

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector float | vector float | vector float | $d$ = si_fm($a, b$) | FM d, a, b |
| vector double | vector double | vector double | $d$ = si_dfm($a, b$) | DFM d, a, b |

**spu_mulh: Vector Multiply High**

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

Each even element of vector $a$ is multiplied by the next (odd) element of vector $b$. The product is shifted left by 16 bits and stored in the corresponding element of vector $d$. Bits shifted out at the left are discarded. Zeros are shifted in at the right.

Table 2-29: Vector Multiply High

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector signed int | vector signed short | vector signed short | $d$ = si_mpyh($a, b$) | MPYH d, a, b |

**spu_mule: Vector Multiply Even**

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

Each even element of vector $a$ is multiplied by the corresponding even element of vector $b$, and the 32-bit result is returned to the corresponding element of vector $d$.

Table 2-30: Vector Multiply Even

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector signed int | vector signed short | vector signed short | $d$ = si_mpyhh($a$, $b$) | MPYHH d, a, b |
| vector unsigned int | vector unsigned short | vector unsigned short | $d$ = si_mpyhhu($a$, | MPYHHU d, a, b |

**spu_mulo: Vector Multiply Odd**

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

Each odd element of vector $a$ is multiplied by the corresponding element of vector $b$. If $b$ is a scalar, the scalar value is replicated for each element and then multiplied by $a$. The results are returned in vector $d$.

Table 2-31: Vector Multiply Odd

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector signed int | vector signed short | vector signed short | $d$ = si_mpy ($a$, $b$) | MPY d, a, b |
| | | 10-bit signed short (literal) | $d$ = si_mpyi($a$, $b$) | MPYI d, a, b |
| | | signed short | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector unsigned int | vector unsigned short | vector unsigned short | $d$ = si_mpyu($a$, $b$) | MPYU d, a, b |
| | | 10-bit signed short (literal) | $d$ = si_mpyui($a$, $b$) | MPYUI d, a, b |
| | | unsigned short | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |

**spu_mulsr: Vector Multiply and Shift Right**

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

Each odd element of vector $a$ is multiplied by the corresponding odd element of vector $b$. The leftmost 16 bits of the resulting 32-bit product is sign-extended and returned in the corresponding 32-bit element of vector $d$.

Table 2-32: Vector Multiply and Shift Right

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector signed int | vector signed short | vector signed short | $d$ = si_mpys($a$, $b$) | MPYS d, a, b |

**spu_nmadd: Negative Vector Multiply and Add**

    d = spu_nmadd(a, b, c)

Each element of vector $a$ is multiplied by the corresponding element in vector $b$ and then added to the corresponding element of vector $c$. The result is negated and returned in the corresponding element of vector $d$.

Table 2-33: Negative Vector Multiply and Add

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | c | | |
| vector double | vector double | vector double | vector double | $d$ = si_dfnma($a$, $b$, $c$) | rt <-- c<br>DFNMA rt, a, b<br>d <-- rt |

**spu_nmsub: Negative Vector Multiply and Subtract**

    d = spu_nmsub(a, b, c)

Each element of vector $a$ is multiplied by the corresponding element in vector $b$. The result is subtracted from the corresponding element in $c$ and returned in the corresponding element of vector $d$.

Table 2-34: Negative Vector Multiply and Subtract

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | c | | |
| vector float | vector float | vector float | vector float | $d$ = si_fnms($a$, $b$, $c$) | FNMS d, a, b, c |
| vector double | vector double | vector double | vector double | $d$ = si_dfnms($a$, $b$, $c$) | rt <--- c<br>DFNMS rt, a, b<br>d <--- rt |

**spu_re: Vector Floating-Point Reciprocal Estimate**

    d = spu_re(a)

For each element of vector $a$, an estimate of its floating-point reciprocal is computed, and the result is returned in the corresponding element of vector $d$. The resulting estimate is accurate to 12 bits.

Table 2-35: Vector Floating-Point Reciprocal Estimate

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | a | | |
| vector float | vector float | t = si_frest($a$)<br>$d$ = si_fi($a$, $t$) | FREST d, a<br>FI d, a, d |

**spu_rsqrte: Vector Floating-Point Reciprocal Square Root Estimate**

    d = spu_rsqrte(a)

For each element of vector $a$, an estimate of its floating-point reciprocal square root is computed, and the result is returned in the corresponding element of vector $d$. The resulting estimate is accurate to 12 bits.

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

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | a | | |
| vector float | vector float | t = si_frsqest($a$)<br>$d$ = si_fi($a$, $t$) | FRSQEST d, a<br>FI d, a, d |

**spu_sub: Vector Subtract**

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

Each element of vector *b* is subtracted from the corresponding element of vector *a*. If *a* is a scalar, the scalar value is replicated for each element of *a*, and then *b* is subtracted from the corresponding element of *a*. Overflows and carries are not detected. The results are returned in the corresponding elements of vector *d*.

Table 2-37: Vector Subtract

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector signed short | vector signed short | vector signed short | *d* = si_sfh(b, a) | SFH d, b, a |
| vector unsigned short | vector unsigned short | vector unsigned short | | |
| vector signed int | vector signed int | vector signed int | *d* = si_sf(b, a) | SF d, b, a |
| vector unsigned int | vector unsigned int | vector unsigned int | | |
| vector signed int | 10-bit signed int (literal) | 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 | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector unsigned int | unsigned int | vector unsigned int | | |
| vector signed short | 10-bit signed short (literal) | 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 | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| 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)
```

Each element of vector *b* is subtracted from the corresponding element of vector *a*. An additional 1 is subtracted from the result if the least significant bit of the corresponding element of vector *c* is 0. The final result is returned in the corresponding element of vector *d*.

Table 2-38: Vector Subtract Extended

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | c | | |
| vector signed int | vector signed int | vector signed int | vector signed int | *d* = si_sfx(*b*,*a*, *c*) | rt <--- c SFX rt, b, a d <--- rt |
| vector unsigned int | vector unsigned int | vector unsigned int | vector unsigned int | | |

## 2.6. Byte Operation Intrinsics

**spu_absd: Vector Absolute Difference**

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

Each element of vector *a* is subtracted from the corresponding element of vector *b*, and the absolute value of the result is returned in the corresponding element of vector *d*.

Table 2-39: Vector Absolute Difference

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector unsigned char | vector unsigned char | $d$ = si_absdb($a$, $b$) | ABSDB d, a, b |

**spu_avg: Average of Two Vectors**

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

Each element of vector *a* is added to the corresponding element of vector *b* plus 1. The result is shifted to the right by 1 bit and placed in the corresponding element of vector *d*.

Table 2-40: Average of Two Vectors

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector unsigned char | vector unsigned char | $d$ = si_avgb($a$, $b$) | AVGB d, a, b |

**spu_sumb: Sum Bytes into Shorts**

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

Each four elements of *b* are summed and returned in the corresponding even elements of vector *d*. Each four elements of *a* are summed and returned in the corresponding odd elements of *d*.

Table 2-41: Sum Bytes into Shorts

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned short | vector unsigned char | vector unsigned char | $d$ = si_sumb($a$, $b$) | SUMB d, a, b |

## 2.7. Compare, Branch and Halt Intrinsics

**spu_bisled: Branch Indirect and Set Link if External Data**

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

The count value of channel 0 (event status) is examined. If it is zero, execution continues with the next sequential instruction. If it is nonzero, the function *func* is called. The parameter *func* is the name of, or pointer to, a parameter-less function with no return value. If *func* is called, the spu_bisled_d and spu_bisled_e forms of the intrinsic do one of the following actions:

- Disable interrupts – use spu_bisled_d
- Enable interrupts – use spu_bisled_e

Because the bisled instruction is assumed to behave as a synchronous software interrupt, and because all volatile registers must be considered non-volatile by the bisled target function, `func`, standard calling conventions are not observed. See the *SPU Application Binary Interface Specification* for additional details about standard calling conventions.

With respect to branch prediction, it is assumed that `func` is not called. Therefore, a branch hint instruction will not be inserted as a result of the `spu_bisled()` intrinsic.

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

| Generic Intrinsic Form | func | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| spu_bisled | | si_bisled(*func*) | BISLED $LR, func |
| spu_bisled_d | void (*func) () | si_bisledd(*func*) | BISLEDD $LR, func |
| spu_bisled_e | | si_bislede(*func*) | BISLEDE $LR, func |

**spu_cmpabseq: Vector Compare Absolute Equal**

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

The absolute value of each element of vector `a` is compared with the absolute value of the corresponding element of vector `b`. If the absolute values are equal, all bits of the corresponding element of vector `d` are set to one; otherwise, all bits of the corresponding element of `d` are set to zero.

Table 2-43: Vector Compare Absolute Equal

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned int | vector float | vector float | $d$ = si_fcmeq($a$, $b$) | FCMEQ d, a, b |
| vector unsigned long long | vector double | vector double | $d$ = si_dfcmeq($a$, $b$) | DFCMEQ d, a, b [†] |

**spu_cmpabsgt: Vector Compare Absolute Greater Than**

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

The absolute value of each element of vector `a` is compared with the absolute value of the corresponding element of vector `b`. If the element of `a` is greater than the element of `b`, all bits of the corresponding element of vector `d` are set to one; otherwise, all bits of the corresponding element of `d` are set to zero.

Table 2-44: Vector Compare Absolute Greater Than

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned int | vector float | vector float | $d$ = si_fcmgt($a$, $b$) | FCMGT d, a, b |
| vector unsigned long long | vector double | vector double | $d$ = si_dfcmgt($a$, $b$) | DFCMGT d, a, b [†] |

**spu_cmpeq: Vector Compare Equal**

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

Each element of vector $a$ is compared with the corresponding element of vector $b$. If $b$ is a scalar, the scalar value is first replicated for each element, and then $a$ and $b$ are compared. If the operands are equal, all bits of the corresponding element of vector $d$ are set to one. If they are unequal, all bits of the corresponding element of $d$ are set to zero.

Table 2-45: Vector Compare Equal

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector signed char | vector signed char | $d$ = si_ceqb($a$, $b$) | CEQb d, a, b |
| | vector unsigned char | vector unsigned char | | |
| vector unsigned short | vector signed short | vector signed short | $d$ = si_ceqh($a$, $b$) | CEQH d, a, b |
| | vector unsigned short | vector unsigned short | | |
| vector unsigned int | vector signed int | vector signed int | $d$ = si_ceq($a$, $b$) | CEQ d, a, b |
| | vector unsigned int | vector unsigned int | | |
| | vector float | vector float | $d$ = si_fceq($a$, $b$) | FCEQ d, a, b |
| vector unsigned char | vector signed char | 10-bit signed int (literal) | $d$ = si_ceqbi($a$, $b$) | CEQBI d, a, b |
| | vector unsigned char | | | |
| | vector signed char | signed char | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| | vector unsigned char | unsigned char | | |
| vector unsigned short | vector signed short | 10-bit signed int (literal) | $d$ = si_ceqhi($a$, $b$) | CEQHI d, a, b |
| | vector unsigned short | | | |
| | vector signed short | signed short | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| | vector unsigned short | unsigned short | | |
| vector unsigned int | vector signed int | 10-bit signed int (literal) | $d$ = si_ceqi($a$, $b$) | CEQI d, a, b |
| | vector unsigned int | | | |
| | vector signed int | signed int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| | vector unsigned int | unsigned int | | |
| vector unsigned long long | vector double | vector double | $d$ = si_dfceq($a$, $b$) | DFCEQ d, a, b † |

**spu_cmpgt: Vector Compare Greater Than**

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

Each element of vector $a$ is compared with the corresponding element of vector $b$. If $b$ is a scalar, the scalar value is replicated for each element and then $a$ and $b$ are compared. If the element of $a$ is greater than the corresponding element of $b$, all bits of the corresponding element of vector $d$ are set to one; otherwise, all bits of the corresponding element of $d$ are set to zero.

Table 2-46: Vector Compare Greater Than

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector signed char | vector signed char | $d$ = si_cgtb($a$, $b$) | CGTB d, a, b |
| | | 10-bit signed int (literal) | $d$ = si_cgtbi($a$, $b$) | CGTBI d, a, b |
| | | signed char | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| | vector unsigned char | vector unsigned char | $d$ = si_clgtb($a$, $b$) | CLGTB d, a, b |
| | | 10-bit signed int (literal) | $d$ = si_clgtbi($a$, $b$) | CLGTBI d, a, b |
| | | unsigned char | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector unsigned short | vector signed short | vector signed short | $d$ = si_cgth($a$, $b$) | CGTH d, a, b |
| | | 10-bit signed int (literal) | $d$ = si_cgthi($a$, $b$) | CGTHI d, a, b |
| | | signed short | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| | vector unsigned short | vector unsigned short | $d$ = si_clgth($a$, $b$) | CLGTH d, a, b |
| | | 10-bit signed int (literal) | $d$ = si_clgthi($a$, $b$) | CLGTHI d, a, b |
| | | unsigned short | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector unsigned int | vector signed int | vector signed int | $d$ = si_cgt($a$, $b$) | CGT d, a, b |
| | | 10-bit signed int (literal) | $d$ = si_cgti($a$, $b$) | CGTI d, a, b |
| | | signed int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| | vector unsigned int | vector unsigned int | $d$ = si_clgt($a$, $b$) | CLGT d, a, b |
| | | 10-bit signed int (literal) | $d$ = si_clgti($a$, $b$) | CLGTI d, a, b |
| | | unsigned int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| | vector float | vector float | $d$ = si_fcgt($a$, $b$) | FCGT d, a, b |
| vector unsigned long long | vector double | vector double | $d$ = si_dfcgt($a$, $b$) | DFCGT d, a, b [†] |

**spu_hcmpeq: Halt If Compare Equal**

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

The contents of *a* and *b* are compared. If they are equal, execution is halted.

Table 2-47: Halt If Compare Equal

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping[1,2] |
|---|---|---|---|
| a | b | | |
| int | int (non-literal) | si_heq(*a*, *b*) | HEQ rt, a, b |
| unsigned int | unsigned int (non-literal) | | |
| int | 10-bit signed int (literal) | si_heqi(*a*, *b*) | HEQI rt, a, b |
| unsigned int | | | |

[1] Immediate values that cannot be represented as a 10-bit signed value are constructed similar to the method described in section "2.2.1. Mapping Intrinsics with Scalar Operands".

[2] The false target parameter *rt* is optimally chosen depending on the register usage of neighboring instructions.

**spu_hcmpgt: Halt If Compare Greater Than**

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

The contents of *a* and *b* are compared. If *a* is greater than *b*, execution is halted.

Table 2-48: Halt If Compare Greater Than

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping[1,2] |
|---|---|---|---|
| a | b | | |
| int | int (non-literal) | si_hgt(*a*, *b*) | HGT rt, a, b |
| unsigned int | unsigned int (non-literal) | si_hlgt(*a*, *b*) | HLGT rt, a, b |
| int | 10-bit signed int (literal) | si_hgti(*a*, *b*) | HGTI rt, a, b |
| unsigned int | 10-bit signed int (literal) | si_hlgti(*a*, *b*) | HLGTI rt, a, b |

[1] Immediate values that cannot be represented as a 10-bit signed values are constructed in a way similar to the method described in section "2.2.1. Mapping Intrinsics with Scalar Operands".

[2] The false target parameter *rt* is optimally chosen depending on the register usage of neighboring instructions.

**spu_testsv: Vector Test Special Value**

```
d = spu_testsv(a,values)
```

Each element of vector *a* is compared with the set of special values specified by *values*. If any one of the specified comparisons is true all ones are placed in the corresponding element of vector *d*. If none of the tests are true, zeros are placed in the corresponding element of vector *d*.

Table 2-49: Vector Test Special Value

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | values | | |
| vector unsigned long long | vector double | 7-bit unsigned int (literal) | *d* = si_dftsv(*a*, *values*) | DFTSV d, a, values † |

The set of bit flag mnemonics that can be used to specify a set of special values to be tested is shown in Table 2-50. These mnemonics are defined in `spu_intrinsics.h`.

Table 2-50: Special Value Bit Flag Mnemonics

| Mnemonic | Value | Description |
|---|---|---|

| Mnemonic | Value | Description |
|---|---|---|
| SPU_SV_NEG_DENORM | 0x01 | Test for a negative denormalized number |
| SPU_SV_POS_DENORM | 0x02 | Test for a positive denormalized number |
| SPU_SV_NEG_ZERO | 0x04 | Test for a negative zero |
| SPU_SV_POS_ZERO | 0x08 | Test for a positive zero |
| SPU_SV_NEG_INFINITY | 0x10 | Test for a negative infinity |
| SPU_SV_POS_INFINITY | 0x20 | Test for a positive infinity |
| SPU_SV_NAN | 0x40 | Test for a Not-a-Number, both signalling and quiet |

## 2.8. Bits and Mask Intrinsics

**spu_cntb: Vector Count Ones for Bytes**

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

For each element of vector $a$, the number of ones are counted, and the count is placed in the corresponding element of vector $d$.

Table 2-51: Vector Count Ones for Bytes

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | a | | |
| vector unsigned char | vector unsigned char | si_cntb | CNTB d, a |
| | vector signed char | | |

**spu_cntlz: Vector Count Leading Zeros**

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

For each element of vector $a$, the number of leading zeros is counted, and the resulting count is placed in the corresponding element of vector $d$.

Table 2-52: Vector Count Leading Zeros

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | a | | |
| vector unsigned int | vector signed int | $d$ = si_clz($a$) | CLZ d, a |
| | vector unsigned int | | |
| | vector float | | |

**spu_gather: Gather Bits from Elements**

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

The rightmost bit (LSB) of each element of vector $a$ is gathered, concatenated, and returned in the rightmost bits of element 0 of vector $d$. For a byte vector, 16 bits are gathered; for a halfword vector, 8 bits are gathered; and for a word vector, 4 bits are gathered. The remaining bits of element 0 of $d$ and all other elements of that vector are zeroed.

Table 2-53: Gather Bits from Elements

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | a | | |
| vector unsigned int | vector unsigned char | $d$ = si_gbb($a$) | GBB d, a |
| | vector signed char | | |
| | vector unsigned short | $d$ = si_gbh($a$) | GBH d, a |
| | vector signed short | | |
| | vector unsigned int | $d$ = si_gb($a$) | GB d, a |
| | vector signed int | | |
| | vector float | | |

**spu_maskb: Form Select Byte Mask**

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

For each of the least significant 16 bits of $a$, each bit is replicated 8 times, producing a 128-bit vector mask that is returned in vector $d$.

Table 2-54: Form Select Byte Mask

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | a | | |
| vector unsigned char | unsigned short | $d$ = si_fsmb($a$) | FSMB d, a |
| | signed short | | |
| | unsigned int | | |
| | signed int | | |
| | 16-bit unsigned int (literal) | $d$ = si_fsmbi($a$) | FSMBI d, a |

**spu_maskh: Form Select Halfword Mask**

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

For each of the least significant 8 bits of $a$, each bit is replicated 16 times, producing a 128-bit vector mask that is returned in vector $d$.

Table 2-55: Form Select Halfword Mask

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | a | | |
| vector unsigned short | unsigned char | $d$ = si_fsmh($a$) | FSMH d, a |
| | signed char | | |
| | unsigned short | | |
| | signed short | | |
| | unsigned int | | |
| | signed int | | |

**spu_maskw: Form Select Word Mask**

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

For each of the least significant 4 bits of $a$, each bit is replicated 32 times, producing a 128-bit vector mask that is returned in vector $d$.

Table 2-56: Form Select Word Mask

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | a | | |
| vector unsigned int | unsigned char | $d$ = si_fsm($a$) | FSM d, a |
| | signed char | | |
| | unsigned short | | |
| | signed short | | |
| | unsigned int | | |
| | signed int | | |

**spu_sel: Select Bits**

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

For each bit in the 128-bit vector *pattern*, the corresponding bit from either vector $a$ or vector $b$ is selected. If the bit is 0, the bit from $a$ is selected; otherwise, the bit from $b$ is selected. The result is returned in vector $d$.

Table 2-57: Select Bits

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | pattern | | |
| vector unsigned char | vector unsigned char | vector unsigned char | vector unsigned char | $d$ = si_selb( $a$, $b$, *pattern*) | SELB d, a, b, pattern |
| vector signed char | vector signed char | vector signed char | | | |
| vector unsigned short | vector unsigned short | vector unsigned short | vector unsigned short | | |
| vector signed short | vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | vector unsigned int | vector unsigned int | | |
| vector signed int | vector signed int | vector signed int | | | |
| vector float | vector float | vector float | | | |
| vector unsigned long long | vector unsigned long long | vector unsigned long long | vector unsigned long long | | |
| vector signed long long | vector signed long long | vector signed long long | | | |
| vector double | vector double | vector double | | | |

**spu_shuffle: Shuffle Two Vectors of Bytes**

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

For each byte of *pattern*, the byte is examined, and a byte is produced, as shown in Figure 2-2. The result is returned in the corresponding byte of vector $d$.

Figure 2-2: Shuffle Pattern

| Value in the Byte of pattern (in binary) | Resulting Byte |
|---|---|
| 10xxxxxx | 0x00 |
| 110xxxxx | 0xFF |

| Value in the Byte of pattern (in binary) | Resulting Byte |
|---|---|
| 111xxxxx | 0x80 |
| otherwise | The byte of (a\|\|b) addressed by the rightmost 5 bits of pattern |

Table 2-58: Shuffle Two Vectors of Bytes

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | pattern | | |
| vector unsigned char | vector unsigned char | vector unsigned char | vector unsigned char | $d$ = si_shufb( $a, b, pattern$) | SHUFB d, a, b, pattern |
| vector signed char | vector signed char | vector signed char | | | |
| vector unsigned short | vector unsigned short | vector unsigned short | | | |
| vector signed short | vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | vector unsigned int | | | |
| vector signed int | vector signed int | vector signed int | | | |
| vector unsigned long long | vector unsigned long long | vector unsigned long long | | | |
| vector signed long long | vector signed long long | vector signed long long | | | |
| vector float | vector float | vector float | | | |
| vector double | vector double | vector double | | | |

## 2.9. Logical Intrinsics

### spu_and: Vector Bit-Wise AND

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

Each bit of vector $a$ is logically ANDed with the corresponding bit of vector $b$. If $b$ is a scalar, the scalar value is first replicated for each element, and then $a$ and $b$ are ANDed. The results are returned in the corresponding bit of vector $d$.

Table 2-59: Vector Bit-Wise AND

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector unsigned char | vector unsigned char | $d$ = si_and($a, b$) | AND d, a, b |
| vector signed char | vector signed char | vector signed char | | |
| vector unsigned short | vector unsigned short | vector unsigned short | | |
| vector signed short | vector signed short | vector signed short | | |
| vector unsigned int | vector unsigned int | vector unsigned int | | |
| vector signed int | vector signed int | vector signed int | | |
| vector unsigned long long | vector unsigned long long | vector unsigned long long | | |
| vector signed long long | vector signed long long | vector signed long long | | |
| vector float | vector float | vector float | | |
| vector double | vector double | vector double | | |

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector unsigned char | 10-bit signed int (literal) | $d$ = si_andbi($a$, $b$) | ANDBI d, a, b |
| vector signed char | vector signed char | | | |
| vector unsigned char | vector unsigned char | unsigned char | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed char | vector signed char | signed char | | |
| vector unsigned short | vector unsigned short | 10-bit signed int (literal) | $d$ = si_andhi($a$, $b$) | ANDHI d, a, b |
| vector signed short | vector signed short | | | |
| vector unsigned short | vector unsigned short | unsigned short | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed short | vector signed short | signed short | | |
| vector unsigned int | vector unsigned int | 10-bit signed int (literal) | $d$ = si_andi($a$, $b$) | ANDI d, a, b |
| vector signed int | vector signed int | | | |
| vector unsigned int | vector unsigned int | unsigned int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed int | vector signed int | signed int | | |

### spu_andc: Vector Bit-Wise AND with Complement

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

Each bit of vector $a$ is ANDed with the complement of the corresponding bit of vector $b$. The result is returned in the corresponding bit of vector $d$.

Table 2-60: Vector Bit-Wise AND with Complement

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector unsigned char | vector unsigned char | $d$ = si_andc( $a$, $b$) | ANDC d, a, b |
| vector signed char | vector signed char | vector signed char | | |
| vector unsigned short | vector unsigned short | vector unsigned short | | |
| vector signed short | vector signed short | vector signed short | | |
| vector unsigned int | vector unsigned int | vector unsigned int | | |
| vector signed int | vector signed int | vector signed int | | |
| vector unsigned long | vector unsigned long | vector unsigned 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)
```

Each bit of vector $a$ is compared with the corresponding bit of vector $b$. The corresponding bit of vector $d$ is set to 1 if the bits in $a$ and $b$ are equivalent; otherwise, the bit is set to 0.

Table 2-61: Vector Bit-Wise Equivalent

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector unsigned char | vector unsigned char | $d$ = si_eqv($a$, $b$) | EQV d, a, b |
| vector signed char | vector signed char | vector signed char | | |
| vector unsigned short | vector unsigned short | vector unsigned short | | |
| vector signed short | vector signed short | vector signed short | | |
| vector unsigned int | vector unsigned int | vector unsigned int | | |
| vector signed int | vector signed int | vector signed int | | |
| vector unsigned long long | vector unsigned long long | vector unsigned long long | | |
| vector signed long long | vector signed long long | vector signed long long | | |
| vector float | vector float | vector float | | |
| vector double | vector double | vector double | | |

**spu_nand: Vector Bit-Wise Complement of AND**

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

Each bit of vector $a$ is ANDed with the corresponding bit of vector $b$. The complement of the result is returned in the corresponding bit of vector $d$.

Table 2-62: Vector Bit-Wise Complement of AND

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector unsigned char | vector unsigned char | $d$ = si_nand($a$, $b$) | NAND d, a, b |
| vector signed char | vector signed char | vector signed char | | |
| vector unsigned short | vector unsigned short | vector unsigned short | | |
| vector signed short | vector signed short | vector signed short | | |
| vector unsigned int | vector unsigned int | vector unsigned int | | |
| vector signed int | vector signed int | vector signed int | | |
| vector unsigned long long | vector unsigned long long | vector unsigned long long | | |
| vector signed long long | vector signed long long | vector signed long long | | |
| vector float | vector float | vector float | | |
| vector double | vector double | vector double | | |

**spu_nor: Vector Bit-Wise Complement of OR**

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

Each bit of vector $a$ is ORed with the corresponding bit of vector $b$. The complement of the result is returned in the corresponding bit of vector $d$.

Table 2-63: Vector Bit-Wise Complement of OR

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector unsigned char | vector unsigned char | $d$ = si_nor($a$, $b$) | NOR d, a, b |
| vector signed char | vector signed char | vector signed char | | |
| vector unsigned short | vector unsigned short | vector unsigned short | | |
| vector signed short | vector signed short | vector signed short | | |
| vector unsigned int | vector unsigned int | vector unsigned int | | |
| vector signed int | vector signed int | vector signed int | | |
| vector unsigned long long | vector unsigned long long | vector unsigned long long | | |
| vector signed long long | vector signed long long | vector signed long long | | |
| vector float | vector float | vector float | | |
| vector double | vector double | vector double | | |

**spu_or: Vector Bit-Wise OR**

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

Each bit of vector $a$ is logically ORed with the corresponding bit of vector $b$. If $b$ is a scalar, the scalar value is first replicated for each element, and then $a$ and $b$ are ORed. The result is returned in the corresponding bit of vector $d$.

Table 2-64: Vector Bit-Wise OR

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector unsigned char | vector unsigned char | $d$ = si_or($a$, $b$) | OR d, a, b |
| vector signed char | vector signed char | vector signed char | | |
| vector unsigned short | vector unsigned short | vector unsigned short | | |
| vector signed short | vector signed short | vector signed short | | |
| vector unsigned int | vector unsigned int | vector unsigned int | | |
| vector signed int | vector signed int | vector signed int | | |
| vector unsigned long long | vector unsigned long long | vector unsigned long | | |
| 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 (literal) | $d$ = si_orbi($a$, $b$) | ORBI d, a, b |
| vector signed char | vector signed char | | | |
| vector unsigned char | vector unsigned char | unsigned char | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed char | vector signed char | signed char | | |
| vector unsigned short | vector unsigned short | 10-bit signed int (literal) | $d$ = si_orhi($a$, $b$) | ORHI d, a, b |

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector signed short | vector signed short | | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector unsigned short | vector unsigned short | unsigned short | | |
| vector signed short | vector signed short | signed short | | |
| vector unsigned int | vector unsigned int | 10-bit signed int (literal) | $d$ = si_ori($a$, $b$) | ORI d, a, b |
| vector signed int | vector signed int | | | |
| vector unsigned int | vector unsigned int | unsigned int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed int | vector signed int | signed int | | |

**spu_orc: Vector Bit-Wise OR with Complement**

    d = spu_orc(a, b)

Each bit of vector $a$ is ORed with the complement of the corresponding bit of vector $b$. The result is returned in the corresponding bit of vector $d$.

Table 2-65: Vector Bit-Wise OR with Complement

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector unsigned char | vector unsigned char | $d$ = si_orc($a$, $b$) | ORC d,a, b |
| vector signed char | vector signed char | vector signed char | | |
| vector unsigned short | vector unsigned short | vector unsigned short | | |
| vector signed short | vector signed short | vector signed short | | |
| vector unsigned int | vector unsigned int | vector unsigned int | | |
| vector signed int | vector signed int | vector signed int | | |
| vector unsigned long long | vector unsigned long long | vector unsigned long long | | |
| vector signed long long | vector signed long long | vector signed long long | | |
| vector float | vector float | vector float | | |
| vector double | vector double | vector double | | |

**spu_orx: OR Word Across**

    d = spu_orx(a)

The four word elements of vector $a$ are logically ORed. The result is returned in word element 0 of vector $d$. All other elements (1,2,3) of $d$ are assigned a value of zero.

Table 2-66: OR Word Across

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | a | | |
| vector unsigned int | vector unsigned int | $d$ = si_orx($a$) | ORX d, a |
| vector signed int | vector signed int | | |

**spu_xor: Vector Bit-Wise Exclusive OR**

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

Each element of vector `a` is exclusive-ORed with the corresponding element of vector `b`. If `b` is a scalar, the scalar value is first replicated for each element. The result is returned in the corresponding bit of vector `d`.

Table 2-67: Vector Bit-Wise Exclusive OR

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | | |
| vector unsigned char | vector unsigned char | vector unsigned char | $d$ = si_xor($a, b$) | XOR d, a, b |
| vector signed char | vector signed char | vector signed char | | |
| vector unsigned short | vector unsigned short | vector unsigned short | | |
| vector signed short | vector signed short | vector signed short | | |
| vector unsigned int | vector unsigned int | vector unsigned int | | |
| vector signed int | vector signed int | vector signed int | | |
| vector unsigned long long | vector unsigned long long | vector unsigned long long | | |
| vector signed long long | vector signed long long | vector signed long long | | |
| vector float | vector float | vector float | | |
| vector double | vector double | vector double | | |
| vector unsigned char | vector unsigned char | 10-bit signed int (literal) | $d$ = si_xorbi($a, b$) | XORBI d, a, b |
| vector signed char | vector signed char | | | |
| vector unsigned char | vector unsigned char | unsigned char | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed char | vector signed char | signed char | | |
| vector unsigned short | vector unsigned short | 10-bit signed int (literal) | $d$ = si_xorhi($a, b$) | XORHI d, a, b |
| vector signed short | vector signed short | | | |
| vector unsigned short | vector unsigned short | unsigned short | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed short | vector signed short | signed short | | |
| vector unsigned int | vector unsigned int | 10-bit signed int (literal) | $d$ = si_xori($a, b$) | XORI d, a, b |
| vector signed int | vector signed int | | | |
| vector unsigned int | vector unsigned int | unsigned int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed int | vector signed int | signed int | | |

## 2.10. Shift and Rotate Intrinsics

**spu_rl: Vector Rotate Left by Bits**

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

Each element of vector `a` is rotated left by the number of bits specified by the corresponding element in vector `count`. Bits rotated out of the left end of the element are rotated in at the right end. A limited number of `count` bits are used depending on the size of the element. For halfword elements, the 4 least significant bits of `count` are used. For word elements, the 5 least significant bits of `count` are used.

The results are returned in the corresponding elements of vector `d`.

Table 2-68: Vector Rotate Left by Bits

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned short | vector unsigned short | vector signed short | $d$ = si_roth($a$, $count$) | ROTH d, a, count |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | vector signed int | $d$ = si_rot($a$, $count$) | ROT d, a, count |
| vector signed int | vector signed int | | | |

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned short | vector unsigned short | 7-bit signed int (literal) | $d$ = si_rothi($a$, $count$) | ROTHI d, a, count |
| vector signed short | vector signed short | | | |
| vector unsigned short | vector unsigned short | int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | 7-bit signed int (literal) | $d$ = si_roti($a$, $count$) | ROTI d, a, count |
| vector signed int | vector signed int | | | |
| vector unsigned int | vector unsigned int | int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed int | vector signed int | | | |

**spu_rlmask: Vector Rotate Left and Mask by Bits**

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

This function uses an element-wise rotate left and mask operation to perform a logical shift right (LSR) by bits of each element of vector $a$, where $count$ represents the negated value, or values, of the desired corresponding right-shift amounts. (The $count$ parameter can be either a vector or a scalar, as shown in Table 2-69.) For example, if scalar $count$ is –5, each element of $a$ is shifted right by 5 bits. The effect of this function is more precisely shown by the following code:

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

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

The results are returned in the corresponding elements of vector $d$.

Table 2-69: Vector Rotate Left and Mask by Bits

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned short | vector unsigned short | vector signed short | $d$ = si_rothm($a$, $count$) | ROTHM d, a, count |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | vector signed int | $d$ = si_rotm($a$, $count$) | ROTM d, a, count |
| vector signed int | vector signed int | | | |
| vector unsigned short | vector unsigned short | 7-bit signed int (literal) | $d$ = si_rothmi($a$, $count$) | ROTHMI d, a, count |
| vector signed short | vector signed short | | | |
| vector unsigned short | vector unsigned short | int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | 7-bit signed int (literal) | $d$ = si_rotmi($a$, $count$) | ROTMI d, a, count |
| vector signed int | vector signed int | | | |
| vector unsigned int | vector unsigned int | int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed int | vector signed int | | | |

**spu_rlmaska: Vector Rotate Left and Mask Algebraic by Bits**

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

This function uses an element-wise rotate left and mask operation to perform an arithmetical shift right (ASR) of each element of vector $a$, where $count$ represents the negated value, or values, of the desired corresponding right-shift amounts. (The $count$ parameter can be either a vector or a scalar, as shown in Table 2-70.) For example, if scalar $count$ is –5, each element of $a$ is shifted right by 5 bits. The effect of this function is more precisely shown by the following code:

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

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

The results are returned in the corresponding elements of vector $d$.

Table 2-70: Vector Rotate Left and Mask Algebraic by Bits

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned short | vector unsigned short | vector signed short | $d$ = si_rotmah($a$, $count$) | ROTMAH d, a, count |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | vector signed int | $d$ = si_rotma($a$, $count$) | ROTMA d, a, count |
| vector signed int | vector signed int | | | |
| vector unsigned short | vector unsigned short | 7-bit signed int (literal) | $d$ = si_rotmahi($a$, $count$) | ROTMAHI d, a, count |
| vector signed short | vector signed short | | | |
| vector unsigned short | vector unsigned short | int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | 7-bit signed int (literal) | $d$ = si_rotmai($a$, $count$) | ROTMAI d, a, count |
| vector signed int | vector signed int | | | |
| vector unsigned int | vector unsigned int | int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed int | vector signed int | | | |

**spu_rlmaskqw: Quadword Rotate Left and Mask by Bits**

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

This function uses a rotate and mask quadword by bits operation to perform a quadword logical shift right (LSR) of up to 7 bits, where *count* represents the negated value of the desired right-shift amount. For example, if *count* is –5, vector *a* is shifted right by 5 bits. The effect of this function is more precisely shown by the following code:

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

The resulting quadword is returned in vector *d*.

Table 2-71: Quadword Rotate Left and Mask by Bits

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned char | vector unsigned char | int (literal) | $d$ = si_rotqmbii($a$, *count*) <br><br> (*count* = 7-bit immediate) | 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 (non-literal) | $d$ = si_rotqmbi($a$, *count*) | ROTQMBI d, a, count |
| vector signed char | vector signed char | | | |
| vector unsigned short | vector unsigned short | | | |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | | | |
| vector signed int | vector signed int | | | |
| vector unsigned long long | vector unsigned long long | | | |
| vector signed long long | vector signed long long | | | |
| vector float | vector float | | | |
| vector double | vector double | | | |

**spu_rlmaskqwbyte: Quadword Rotate Left and Mask by Bytes**

    d = spu_rlmaskqwbyte(a, count)

This function uses a rotate and mask quadword by bytes operation to perform a quadword logical shift right (LSR) by bytes, where *count* represents the negated value of the desired byte right-shift amount. For example, if *count* is –5, vector *a* is shifted right by 5 bytes. The effect of this function is more precisely shown by the following code:

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

The resulting quadword is returned in vector *d*.

Table 2-72: Quadword Rotate Left and Mask by Bytes

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned char | vector unsigned char | int (literal) | $d$ = si_rotqmbyi($a$, *count*)  (*count* = 7-bit immediate) | ROTQMBYI d, a, count |
| vector signed char | vector signed char | | | |
| vector unsigned short | vector unsigned short | | | |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | | | |
| vector signed int | vector signed int | | | |
| vector unsigned long long | vector unsigned long long | | | |
| vector signed long long | vector signed long long | | | |
| vector float | vector float | | | |
| vector double | vector double | | | |
| vector unsigned char | vector unsigned char | int (non-literal) | $d$ = si_rotqmby($a$, *count*) | ROTQMBY d, a, count |
| vector signed char | vector signed char | | | |
| vector unsigned short | vector unsigned short | | | |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | | | |
| vector signed int | vector signed int | | | |
| vector unsigned long long | vector unsigned long long | | | |
| vector signed long long | vector signed long long | | | |
| vector float | vector float | | | |
| vector double | vector double | | | |

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

    d = spu_rlmaskqwbytebc(a, count)

This function uses a rotate and mask quadword by bytes from bit shift count operation to perform a quadword logical shift right (LSR) by bytes, where bits 24-28 of *count* represent the negated value of the desired byte right-shift amount. For example, if the bit shift *count* is –10, vector *a* is shifted right by 2 bytes. The effect of this function is more precisely shown by the following code:

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

The resulting quadword is returned in vector *d*.

The following example code shows typical usage of this function; it computes a vector $d$ that is the value of vector $a$ logically shifted right by $n$ bits:

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

Table 2-73: Quadword Rotate Left and Mask by Bytes from Bit Shift Count

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned char | vector unsigned char | int | $d$ = si_rotqmbybi($a$, $count$) | ROTQMBYBI d, a, count |
| vector signed char | vector signed char | | | |
| vector unsigned short | vector unsigned short | | | |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | | | |
| vector signed int | vector signed int | | | |
| vector unsigned long long | vector unsigned long long | | | |
| vector signed long long | vector signed long long | | | |
| vector float | vector float | | | |
| vector double | vector double | | | |

**spu_rlqw: Quadword Rotate Left by Bits**

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

Vector $a$ is rotated to the left by the number of bits specified by the 3 least significant bits of $count$. Bits rotated out of the left end of the vector are rotated in on the right. The result is returned in vector $d$.

Table 2-74: Quadword Rotate Left by Bits

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned char | vector unsigned char | int (literal) | $d$ = si_rotqbii($a$, $count$)  ($count$ = 7-bit immediate) | 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 (non-literal) | $d$ = 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 | | | |

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector double | vector double | | | |

**spu_rlqwbyte: Quadword Rotate Left by Bytes**

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

Vector *a* is rotated to the left by the number of bytes specified by the 4 least significant bits of `count`. Bytes rotated out of the left end of the vector are rotated in on the right. The result is returned in vector *d*.

Table 2-75: Quadword Rotate Left by Bytes

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned char | vector unsigned char | int (literal) | $d$ = si_rotqbyi($a$, $count$) (`count` = 7-bit immediate) | 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 (non-literal) | $d$ = 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: Quadword Rotate Left by Bytes from Bit Shift Count**

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

Vector *a* is rotated to the left by the number of bytes specified by bits 24-28 of *count*. Bytes rotated out of the left end of the vector are rotated in at the right. The result is returned in vector *d*.

Table 2-76: Quadword Rotate Left by Bytes from Bit Shift Count

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned char | vector unsigned char | int | $d$ = si_rotqbybi($a$, $count$) | ROTQBYBI d, a, count |
| vector signed char | vector signed char | | | |
| vector unsigned short | vector unsigned short | | | |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | | | |
| vector signed int | vector signed int | | | |
| vector unsigned long long | vector unsigned long long | | | |
| vector signed long long | vector signed long long | | | |
| vector float | vector float | | | |
| vector double | vector double | | | |

**spu_sl: Vector Shift Left by Bits**

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

Each element of vector *a* is shifted left by the number of bits specified by the corresponding element in vector *count*. If *count* is a scalar, the scalar value is first replicated for each element, and then *a* is shifted.

Bits shifted out of the left end of the element are discarded, and zeros are shifted in at the right. A limited number of *count* bits are used depending on the size of the element. For halfword elements, the 5 least significant bits of *count* are used, and for word elements, the 6 least significant bits are used. The result is returned in the corresponding bit of vector *d*.

Table 2-77: Vector Shift Left by Bits

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned short | vector unsigned short | vector unsigned short | $d$ = si_shlh($a$, $count$) | SHLH d, a, count |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | vector unsigned int | $d$ = si_shl($a$, $count$) | SHL d, a, count |
| vector signed int | vector signed int | | | |
| vector unsigned short | vector unsigned short | 7-bit unsigned int (literal) | $d$ = si_shlhi($a$, $count$) | SHLHI d, a, count |
| vector signed short | vector signed short | | | |
| vector unsigned short | vector unsigned short | unsigned int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | 7-bit unsigned int (literal) | $d$ = si_shli($a$, $count$) | SHLI d, a, count |
| vector signed int | vector signed int | | | |
| vector unsigned int | vector unsigned int | unsigned int | See section "2.2.1. Mapping Intrinsics with Scalar Operands". | |
| vector signed int | vector signed int | | | |

**spu_slqw: Quadword Shift Left by Bits**

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

Vector *a* is shifted left by the number of bits specified by the 3 least significant bits of *count*. Bits shifted out of the left end of the vector are discarded, and zeros are shifted in at the right. The result is returned in vector *d*.

Table 2-78: Quadword Shift Left by Bits

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned char | vector unsigned char | unsigned int (literal) | $d$ = si_shlqbii($a$, $count$)  ($count$ = 7-bit immediate) | 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 (non-literal) | $d$ = si_shlqbi($a$, $count$) | SHLQBI d, a, count |
| vector signed char | vector signed char | | | |
| vector unsigned short | vector unsigned short | | | |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | | | |
| vector signed int | vector signed int | | | |
| vector unsigned long long | vector unsigned long long | | | |
| vector signed long long | vector signed long long | | | |
| vector float | vector float | | | |
| vector double | vector double | | | |

**spu_slqwbyte: Quadword Shift Left by Bytes**

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

Vector *a* is shifted left by the number of bytes specified by the 5 least significant bits of *count*. Bytes shifted out of the left end of the vector are discarded, and zeros are shifted in at the right. The result is returned in vector *d*.

Table 2-79: Quadword Shift Left by Bytes

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned char | vector unsigned char | unsigned int (literal) | $d$ = si_shlqbyi($a$, $count$)  ($count$ = 7-bit immediate) | 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 | | | |

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector double | vector double | | | |
| vector unsigned char | vector unsigned char | unsigned int (non-literal) | $d$ = si_shlqby($a$, $count$) | SHLQBY d, a, count |
| vector signed char | vector signed char | | | |
| vector unsigned short | vector unsigned short | | | |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | | | |
| vector signed int | vector signed int | | | |
| vector unsigned long long | vector unsigned long long | | | |
| vector signed long long | vector signed long long | | | |
| vector float | vector float | | | |
| vector double | vector double | | | |

**spu_slqwbytebc: Quadword Shift Left by Bytes from Bit Shift Count**

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

Vector $a$ is shifted left by the number of bytes specified by bits 24-28 of $count$. Bytes shifted out of the left end of the vector are discarded, and zeros are shifted in at the right. The result is returned in vector $d$.

Table 2-80: Quadword Shift Left by Bytes from Bit Shift Count

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|
| d | a | count | | |
| vector unsigned char | vector unsigned char | unsigned int | $d$ = si_shlqbybi($a$, $count$) | SHLQBYBI d, a, count |
| vector signed char | vector signed char | | | |
| vector unsigned short | vector unsigned short | | | |
| vector signed short | vector signed short | | | |
| vector unsigned int | vector unsigned int | | | |
| vector signed int | vector signed int | | | |
| vector unsigned long long | vector unsigned long long | | | |
| vector signed long long | vector signed long long | | | |
| vector float | vector float | | | |
| vector double | vector double | | | |

## 2.11. Control Intrinsics

**spu_idisable: Disable Interrupts**

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

Asynchronous interrupts are disabled.

This intrinsic is considered volatile with respect to all other instructions; thus, the BID instruction will not be reordered with any other instructions.

Table 2-81: Disable Interrupts

| Specific Intrinsics | Assembly Mapping |
|---|---|
| N/A | **position dependent:**<br>  ILA t, *next_inst*<br>  BID t<br>*next_inst:*<br><br>**position independent:**<br>  BRSL t, *next_inst*<br>*next_inst:*<br>  AI t, t, 8<br>  BID t |

**spu_ienable: Enable Interrupts**

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

Asynchronous interrupts are enabled.

This intrinsic is considered volatile with respect to all other instructions; thus, the BIE instruction will not be reordered with any other instructions.

Table 2-82: Enable Interrupts

| Specific Intrinsics | Assembly Mapping |
|---|---|
| N/A | **position dependent:**<br>  ILA t, *next_inst*<br>  BIE t<br>*next_inst:*<br><br>**position independent:**<br>  BRSL t, *next_inst*<br>*next_inst:*<br>  AI t, t, 8<br>  BIE t |

**spu_mffpscr: Move from Floating-Point Status and Control Register**

```
d = spu_mffpscr()
```

The floating-point status and control register (FPSCR) Special Purpose Register is read, and the contents are returned in $d$. Unused bits of the FPSCR are forced to zero.

This intrinsic is considered volatile with respect to the floating-point instructions and will not be reordered with respect to these instructions. The floating-point instructions include: `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`, and `fscrwr`.

Table 2-83: Move from Floating-Point Status and Control Register

| Return/Argument Types d | Specific Intrinsics | Assembly Mapping |
|---|---|---|
| vector unsigned int | $d$ = si_fscrrd() | FSCRRD d |

**spu_mfspr: Move from Special Purpose Register**

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

The Specal Purpose Register specified by enumeration constant `register` is read, and the contents are returned in $d$.

Table 2-84: Move from Special Purpose Register

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | register | | |
| unsigned int | enumeration | $d$ = si_to_uint(si_mfspr(`register`)) | MFSPR d, register |

**spu_mtfpscr: Move to Floating-Point Status and Control Register**

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

The argument $a$ is written to the floating-point status and control register (FPSCR).

This intrinsic is considered volatile with respect to the floating-point instructions, and it will not be reordered with respect to these instructions.

Table 2-85: Move to Floating-Point Status and Control Register

| Return/Argument Types a | Specific Intrinsics | Assembly Mapping |
|---|---|---|
| vector unsigned int | si_fscrwr($a$) | FSCRWR rt[1], a |

[1] The false target parameter `rt` is optimally chosen depending on register usage of neighboring instructions.

**spu_mtspr: Move to Special Purpose Register**

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

The argument $a$ is written to the Special Purpose Register specified by the enumeration constant `register`.

Table 2-86: Move to Special Purpose Register

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| register | a | | |
| enumeration | unsigned int | si_mtspr(`register`, si_from_uint(`a`)) | MTSPR register, a |

**spu_dsync: Synchronize Data**

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

All earlier store instructions are forced to complete before proceeding. This function ensures that all stores to local storage are visible to the MFC or PPU.

This intrinsic is considered volatile with respect to the store and MFC write instructions, and it will not be reordered with respect to these instructions. The store and MFC instructions include: `stqa`, `stqd`, `stqr`, `stqx`, and `wrch`.

Table 2-87: Synchronize Data

| Specific Intrinsics | Assembly Mapping |
|---|---|
| si_dsync() | DSYNC |

**spu_stop: Stop and Signal**

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

Execution of the SPU program is stopped. The address of the `stop` instruction is placed into the least significant bits of the SPU NPC register. The signal *type* is written to the SPU status register, and the PPU is interrupted.

This intrinsic is considered volatile with respect to all instructions, and it will not be reordered with any other instructions.

Table 2-88: Stop and Signal

| Specific Intrinsics | type | Assembly Mapping |
|---|---|---|
| si_stop(type) | unsigned int (14-bit literal) | STOP type |

**spu_sync: Synchronize**

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

The processor waits until all pending store instructions have been completed before fetching the next sequential instruction. The `spu_sync_c` form of the intrinsic also performs channel synchronization prior to the instruction synchronization. This operation must be used following a store instruction that modifies the instruction stream.

These synchronization intrinsics are considered volatile with respect to all instructions, and they will not be reordered with any other instructions.

Table 2-89: Synchronize

| Generic Intrinsic Form | Specific Intrinsics | Assembly Mapping |
|---|---|---|
| spu_sync | si_sync() | SYNC |
| spu_sync_c | si_syncc() | SYNCC |

## 2.12. Channel Control Intrinsics

The channel control intrinsics each take a *channel* number as an input. Channel numbers are literal unsigned integer values in the range from 0 to 127. Table 2-90 and Table 2-91 show the respective SPU and MFC channel numbers and their associated mnemonics. For additional details on the channels, see the *Cell Broadband Engine Architecture*.

The channel intrinsics must never be reordered with respect to other channel commands or volatile local-storage memory accesses.

The MFC channels are only valid for SPUs within a CBEA-compliant system. MFC and SPU channel enumerants are defined in `spu_intrinsics.h`

Table 2-90: SPU Channel Numbers

| Channel Number | Mnemonic | Description |
| --- | --- | --- |
| 0 | SPU_RdEventStat | Read event status with mask applied. |
| 1 | SPU_WrEventMask | Write event mask. |
| 2 | SPU_WrEventAck | Write End of event processing. |
| 3 | SPU_RdSigNotify1 | Signal notification 1. |
| 4 | SPU_RdSigNotify2 | Signal notification 2. |
| 7 | SPU_WrDec | Write decrementer count. |
| 8 | SPU_RdDec | Read decrementer count. |
| 11 | SPU_RdEventMask | Read event mask. |
| 13 | SPU_RdMachStat | Read SPU run status. |
| 14 | SPU_WrSRR0 | Write SPU machine state save/restore register 0 (SRR0). |
| 15 | SPU_RdSRR0 | Read SPU machine state save/restore register 0 (SRR0). |
| 28 | SPU_WrOutMbox | Write outbound mailbox contents. |
| 29 | SPU_RdInMbox | Read inbound mailbox contents. |
| 30 | SPU_WrOutIntrMbox | Write outbound interrupt mailbox contents (interrupting PPU). |

Table 2-91: MFC Channel Numbers

| Channel Number | Mnemonic | Description |
| --- | --- | --- |
| 9 | MFC_WrMSSyncReq | Write multisource synchronization request. |
| 12 | MFC_RdTagMask | Read tag mask. |
| 16 | MFC_LSA | Write local memory address command parameter. |
| 17 | MFC_EAH | Write high order DMA effective address command parameter. |
| 18 | MFC_EAL | Write low order DMA effective address command parameter. |
| 19 | MFC_Size | Write DMA transfer size command parameter. |
| 20 | MFC_TagID | Write tag identifier command parameter. |
| 21 | MFC_Cmd | Write and enqueue DMA command with associated class ID. |
| 22 | MFC_WrTagMask | Write tag mask. |
| 23 | MFC_WrTagUpdate | Write request for conditional/unconditional tag status update. |
| 24 | MFC_RdTagStat | Read tag status with mask applied. |
| 25 | MFC_RdListStallStat | Read DMA list stall-and-notify status. |
| 26 | MFC_WrListStallAck | Write DMA list stall-and-notify acknowledge. |
| 27 | MFC_RdAtomicStat | Read completion status of last completed immediate MFC atomic update command. |

**spu_readch: Read Word Channel**

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

The word channel that is specified by *channel* is read, and the contents are placed in *d*. If the channel does not exist, a value of zero is returned.

Table 2-92: Read Word Channel

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | channel | | |
| unsigned int | enumeration | *d* = si_to_uint(si_rdch(*channel*)) | RDCH d, channel |

**spu_readchqw: Read Quadword Channel**

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

The quadword channel that is specified by *channel* is read, and the contents are placed in vector *d*. If the channel does not exist, a value of zero is returned.

Table 2-93: Read Quadword Channel

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | channel | | |
| vector unsigned int | enumeration | *d* = si_rdch(*channel*) | RDCH d, channel |

**spu_readchcnt: Read Channel Count**

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

A Read Count operation is performed on thes channel that is specified by *channel*, and the count is placed in *d*. If the channel does not exist, a value of zero is returned in *d*.

Table 2-94: Read Channel Count

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| d | channel | | |
| unsigned int | enumeration | *d* = si_rchcnt(*channel*) | RCHCNT d, channel |

**spu_writech: Write Word Channel**

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

The contents of scalar *a* are written to the channel that is specified by the enumeration constant *channel*.

Table 2-95: Write Word Channel

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| channel | a | | |
| enumeration | int | si_wrch(*channel*, si_from_int(*a*)) | WRCH channel, a |
| | unsigned int | si_wrch(*channel*, si_from_uint(*a*)) | |

**spu_writechqw: Write Quadword Channel**

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

The contents of vector *a* are written to the channel that is specified by the enumeration constant *channel*.

Table 2-96: Write Quadword Channel

| Return/Argument Types | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|
| channel | a | | |
| enumeration | vector unsigned char | si_wrch(*channel*, *a*) | WRCH channel, a |
| | vector signed char | | |
| | vector unsigned short | | |
| | vector signed short | | |
| | vector unsigned int | | |
| | vector signed int | | |
| | vector unsigned long long | | |
| | vector signed long long | | |
| | vector float | | |
| | vector double | | |

## 2.13. Scalar Intrinsics

All of the previous intrinsic functions perform operations only on vector data types. This section describes special utility intrinsics that allow programmers to efficiently coerce scalars to vectors, or vectors to scalars. With the aid of these intrinsics, programmers can use intrinsic functions to perform operations between vectors and scalars without having to revert to assembly language. This is especially important when there is a need is to perform an operation that cannot be conveniently expressed in C, such as shuffling bytes.

**spu_extract: Extract Vector Element from Vector**

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

The element that is specified by *element* is extracted from vector *a* and returned in *d*. Depending on the size of the element, only a limited number of the least significant bits of the *element* index are used. For 1-, 2-, 4-, and 8-byte elements, only 4, 3, 2, and 1 of the least significant bits of the element index are used, respectively.

Table 2-97: Extract Vector Element from Vector

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping[1] |
|---|---|---|---|---|
| d | a | element | | |
| unsigned char | vector unsigned char | int (non-literal) | N/A | ROTQBY d, a, element |
| signed char | vector signed char | | N/A | ROTQBY d, a, element |
| unsigned short | vector unsigned short | | N/A | SHLI t, element, 1 |
| signed short | vector signed short | | N/A | SHLI t, element, 1 |
| unsigned int | vector unsigned int | | N/A | SHLI t, element, 2 |
| signed int | vector signed int | | N/A | SHLI t, element, 2 |
| unsigned long long | vector unsigned long long | | N/A | SHLI t, element, 3 |
| signed long long | vector signed long long | | N/A | SHLI t, element, 3 |
| float | vector float | | N/A | SHLI t, element, 2 |
| double | vector double | | N/A | SHLI t, element, 3 |

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping[1] |
|---|---|---|---|---|
| d | a | element | | |
| unsigned char | vector unsigned char | int (literal) | 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] If the specified element is a known value (literal) and specifies the preferred (scalar) element, no instructions are produced. For 1 byte elements, the scalar element is 3. For 2 byte elements, the scalar element is 1. For 4 and 8 byte elements, the scalar element is 0. Sign extension may still be performed if a subsequent operation requires the resulting scalar to be cast to a larger data type. This sign extension may be deferred until the subsequent operation.

**spu_insert: Insert Scalar into Specified Vector Element**

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

Scalar `a` is inserted into the element of vector `b` that is specified by the `element` parameter, and the modified vector is returned. All other elements of `b` are unmodified. Depending on the size of the element, only a limited number of the least significant bits of the `element` index are used. For 1-, 2-, 4-, and 8-byte elements, only 4, 3, 2, and 1 of the least significant bits of the `element` index are used, respectively.

Table 2-98: Insert Scalar into Specified Vector Element

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | element | | |
| vector unsigned char | unsigned char | vector unsigned char | int (non-literal) | 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 | |

| Return/Argument Types | | | | Specific Intrinsics | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | element | | |
| vector unsigned char | unsigned char | vector unsigned char | int (literal) | 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] If the specified element is a known value (literal), a shuffle pattern can be loaded from the constant area. The contents of the pattern depend on the size of the element and the element being replaced.

**spu_promote: Promote Scalar to Vector**

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

Scalar *a* is promoted to a vector containing *a* in the element that is specified by the *element* parameter, and the vector is returned in *d*. All other elements of the vector are undefined. Depending on the size of the element/scalar, only a limited number of the least significant bits of the *element* index are used. For 1-, 2-, 4-, and 8-byte elements, only 4, 3, 2, and 1 of the least significant bits of the *element* index are used, respectively.

Table 2-99: Promote Scalar to Vector

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping[1] |
|---|---|---|---|---|
| d | a | element | | |
| vector unsigned char | unsigned char | int (non-literal) | 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 | |

| Return/Argument Types | | | Specific Intrinsics | Assembly Mapping[1] |
|---|---|---|---|---|
| d | a | element | | |
| vector unsigned char | unsigned char | int (literal) | 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] If the specified element is of known value (literal) and specifies the preferred (scalar) element, no instructions are produced. For 1 byte elements, the scalar element is 3. For 2 byte elements, the scalar element is 1. For 4 and 8 byte elements, the scalar element is 0.

# 3. Composite Intrinsics

This chapter describes several composite intrinsics that have practical use for a wide variety of SPU programs. Composite intrinsics are those intrinsics that can be constructed from a series of low-level intrinsics. In this context, "low-level" means generic or specific. Because of the complexity of these operations, frequency of use, and scheduling constraints, the particular services are provided as intrinsics.

Composite intrinsics are DMA intrinsics. The DMA intrinsics rely heavily on the channel control intrinsics.

### spu_mfcdma32: Initiate DMA to/from 32-Bit Effective Address

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

A DMA transfer of *size* bytes is initiated from local to system memory or from system memory to local storage. The effective address that is specified by *ea* is a 32-bit virtual memory address. The local-storage address is specified by the *ls* parameter. The DMA request is issued using the specified *tagid*. The type and direction of DMA, bandwidth reservation, and class ID are encoded in the *cmd* parameter. For additional details about the commands and restrictions on the size of supported DMA operations, see the *Cell Broadband Engine Architecture*.

Table 3-100: Initiate DMA to/from 32-Bit Effective Address

| Return/Argument Types | | | | | Assembly Mapping |
|---|---|---|---|---|---|
| ls | ea | size | tagid | cmd | |
| volatile void * | unsigned int | unsigned int | unsigned int | unsigned int | spu_writech(MFC_LSA, *ls*)<br>spu_writech(MFC_EAL, *ea*)<br>spu_writech(MFC_Size, *size*)<br>spu_writech(MFC_TagID, *tagid*)<br>spu_writech(MFC_Cmd, *cmd*) |

### spu_mfcdma64: Initiate DMA to/from 64-Bit Effective Address

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

A DMA transfer of *size* bytes is initiated from local to system memory or from system memory to local storage. The effective address that is specified by the concatenation of *eahi* and *ealow* is a 64-bit virtual memory address. The local-storage address is specified by the *ls* parameter. The DMA request is issued using the specified *tagid*. The type and direction of DMA, bandwidth reservation, and class ID are encoded in the *cmd* parameter. For additional details about the commands and restrictions on the size of supported DMA operations, see the *Cell Broadband Engine Architecture*.

Table 3-101: Initiate DMA to/from 64-Bit Effective Address

| Return/Argument Types | | | | | | Assembly Mapping |
|---|---|---|---|---|---|---|
| ls | eahi | ealow | size | tagid | cmd | |
| volatile void * | unsigned int | unsigned int | unsigned int | unsigned int | unsigned int | spu_writech(MFC_LSA, *ls*)<br>spu_writech(MFC_EAH, *eahi*)<br>spu_writech(MFC_EAL, *ealow*)<br>spu_writech(MFC_Size, *size*)<br>spu_writech(MFC_TagID, *tagid*)<br>spu_writech(MFC_CMD, *cmd*) |

**spu_mfcstat: Read MFC Tag Status**

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

The current MFC tag status is read and logically ANDed with the current tag mask, and the result is returned in *d*. The type of read to be performed is specified by the *type* parameter. If the *type* is 0, the function reads and immediately returns the current MFC tag status. If the *type* is 1, the function reads and blocks for any outstanding MFC tags to complete, and if the *type* is 2, the function reads and blocks for all outstanding MFC tags to complete.

Table 3-102: Read MFC Tag Status

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | type | |
| unsigned int | unsigned int | spu_writech(MFC_WrTagUpdate, *type*)<br>*d* = spu_readch(MFC_RdTagStat) |

# 4. Programming Support for MFC Input and Output

Several MFC utility functions are described in this chapter. These functions may be provided as a programming convenience; none of them are required. The functions that are described can be implemented either as macro definitions or as built-in functions within the compiler. To access these functions, programmers must include the header file `spu_mfcio.h`.

For each function listed in the sections below, the function usage is shown, followed by a brief description and the function implementation.

## 4.1. Structures

A principal data structure is the MFC List DMA. The elements in this list are described below.

**mfc_list_element: DMA List Element for MFC List DMA**

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

The `mfc_list_element` is an element in the array MFC List DMA. The structure is comprised of several bit-fields: *notify* is the stall-and-notify bit, *reserved* is set to zero. *size* is the list element transfer size, and *eal* is the low word of the 64-bit effective address.

## 4.2. Effective Address Utilities

A frequent requirement for MFC programming is to manipulate effective addresses. This section describes several functions for performing the most common operations.

**mfc_ea2h: Extract Higher 32 Bits from Effective Address**

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

The higher 32 bits are extracted from the 64-bit effective address *ea*.

**Implementation**

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

**mfc_ea2l: Extract Lower 32 Bits from Effective Address**

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

The lower 32 bits are extracted from the 64-bit effective address *ea*.

**Implementation**

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

**mfc_hl2ea: Concatenate Higher 32 Bits and Lower 32 Bits**

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

The higher 32 bits of a 64-bit address *high* and the lower 32 bits *low* are concatenated.

**Implementation**

```
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)
```

The argument *value* is rounded to the next higher multiple of 128.

**Implementation**

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

**Example**

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

## 4.3. MFC Tag Manager

This section describes functions that facilitate interoperability through a cooperative use of tag identifiers. Applications, libraries, and tools that initiate DMAs should use these functions to reserve a tag ID or a set of IDs.

An implementation of the tag manager is not required to make all 32 architected tag IDs available for user allocation. Some tags may be pre-allocated and used by the operating environment.

These functions are provided in a system library; therefore, they do not require explicit library linking by the programmer.

MFC tag manager mnemonics are listed in Table 4-103. These mnemonics are defined in spu_mfcio.h.

Table 4-103: MFC Tag Manager Mnemonics

| Mnemonic | Value | Description |
|---|---|---|
| MFC_TAG_VALID | 0x00000000 | The specified tag or tag group release was successful. |
| MFC_TAG_INVALID | 0xFFFFFFFF | The tag or tag group reservation or tag release failed. |

**mfc_tag_reserve: Reserve a Tag for Exclusive Use**

```
(uint32_t) mfc_tag_reserve(void)
```

Reserve a tag for exclusive use. This routine returns an available tag ID in the range 0 to 31 and marks the tag as reserved. If no tags are available, MFC_TAG_INVALID is returned, indicating that all tags have already been reserved.

**mfc_tag_release: Release a Tag from Exclusive Use**

```
(uint32_t) mfc_tag_release(uint32_t tag)
```

Release the specified tag from exclusive use. After it is released, it is available for future reservation. Upon successful release, `MFC_TAG_VALID` is returned. If the specified tag is not in the range 0 to 31 or if it was not reserved, no action is taken and `MFC_TAG_INVALID` is returned.

**mfc_multi_tag_reserve: Reserve a Group of Tags for Exclusive Use**

```
(uint32_t) mfc_multi_tag_reserve(uint32_t number_of_tags)
```

Reserve a sequential group of tags for exclusive use. The number of tags to be reserved is specified by the *number_of_tags* parameter. This routine returns the first tag ID in a sequential list of available tags and marks them as reserved. The reserved group of tags is in the range of IDs starting from the returned tag ID through the returned tag ID + *number_of_tags* - 1.

If the number of tags requested exceeds the number of available sequential tags, `MFC_TAG_INVALID` is returned, indicating that the request could not be performed.

**mfc_multi_tag_release: Release a Group of Tags from Exclusive Use**

```
(uint32_t) mfc_multi_tag_release(uint32_t first_tag, uint32_t number_of_tags)
```

Release a sequential group of tags from exclusive use. The sequential group of tags is the range of tag IDs starting from *first_tag* through *first_tag* + *number_of_tags* - 1. Upon successful release, the tags become available for future reservation, and `MFC_TAG_VALID` is returned. If the specified tags were not previously reserved, no action is taken, and `MFC_TAG_INVALID` is returned.

## 4.4. MFC DMA Commands

This section describes functions that implement the various MFC DMA commands. See the *Cell Broadband Engine Architecture f*or a description of the DMA commands, including restrictions on the size of the supported operations.

MFC DMA command mnemonics are listed in Table 4-104. MFC command enumerants are defined in `spu_mfcio.h`.

Table 4-104: MFC DMA Command Mnemonics

| Mnemonic | Opcode | Command |
|---|---|---|
| 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 |

**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)
```

Data is moved from local storage to system memory. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: *ls* is the local-storage address, *ea* is the effective address in system memory, *size* is the DMA transfer size, *tag* is the DMA tag, *tid* is the transfer class identifier, and *rid* is the replacement class identifier.

**Implementation**

```
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)
```

Data is moved from local storage to system memory. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: `ls` is the local-storage address, `ea` is the effective address in system memory, `size` is the DMA transfer size, `tag` is the DMA tag, `tid` is the transfer class identifier, and `rid` is the replacement class identifier. This command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue.

**Implementation**

```
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)
```

Data is moved from local storage to system memory. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: `ls` is the local-storage address, `ea` is the effective address in system memory, `size` is the DMA transfer size, `tag` is the DMA tag, `tid` is the transfer class identifier, and `rid` is the replacement class identifier. This command is locally ordered with respect to all previously issued commands within the same tag group and command queue.

**Implementation**

```
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)
```

Data is moved from system memory to local storage. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: `ls` is the local-storage address, `ea` is the effective address in system memory, `size` is the DMA transfer size, `tag` is the DMA tag, `tid` is the transfer class identifier, and `rid` is the replacement class identifier.

**Implementation**

```
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)
```

Data is moved from system memory to local storage. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: `ls` is the local-storage address, `ea` is the effective address in system memory, `size` is the DMA transfer size, `tag` is the DMA tag, `tid` is the transfer class identifier, and `rid` is the replacement class identifier. This command is locally ordered with respect to all previously issued commands within the same tag group and command queue.

**Implementation**
```
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)
```

Data is moved from system memory to local storage. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: `ls` is the local-storage address, `ea` is the effective address in system memory, `size` is the DMA transfer size, `tag` is the DMA tag, `tid` is the transfer class identifier, and `rid` is the replacement class identifier. This command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue.

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

## 4.5. MFC List DMA Commands

This section describes utility functions that can be used to manage the MFC List DMA. See the *Cell Broadband Engine Architecture f*or a description of the DMA commands, including restrictions on the size of the supported operations.

MFC List DMA command mnemonics are listed in Table 4-105. MFC command enumerants are defined in `spu_mfcio.h`.

Table 4-105: MFC List DMA Command Mnemonics

| Mnemonic | Opcode | Command |
|----------|--------|---------|
| 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 |

**mfc_putl: Move Data from Local Storage to Effective Address Using MFC List**
```
(void) mfc_putl(volatile void *ls, uint64_t ea, volatile mfc_list_element_t *list,
    uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

Data is moved from local storage to system memory using the MFC list. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: `ls` is the local-storage address, `ea` is the effective address in system memory, `list` is the DMA list address, `list_size` is the DMA list size, `tag` is the DMA tag, `tid` is the transfer class identifier, and `rid` is the replacement class identifier.

**Implementation**
```
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, volatile mfc_list_element_t *list,
    uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

Data is moved from local storage to system memory using the MFC list. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: *ls* is the local-storage address, *ea* is the effective address in system memory, *list* is the DMA list address, *list_size* is the DMA list size, *tag* is the DMA tag, *tid* is the transfer class identifier, and *rid* is the replacement class identifier. This command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue.

**Implementation**

```
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, volatile mfc_list_element_t *list,
    uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

Data is moved from local storage to system memory using the MFC list. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: *ls* is the local-storage address, *ea* is the effective address in system memory, *list* is the DMA list address, *list_size* is the DMA list size, *tag* is the DMA tag, *tid* is the transfer class identifier, and *rid* is the replacement class identifier. This command is locally ordered with respect to all previously issued commands within the same tag group and command queue.

**Implementation**

```
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, volatile mfc_list_element_t *list,
    uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

Data is moved from system memory to local storage using the MFC list. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: *ls* is the local-storage address, *ea* is the effective address in system memory, *list* is the DMA list address, *list_size* is the DMA list size, *tag* is the DMA tag, *tid* is the transfer class identifier, and *rid* is the replacement class identifier.

**Implementation**

```
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, volatile mfc_list_element_t *list,
    uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

Data is moved from system memory to local storage using the MFC list. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: *ls* is the local-storage address, *ea* is the effective address in system memory, *list* is the DMA list address, *list_size* is the DMA list size, *tag* is the DMA tag, *tid* is the transfer class identifier, and *rid* is the replacement class identifier. This command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue.

**Implementation**

```
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, volatile mfc_list_element_t *list,
      uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)
```

Data is moved from system memory to local storage using the MFC list. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: *ls* is the local-storage address, *ea* is the effective address in system memory, *list* is the DMA list address, *list_size* is the DMA list size, *tag* is the DMA tag, *tid* is the transfer class identifier, and *rid* is the replacement class identifier. This command is locally ordered with respect to all previously issued commands within the same tag group and command queue.

**Implementation**

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

## 4.6. MFC Atomic Update Commands

This section describes utility functions that can be used to manage the MFC Atomic DMA. See the *Cell Broadband Engine Architecture f*or a description of the DMA commands, including restrictions on the size of the supported operations.

MFC Atomic DMA command mnemonics are listed in Table 4-106. MFC command enumerants are defined in `spu_mfcio.h`.

Table 4-106: MFC Atomic Update Command Mnemonics

| Mnemonic | Opcode | Command |
|---|---|---|
| MFC_GETLLAR_CMD | 0x00D0 | getllar |
| MFC_PUTLLC_CMD | 0x00B4 | putllc |
| MFC_PUTLLUC_CMD | 0x00B0 | putlluc |
| MFC_PUTQLLUC_CMD | 0x00B8 | putqlluc |

**mfc_getllar: Get Lock Line and Create Reservation**

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

The lock line is obtained and a reservation is created. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: *ls* is the 128-byte-aligned local-storage address, *ea* is the effective address in system memory, *tid* is the transfer class identifier, and *rid* is the replacement class identifier.

The `mfc_getllar` command does not have a tag ID. The command is immediately executed by the MFC. The transfer size is fixed at 128 bytes. An `mfc_read_atomic_status()` must follow this function to verify completion of the command.

**Implementation**

```
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)
```

The lock line is put if a reservation for effective address exists. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: *ls* is the 128-byte-aligned local-storage address, *ea* is the effective address in system memory, *tid* is the transfer class identifier, and *rid* is the replacement class identifier.

The `mfc_putllc` command does not have a tag ID and is immediately executed by MFC. Transfer size is fixed at 128 bytes. An `mfc_read_atomic_status()` must follow this command to verify completion of the command.

**Implementation**

```
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)
```

The lock line is put regardless of the existence of a previously made reservation. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: `ls` is the 128-byte-aligned local-storage address, `ea` is the effective address in system memory, `tid` is the transfer class identifier, and `rid` is the replacement class identifier.

This command does not have a tag ID and is immediately executed by MFC. The transfer size is fixed at 128 bytes. The `mfc_read_atomic_status()` must follow this function to verify completion of the command.

**Implementation**

```
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)
```

The lock line is put in the queue regardless of the existence of a previously made reservation. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: `ls` is the 128-byte-aligned local-storage address, `ea` is the effective address in system memory, `tid` is the transfer class identifier, and `rid` is the replacement class identifier.

Transfer size is fixed at 128 bytes. This command is functionally equivalent to the `mfc_putlluc` command. The difference between the two commands is the order in which the commands are executed and the way that completion is determined. `mfc_putlluc` is performed immediately; in contrast, `mfc_putqlluc` is placed into the MFC command queue, along with other MFC commands. Because this command is queued, it is executed independently of any pending immediate `mfc_getllar`, `mfc_putllc`, or `mfc_putlluc` commands. To determine if this command has been performed, a program must wait for a tag-group completion.

**Implementation**

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

## 4.7. MFC Synchronization Commands

This section describes functions that implement the MFC synchronization commands, including signal notification and storage ordering. See the *Cell Broadband Engine Architecture f*or a description of the DMA commands, including restrictions on the size of the supported operations.

MFC synchronization command mnemonics are listed in Table 4-107. MFC command enumerants are defined in `spu_mfcio.h`.

Table 4-107: MFC Synchronization Command Mnemonics

| Mnemonic | Opcode | Command |
|---|---|---|
| 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 |

| Mnemonic | Opcode | Command |
|----------|--------|---------|
| MFC_SYNC_CMD | 0x00CC | mfcsync |

**mfc_sndsig: Send Signal**

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

An `mfc_sndsig` command is enqueued into the DMA queue, or is stalled when the DMA queue is full. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: *ls* is the local-storage address, *ea* is the effective address in system memory, *tag* is the DMA tag, *tid* is the transfer class identifier, and *rid* is the replacement class identifier. Transfer size is fixed at 4 bytes.

**Implementation**

```
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)
```

An `mfc_sndsigb` command is enqueued into the DMA queue, or is stalled when the DMA queue is full. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: *ls* is the local-storage address, *ea* is the effective address in system memory, *tag* is the DMA tag, *tid* is the transfer class identifier, and *rid* is the replacement class identifier. Transfer size is fixed at 4 bytes. This command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue.

**Implementation**

```
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)
```

An `mfc_sndsigf` command is enqueued into the DMA queue, or is stalled when the DMA queue is full. The arguments to this function correspond to the arguments of the `spu_mfcdma64` command: *ls* is the local-storage address, *ea* is the effective address in system memory, *tag* is the DMA tag, *tid* is the transfer class identifier, and *rid* is the replacement class identifier. Transfer size is fixed at 4 bytes. This command is locally ordered with respect to all previously issued commands within the same tag group and command queue.

**Implementation**

```
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)
```

An `mfc_barrier` command is enqueued into the DMA queue, or the command is stalled when the DMA queue is full. *tag* is the DMA tag. An `mfc_barrier` command guarantees that MFC commands preceding the barrier will be executed before the execution of MFC commands following it, regardless of the *tag* of preceding or subsequent MFC commands.

**Implementation**

```
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)
```

An `mfc_eieio` command is enqueued into the DMA queue, or the command is stalled when the DMA queue is full. *tag* is the DMA tag, *tid* is the transfer class identifier, and *rid* is the replacement class identifier. Do not use this command to maintain the order of commands immediately inside a single SPE. The `mfc_eieio` command is designed to use inter-processor/device synchronization. This command creates a large load on the memory system.

**Implementation**

```
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)
```

An `mfc_sync` command is enqueued into the DMA queue, where *tag* is the DMA tag, or the command is stalled when the DMA queue is full. This function must not be used to maintain the order of commands immediately inside a single SPE. The `mfc_sync` command is designed to use inter-processor/device synchronization. This command creates a large load on the memory system.

**Implementation**

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

## 4.8. MFC DMA Status

This section describes functions that can be used to check the completion of MFC commands or the status of entries in the MFC DMA queue.

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

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

The number of available entries in the MFC DMA queue is checked. This information can be used to avoid stalling the execution of an SPU program if a DMA command is issued to a full queue. A full queue is 16 entries.

**Implementation**

```
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)
```

A tag mask is set to select the MFC tag groups to be included in the query operation, where *mask* is the DMA tag-group query mask. Each bit of *mask* indicates each tag group; tag 0 is mapped to LSB.

**Implementation**

```
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)
```

The tag mask is read to identify MFC tag groups to be included in the query operation. Each bit of the mask indicates each tag group; tag 0 is mapped to LSB. The result represents a DMA tag-group query mask.

**Implementation**

```
spu_readch(MFC_RdTagMask)
```

**mfc_write_tag_update: Request That Tag Status be Updated**

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

A request is sent to the MFC to update tag status, where *ts* specifies a tag-status update condition shown in Table 4-108. Condition enumerants are defined in `spu_mfcio.h`.

This function must precede a tag-status read with the `mfc_read_tag_status()` function. A tag-status update request should be performed after setting the tag-group mask with the `mfc_write_tag_mask()` function.

Table 4-108: MFC Write Tag Update Conditions

| Number | Mnemonic | Description |
|--------|----------|-------------|
| 0 | MFC_TAG_UPDATE_IMMEDIATE | Update immediately, unconditionally. |
| 1 | MFC_TAG_UPDATE_ANY | Update tag status if or when any enabled tag group has "no outstanding operation" status. |
| 2 | MFC_TAG_UPDATE_ALL | Update tag status if or when all enabled tag groups have "no outstanding operation" status. |

**Implementation**

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

**mfc_write_tag_update_immediate: Request That Tag Status be Immediately Updated**

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

A request is sent to immediately update tag status.

**Implementation**

```
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)
```

A request is sent to update tag status when any enabled MFC tag-group completion has a "no operation outstanding" status.

**Implementation**

```
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)
```

A request is sent to update tag status when all enabled MFC tag groups have a "no operation outstanding" status.

**Implementation**

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

**mfc_stat_tag_update: Check Availability of Tag Status Update Request Channel**

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

The availability of the Tag Status Update Request channel is checked. The result has one of the following values:

- 0: The Tag Status Update Request channel is not yet available.
- 1: The Tag Status Update Request channel is available.

**Implementation**

```
spu_readchcnt(MFC_WrTagUpdate)
```

**mfc_read_tag_status: Wait for an Updated Tag Status**

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

The status of the tag groups is requested. Unless the tag update is set to `MFC_TAG_UPDATE_IMMEDIATE`, this call could be blocked. Each bit of a returned value indicates the status of each tag group; tag 0 is mapped to LSB. If set, the tag group has no outstanding operation (that is, commands completed) and is not masked by the query.

Only the status of the enabled tag groups at the time of the tag-group status update are valid. The bit positions that correspond to the tag groups that are disabled at the time of the tag-group status update are set to 0.

**Implementation**

```
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)
```

A request is sent to immediately update tag status. The processor waits for the status to be updated.

**Implementation**

```
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)
```

A request is sent to update tag status when any enabled MFC tag-group completion has a "no operation outstanding" status. The processor waits for the status to be updated.

**Implementation**

```
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)
```

A request is sent to update tag status when all enabled MFC tag groups have a "no operation outstanding" status. The processor waits for the status to be updated.

**Implementation**

```
spu_mfcstat(MFC_TAG_UPDATE_ALL)
```

**mfc_stat_tag_status: Check Availability of MFC_RdTagStat Channel**

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

The availability of `MFC_RdTagStat` channel is checked, and one of the following values is returned:

- 0: The status is not yet available.
- 1: The status is available.

This function is used to avoid a channel stall caused by reading the `MFC_RdTagStat` channel when a status is not available.

**Implementation**

```
spu_readchcnt(MFC_RdTagStat)
```

**mfc_read_list_stall_status: Read List DMA Stall-and-Notify Status**

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

The List DMA stall-and-notify status is read and returned, or the program is stalled until the status is available.

**Implementation**

```
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)
```

The availability of the List DMA stall-and-notify status is checked, and one of the following values is returned:

- 0: The status is not yet available.
- 1: The status is available.

**Implementation**

```
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)
```

An acknowledgement is sent with respect to a prior stall-and-notify event. (See `mfc_read_list_status` and `mfc_stat_list_stall_status`.) The argument `tag` is the DMA tag.

**Implementation**

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

**mfc_read_atomic_status: Read Atomic Command Status**

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

The atomic command status is read, or the program is stalled until the status is available. As shown in Table 4-109, one of the following atomic command status results (binary value of bits 29 through 31) is returned. Status enumerants are defined in `spu_mfcio.h`.

Table 4-109: Read Atomic Command Status or Stall Until Status Is Available

| Status | Mnemonic | Description |
|---|---|---|
| 1 | MFC_PUTLLC_STATUS | The `mfc_putllc` command failed (reservation lost). |
| 2 | MFC_PUTLLUC_STATUS | The `mfc_putlluc` command was completed successfully. |
| 4 | MFC_GETLLAR_STATUS | The `mfc_getllar` command was completed successfully. |

**Implementation**

```
spu_readch(MFC_RdAtomicStat)
```

**mfc_stat_atomic_status: Check Availability of Atomic Command Status**

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

The availability of the atomic command status is checked, and one of the following values is returned:

- 0: An atomic DMA command has not yet completed.
- 1: An atomic DMA command has completed and the status is available.

**Implementation**

```
spu_readchcnt(MFC_RdAtomicStat)
```

## 4.9. MFC Multisource Synchronization Request

The *Cell Broadband Engine Architecture* describes the MFC Multisource Synchronization Facility. In that document, a cumulative ordering is broadly defined as an ordering of storage accesses performed by multiple processors or units with respect to another processor or unit. In this section, several functions are described that can be used to achieve a cumulative ordering across local and main storage address domains.

**mfc_write_multi_src_sync_request: Request Multisource Synchronization**

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

A request is sent to start tracking outstanding transfers sent to the associated MFC. When the requested synchronization is complete, the channel count of the MFC Multisource Synchronization Request channel is reset to one.

**Implementation**

```
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)
```

The channel count of the MFC Multisource Synchronization Request channel is read, and one of the following values is returned:

- 0: Outstanding transfers are being tracked.
- 1: The synchronization requested by `mfc_write_multi_src_sync_request` is complete.

**Implementation**

```
spu_readchcnt(MFC_WrMSSyncReq)
```

## 4.10. SPU Signal Notification

In this section, functions are described that can be used to read signals from other processors and other devices in the system.

**spu_read_signal1: Atomically Read and Clear Signal Notification 1 Channel**

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

The Signal Notification 1 channel is read, and any bits that are set are atomically reset. A signal is returned. If no signals are pending, this function will stall the SPU until a signal is issued.

**Implementation**

```
spu_readch(SPU_RdSigNotify1)
```

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

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

A check is made to determine whether any pending signals exist on the Signal Notification 1 channel. One of the following values is returned:

- 0: No signals are pending.
- 1: Signals are pending.

**Implementation**

```
spu_readchcnt(SPU_RdSigNotify1)
```

**spu_read_signal2: Atomically Read and Clear Signal Notification 2 Channel**

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

The Signal Notification 2 channel is read, and any bits that are set are atomically reset. A signal is returned. If no signals are pending, a call of this function stalls the SPU until a signal is issued.

**Implementation**

```
spu_readch(SPU_RdSigNotify2)
```

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

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

A check is made to determine whether pending signals exist on the Signal Notification 2 channel. One of the following values is returned:

- 0: No signals are pending.
- 1: Signals are pending.

**Implementation**

```
spu_readchcnt(SPU_RdSigNotify2)
```

## 4.11. SPU Mailboxes

This section describes functions that can be used to manage SPU Mailboxes.

**spu_read_in_mbox: Read Next Data Entry in SPU Inbound Mailbox**

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

The next data entry in the SPU Inbound Mailbox queue is read. The command stalls when the queue is empty. The application-specific mailbox data is returned. Each application can uniquely define the mailbox data.

**Implementation**

```
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)
```

The number of data entries in the SPU Inbound Mailbox is returned. If the returned value is nonzero, the mailbox contains data entries that have not been read by the SPU.

**Implementation**

```
spu_readchcnt(SPU_RdInMbox)
```

**spu_write_out_mbox: Send Data to SPU Outbound Mailbox**

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

Data is sent to the SPU Outbound Mailbox, where *data* is application-specific mailbox data, or the command stalls when the SPU Outbound Mailbox is full.

**Implementation**

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

**spu_stat_out_mbox: Get Available Capacity of SPU Outbound Mailbox**

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

The available capacity of the SPU Outbound Mailbox is returned. A value of zero indicates that the mailbox is full.

**Implementation**

```
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)
```

Data is sent to the SPU Outbound Interrupt Mailbox, where *data* is application-specific mailbox data. The command stalls when the SPU Outbound Interrupt Mailbox is full.

**Implementation**

```
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)
```

The available capacity of the SPU Outbound Interrupt Mailbox is returned. A value of zero indicates that the mailbox is full.

**Implementation**

```
spu_readchcnt(SPU_WrOutIntrMbox)
```

## 4.12. SPU Decrementer

This section describes functions that use the SPU 32-bit decrementer.

**spu_read_decrementer: Read Current Value of Decrementer**

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

The current value of the decrementer is read and returned.

**Implementation**

```
spu_readch(SPU_RdDec)
```

**spu_write_decrementer: Load a Value to Decrementer**

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

A count is loaded to the decrementer.

**Implementation**

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

## 4.13. SPU Event

This section describes several functions that can be used to monitor SPU events. See the *Cell Broadband Engine Architecture f*or a description of the SPU Event Facility.

The bit-fields of the Event Status, the Event Mask, and the Event Ack are shown in Table 4-110. Bit-field names are defined in spu_mfcio.h.

Table 4-110: MFC Event Bit-Fields

| Bits | Field Name | Description |
|---|---|---|
| 0x1000 | MFC_MULTI_SRC_SYNC_EVENT | Multisource synchronization event |
| 0x0800 | MFC_PRIV_ATTN_EVENT | SPU privileged attention event |
| 0x0400 | MFC_LLR_LOST_EVENT | Lock-line reservation lost event |
| 0x0200 | MFC_SIGNAL_NOTIFY_1_EVENT | SPU Signal Notification 1 available event |
| 0x0100 | MFC_SIGNAL_NOTIFY_2_EVENT | SPU Signal Notification 2 available event |
| 0x0080 | MFC_OUT_MBOX_AVAILABLE_EVENT | SPU Outbound Mailbox available event |
| 0x0040 | MFC_OUT_INTR_MBOX_AVAILABLE_EVENT | SPU Outbound Interrupt Mailbox available event |
| 0x0020 | MFC_DECREMENTER_EVENT | SPU decrementer event |
| 0x0010 | MFC_IN_MBOX_AVAILABLE_EVENT | SPU Inbound Mailbox available event |
| 0x0008 | MFC_COMMAND_QUEUE_AVAILABLE_EVENT | MFC SPU command queue available event |
| 0x0002 | MFC_LIST_STALL_NOTIFY_EVENT | MFC DMA List command stall-and-notify event |
| 0x0001 | MFC_TAG_STATUS_UPDATE_EVENT | MFC tag-group status update event |

**spu_read_event_status: Read Event Status or Stall Until Status is Available**

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

The event status is read and returned. The command stalls until the status is available. Events that have been reported but not acknowledged will continue to be reported until acknowledged.

The return value is the value of the SPU Read Event Status channel.

**Implementation**

```
spu_readch(SPU_RdEventStat)
```

**spu_stat_event_status: Check Availability of Event Status**

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

The event status is checked, and one of the following values is returned:

- 0: No enabled events occurred.
- 1: Enabled events are pending.

**Implementation**

```
spu_readchcnt(SPU_RdEventStat)
```

**spu_write_event_mask: Select Events to be Monitored by Event Status**

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

Events are selected to be monitored by event status. The argument, *mask*, is the event mask.

**Implementation**

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

**spu_write_event_ack: Acknowledge Events**

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

This function acknowledges that the corresponding events are being serviced by the software. The status of acknowledged events is reset, and the events are resampled. The argument, *ack*, represents events acknowledgment.

**Implementation**

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

**spu_read_event_mask: Read Event Status Mask**

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

The current Event Status Mask is read, and the mask is returned.

**Implementation**

```
spu_readch(SPU_RdEventMask)
```

## 4.14. SPU State Management

This section describes functions that relate to interrupts. See the *Cell Broadband Engine Architecture* for a description of the SPU Machine Status channel and the SPU interrupt-related channels.

**spu_read_machine_status: Read Current SPU Machine Status**

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

The current SPU machine status is read, and the status is returned.

**Implementation**

```
spu_readch(SPU_RdMachStat)
```

**spu_write_srr0: Write to SPU SRR0**

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

The value of *srr0* is written to the SPU state save/restore register 0 (SRR0).

**Implementation**

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

**spu_read_srr0: Read SPU SRR0**

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

The SPU state save/restore register 0 (SRR0) is read, and the state is returned.

**Implementation**

```
spu_readch(SPU_RdSRR0)
```

# 5. SPU and PPU Vector Multimedia Extension Intrinsics

Function mapping techniques can be used to increase the portability of source code written with SPU intrinsics or PPU Vector Multimedia Extension (VMX) intrinsics. By including the appropriate portability headers, SPU intrinsics can be used on the PPU, or VMX intrinsics can be used on the SPU. This chapter describes a minimal mapping between the two sets of intrinsics.

For many intrinsic functions, an efficient one-to-one mapping between architectures will exist. For some functions, there could be a less efficient one-to-many instruction mapping; and for other functions, no straightforward mapping will exist because a mapping is either impractical or impossible to implement. In this document, only one-to-one mappings are identified for the SPU and PPU. For those SPU and PPU intrinsic functions for which there is no straightforward mapping, an explanation of the difficulty in mapping is provided.

The mappings between SPU and PPU VMX intrinsics are defined in two header files: `vmx2spu.h` and `spu2vmx.h`. The former maps PPU VMX intrinsics to generic SPU intrinsics, and the latter maps generic SPU intrinsics to PPU VMX intrinsics. The functions that are defined in these two header files may be implemented as overloaded inline functions. To facilitate implementation, the vector data types are also mapped.

On the SPU, the header file `vec_types.h` defines the single token vector data types that are available on the PPU. These data types are listed in Table 1-4. The actual PPU VMX types of `vec_bchar16`, `vec_bshort8`, `vec_bint4`, and `vec_pixel8` will be as described in Table 1-2. On the PPU, the header file `spu2vmx.h` defines the single token vector data types that are available on the SPU. These data types are also listed in Table 1-4. The actual PPU VMX types of `vec_llong2`, `vec_ullong2`, and `vec_double2` will be as described in Table 1-3.

The following guidelines describe how to write code that uses these intrinsics and that is portable between the SPU and PPU:

- Always use the single vector token typedefs described in Table 1-4.
- Only use the intrinsics that are mapped in `spu2vmx.h` or `vmx2spu.h`.
- When using SPU intrinsics, include the headers in the following way:
  ```
  #ifdef __SPU__
  #include <spu_intrinsics.h>
  #else
  #include <spu2vmx.h>
  #endif
  ```
- When using PPU VMX intrinsics, include the headers in the following way:
  ```
  #include <vec_types.h>
  #ifdef __PPU__
  #include <altivec.h>
  #else
  #include <vmx2spu.h>
  #endif
  ```

## 5.1. Mapping of PPU VMX Intrinsics to SPU Intrinsics

This section lists the one-to-one mapping of PPU VMX intrinsics to SPU intrinsics. It also lists those PPU VMX intrinsics that are difficult to map to SPU intrinsics.

### 5.1.1. One-to-One Mapped Intrinsics

The PPU VMX intrinsics that map one-to-one with the generic SPU intrinsics are shown in Table 5-111.

Table 5-111: PPU VMX Intrinsics That Map One-to-One with SPU Intrinsics

| Generic PPU VMX Intrinsic | Maps to SPU Intrinsic | Applicable Data Type(s) |
|---|---|---|
| vec_add | spu_add | halfword, word, and float (not byte) |
| vec_addc | spu_genc | All |
| vec_and | spu_and | All |
| vec_andc | spu_andc | All |
| vec_avg | spu_avg | unsigned char |
| vec_cmpeq | spu_cmpeq | All |
| vec_cmpgt | spu_cmpgt | All |
| vec_cmplt | spu_cmpgt | All (requires parameter reordering) |
| vec_ctf | spu_convtf | All |
| vec_cts | spu_convts | All |
| vec_ctu | spu_convtu | All |
| vec_madd | spu_madd | all |
| vec_mule | spu_mule | halfword (not byte) |
| vec_mulo | spu_mulo | halfword (not byte) |
| vec_nmusb | spu_nmsub | All |
| vec_nor | spu_nor | All |
| vec_or | spu_or | All |
| vec_re | spu_re | All |
| vec_rl | spu_rl | halfword, word (not byte) |
| vec_rsqrte | spu_rsqrte | All |
| vec_sel | spu_sel | All |
| vec_sub | spu_sub | halfword, word, float |
| vec_subc | spu_genb | All |
| vec_xor | spu_xor | all |

### 5.1.2. PPU VMX Intrinsics That Are Difficult to Map to SPU Intrinsics

The PPU VMX intrinsics that are shown in Table 5-112 are not likely to be mapped to generic SPU intrinsics because a straightforward mapping does not exist.

Table 5-112: PPU VMX Intrinsics That Are Difficult to Map to SPU Intrinsics

| Generic PPU VMX Intrinsic(s) | Explanation |
|---|---|
| vec_unpackh, vec_unpackl | These functions cannot be mapped without creating additional SPU data types. A mapping of `pixel` and `bool short` vector types to an `unsigned short` (as described in Table 1-2) will cause an overloaded function selection conflict. |
| vec_mfvscr, vec_mtvscr | Support of the VSCR register is difficult because the SPU does not support IEEE rounding modes on single-precision floating-point operations. |
| vec_step | Mapping requires specific compiler support that is not mandated by this specification. |

## 5.2. Mapping of SPU Intrinsics to PPU VMX Intrinsics

This section lists the one-to-one mapping of SPU intrinsics to PPU VMX intrinsics. It also lists those SPU intrinsics that are difficult to map to PPU VMX intrinsics.

### 5.2.1. One-to-One Mapped Intrinsics

Many of the generic SPU intrinsics map one-to-one with PPU VMX intrinsics. These mappings are shown in Table 5-113.

Table 5-113: SPU Intrinsics That Map One-to-One with PPU VMX Intrinsics

| Generic SPU Intrinsic | Maps to PPU VMX Intrinsic | Applicable Data Type(s) |
|---|---|---|
| spu_add | vec_add | vector/vector (no scalar operands) |
| spu_and | vec_and | vector/vector (no scalar operands) |
| spu_andc | vec_andc | All |
| spu_avg | vec_avg | All |
| spu_cmpeq | vec_cmpeq | vector/vector (no scalar operands) |
| spu_cmpgt | vec_cmpgt | vector/vector (no scalar operands) |
| spu_convtf | vec_ctf | Limited scale range (5 bits) |
| spu_convts | vec_cts | Limited scale range (5 bits) |
| spu_convtu | vec_ctu | Limited scale range (5 bits) |
| spu_genb | vec_subc | All |
| spu_genc | vec_addc | All |
| spu_madd | vec_madd | float |
| spu_mule | vec_mule | All |
| spu_mulo | vec_mulo | Halfword vector/vector (no scalar operands) |
| spu_nmsub | vec_nmsub | float |
| spu_nor | vec_nor | All |
| spu_or | vec_or | vector/vector (no scalar operands) |
| spu_re | vec_re | All |
| spu_rl | vec_rl | vector/vector (no scalar operands) |
| spu_rsqrte | vec_rsqrte | all |
| spu_sel | vec_sel | All |
| spu_sub | vec_sub | vector/vector (no scalar operands) |
| spu_xor | vec_xor | vector/vector (no scalar operands) |

### 5.2.2. SPU Intrinsics That Are Difficult to Map to PPU VMX Intrinsics

The generic SPU intrinsics that are shown in Table 5-114 are not likely to be mapped to PPU VMX intrinsics because a straightforward mapping does not exist.

Table 5-114: SPU Intrinsics That Are Difficult to Map to PPU VMX Intrinsics

| Generic SPU Intrinsic(s) | Explanation |
|---|---|
| spu_bisled, spu_bislede, spu_bisledi<br>spu_idisable, spu_ienable | Event handling and interrupt handling on the SPU cannot be precisely mapped. |
| spu_readch, spu_readchqw, spu_readchcnt<br>spu_writech, spu_writechqw | Specific channel functionality cannot be easily supported on the PPU, nor would it generally be desirable to do so. Whereas some channel sequences could be mapped, most would require special programmer insight and direction. |
| spu_mfcdma32, spu_mfcdma64, spu_mfcstat | The mapping of DMA transactions typically is not needed because the PPU has full memory access. Nevertheless, these intrinsics could be used to perform memory synchronization that might not be precisely mappable. |
| spu_sync, spu_sync_c<br>spu_dsync | These intrinsics could be mapped to one of the PPU sync instructions, but the results might not be what was intended. |

| Generic SPU Intrinsic(s) | Explanation |
|---|---|
| spu_convts, spu_convtu, spu_convtf | The full dynamic range of scale factors is not easily supported. Vector Multimedia Extension provides a 5-bit scale factor; the SPU has an 8-bit scale factor. Some implementations might support only the 5-bit range provided by the direct mapping of the equivalent intrinsics. |
| spu_hcmpeq, spu_hcmpgt | The halt instruction might be mappable to an exit function, but this will not work in all environments. |
| spu_stop, spu_stopd | It is not always appropriate to stop execution of the PPU. |

# 6. PPU Specific Intrinsics

This chapter specifies a minimal set of specific intrinsics to make the underlying PPU instruction set accessible from the C programming language. Except for `__setflm`, each of these intrinsics has a one-to-one assembly language mapping, unless compiled for a 32-bit ABI in which the high and low halves of a 64-bit doubleword are maintained in separate registers. In this latter situation, the corresponding 32-bit intrinsic might generate a sequence of instructions. In other instances, a corresponding 32-bit implementation cannot be supported.

The PPU intrinsics will be declared in the system header file, `ppu_intrinsics.h`. They may be either defined within this header as macros or implemented internally within the compiler.

Some intrinsics take a literal value of either 3, 4, 5, 6, 8, or 10 bits in length. By default, a call to an intrinsic with an out-of-range literal is reported by the compiler as an error. Compilers may provide an option to issue a warning for out-of-range literal values and use only the specified number of least significant bits for the out-of-range argument.

The intrinsics do not have a specific ordering unless otherwise noted. The intrinsics can be optimized by the compiler and be scheduled like normal operations.

### __cctph: Change Thread Priority to High

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

The current thread priority is changed to high priority. This intrinsic will not be reordered by the compiler.

Table 6-115: Change Thread Priority to High

| Return/Argument Types | Assembly Mapping |
|---|---|
| none | cctph |

### __cctpl: Change Thread Priority to Low

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

The current thread priority is changed to low priority. This intrinsic will not be reordered by the compiler.

Table 6-116: Change Thread Priority to Low

| Return/Argument Types | Assembly Mapping |
|---|---|
| none | cctpl |

### __cctpm: Change Thread Priority to Medium

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

The current thread priority is changed to medium priority. This intrinsic will not be reordered by the compiler.

Table 6-117: Change Thread Priority to Medium

| Return/Argument Types | Assembly Mapping |
|---|---|
| none | cctpm |

### __cntlzd: Count Leading Doubleword Zeros

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

The number of leading zeros in the doubleword $a$ is returned in $d$.

Table 6-118: Count Leading Doubleword Zeros

| Return/Argument Types | | Assembly Mapping | |
|---|---|---|---|
| d | a | 64-bit ABI | 32-bit ABI |
| unsigned int | unsigned long long | cntlzd d, a | cntlzw hi_cnt, a_hi<br>cntlzw lo_cnt, a_lo<br>rlwinm mask, hi_cnt, 26, 0, 5<br>srawi  mask, mask, 31<br>and    lo_cnt, lo_cnt, mask<br>add    d, hi_cnt, lo_cnt |

### __cntlzw: Count Leading Word Zeros

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

The number of leading zeros in the word $a$ is returned in $d$.

Table 6-119: Count Leading Word Zeros

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| unsigned int | unsigned int | cntlzw d, a |

### __db10cyc: Delay 10 Cycles at Dispatch

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

The current thread is blocked at dispatch for 10 cycles. This intrinsic will not be reordered by the compiler.

Table 6-120: Delay 10 Cycles at Dispatch

| Return/Argument Types | Assembly Mapping |
|---|---|
| none | db10cyc |

### __db12cyc: Delay 12 Cycles at Dispatch

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

The current thread is blocked at dispatch for 12 cycles. This intrinsic will not be reordered by the compiler.

Table 6-121: Delay 12 Cycles at Dispatch

| Return/Argument Types | Assembly Mapping |
|---|---|
| none | db12cyc |

### __db16cyc: Delay 16 Cycles at Dispatch

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

The current thread is blocked at dispatch for 16 cycles. This intrinsic will not be reordered by the compiler.

Table 6-122: Delay 16 Cycles at Dispatch

| Return/Argument Types | Assembly Mapping |
|---|---|
| none | db16cyc |

### __db8cyc: Delay 8 Cycles at Dispatch

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

The current thread is blocked at dispatch for 8 cycles. This intrinsic will not be reordered by the compiler.

Table 6-123: Delay 8 Cycles at Dispatch

| Return/Argument Types | Assembly Mapping |
|---|---|
| none | db8cyc |

### __dcbf: Data Cache Block Flush

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

The cache block that contains the argument *pointer* is flushed and removed from the cache.

The *base* and *index* arguments for the assembly mapping are calculated from *pointer*.

Table 6-124: Data Cache Block Flush

| Return/Argument Types | Assembly Mapping |
| pointer | |
|---|---|
| void* | dcbf base, index |

### __dcbst: Data Cache Block Store

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

The cache block that contains the argument *pointer* is written to main memory. This intrinsic will not be reordered by the compiler.

The *base* and *index* arguments for the assembly mapping are calculated from *pointer*.

Table 6-125: Data Cache Block Store

| Return/Argument Types | Assembly Mapping |
| pointer | |
|---|---|
| void* | dcbst base, index |

**__dcbt: Data Cache Block Touch**

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

The processor receives a hint that the cache block which contains the argument *pointer* will soon be loaded. This intrinsic will not be reordered by the compiler.

The *base* and *index* arguments for the assembly mapping are calculated from *pointer*.

Table 6-126: Data Cache Block Touch

| Return/Argument Types | Assembly Mapping |
|---|---|
| pointer | |
| void* | dcbt base, index |

**__dcbt_TH1000: Set Up Streaming Data**

```
(void) __dcbt_TH1000(eatrunc, d, ug, id)
```

A stream is set up with an id of *id* and an effective address of `eatrunc`. The argument *d* describes which direction the stream is going: `true` for forwards and `false` for backwards. The argument *ug* says if the stream is unlimited in bounds or not. This intrinsic will not be reordered by the compiler.

The effective address for this instruction is calculated as:

```
((unsigned long long) eatrunc) & ~0x7F) | (((d & 1) << 6) | ((ug & 1) << 5) | (id & 0xF)
```

The *base* and *index* arguments for the assembly mapping are calculated from the above effective address.

Table 6-127: Set Up Streaming Data

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| eatrunc | d | ug | id | |
| void* | bool | bool | int | dcbt base, index, 8 |

**__dcbt_TH1010: Start or Stop Streaming Data**

```
(void) __dcbt_TH1010(go, s, unitcnt, t, u, id)
```

The processor receives a hint that the stream identified by *id* will no longer be needed. If *go* is set, the program will soon load from all nascent data streams that have been completely described, and it will probably no longer load from any other nascent data streams; all the rest of the arguments are ignored in this case. If *s* is '10', the stream associated with *id* will stop and all other arguments except for *id* are ignored. If *s* is '11', all streams IDs are stopped and all other arguments are ignored. *unitcnt* specifies the number of units in a data stream. *t* tells if the program's need for each block of the data stream is likely to be transient. *u* tells if the data stream is unlimited and the *unitcnt* argument is ignored. This intrinsic will not be reordered by the compiler.

The effective address for this instruction is calculated as:

```
(((unsigned long long) go & 1) << 31)

  | ((s & 0x3) << 29)
  | ((unitcnt & 0x3FF) << 7)
  | ((t & 1) << 6)
  | ((u & 1) << 5)
  | (id & 0xF)
```

The *base* and *index* arguments for the assembly mapping are calculated from the above effective address.

Table 6-128: Start or Stop Streaming Data

| Return/Argument Types | | | | | | Assembly Mapping |
|---|---|---|---|---|---|---|
| go | s | unitcnt | t | u | id | |
| bool | int | int | bool | bool | int | dcbt base, index, 10 |

**__dcbtst: Data Cache Block Touch for Store**

```
(void) __dcbtst(pointer)
```

The processor receives a hint that the cache block that contains the argument *pointer* will soon be stored. This intrinsic will not be reordered by the compiler.

The *base* and *index* arguments for the assembly mapping are calculated from *pointer*.

Table 6-129: Data Cache Block Touch for Store

| Return/Argument Types | Assembly Mapping |
|---|---|
| pointer | |
| void* | dcbtst base, index |

**__dcbz: Data Cache Block Set to Zero**

```
(void) __dcbz(pointer)
```

The cache block that contains the argument *pointer* is zeroed out. If the address is already in cache, the cache block containing it is zeroed. If the address was not already in a cache block, a cache block for it is created with all zeros. This intrinsic will not be reordered by the compiler.

The *base* and *index* arguments for the assembly mapping are calculated from *pointer*.

Table 6-130: Data Cache Block Set to Zero

| Return/Argument Types | Assembly Mapping |
|---|---|
| pointer | |
| void* | dcbz base, index |

**__eieio: Enforce In-Order Execution of I/O**

```
(void) __eieio()
```

A memory barrier is created, which provides an ordering function for the storage accesses caused by *Load*, *Store*, __dcbz(), __eciwx(), and __ecowx() instructions executed by the processor executing the __eieio() instruction. The memory barrier and ordering function are described in section 1.7.1 of *PowerPC Architecture Book, Book II: PowerPC Virtual Environment Architecture*, version 2.02.

Table 6-131: Enforce In-Order Execution of I/O

| Return/Argument Types | Assembly Mapping |
|---|---|
| none | eieio |

**__fabs: Double Absolute Value**

```
d = __fabs(a)
```

The absolute value of the argument *a* is returned in *d*  with the sign bit set to zero.

Table 6-132: Double Absolute Value

| Return/Argument Types | | Assembly Mapping |
| --- | --- | --- |
| d | a | |
| double | double | fabs d, a |

**__fabsf: Float Absolute Value**

```
d = __fabsf(a)
```

The absolute value of the argument *a* is returned in *d* with the sign bit set to zero.

Table 6-133: Float Absolute Value

| Return/Argument Types | | Assembly Mapping |
| --- | --- | --- |
| d | a | |
| float | float | fabs d, a |

**__fcfid: Convert Doubleword to Double**

```
d = __fcfid(a)
```

The doubleword in *a* is converted to a floating-point and returned in *d*.

Table 6-134: Convert Doubleword to Double

| Return/Argument Types | | Assembly Mapping |
| --- | --- | --- |
| d | a | |
| double | long long | fcfid d, a |

**__fctid: Convert Double to Doubleword**

```
d = __fctid(a)
```

The `double` *a* is converted to a doubleword integer and returned in *d*. This function takes into account the current rounding mode.

Table 6-135: Convert Double to Doubleword

| Return/Argument Types | | Assembly Mapping |
| --- | --- | --- |
| d | a | |
| long long | double | fctid d, a |

**__fctidz: Convert Double to Doubleword with Round Towards Zero**

```
d = __fctidz(a)
```

The `double` *a* is converted to a doubleword integer and returned in *d*. This function always rounds towards zero.

Table 6-136: Convert Double to Doubleword with Round Towards Zero

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| long long | double | fctidz d, a |

**__fctiw: Convert Double to Word**

```
d = __fctiw(a)
```

The `double` *a* is converted to a word integer and returned in *d*. This function takes into account the current rounding mode.

Table 6-137: Convert Double to Word

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| int | double | fctiw tmp, a<br>stfiwx tmp, r1, tempspace<br>lwzx d, r1, tempspace |

**__fctiwz: Convert Double to Word with Round Towards Zero**

```
d = __fctiwz(a)
```

The `double` *a* is converted to a word integer and returned in *d*. This function always rounds towards zero.

Table 6-138: Convert Double to Word with Round Towards Zero

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| int | double | fctiwz tmp, a<br>stfiwx tmp, r1, tempspace<br>lwzx d, r1, tempspace |

**__fmadd: Double Fused Multiply and Add**

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

The argument *a* is multiplied by the argument *b*, and the argument *c* is added to that product. The resulting value ($a{\times}b{+}c$) is returned in *d*.

Table 6-139: Double Fused Multiply and Add

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | c | |
| double | double | double | double | fmadd d, a, b, c |

**__fmadds: Float Fused Multiply and Add**

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

The argument $a$ is multiplied by the argument $b$, and the argument $c$ is added to that product. The resulting value ($a{\times}b{+}c$) is returned in $d$.

Table 6-140: Float Fused Multiply and Add

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | c | |
| float | float | float | float | fmadds d, a, b, c |

**__fmsub: Double Fused Multiply and Subtract**

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

The argument $a$ is multiplied by the argument $b$, and the argument $c$ is subtracted from that product. The resulting value ($a{\times}b{-}c$) is returned in $d$.

Table 6-141: Double Fused Multiply and Subtract

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | c | |
| double | double | double | double | fmsub d, a, b, c |

**__fmsubs: Float Fused Multiply and Subtract**

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

The argument $a$ is multiplied by the argument $b$, and the argument $c$ is subtracted from that product. The resulting value ($a{\times}b{-}c$) is returned in $d$.

Table 6-142: Float Fused Multiply and Subtract

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | c | |
| float | float | float | float | fmsubs d, a, b, c |

**__fmul: Double Multiply**

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

The doubles $a$ and $b$ are multiplied, and their product ($a{\times}b$) is returned in $d$.

Table 6-143: Double Multiply

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| d | a | b | |
| double | double | double | fmul d, a, b |

**__fmuls: Float Multiply**

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

The floats $a$ and $b$ are multiplied, and their product (a×b) is returned in $d$.

Table 6-144: Float Multiply

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| d | a | b | |
| float | float | float | fmuls d, a, b |

**__fnabs: Double Negative**

```
d = __fnabs(a)
```

The negative absolute value of the argument $a$ is returned in $d$. The sign bit is set to 1.

Table 6-145: Double Negative

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| double | double | fnabs d, a |

**__fnabsf: Float Negative**

```
d = __fnabsf(a)
```

The negative absolute value of the argument $a$ is returned in the $d$. The sign bit is set to 1.

Table 6-146: Float Negative

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| float | float | fnabs d, a |

**__fnmadd: Double Fused Negative Multiply and Add**

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

The arguments $a$ and $b$ are multiplied, and the argument $c$ is added to their product. The sum is negated, and the resulting value -($a$×$b$+$c$) is returned in $d$.

Table 6-147: Double Fused Negative Multiply and Add

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | c | |
| double | double | double | double | fnmadd d, a, b, c |

### __fnmadds: Float Fused Negative Multiply and Add

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

The arguments $a$ and $b$ are multiplied, and the argument $c$ is added to their product. The sum is negated, and the resulting value -($a{\times}b$+$c$) is returned in $d$.

Table 6-148: Float Fused Negative Multiply and Add

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | c | |
| float | float | float | float | fnmadds d, a, b, c |

### __fnmsub: Double Fused Negative Multiply and Subtract

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

The arguments $a$ and $b$ are multiplied, and the argument $c$ is subtracted from their product. The sum is negated, and the resulting value -($a{\times}b$-$c$) is returned in $d$.

Table 6-149: Double Fused Negative Multiply and Subtract

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | c | |
| double | double | double | double | fnmsub d, a, b, c |

### __fnmsubs: Float Fused Negative Multiply and Subtract

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

The arguments $a$ and $b$ are multiplied, and the argument $c$ is subtracted from their product. The sum is negated, and the resulting value -($a{\times}b$-$c$) is returned in $d$.

Table 6-150: Float Fused Negative Multiply and Subtract

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | c | |
| float | float | float | float | fnmsubs d, a, b, c |

### __fres: Float Reciprocal Estimate

```
d = __fres(a)
```

An estimate of the reciprocal of the argument $a$ is returned in $d$. The estimate is correct to a precision of one part in 256 of the reciprocal.

Beyond this precision, the value is indeterminate; the results of executing this instruction may vary between implementations and between different executions on the same implementation.

Table 6-151: Float Reciprocal Estimate

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| float | float | fres d, a |

### __frsp: Round to Single Precision

```
d = __frsp(a)
```

The argument *a* is rounded to single precision and returned in *d*.

Table 6-152: Round to Single Precision

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| float | double | frsp d, a |

### __frsqrte: Double Reciprocal Square Root Estimate

```
d = __frsqrte(a)
```

An estimate of the reciprocal of the square root of the argument *a* is returned in *d*.

The estimate is correct to a precision of one part in 32 of the reciprocal of the square root. Beyond this precision, the value is indeterminate; the results of executing this instruction may vary between implementations and between different executions on the same implementation.

Table 6-153: Double Reciprocal Square Root Estimate

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| double | double | frsqrte d, a |

### __fsel: Floating-Point Select of Double

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

The argument *b* is returned in *d* if the argument *a* is greater than or equal to 0.0; otherwise *c* is returned.

Table 6-154: Floating-Point Select of Double

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | c | |
| double | double | double | double | fsel d, a, b, c |

### __fsels: Floating-Point Select of Float

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

The argument *b* is returned in *d* if the argument *a* is greater than or equal to 0.0; otherwise *c* is returned.

Table 6-155: Floating-Point Select of Float

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | c | |
| float | float | float | float | fsel d, a, b, c |

### __fsqrt: Double Square Root

```
d = __fsqrt(a)
```

The square root of the argument *a* is returned in *d*.

Table 6-156: Double Square Root

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| double | double | fsqrt d, a |

### __fsqrts: Float Square Root

```
d = __fsqrts(a)
```

The square root of the argument *a* is returned in *d*.

Table 6-157: Float Square Root

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| float | float | fsqrts d, a |

### __icbi: Instruction Cache Block Invalidate

```
(void) __icbi(pointer)
```

The instruction cache block that contains the argument `pointer` is invalidated, if such a block is in the cache. This intrinsic will not be reordered by the compiler.

The `base` and `index` arguments for the assembly mapping are calculated from `pointer`.

Table 6-158: Instruction Cache Block Invalidate

| Return/Argument Types | Assembly Mapping |
|---|---|
| pointer | |
| void* | icbi base, index |

### __isync: Instruction Sync

```
(void) __isync()
```

The processor waits until all previous instructions have finished. The `__isync()` function ensures that all `icbi` have been performed.

Table 6-159: Instruction Sync

| Return/Argument Types | Assembly Mapping |
|---|---|
| none | isync |

### __ldarx: Load Doubleword with Reserved

```
d = __ldarx(pointer)
```

The reserved address of the processor is set to the value of *pointer*. A doubleword from the address in *pointer* is returned in *d*.

The *base* and *index* arguments for the assembly mapping are calculated from *pointer*.

This intrinsic might not be supported when compiling for 32-bit ABIs in which a 64-bit doubleword is maintained in two separate registers.

Table 6-160: Load Doubleword with Reserved

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | pointer | |
| unsigned long long | void* | ldarx d, base, index |

### __ldbrx: Load Reversed Doubleword

```
d = __ldbrx(pointer)
```

A doubleword from the address in *pointer* is loaded in reversed endian order into *d* and returned.

The *base* and *index* arguments for the assembly mapping are calculated from *pointer*.

Table 6-161: Load Reversed Doubleword

| Return/Argument Types | | Assembly Mapping | |
|---|---|---|---|
| d | pointer | 64-bit ABI | 32-bit ABI |
| unsigned long long | void* | ldbrx d, base, index | lwbrx d_lo, base, index<br>lwbrx d_hi, base, index+4 |

### __lhbrx: Load Reversed Halfword

```
d = __lhbrx(pointer)
```

A halfword from the address in *pointer* is loaded in reversed endian order into *d* and returned.

The *base* and *index* arguments for the assembly mapping are calculated from *pointer*.

Table 6-162: Load Reversed Halfword

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | pointer | |
| unsigned short | void* | lhbrx d, base, index |

### __lwarx: Load Word with Reserved

```
d = __lwarx(pointer)
```

The reserved address of the processor is set to the value of *pointer*. A word from the address in *pointer* is returned in *d*.

The *base* and *index* arguments for the assembly mapping are calculated from *pointer*.

Table 6-163: Load Word with Reserved

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | pointer | |
| unsigned | void* | lwarx d, base, index |

### __lwbrx: Load Reversed Word

```
d = __lwbrx(pointer)
```

A word from the address in *pointer* is loaded in reversed endian order into *d*.

The *base* and *index* arguments for the assembly mapping are calculated from *pointer*.

Table 6-164: Load Reversed Word

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | pointer | |
| unsigned | void* | lwbrx d, base, index |

### __lwsync: Light Weight Sync

```
(void) __lwsync()
```

A memory barrier is created, providing an ordering function for the storage accesses caused by prior *Load*, *Store*, and __dcbz() instructions that are executed by the processor executing __lwsync(). The memory barrier and ordering function are described in section 1.7.1 of *PowerPC Architecture Book, Book II: PowerPC Virtual Environment Architecture*, version 2.02.

Table 6-165: Light Weight Sync

| Return/Argument Types | Assembly Mapping |
|---|---|
| none | lwsync |

### __mffs: Move from Floating-Point Status and Control Register

```
d = __mffs()
```

The current Floating-Point Status and Control Register is returned in *d*. This intrinsic will not be reordered by the compiler.

Table 6-166: Move from Floating-Point Status and Control Register

| Return/Argument Types | Assembly Mapping |
|---|---|
| d | |
| double | mffs d |

### __mfspr: Move from Special Purpose Register

```
d = __mfspr(spr)
```

The contents of the special purpose register specified by *spr* are returned in *d*. This intrinsic will not be reordered by the compiler.

This intrinsic might not be supported when compiling for 32-bit ABIs in which a 64-bit doubleword is maintained in two separate registers.

Table 6-167: Move from Special Purpose Register

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | spr | |
| unsigned long long | 10-bit literal unsigned int | mfspr d, spr |

**__mftb: Move from Time Base**

```
d = __mftb()
```

The time base register is returned in *d*. This intrinsic will not be reordered by the compiler.

Table 6-168: Move from Time Base

| Return/Argument Types | Assembly Mapping | |
|---|---|---|
| d | 64-bit ABI | 32-bit ABI |
| unsigned long long | mftb d | retry:<br> mftbu d_hi<br> mftb d_lo<br> mftbu tmp<br> cmp d_hi, tmp<br> bne retry |

**__mtfsb0: Reset Bit of FPSCR**

```
(void) __mtfsb0(bt)
```

Bit *bt* of Floating-Point Status and Control Register (FPSCR) is set to 0. This intrinsic will not be reordered by the compiler. It will also cause a barrier for floating-point operations.

Table 6-169: Reset Bit of FPSCR

| Return/Argument Types | Assembly Mapping |
|---|---|
| bt | |
| 5-bit unsigned int (literal) | mtfsb0 bt |

**__mtfsb1: Set Bit of FPSCR**

```
(void) __mtfsb1(bt)
```

Bit *bt* of Floating-Point Status and Control Register is set to 1. This intrinsic will not be reordered by the compiler. It will also cause a barrier for floating-point operations.

Table 6-170: Set Bit of FPSCR

| Return/Argument Types | Assembly Mapping |
|---|---|
| bt | |
| 5-bit unsigned int (literal) | mtfsb1 bt |

**__mtfsf: Set Fields in FPSCR**

```
(void) __mtfsf(flm,b)
```

The fields of Floating-Point Status and Control Register are set to *b* masked by the argument *flm*. This intrinsic will not be reordered by the compiler. It will also cause a barrier for floating-point operations.

Table 6-171: Set Fields in FPSCR

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| flm | b | |
| 8-bit unsigned int (literal) | double | mtfsf flm, b |

### __mtfsfi: Set Field of FPSCR

```
(void) __mtfsfi(bf, u)
```

The *bf* field of FPSCR is set to the argument *u*. This intrinsic will not be reordered by the compiler. It will also cause a barrier for floating-point operations.

Table 6-172: Set Field of FPSCR

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| bf | u | |
| 3-bit unsigned int (literal) | 4-bit unsigned int (literal) | mtfsfi bf, u |

### __mtspr: Move to Special Purpose Register

```
(void) __mtspr(spr, value)
```

The special purpose register specified by *spr* is set to the argument *value*. This intrinsic will not be reordered by the compiler.

This intrinsic might not be supported when compiling for 32-bit ABIs in which a 64-bit doubleword is maintained in two separate registers.

Table 6-173: Move to Special Purpose Register

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| spr | value | |
| 10-bit unsigned int (literal) | unsigned long long | mtspr spr, value |

### __mulhd: Multiply Doubleword, High Part

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

The high part of the signed product of the doubleword arguments *a* and *b* is returned in *d*.

This intrinsic might not be supported when compiling for 32-bit ABIs in which a 64-bit doubleword is maintained in two separate registers.

Table 6-174: Multiply Doubleword, High Part

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| d | a | b | |
| long long | long long | long long | mulhd d, a, b |

### __mulhdu: Multiply Double Unsigned Word, High Part

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

The high part of the unsigned product of the doubleword arguments *a* and *b* is returned in *d*.

This intrinsic might not be supported when compiling for 32-bit ABIs in which a 64-bit doubleword is maintained in two separate registers.

Table 6-175: Multiply Double Unsigned Word, High Part

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| d | a | b | |
| unsigned long long | unsigned long long | unsigned long long | mulhdu d, a, b |

### __mulhw: Multiply Word, High Part

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

The high part of the signed product of the word arguments *a* and *b* is returned in *d*.

Table 6-176: Multiply Word, High Part

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| d | a | b | |
| int | int | int | mulhw d, a, b |

### __mulhwu: Multiply Unsigned Word, High Part

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

The high part of the unsigned product of the word arguments *a* and *b* is returned in *d*.

Table 6-177: Multiply Unsigned Word, High Part

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| d | a | b | |
| unsigned int | unsigned int | unsigned int | mulhwu d, a, b |

### __nop: No Operation

```
(void) __nop()
```

The preferred nop instruction is generated. This intrinsic will not be reordered by the compiler.

Table 6-178: No Operation

| Return/Argument Types | Assembly Mapping |
|---|---|
| none | nop |

### __protected_stream_count: Set the Number of Blocks to Stream

```
(void) __protected_stream_count(COUNT, ID)
```

Set the number of units in the data stream corresponding to stream ID. This intrinsic is an alias for __dcbt_TH1010(0, 0, COUNT, 0, 0, ID). The compiler will not reorder this intrinsic.

### __protected_stream_go: Start All Streams

```
(void) __protected_stream_go()
```

Start all of the completely described streams. This intrinsic is an alias for __dcbt_TH1010(1, 0, 0, 0, 0, 0). The compiler will not reorder this intrinsic.

### __protected_stream_set: Set Up a Stream

```
(void) __protected_stream_set(D, ADDR, ID)
```

Set up the ID stream to start at *ADDR* and run in the direction of *D*. When *D* is 1, the direction is backwards (decrementing), and when *D* is 3, the direction is forwards (incrementing). The stream is started by setting the count

and then calling `__protected_stream_go`. This intrinsic is an alias for `__dcbt_TH1000(ADDR, (D>>1), 0, ID)`. The compiler will not reorder this intrinsic.

### __protected_stream_stop: Stop a Stream

```
(void) __protected_stream_stop(ID)
```

Stop the ID stream. This intrinsic is an alias for `__dcbt_TH1010(0, 2, 0, 0, 0, ID)`. The compiler will not reorder this intrinsic.

### __protected_stream_stop_all: Stop All Streams

```
(void) __protected_stream_stop_all()
```

Stop all data streams. This intrinsic is an alias for `__dcbt_TH1010(0, 3, 0, 0, 0, 0)`. The compiler will not reorder this intrinsic.

### __protected_unlimited_stream_set: Set Up an Unlimited Stream

```
(void) __protected_unlimited_stream_set(D, ADDR, ID)
```

Set up the ID stream to start at *ADDR* and run for an unlimited count in the direction of *D*. When *D* is 1, the direction is backwards (decrementing), and when *D* is 3, the direction is forwards (incrementing). The stream is started by calling `__protected_stream_go`. This intrinsic is an alias for `__dcbt_TH1000(ADDR, (D>>1), 1, ID)`. The compiler will not reorder this intrinsic.

### __rldcl: Rotate Left Doubleword then Clear Left

```
d = __rldcl(a, b, mb)
```

The value in the argument *a* is rotated leftwards by the number of bits specified by the argument *b*. A mask is generated having 1-bits from bit *mb* through bit 63, and 0-bits elsewhere. The rotated data ANDed with the generated mask is returned into *d*.

This intrinsic might not be supported when compiling for 32-bit ABIs in which a 64-bit doubleword is maintained in two separate registers.

Table 6-179: Rotate Left Doubleword then Clear Left

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | mb | |
| unsigned long long | unsigned long long | unsigned long long | 6-bit unsigned int (literal) | rldcl d, a, b, mb |

**__rldcr: Rotate Left Doubleword then Clear Right**

```
d = __rldcr(a, b, me)
```

The value in the argument *a* is rotated leftwards by the number of bits specified by the argument *b*. A mask is generated having 1-bits from bit 0 though bit *me* and 0-bits elsewhere. The rotated data ANDed with the generated mask is returned in *d*.

This intrinsic might not be supported when compiling for 32-bit ABIs in which a 64-bit doubleword is maintained in two separate registers.

Table 6-180: Rotate Left Doubleword then Clear Right

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | b | me | |
| unsigned long long | unsigned long long | unsigned long long | 6-bit unsigned int (literal) | rldcr d, a, b, me |

**__rldic: Rotate Left Doubleword Immediate then Clear**

```
d = __rldic(a, sh, mb)
```

The value in the argument *a* is rotated leftwards by the number of bits specified by the argument *sh*. A mask is generated having 1-bits from bit *mb* through bit $63-sh$ and 0-bits elsewhere. The rotated data ANDed with the generated mask is returned in *d*.

This intrinsic might not be supported when compiling for 32-bit ABIs in which a 64-bit doubleword is maintained in two separate registers.

Table 6-181: Rotate Left Doubleword Immediate then Clear

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | sh | mb | |
| unsigned long long | unsigned long long | 6-bit unsigned int (literal) | 6-bit unsigned int (literal) | rldic d, a, sh, mb |

**__rldicl: Rotate Left Doubleword Immediate then Clear Left**

```
d = __rldicl(a, sh, mb)
```

The value in the argument *a* is rotated leftwards by the number of bits specified by the argument *sh*. A mask is generated having 1-bits from bit *mb* through bit 63 and 0-bits elsewhere. The rotated data ANDed with the generated mask is returned in *d*.

This intrinsic might not be supported when compiling for 32-bit ABIs in which a 64-bit doubleword is maintained in two separate registers.

Table 6-182: Rotate Left Doubleword Immediate then Clear Left

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | sh | mb | |
| unsigned long long | unsigned long long | 6-bit unsigned int (literal) | 6-bit unsigned int (literal) | rldicl d, a, sh, mb |

**__rldicr: Rotate Left Doubleword Immediate then Clear Right**

```
d = __rldicr(a, sh, me)
```

The value in the argument *a* is rotated leftwards by the number of bits specified by the argument *sh*. A mask is generated having 1-bits from bit 0 though bit *me* and 0-bits elsewhere. The rotated data ANDed with the generated mask is returned in *d*.

This intrinsic might not be supported when compiling for 32-bit ABIs in which a 64-bit doubleword is maintained in two separate registers.

Table 6-183: Rotate Left Doubleword Immediate then Clear Right

| Return/Argument Types | | | | Assembly Mapping |
|---|---|---|---|---|
| d | a | sh | me | |
| unsigned long long | unsigned long long | 6-bit unsigned int (literal) | 6-bit unsigned int (literal) | rldicr d, a, sh, me |

**__rldimi: Rotate Left Doubleword Immediate then Mask Insert**

```
d = __rldimi(a, b, sh, mb)
```

A mask is generated with 1-bits from bit *mb* through bit `63-sh`, and 0-bits elsewhere. The value in *a* is ANDed with the complement of this mask, zeroing out just the bits inside the range *mb* through 63-*sh*. The argument *b* is rotated left by *sh* bits and ANDs the result with the mask, zeroing out all bits outside the range *mb* through 63-*sh*. The two masked values are combined together with inclusive OR, and returned in *d*.

This intrinsic might not be supported when compiling for 32-bit ABIs in which a 64-bit doubleword is maintained in two separate registers.

Table 6-184: Rotate Left Doubleword Immediate then Mask Insert

| Return/Argument Types | | | | | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | sh | mb | |
| unsigned long long | unsigned long long | unsigned long long | 6-bit unsigned int (literal) | 6-bit unsigned int (literal) | mr d, a<br>rldimi d, b, sh, mb |

**__rlwimi: Rotate Left Word Immediate then Mask Insert**

```
d = __rlwimi(a, b, sh, mb, me)
```

A mask is generated with 1-bits from bit *mb* through bit *me*, and 0-bits elsewhere. The value in *a* is ANDed with the complement of this mask, zeroing out just the bits inside the range *mb* through me. The argument *b* is rotated left by *sh* bits and ANDs the result with the mask, zeroing out all bits outside the range *mb* through *me*. The two masked values are combined together with inclusive OR, and returned in *d*.

Table 6-185: Rotate Left Word Immediate then Mask Insert

| Return/Argument Types | | | | | | Assembly Mapping |
|---|---|---|---|---|---|---|
| d | a | b | sh | mb | me | |
| unsigned int | unsigned int | unsigned int | 5-bit unsigned int (literal) | 5-bit unsigned int (literal) | 5-bit unsigned int (literal) | mr d, a<br>rlwimi d, b, sh, mb, me |

**__rlwinm: Rotate Left Word Immediate then AND With Mask**

```
d = __rlwinm(a,sh,mb,me)
```

A mask is generated with 1-bits from *mb* through bit *me*, and 0-bits elsewhere. The value in *a* is rotated left by *sh* bits, then ANDed with this mask, and returned in *d*.

Table 6-186: Rotate Left Word Immediate then AND With Mask

| Return/Argument Types | | | | | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | sh | mb | me | |
| unsigned int | unsigned int | 5-bit unsigned int (literal) | 5-bit unsigned int (literal) | 5-bit unsigned int (literal) | rlwinm d, a, sh, mb, me |

**__rlwnm: Rotate Left Word then AND With Mask**

```
d = __rlwnm(a,b,mb,me)
```

The argument *a* is rotated leftwards by the argument *b*. A mask is generated having 1-bits from bit *mb* through bit *me*, and 0-bits elsewhere. The rotated data ANDed with the generated mask is returned in *d*.

Table 6-187: Rotate Left Word then AND With Mask

| Return/Argument Types | | | | | Assembly Mapping |
|---|---|---|---|---|---|
| d | a | b | mb | me | |
| unsigned int | unsigned int | unsigned int | 5-bit unsigned int (literal) | 5-bit unsigned int (literal) | rlwnm d, a, b, mb, me |

**__setflm: Save and Set the FPSCR**

```
d = __setflm(a)
```

The Floating-Point Status and Control Register is set to *a*, and the context of that register is returned in *d*. This intrinsic will not be reordered by the compiler. It will also cause a barrier for floating-point operations.

Table 6-188: Save and Set the FPSCR

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| double | double | mffs d; mtfsf 0xFF, a |

**__stdbrx: Store Reversed Doubleword**

```
(void) __stdbrx(pointer,b)
```

The argument *b* is stored in reversed endian order into the doubleword located at the argument *pointer*.

The *base* and *index* arguments for the assembly mapping are calculated from *pointer*.

Table 6-189: Store Reversed Doubleword

| Return/Argument Types | | Assembly Mapping | |
|---|---|---|---|
| pointer | b | 64-bit ABI | 32-bit ABI |
| void* | unsigned long long | stdbrx b, base, index | stwbrx b_lo, base, index stwbrx b_hi, base, index+4 |

## __stdcx: Store Doubleword Conditional

```
d = __stdcx(pointer, b)
```

If the reserved address of the processor is the value in the argument `pointer`, `b` is stored into the doubleword at the argument `pointer`, and the value of 1 is returned in `d`. Otherwise, the store is not performed, and the value of 0 is returned in `d`.

The `base` and `index` arguments for the assembly mapping are calculated from `pointer`.

The instruction `stdcx.` returns its value in `cr0.eq`, the `equals` field of conditional register 0.

This intrinsic might not be supported when compiling for 32-bit ABIs in which a 64-bit doubleword is maintained in two separate registers.

Table 6-190: Store Doubleword Conditional

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| d | pointer | b | |
| bool | void* | unsigned long long | stdcx. b, base, index; d = cr0.eq |

## __sthbrx: Store Reversed Halfword

```
(void) __sthbrx(pointer, b)
```

The argument `b` is stored in reversed endian order into the halfword located at the argument `pointer`.

The `base` and `index` arguments for the assembly mapping are calculated from `pointer`.

Table 6-191: Store Reversed Halfword

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| pointer | b | |
| void* | unsigned short | sthbrx b, base, index |

## __stwbrx: Store Reversed Word

```
(void) __stwbrx(pointer, b)
```

The argument `b` is stored in reversed endian order into the word located at the argument `pointer`.

The `base` and `index` arguments for the assembly mapping are calculated from `pointer`.

Table 6-192: Store Reversed Word

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| pointer | b | |
| void* | unsigned | stwbrx b, base, index |

**__stwcx: Store Word Conditional**

```
d = __stwcx(pointer, b)
```

If the reserved address of the processor is the value in the argument *pointer*, *b* is stored into the word at the argument *pointer*, and the value of 1 is returned in *d*. Otherwise, the store is not performed, and the value of 0 is returned in *d*.

The *base* and *index* arguments for the assembly mapping are calculated from *pointer*.

The instruction `stwcx.` returns its value in `cr0.eq,` the `equals` field of conditional register 0.

Table 6-193: Store Word Conditional

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| d | pointer | b | |
| bool | void* | unsigned | stwcx. b, base, index; d = cr0.eq |

**__sync: Sync**

```
(void) __sync()
```

A memory barrier is created, providing an ordering function for all instructions executing on the same processor. The memory barrier and ordering function are described in section 1.7.1 of *PowerPC Architecture Book, Book II: PowerPC Virtual Environment Architecture*, version 2.02.

Table 6-194: Sync

| Return/Argument Types | Assembly Mapping |
|---|---|
| none | sync |

# 7. PPU Vector Multimedia Extension Intrinsics

This chapter describes intrinsics which make the underlying PPU Vector Multimedia Extension (VMX) instruction set accessible from the C and C++ programming languages. The *AltiVec™ Technology Programming Interface Manual*, Section 4.4, defines most of the generic intrinsics for the PPU VMX instruction set, except for a few new instructions which are specified in this chapter. The new intrinsics are in two different categories: intrinsics for extracting vector elements and intrinsics for inserting vector elements.

The PPU VMX intrinsics will be declared in the system header file `altivec.h`. These intrinsics may be either defined as macros within this header or implemented internally within the compiler.

For data prefetches, the `__dcbt`, `__dcbtst`, `__dcbt_TH1000`, and `__dcbt_TH1010` intrinsics should be used. The related stream control operations that are defined in the *AltiVec™ Technology Programming Interface Manual*, which are listed below, have been deprecated on the PPU and will execute as a NOP.

Table 7-195: Stream Control Operators That Have Been Deprecated on the PPU

| Stream Control Operator | Description |
|---|---|
| vec_dss(a) | Vector Data Stream Stop |
| vec_dssall() | Vector Stream Stop All |
| vec_dst(a,b,c) | Vector Stream Touch |
| vec_dstst(a,b,c) | Vector Data Stream Touch for Store Transient |

**vec_extract: Extract Vector Element from Vector**

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

The element that is specified by `element` is extracted from vector `a` and returned in scalar `d`. Depending on the size of the element, only a limited number of the least significant bits of the `element` index are used. Specifically for 1-, 2-, and 4-byte elements, only four, three, and two of the least significant bits are used, respectively.

Table 7-196: Extract Vector Element from Vector

| Return/Argument Types | | | Assembly Mapping[1] |
|---|---|---|---|
| d | a | element | |
| unsigned char | vector unsigned char | int | EA=memaddr + (element&0xF)<br>stvebx a, 0, EA<br>lbzx d, 0, EA |
| signed char | vector signed char | | EA=memaddr + (element&0xF)<br>stvebx a, 0, EA<br>lbzx d, 0, EA<br>extsb d, d |
| unsigned short | vector unsigned short | | EA=memaddr + (element&0x7)<<2<br>stvehx a, 0, EA<br>lhzx d, 0, EA |
| signed short | vector signed short | | EA=memaddr + (element&0x7)<<2<br>stvehx a, 0, EA<br>lhzx d, 0, EA<br>extsh d, d |
| unsigned int | vector unsigned int | | EA=memaddr + (element&0x3)<<3<br>stvewx a, 0, EA<br>lwzx a, 0, EA |
| signed int | vector signed int | | EA=memaddr + (element&0x3)<<3<br>stvewx a, 0, EA<br>lwzx a, 0, EA<br>extsw d, d [2] |
| float | vector float | | EA=memaddr + (element&0x3)<<3<br>stvewx a, 0, EA<br>lfsx a, 0, EA |

[1] memaddr is the address of a temporary memory location which is 16-byte aligned.

[2] The sign extend from word to doubleword can be omitted if the processor is running in 32-bit mode.

**vec_insert: Insert Scalar into Specified Vector Element**

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

Scalar $a$ is inserted into the element of vector $b$ that is specified by the $element$ parameter, and the modified vector is returned. All other elements of $b$ are unmodified. Depending on the size of the element, only a limited number of the least significant bits of the $element$ index are used. Specifically for 1-, 2-, and 4-byte elements, only four, three, and two of the least significant bits are used, respectively.

Table 7-197: Insert Scalar into Specified Vector Element

| Return/Argument Types | | | | Assembly Mapping[1] |
|---|---|---|---|---|
| d | a | b | element | |
| vector unsigned char | unsigned char | vector unsigned char | int | EA=memaddr + (element&0xF) stbx a, 0, EA lvebx d, 0, EA vperm d, d, a, pattern |
| vector signed char | signed char | vector signed char | | |
| vector unsigned short | unsigned short | vector unsigned short | | EA=memaddr + (element&0x7)<<2 sthx a, 0, EA lvehx d, 0, EA vperm d, d, a, pattern |
| vector signed short | signed short | vector signed short | | |
| vector unsigned int | unsigned int | vector unsigned int | | EA=memaddr + (element&0x3)<<3 stwx a, 0, EA lvewx d, 0, EA vperm d, d, a, pattern |
| vector signed int | signed int | vector signed int | | |
| vector float | float | vector float | | EA=memaddr + (element&0x3)<<3 stfsx a, EA lvewx d, 0, EA vperm d, d, a, pattern |

[1] memaddr is the address of a temporary memory location which is 16-byte aligned.

**vec_lvlx: Load Vector Left Indexed**

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

Let `EA` be the effective address formed from the sum of the contents of `a` and the contents of `b` and let `eb` be the value of the four least significant bits of `EA`. The (16 - `eb`) bytes addressed by `EA` are loaded into the leftmost (16 - `eb`) byte elements of `d` and the rightmost `eb` byte of `d` are set to zero.

Table 7-198: Load Vector Left Indexed

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| d | a | b | |
| vector unsigned char | any integral type | unsigned char * | lvlx d, a, b |
| | | vector unsigned char * | |
| vector signed char | any integral type | signed char * | |
| | | vector signed char * | |
| vector bool char | any integral type | vector bool char * | |
| vector unsigned short | any integral type | unsigned short * | |
| | | vector unsigned short * | |
| vector signed short | any integral type | signed short * | |
| | | vector signed short * | |
| vector bool short | any integral type | vector bool short * | |
| vector pixel | any integral type | vector pixel * | |
| vector unsigned int | any integral type | unsigned int * | |
| | | vector unsigned int * | |
| vector signed int | any integral type | signed int * | |
| | | vector signed int * | |
| vector bool int | any integral type | vector bool int * | |
| vector float | any integral type | float * | |
| | | vector float * | |

**vec_lvlxl: Load Vector Left Indexed Last**

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

Let EA be the effective address formed from the sum of the contents of $a$ and the contents of $b$ and let eb be the value of the four least significant bits of EA. The (16 - eb) bytes addressed by EA are loaded into the leftmost (16 - eb) bytes of $d$ and the rightmost eb bytes of $d$ are set to zero. vec_lvlxl provides a hint that the quadword in memory addressed by EA will probably not be needed again by the program in the near future.

Table 7-199: Load Vector Left Indexed Last

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| d | a | b | |
| vector unsigned char | any integral type | unsigned char * | lvlxl d, a, b |
| | | vector unsigned char * | |
| vector signed char | any integral type | signed char * | |
| | | vector signed char * | |
| vector bool char | any integral type | vector bool char * | |
| vector unsigned short | any integral type | unsigned short * | |
| | | vector unsigned short * | |
| vector signed short | any integral type | signed short * | |
| | | vector signed short * | |
| vector bool short | any integral type | vector bool short * | |
| vector pixel | any integral type | vector pixel * | |
| vector unsigned int | any integral type | unsigned int * | |
| | | vector unsigned int * | |
| vector signed int | any integral type | signed int * | |
| | | vector signed int * | |
| vector bool int | any integral type | vector bool int * | |
| vector float | any integral type | float * | |
| | | vector float * | |

**vec_lvrx: Load Vector Right Indexed**

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

Let `EA` be the effective address formed from the sum of the contents of `a` and the contents of `b` and let `eb` be the value of the four least significant bits of `EA`. If `eb` is not equal to zero (for example, `EA` is not quadword-aligned), then `eb` bytes in memory addressed by (`EA` - `eb`) are loaded into the rightmost `eb` bytes of `d` and the leftmost (16 - `eb`) bytes of `d` are set to zero. If `eb` is equal to zero (for example, `EA` is quadword-aligned), then the contents of `d` are set to zero.

Table 7-200: Load Vector Right Indexed

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| d | a | b | |
| vector unsigned char | any integral type | unsigned char * | lvrx d, a, b |
| | | vector unsigned char * | |
| vector signed char | any integral type | signed char * | |
| | | vector signed char * | |
| vector bool char | any integral type | vector bool char * | |
| vector unsigned short | any integral type | unsigned short * | |
| | | vector unsigned short * | |
| vector signed short | any integral type | signed short * | |
| | | vector signed short * | |
| vector bool short | any integral type | vector bool short * | |
| vector pixel | any integral type | vector pixel * | |
| vector unsigned int | any integral type | unsigned int * | |
| | | vector unsigned int * | |
| vector signed int | any integral type | signed int * | |
| | | vector signed int * | |
| vector bool int | any integral type | vector bool int * | |
| vector float | any integral type | float * | |
| | | vector float * | |

**vec_lvrxl: Load Vector Right Indexed Last**

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

Let `EA` be the effective address formed from the sum of the contents of `a` and the contents of `b` and let `eb` be the value of the four least significant bits of `EA`. If `eb` is not equal to zero (for example, `EA` is not quadword-aligned), then `eb` bytes in memory addressed by (`EA` - eb) are loaded into the rightmost `eb` bytes of `d` and the leftmost (16 - `eb`) bytes of `d` are set to zero. If `eb` is equal to zero (for example, `EA` is quadword-aligned), then the contents of `d` are set to zero. `vec_lvrxl` provides a hint that the quadword in memory addressed by `EA` will probably not be needed again by the program in the near future.

Table 7-201: Load Vector Right Indexed Last

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| d | a | b | |
| vector unsigned char | any integral type | unsigned char * | lvrxl d, a, b |
| | | vector unsigned char * | |
| vector signed char | any integral type | signed char * | |
| | | vector signed char * | |
| vector bool char | any integral type | vector bool char * | |
| vector unsigned short | any integral type | unsigned short * | |
| | | vector unsigned short * | |
| vector signed short | any integral type | signed short * | |
| | | vector signed short * | |
| vector bool short | any integral type | vector bool short * | |
| vector pixel | any integral type | vector pixel * | |
| vector unsigned int | any integral type | unsigned int * | |
| | | vector unsigned int * | |
| vector signed int | any integral type | signed int * | |
| | | vector signed int * | |
| vector bool int | any integral type | vector bool int * | |
| vector float | any integral type | float * | |
| | | vector float * | |

**vec_stvlx: Store Vector Left Indexed**

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

Let EA be the effective address formed from the sum of the contents of *b* and the contents of *c*, and let eb be the value of the four least significant bits of EA. Store the (16 - eb) leftmost bytes of *a* into the memory addressed by EA.

Table 7-202: Store Vector Left Indexed

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| a | b | c | |
| vector unsigned char | any integral type | unsigned char * | stvlx a, b, c |
| | | vector unsigned char * | |
| vector signed char | any integral type | signed char * | |
| | | vector signed char * | |
| vector bool char | any integral type | vector bool char * | |
| vector unsigned short | any integral type | unsigned short * | |
| | | vector unsigned short * | |
| vector signed short | any integral type | signed short * | |
| | | vector signed short * | |
| vector bool short | any integral type | vector bool short * | |
| vector pixel | any integral type | vector pixel * | |
| vector unsigned int | any integral type | unsigned int * | |
| | | vector unsigned int * | |
| vector signed int | any integral type | signed int * | |
| | | vector signed int * | |
| vector bool int | any integral type | vector bool int * | |
| vector float | any integral type | float * | |
| | | vector float * | |

**vec_stvlxl: Store Vector Left Indexed Last**

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

Let `EA` be the effective address formed from the sum of the contents of `b` and the contents of `c`, and let `eb` be the value of the four least significant bits of `EA`. Store the (16 - `eb`) leftmost bytes of `a` into the memory addressed by `EA`. `vec_stvlxl` provides a hint that the quadword in memory addressed by `EA` will probably not be needed again by the program in the near future.

Table 7-203: Store Vector Left Indexed Last

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| a | b | c | |
| vector unsigned char | any integral type | unsigned char * | stvlxl a, b, c |
| | | vector unsigned char * | |
| vector signed char | any integral type | signed char * | |
| | | vector signed char * | |
| vector bool char | any integral type | vector bool char * | |
| vector unsigned short | any integral type | unsigned short * | |
| | | vector unsigned short * | |
| vector signed short | any integral type | signed short * | |
| | | vector signed short * | |
| vector bool short | any integral type | vector bool short * | |
| vector pixel | any integral type | vector pixel * | |
| vector unsigned int | any integral type | unsigned int * | |
| | | vector unsigned int * | |
| vector signed int | any integral type | signed int * | |
| | | vector signed int * | |
| vector bool int | any integral type | vector bool int * | |
| vector float | any integral type | float * | |
| | | vector float * | |

**vec_stvrx: Store Vector Right Indexed**

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

Let `EA` be the effective address formed from the sum of the contents of *b* and the contents of *c*, and let `eb` be the value of the four least significant bits of `EA`. Store the `eb` rightmost bytes of *a* into the memory addressed by (`EA` - `eb`). If `eb` is zero, `EA` is 16-byte aligned, and no memory is stored.

Table 7-204: Store Vector Right Indexed

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| a | b | c | |
| vector unsigned char | any integral type | unsigned char * | stvrx a, b, c |
| | | vector unsigned char * | |
| vector signed char | any integral type | signed char * | |
| | | vector signed char * | |
| vector bool char | any integral type | vector bool char * | |
| vector unsigned short | any integral type | unsigned short * | |
| | | vector unsigned short * | |
| vector signed short | any integral type | signed short * | |
| | | vector signed short * | |
| vector bool short | any integral type | vector bool short * | |
| vector pixel | any integral type | vector pixel * | |
| vector unsigned int | any integral type | unsigned int * | |
| | | vector unsigned int * | |
| vector signed int | any integral type | signed int * | |
| | | vector signed int * | |
| vector bool int | any integral type | vector bool int * | |
| vector float | any integral type | float * | |
| | | vector float * | |

**vec_stvrxl: Store Vector Right Indexed Last**

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

Let $EA$ be the effective address formed from the sum of the contents of $b$ and the contents of $c$, and let $eb$ be the value of the four least significant bits of $EA$. Store the $eb$ rightmost bytes of $a$ into the memory addressed by ($EA$ - $eb$). If $eb$ is zero, $EA$ is 16-byte aligned, no memory is stored. vec_stvrxl provides a hint that the quadword in memory addressed by $EA$ will probably not be needed again by the program in the near future.

Table 7-205: Store Vector Right Indexed Last

| Return/Argument Types | | | Assembly Mapping |
|---|---|---|---|
| a | b | c | |
| vector unsigned char | any integral type | unsigned char * | stvrxl a, b, c |
| | | vector unsigned char * | |
| vector signed char | any integral type | signed char * | |
| | | vector signed char * | |
| vector bool char | any integral type | vector bool char * | |
| vector unsigned short | any integral type | unsigned short * | |
| | | vector unsigned short * | |
| vector signed short | any integral type | signed short * | |
| | | vector signed short * | |
| vector bool short | any integral type | vector bool short * | |
| vector pixel | any integral type | vector pixel * | |
| vector unsigned int | any integral type | unsigned int * | |
| | | vector unsigned int * | |
| vector signed int | any integral type | signed int * | |
| | | vector signed int * | |
| vector bool int | any integral type | vector bool int * | |
| vector float | any integral type | float * | |
| | | vector float * | |

**vec_promote: Promote Scalar to Vector**

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

Scalar `a` is promoted to a vector containing `a` in the element that is specified by the `element` parameter, and the result is returned in vector `d`. All other elements of `d` are undefined. Depending on the size of `a`, only a limited number of the least significant bits of the `element` index are used. Specifically for 1-, 2-, and 4-byte elements, only four, three, and two of the least significant bits are used, respectively.

Table 7-206: Promote Scalar to Vector

| Return/Argument Types | | | Assembly Mapping[1] |
|---|---|---|---|
| d | a | element | |
| vector unsigned char | unsigned char | | EA=memaddr + (element&0xF)<br>stbx a, 0, EA<br>lvebx d, 0, EA |
| vector signed char | signed char | | |
| vector unsigned short | unsigned short | | EA=memaddr + (element&0x7)<<2<br>sthx a, 0, EA<br>lvehx d, 0, EA |
| vector signed short | signed short | int | |
| vector unsigned int | unsigned int | | EA=memaddr + (element&0x3)<<3<br>stwx a, 0, EA<br>lvewx d, 0, EA |
| vector signed int | signed int | | |
| vector float | float | | EA=memaddr + (element&0x3)<<3<br>stfsx a, EA<br>lvewx d, 0, EA |

[1] memaddr is the address of a temporary memory location which is 16-byte aligned.

**vec_splats: Splat Scalar to Vector**

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

The single scalar `a` value is replicated across all elements of a vector of the same type and the result is returned in vector `d`.

Table 7-207: Splat Scalar to Vector

| Return/Argument Types | | Assembly Mapping |
|---|---|---|
| d | a | |
| vector unsigned char | unsigned char | |
| vector signed char | signed char | |
| vector unsigned short | unsigned short | store a into memory (EA) that 16-byte aligned |
| vector signed short | signed short | lvebx/lvehx/lvewx tmp, 0, EA |
| vector unsigned int | unsigned int | vspltb/vsplth/vspltw d, tmp, 0 |
| vector signed int | signed int | |
| vector float | float | |
| vector unsigned char | unsigned char (5-bit unsigned literal) | vspltisb d, a<br>  *or* |
| vector signed char | signed char (5-bit unsigned literal) | vspltish d, a<br>  *or* |
| vector unsigned short | unsigned short (5-bit unsigned literal) | vspltisw d, a<br>  *or* |
| vector signed short | signed short (5-bit unsigned literal) | vspltisw d, a |
| vector unsigned int | unsigned int (5-bit unsigned literal) | |
| vector signed int | signed int (5-bit unsigned literal) | |
| vector float | float (5-bit unsigned literal) | |

# 8. SPU C and C++ Standard Libraries and Language Support

This chapter describes differences between the implementations of the C and C++ standard libraries on the SPU and the corresponding ISO/IEC standards. It also identifies common language features that are specifically not supported on the SPU.

## 8.1. Standard Libraries

The C and C++ standard libraries that are required for the SPU are based on the Standard C Library described in ISO/IEC Standard 9899:1999 and the C++ Standard Library described in ISO/IEC Standard 14882:1998. However, neither library must be a fully compliant implementation of the respective ISO/IEC standard.

The proposed differences from ISO/IEC compliant implementations are due to two reasons: 1) The SPU does not have the same system resources and operating system support that are available to most stand-alone processors; and 2) the SPU hardware doesn't fully support the IEEE floating-point standard. Because of the SPU's limited operating system support, library functions that require system calls, thread facilities, and file input/output (I/0) may not be supported. Because of differences in floating-point behavior, the results of single-precision floating-point functions will probably be less accurate than defined by the Standard, and floating-point exceptions will be less reliable. Nevertheless, the standard library functions that are provided should execute fast, in most cases.

The minimum C and C++ library features that must be provided for the SPU are described in the following sections.

### 8.1.1. C Standard Library

This section describes the minimum requirements of a compliant C standard library implementation.

#### Library Contents

All of the entities required in the C standard library must be declared and defined within the library header files listed in Table 8-208. Differences between the contents of these header files and the header files that comprise the ISO Standard Library are identified in the table. For a detailed description of the particular entities, see the ISO/IEC C Standard listed in the "Related Documentation" section.

Table 8-208: C Library Header Files

| Header Name | Description |
|---|---|
| assert.h | Enforce assertions when functions execute. The `assert` macro reports assertion failures using the special debug `printf` (described later in this chapter). |
| complex.h | Perform complex arithmetic. |
| ctype.h | Classify characters. The functions declared in this header use only the "C" locale. |
| errno.h | Test error codes reported by library functions. |
| fenv.h | Control IEEE style floating-point arithmetic. Macros for single- and double-precision exceptions are described in "9.2.2. Floating-Point Exceptions". |
| float.h | Test floating-point type properties. These properties are specified in section "9.1. Properties of Floating-Point Data Type Representations". |
| inttypes.h | Convert various integer types. |
| iso646.h | Program in ISO 646 variant character sets. |
| limits.h | Test integer type properties. The macro `MB_LEN_MAX` is defined as 1. |
| locale.h | Not available. |
| math.h | Compute common mathematical functions. The floating-point behavior of these functions will adhere to the specifications described in section "9.3. Floating-Point Operations". Although not specified or required, corresponding vector versions of the math functions may be added to the library to take advantage of the many high-performance SIMD (single instruction, multiple data) instructions provided by the SPU hardware. |
| setjmp.h | Execute nonlocal goto statements. |

| Header Name | Description |
|---|---|
| signal.h | Not available. |
| stdarg.h | Access a varying number of arguments. |
| stdbool.h | Define a convenient Boolean type name and constants. |
| stddef.h | Define several useful types and macros. The `wchar_t` is not defined. |
| stdint.h | Define various integer types with size constraints. `SIG_ATOMIC_MAX` and `SIG_ATOMIC_MIN` are not defined, nor are any of the `WCHAR_MAX`, `WCHAR_MIN`, `WINT_MAX`, and `WINT_MIN`. |
| stdio.h | Not available, except for `printf`, which is provided for debugging. (See section "Debug printf()".) |
| stdlib.h | Perform a variety of operations. The functions `getenv`, `mblen`, `mbstowcs`, `mbtowc`, `system`, `wcstombs`, and `wctomb` are not defined. The type `wchar_t` and the macro `MB_CUR_MAX` are also not defined. |
| string.h | Manipulate several kinds of strings. The function `strxfrm` uses only the "C" locale. |
| tgmath.h | Declare various type-generic math functions. Single-precision functions declared in this header adhere to the same specifications described for the corresponding functions that are declared in `math.h`. |
| time.h | Not available. |
| wchar.h | Not available. |
| wctype.h | Not available. |

**Fastest Minimum-Width Integer Types**

The typedefs named `int_fast`*N*`_t` and `uint_fast`*N*`_t` designate the fastest signed and unsigned integer types with a width of at least *N*. These typedefs are defined as shown in Table 8-209. The size of these types is not guaranteed to be equal to the types defined for the PPU.

Table 8-209: Fastest Minimum-Width Integer Types

| Types | Size (in bits) |
|---|---|
| int_fast8_t/uint_fast8_t | 32 |
| int_fast16_t/uint_fast16_t | 32 |
| int_fast32_t/uint_fast32_t | 32 |
| int_fast64_t/uint_fast64_t | 64 |

**Debug printf()**

A `printf()` function will be provided for application debugging. The implementation of this function depends on the particular services provided by the underlying operating system. Although detailed specifications for this function are not mandated by this document, a full-featured implementation is recommended. Such an implementation would include all of the usual output format conversion specifiers required by the C standard. In addition, conversion specifiers of the type described in the *AltiVec™ Technology Programming Interface Manual* are recommended to handle vector output formatting. Output conversion specifiers take the following form:

```
%[<flags>][<width>][<precision>][<size>]<conversion>
```

where

```
<flags>           ::= <flag-char> | <flags><flag-char>

<flag-char>       ::= <std-flag-char> | <c-sep>

<std-flag-char>   ::= '-' | '+' | '0' | '#' | ' '

<c-sep>           ::= ',' | ';' | ':' | '_'

<width>           ::= <decimal-integer> | '*'

<precision>       ::= '.' <width> | '.' | '.*'
```

```
<size>                    ::= 'hh' | 'h' | 'l' | 'll' | 'L' | <vector-size>

<vector-size>             ::= 'v' | 'vhh' | 'vh' | 'vl' | 'vll' | 'vL' | 'hhv'
                              | 'hv' | 'lv'| 'llv' | 'Lv'

<conversion>              ::= <char-conv> | <str_conv> | <fp-conv> | <int-conv>
                              | <byte-conv> | <misc-conv>

<char-conv>               ::= 'c'

<str-conv>                ::= 's'

<fp-conv>                 ::= 'e' | 'E' | 'f' | 'F' | 'g' | 'G'

<int-conv>                ::= 'd' | 'i' | 'u' | 'p' | 'o' |'x' | 'X'

<byte-conv>               ::= 'uc' | 'co' | 'cx' | 'cX'

<misc-conv>               ::= 'n' | '%'
```

Extensions to the C standard output conversion specification are shown in bold for vector types. Vector types are formatted using the conversions shown in Table 8-210. String conversions (<str-conv>) and miscellaneous conversions (<misc-conv>) are not defined for vectors. The 'p' integer conversion (<int-conv>) is also not defined. The default separator (<c-sep>) is a space, except for character conversion (<char-conv>), which has no separator.

Table 8-210: Vector Formats

| Vector Size | Conversion | Description |
|---|---|---|
| v | <char-conv> | A vector is printed as a vector char, consisting of 16 one-byte elements. The 'c' conversion prints contiguous ASCII characters. |
| v | <int-conv> <byte-conv> | With the 'uc' conversion, a vector is printed as a vector unsigned char, consisting of 16 one-byte elements. Similarly, the 'co', 'cx', and 'cX' conversions print either a vector unsigned char or a qword, in octal format or in hexadecimal format. For all other integer conversions, a vector is printed in the respective octal (o), integer (d, i, u) or hexadecimal (x, X) format, either as a vector unsigned int or as a vector signed int, consisting of 4 four-byte elements. |
| v | <fp-conv> | A vector is printed in a signed decimal fractional representation, either in standard decimal notation (f or F) or with a decimal power-of-ten exponent (e, E, g, G). The representation is printed as a vector float, containing 4 four-byte elements. |
| vhh or hhv | <int-conv> | A vector is printed in the respective octal (o), integer (d, i, u), or hexadecimal (x, X) format, either as a vector unsigned char  or as a vector signed char, consisting of 16  one-byte elements. |
| vh or hv | <int-conv> | A vector is printed in the respective octal (o), integer (d, i, u), or hexadecimal (x, X) format, either as a vector unsigned short or as a vector signed short, consisting of 8 two-byte elements. |
| vl or lv | <int-conv> | A vector is printed in the respective octal (o), integer (d, i, u), or hexadecimal (x, X) format, as a vector unsigned int or as a vector signed int, consisting of 4 four-byte elements. |
| vll or llv | <int-conv> | A vector is printed in the respective octal (o), integer (d, i, u), or hexadecimal (x, X) format, as a vector unsigned long long or as a vector signed long long, consisting of 2 eight-byte elements. |
| vL or Lv | <fp-conv> | A vector is printed in a signed decimal fractional representation, either in standard decimal notation (f or F) or with a decimal power-of-ten exponent (e, E, g, G). The representation is printed as a vector double, consisting of 2 eight-byte elements. |

**Malloc Heap**

The malloc heap is defined to begin at _end and to extend to the end of the stack. The memory heap may be enlarged by a heap-extending function. This function would negatively adjust the Available Stack Size element of

the current Stack Pointer Information register and all Available Stack Sizes residing in the saved SP registers found in the sequence of Back Chain quadwords.

Whenever the `malloc` heap is enlarged, code should verify that the enlarged `malloc` heap does not extend into the currently used stack. If it does, the operation should fail.

Implementations of `setjmp`/`longjmp` are also affected by the use of heap-extending functions. When restoring the Stack Pointer Information register as a result of invoking the `longjmp` function, the function must detect any change to the Available Stack Size between `setjmp` and `longjmp`, and it must correct the saved Stack Pointer Information register. For example:

```
SP.avail_stack_size = SP_set.stack_ptr – SP.stack_ptr +
        SP.avail_stack_size;
```

where `SP` is the current Stack Pointer Information register, and `SP_set` is the Stack Pointer Information register saved at the last `setjmp` call.

### 8.1.2. C++ Standard Library

This section describes the minimum contents of the C++ standard library.

As with the C library, the C++ library header files declare or define the contents of the C++ library. Table 8-211 lists the header files that comprise the core of the C++ standard library. Differences between the contents of the C++ header files and the header files that comprise the ISO Standard Library are noted in this table.

Table 8-211: C++ Library Header Files

| Header Name | Description |
|---|---|
| algorithm | Define numerous templates that implement useful algorithms. |
| bitset | Define a template class that administers sets of bits. |
| complex | Define a template class that supports complex arithmetic. |
| deque | Define a template class that implements a deque container. |
| exception | Not available. |
| fstream | Not available. |
| functional | Define several templates that help construct predicates for the templates defined in algorithm and numeric. |
| iomanip | Not available. |
| ios | Not available. |
| iosfwd | Not available. |
| iostream | Not available. |
| istream | Not available. |
| iterator | Define several templates that help define and manipulate iterators. |
| limits | Test numeric type properties. |
| list | Define a template class that implements a doubly linked list container. |
| locale | Not available. |
| map | Define template classes that implement associative containers that map keys to values. |
| memory | Define several templates that allocate and free storage for various container classes. |
| new | Declare several functions that allocate and free storage. |
| numeric | Define several templates that implement useful numeric functions. |
| ostream | Not available. |
| queue | Define a template class that implements a queue container. |
| set | Define template classes that implement associative containers. |
| slist | Define a template class that implements a singly linked list container. |

| Header Name | Description |
|---|---|
| sstream | Not available. |
| stack | Define a template class that implements a stack container. |
| stdexcept | Not available. |
| streambuf | Not available. |
| string | Define a template class that implements a string container. |
| strstream | Not available. |
| typeinfo | Not available. |
| utility | Define several templates of general utility. |
| valarray | Define several classes and template classes that support value-oriented arrays. |
| vector | Define a template class that implements a vector container. |

The C++ standard library contains new-style C++ header files that correspond to 12 traditional C header files. Both the new-style and the traditional-style header files are included in the library. These header files are listed in Table 8-212.

Table 8-212: New and Traditional C++ Library Header Files

| New-Style Header Name | Traditional Header Name | Description |
|---|---|---|
| cassert | assert.h | Enforce assertions when functions execute.[1] |
| cctype | ctype.h | Classify characters.[1] |
| cerrno | errno.h | Test error codes reported by library functions.[1] |
| cfloat | float.h | Test floating-point type properties. |
| ciso646 | iso646.h | Program in ISO 646 variant character sets. |
| climits | limits.h | Test integer type properties.[1] |
| clocale | locale.h | Not available. |
| cmath | math.h | Compute common mathematical functions.[1] |
| csetjmp | setjmp.h | Execute nonlocal goto statements. |
| csignal | signal.h | Not available. |
| cstdarg | stdarg.h | Access a varying number of arguments. |
| cstddef | stddef.h | Define several useful types and macros.[1] |
| cstdio | stdio.h | Not available. |
| cstdlib | stdlib.h | Perform a variety of operations.[1] |
| cstring | string.h | Manipulate several kinds of strings.[1] |
| ctime | time.h | Not available. |
| cwchar | wchar.h | Not available. |
| cwctype | wctype.h | Not available. |

[1] See Table 8-208: C Library Header Files, for specific implementation limitations.

## 8.2. Non-Supported Language Features

C and C++ implementations should comply with the language features prescribed in the respective ISO/IEC standards, as much as possible. However, certain features are specifically not supported because of SPU architecture limitations. Currently, the only non-supported feature is C++ exception handling.

# 9. Floating-Point Arithmetic on the SPU

Annex F of the C99 language standard (ISO/IEC 9899) specifies support for the IEC 60559 floating-point standard. This chapter describes differences from Annex F and ISO/IEC Standard 60559 that apply to SPU compilers and libraries.

Floating-point behavior is essentially dictated by the SPU hardware. For single precision, the hardware provides an extended single-precision number range. Denorm arguments are treated as 0, and NaN (not-a-number) and Infinity are not supported. The only rounding mode that is supported is truncation (round towards 0), and exceptions apply only to certain extended range floating-point instructions). For double precision, the hardware provides the standard IEEE number range, but again, denorm arguments are treated as 0. IEEE exceptions are detected and accumulated in the FPSCR register, and the IEEE rules for propagation of NaNs are not implemented in the architecture. (For details, see the *Synergistic Processor Unit Instruction Set Architecture*.)  These and other IEEE differences affect almost every aspect of floating-point computation, including data-type properties, rounding modes, exception status, error reporting, and expression evaluation. The particular effect of these differences on the compiler and libraries are described in the following sections.

## 9.1. Properties of Floating-Point Data Type Representations

The properties of floating-point data type representations are declared as macros in float.h. Table 9-213 lists these macros and the corresponding values that are applicable for the SPU.

Table 9-213: Values for Floating-Point Type Properties

| Macro | Value |
|---|---|
| FLT_DIG | 6 |
| FLT_EPSILON | 0x1p-23f (1.19209290E-07f) |
| FLT_MANT_DIG | 24 |
| FLT_MAX_10_EXP | 38 |
| FLT_MAX_EXP | 129 |
| FLT_MIN_10_EXP | -37 |
| FLT_MIN_EXP | -125 |
| FLT_MAX | 0x1.FFFFFEp128f (6.80564694E+38f) |
| FLT_MIN | 0x1p-126f (1.17549436E-38f) |
| FLT_ROUNDS | Initialized to 16 (to nearest for both elements) |
| FLT_EVAL_METHOD | 0 (no promotions occur) |
| FLT_RADIX | 2 |
| DBL_DIG | 15 |
| DBL_EPSILON | 0x1p-52 (2.2204460492503131E-016) |
| DBL_MANT_DIG | 53 |
| DBL_MAX_10_EXP | 308 |
| DBL_MAX_EXP | 1024 |
| DBL_MIN_10_EXP | -307 |
| DBL_MIN_EXP | -1021 |
| DBL_MAX | 0x1.FFFFFFFFFFFFFp1023 (1.7976931348623157E+308) |
| DBL_MIN | 0x1p-1022 (2.2250738585072014E-308) |
| DECIMAL_DIG | 17 |

## 9.2. Floating-Point Environment

The macros defined within `fenv.h` control the directed-rounding control mode and floating-point exception status flags for floating-point operations.

### 9.2.1. Rounding Modes

Whereas the C language specification requires that all floating-point data types use the same rounding modes, the SPU hardware supports different rounding modes for single- and double-precision arithmetic. On the SPU, the rounding mode for single precision is round-towards-zero, and the default rounding mode for double precision is round-to-nearest.

According to the C99 standard, the rounding mode for floating-point addition is characterized by the implementation-defined value of `FLT_ROUNDS`. On the SPU, this macro is only used for double precision. Single-precision rounding mode is always truncation. (See Table 9-213.)

`FLT_ROUNDS` will return a 5-bit value which represents the rounding mode for both double precision elements. The highest bit is always 1. The next two bits are the rounding mode for element 0 and the two lowest bits are the rounding mode for element 1. Table 9-214 lists the rounding mode represented by the two bits for each element.

Table 9-214: Rounding Mode for Two Bits of FLT_ROUNDS

| Last Two Bits | Rounding Mode |
|---------------|---------------|
| 00 | Round to nearest even |
| 01 | Round toward zero (truncate) |
| 10 | Round toward +infinity |
| 11 | Round towards -infinity |

Because the SPU hardware only supports rounding towards zero for single precision, some single-precision math functions will necessarily deviate from the C99 standard. The standard library math functions and macros that deviate are described later, in section "9.3.2. Overall Behavior of C Operators and Standard Library Math Functions".

Table 9-215 lists the macros that can be used to set the double precision rounding modes for element 0 and element 1. The macros for element 0 and element 1 may be used together with a bitwise OR to set the rounding mode for both elements, or  the macros can be used separately to set the rounding mode for only that element.

Table 9-215: Macros for Double Precision Rounding Modes

| Macro | Comment |
|-------|---------|
| FE_TONEAREST | Set element 0 to round to nearest even |
| FE_TOWARDZERO | Set element 0 to round towards zero |
| FE_UPWARD | Set element 0 to round towards +infinity |
| FE_DOWNWARD | Set element 0 to round towards –infinity |
| FE_TONEAREST_1 | Set element 1 to round to nearest even |
| FE_TOWARDZERO_1 | Set element 1 to round towards zero |
| FE_UPWARD_1 | Set element 1 to round towards +infinity |
| FE_DOWNWARD_1 | Set element 1 to round towards -infinity |

### 9.2.2. Floating-Point Exceptions

Table 9-216 andTable 9-217 list the macros for floating-point exceptions that will be defined in fenv.h. Because of the restricted behavior of the SPU floating-point hardware, single-precision library functions can have an undefined effect on these exception flags. Moreover, hardware traps will not result from any raised exception.

Table 9-216: Macros for Single Precision Floating-Point Exceptions

| Macro | Comment |
|---|---|
| FE_OVERFLOW_SNGL | Overflow exception for element 0 |
| FE_UNDERFLOW_SNGL | Underflow exception for element 0 |
| FE_DIFF_SNGL | Different from IEEE exception for element 0 |
| FE_DIVBYZERO_SNGL | Divide by zero exception for element 0 |
| FE_OVERFLOW_SNGL_1 | Overflow exception for element 1 |
| FE_UNDERFLOW_SNGL_1 | Underflow exception for element 1 |
| FE_DIFF_SNGL_1 | Different from IEEE exception for element 1 |
| FE_DIVBYZERO_SNGL_1 | Divide by zero exception for element 1 |
| FE_OVERFLOW_SNGL_2 | Overflow exception for element 2 |
| FE_UNDERFLOW_SNGL_2 | Underflow exception for element 2 |
| FE_DIFF_SNGL_2 | Different from IEEE exception for element 2 |
| FE_DIVBYZERO_SNGL_2 | Divide by zero exception for element 2 |
| FE_OVERFLOW_SNGL_3 | Overflow exception for element 3 |
| FE_UNDERFLOW_SNGL_3 | Underflow exception for element 3 |
| FE_DIFF_SNGL_3 | Different from IEEE exception for element 3 |
| FE_DIVBYZERO_SNGL_3 | Divide by zero exception for element 3 |
| FE_ALL_EXCEPT_SNGL | Bitwise OR of all macros for element 0 |
| FE_ALL_EXCEPT_SNGL_1 | Bitwise OR of all macros for element 1 |
| FE_ALL_EXCEPT_SNGL_2 | Bitwise OR of all macros for element 2 |
| FE_ALL_EXCEPT_SNGL_3 | Bitwise OR of all macros for element 3 |

Table 9-217: Macros for Double Precision Floating-Point Exceptions

| Macro | Comment |
|---|---|
| FE_OVERFLOW_DBL | Overflow exception for element 0 |
| FE_UNDERFLOW_DBL | Underflow exception for element 0 |
| FE_INEXACT_DBL | ISO/IEC inexact for element 0 |
| FE_INVALID_DBL | ISO/IEC invalid for element 0 |
| FE_NC_NAN_DBL | Possibly non-compliant NaN for element 0 |
| FE_NC_DENORM_DBL | Possibly non-compliant denormal for element 0 |
| FE_OVERFLOW_DBL_1 | Overflow exception for element 1 |
| FE_UNDERFLOW_DBL_1 | Underflow exception for element 1 |
| FE_INEXACT_DBL_1 | ISO/IEC inexact for element 1 |
| FE_INVALID_DBL_1 | ISO/IEC invalid for element 1 |
| FE_NC_NAN_DBL_1 | Possibly non-compliant NaN for element 1 |
| FE_NC_DENORM_DBL_1 | Possibly non-compliant denormal for element 1 |
| FE_ALL_EXCEPT_DBL | Bitwise OR of all macros for element 0 |
| FE_ALL_EXCEPT_DBL_1 | Bitwise OR of all macros for element 1 |
| FE_ALL_EXCEPT | Bitwise OR of all macros from this table |

The floating-point environment variables defined in the C99 specification only apply to double-precision.

The pragma `FENV_ACCESS` will be used to inform the compiler whether the program intends to control and test floating-point status. If the pragma is on, the compiler will take appropriate action to ensure that code transformations preserve the behavior specified in this document.

### 9.2.3. Other Floating-Point Constants in math.h

Several additional floating-point constants are defined in `math.h`. These constants are used by functions to report various domain and range errors. Many have a non-standard definition for the SPU. A description of these particular constants is shown in Table 9-218.

Table 9-218: Floating-Point Constants

| Macro | Description |
|---|---|
| HUGE_VAL | Infinity |
| HUGE_VALF | `FLT_MAX` |
| HUGE_VALL | Infinity |
| INFINITY<br>NAN | Double precision adheres to the IEEE definition. These macros are not used for single-precision operations. |
| FP_INFINITE<br>FP_NAN<br>FP_NORMAL<br>FP_SUBNORMAL<br>FP_ZERO | For single precision, the `fpclassify()` function will only return `FP_NORMAL` and `FP_ZERO` classes; `FP_NAN`, `FP_INFINITE`, and `FP_SUBNORMAL` are never generated. |
| FP_FAST_FMA<br>FP_FAST_FMAF<br>FP_FAST_FMAL | These are defined to indicate that the `fma` function executes more quickly than a multiply and an add of float and double operands. |
| FP_ILOGB0<br>FP_ILOGBNAN | `FP_ILOGB0` is the value returned by `ilogb(x)` and `ilogbf(x)` if `x` is zero or a denorm number. Its value is `INT_MIN`.<br><br>`FP_ILOGBNAN` is the value returned by `ilogb(x)` if `x` is a `NaN`. This does not apply to the single-precision case of `ilogbf`. Its value is `INT_MAX`. |
| MATH_ERRNO<br>MATH_ERREXCEPT | These will expand to the integer constants 1 and 2, respectively. |
| math_errhandling | Expands to an expression that has type `int` and the value `MATH_ERRNO`, `MATH_ERREXCEPT`, or the bitwise `OR` of both. The value of `math_errhandling` is constant for the duration of a program. |

## 9.3. Floating-Point Operations

This section specifies floating-point data conversions, and it describes the overall behavior of C operators and standard library functions. It also describes several special cases where floating-point results might vary from the IEEE standard. Lastly, the section describes the specific behavior of several specific math functions.

### 9.3.1. Floating-Point Conversions

This section provides specifications for the four types of floating-point data conversions: 1) conversions from integers
to floating-point; 2) conversions from floating-point to integer; 3) conversion between floating-point precisions; and, 4) conversions between floating-point and string.

#### Integer to Floating-Point Conversions

Conversions from integers to floats will adhere to the following rules:

- A single-precision conversion from integer to float produces a result within the extended single-precision floating-point range. See Table 9-213 for details about this range.
- A single-precision conversion from integer to float rounds towards zero.
- A double-precision conversion from integer to float produces a result within the C99 standard double-precision floating-point range.
- A double-precision conversion from integer to float rounds according to the rounding mode indicated by the value of `FLT_ROUNDS`.

**Floating-Point to Integer Conversions**

Conversions from floats to integers will have the following behavior:

- When converting from a float to an integer, exceptions are raised for overflow, underflow, and IEEE non-compliant result.

- Overflow and underflow exceptions are raised when converting from a double to an integer. If a double-precision value is infinite or NaN or if the integral part of the floating value exceeds the range of the integer type, an "invalid" floating-point exception is raised, and the resulting value is unspecified. An "inexact" floating-point exception is raised by the hardware when a conversion involves an integral floating-point value that is outside the range of the integer data type.

**Conversions between Floating-Point Precision**

To achieve maximum performance, compilers only perform conversion from float to double and from double to float within the IEEE standard range. These conversions will comply with the IEEE standard, except for denormal inputs, which are forced to zero. Conversion of numbers outside of the IEEE standard range is unspecified. Conversions with NaNs, infinities, or denormal results are also unspecified.

**Conversions between Floating-Point and Strings**

Conversions between floating-point and string values will adhere to both the extended single-precision floating-point range and the IEEE standard double-precision floating-point range.

**9.3.2. Overall Behavior of C Operators and Standard Library Math Functions**

Library functions and compilers will obey the same general rules with respect to rounding and overflow. These rules differ, however, depending on whether the code is single precision or double precision.

**Single-Precision Code**

For single precision, the C operators (+, -, *, and /) and the standard library math functions will have the following behavior:

- If the operation produces a value with a magnitude greater than the largest positive representable extended-precision number, the result will be FLT_MAX with appropriate sign, and the overflow flag will be raised.

- For all operators and standard functions, except the negate operator and the fabsf() and copysignf() functions, an argument with a denormal value will be treated as +0.0.

- Except for the negate operator and the fabsf() and copysignf() functions, operators and standard functions will never return a denormal value or -0.0.

- The negate operator and the fabsf() and copysignf() functions must be implemented such that only the sign bit is changed.

- Expressions will be evaluated using the round-towards-zero mode. Implementations that depend on other rounding directions for algorithm correctness will produce incorrect results and therefore cannot be used.

- The overflow flag will be set when FLT_MAX is returned instead of a value whose magnitude is too large. Because infinity is undefined for single precision, FLT_MAX will be used to signal infinity in situations where infinity would otherwise be generated on an IEEE754-compliant system. This modification will enable common trig identities to work.

- NaN is not supported and does not need to be copied from any input parameter.

- By default, compilers may perform optimizations for single-precision floating-point arithmetic that assume 1) that NaNs are never given as arguments; and, 2) that ±Inf will never be generated as a result.

- Compilers can assume that floating-point operations will not generate user-visible traps, such as division by zero, overflow, and underflow.

- Constant expressions that are evaluated at compile time will produce the same result as they would if they were evaluated at runtime. For example,

```
float x = 6.0e38f * 8.1e30f;
```

will be evaluated as `FLT_MAX`.

- Compilers may use single-precision contracted operations, such as Floating Reciprocal Absolute Square Root Estimate (`frsqest`) or Floating Multiply and Add (`fma`), unless explicitly prohibited by `FP_CONTRACT` pragma or a *no-fast-float* compiler option. When contracted operations are used, `errno` does not need to be set.

**Double-Precision Code**

For double-precision floating-point, the C operators and standard library math functions will be compliant with the IEEE standard, with the following exceptions:

- When a `NaN` is produced as a result of an operation, it will always be a `QNaN`.

- Except for the negate operator and the `fabs()` and `copysign()` functions, denormal values will only be supported as results. A denormal operand is treated as 0 with same sign as the denormal operand.

- The default rounding mode for double precision is rounding to nearest.

- Compilers may use double precision contracted operations, such as Double Floating Multiply and Add (`dfma`), unless explicitly prohibited by the `FP_CONTRACT` pragma or a *no-fast-double* compiler option. When contracted operations are used, `errno` does not need to be set.

### 9.3.3. Floating-Point Expression Special Cases

The C99 standard describes several standard expression transformations that might fail to produce the required effect on the SPU:

- `x/2 -> x*0.5`

  Valid for this particular value because the value is an exact power of 2, but it is invalid in general (for example, `x/10 != x*0.1`) because the floating-point constant is not exactly representable in any finite base-2 floating-point system.

- `x*1 -> x` and `x/1 -> x`

  Invalid when: 1) x is a `SNaN` or a non-default `QNaN` (double precision only); 2) x is a denormal number; or, 3) x is `-0.0` (single precision only).

- `x/x -> 1.0`

  Invalid for single precision when x is zero or a denormal, and invalid for double precision when x is zero, or a denormal, `Inf`, or `NaN`.

- `x-y -> -(y-x)`

  Invalid for zero results which might have different signs, or, for double precision, round to +/- infinity, nonzero results might differ by 1 ULP.

- `x-x -> 0.0`

  Always valid for single precision, but the equivalence is invalid for double precision when x is either `NaN` or `Inf`. It is also invalid for double precision for round to –infinity, in which case the result will be `-0.0`.

- `0*x -> 0.0`

  Always valid for single precision, but invalid for double precision when x is a `NaN`, `Inf`, negative number, or `-0`.

- `x+0 -> x`

  Invalid in single precision, if x is a denormal operand or `-0`. Invalid in double precision if x=`-0` under round-to-nearest, round to +infinity and truncate. Also invalid in double precision if x is a `SNaN` or non-default `QNaN` and if x is a denormal number, in which case `x+0` becomes a zero with appropriate sign.

- `x-0 -> x`

Valid for single precision, except if `x` is a denormal operand or `-0`. Invalid for double precision if `x` is an SNaN or non-default QNaN, if `x` is a denormal number, or if `x` is `+0` and rounding mode is rounding to –infinity. In this last case, `x-0 = +0-0 =-0`. For any normalized operand the result is valid even with round to –infinity.

- `-x -> 0-x`

  Invalid for single precision when `x` is `+0.0` or a denormal. Invalid for double precision in the following cases: 1) For NaNs the value of `-x` is undefined; the result will be different for all NaNs. 2) If `x` is `+0` and the rounding mode is rounding to nearest-even, +infinity, or truncation, `0-x = +0` and `-x = -0`.

- `x!=x -> false`

  Always valid for single precision. For double precision, `x=NaN` always compares unordered, so `x!=x -> true`.

- `x==x -> true`

  Always valid for single precision. For double precision, `x=NaN` always compares unordered, so `x==x -> false`.

- `x<y -> isless(x,y)`,
  `x<=y -> islessequal(x,y)`,
  `x>y -> isgreater(x,y)`, and
  `x>=y -> isgreaterequal(x,y)`

  Valid. Exceptions are due to flags that are set as side effects when `x` or `y` are NaN under double precision. The `FENV_ACCESS` pragma can change the invalid flag behavior.

### 9.3.4. Specific Behavior of Standard Math Functions

This section describes the specific behavior of various floating-point functions declared in `math.h`. As noted, the SPU hardware has a direct effect on the behavior of floating-point functions. Because of the many differences between strict IEEE behavior and the hardware behavior, the standard math functions do not need to provide rigorous checks for exception situations and out-of-range conditions. Consequently, the results of many functions are redefined. The following is a list of differences:

- The function `nanf()` will return zero.

- The `isfinite()` macro will always return a nonzero value for single precision.

- The `isinf()` macro will always return zero for single precision.

- The `isnan()` macro will always return zero for single precision.

- Unlike the C99 standard specifications, the single-precision functions `nearbyintf()`, `lrintf()`, `llrintf()`, and `fmaf()` round towards zero.

- Trig, hyperbolic, exponential, logarithmic, and gamma functions do not need to set the inexact flag when values are rounded.

- The boundary cases with a NaN argument will not be supported for single precision because NaN is not a valid argument.

- `nextafterf(subnormal,y)` will never raise an underflow flag. The functions `nextafterf()` and `nexttowardf()` will succeed when incrementing past the IEEE maximal float value.

- The following boundary cases will not be supported for single precision because infinity is not a valid argument: `atanf(±inf)`, `atan2f(±y, ±inf)`, `atan2f(±inf,x)`, `atan2f(±inf,±inf)`, `acoshf(+inf)`, `asinhf(±inf)`, `atanhf(±1)`, `atanhf(±inf)`, `coshf(±inf)`, `sinhf(±inf)`, `tanhf(±inf)`, `expf(±inf)`, `exp2f(±inf)`, `expm1f(±inf)`, `frexpf(±inf,&exp)`, `ldexpf(±inf,exp)`, `logf(+inf)`, `log10f(+inf)`, `log1pf(+inf)`, `log2f(+inf)`, `logbf(±inf)`, `modff(±inf,iptr)`, `scalbnf(±inf,n)`, `cbrtf(±inf)`, `fabsf(±inf)`, `hypotf(±inf,y)`, `powf(-1,±inf)`, `powf(x,±inf)`, `powf(±inf,y)`, `sqrtf(±inf)`, `erff(±inf)`, `erfcf(±inf)`, `lgammaf(±inf)`, `tgammaf(+inf)`, `ceilf(±inf)`, `floorf(±inf)`, `nearbyintf(±inf)`, `roundf(±inf)`, `rintf(±inf)`, `lrintf(±inf)`, `llrintf(±inf)`, `lroundf(±inf)`, `llroundf(±inf)`, `truncf(±inf)`, `fmodf(x,±inf)`, `remainderf(±inf)`, `remquof(±inf)`, and `copysignf(±inf)`.

- For single precision, the following boundary cases will produce a non-IEEE-compliant result: `acosf(|x|>1)`, `asinf(|x|>1)`, `acoshf(x<1.0)`, `atanhf(|x|>1)`, `tgammaf(x<0)`, `fmodf(x,0)`, `ldexpf(x,BIG_INT)`, `logf(±0)`, `logf(x<0)`, `log10f(±0)`, `log10f(x<0)`, `log1pf(-1)`, `log1pf(x<-1)`, `log2f(±0)`, `log2f(x<0)`, `logbf(±0)`, `powf(±0,y)`, and `tgammaf(±0)`

- For single precision, the following boundary cases will not return `NaN`: `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)`.

- Section "9.3.1. Floating-Point Conversions" describes the behavior of implicit conversions when a single precision value is passed as an argument to a double precision function or when a single precision variable is assigned the result of a double-precision function.

# 10. Operator Overloading for Vector Data Types

Operator overloading is a syntactic feature in which common operators, such '+' or '-', have different implementations depending upon the type of their arguments. This section describes the vector data types that may be used with certain standard C/C++ operators and the behavior of these operators.

## 10.1. Supported Types

Operator overloading is valid on the vector data types listed in Table 10-219 and Table 10-220.

Table 10-219: Integer Vector Types

| Type | SPU/PPU |
|------|---------|
| vector signed char | Both |
| vector unsigned char | Both |
| vector signed short | Both |
| vector unsigned short | Both |
| vector signed int | Both |
| vector unsigned int | Both |
| vector signed long long | SPU |
| vector unsigned long long | SPU |

Table 10-220: Floating-Point Vector Types

| Type | SPU/PPU |
|------|---------|
| vector float | Both |
| vector double | SPU |

## 10.2. Vector Subscripting

Given `E1[E2]`, where `E1` has a vector type with base type `T` and `E2` has an integer type, the result is equivalent to:

```
(((T *)&(E1))[E2])
```

When the value of `E2` does not designate a valid element of `E1`, the behavior is undefined.

## 10.3. Unary Operators

Given `OP E1`, where `E1` is a vector type `T` with `N` elements and `OP` is one of the operators in Table 10-221, the result has a value equivalent to:

```
(T){ OP E1[0], ..., OP E1[N-1] }
```

Table 10-221: Valid Types for Specified Unary Operators

| OP | Integer Vector Types | Floating-Point Vector Types |
|----|---------------------|----------------------------|
| ++ | yes | yes |
| -- | yes | yes |
| + | yes | yes |
| - | yes | yes |
| ~ | yes | no |

## 10.4. Binary Operators

Given `E1 OP E2`, where `E1` and `E2` have equivalent vector types `T` with `N` elements and `OP` is one of the operators in Table 10-222, the result has a value equivalent to:

```
(T){ E1[0] OP E2[0], ..., E1[N-1] OP E2[N-1] }
```

For the assignment operators, `E1` shall be a modifiable lvalue, and the result value will be assigned to the object it designates.

Table 10-222: Valid Types for Specified Binary Operators

| OP | Integer Vector Types | Floating-Point Vector Types |
|---|---|---|
| + += | yes | yes |
| – –= | yes | yes |
| * *= | yes | yes |
| / /= | yes | yes |
| % %= | yes | no |
| & &= | yes | no |
| \| \|= | yes | no |
| ^ ^= | yes | no |
| << <<= | yes | no |
| >> >>= | yes | no |

## 10.5. Relational Operators

Given `E1 OP E2`, where `E1` and `E2` have equivalent vector types `T` with `N` elements and `OP` is one of the operators in Table 10-223, the result has a value equivalent to:

```
((E1[0] OP E2[0]) & ... & (E1[N-1] OP E2[N-1]))
```

Table 10-223: Valid Types for Specified Relational Operators

| OP | Integer Vector Types | Floating-Point Vector Types |
|---|---|---|
| == | yes | yes |
| != | yes | yes |
| < | yes | yes |
| > | yes | yes |
| <= | yes | yes |
| >= | yes | yes |

**IBM**

# Index

**IBM**

**End of Document**