XL C/C++ for Multicore Acceleration for Linux, V9.0
and XL Fortran for Multicore Acceleration for Linux,
V11.1

April 2008 PTF for XL C/C++ for Multicore Acceleration for Linux, V9.0 and XL Fortran for Multicore Acceleration for Linux, V11.1

XL C/C++ for Multicore Acceleration for Linux, V9.0
and XL Fortran for Multicore Acceleration for Linux,
V11.1

April 2008 PTF for XL C/C++ for Multicore Acceleration for Linux, V9.0 and XL Fortran for Multicore Acceleration for Linux, V11.1

Note

Before using this information and the product it supports, read the information in “Notices” on page 33.

First edition

This edition applies to the following:

- April 2008 PTF for IBM XL C/C++ for Multicore Acceleration for Linux on x86, V9.0
- April 2008 PTF for IBM XL C/C++ for Multicore Acceleration for Linux on System p™, V9.0
- April 2008 PTF for IBM XL Fortran for Multicore Acceleration for Linux on System p, V9.0

These PTFs update XL C/C++ for Multicore Acceleration for Linux, V9.0 and XL Fortran for Multicore Acceleration for Linux, V11.1 (Program numbers 5724-T42, 5724-T43 & 5724-T44).

Contents

About this information

This document provides information on the new features added to XL C/C++ for Multicore Acceleration for Linux, V9.0 and XL Fortran for Multicore Acceleration for Linux, V11.1 with the April 2008 PTF for those products.

Who should read this document

This document is intended for developers who want information on the April 2008 PTF for XL C/C++ for Multicore Acceleration for Linux, V9.0 and XL Fortran for Multicore Acceleration for Linux, V11.1. It assumes that you have some familiarity with command-line compilers and a basic knowledge of operating system commands, the C and Fortran programming languages, and the Cell Broadband Engine™ architecture.

How to use this document

The information in this document is divided into two sections: Chapter 1, “Effective address space support (SPU only),” on page 1 and Chapter 2, “Fortran intrinsics (SPU only),” on page 9. The section on effective address space support applies only to XL C/C++ for Multicore Acceleration for Linux, V9.0 and the section on Fortran intrinsics applies only to XL Fortran for Multicore Acceleration for Linux, V11.1. Both sections only apply to compilations which target the Synergistic Processor Unit (SPU).

This document describes only new features introduced in the April 2008 PTF for XL C/C++ for Multicore Acceleration for Linux, V9.0 and XL Fortran for Multicore Acceleration for Linux, V11.1. For information on how to install the updates described in this document, see the *IBM XL C/C++ for Multicore Acceleration for Linux, V9.0 Installation Guide* and *IBM XL Fortran for Multicore Acceleration for Linux, V11.1 Installation Guide*

For more comprehensive information on the compilers, consult the references given in “Related information” on page ix.

Conventions and terminology

Typographical conventions

The following table explains the typographical conventions used in the XL C/C++ for Multicore Acceleration for Linux, V9.0 and XL Fortran for Multicore Acceleration for Linux, V11.1 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, **ppuxlc** and **ppuxlC** (**ppuxlc++**), along with several other compiler invocation commands to support various C/C++ language levels and compilation environments.

Table 1. Typographical conventions (continued)

Typeface	Indicates	Example
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
<u>underlining</u>	The default setting of a parameter of a compiler option or directive.	nomaf | <u>maf</u>
`monospace`	Programming keywords and library functions, compiler builtins, examples of program code, command strings, or user-defined names.	To compile and optimize myprogram.f, enter: ppuxlf myprogram.f -O3.
UPPERCASE bold	Fortran programming keywords, statements, directives, and intrinsic procedures.	The ASSERT directive applies only to the DO loop immediately following the directive, and not to any nested DO loops.

Syntax diagrams

Throughout this information, diagrams illustrate XL C/C++ and XL Fortran syntax. This section will help you to interpret and use those diagrams.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►── symbol indicates the beginning of a command, directive, or statement.

  The ───► symbol indicates that the command, directive, or statement syntax is continued on the next line.

  The ►── symbol indicates that a command, directive, or statement is continued from the previous line.

  The ───►◄ symbol indicates the end of a command, directive, or statement.

  Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the │─── symbol and end with the ───│ symbol.

  IBM® XL Fortran extensions are marked by a number in the syntax diagram with an explanatory note immediately following the diagram.

  Program units, procedures, constructs, interface blocks and derived-type definitions consist of several individual statements. For such items, a box encloses the syntax representation, and individual syntax diagrams show the required order for the equivalent Fortran statements.
- Required items are shown on the horizontal line (the main path):

```
►►──keyword──required_argument──────────────────────────────────────────►◄
```

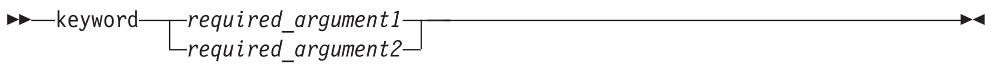
- Optional items are shown below the main path:

```
►►──keyword──┬───────────────────┬──────────────────────────────────────►◄
             └─optional_argument─┘
```

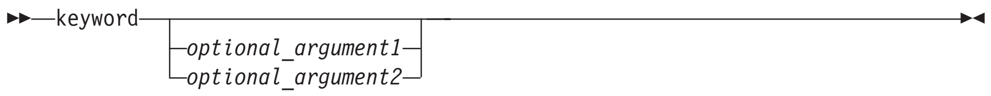
  Note: Optional items (not in syntax diagrams) are enclosed by square brackets ([and]). For example, `[UNIT=]u`

- If you can choose from two or more items, they are shown vertically, in a stack.

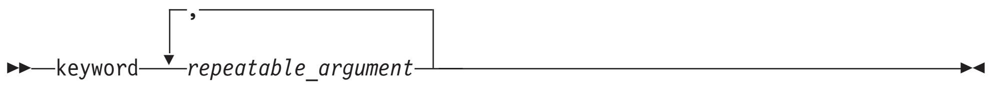
  If you *must* choose one of the items, one item of the stack is shown on the main path.



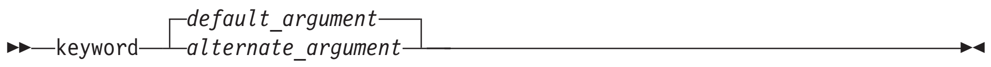
  If choosing one of the items is optional, the entire stack is shown below the main path.



- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:



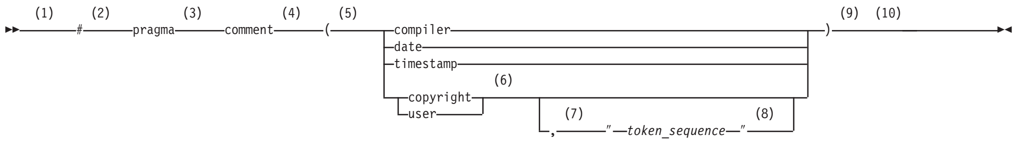
- The item that is the default is shown above the main path.



- Keywords are shown in nonitalic letters and should be entered exactly as shown.
- Variables are shown in italicized lowercase letters. They represent user-supplied names or values. If a variable or user-specified name ends in *_list*, you can provide a list of these terms separated by commas.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sample syntax diagram

The following syntax diagram example shows the syntax for the **#pragma comment** directive.



Notes:

1 This is the start of the syntax diagram.

2 The symbol # must appear first.

3 The keyword `pragma` must appear following the # symbol.

4 The name of the pragma `comment` must appear following the keyword `pragma`.

5 An opening parenthesis must be present.

6 The comment type must be entered only as one of the types indicated: `compiler`, `date`, `timestamp`, `copyright`, or `user`.

7 A comma must appear between the comment type `copyright` or `user`, and an optional character string.

8 A character string must follow the comma. The character string must be enclosed in double quotation marks.

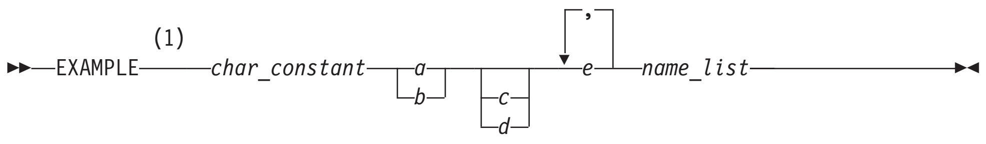
9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

The following examples of the #pragma comment directive are syntactically correct according to the diagram shown above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

The following is an example of a syntax diagram with an interpretation:



Notes:

1 IBM extension

Interpret the diagram as follows:

- Enter the keyword EXAMPLE.
- EXAMPLE is an IBM extension.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- Enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each. (The *_list* syntax is equivalent to the previous syntax for *e*.)

How to read syntax statements

Syntax statements are read from left to right:

- Individual required arguments are shown with no special notation.
- When you must make a choice between a set of alternatives, they are enclosed by { and } symbols.
- Optional arguments are enclosed by [and] symbols.
- When you can select from a group of choices, they are separated by | characters.
- Arguments that you can repeat are followed by ellipses (...).

Example of a syntax statement

`EXAMPLE` *char_constant* {*a*|*b*}[*c*|*d*]*e*[,*e*]... *name_list*{*name_list*}...

The following list explains the syntax statement:

- Enter the keyword EXAMPLE.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- Optionally, enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each *name*.

Note: The same example is used in both the syntax-statement and syntax-diagram representations.

Examples in this information

The examples in this information, except where otherwise noted, are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate all possible methods to achieve a specific result. Also, examples may use different compiler invocation commands interchangeably or simply indicate *invocation*. For detailed information on the commands available to invoke the compiler see the *IBM XL C/C++ for Multicore Acceleration for Linux, V9.0 Compiler Reference* or *IBM XL Fortran for Multicore Acceleration for Linux, V11.1 Compiler Guide*.

Related information

The following sections provide related information for XL C/C++ for Multicore Acceleration for Linux, V9.0 and XL Fortran for Multicore Acceleration for Linux, V11.1.

IBM XL C/C++ and XL Fortran information

XL C/C++ for Multicore Acceleration for Linux, V9.0 and XL Fortran for Multicore Acceleration for Linux, V11.1 provide product information in the following formats:

- README files

  README files contain late-breaking information, including changes and corrections to the product information. README files are located by default in the XL C/C++ and XL Fortran directories and in the root directories of the installation CDs.
- Installable man pages

  Man pages are provided for the compiler invocations and all command-line utilities provided with the products. Instructions for installing and accessing the man pages are provided in the *IBM XL C/C++ for Multicore Acceleration for Linux, V9.0 Installation Guide* and *IBM XL Fortran for Multicore Acceleration for Linux, V11.1 Installation Guide*.
- Information center

  The information center of searchable HTML files is viewable on the Web at:

  http://publib.boulder.ibm.com/infocenter/cellcomp/v9v111/index.jsp.
- PDF documents

The XL C/C++ for Multicore Acceleration for Linux PDF documents are located by default in the /opt/ibmcmp/xlcpp/cbe/9.0/doc/en_US/pdf/ directory and online at:

http://www.ibm.com/software/awdtools/xlcpp/library

The XL Fortran for Multicore Acceleration for Linux PDF documents are located by default in the /opt/ibmcmp/xlf/cbe/11.1/doc/en_US/pdf/ directory and online at:

http://www.ibm.com/software/awdtools/fortran/xlfortran/library

This document together with the following files comprise the full set of XL C/C++ for Multicore Acceleration for Linux, V9.0 and XL Fortran for Multicore Acceleration for Linux, V11.1 product information:

Table 2. XL C/C++ for Multicore Acceleration for Linux PDF files

Document title	**PDF file name**	**Description**
IBM XL C/C++ for Multicore Acceleration for Linux, V9.0 Installation Guide, GC23-8520-00	install.pdf	Contains information for installing XL C/C++ for Multicore Acceleration for Linux and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL C/C++ for Multicore Acceleration for Linux, V9.0, GC23-8518-00	getstart.pdf	Contains an introduction to the XL C/C++ for Multicore Acceleration for Linux product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C/C++ for Multicore Acceleration for Linux, V9.0 Compiler Reference, SC23-8516-00	compiler.pdf	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions.
IBM XL C/C++ for Multicore Acceleration for Linux, V9.0 Language Reference, SC23-8519-00	langref.pdf	Contains information about the C and C++ programming languages, as supported by IBM, including language extensions for portability and conformance to nonproprietary standards.
IBM XL C/C++ for Multicore Acceleration for Linux, V9.0 Programming Guide, SC23-5827-00	proguide.pdf	Contains information on advanced programming topics, such as application porting, library development, application optimization, and the XL C/C++ for Multicore Acceleration for Linux high-performance libraries.

Table 3. XL Fortran for Multicore Acceleration for Linux PDF files

Document title	**PDF file name**	**Description**
IBM XL Fortran for Multicore Acceleration for Linux, V11.1 Installation Guide, GC23-5834-00	install.pdf	Contains information for installing XL Fortran for Multicore Acceleration for Linux and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL Fortran for Multicore Acceleration for Linux, V11.1, GC23-5835-00	getstart.pdf	Contains an introduction to the XL Fortran for Multicore Acceleration for Linux product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.

Table 3. XL Fortran for Multicore Acceleration for Linux PDF files (continued)

Document title	PDF file name	Description
IBM XL Fortran for Multicore Acceleration for Linux, V11.1 Compiler Guide, SC23-5833-00	cr.pdf	Contains information about the various compiler options and environment variables.
IBM XL Fortran for Multicore Acceleration for Linux, V11.1 Language Reference, SC23-5832-00	lr.pdf	Contains information about the Fortran programming language as supported by IBM, including language extensions for portability and conformance to nonproprietary standards, compiler directives and intrinsic procedures.
IBM XL Fortran for Multicore Acceleration for Linux, V11.1 Optimization and Programming Guide, SC23-5836-00	opg.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls, floating-point operations, input/output, application optimization and parallelization, and the XL Fortran high-performance libraries.

To read a PDF file, use the Adobe® Reader. If you do not have the Adobe Reader, you can download it (subject to license terms) from the Adobe Web site at http://www.adobe.com.

More information related to XL C/C++ for Multicore Acceleration for Linux, V9.0 and XL Fortran for Multicore Acceleration for Linux, V11.1, including redbooks, white papers, tutorials, and other articles, is available on the Web at:

- http://www.ibm.com/software/awdtools/fortran/xlfortran/library (Fortran), and
- http://www.ibm.com/software/awdtools/xlcpp/library (C/C++)

Standards and specifications

XL C/C++ for Multicore Acceleration for Linux, V9.0 and XL Fortran for Multicore Acceleration for Linux, V11.1 are designed to support the following standards and specifications. You can refer to these standards for precise definitions of some of the features found in this information.

- *Information Technology – Programming languages – C, ISO/IEC 9899:1990*, also known as *C89*.
- *Information Technology – Programming languages – C, ISO/IEC 9899:1999*, also known as *C99*.
- *Information Technology – Programming languages – Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769*. This draft technical report has been accepted by the C standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf.
- *AltiVec Technology Programming Interface Manual*, Motorola Inc. This specification for vector data types, to support vector processing technology, is available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.
- *American National Standard Programming Language FORTRAN, ANSI X3.9-1978*.
- *American National Standard Programming Language Fortran 90, ANSI X3.198-1992*.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- *Federal (USA) Information Processing Standards Publication Fortran, FIPS PUB 69-1*.
- *Information technology - Programming languages - Fortran, ISO/IEC 1539-1:1991 (E)*.

- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:1997*. (This information uses its informal name, Fortran 95.)
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:2004*. (This information uses its informal name, Fortran 2003.)
- *Information technology - Programming languages - Fortran - Enhanced data type facilities, ISO/IEC JTC1/SC22/WG5 N1379*.
- *Information technology - Programming languages - Fortran - Floating-point exception handling, ISO/IEC JTC1/SC22/WG5 N1378*.
- *Military Standard Fortran DOD Supplement to ANSI X3.9-1978, MIL-STD-1753* (United States of America, Department of Defense standard). Note that XL Fortran supports only those extensions documented in this standard that have also been subsequently incorporated into the Fortran 90 standard.

Other IBM information

- *IBM C/C++ Language Extensions for Cell Broadband Engine Architecture*, available at http://www.ibm.com/developerworks/power/cell/documents.html
- Specifications, white papers, and other technical informations for the Cell Broadband Engine architecture are available at http://www.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine.
- The Cell Broadband Engine resource center, at http://www.ibm.com/developerworks/power/cell/, is the central repository for technical information, including articles, tutorials, programming guides, and educational resources.

Other information

- *Using the GNU Compiler Collection* available at http://gcc.gnu.org/onlinedocs

Technical support

Additional technical support is available from the XL C/C++ for Multicore Acceleration for Linux Support page at http://www.ibm.com/software/awdtools/xlcpp/support and the XL Fortran for Multicore Acceleration for Linux Support page at http://www.ibm.com/software/awdtools/fortran/xlfortran/support. These pages provide portals with search capabilities to a large selection of Technotes and other support information.

If you cannot find what you need, you can send e-mail to compinfo@ca.ibm.com.

For the latest information about XL C/C++ and XL Fortran for Multicore Acceleration for Linux, visit the product information sites at http://www.ibm.com/software/awdtools/xlcpp/support and http://www.ibm.com/software/awdtools/fortran/xlfortran.

How to send your comments

Your feedback is important in helping to provide accurate and high-quality information. If you have any comments about this information or any other XL C/C++ or XL Fortran for Multicore Acceleration for Linux information, send your comments by e-mail to compinfo@ca.ibm.com.

Be sure to include the name of the information, the part number of the information, the version of the compiler, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

Chapter 1. Effective address space support (SPU only)

The April 2008 PTF for XL C/C++ for Multicore Acceleration for Linux, V9.0 extends the C programming language syntax to allow programs written for the Synergistic Processor Unit (SPU) to access variables stored in the PowerPC® Processor Element's (PPE's) address space. This capability is known as *effective address space support*. By allowing SPU programs to access the PPE address space, effective address space support substantially increases the amount of memory space available to SPU programs. This feature is also known as PPE address space support on SPE as defined in the IBM Software Development Kit (SDK) for Multicore Acceleration V3.0 Programmer's Guide.

Before variables stored in the PPE address space can be accessed by the SPU program, they must be declared in SPU code using the type qualifier `__ea`. The `__ea` type qualifier indicates that the variable being declared in SPU code already exists in PPE address space. For more on the `__ea` type qualifier, see "Writing source code that uses effective address space support."

Effective address space support uses a software cache in combination with a cache manager library to maximize the speed with which variables stored in the PPE address space can be retrieved by the SPU. You can customize the size of the software cache by using the `-qswcache_size` compiler option and control how the PowerPC Processor Unit (PPU) memory is updated by the SPU software cache manager by using the `-qatomic_updates` and `-qnoatomic_updates` compiler options. For more on how to use effective address space support compiler options, see "Compiler options" on page 3.

Effective address space support is only available for the XL C/C++ for Multicore Acceleration for Linux, V9.0 C compiler. Furthermore, for effective address support to function, the IBM Software Development Kit (SDK) for Multicore Acceleration V3.0 must be installed and available on your system.

Writing source code that uses effective address space support

To create a variable that is stored in PPE address space but can be accessed by an SPU program, you must first allocate some PPE address space for its use, either by defining the variable in the PPU-targeted source code or by using an effective address space support memory allocation subroutine.

In addition to allocating PPE address space for the variable, you must also declare the variable in your SPU program. When you do, use the `__ea` type qualifier to mark it as a variable that already exists in PPE address space.

The `__ea` type qualifier is a PPE address namespace identifier. The syntax for using it is the same as for using type qualifiers `const` and `volatile`.

To create a variable that is stored in PPE address space, but can be accessed by SPU programs, do the following:

1. Define the variable in PPU-targeted source code. This causes some of the PPE's address space to be allocated for the variable's storage.
2. Declare an external variable of the same name in an SPU program that will be linked to the PPU code. When you declare the external variable, use the `__ea`

type qualifier; this will give the SPU program access to the variable defined in the PPU code. If you specify the `__ea` type qualifier for a non-pointer variable, it is your responsibility to ensure that the variable it qualifies has been declared and allocated by the PPU program.

Alternatively, if you use an effective address support memory allocation subroutine, you can allocate PPU memory and declare an `__ea` variable in a single statement. For example:

```
__ea int * p = malloc_ea(sizeof(int));
```

Example instructions

To declare a variable *a* that refers to an int in PPE address space that has been defined in PPU code, use the following source code:

```
extern __ea int a;
```

To create a pointer *ip1* in the Synergistic Processor Element (SPE) address space that points to a PPE address, use the following code:

```
__ea int* ip1
```

To create a pointer *ip2* in the PPE address space that points to an SPE address, use the following code:

```
extern int *__ea ip2
```

Casting pointers

The address space of each SPE is mapped onto the PPE address space, effectively making SPE address space a subset of PPE address space. As a result, all pointers to addresses in SPE address space can be cast to `__ea` pointers (pointers to addresses in PPE address space). Casting a non-null SPE pointer to an `__ea` pointer has the effect of changing the address the pointer points to from one in the SPE address space into an offset into the SPE local store mapped in the PPE address space.

However, because SPE address space only makes up a small subset of PPE address space, most `__ea` pointers cannot be cast to generic pointers. A non-null `__ea` pointer can only be successfully cast to an SPE pointer if the `__ea` pointer being cast points to an address inside the part of PPE address space to which the SPE's address space is mapped.

A null SPE address-space pointer always may be cast to a null `__ea` pointer, and a null `__ea` pointer always may be cast to a null SPE address-space pointer.

__ea type qualifier

The `__ea` type qualifier is an implementation-defined intrinsic named address space identifier, and it identifies a new type qualifier. This C Language syntax extension is based on the ″Named address space″ syntax provided by the ISO/IEC JTC1 SC22 WG14 Technical Report ″TR 18037: Embedded C″ (http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1169.pdf).

The following is a summary of some of the characteristics of the `__ea` type qualifier based on the proposed changes to the C99 standard by the Technical Report ″TR 18037″. For more detail please refer to that document.

- `__ea` denotes the 32–bit or 64–bit PPE named address space.

- Variables declared with the `__ea` type qualifier have external linkage.
- Each unqualified type has several qualified versions of its type, corresponding to the combinations of one, two, or all three of the const, volatile, and restrict qualifiers, and all combinations of any subset of these three qualifiers with the `__ea` type qualifier.
- Pointers to void in any address space (PPE or SPE) have the same representation and alignment requirements as a pointer to a character type in the same address space.
- Pointers to differently access-qualified versions of compatible types have the same representation and alignment requirements.
- All pointers to structure or union types in the same address space shall have the same representation and alignment requirements as each other.
- An `__ea` qualified pointer to void may be converted to or from a pointer to any incomplete or object type.
- Operands in a Relational, Equality, or Conditional expression can be in different address spaces.

The `__ea` type qualifier has the following constraints:
- The `__ea` type qualifier cannot be used for variable definitions or objects with automatic storage.
- The `__ea` type qualifier cannot be specified for a function type.
- The `__ea` type qualifier cannot be used to qualify the type of a compound literal in a function body.
- If both operands in a Simple Assignment expression are pointers to qualified or unqualified versions of compatible types, the referenced address space of the left encloses the referenced address space of the right, and the referenced type of the left has all the access qualifiers of the referenced type of the right.
- If one operand in a Simple Assignment expression is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of void, the referenced address space of the left encloses the referenced address space of the right, and the referenced type of the left has all the access qualifiers of the referenced type of the right.
- The `__ea` type qualifier cannot be used in the declaration of a structure or union member.

Compiler options

This PTF includes support for five new compiler options not documented in the *XL C/C++ Compiler Reference*.

-qea32, -qea64

Category

Pragma equivalent

None.

Purpose

Sets the compiler to compile for either a 32-bit or 64-bit PPE address space.

Syntax

```
          ┌ea32┐
►►── -q ──┴ea64┴───────────────────────────────────────────────────►◄
```

Defaults

-qea32 is the default setting.

Parameters

-qea32 | -qea64
If you are compiling for a 32-bit PPE address space, use `-qea32`. If you are compiling for a 64-bit PPE address space, use `-qea64`.

Usage

The compiler option `-qea32` should be used if you are linking your SPU program to a 32-bit PPU object, and `-qea64` should be used if you are linking your SPU program to a 64-bit PPU object. Only one of these options should be used at a time. If an SPU object is compiled using `-qea32` and then linked to a 64-bit PPU object, the behavior of the overall program is undefined; this is an invalid use of effective address space support.

These options are valid only for SPU invocation commands; for example, spuxlc.

Predefined macros

- __EA32__ is predefined when `-qea32` is specified.
- __EA64__ is predefined when `-qea64` is specified.

Examples

To compile a program that uses effective address space support, myprogram.c, that can access 64-bit PPU address space, enter:

```
spuxlc myprogram.c -qea64
```

-qswcache_size

Category

Pragma equivalent

None.

Purpose

Sets the size of the software cache.

You may need to experiment with different cache sizes to determine the optimal cache size for your SPU application. The optimal size will likely depend on the number of PPU references in your application and how much of the SPU's memory you are willing to devote to the software cache.

Syntax

```
►►── -q──swcache_size──=size──────────────────────────────────────────────►◄
```

Defaults

The default size of the software cache is 64 KB.

Parameters

size

The parameter **size** can be set to 8, 16, 32, 64, or 128, where the number given corresponds the size of software cache in KB.

Predefined macros

None.

Examples

To compile the program `myprogram.c` with the software cache size set to 16 KB, enter:

```
spuxlc myprogram.c -qswcache_size=16
```

-qatomic_updates, -qnoatomic_updates

Category

Pragma equivalent

None.

Purpose

Enables atomic updates to the software cache manager.

Atomic updates to the software cache ensure that the updates from SPU to PPU are done with no interleaving from other processors, so concurrent changes to adjacent bytes in the cache line are not lost. Atomic updates are safer, but also slower than non-atomic updates.

If you know that all the SPUs are writing to distinct memory areas, then there is no need for atomic updates. Similarly, if you are compiling a stand-alone SPU executable (spulet) then atomic updates are not necessary because the PPU and SPU will never run simultaneously.

Syntax

```
            ┌atomic_updates──┐
►►── -q─────┴noatomic_updates┴──────────────────────────────────────────►◄
```

Defaults

-qatomic_updates is the default setting.

Parameters

-qatomic_updates | **-qnoatomic_updates**
If -qatomic_updates is specified, all updates from the SPU to the PPU will be atomic. If -qnoatomic_updates is specified then updates to the PPU need not necessarily be atomic.

Predefined macros

None.

Examples

To compile the program `myprogram.c` and disable atomic updates, enter:

```
spuxlc myprogram.c -qnoatomic_updates
```

SDK 3.0 and the libc.a library

Effective address space support routines are included in the IBM Software Development Kit (SDK) for Multicore Acceleration V3.0 library libc.a. The routines in libc.a are declared in the header file `<ea.h>`.

The libc.a library contains the following system and support routines to help manage PPU-side memory:

EA-related PPE-assisted calls
These routines, implemented via PPE assists, are available as part of libc.a in the SDK.

EA memory and string access routines
These routines, implemented as SPU code, provide facilities analogous to the C99 string functions for values stored in PPU address space.

Sample program

The following is a simple example program which incorporates effective address space support. To create and compile the program, do the following:

1. Create two C files, spu.c and ppu.c.

   The file **spu.c**:

```
#include <stdio.h>
#include <spu_cache.h>
#include <spu_intrinsics.h>

/* pointer in PPU address space pointing to PPU address space */
extern __ea int* __ea p;

int i = 9;

int main(unsigned long long speid, unsigned long long argp, unsigned long long envp)
{
        /* p points to PPU-object i, not local SPU-object i */
        printf("From SPU: 0x%x\n", (int) p);
        printf("From SPU: %d\n", *p);

        /* PPU pointer p assigned to point to local SPU-object i */
        p = (__ea int*) &i;

        /* p now points to local SPU-object i, not PPU-object i */
        printf("From SPU: 0x%x\n", (int) p);
        printf("From SPU: %d\n", *p);

        /* flush the cache to make sure PPU sees the modified value */
        __cache_flush();

        /* pause the SPU, so that PPU can print the pointer */
        spu_stop(1);

        return 55;
}
```

The file **ppu.c**:

```
#include <stdio.h>
#include <libspe2.h>

int i = 5;
int *p = &i;

/* pointer to the SPE program, generated by ppu-embedspu */
extern spe_program_handle_t spu_prog;

int main()
{
        /* context returned by spe_context_create */
        spe_context_ptr_t speid;

        /* SPU program entry point - default */
        unsigned int entry = SPE_DEFAULT_ENTRY;

        /* structure used by spe_context_run to return information
           about SPU execution */
        spe_stop_info_t stop_info;

        /* create the SPE context */
        speid = spe_context_create(0, NULL);

        /* load the SPE program into the SPE context */
        spe_program_load(speid, &spu_prog);

        /* start the SPE program */
        spe_context_run(speid, &entry, 0, NULL, NULL, &stop_info);

        /* SPU program is now paused to let PPU output values */
        printf("From PPU: 0x%x\n", (int) p);
        printf("From PPU: %d\n", *p);

        /* resume the SPE program so that it finishes completely */
        spe_context_run(speid, &entry, 0, NULL, NULL, &stop_info);

        /* destroy the SPE context */
        spe_context_destroy(speid);

        return stop_info.result.spe_exit_code;
}
```

2. Compile and link the programs using:

```
ppuxlc pp.c -o ppu.o
spuxlc spu.c -o spuexe
ppu32-embedspu spu_prog spuexe spu.o
ppuxlc ppu.o spu.o -lspe2 -o cell_prog
```

3. Run the program `cell_prog`. The expected output is:

```
From SPU: 0x10030888
From SPU: 5
From SPU: 0xf7fb1c80
From SPU: 9
From PPU: 0xf7fb1c80
From PPU: 9
```

The PPU program declares and initializes the identifier *i* to a value of 5, then uses the pointer *p* to hold the address of *i*. Next, the SPU application declares `__ea` pointer *p*, which holds the address of the identifier *i* on the PPU. The SPU code also declares and initializes an *i* identifier (with a value of 9).

From the output we can see that *p* initially contains the address of the PPU object *i*, not the SPU object *i*. However, after *p* is assigned to point to `(__ea int*) &i` in spu.c, *p* points to the address of the SPU object *i*. More precisely, it points to the PPU address in which the SPU object *i*'s value is mapped.

After the call to `__cache_flush()`, which is also implicitly invoked when the SPU program finishes, the pointer p in the PPU program still points to the address of the SPU object *i*. This is demonstrated by the final two lines of output from the PPU.

Chapter 2. Fortran intrinsics (SPU only)

The April 2008 PTF for XL Fortran for Multicore Acceleration for Linux, V11.1 introduces 26 new generic intrinsics that make the underlying Instruction Set Architecture (ISA) and Synergistic Processor Unit (SPU) hardware accessible from the Fortran programming language. These intrinsics make it easier for you to access SPU hardware features and obtain the best performance from your SPU programs.

Each generic intrinsic maps to one or more assembly instructions. The mapping of a generic intrinsic to a specific assembly instruction or set of instructions depends on the input arguments to the intrinsic.

This PTF also adds three new vector types to XL Fortran for Multicore Acceleration for Linux, V11.1:

- VECTOR(INTEGER(8))
- VECTOR(UNSIGNED(8))
- VECTOR(REAL(8))

These vector types are only available for the SPU, and can only be used when spuxlf, spuxlf90, spuxlf95, or spuxlf2003 is invoked. For a complete list of supported vector types, see ″Vector types″ in the *IBM XL Fortran for Multicore Acceleration for Linux, V11.1 Language Reference*.

The following table lists the SPU intrinsics and their classifications:

Table 4. SPU Intrinsics for Fortran

Classification	**Intrinsics**
“Constant formation intrinsics” on page 11	spu_splats: Splat scalar to vector
“Conversion intrinsics” on page 12	spu_convtf: Convert integer vector to vector float
	spu_convts: Convert vector float to signed integer vector
	spu_extend: Extend vector
“Arithmetic intrinsics” on page 13	spu_add: Vector add
	spu_madd: Vector multiply and add
	spu_msub: Vector multiply and subtract
	spu_mul: Vector multiply
	spu_nmsub: Negative vector multiply and subtract
	spu_sub: Vector subtract
“Comparison intrinsics” on page 16	spu_cmpeq: Vector compare equal
	spu_cmpgt: Vector compare greater than

Table 4. SPU Intrinsics for Fortran (continued)

Classification	Intrinsics
"Bits and mask intrinsics" on page 19	spu_gather: Gather bits from elements
	spu_maskb: Form select byte mask
	spu_maskw: Form select word mask
	spu_sel: Select bits
	spu_shuffle: Shuffle two vectors of bytes
"Logical intrinsics" on page 22	spu_and: Vector bit-wise AND
	spu_andc: Vector bit-wise AND with complement
	spu_or: Vector bit-wise OR
	spu_xor: Vector bit-wise exclusive OR
"Shift and rotate intrinsics" on page 27	spu_rlqw: Quadword rotate left by bits
	spu_rlqwbyte: Quadword rotate left by bytes
"Scalar intrinsics" on page 29	spu_extract: Extract vector element from vector
	spu_insert: Insert scalar into specified vector element
	spu_promote: Promote scalar to vector

Mapping intrinsics with scalar operands

SPU intrinsics with scalar arguments can be used to achieve the same effect as SPU assembly instructions with immediate fields. For example, when the general intrinsic `spu_add` is called with operands of type VECTOR(INTEGER(4)) and INTEGER(4), it maps to the immediate-field assembly instruction `ai`.

For more on SPU assembly instructions and immediate fields, see the *SPU Assembly Language Specification*.

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:

```
vector(unsigned(4)) :: a, d
integer(4) :: b
d = spu_and(a, 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 5 shows possible uses of the immediate load instructions for various constants *b*.

Table 5. Possible uses of immediate load instructions for various values of constant b

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

- 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
```

Constant formation intrinsics

These intrinsics create a vector by replicating a scalar value across all elements of a vector of the same type.

spu_splats: Splat scalar to vector

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

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

Table 6. Splat scalar to vector

Return/Argument Types		**Assembly Mapping**
d	**a**	
VECTOR(INTEGER(1))	INTEGER(1)	SHUFB *d, a, a, pattern*
VECTOR(INTEGER(2))	INTEGER(2)	
VECTOR(INTEGER(4))	INTEGER(4)	
VECTOR(INTEGER(8))	INTEGER(8)	
VECTOR(REAL(4))	REAL(4)	
VECTOR(REAL(8))	REAL(8)	

Conversion intrinsics

These intrinsics convert vectors from one type to another.

spu_convtf: Convert integer vector to real vector

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*.

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

Table 7. Convert integer vector to real vector

Return/Argument Types			Assembly Mapping
d	**a**	**scale**	
VECTOR(REAL(4))	VECTOR(UNSIGNED(4))	INTEGER(4) • Literal • Range restricted to [0 to +127]	CUFLT *d*, *a*, *scale*
VECTOR(REAL(4))	VECTOR(INTEGER(4))		CSFLT *d*, *a*, *scale*

spu_convts: Convert real vector to signed integer vector

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*.

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

Table 8. Convert real vector to signed integer vector

Return/Argument Types			Assembly Mapping
d	**a**	**scale**	
VECTOR(INTEGER(4))	VECTOR(REAL(4))	INTEGER(4) • Literal • Range restricted to [0 to +127]	CUFLTS *d*, *a*, *scale*

spu_extend: Extend vector

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*.

d = spu_extend(*a*)

Table 9. Extend vector

Return/Argument Types		Assembly Mapping
d	a	
VECTOR(INTEGER(2))	VECTOR(INTEGER(1))	XSBH *d*, *a*
VECTOR(INTEGER(4))	VECTOR(INTEGER(2))	XSHW *d*, *a*
VECTOR(INTEGER(8))	VECTOR(INTEGER(4))	XSWD *d*, *a*
VECTOR(REAL(8))	VECTOR(REAL(4))	FESD *d*, *a*

Arithmetic intrinsics

These intrinsics perform simple mathematical calculations such as addition, subtraction, and multiplication.

spu_add: Vector add

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*.

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

Table 10. Vector add

Return/Argument Types			Assembly Mapping
d	a	b	
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	A *d*, *a*, *b*
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	AH *d*, *a*, *b*
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	AI *d*, *a*, *b*
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	INTEGER(4)	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	INTEGER(2) • Literal • Range restricted to [-512 to +511]	AHI *d*, *a*, *b*
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		

Table 10. Vector add (continued)

Return/Argument Types			Assembly Mapping
d	a	b	
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	INTEGER(2)	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		
VECTOR(REAL(4))	VECTOR(REAL(4))	VECTOR(REAL(4))	FA *d*, *a*, *b*
VECTOR(REAL(8))	VECTOR(REAL(8))	VECTOR(REAL(8))	DFA *d*, *a*, *b*

spu_madd: Vector multiply and add

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.

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

Table 11. Vector multiply and add

Return/Argument Types				Assembly Mapping
d	a	b	c	
VECTOR (INTEGER(4))	VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	VECTOR (INTEGER(4))	MPYA *d*, *a*, *b*, *c*
VECTOR (REAL(4))	VECTOR (REAL(4))	VECTOR (REAL(4))	VECTOR (REAL(4))	FMA *d*, *a*, *b*, *c*
VECTOR (REAL(8))	VECTOR (REAL(8))	VECTOR (REAL(8))	VECTOR (REAL(8))	rt <--- *c* DFMA rt, *a*, *b* *d* <--- rt

spu_msub: Vector multiply and subtract

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*.

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

Table 12. Vector multiply and subtract

Return/Argument Types				Assembly Mapping
d	a	b	c	
VECTOR (REAL(4))	VECTOR (REAL(4))	VECTOR (REAL(4))	VECTOR (REAL(4))	FMS *d*, *a*, *b*, *c*
VECTOR (REAL(8))	VECTOR (REAL(8))	VECTOR (REAL(8))	VECTOR (REAL(8))	rt <--- *c* DFMS rt, *a*, *b* *d* <--- rt

spu_mul: Vector multiply

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

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

Table 13. Vector multiply

Return/Argument Types			Assembly Mapping
d	a	b	
VECTOR(REAL(4))	VECTOR(REAL(4))	VECTOR(REAL(4))	FM *d*, *a*, *b*
VECTOR(REAL(8))	VECTOR(REAL(8))	VECTOR(REAL(8))	DFM *d*, *a*, *b*

spu_nmsub: Negative vector multiply and subtract

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*.

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

Table 14. Negative vector multiply and subtract

Return/Argument Types				Assembly Mapping
d	a	b	c	
VECTOR (REAL(4))	VECTOR (REAL(4))	VECTOR (REAL(4))	VECTOR (REAL(4))	FNMS *d*, *a*, *b*, *c*
VECTOR (REAL(8))	VECTOR (REAL(8))	VECTOR (REAL(8))	VECTOR (REAL(8))	rt <--- *c* DFNMS rt, *a*, *b* *d* <--- rt

spu_sub: Vector subtract

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*.

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

Table 15. Vector sub

Return/Argument Types			Assembly Mapping
d	a	b	
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	SFH *d*, *b*, *a*
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	SF *d*, *b*, *a*
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	
VECTOR (INTEGER(4))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	VECTOR (INTEGER(4))	SFI *d*, *b*, *a*
VECTOR (UNSIGNED(4))		VECTOR (UNSIGNED(4))	

Table 15. Vector sub (continued)

Return/Argument Types			Assembly Mapping
d	a	b	
VECTOR (INTEGER(4))	INTEGER(4)	VECTOR (INTEGER(4))	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(4))		VECTOR (UNSIGNED(4))	
VECTOR (INTEGER(2))	INTEGER(2) • Literal • Range restricted to [-512 to +511]	VECTOR (INTEGER(2))	SFHI *d*, *b*, *a*
VECTOR (UNSIGNED(2))		VECTOR (UNSIGNED(2))	
VECTOR (INTEGER(2))	INTEGER(2)	VECTOR (INTEGER(2))	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(2))		VECTOR (UNSIGNED(2))	
VECTOR(REAL(4))	VECTOR(REAL(4))	VECTOR(REAL(4))	FS *d*, *a*, *b*
VECTOR(REAL(8))	VECTOR(REAL(8))	VECTOR(REAL(8))	DFS *d*, *a*, *b*

Comparison intrinsics

These intrinsics compare the values in one vector to the values in another to determine which elements are equal to or greater than the corresponding elements in the other vector.

spu_cmpeq: Vector compare equal

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.

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

Table 16. Vector compare equal

Return/Argument Types			Assembly Mapping
d	a	b	
VECTOR (UNSIGNED(1))	VECTOR(INTEGER(1))	VECTOR(INTEGER(1))	CEQB *d*, *a*, *b*
	VECTOR(UNSIGNED(1))	VECTOR(UNSIGNED(1))	
VECTOR (UNSIGNED(2))	VECTOR(INTEGER(2))	VECTOR(INTEGER(2))	CEQH *d*, *a*, *b*
	VECTOR(UNSIGNED(2))	VECTOR(UNSIGNED(2))	
VECTOR (UNSIGNED(4))	VECTOR(INTEGER(4))	VECTOR(INTEGER(4))	CEQ *d*, *a*, *b*
	VECTOR(UNSIGNED(4))	VECTOR(UNSIGNED(4))	
	VECTOR(REAL(4))	VECTOR(REAL(4))	FCEQ *d*, *a*, *b*

Table 16. Vector compare equal (continued)

Return/Argument Types			Assembly Mapping
d	a	b	
VECTOR (UNSIGNED(1))	VECTOR(INTEGER(1))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	CEQBI *d*, *a*, *b*
	VECTOR(UNSIGNED(1))		
	VECTOR(INTEGER(1))	INTEGER(1)	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(2))	VECTOR(INTEGER(2))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	CEQHI *d*, *a*, b
	VECTOR(UNSIGNED(2))		
	VECTOR(INTEGER(2))	INTEGER(2)	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(4))	VECTOR(INTEGER(4))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	CEQI *d*, *a*, *b*
	VECTOR(UNSIGNED(4))		
	VECTOR(INTEGER(4))	INTEGER(4)	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(8))	VECTOR(REAL(8))	VECTOR(REAL(8))	DFCEQ *d*, *a*, *b*

spu_cmpgt: Vector compare greater than

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.

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

Table 17. Vector compare greater than

Return/Argument Types			Assembly Mapping
d	**a**	**b**	
VECTOR (UNSIGNED(1))	VECTOR (INTEGER(1))	VECTOR(INTEGER(1))	CGTB *d, a, b*
		INTEGER(4) • Literal • Range restricted to [-512 to +511]	CGTBI *d, a, b*
		INTEGER(1)	Refer to "Mapping intrinsics with scalar operands" on page 10
	VECTOR (UNSIGNED(1))	VECTOR(UNSIGNED(1))	CLGTB *d, a, b*
		INTEGER(4) • Literal • Range restricted to [-512 to +511]	CLGTBI *d, a, b*
VECTOR (UNSIGNED(2))	VECTOR (INTEGER(2))	VECTOR(INTEGER(2))	CGTH *d, a, b*
		INTEGER(4) • Literal • Range restricted to [-512 to +511]	CGTHI *d, a, b*
		INTEGER(2)	Refer to "Mapping intrinsics with scalar operands" on page 10
	VECTOR (UNSIGNED(2))	VECTOR(UNSIGNED(2))	CLGTH *d, a, b*
		INTEGER(4) • Literal • Range restricted to [-512 to +511]	CLGTHI *d, a, b*

Table 17. Vector compare greater than (continued)

Return/Argument Types			Assembly Mapping
d	**a**	**b**	
VECTOR (UNSIGNED(4))	VECTOR (INTEGER(4))	VECTOR(INTEGER(4))	CGT *d*, *a*, *b*
		INTEGER(4) • Literal • Range restricted to [-512 to +511]	CGTI *d*, *a*, *b*
		INTEGER(4)	Refer to "Mapping intrinsics with scalar operands" on page 10
	VECTOR (UNSIGNED(4))	VECTOR(UNSIGNED(4))	CLGT *d*, *a*, *b*
		INTEGER(4) • Literal • Range restricted to [-512 to +511]	CLGTI *d*, *a*, *b*
	VECTOR (REAL(4))	VECTOR(REAL(4))	FCGT *d*, *a*, *b*
VECTOR (UNSIGNED(8))	VECTOR (REAL(8))	VECTOR(REAL(8))	DFCGT *d*, *a*, *b*

Bits and mask intrinsics

These intrinsics gather bits, create vector masks, or combine two vectors into one according to a given pattern.

spu_gather: Gather bits from elements

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.

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

Table 18. Gather bits from elements

Return/Argument Types		Assembly Mapping
d	**a**	
VECTOR(UNSIGNED(4))	VECTOR(UNSIGNED(1))	GBB *d*, *a*
	VECTOR(INTEGER(1))	
	VECTOR(UNSIGNED(2))	GBH *d*, *a*
	VECTOR(INTEGER(2))	
	VECTOR(UNSIGNED(4))	GB *d*, *a*
	VECTOR(INTEGER(4))	
	VECTOR(REAL(4))	

spu_maskb: Form select byte mask

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*.

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

Table 19. Form select byte mask

Return/Argument Types		Assembly Mapping
d	a	
VECTOR(UNSIGNED(1))	INTEGER(2)	FSMB *d*, *a*
	INTEGER(4)	

spu_maskw: Form select word mask

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*.

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

Table 20. Form select word mask

Return/Argument Types		Assembly Mapping
d	a	
VECTOR(UNSIGNED(4))	INTEGER(1)	FSM *d*, *a*
	INTEGER(2)	
	INTEGER(4)	

spu_sel: Select bits

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*.

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

Table 21. Convert integer vector to vector float

Return/Argument Types				Assembly Mapping
d	**a**	**b**	**pattern**	
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))	SELB *d*, *a*, *b*, *pattern*
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	VECTOR (INTEGER(1))		
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	VECTOR (INTEGER(2))		
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	VECTOR (INTEGER(4))		
VECTOR(REAL(4))	VECTOR(REAL(4))	VECTOR(REAL(4))		
VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))	
VECTOR (INTEGER(8))	VECTOR (INTEGER(8))	VECTOR (INTEGER(8))		
VECTOR(REAL(8))	VECTOR(REAL(8))	VECTOR(REAL(8))		

spu_shuffle: Shuffle two vectors of bytes

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*.

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

Table 22. Shuffle pattern

Value in the Byte of pattern (in binary)	Resulting Byte
10xxxxxx	0x00
110xxxxx	0xFF
111xxxxx	0x80
otherwise	The byte of (*a*||*b*) addressed by the rightmost 5 bits of pattern

Table 23. Shuffle two vectors of bytes

Return/Argument Types				**Assembly Mapping**
d	**a**	**b**	**pattern**	
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	VECTOR (UNSIGNED(1))	SHUFB *d*, *a*, *b*, *pattern*
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))		
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	VECTOR (INTEGER(2))		
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	VECTOR (INTEGER(4))		
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		
VECTOR (INTEGER(8))	VECTOR (INTEGER(8))	VECTOR (INTEGER(8))		
VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))		
VECTOR (REAL(4))	VECTOR (REAL(4))	VECTOR (REAL(4))		
VECTOR (REAL(8))	VECTOR (REAL(8))	VECTOR (REAL(8))		

Logical intrinsics

These intrinsics perform logical operations such as AND, OR, and XOR.

spu_and: Vector bit-wise AND

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*.

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

Table 24. Vector bit-wise AND

Return/Argument Types			Assembly Mapping
d	**a**	**b**	
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	AND *d*, *a*, *b*
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))	
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	
VECTOR (INTEGER(8))	VECTOR (INTEGER(8))	VECTOR (INTEGER(8))	
VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))	
VECTOR(REAL(4))	VECTOR(REAL(4))	VECTOR(REAL(4))	
VECTOR(REAL(8))	VECTOR(REAL(8))	VECTOR(REAL(8))	
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	ANDBI *d*, *a*, *b*
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))		
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	INTEGER(1)	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))		
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	ANDHI *d*, *a*, *b*
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	INTEGER(2)	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	ANDI *d*, *a*, *b*
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		

Table 24. Vector bit-wise AND (continued)

Return/Argument Types			Assembly Mapping
d	a	b	
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	INTEGER(4)	Refer to “Mapping intrinsics with scalar operands” on page 10
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		

spu_andc: Vector bit-wise AND with complement

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*.

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

Table 25. Vector bit-wise AND

Return/Argument Types			Assembly Mapping
d	a	b	
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	ANDC *d*, *a*, *b*
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))	
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	
VECTOR (INTEGER(8))	VECTOR (INTEGER(8))	VECTOR (INTEGER(8))	
VECTOR(REAL(4))	VECTOR(REAL(4))	VECTOR(REAL(4))	
VECTOR(REAL(8))	VECTOR(REAL(8))	VECTOR(REAL(8))	

spu_or: Vector bit-wise OR

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*.

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

Table 26. Vector bit-wise OR

Return/Argument Types			Assembly Mapping
d	**a**	**b**	
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	OR *d*, *a*, *b*
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))	
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	
VECTOR (INTEGER(8))	VECTOR (INTEGER(8))	VECTOR (INTEGER(8))	
VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))	
VECTOR(REAL(4))	VECTOR(REAL(4))	VECTOR(REAL(4))	
VECTOR(REAL(8))	VECTOR(REAL(8))	VECTOR(REAL(8))	
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	ORBI *d*, *a*, *b*
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))		
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	INTEGER(1)	Refer to “Mapping intrinsics with scalar operands” on page 10
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))		
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	ORHI *d*, *a*, *b*
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	INTEGER(2)	Refer to “Mapping intrinsics with scalar operands” on page 10
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	ORI *d*, *a*, *b*
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		

Table 26. Vector bit-wise OR (continued)

Return/Argument Types			Assembly Mapping
d	a	b	
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	INTEGER(4)	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		

spu_xor: Vector bit-wise exclusive OR

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*.

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

Table 27. Vector bit-wise exclusive OR

Return/Argument Types			Assembly Mapping
d	a	b	
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	XOR *d*, *a*, *b*
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))	
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))	
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))	
VECTOR (INTEGER(8))	VECTOR (INTEGER(8))	VECTOR (INTEGER(8))	
VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))	
VECTOR(REAL(4))	VECTOR(REAL(4))	VECTOR(REAL(4))	
VECTOR(REAL(8))	VECTOR(REAL(8))	VECTOR(REAL(8))	
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	XORBI *d*, *a*, *b*
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))		
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	INTEGER(1)	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))		

Table 27. Vector bit-wise exclusive OR (continued)

Return/Argument Types			Assembly Mapping
d	a	b	
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	XORHI *d*, *a*, *b*
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))	INTEGER(2)	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	INTEGER(4) • Literal • Range restricted to [-512 to +511]	XORI *d*, *a*, *b*
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))	INTEGER(4)	Refer to "Mapping intrinsics with scalar operands" on page 10
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		

Shift and rotate intrinsics

These intrinsics move the values inside a vector, rotating them out on one end and in on the other.

spu_rlqw: Quadword rotate left by bits

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*.

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

Table 28. Quadword rotate left by bits

Return/Argument Types			Assembly Mapping
d	**a**	**count**	
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	INTEGER(4) • Literal • Range restricted to [-64 to +63]	ROTQBII *d*, *a*, *count*
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))		
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))		
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))		
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		
VECTOR (INTEGER(8))	VECTOR (INTEGER(8))		
VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))		
VECTOR(REAL(4))	VECTOR(REAL(4))		
VECTOR(REAL(8))	VECTOR(REAL(8))		
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	INTEGER(4) • Non-literal	ROTQBI *d*, *a*, *count*
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))		
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))		
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))		
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		
VECTOR (INTEGER(8))	VECTOR (INTEGER(8))		
VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))		
VECTOR(REAL(4))	VECTOR(REAL(4))		
VECTOR(REAL(8))	VECTOR(REAL(8))		

spu_rlqwbyte: Quadword rotate left by bytes

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*.

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

Table 29. Quadword rotate left by bytes

Return/Argument Types			Assembly Mapping
d	a	count	
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	INTEGER(4) • Literal • Range restricted to [-64 to +63]	ROTQBYI *d*, *a*, *count*
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))		
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))		
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))		
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		
VECTOR (INTEGER(8))	VECTOR (INTEGER(8))		
VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))		
VECTOR(REAL(4))	VECTOR(REAL(4))		
VECTOR(REAL(8))	VECTOR(REAL(8))		
VECTOR (INTEGER(1))	VECTOR (INTEGER(1))	INTEGER(4) • Non-literal	ROTQBY *d*, *a*, *count*
VECTOR (UNSIGNED(1))	VECTOR (UNSIGNED(1))		
VECTOR (INTEGER(2))	VECTOR (INTEGER(2))		
VECTOR (UNSIGNED(2))	VECTOR (UNSIGNED(2))		
VECTOR (INTEGER(4))	VECTOR (INTEGER(4))		
VECTOR (UNSIGNED(4))	VECTOR (UNSIGNED(4))		
VECTOR (INTEGER(8))	VECTOR (INTEGER(8))		
VECTOR (UNSIGNED(8))	VECTOR (UNSIGNED(8))		
VECTOR(REAL(4))	VECTOR(REAL(4))		
VECTOR(REAL(8))	VECTOR(REAL(8))		

Scalar intrinsics

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 Fortran, such as shuffling bytes.

spu_extract: Extract vector element from vector

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.

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

Table 30. Extract vector element from vector

Return/Argument Types			Assembly Mapping
d	**a**	**element**	
INTEGER(1)	VECTOR(INTEGER(1))	INTEGER(4)	ROTQBY *d*, *a*, *element*
INTEGER(2)	VECTOR(INTEGER(2))		SHLI t, *element*, 1
INTEGER(4)	VECTOR(INTEGER(4))		SHLI t, *element*, 2
INTEGER(8)	VECTOR(INTEGER(8))		SHLI t, *element*, 3
REAL(4)	VECTOR(REAL(4))		SHLI t, *element*, 2
REAL(8)	VECTOR(REAL(8))		SHLI t, *element*, 3

spu_insert: Insert scalar into specified vector 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.

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

Table 31. Insert scalar into specified vector element

				Assembly Mapping
d	**a**	**b**	**element**	
VECTOR (INTEGER(1))	INTEGER(1)	VECTOR (INTEGER(1))	INTEGER(4)	CBD t, 0(*element*) SHUFB *d*, *a*, *b*, t
VECTOR (INTEGER(2))	INTEGER(2)	VECTOR (INTEGER(2))		SHLI t, *element*, 1 CHD t, 0(t)
VECTOR (INTEGER(4))	INTEGER(4)	VECTOR (INTEGER(4))		SHLI t, *element*, 2 CWD t, 0(t) SHUFB *d*, *a*, *b*, t
VECTOR (INTEGER(8))	INTEGER(8)	VECTOR (INTEGER(8))		SHLI t, *element*, 3 CWD t, 0(t) SHUFB *d*, *a*, *b*, t
VECTOR (REAL(4))	REAL(4)	VECTOR (REAL(4))		SHLI t, *element*, 2 CWD t, 0(t) SHUFB *d*, *a*, *b*, t
VECTOR (REAL(8))	REAL(8)	VECTOR (REAL(8))		SHLI t, *element*, 3 CWD t, 0(t) SHUFB *d*, *a*, *b*, t

spu_promote: Promote scalar to vector

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.

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

Table 32. Promote scalar to vector

			Assembly Mapping
d	**a**	**element**	
VECTOR(INTEGER(1))	INTEGER(1)	INTEGER(4)	SFI t, *element*, 3 ROTQBY *d*, *a*, t
VECTOR(INTEGER(2))	INTEGER(2)		SFI t, *element*, 1 SHLI t, t, 1 ROTQBY *d*, *a*, t
VECTOR(INTEGER(4))	INTEGER(4)		SFI t, *element*, 0 SHLI t, t, 2 ROTQBY *d*, *a*, t
VECTOR(INTEGER(8))	INTEGER(8)		SHLI t, *element*, 3 ROTQBY *d*, *a*, t
VECTOR(REAL(4))	REAL(4)		SFI t, *element*, 0 SHLI t, t, 2 ROTQBY *d*, *a*, t
VECTOR(REAL(8))	REAL(8)		SHLI t, *element*, 3 ROTQBY *d*, *a*, t

Notices

Trademarks and service marks

Printed in USA