IBM XL C/C++ for Linux on z Systems, V1.2

Optimization and Programming Guide

Version 1.2

SC27-5997-01

IBM XL C/C++ for Linux on z Systems, V1.2

Optimization and Programming Guide

Version 1.2

SC27-5997-01

Note

Before using this information and the product it supports, read the information in “Notices” on page 251.

First edition

This edition applies to IBM XL C/C++ for Linux on z Systems, V1.2 (Program 5725-N01) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

About this document

This guide discusses advanced topics related to the use of IBM® XL C/C++ for Linux on z Systems™, V1.2, with a particular focus on program portability and optimization. The guide provides both reference information and practical tips for getting the most out of the compiler's capabilities through recommended programming practices and compilation procedures.

Who should read this document

This document is addressed to programmers building complex applications, who already have experience compiling with XL C/C++ and would like to take further advantage of the compiler's capabilities for program optimization and tuning, support for advanced programming language features, and add-on tools and utilities.

How to use this document

This document uses a task-oriented approach to present the topics by concentrating on a specific programming or compilation problem in each section. Each topic contains extensive cross-references to the relevant sections of the reference guides in the IBM XL C/C++ for Linux on z Systems, V1.2 documentation set, which provides detailed descriptions of compiler options, pragmas, and specific language extensions.

How this document is organized

This guide includes the following chapters:

- Chapter 1, "Using 31-bit and 64-bit modes," on page 1 discusses common problems that arise when you port existing 31-bit applications to 64-bit mode, and it provides recommendations for avoiding these problems.
- Chapter 2, "Aligning data," on page 5 discusses options available for controlling the alignment of data in aggregates, such as structures and classes.
- Chapter 3, "Handling floating-point operations," on page 11 discusses options available for controlling how floating-point operations are handled by the compiler.
- Chapter 4, "Constructing a library," on page 15 discusses how to compile and link static and shared libraries and how to specify the initialization order of static objects in C++ programs.
- Chapter 5, "Optimizing your applications," on page 21 discusses various options provided by the compiler for optimizing your programs, and it provides recommendations on how to use these options.
- Chapter 6, "Debugging optimized code," on page 39 discusses the potential usability problems of optimized programs and the options that can be used to debug optimized code.
- Chapter 7, "Coding your application to improve performance," on page 43 discusses recommended programming practices and coding techniques to enhance program performance and compatibility with the compiler's optimization capabilities.
- Chapter 8, "Using vector programming support," on page 63 introduces how to use the Vector Facility for z/Architecture® on the Linux distributions that have

vector support and run on the IBM z13™ models. The provided vector programming support includes vector data types, expressions, operators, and vector built-in functions.

- Chapter 9, "Using the high performance libraries," on page 231 discusses two performance libraries that are shipped with XL C/C++: the Mathematical Acceleration Subsystem (MASS), which contains tuned versions of standard math library functions, and the Automatically Tuned Liner Algebra Software libraries for algebra high-performance computing.

Conventions

Typographical conventions

The following table shows the typographical conventions used in the IBM XL C/C++ for Linux on z Systems, V1.2 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, **xlc** and **xlC** (**xlc++**), along with several other compiler invocation commands to support various C/C++ language levels and compilation environments.
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.c, enter: `xlc myprogram.c -O3`.

Qualifying elements (icons)

Most features described in this information apply to both C and C++ languages. In descriptions of language elements where a feature is exclusive to one language, or where functionality differs between languages, this information uses icons to delineate segments of text as follows:

Table 2. Qualifying elements

Qualifier/Icon	Meaning
C only begins ▶ C C ◀ C only ends	The text describes a feature that is supported in the C language only; or describes behavior that is specific to the C language.

Table 2. Qualifying elements (continued)

Qualifier/Icon	**Meaning**
C++ only begins ▶ C++ C++ ◀ C++ only ends	The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.
IBM extension begins ▶ IBM IBM ◀ IBM extension ends	The text describes a feature that is an IBM extension to the standard language specifications.
C11 begins ▶ C11 C11 ◀ C11 ends	The text describes a feature that is introduced into standard C as part of C11.
C++11 begins ▶ C++11 C++11 ◀ C++11 ends	The text describes a feature that is introduced into standard C++ as part of C++11.
C++14 begins ▶ C++14 C++14 ◀ C++14 ends	The text describes a feature that is introduced into standard C++ as part of C++14.

Syntax diagrams

Throughout this information, diagrams illustrate XL C/C++ syntax. This section helps 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.

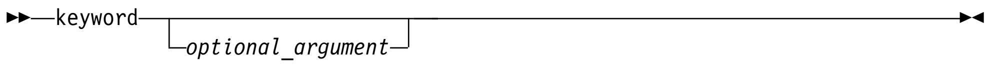
  The ▶── symbol indicates that a command, directive, or statement is continued from the previous line.

  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.
- Required items are shown on the horizontal line (the main path):

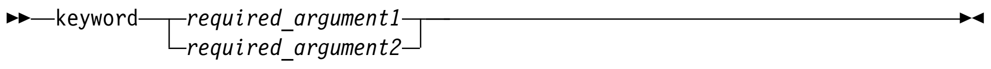
```
▶▶──keyword──required_argument──────────────────────────────────────▶◀
```

- Optional items are shown below the main path:

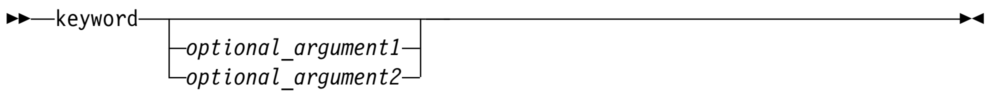


- If you can choose from two or more items, they are shown vertically, in a stack.

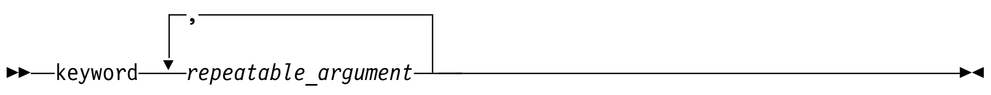
  If you *must* choose one of the items, one item of the stack is shown on the main path.



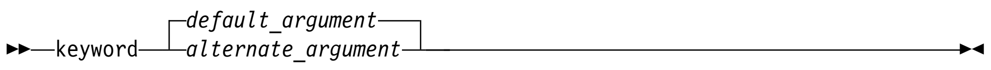
  If choosing one of the items is optional, the entire stack is shown below the main path.



- 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 punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

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.

The examples for installation information are labelled as either *Example* or *Basic example*. *Basic examples* are intended to document a procedure as it would be performed during a basic, or default, installation; these need little or no modification.

Related information

The following sections provide related information for XL C/C++:

IBM XL C/C++ information

XL C/C++ provides product information in the following formats:

- Quick Start Guide

  The Quick Start Guide (`quickstart.pdf`) is intended to get you started with IBM XL C/C++ for Linux on z Systems, V1.2. It is located by default in the XL C/C++ directory and in the `\quickstart` directory of the installation DVD.
- 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++ directory, and in the root directory and subdirectories of the installation DVD.
- Installable man pages

  Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *IBM XL C/C++ for Linux on z Systems, V1.2 Installation Guide*.
- Online product documentation

  The fully searchable HTML-based documentation is viewable in IBM Knowledge Center at http://www.ibm.com/support/knowledgecenter/SSVUN6_1.2.0/com.ibm.compilers.loz.doc/welcome.html.
- PDF documents

  PDF documents are available on the web at http://www.ibm.com/support/docview.wss?uid=swg27044043.

  The following files comprise the full set of XL C/C++ product information:

Table 3. XL C/C++ PDF files

Document title	PDF file name	Description
IBM XL C/C++ for Linux on z Systems, V1.2 Installation Guide, GC27-5995-01	`install.pdf`	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL C/C++ for Linux on z Systems, V1.2, GI13-2865-01	`getstart.pdf`	Contains an introduction to the XL C/C++ product, with information about setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.

Table 3. XL C/C++ PDF files (continued)

Document title	PDF file name	Description
IBM XL C/C++ for Linux on z Systems, V1.2 Compiler Reference, SC27-5998-01	`compiler.pdf`	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions.
IBM XL C/C++ for Linux on z Systems, V1.2 Language Reference, SC27-5996-01	`langref.pdf`	Contains information about language extensions for portability and conformance to nonproprietary standards.
IBM XL C/C++ for Linux on z Systems, V1.2 Optimization and Programming Guide, SC27-5997-01	`proguide.pdf`	Contains information about advanced programming topics, such as application porting, library development, application optimization, and the XL C/C++ high-performance libraries.

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

More information related to XL C/C++, including IBM Redbooks® publications, white papers, and other articles, is available on the web at http://www.ibm.com/support/docview.wss?uid=swg27044043.

For more information about C/C++, see the C/C++ café at https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=5894415f-be62-4bc0-81c5-3956e82276f3.

Standards and specifications

XL C/C++ is designed to support the following standards and specifications. You can refer to these standards and specifications 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 - C, ISO/IEC 9899:2011*, also known as *C11*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:1998*, also known as *C++98*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2003*, also known as *C++03*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2011*, also known as *C++11* (Partial support).
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2014*, also known as *C++14* (Partial support).
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985.*

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++ Support page at http://www.ibm.com/support/entry/portal/product/rational/xl_c/c++_for_linux_on_z_systems. This page provides a portal with search capabilities to a large selection of Technotes and other support information.

If you cannot find what you need, you can send an email to compinfo@ca.ibm.com.

For the latest information about XL C/C++, visit the product information site at http://www.ibm.com/software/products/en/xlcpp-loz.

How to send your comments

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

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

Chapter 1. Using 31-bit and 64-bit modes

You can use the XL C/C++ compiler to develop either 31-bit or 64-bit applications. To do so, specify **-m31** or **-m64** (the default), respectively, during compilation.

However, porting existing applications from 31-bit to 64-bit mode can lead to a number of problems, mostly related to the differences in C/C++ long and pointer data type sizes and alignment between the two modes. The following table summarizes these differences.

Table 4. Size and alignment of data types in 31-bit and 64-bit modes

Data type	**31-bit mode**		**64-bit mode**	
	Size	**Alignment**	**Size**	**Alignment**
long, signed long, unsigned long	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
pointer	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
size_t (defined in the header file <cstddef>)	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
ptrdiff_t (defined in the header file <cstddef>)	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries

The following sections discuss some of the common pitfalls implied by these differences, as well as recommended programming practices to help you avoid most of these issues:
- "Assigning long values"
- "Assigning pointers" on page 3
- "Aligning aggregate data" on page 4

For suggestions on improving performance in 64-bit mode, see "Optimize operations in 64-bit mode".

Related information in the *XL C/C++ Compiler Reference*

-m31, -m64 (-q31, -q64)

Compile-time and link-time environment variables

Assigning long values

The limits of `long` type integers that are defined in the `limits.h` standard library header file are different in 31-bit and 64-bit modes, as shown in the following table.

Table 5. Constant limits of long integers in 31-bit and 64-bit modes

Symbolic constant	**Mode**	**Value**	**Hexadecimal**	**Decimal**
LONG_MIN (smallest signed long)	31-bit	$-(2^{31})$	0x80000000L	–2,147,483,648
	64-bit	$-(2^{63})$	0x8000000000000000L	–9,223,372,036,854,775,808

Table 5. Constant limits of long integers in 31-bit and 64-bit modes (continued)

Symbolic constant	Mode	Value	Hexadecimal	Decimal
LONG_MAX (largest signed long)	31-bit	$2^{31}–1$	0x7FFFFFFFL	2,147,483,647
	64-bit	$2^{63}–1$	0x7FFFFFFFFFFFFFFFL	9,223,372,036,854,775,807
ULONG_MAX (largest unsigned long)	31-bit	$2^{32}–1$	0xFFFFFFFFUL	4,294,967,295
	64-bit	$2^{64}–1$	0xFFFFFFFFFFFFFFFFUL	18,446,744,073,709,551,615

These differences have the following implications:
- Assigning a `long` value to a `double` variable can cause loss of accuracy.
- Assigning constant values to `long` variables can lead to unexpected results. This issue is explored in more detail in "Assigning constant values to long variables."
- Bit-shifting long values will produce different results, as described in "Bit-shifting long values" on page 3.
- Using `int` and `long` types interchangeably in expressions will lead to implicit conversion through promotions, demotions, assignments, and argument passing, and it can result in truncation of significant digits, sign shifting, or unexpected results, without warning. These operations can impact performance.

In situations where a `long` value can overflow when assigned to other variables or passed to functions, you must observe the following guidelines:
- Avoid implicit type conversion by using explicit type casting to change types.
- Ensure that all functions that accept or return `long` types are properly prototyped.
- Ensure that `long` type parameters can be accepted by the functions to which they are being passed.

Assigning constant values to long variables

Although type identification of constants follows explicit rules in C and C++, many programs use hexadecimal or unsuffixed constants as "typeless" variables and rely on a twos complement representation to truncate values that exceed the limits permitted on a 31-bit system. As these large values are likely to be extended into a 64-bit `long` type in 64-bit mode, unexpected results can occur, generally at the following boundary areas:
- constant > `UINT_MAX`
- constant < `INT_MIN`
- constant > `INT_MAX`

Some examples of unexpected boundary side effects are listed in the following table.

Table 6. Unexpected boundary results of constants assigned to long types

Constant assigned to long	Equivalent value	31-bit mode	64-bit mode
–2,147,483,649	INT_MIN–1	+2,147,483,647	–2,147,483,649
+2,147,483,648	INT_MAX+1	–2,147,483,648	+2,147,483,648
+4,294,967,726	UINT_MAX+1	0	+4,294,967,296
0xFFFFFFFF	UINT_MAX	–1	+4,294,967,295

Table 6. Unexpected boundary results of constants assigned to long types (continued)

Constant assigned to long	Equivalent value	31-bit mode	64-bit mode
0x100000000	UINT_MAX+1	0	+4,294,967,296
0xFFFFFFFFFFFFFFFF	ULONG_MAX	–1	–1

Unsuffixed constants can lead to type ambiguities that can affect other parts of your program, such as when the results of `sizeof` operations are assigned to variables. For example, in 31-bit mode, the compiler types a number like 4294967295 (`UINT_MAX`) as an unsigned long and `sizeof` returns 4 bytes. In 64-bit mode, this same number becomes a signed long and `sizeof` returns 8 bytes. Similar problems occur when the compiler passes constants directly to functions.

You can avoid these problems by using the suffixes `L` (for long constants), `UL` (for unsigned long constants), `LL` (for long long constants), or `ULL` (for unsigned long long constants) to explicitly type all constants that have the potential of affecting assignment or expression evaluation in other parts of your program. In the example cited in the preceding paragraph, suffixing the number as `4294967295U` forces the compiler to always recognize the constant as an `unsigned int` in 31-bit or 64-bit mode. These suffixes can also be applied to hexadecimal constants.

Bit-shifting long values

Left-bit-shifting long values produces different results in 31-bit and 64-bit modes. The examples in Table 7 show the effects of performing a bit-shift on long constants using the following code segment:

```
long l=valueL<<1;
```

Table 7. Results of bit-shifting long values

Initial value	Symbolic constant	Value after bit shift by one bit	
		31-bit mode	64-bit mode
0x7FFFFFFFL	INT_MAX	0xFFFFFFFE	0x00000000FFFFFFFE
0x80000000L	INT_MIN	0x00000000	0x0000000100000000
0xFFFFFFFFL	UINT_MAX	0xFFFFFFFE	0x00000001FFFFFFFE

In 31-bit mode, `0xFFFFFFFE` is negative. In 64-bit mode, `0x00000000FFFFFFFE` and `0x00000001FFFFFFFE` are both positive.

Assigning pointers

In 64-bit mode, pointers and `int` types are no longer of the same size. The implications of this are as follows:

- Exchanging pointers and `int` types causes segmentation faults.
- Passing pointers to a function expecting an `int` type results in truncation.
- Functions that return a pointer but are not explicitly prototyped as such, return an `int` instead and truncate the resulting pointer, as illustrated in the following example.

  In C, the following code is valid in 31-bit mode without a prototype:

  ```
  a=(char*) calloc(25);
  ```

Without a function prototype for `calloc`, when the same code is compiled in 64-bit mode, the compiler assumes the function returns an `int`, so `a` is silently truncated and then sign-extended. Type casting the result does not prevent the truncation, as the address of the memory allocated by `calloc` was already truncated during the return. In this example, the best solution is to include the header file, `stdlib.h`, which contains the prototype for `calloc`. An alternative solution is to prototype the function as it is in the header file.

To avoid these types of problems, you can take the following measures:
- Prototype any functions that return a pointer, where possible by using the appropriate header file.
- Ensure that the type of parameter you are passing in a function, pointer or `int`, call matches the type expected by the function being called.
- For applications that treat pointers as an integer type, use type `long` or `unsigned long` in either 31-bit or 64-bit mode.

Aligning aggregate data

Normally, structures are aligned according to the most strictly aligned member in both 31-bit and 64-bit modes. However, since `long` types and pointers change size and alignment in 64-bit modes, the alignment of a structure's strictest member can change, resulting in changes to the alignment of the structure itself.

Structures that contain pointers or `long` types cannot be shared between 31-bit and 64-bit applications. Unions that attempt to share `long` and `int` types or overlay pointers onto `int` types can change the alignment. In general, you need to check all but the simplest structures for alignment and size dependencies.

Any aggregate data written to a file in one mode cannot be correctly read in the other mode. Data exchanged with other languages has the similar problems.

For detailed information about aligning data structures, including structures that contain bit fields, see Chapter 2, "Aligning data," on page 5.

Chapter 2. Aligning data

XL C/C++ provides many mechanisms for specifying data alignment at the levels of individual variables, members of aggregates, entire aggregates, and entire compilation units. If you are porting applications between different platforms, or between 31-bit and 64-bit modes, you need to take into account the differences between alignment settings available in different environments, to prevent possible data corruption and deterioration in performance.

XL C/C++ provides alignment modes and alignment modifiers for specifying data alignment. Using alignment modes, you can set alignment defaults for all data types for a compilation unit or subsection of a compilation unit by specifying a predefined suboption.

Using alignment modifiers, you can set the alignment for specific variables or data types within a compilation unit by specifying the exact number of bytes that should be used for the alignment.

"Using alignment modes" discusses the default alignment modes for all data types on different platforms and addressing models, the suboptions and pragmas that you can use to change or override the defaults, and rules for the alignment modes for simple variables, aggregates, and bit fields.

"Using alignment modifiers" on page 8 discusses the different specifiers, pragmas, and attributes you can use in your source code to override the alignment mode currently in effect, for specific variable declarations. It also provides the rules that govern the precedence of alignment modes and modifiers during compilation.

Using alignment modes

Each data type that is supported by XL C/C++ is aligned along byte boundaries according to platform-specific default alignment modes. The default alignment mode is **zlinux**.

Each of the valid alignment modes is defined in Table 8, which provides the alignment value, in bytes, for scalar variables of all data types. Where there are differences between 31-bit and 64-bit modes, these are indicated. Also, where there are differences between the first (scalar) member of an aggregate and subsequent members of the aggregate, these are indicated.

Table 8. Alignment settings (values given in bytes)

		Alignment setting	
Data type	**Storage**	**zlinux**	**bit_packed**
_Bool (C), bool (C++)	1	1	1
char, signed char, unsigned char	1	1	1
wchar_t (31-bit mode)	2	2	1
wchar_t (64-bit mode)	4	4	1
int, unsigned int	4	4	1
short int, unsigned short int	2	2	1
long int, unsigned long int (31-bit mode)	4	4	1

Table 8. Alignment settings (values given in bytes) (continued)

Data type	Storage	Alignment setting	
		zlinux	**bit_packed**
long int, unsigned long int (64-bit mode)	8	8	1
long long	8	8	1
float	4	4	1
double	8	8	1
long double	16	16	1
pointer (31-bit mode)	4	4	1
pointer (64-bit mode)	8	8	1

If you generate data with an application on one platform and read the data with an application on another platform, it is recommended that you use the **bit_packed** mode, which results in equivalent data alignment on all platforms.

"Alignment of aggregates" discusses the rules for the alignment of entire aggregates and provides examples of aggregate layouts. "Alignment of bit-fields" on page 7 discusses additional rules and considerations for the use and alignment of bit fields and provides an example of bit-packed alignment.

Related information in the *XL C/C++ Compiler Reference*

-fpack-struct (-qalign)

Alignment of aggregates

The data contained in Table 8 on page 5 in "Using alignment modes" on page 5 apply to scalar variables, and variables that are members of aggregates such as structures, unions, and classes. The following rules apply to aggregate variables, namely structures, unions or classes, as a whole (in the absence of any modifiers):

- For all alignment modes, the size of an aggregate is the smallest multiple of its alignment value that can encompass all of the members of the aggregate.
- C Empty aggregates are assigned a size of zero bytes. As a result, two distinct variables might have the same address.
- C++ Empty aggregates are assigned a size of one byte. Note that static data members do not participate in the alignment or size of an aggregate; therefore, a structure or class containing only a single static data member has a size of one byte.
- For all alignment modes, the alignment of an aggregate is equal to the largest alignment value of any of its members. With the exception of packed alignment modes, members whose natural alignment is smaller than that of their aggregate's alignment are padded with empty bytes.
- Aligned aggregates can be nested, and the alignment rules applicable to each nested aggregate are determined by the alignment mode that is in effect when a nested aggregate is declared.

Notes:

- C++ The C++ compiler might generate extra fields for classes that contain base classes or virtual functions. Objects of these types might not conform to the usual mappings for aggregates.

- The alignment of an aggregate must be the same in all compilation units. For example, if the declaration of an aggregate is in a header file and you include that header file into two distinct compilations units, choose the same alignment mode for both compilations units.

For rules on the alignment of aggregates containing bit fields, see "Alignment of bit-fields."

Alignment of bit-fields

You can declare a bit-field as a C `_Bool` C , C++ `bool`, `char`, `signed char`, `unsigned char`, `short`, `unsigned short` C++ , `int`, `unsigned int`, `long`, `unsigned long`, C++ `long long`, or `unsigned long long` C++ data type. The alignment of a bit-field depends on its base type and the compilation mode (31-bit or 64-bit).

C The length of a bit-field cannot exceed the length of its base type. In extended mode, you can use the `sizeof` operator on a bit-field. The `sizeof` operator on a bit-field returns the size of the base type. C

C++ The length of a bit-field can exceed the length of its base type, but the remaining bits are used to pad the field and do not actually store any value. C++

However, alignment rules for aggregates containing bit-fields are different depending on the alignment mode in effect. These rules are described below.

Rules for Linux on z Systems alignment

- Bit-fields are allocated from a bit-field container. The size of this container is determined by the declared type of the bit-field. For example, a `char` bit-field uses an 8-bit container, and an `int` bit-field uses 31 bits. The container must be large enough to contain the bit-field because the bit-field will not be split across containers.
- Containers are aligned in the aggregate as if they start on a natural boundary for that type of container. Bit-fields are not necessarily allocated at the start of the container.
- If a zero-length bit-field is the first member of an aggregate, it has no effect on the alignment of the aggregate and is overlapped by the next data member. If a zero-length bit-field is a non-first member of the aggregate, it pads to the next alignment boundary determined by its base declared type but does not affect the alignment of the aggregate.
- Unnamed bit-fields do not affect the alignment of the aggregate.

Rules for bit-packed alignment

- Bit-fields have an alignment of one byte and are packed with no default padding between bit-fields.
- A zero-length bit-field causes the next member to start at the next byte boundary. If the zero-length bit-field is already at a byte boundary, the next member starts at this boundary. A non-bit-field member that follows a bit-field is aligned on the next byte boundary.

Using alignment modifiers

XL C/C++ also provides alignment modifiers, with which you can exercise even finer-grained control over alignment, at the level of declaration or definition of individual variables or aggregate members. Available modifiers are as follows:

`#pragma pack(...)`

Valid application:
The entire aggregate (as a whole) immediately following the directive.

Effect: Sets the maximum alignment of the members of the aggregate to which it applies, to a specific number of bytes. Also allows a bit-field to cross a container boundary. Used to reduce the effective alignment of the selected aggregate.

For details about **#pragma pack(...)**, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

`__attribute__((aligned(n)))`

Valid application:
As a variable attribute, it applies to a single aggregate (as a whole), namely a structure, union, or class, or it applies to an individual member of an aggregate.[1] As a type attribute, it applies to all aggregates declared of that type. If it is applied to a `typedef` declaration, it applies to all instances of that type.[2]

Effect:
Sets the minimum alignment of the specified variable or variables to a specific number of bytes. Typically used to increase the effective alignment of the selected variables.

Valid values:
n must be a positive power of two, or NIL. NIL can be specified as either `__attribute__((aligned()))` or `__attribute__((aligned))`; this is the same as specifying the maximum system alignment (16 bytes on all UNIX platforms).

`__attribute__((packed))`

Valid application:
As a variable attribute, it applies to simple variables or individual members of an aggregate, namely a structure or class[1]. As a type attribute, it applies to all members of all aggregates declared of that type.

Effect: Sets the maximum alignment of the selected variable or variables, to which it applies, to the smallest possible alignment value, namely one byte for a variable and one bit for a bit field.

Notes:

1. In a comma-separated list of variables in a declaration, if the modifier is placed at the beginning of the declaration, it applies to all the variables in the declaration. Otherwise, it applies only to the variable immediately preceding it.
2. Depending on the placement of the modifier in the declaration of a `struct`, it can apply to the definition of the type, and hence applies to all instances of that type; or it can apply to only a single instance of the type. For details, see the information about type attributes in the *XL C/C++ Language Reference* and the C C++ and C++ C++ language standards.

   Related information in the *XL C/C++ Compiler Reference*

#pragma pack

Related information in the *XL C/C++ Language Reference*

The aligned type attribute (IBM extension)

The packed type attribute (IBM extension)

Type attributes (IBM extension)

The aligned variable attribute (IBM extension)

The packed variable attribute (IBM extension)

Chapter 3. Handling floating-point operations

The following sections provide reference information, portability considerations, and suggested procedures for using compiler options to manage floating-point operations:

- "Floating-point formats"
- "Handling multiply-and-add operations"
- "Handling floating-point constant folding and rounding"
- "Handling floating-point exceptions" on page 13

Floating-point formats

XL C/C++ supports the following binary floating-point formats:

- 32-bit single precision, with an approximate absolute normalized range of 0 and 10^{-38} to 10^{38} and with a precision of about 7 decimal digits
- 64-bit double precision, with an approximate absolute normalized range of 0 and 10^{-308} to 10^{308} and with a precision of about 16 decimal digits
- 128-bit extended precision, with slightly greater range than double-precision values, and with a precision of about 32 decimal digits

The 128-bit extended precision format of XL C/C++ is different from the **binary128** formats that are suggested by the IEEE standard. The IEEE standard suggests that extended formats use more bits in the exponent for greater range and the fraction for higher precision.

It is possible that special numbers, such as NaN, infinity, and negative zero, cannot be represented by the 128-bit extended precision values. Arithmetic operations do not necessarily propagate these numbers in extended precision.

Handling multiply-and-add operations

By default, the compiler generates a single non-IEEE 754 compatible multiply-and-add instruction for binary floating-point expressions, such as *a + b * c*, partly because one instruction is faster than two. Because no rounding occurs between the multiply and add operations, this might also produce a more precise result. However, the increased precision might lead to different results from those obtained in other environments, and might cause *x*y-x*y* to produce a nonzero result. To avoid these issues, you can suppress the generation of multiply-add instructions by using the **-qfloat=nomaf** option.

Related information in the *XL C/C++ Compiler Reference*

-qfloat

Handling floating-point constant folding and rounding

By default, the compiler replaces most operations involving constant operands with their result at compile time. The result of a floating-point operation folded at compile time normally produces the same result as that obtained at execution time, except in the following case:

Expressions like *a* + *b* * *c* are partially or fully evaluated at compile time. The results might be different from those produced at execution time, because *b* * *c* might be rounded before being added to *a*, while the runtime multiply-add instruction does not use any intermediate rounding. To avoid differing the results, suppress the use of multiply-add instructions by specifying **-qfloat=nomaf**.

Related information:

"Handling floating-point exceptions" on page 13

Related information in the *XL C/C++ Compiler Reference*

-qfloat

Matching compile-time and runtime rounding modes

The default rounding mode used at compile time and run time is round-to-nearest, ties to even. If your program changes the rounding mode at run time, the results of a floating-point calculation might be slightly different from those that are obtained at compile time. The following example illustrates this:

```
#include <float.h>
#include <fenv.h>
#include <stdio.h>

int main ( )
{
 volatile double one = 1.f, three = 3.f;  /* volatiles are not folded */
 double one_third;

 one_third = 1. / 3.;  /* folded */
 printf ("1/3 with compile-time rounding = %.17f\n", one_third);

 fesetround (FE_TOWARDZERO);
 one_third = one / three;  /* not folded */

 printf ("1/3 with execution-time rounding to zero = %.17f\n", one_third);

 fesetround (FE_TONEAREST);
 one_third = one / three;  /* not folded */

 printf ("1/3 with execution-time rounding to nearest = %.17f\n", one_third);

 fesetround (FE_UPWARD);
 one_third = one / three;  /* not folded */

 printf ("1/3 with execution-time rounding to +infinity = %.17f\n", one_third);

 fesetround (FE_DOWNWARD);
 one_third = one / three;  /* not folded */

 printf ("1/3 with execution-time rounding to -infinity = %.17f\n", one_third);

 return 0;
 }
```

When compiled with the default options, this code produces the following results:

```
1/3 with compile-time rounding = 0.33333333333333331
1/3 with execution-time rounding to zero = 0.33333333333333331
1/3 with execution-time rounding to nearest   = 0.33333333333333331
1/3 with execution-time rounding to +infinity = 0.33333333333333337
1/3 with execution-time rounding to -infinity = 0.33333333333333331
```

Because the fourth computation changes the rounding mode to round-to-infinity, the results are slightly different from the first computation, which is performed at compile time, using round-to-nearest.

Related information in the *XL C/C++ Compiler Reference*

-qfloat

Handling floating-point exceptions

By default, invalid operations such as division by zero, division by infinity, overflow, and underflow are ignored at run time.

You can add suitable support code to your program to make program execution continue after an exception occurs and to modify the results of operations causing exceptions.

Because, however, floating-point computations involving constants are usually folded at compile time, the potential exceptions that would be produced at run time might not occur.

Related information in the *XL C/C++ Compiler Reference*

-qfloat

Chapter 4. Constructing a library

You can include static and shared libraries in your C and C++ applications.

"Compiling and linking a library" describes how to compile your source files into object files for inclusion in a library, how to link a library into the main program, and how to link one library into another.

"Initializing static objects in libraries (C++)" on page 16 describes how to use priorities to control the order of initialization of objects across multiple files in a C++ application.

Compiling and linking a library

This section describes how to compile your source files into object files for inclusion in a library, how to link a library into the main program, and how to link one library into another.

Related information in the *Getting Started with XL C/C++*

📄 Dynamic and static linking

Compiling a static library

To compile a static library, follow this procedure:

1. Compile each source file to get an object file. For example:
   ```
   xlc -c test.c example.c
   ```
2. Use the ar command to add the generated object files to an archive library file. For example:
   ```
   ar -rv libex.a test.o example.o
   ```

Compiling a shared library

To compile a shared library, follow this procedure:

1. Compile your source files to get an object file. Note that in the case of compiling a shared library, you must use the **-fPIC (-qpic)** compiler option. For example:
   ```
   xlc -fPIC -c foo.c
   ```
2. Use the **-shared (-qmkshrobj)** compiler option to create a shared object from the generated object files. For example:
   ```
   xlc -shared -o libfoo.so foo.o
   ```

   Related information in the *XL C/C++ Compiler Reference*

   📄 -fPIC (-qpic)

   📄 -shared (-qmkshrobj)

Linking a library to an application

You can use the following command string to link a static or shared library to your main program. For example:

```
xlc -o myprogram main.c -Ldirectory1:directory2 [-Rdirectory] -ltest
```

At compile time, you instruct the linker to search for `libtest.so` in the first directory specified by the **-L** option. If `libtest.so` is not found, the linker searches for `libtest.a`. If neither file is found, the search continues with the next directory specified by the **-L** option.

At run time, the runtime linker searches for `libtest.so` in the first directory specified by the **-R** option. If `libtest.so` is not found, the search continues with the next directory specified by the **-R** option. The path specified by the **-R** option can be overridden at run time by the LD_LIBRARY_PATH environment variable.

For additional linkage options, including options that modify the default behavior, see the operating system **ld** documentation .

Related information in the *XL C/C++ Compiler Reference*

-l

-L

-R

Linking a shared library to another shared library

Just as you link modules into an application, you can create dependencies between shared libraries by linking them together. For example:

```
xlc -shared -o mylib.so myfile.o -Ldirectory -Rdirectory -ltest
```

Related information in the *XL C/C++ Compiler Reference*

-shared (-qmkshrobj)

-R

-L

Initializing static objects in libraries (C++)

The C++ language definition specifies that all non-local objects with constructors from all the files included in the program must be properly constructed before the `main` function in a C++ program is executed. Although the language definition specifies the order of initialization for these objects within a file (which follows the order in which they are declared), it does not specify the order of initialization for these objects across files and libraries. You might want to specify the initialization order of static objects declared in various files and libraries in your program.

To specify an initialization order for objects, you assign relative priority numbers to objects. The mechanisms by which you can specify priorities for entire files or objects within files are discussed in “Assigning priorities to objects.” The mechanisms by which you can control the initialization order of objects across modules are discussed in “Order of object initialization across libraries” on page 17.

Assigning priorities to objects

You can assign a priority level number to objects and files within a single library using the following approaches. The objects will be initialized at run time according to the order of priority level. In addition, because modules are loaded and objects are initialized differently on different platforms, you can choose an approach that fits the platform better.

Set the priority level for an entire file

To use this approach, specify the **-qpriority** compiler option during compilation. By default, all objects within a single file are assigned the same priority level; they are initialized in the order in which they are declared, and they are terminated in reverse declaration order.

Set the priority level for individual objects

To use this approach, use `init_priority` variable attributes in the source files. The `init_priority` attribute can be applied to objects in any declaration order. On Linux, the objects are initialized according to their priority and terminated in reverse priority across compilation units.

Using priority numbers

Priority numbers can range from 101 to 65535. The smallest priority number that you can specify, 101, is initialized first. The largest priority number, 65535, is initialized last. If you do not specify a priority level, the default priority is 65535.

The examples below show how to specify the priority of objects within a single file and across two files. "Order of object initialization across libraries" provides detailed information on the order of initialization of objects.

Related information in the *XL C/C++ Compiler Reference*

-qpriority

-shared (-qmkshrobj)

Related information in the *XL C/C++ Language Reference*

The init_priority variable attribute

Order of object initialization across libraries

Each static library and shared library is loaded and initialized at runtime in reverse link order, once all of its dependencies have been loaded and initialized. Link order is the order in which each library was listed on the command line during linking into the main application. For example, if library A calls library B, library B is loaded before library A.

As each module is loaded, objects are initialized in order of priority according to the rules outlined in "Assigning priorities to objects" on page 16. If objects do not have priorities assigned or have the same priorities, object files are initialized in reverse link order — where link order is the order in which the files were given on the command line during linking into the library — and the objects within the files are initialized according to their declaration order. Objects are terminated in reverse order of their construction.

Example of object initialization across libraries

In this example, the following modules are used:

- `main.out`, the executable containing the main function
- `libS1` and `libS2`, two shared libraries
- `libS3` and `libS4`, two shared libraries that are dependencies of `libS1`
- `libS5` and `libS6`, two shared libraries that are dependencies of `libS2`

The source files are compiled into object files with the following commands. You must use the **-fPIC (-qpic)** option to compile what is to be included in a shared library.

```
xlC -qpriority=101 -c fileA.C -o fileA.o
xlC -qpriority=150 -c fileB.C -o fileB.o
xlC -c fileC.C -o fileC.o
xlC -c fileD.C -o fileD.o
xlC -c fileE.C -o fileE.o
xlC -c fileF.C -o fileF.o
xlC -qpriority=300 -c fileG.C -o fileG.o
xlC -qpriority=200 -c fileH.C -o fileH.o
xlC -qpriority=500 -c fileI.C -o fileI.o
xlC -c fileJ.C -o fileJ.o
xlC -c fileK.C -o fileK.o
xlC -qpriority=600 -c fileL.C -o fileL.o
```

The dependent libraries are created with the following commands:

```
xlC -shared -o libS3.so fileE.o fileF.o
xlC -shared -o libS4.so fileG.o fileH.o
xlC -shared -o libS5.so fileI.o fileJ.o
xlC -shared -o libS6.so fileK.o fileL.o
```

The dependent libraries are linked with their parent libraries by using the following commands:

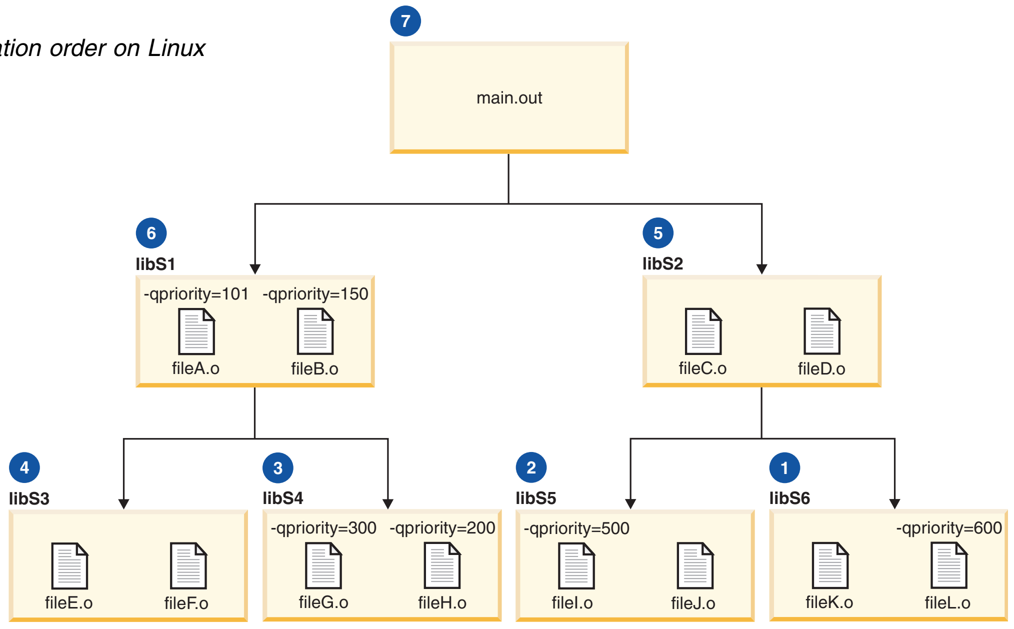
```
xlC -shared -o libS1.so fileA.o fileB.o -L. -R. -lS3 -lS4
xlC -shared -o libS2.so fileC.o fileD.o -L. -R. -lS5 -lS6
```

The parent libraries are linked with the main program with the following command:

```
xlC main.C -o main.out -L. -R. -lS1 -lS2
```

The following diagram shows the initialization order of the shared libraries.

Figure 1. Object initialization order on Linux



Objects are initialized as follows:

Sequence	Object	Priority value	Comment
1	libS6	n/a	`libS2` was entered last on the command line when linked with `main`, and so is initialized before `libS1`. However, `libS5` and `libS6` are dependencies of `libS2`, so they are initialized first. Since it was entered last on the command line when linked to create `libS2`, `libS6` is initialized first. The objects in this library are initialized according to their priority.
2	fileL	600	The objects in `fileL` are initialized next (lowest priority number in this module).
3	fileK	65535	The objects in `fileK` are initialized next (next priority number in this module (default priority of 65535)).
4	libS5	n/a	`libS5` was entered before `libS6` on the command line when linked with `libS2`, so it is initialized next. The objects in this library are initialized according to their priority.
5	fileI	500	The objects in `fileI` are initialized next (lowest priority number in this module).
6	fileJ	65535	The objects in `fileJ` are initialized next (next priority number in this module (default priority of 65535)).
7	libS4	n/a	`libS4` is a dependency of `libS1` and was entered last on the command line when linked to create `libS1`, so it is initialized next. The objects in this library are initialized according to their priority.
8	fileH	200	The objects in `fileH` are initialized next (lowest priority number in this module).
9	fileG	300	The objects in `fileG` are initialized next (next priority number in this module).
10	libS3	n/a	`libS3` is a dependency of `libS1` and was entered first on the command line during the linking with `libS1`, so it is initialized next. The objects in this library are initialized according to their priority.
11	fileF	65535	Both `fileF` and `fileE` are assigned a default priority of 65535. However, because `fileF` was listed last on the command line when the object files were linked into `libS3`, `fileF` is initialized first.
12	fileE	65535	Initialized next.
13	libS2	n/a	`libS2` is initialized next. The objects in this library are initialized according to their priority.
14	fileD	65535	Both `fileD` and `fileC` are assigned a default priority of 65535. However, because `fileD` was listed last on the command line when the object files were linked into `libS2`, `fileD` is initialized first.
15	fileC	65535	Initialized next.
16	libS1		`libS1` is initialized next. The objects in this library are initialized according to their priority.
17	fileA	101	The objects in `fileA` are initialized next (lowest priority number in this module).
18	fileB	150	The objects in `fileB` are initialized next (next priority number in this module).

Sequence	Object	Priority value	Comment
19	main.out	n/a	Initialized last. The objects in `main.out` are initialized according to their priority.

Related information in the *XL C/C++ Compiler Reference*

-shared (-qmkshrobj)

-W

Chapter 5. Optimizing your applications

The XL compilers enable development of high performance 31-bit and 64-bit applications by offering a comprehensive set of performance enhancing techniques that exploit the multilayered System z® architecture. These performance advantages depend on good programming techniques, thorough testing and debugging, followed by optimization and tuning.

Distinguishing between optimization and tuning

You can use optimization and tuning separately or in combination to increase the performance of your application. Understanding the difference between them is the first step in understanding how the different levels, settings, and techniques can increase performance.

Optimization

Optimization is a compiler-driven process that searches for opportunities to restructure your source code and give your application better overall performance at run time, without significantly impacting development time. The XL compiler optimization suite, which you control using compiler options and directives, performs best on well-written source code that has already been through a thorough debugging and testing process. These optimization transformations can bring the following benefits:

- Reduce the number of instructions that your application executes to perform critical operations.
- Improve memory subsystem usage.

Each basic optimization technique can result in a performance benefit, although not all optimizations can benefit all applications. Consult the “Steps in the optimization process” on page 22 for an overview of the common sequence of steps that you can use to increase the performance of your application.

Tuning

Tuning is a user-driven process where you experiment with changes, for example to source code or compiler options, to make the compiler better optimize your program. While optimization applies general transformations designed to improve the performance of any application in any supported environment, tuning offers you opportunities to adjust specific characteristics or target execution environments of your application to improve its performance. Even at low optimization levels, tuning for your application and target architecture can have a positive impact on performance. With proper tuning, the compiler can make the following improvements:

- Select more efficient machine instructions.
- Generate instruction sequences that are more relevant to your application.
- Select from more focussed optimizations to improve your code.

For instructions, see “Tuning for your system architecture” on page 25.

Steps in the optimization process

When you begin the optimization process, consider that not all optimization techniques suit all applications. Trade-offs sometimes occur between an increase in compile time, a reduction in debugging capability, and the improvements that optimization can provide.

Learning about and experimenting with different optimization techniques can help you strike the right balance for your XL compiler applications while achieving the best possible performance. Also, though it is unnecessary to hand-optimize your code, compiler-friendly programming can be extremely beneficial to the optimization process. Unusual constructs can obscure the characteristics of your application and make performance optimization difficult. Use the steps in this section as a guide for optimizing your application.

1. The Basic optimization step begins your optimization processes at levels 0 and 2.
2. The Advanced optimization step exposes your application to more intense optimizations at levels 3.
3. The Using high-order loop analysis and transformations step can help you limit loop execution time.
4. The Using interprocedural analysis step can optimize your entire application at once.
5. The Using profile-directed feedback step focuses optimizations on specific characteristics of your application.
6. The Debugging optimized code step can help you identify issues and problems that can occur with optimized code.

Basic optimization

The XL compiler supports several levels of optimization, with each option level building on the levels below through increasingly aggressive transformations and consequently using more machine resources.

Ensure that your application compiles and executes properly at low optimization levels before you try more aggressive optimizations. This topic discusses two optimizations levels, listed with complementary options in Table 9. The table also includes a column for compiler options that can have a performance benefit at that optimization level for some applications.

Table 9. Basic optimizations

Optimization level	Additional options implied by default	Complementary options	Other options with possible benefits
-O0	None	**-march**	None
-O2	None	**-march** **-mtune**	**-qhot=level=0**

Optimizing at level 0

Benefits at level 0

- Provides minimal performance improvement with minimal impact on machine resources
- Exposes some source code problems that can be helpful in the debugging process

Begin your optimization process at **-O0**, which the compiler already specifies by default. This level performs basic analytical optimization by removing obviously redundant code, and it can result in better compile time. It also ensures your code is algorithmically correct so you can move forward to more complex optimizations. **-O0** also includes some redundant instruction elimination and constant folding. Optimizing at this level accurately preserves all debugging information and can expose problems in existing code, such as uninitialized variables and bad casting.

Additionally, specifying **-march** at this level targets your application for a particular machine and can significantly improve performance by ensuring that your application takes advantage of all applicable architectural benefits.

Related information in the *XL C/C++ Compiler Reference*

-march

Optimizing at level 2

Benefits at level 2

- Eliminates redundant code
- Performs basic loop optimization
- Structures code to take advantage of **-march** and **-mtune** settings

After you successfully compile, execute, and debug your application using **-O0**, recompiling at **-O2** opens your application to a set of comprehensive low-level transformations that apply to subprogram or compilation unit scopes and can include some inlining. Optimizations at **-O2** attain a relative balance between increasing performance while limiting the impact on compilation time and system resources.

▶ C In C, compile with **-qlibansi** unless your application defines functions with names identical to those of library functions. If you encounter problems with **-O2**, consider using **-qalias=noansi** rather than turning off optimization.

Also, ensure that pointers in your C code follow these type restrictions:
- Generic pointers can be `char*` or `void*`.
- Mark all shared variables and pointers to shared variables `volatile`.

C ◀

Starting to tune at O2

Choosing the right hardware architecture target or family of targets becomes even more important at **-O2** and higher. By targeting the proper hardware, the optimizer can make the best use of the available hardware facilities. If you choose a family of hardware targets, the **-mtune** option can direct the compiler to emit code that is consistent with the architecture choice and that can execute optimally on the chosen tuning hardware target. With this option, you can compile for a general set of targets and have the code run best on a particular target.

For details on the **-march** and **-mtune** options, see "Tuning for your system architecture" on page 25.

The **-O2** option can perform a number of additional optimizations as follows:
- Common subexpression elimination: Eliminates redundant instructions
- Constant propagation: Evaluates constant expressions at compile time

- Dead code elimination: Eliminates instructions that a particular control flow does not reach or that generate an unused result
- Dead store elimination: Eliminates unnecessary variable assignments
- Global register allocation: Globally assigns user variables to registers
- Value numbering: Simplifies algebraic expressions by eliminating redundant computations
- Instruction scheduling for the target machine
- Loop unrolling and software pipelining
- Method inlining: Inlines some methods into the calling method
- Moving loop-invariant code out of loops
- Simplifying control flow
- Strength reduction and effective use of addressing modes
- Widening: Merges adjacent load/stores and other operations
- Pointer aliasing improvements to enhance other optimizations

Even with **-O2** optimizations, some useful information about your source code is made available to the debugger if you specify **-g**. Using a higher **-g** level increases the information provided to the debugger but reduces the optimization that can be done. Conversely, higher optimization levels can transform code to an extent to which debugging information is no longer accurate.

Advanced optimization

Higher optimization levels can have a tremendous impact on performance, but some trade-offs can occur in terms of code size, compile time, resource requirements, and numeric or algorithmic precision.

After applying "Basic optimization" on page 22 and successfully compiling and executing your application, you can apply more powerful optimization tools. The XL compiler optimization portfolio includes many options for directing advanced optimization, and the transformations that your application undergoes are largely under your control. The discussion of the optimization level in Table 10 includes information on the performance benefits and the possible trade-offs and information on how you can help guide the optimizer to find the best solutions for your application.

Table 10. Advanced optimizations

Optimization Level	Additional options implied	Complementary options	Options with possible benefits
-O3	**-qhot=level=0**	**-march** **-mtune**	**-qpdf**

Optimizing at level 3

Benefits at level 3

- In-depth memory access analysis
- Better loop scheduling
- High-order loop analysis and transformations (**-qhot=level=0**)
- Inlining of small procedures within a compilation unit by default
- Eliminating implicit compile-time memory usage limits

Specifying **-O3** initiates more intense low-level transformations that remove many of the limitations present at **-O2**. Additionally, optimizations encompass larger program regions and attempt more in-depth analysis. Although not all applications contain opportunities for the optimizer to provide a measurable increase in performance, most applications can benefit from this type of analysis.

Potential trade-offs at level 3

With the in-depth analysis of **-O3** comes a trade-off in terms of compilation time and memory resources. This typically involves precision trade-offs as follows:

- Reordering of floating-point computations
- Reordering or elimination of possible exceptions, such as division by zero or overflow
- Using alternative calculations that might give slightly less precise results or not handle infinities or NaNs in the same way

For information about the **-O** level syntax, see "-O -qoptimize" in the *XL C/C++ Compiler Reference* .

Increasing -qhot at level 3

At **-O3**, the optimization includes minimal **-qhot** loop transformations at **level=0** to increase performance. To further increase your performance benefit from **-qhot**, increase the optimization aggressiveness by increasing the optimization level of **-qhot**. Try specifying **-qhot** without any suboptions or **-qhot=level=1**.

For more information about **-qhot**, see "Using high-order loop analysis and transformations" on page 27.

Conversely, if the application does not use loops processing arrays, which **-qhot** improves, you can improve compile speed significantly, usually with minimal performance loss by using **-qnohot** after **-O3**.

Tuning for your system architecture

You can instruct the compiler to generate code for optimal execution on a given microprocessor or architecture family. By selecting appropriate target machine options, you can optimize to suit the broadest possible selection of target processors, a range of processors within a given family of processor architectures, or a specific processor.

The following table lists the optimization options that affect individual aspects of the target machine. Using a predefined optimization level sets default values for these individual options.

Table 11. Target machine options

Option	Behavior
-m31	Generates code for a 31-bit (4 byte integer / 4 byte long / 4 byte pointer) addressing model (31-bit execution mode).
-m64	Generates code for a 64-bit (4 byte integer / 8 byte long / 8 byte pointer) addressing model (64-bit execution mode). This is the default setting.
-march	Selects a family of processor architectures for which instruction code should be generated. See "Getting the most out of target machine options" on page 26 for more information about this option.

Table 11. Target machine options (continued)

Option	Behavior
-mtune	Biases optimization toward execution on a given microprocessor, without implying anything about the instruction set architecture to be used as a target. See "Getting the most out of target machine options" for more information about this option.

For a complete list of valid hardware related suboptions and combinations of suboptions, see the following information in the *XL C/C++ Compiler Reference*.

- *Acceptable -march and -mtune combinations* in -mtune (-qtune)
- Specifying compiler options for architecture-specific compilation

Related information in the *XL C/C++ Compiler Reference*

-m31, -m64 (-q31, -q64)

-march

-qipa

Getting the most out of target machine options

Using the -march (-qarch) option

Use the **-march (-qarch)** compiler option to generate instructions that are optimized for a specific machine architecture. For example, if you want to generate an object code that contains instructions optimized for the z196 architectures, use **-march=z196**. If your application runs on the same machine on which you compile it, use the **-qarch=auto** option, which automatically detects the specific architecture of the compiling machine and generates code to take advantage of instructions available only on that machine (or on a system that supports the equivalent processor architecture). Otherwise, use the **-march (-qarch)** option to specify the smallest possible family of the machines that can run your code reasonably well.

Using the -mtune (-qtune) option

Use the **-mtune (-qtune)** compiler option to control the scheduling of instructions that are optimized for your machine architecture. If you specify a particular architecture with **-march (-qarch)**, **-mtune (-qtune)** automatically selects the suboption that generates instruction sequences with the best performance for that architecture. If you specify a *group* of architectures with **-qarch**, compiling with **-qtune=auto** generates code that runs on all of the architectures in the specified group, but the instruction sequences are those with the best performance on the architecture of the compiling machine.

Try to specify with **-mtune (-qtune)** the particular architecture that the compiler should target for best performance but still allow execution of the produced object file on all architectures specified in the **-march (-qarch)** option. For information about the valid combinations of **-march (-qarch)** and **-mtune (-qtune)** settings, see *Acceptable -march and -mtune combinations* in the **-mtune (-qtune)** section of the *XL C/C++ Compiler Reference*.

Related information in the *XL C/C++ Compiler Reference*

-march

-mtune

Using high-order loop analysis and transformations

High-order transformations are optimizations that specifically improve the performance of loops through techniques such as interchange, fusion, and unrolling.

The goals of these loop optimizations include:

- Reducing the costs of memory access through the effective use of caches and address translation look-aside buffers
- Overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware
- Improving the utilization of microprocessor resources through reordering and balancing the usage of instructions with complementary resource requirements
- Generating SIMD vector instructions to offer better program performance when **-ftree-vectorize** or **-qsimd=auto** is specified
- Generating calls to vector math library functions

To enable high-order loop analysis and transformations, use the **-qhot** option, which implies an optimization level of **-O2**.

Table 12. -qhot suboptions

Suboption	Behavior
level=0	Instructs the compiler to perform a subset of high-order transformations that enhance performance by improving data locality. This suboption implies **-qhot=noarraypad**. This level is automatically enabled if you compile with **-O3**.
level=1	This is the default suboption if you specify **-qhot** with no suboptions.
arraypad	Instructs the compiler to pad any arrays where it infers there might be a benefit and to pad by whatever amount it chooses.

Related information in the *XL C/C++ Compiler Reference*

-march

Getting the most out of -qhot

Here are some suggestions for using **-qhot**:

- Try using **-qhot** along with **-O3** for all of your code. It is designed to have a neutral effect when no opportunities for transformation exist. However, it increases compilation time and might have little benefit if the program has no loop processing vectors or arrays. In this case, using **-O3 -qnohot** might be better.
- If you encounter unacceptably long compilation time (this can happen with complex loop nests), try **-qhot=level=0** or **-qnohot**.
- You can compile some source files with the **-qhot** option and some files without the **-qhot** option, allowing the compiler to improve only the parts of your code that need optimization.
- Use **-qreport** along with **-qhot** to generate a loop transformation listing. The listing file identifies how loops are transformed in a section marked `LOOP TRANSFORMATION SECTION`. Use the listing information as feedback about how the loops in your program are being transformed. Based on this information, you might want to adjust your code so that the compiler can transform loops more effectively. For example, you can use this section of the listing to identify non-stride-one references that might prevent loop vectorization.

- Use **-qreport** along with **-qhot** or any optimization option that implies **-qhot** to generate information about nested loops in the `LOOP TRANSFORMATION SECTION` of the listing file.

Related information in the *XL C/C++ Compiler Reference*

-qhot

Generating vector instructions

When you run the compiler on the IBM z13 models and target a Linux distribution that has vector support, you can specify **-ftree-vectorize (-qsimd)** to enable the compiler to transform code into vector instructions.

These vector instructions take advantage of the Vector Facility for z/Architecture to execute several operations in parallel. This transformation mostly applies to the loops that iterate over contiguous array data and perform calculations on each element.

You can specify **-qsimd=auto** with **-O3** or **-O3 -qhot** to expose additional optimization opportunities.

You can use the `#pragma nosimd` directive to disable the transformation of a particular loop into vector instructions.

Related information in the *XL C/C++ Compiler Reference*

-qhot

Using interprocedural analysis

Interprocedural analysis (IPA) enables the compiler to optimize across different files (whole-program analysis), and it can result in significant performance improvements.

You can specify interprocedural analysis on the compilation step only or on both compilation and link steps in whole program mode. Whole program mode expands the scope of optimization to an entire program unit, which can be an executable or a shared object. As IPA can significantly increase compilation time, you should limit using IPA to the final performance tuning stage of development.

You can enable IPA by specifying the **-qipa** option. The most commonly used suboptions and their effects are described in the following table. The full set of suboptions and syntax is described in the **-qipa** section of the *XL C/C++ Compiler Reference*.

The steps to use IPA are as follows:

1. Do preliminary performance analysis and tuning before compiling with the **-qipa** option, because the IPA analysis uses a two-pass mechanism that increases compilation time and link time. You can reduce some compilation and link overhead by using the **-qipa=noobject** option.
2. Specify the **-qipa** option on both the compilation and the link steps of the entire application, or as much of it as possible. Use suboptions to indicate assumptions to be made about parts of the program *not* compiled with **-qipa**.

Table 13. Commonly used **-qipa** *suboptions*

Suboption	**Behavior**
level=0	Program partitioning and simple interprocedural optimization, which consists of: • Automatic recognition of standard libraries. • Localization of statically bound variables and procedures. • Partitioning and layout of procedures according to their calling relationships. (Procedures that call each other frequently are located closer together in memory.) • Expansion of scope for some optimizations, notably register allocation.
level=1	Inlining and global data mapping. Specifically: • Procedure inlining. • Partitioning and layout of static data according to reference affinity. (Data that is frequently referenced together will be located closer together in memory.) This is the default level if you do not specify any suboptions with the **-qipa** option.
level=2	Global alias analysis, specialization, interprocedural data flow: • Whole-program alias analysis. This level includes the disambiguation of pointer dereferences and indirect function calls, and the refinement of information about the side effects of a function call. • Intensive intraprocedural optimizations. This can take the form of value numbering, code propagation and simplification, moving code into conditions or out of loops, and elimination of redundancy. • Interprocedural constant propagation, dead code elimination, pointer analysis, code motion across functions, and interprocedural strength reduction. • Procedure specialization (cloning). • Whole program data reorganization.
inline=*suboptions*	Provides precise control over function inlining.
fine_tuning	Other values for **-qipa** provide the ability to specify the behavior of library code, tune program partitioning, read commands from a file, and so on.

Note:

The XL C/C++ for Linux on z Systems compilers provide backwards compatibility with IPA objects that are created by earlier compiler versions. If IPA object files that are compiled with newer versions of compilers are linked by an earlier version, errors occur during the link step. For example, if IPA object file `a.o` is compiled by XL C/C++ for Linux on z Systems, V1.2 and is to be linked with IPA object file `b.o` that is compiled by XL C/C++ for Linux on z Systems, V1.1, then you must use a compiler whose version is XL C/C++ for Linux on z Systems, V1.2 or later.

Related information in the *XL C/C++ Compiler Reference*

-qipa

Getting the most from -qipa

It is not necessary to compile everything with **-qipa**, but try to apply it to as much of your program as possible. Here are some suggestions:

- Specify the **-qipa** option on both the compile and the link steps of the entire application. Although you can also use **-qipa** with libraries, shared objects, and executable files, be sure to use **-qipa** to compile the main and exported functions.
- When compiling and linking separately, use **-qipa=noobject** on the compile step for faster compilation.
- When specifying optimization options in a makefile, use the compiler driver (**xlC**) to link with all the compiler options on the link step included.
- As IPA can generate significantly larger object files than traditional compilations, ensure that there is enough space in the /tmp directory (at least 200 MB). You can use the TMPDIR environment variable to specify a directory with sufficient free space.
- Try varying the **level** suboption if link time is too long. Compiling with **-qipa=level=0** can still be very beneficial for little additional link time.
- Use **-qipa=list=long** to generate a report of functions that were previously inlined. If too few or too many functions are inlined, consider using **-finline-functions (-qinline)** or **-qnoinline**. To control the inlining of specific functions, use **-qinline+***function_name* or **-qinline-***function_name*.
- To generate data reorganization information in the listing file, specify the optimization level **-qipa=level=2** together with **-qreport**. During the IPA link pass, the data reorganization messages for program variable data will be produced to the data reorganization section of the listing file with the label `DATA REORGANIZATION SECTION`. Reorganizations include array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.

Note: While IPA's interprocedural optimizations can significantly improve performance of a program, they can also cause incorrect but previously functioning programs to fail. Here are examples of programming practices that can work by accident without aggressive optimization but are exposed with IPA:
- Relying on the allocation order or location of automatic variables, such as taking the address of an automatic variable and then later comparing it with the address of another local variable to determine the growth direction of a stack. The C language does not guarantee where an automatic variable is allocated, or its position relative to other automatic variables. Do not compile such a function with IPA.
- Accessing a pointer that is either invalid or beyond an array's bounds. Because IPA can reorganize global data structures, a wayward pointer that might have previously modified unused memory might now conflict with user-allocated storage.
- Dereferencing a pointer that has been cast to an incompatible type.

Related information in the *XL C/C++ Compiler Reference*

-finline-functions

-qlist

-qipa

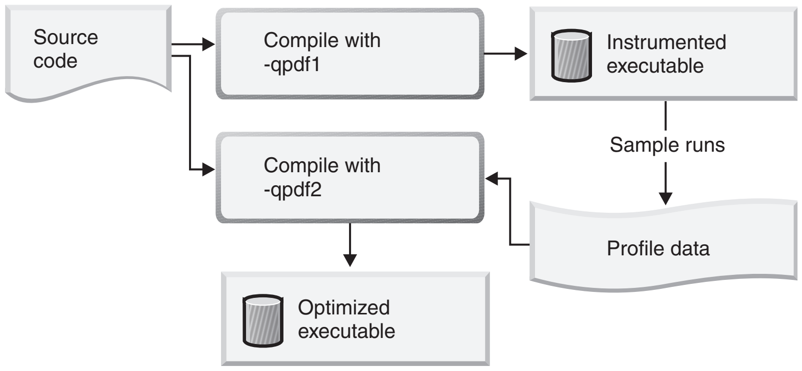
Using profile-directed feedback

You can use profile-directed feedback (PDF) to tune the performance of your application for a typical usage scenario. The compiler optimizes the application based on an analysis of how often branches are taken and blocks of code are run.

Use the PDF process as one of the last steps of optimization before putting the application into production. Optimization at all levels from **-O2** up can benefit from PDF. Other optimizations such as the **-qipa** option can also benefit from PDF process.

The following diagram illustrates the PDF process.

Figure 2. Profile-directed feedback



To use the PDF process to optimize your application, follow these steps:

1. Compile some or all of the source files in a program with the **-qpdf1** option. You must specify at least the **-O2** optimization level.

   Notes:

   - A PDF map file is generated at this step. It is used for the **showpdf** utility to display part of the profiling information in text or XML format. For details, see "Viewing profiling information with showpdf" on page 34. If you do not need to view the profiling information, specify the **-qnoshowpdf** option at this step so that the PDF map file is not generated. For details of **-qnoshowpdf**, see **-qshowpdf** in the *XL C/C++ Compiler Reference*.
   - You do not have to compile all of the files of the programs with the **-qpdf1** option. In a large application, you can concentrate on those areas of the code that can benefit most from the optimization.
   - When any level of option **-qipa** is in effect, and you specify the **-qpdf1** option at the link step but not at the compile step, the compiler issues a warning message. The message indicates that you must recompile your program to get all the profiling information.

   Restriction: When you run an application that is compiled with **-qpdf1**, you must end the application using normal methods, including reaching the end of the execution for the main function and calling the exit() function in libc (stdlib.h) for C/C++ programs. System calls `exit()`, `_Exit()`, and `abort()` are considered abnormal termination methods and are not supported. Using abnormal program termination might result in incomplete instrumentation data generated by using the PDF file or PDF data not being generated at all.

2. Run the resulting application with a typical data set. When the application exits, profile information is written to one or more PDF files. You can train the resulting application multiple times with different data sets. The profiling information is accumulated to provide a count of how often branches are taken

and blocks of code are run, based on the input data used. This step is called the PDF training step. By default, the PDF file is named `._pdf`, and it is placed in the current working directory or the directory specified by the PDFDIR environment variable. If the PDFDIR environment variable is set but the specified directory does not exist, the compiler issues a warning message. To override the defaults, use the **-qpdf1=pdfname** or **-qpdf1=exename** option.

If you recompile your program with the **-qpdf1** option, the compiler removes the existing PDF file or files whose names and locations are the same as the file or files that will be created in the training step before generating a new application.

Notes:

- When you compile your program with the **-qpdf1** or **-qpdf2** option, by default, the **-qipa** option is also invoked with **level=0**.
- To avoid wasting compile and run time, make sure that the PDFDIR environment variable is set to an absolute path. Otherwise, you might run the application from a wrong directory, and the compiler cannot locate the profiling information files. When it happens, the program might not be optimized correctly or might be stopped by a segmentation fault. A segmentation fault might also happen if you change the value of the PDFDIR environment variable and run the application before the PDF process finishes.
- Avoid using atypical data. Otherwise, it might distort the analysis of infrequently executed code paths.

3. If you have several PDF files, use the **mergepdf** utility to combine these PDF files into one PDF file. For example, if you produce three PDF files that represent usage patterns that occur 53%, 32%, and 15% of the time respectively, you can use this command:

```
mergepdf -r 53 file_path1  -r 32 file_path2  -r 15 file_path3 -o file_path4
```

where *file_path1*, *file_path2*, and *file_path3* specify the directories and names of the PDF files that are to be merged, and *file_path4* specifies the directory and name of the output PDF file.

Notes:

- Avoid mixing the PDF files created by different versions or PTF levels of the XL C/C++ compiler.
- You cannot edit PDF files that are generated by the resulting application. Otherwise, the performance or function of the generated executable application might be affected.

4. Recompile your program using the same compiler options as before, but change **-qpdf1** to **-qpdf2**. In this second compilation, the accumulated profiling information is used to fine-tune the optimizations. The resulting program contains no profiling overhead and runs at full speed.

It is recommended that you use the **-qpdf2** option to link the object files that are created during the **-qpdf1** phase without recompiling your program. Using this approach, you can save considerable compilation time and achieve the same optimization result as if you had recompiled your program during the **-qpdf2** phase.

Notes:

- If the compiler cannot read any PDF files in this step, the compiler issues error message 1586-401 but continues the compilation.

- You are highly recommended to use the same optimization level at all compilation steps for a particular program. Otherwise, the PDF process cannot optimize your program correctly and might even slow it down. All compiler settings that affect optimization must be the same, including any supplied by configuration files.
- You can modify your source code and use the **-qpdf1** and **-qpdf2** options to compile your program. Old profiling information can still be preserved and used during the second stage of the PDF process. The compiler issues a list of warnings but the compilation does not stop. An information message is also issued with a number in the range of 0 - 100 to indicate how outdated the old profiling information is.
- When any level of option **-qipa** is in effect, and you specify the **-qpdf2** option at the link step but not at the compile step, the compiler issues a warning message. The message indicates that you must recompile your program to get all the profiling information.
- When using the **-qreport** option with the **-qpdf2** option, you can get additional information in your listing file to help you tune your program. This information is written to the `PDF Report` section.

5. If you want to erase the PDF information, use the **cleanpdf** utility.

Examples

The following example demonstrates that you can concentrate on compiling with **-qpdf1** only the code that can benefit most from the optimization, instead of compiling all the code with the **-qpdf1** option:

```
#Set the PDFDIR variable
export PDFDIR=$HOME/project_dir

#Compile most of the files with -qpdf1
xlc -qpdf1 -O3 -c file1.c file2.c file3.c

#This file does not need optimization
xlc -c file4.c

#Non-PDF object files such as file4.o can be linked
xlc -qpdf1 -O3 file1.o file2.o file3.o file4.o

#Run several times with different input data
./a.out < polar_orbit.data
./a.out < elliptical_orbit.data
./a.out < geosynchronous_orbit.data

#Link all the object files into the final application
xlc -qpdf2 -O3 file1.o file2.o file3.o file4.o
```

The following example bypasses recompiling the source with the **-qpdf2** option:

```
#Compile source with -qpdf1
xlc -c -qpdf1 -O3 file1.c file2.c

#Link object files
xlc -qpdf1 -O3 file1.o file2.o

#Run with one set of input data
./a.out < sample.data

#Link object files
xlc -qpdf2 -O3 file1.o file2.o
```

Related information in the *XL C/C++ Compiler Reference*

-qpdf1, -qpdf2

-O, -qoptimize

Runtime environment variables

Viewing profiling information with showpdf

With the **showpdf** utility, you can view the following types of profiling information that is gathered from your application:

- Block-counter profiling
- Call-counter profiling
- Value profiling

Syntax

```
►►──showpdf──┬────────┬──┬──────────────┬──┬─────────────────┬──────────────────►◄
             └─pdfdir─┘  └─-f──pdfname─┘  └─-m──pdfmapdir─┘
```

Parameters

pdfdir
: Is the directory that contains the profile-directed feedback (PDF) file. If the PDFDIR environment variable is not changed after the **-qpdf1** phase, the PDF map file is also contained in this directory. If this parameter is not specified, the compiler uses the value of the PDFDIR environment variable as the name of the directory.

pdfname
: Is the name of the PDF file. If this parameter is not specified, the compiler uses `._pdf` as the name of the PDF file.

pdfmapdir
: Is the directory that contains the PDF map file. If this parameter is not specified, the compiler uses the value of the PDFDIR environment variable as the name of the directory.

Usage

A PDF map file that contains static information is generated during the **-qpdf1** phase, and a PDF file is generated during the execution of the resulting application. The **showpdf** utility needs both the PDF and PDF map files to display PDF information.

By default, the PDF file is named `._pdf`, and the PDF map file is named `._pdf_map`. If the PDFDIR environment variable is set, the compiler places the PDF and PDF map files in the directory specified by PDFDIR. Otherwise, the compiler places these files in the current working directory. If the PDFDIR environment variable is set but the specified directory does not exist, the compiler issues a warning message. To override the defaults, use the **-qpdf1=pdfname** option to specify the paths and names for the PDF and PDF map files. For example, if you specify the **-qpdf1=pdfname=/home/joe/func** option, the resulting PDF file is named `func`, and the PDF map file is named `func_map`. Both of the files are placed in the `/home/joe` directory.

If the PDFDIR environment variable is changed between the **-qpdf1** phase and the execution of the resulting application, the PDF and PDF map files are generated in separate directories. In this case, you must specify the directories for both of these files to the **showpdf** utility.

Notes:

- PDF and PDF map files must be generated from the same compilation instance. Otherwise, the compiler issues an error.
- PDF and PDF map files must be generated during the same profiling process. This means that you cannot mix and match PDF and PDF map files that are generated from different profiling processes.
- You must use the same version and PTF level of the compiler to generate the PDF file and the PDF map file.
- The **showpdf** utility accepts only PDF files that are in binary format.
- You can use the PDF_WL_ID environment variable to distinguish the multiple sets of PDF counters that are generated by multiple training runs of the user program.

The following example shows how to use the **showpdf** utility to view the profiling information for a Hello World application:

The source for the program file `hello.c` is as follows:

```
#include <stdio.h>
void HelloWorld()
{
   printf("Hello World");
}
main()
{
   HelloWorld();
   return 0;
}
```

1. Compile the source file.

   ```
   xlc -qpdf1 -O hello.c
   ```

2. Run the resulting executable program **a.out** using a typical data set or several typical data sets.
3. If you want to view the profiling information for the executable file in text format, run the **showpdf** utility without any parameters.

   ```
   showpdf
   ```

   The result is as follows:

   ```
   HelloWorld(67):  1 (hello.c)

   Call Counters:
   4 | 1  printf(69)

   Call coverage = 100% ( 1/1 )

   Block Counters:
   2-4 | 1
   5 |
   5 | 1

   Block coverage = 100% ( 2/2 )

   -----------------------------------
   main(68):  1 (hello.c)
   ```

```
Call Counters:
8 | 1  HelloWorld(67)

Call coverage = 100% ( 1/1 )

Block Counters:
6-9 | 1
10 |

Block coverage = 100% ( 1/1 )

Total Call coverage = 100% ( 2/2 )
Total Block coverage = 100% ( 3/3 )
```

Related information in the *XL C/C++ Compiler Reference*

-qpdf1, -qpdf2

-qshowpdf

Object level profile-directed feedback

About this task

In addition to optimizing entire executables, profile-directed feedback (PDF) can also be applied to specific object files. This can be an advantage in applications where patches or updates are distributed as object files or libraries rather than as executables. Also, specific areas of functionality in your application can be optimized without the process of relinking the entire application. In large applications, you can save the time and trouble that otherwise need to be spent relinking the application.

The process for using object level PDF is essentially the same as the standard PDF process but with a small change to the **-qpdf2** step. For object level PDF, compile your program using the **-qpdf1** option, execute the resulting application with representative data, compile the program again with the **-qpdf2** option, but now also use the **-qnoipa** option so that the linking step is skipped.

The steps below outline this process:

1. Compile your program using the **-qpdf1** option. For example:
   ```
   xlc -c -O3 -qpdf1 file1.c file2.c file3.c
   ```
   In this example, we are using the optimization level **-O3** to indicate that we want a moderate level of optimization.
2. Link the object files to get an instrumented executable:
   ```
   xlc -O3 -qpdf1 file1.o file2.o file3.o
   ```
3. Run the instrumented executable with sample data that is representative of the data you want to optimize for.
   ```
   a.out < sample_data
   ```
4. Compile the program again using the **-qpdf2** option. Specify the **-qnoipa** option so that the linking step is skipped and PDF optimization is applied to the object files rather than to the entire executable.
   ```
   xlc -c -O3 -qpdf2 -qnoipa file1.c file2.c file3.c
   ```
   The resulting output of this step are object files optimized for the sample data processed by the original instrumented executable. In this example, the optimized object files would be file1.o, file2.o, and file3.o. These can be linked by using the system loader **ld** or by omitting the **-c** option in the **-qpdf2** step.

Notes:

- You must use the same optimization level in all the steps. In this example, the optimization level is **-O3**.
- If you want to specify a file name for the profile that is created, use the **pdfname** suboption in both the **-qpdf1** and **-qpdf2** steps. For example:

  ```
  xlc -O3 -qpdf1=pdfname=myprofile file1.c file2.c file3.c
  ```

  Without the **pdfname** suboption, by default the file name is `._pdf`; the location of the file is the current working directory or whatever directory you have set using the PDFDIR environment variable. If the PDFDIR environment variable is set but the specified directory does not exist, the compiler issues a warning message.
- Because the **-qnoipa** option needs to be specified in the **-qpdf2** step so that linking of your object files is skipped, you cannot use interprocedural analysis (IPA) optimizations and object level PDF at the same time.

For details, see -qpdf1, -qpdf2 in the XL C/C++ Compiler Reference.

Other optimization options

Options are available to control particular aspects of optimization. They are often enabled as a group or given default values when you enable a more general optimization option or level.

For more information about these options, see the heading for each option in the *XL C/C++ Compiler Reference.*

Table 14. Selected compiler options for optimizing performance

Option	Description
-finline-functions	Controls inlining.
▶ C++ **-qnoeh**	Informs the compiler that no C++ exceptions will be thrown and that cleanup code can be omitted. If your program does not throw any C++ exceptions, use this option to compact your program by removing exception-handling code.

Related information in the *XL C/C++ Compiler Reference*

-finline-functions

-qeh (C++ only)

Chapter 6. Debugging optimized code

Debugging optimized programs presents special usability problems. Optimization can change the sequence of operations, add or remove code, change variable data locations, and perform other transformations that make it difficult to associate the generated code with the original source statements.

For example:

Data location issues
: With an optimized program, it is not always certain where the most current value for a variable is located. For example, a value in memory might not be current if the most current value is being stored in a register. Most debuggers cannot follow the removal of stores to a variable, and to the debugger it appears as though that variable is never updated, or possibly even never set. This contrasts with no optimization where all values are flushed back to memory and debugging can be more effective and usable.

Instruction scheduling issues
: With an optimized program, the compiler might reorder instructions. That is, instructions might not be executed in the order you would expect based on the sequence of lines in the original source code. Also, the sequence of instructions for a statement might not be contiguous. As you step through the program with a debugger, the program might appear as if it is returning to a previously executed line in the code (interleaving of instructions).

Consolidating variable values
: Optimizations can result in the removal and consolidation of variables. For example, if a program has two expressions that assign the same value to two different variables, the compiler might substitute a single variable. This can inhibit debug usability because a variable that a programmer is expecting to see is no longer available in the optimized program.

There are a couple of different approaches you can take to improve debug capabilities while also optimizing your program:

Debug non-optimized code first
: Debug a non-optimized version of your program first, and then recompile it with your desired optimization options. See "Debugging in the presence of optimization" on page 40 for some compiler options that are useful in this approach.

Use -g level
: Use the **-g** level suboption to control the amount of debugging information made available. Increasing it improves debug capability but prevents some optimizations. For more information, see **-g**.

Understanding different results in optimized programs

Here are some reasons why an optimized program might produce different results from one that has not undergone the optimization process:

- Optimized code can fail if a program contains code that is not valid. The optimization process relies on your application conforming to language standards.
- If a program that works without optimization fails when you optimize, check the cross-reference listing and the execution flow of the program for variables that are used before they are initialized. Compile with the **-qinitauto=***hex_value* option to try to produce the incorrect results consistently. For example, using **-qinitauto=FF** gives variables an initial value of "negative not a number" (-NAN). Any operations on these variables will also result in NAN values. Other bit patterns (***hex_value***) might yield different results and provide further clues as to what is going on. Programs with uninitialized variables can appear to work properly when compiled without optimization because of the default assumptions the compiler makes, but such programs might fail when you optimize. Similarly, a program can appear to execute correctly after optimization, but it fails at lower optimization levels or when it is run in a different environment.
- Referring to an automatic-storage variable by its address after the owning function has gone out of scope leads to a reference to a memory location that can be overwritten as other auto variables come into scope as new functions are called.

Use with caution debugging techniques that rely on examining values in storage, unless the **-g8** or **-g9** option is in effect and the optimization level is **-O2**. The compiler might have deleted or moved a common expression evaluation. It might have assigned some variables to registers so that they do not appear in storage at all.

Debugging in the presence of optimization

Debug and compile your program with your desired optimization options. Test the optimized program before placing it into production. If the optimized code does not produce the expected results, you can attempt to isolate the specific optimization problems in a debugging session.

The following list presents options that provide specialized information, which can be helpful during the debugging of optimized code:

-qlist Instructs the compiler to emit an object listing. The object listing includes hex and pseudo-assembly representations of the generated instructions, traceback tables, and text constants.

-qreport
Instructs the compiler to produce a report of the loop transformations it performed, what inlining was done, and some other transformations. To generate a listing file, you must specify the **-qreport** option with at least one optimization option such as **-qhot**, **-finline-functions (-qinline)**, or **-ftree-vectorize (-qsimd)**.

-qipa=list
Instructs the compiler to emit an object listing that provides information for IPA optimization.

-qkeepparm
Ensures that procedure parameters are stored on the stack even during optimization. This can negatively impact execution performance. The

-qkeepparm option then provides access to the values of incoming parameters to tools, such as debuggers, simply by preserving those values on the stack.

-qinitauto
: Instructs the compiler to emit code that initializes all automatic variables to a given value.

-g
: Generates debugging information to be used by a symbolic debugger. You can use different **-g** levels to debug optimized code by viewing or possibly modifying accessible variables at selected source locations in the debugger. Higher **-g** levels provide a more complete debug support, while lower levels provide higher runtime performance. For details, see **-g**.

Chapter 7. Coding your application to improve performance

Chapter 5, "Optimizing your applications," on page 21 discusses the various compiler options that the XL C/C++ compiler provides for optimizing your code with minimal coding effort. If you want to take your application a step further to complement and take the most advantage of compiler optimizations, the topics in this section discuss C and C++ programming techniques that can improve performance of your code.

Finding faster input/output techniques

There are a number of ways to improve your program's performance of input and output:

- If your file I/O accesses do not exhibit locality (that is truly random access such as in a database), implement your own buffering or caching mechanism on the low-level I/O functions.
- If you do your own I/O buffering, make the buffer a multiple of 4KB, which is the minimum size of a page.
- Use buffered I/O to handle text files.
- If you have to process an entire file, determine the size of the data to be read in, allocate a single buffer to read it to, read the whole file into that buffer at once using `read`, and then process the data in the buffer. This reduces disk I/O, provided the file is not so big that excessive swapping will occur. Consider using the `mmap` function to access the file.

Reducing function-call overhead

When you write a function or call a library function, consider the following guidelines:

- Call a function directly, rather than using function pointers.
- Use `const` arguments in inlined functions whenever possible. Functions with constant arguments provide more opportunities for optimization.
- Use the `restrict` keyword for pointers that can never point to the same memory.
- Use **#pragma disjoint** within functions for pointers or reference parameters that can never point to the same memory.
- Declare a nonmember function as static whenever possible. This can speed up calls to the function and increase the likelihood that the function will be inlined.
- C++ Usually, you should not declare all your virtual functions inline. If all virtual functions in a class are inline, the virtual function table and all the virtual function bodies will be replicated in each compilation unit that uses the class.
- C++ When declaring functions, use the `const` specifier whenever possible.
- C Fully prototype all functions. A full prototype gives the compiler and optimizer complete information about the types of the parameters. As a result, promotions from unwidened types to widened types are not required, and parameters can be passed in appropriate registers.
- C Avoid using unprototyped variable argument functions.
- Design functions so that they have few parameters and the most frequently used parameters are in the leftmost positions in the function prototype.

- Avoid passing by value large structures or unions as function parameters or returning a large structure or a union. Passing such aggregates requires the compiler to copy and store many values. This is worse in C++ programs in which class objects are passed by value because a constructor and destructor are called when the function is called. Instead, pass or return a pointer to the structure or union, or pass it by reference.
- Pass non-aggregate types such as `int` and `short` or small aggregates by value rather than passing by reference, whenever possible.
- If your function exits by returning the value of another function with the same parameters that were passed to your function, put the parameters in the same order in the function prototypes. The compiler can then branch directly to the other function.
- Use the built-in functions, which include string manipulation, floating-point, and trigonometric functions, instead of coding your own. Intrinsic functions require less overhead and are faster than a function call, and they often allow the compiler to perform better optimization.

  C++ Many functions from the C++ standard libraries are mapped to optimized built-in functions by the compiler.

  C Many functions from `string.h` and `math.h` are mapped to optimized built-in functions by the compiler.
- Selectively mark your functions for inlining using the `inline` keyword. An inlined function requires less overhead and is generally faster than a function call. The best candidates for inlining are small functions that are called frequently from a few places, or functions called with one or more compile-time constant parameters, especially those that affect `if`, `switch`, or `for` statements. You might also want to put these functions into header files, which allows automatic inlining across file boundaries even at low optimization levels. Be sure to inline all functions that only load or store a value, or use simple operators such as comparison or arithmetic operators. Large functions and functions that are called rarely are generally not good candidates for inlining. Neither are medium size functions that are called from many places.
- Avoid breaking your program into too many small functions. If you must use small functions, you can use the **-qipa** compiler option to automatically inline such functions and use other techniques to optimize calls between functions.
- C++ Avoid virtual functions and virtual inheritance unless required for class extensibility. These language features are costly in object space and function invocation performance.

Related information in the *XL C/C++ Compiler Reference*

-qisolated_call

#pragma disjoint

-qipa

Managing memory efficiently (C++ only)

Because C++ objects are often allocated from the heap and have limited scope, memory use affects performance more in C++ programs than it does in C programs. For that reason, consider the following guidelines when you develop C++ applications:

- In a structure, declare the largest aligned members first. Members of similar alignment should be grouped together where possible.

- In a structure, place variables near each other if they are frequently used together.
- Ensure that objects that are no longer needed are freed or otherwise made available for reuse. One way to do this is to use an *object manager*. Each time you create an instance of an object, pass the pointer to that object to the object manager. The object manager maintains a list of these pointers. To access an object, you can call an object manager member function to return the information to you. The object manager can then manage memory usage and object reuse.
- Storage pools are a good way of keeping track of used memory (and reclaiming it) without having to resort to an object manager or reference counting. Do not use storage pools for objects with non-trivial destructors, because in most implementations the destructors cannot be run when the storage pool is cleared.
- Avoid copying large and complicated objects.
- Avoid performing a *deep copy* if you only need a *shallow copy*. For an object that contains pointers to other objects, a shallow copy copies only the pointers and not the objects to which they point. The result is two objects that point to the same contained object. A deep copy, however, copies the pointers and the objects they point to, as well as any pointers or objects that are contained within that object, and so on. A deep copy must be performed in multithreaded environments, because it reduces sharing and synchronization.
- Use virtual methods only when absolutely necessary.
- Use the "Resource Acquisition is Initialization" (RAII) pattern.
- Use `shared_ptr` and `weak_ptr`.

Optimizing variables

Consider the following guidelines:

- Use local variables, preferably automatic variables, as much as possible. The compiler must make several worst-case assumptions about global variables. For example, if a function uses external variables and also calls external functions, the compiler assumes that every call to an external function could use and change the value of every external variable. If you know that a global variable is not read or affected by any function call and this variable is read several times with function calls interspersed, copy the global variable to a local variable and then use this local variable.
- If you must use global variables, use static variables with file scope rather than external variables whenever possible. In a file with several related functions and static variables, the optimizer can gather and use more information about how the variables are affected.
- If you must use external variables, group external data into structures or arrays whenever it makes sense to do so. All elements of an external structure use the same base address. Do not group variables whose addresses are taken with variables whose addresses are not taken.
- Avoid taking the address of a variable. If you use a local variable as a temporary variable and must take its address, avoid reusing the temporary variable for a different purpose. Taking the address of a local variable can inhibit optimizations that would otherwise be done on calculations involving that variable.
- Use constants instead of variables where possible. The optimizer is able to do a better job reducing runtime calculations by doing them at compile time instead. For instance, if a loop body has a constant number of iterations, use constants in the loop condition to improve optimization (`for (i=0; i<4; i++)` can be better

optimized than `for (i=0; i<x; i++)`). An enumeration declaration can be used to declare a named constant for maintainability.

- Use register-sized integers (`long` data type) for scalars to avoid sign extension instructions after each change in 64-bit mode. For large arrays of integers, consider using one-byte or two-byte integers or bit fields.
- Use the smallest floating-point precision appropriate to your computation.

Related information in the *XL C/C++ Compiler Reference*

-qisolated_call

Manipulating strings efficiently

The handling of string operations can affect the performance of your program.

- When you store strings into allocated storage, align the start of the string on an 8-byte or 16-byte boundary.
- Keep track of the length of your strings. If you know the length of a string, you can use `mem` functions instead of `str` functions. For example, `memcpy` is faster than `strcpy` because it does not have to search for the end of the string.
- If you are certain that the source and target do not overlap, use `memcpy` instead of `memmove`. This is because `memcpy` copies directly from the source to the destination, while `memmove` might copy the source to a temporary location in memory before copying to the destination, or it might copy in reverse order depending on the length of the string.
- When manipulating strings using `mem` functions, faster code can be generated if the *count* parameter is a constant rather than a variable. This is especially true for small count values.

Optimizing expressions and program logic

Consider the following guidelines:

- If components of an expression are used in other expressions and they include function calls or there are function calls between the uses, assign the duplicated values to a local variable.
- Avoid forcing the compiler to convert numbers between integer and floating-point internal representations. For example:

```
float array[10];
float x = 1.0;
int i;
for (i = 0; i< 9; i++)  {     /* No conversions needed */
     array[i] = array[i]*x;
     x = x + 1.0;
    }
for (i = 0; i< 9; i++)  {     /* Multiple conversions needed */
     array[i] = array[i]*i;
    }
```

  When you must use mixed-mode arithmetic, code the integer and floating-point arithmetic in separate computations whenever possible.
- Do not use global variables as loop indices or bounds.
- Avoid `goto` statements that jump into the middle of loops. Such statements inhibit certain optimizations.
- Improve the predictability of your code by making the fall-through path more probable. Code such as:

```
if (error) {handle error} else {real code}
```

should be written as:

```
if (!error) {real code} else {error}
```

- If one or two cases of a `switch` statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the `switch` statement. If possible, replace the `switch` statement by checking whether the value is in range to be obtained from an array.
- C++ Use `try` blocks for exception handling only when necessary because they can inhibit optimization.
- Keep array index expressions as simple as possible.

Optimizing operations in 64-bit mode

The ability to handle larger amounts of data directly in physical memory rather than relying on disk I/O is perhaps the most significant performance benefit of 64-bit machines. However, some applications compiled in 31-bit mode perform better than when they are recompiled in 64-bit mode. Some reasons for this include:

- 64-bit programs are larger. The increase in program size places greater demands on physical memory.
- 64-bit long division is more time-consuming than 31-bit integer division.
- 64-bit programs that use 31-bit signed integers as array indexes or loop counts might require additional instructions to perform sign extension each time the array is referenced or the loop count is incremented.

Some ways to compensate for the performance liabilities of 64-bit programs include:

- Avoid performing mixed 31-bit and 64-bit operations. For example, adding a 31-bit data type to a 64-bit data type requires that the 31-bit be sign-extended to clear or set the upper 31-bit of the register. This slows the computation.
- Use `long` types instead of `signed`, `unsigned`, and plain `int` types for variables that will be frequently accessed, such as loop counters and array indexes. Doing so frees the compiler from having to truncate or sign-extend array references, parameters during function calls, and function results during returns.

The C++ template model

In C++, you can use a template to declare a set of related following entities:

- Classes (including structures)
- Functions
- Static data members of template classes

Each compiler implements templates according to a model that determines the meaning of a template at various stages of the translation of a program. In particular, the compiler determines what the various constructs in a template mean when the template is instantiated. Name lookup is an essential ingredient of the compilation model.

Template instantiation is a process that generates types and functions from generic template definitions. The concept of instantiation of C++ templates is fundamental but also intricate because the definitions of entities generated by a template are no longer limited to a single location in the source code. The location of the template, the location where the template is used, and the locations where the template arguments are defined all contribute to the meaning of the entity.

XL C/C++ supports *Greedy instantiation*. The compiler generates a template instantiation in each compilation unit that uses it. The linker discards the duplicates.

Related information in the *XL C/C++ Compiler Reference*

-qtmplinst (C++ only)

Using delegating constructors (C++11)

Note: IBM supports the majority of C++11 features and will continue to develop and implement the features of this standard.

Before C++11, common initialization in multiple constructors of the same class cannot be concentrated in one place in a robust and maintainable manner. Starting from C++11, with the delegating constructors feature, you can concentrate common initialization in one constructor, which can make the program more readable and maintainable. Delegating constructors help reduce code size and collective size of object files.

Syntactically, *delegating constructors* and *target constructors* present the same interface as other constructors.

Consider the following points when you use the delegating constructors feature:

- Call the target constructor implementation in such a way that virtual bases, direct nonvirtual bases, and class members are initialized by the target constructor as appropriate.
- The feature has minimal impact on compile-time and runtime performance. However, use of default arguments with an existing constructor is recommended in place of a delegating constructor where possible. Without inlining and interprocedural analysis, runtime performance might degrade because of function call overhead and increased opacity.

Using rvalue references (C++11)

Note: IBM supports the majority of C++11 features and will continue to develop and implement the features of this standard.

In C++11, you can overload functions based on the value categories of arguments and similarly have lvalueness detected by template argument deduction. You can also have an rvalue bound to an rvalue reference and modify the rvalue through the reference. This enables a programming technique with which you can reuse the resources of expiring objects and therefore improve the performance of your libraries, especially if you use generic code with class types, for example, template data structures. Additionally, the value category can be considered when writing a forwarding function.

Move semantics

When you want to optimize the use of temporary values, you can use a move operation in what is known as destructive copying. Consider the following string concatenation and assignment:

```
std::string a, b, c;
c = a + b;
```

In this program, the compiler first stores the result of a + b in an internal temporary variable, that is, an rvalue.

The signature of a normal copy assignment operator is as follows:

```
string& operator = (const string&)
```

With this copy assignment operator, the assignment consists of the following steps:

1. Copy the temporary variable into c using a deep-copy operation.
2. Discard the temporary variable.

Deep copying the temporary variable into c is not efficient because the temporary variable is discarded at the next step.

To avoid the needless duplication of the temporary variable, you can implement an assignment operator that moves the variable instead of copying the variable. That is, the argument of the operator is modified by the operation. A move operation is faster because it is done through pointer manipulation, but it requires a reference through which the source variable can be manipulated. However, a + b is a temporary value, which is not easily differentiated from a const-qualified value in C++ before C++11 for the purposes of overload resolution.

With rvalue references, you can create a move assignment operator as follows:

```
string& operator= (string&&)
```

With this move assignment operator, the memory allocated for the underlying C-style string in the result of a + b is assigned to c. Therefore, it is not necessary to allocate new memory to hold the underlying string in c and to copy the contents to the new memory.

The following code can be an implementation of the string move assignment operator:

```
string& string::operator=(string&& str)
{
  // The named rvalue reference str acts like an lvalue
  std::swap(_capacity, str._capacity);
  std::swap(_length, str._length);

  // char* _str points to a character array and is a
  // member variable of the string class
  std::swap(_str, str._str);
  return *this;
}
```

However, in this implementation, the memory originally held by the string being assigned to is not freed until str is destroyed. The following implementation that uses a local variable is more memory efficient:

```
string& string::operator=(string&& parm_str)
{
  // The named rvalue reference parm_str acts like an lvalue
  string sink_str;
  std::swap(sink_str, parm_str);
  std::swap(*this, sink_str);
  return *this;
}
```

In a similar manner, the following program is a possible implementation of a string concatenation operator:

```
string operator+(string&& a, const string& b)
{
  return std::move(a+=b);
}
```

Note: The `std::move` function only casts the result of `a+=b` to an rvalue reference, without moving anything. The return value is constructed using a move constructor because the expression `std::move(a+=b)` is an rvalue. The relationship between a move constructor and a copy constructor is analogous to the relationship between a move assignment operator and a copy assignment operator.

Perfect forwarding

The `std::forward` function is a helper template, much like `std::move`. It returns a reference to its function argument, with the resulting value category determined by the template type argument. In an instantiation of a forwarding function template, the value category of an argument is encoded as part of the deduced type for the related template type parameter. The deduced type is passed to the `std::forward` function.

The `wrapper` function in the following example is a forwarding function template that forwards to the `do_work` function. Use `std::forward` in forwarding functions on the calls to the target functions. The following example also uses the decltype and trailing return type features to produce a forwarding function that forwards to one of the `do_work` functions. Calling the `wrapper` function with any argument results in a call to a `do_work` function if a suitable overload function exists. Extra temporaries are not created and overload resolution on the forwarding call resolves to the same overload as it would if the `do_work` function were called directly.

```
struct s1 *do_work(const int&);             // #1
struct s2 *do_work(const double&);          // #2
struct s3 *do_work(int&&);                  // #3
struct s4 *do_work(double&&);               // #4
template <typename T> auto wrapper(T && a)->
   decltype(do_work(std::forward<T>(*static_cast<typename std::remove_reference<T>
   ::type*>(0))))
{
   return do_work(std::forward<T>(a));
}
template <typename T> void tPtr(T *t);
int main()
{
   int x;
   double y;
   tPtr<s1>(wrapper(x));     // calls #1
   tPtr<s2>(wrapper(y));     // calls #2
   tPtr<s3>(wrapper(0));     // calls #3
   tPtr<s4>(wrapper(1.0));   // calls #4
}
```

Related information in the *XL C/C++ Compiler Reference*

-qlanglvl

Using visibility attributes (IBM extension)

Visibility attributes describe whether and how an entity that is defined in one module can be referenced or used in other modules. Visibility attributes affect entities with external linkage only, and they cannot increase the visibility of other entities. By specifying visibility attributes for entities, you can export only the entities that are necessary to shared libraries. With this feature, you can get the following benefits:

- Decrease the size of shared libraries.
- Reduce the possibility of symbol collision.
- Allow more optimization for the compile and link phases.
- Improve the efficiency of dynamic linking.

Supported types of entities

C++

The compiler supports visibility attributes for the following entities:

- Function
- Variable
- Structure/union/class
- Enumeration
- Template
- Namespace

C++

C

The compiler supports visibility attributes for the following entities:

- Function
- Variable

Note: Data types in the C language do not have external linkage, so you cannot specify visibility attributes for C data types.

C

Related information in the *XL C/C++ Compiler Reference*

-shared (-qmkshrobj)

#pragma GCC visibility push, #pragma GCC visibility pop

Related information in the *XL C/C++ Language Reference*

The visibility variable attribute (IBM extension)

The visibility function attribute (IBM extension)

The visibility type attribute (C++ only) (IBM extension)

The visibility namespace attribute (C++ only) (IBM extension)

Types of visibility attributes

The following table describes different visibility attributes.

Table 15. Visibility attributes

Attribute	**Description**
default	Indicates that external linkage entities have the default attribute in object files. These entities are exported in shared libraries, and can be preempted.
protected	Indicates that external linkage entities have the protected attribute in object files. These entities are exported in shared libraries, but cannot be preempted.
hidden	Indicates that external linkage entities have the hidden attribute in object files. These entities are not exported in shared libraries, but their addresses can be referenced indirectly through pointers.
internal	Indicates that external linkage entities have the internal attribute in object files. These entities are not exported in shared libraries, and their addresses are not available to other modules in shared libraries.
Notes: • In this release, the hidden and internal visibility attributes are the same. The addresses of the entities that are specified with either of these visibility attributes can be referenced indirectly through pointers.	

Example: Differences among the default, protected, hidden, and internal visibility attributes

```
//a.c
#include <stdio.h>
void __attribute__((visibility("default"))) func1(){
   printf("func1 in the shared library");
}
void __attribute__((visibility("protected"))) func2(){
   printf("func2 in the shared library");
}
void __attribute__((visibility("hidden"))) func3(){
   printf("func3 in the shared library");
}
void __attribute__((visibility("internal"))) func4(){
   printf("func4 in the shared library");
}

//a.h
extern void func1();
extern void func2();
extern void func3();
extern void func4();

//b.c
#include "a.h"
void temp(){
   func1();
   func2();
}

//b.h
extern void temp();

//main.c
#include "a.h"
#include "b.h"

void func1(){
   printf("func1 in b.c");
}
```

```
void func2(){
   printf("func2 in b.c");
}
void main(){
   temp();
   // func3(); // error
   // func4(); // error
}
```

You can use the following commands to create a shared library named `libtest.so`:

```
xlc -c -fPIC a.c b.c
xlc -shared -o libtest.so a.o b.o
```

Then, you can dynamically link `libtest.so` during run time by using the following commands:

```
xlc main.c -L. -ltest -o main
./main
```

The output of the example is as follows:

```
func1 in b.c
func2 in the shared library
```

The visibility attribute of function `func1()` is default, so it is preempted by the function with the same name in `main.c`. The visibility attribute of function `func2()` is protected, so it cannot be preempted. The compiler always calls `func2()` that is defined in the shared library `libtest.so`. The visibility attribute of function `func3()` is hidden, so it is not exported in the shared library. The compiler issues a link error to indicate that the definition of `func3()` cannot be found. The same issue is with function `func4()` whose visibility attribute is internal.

Rules of visibility attributes

Priority of visibility attributes

The visibility attributes have a priority sequence, which is default < protected < hidden < internal. You can see Example 9 for reference.

Rules of determining the visibility attributes

▶ C

The visibility attribute of an entity is determined by the following rules:

1. If the entity has an explicitly specified visibility attribute, the specified visibility attribute takes effect.
2. Otherwise, if the entity has a pair of enclosing pragma directives, the visibility attribute that is specified by the pragma directives takes effect.

C ◀

▶ C++

The visibility attribute of an entity is determined by the following rules:

1. If the entity has an explicitly specified visibility attribute, the specified visibility attribute takes effect.

2. Otherwise, if the entity is a template instantiation or specialization, and the template has a visibility attribute, the visibility attribute of the entity is propagated from that of the template. See Example 1.
3. Otherwise, if the entity has any of the following enclosing contexts, the visibility attribute of this entity is propagated from that of the nearest context. See Example 2. For the details of propagation rules, see "Propagation rules (C++ only)" on page 58.
   - Structure/class
   - Enumeration
   - Namespace
   - Pragma directives

   Restriction: Pragma directives do not affect the visibility attributes of class members and template specializations.
4. Otherwise, the visibility attribute of the entity is determined by the following visibility attribute settings. The visibility attribute that has the highest priority is the actual visibility attribute of the entity. For the priority of the visibility attributes, see Priority of visibility attributes.
   - The visibility attribute of the type of the entity, if the entity is a variable and its type has a visibility attribute.
   - The visibility attribute of the return type of the entity, if the entity is a function and its return type has a visibility attribute.
   - The visibility attributes of the parameter types of the entity, if the entity is a function and its parameter types have visibility attributes.
   - The visibility attributes of template arguments or template parameters of the entity, if the entity is a template and its arguments or parameters have visibility attributes.

Example 1

In the following example, template `template<typename T, typename U> B{}` has the protected visibility attribute. The visibility attribute is propagated to those of template specialization `template<> class B<char, char>{}`, partial specialization `template<typename T> class B<T, float>{}`, and all the types of template instantiations.

```
class __attribute__((visibility("internal"))) A{} vis_v_a;   //internal

//protected
template<typename T, typename U>
class __attribute__((visibility("protected"))) B{
   public:
   void func(){}
};

//protected
template<>
class B<char, char>{
   public:
   void func(){}
};

//protected
template<typename T>
class B<T, float>{
   public:
   void func(){}
};
```

```
B<int, int> a;    //protected
B<A, int> b;      //protected
B<char, char> c;  //protected
B<int, float> d;  //protected
B<A, float> e;    //protected

int main(){
   a.func();
   b.func();
   c.func();
   d.func();
   e.func();
}
```

Example 2

In the following example, the nearest enclosing context of function `func()` is class `B`, so the visibility attribute of `func()` is propagated from that of class `B`, which is hidden. The nearest enclosing context of class `A` is the pragma directives whose setting is protected, so the visibility of class `A` is protected.

```
namespace __attribute__((visibility("internal"))) ns{
#pragma GCC visibility push(protected)
   class A{
      class __attribute__((visibility("hidden"))) B{
         int func(){};
      };
   };
#pragma GCC visibility pop
};
```

C++

Rules and restrictions of using the visibility attributes

When you specify visibility attributes for entities, consider the following rules and restrictions:

- You can specify visibility attributes only for entities that have external linkage. The compiler issues a warning message when you set the visibility attribute for entities with other linkages, and the specified visibility attribute is ignored. See Example 4.
- You cannot specify different visibility attributes in the same declaration or definition of an entity; otherwise, the compiler issues an error message. See Example 5.
- If an entity has more than one declaration that is specified with different visibility attributes, the visibility attribute of the entity is the first visibility attribute that the compiler processes. See Example 6.
- You cannot specify visibility attributes in the `typedef` statements. See Example 7.
- C++ If type `T` has a visibility attribute, types `T*`, `T&`, and `T&&` have the same visibility attribute with that of type `T`. See Example 8.
- C++ If a class and its enclosing classes do not have explicitly specified visibilities and the visibility attribute of the class has a lower priority than those of its nonstatic member types and its bases classes, the compiler issues a warning message. See Example 9. For the priority of the visibility attributes, see Priority of visibility attributes. C++
- C++ The visibility attribute of a namespace does not apply for the namespace with the same name. See Example 10. C++

- C++ If you specify a visibility attribute for a global new or delete operator, the compiler issues a warning message to ignore the visibility attribute unless the visibility attribute is default. See Example 11. C++

Example 3

In this example, because `m` and `i` have internal linkage and `j` has no linkage, the compiler ignores the visibility attributes of variables `m`, `i`, and `j`.

```
static int m __attribute__((visibility("protected")));
int n __attribute__((visibility("protected")));

int main(){
   int i __attribute__((visibility("protected")));
   static int j __attribute__((visibility("protected)));
}
```

Example 4

In this example, the compiler issues an error message to indicate that you cannot specify two different visibility attributes at the same time in the definition of variable `m`.

```
//error
int m __attribute__((visibility("hidden"))) __attribute__((visibility("protected")));
```

Example 5

In this example, the first declaration of function `fun()` that the compiler processes is `extern void fun() __attribute__((visibility("hidden")))`, so the visibility attribute of `fun()` is hidden.

```
extern void fun() __attribute__((visibility("hidden")));
extern void fun() __attribute__((visibility("protected")));

int main(){
   fun();
}
```

Example 6

In this example, the visibility attribute of variable `vis_v_ti` is default, which is not affected by the setting in the `typedef` statement.

```
typedef int __attribute__((visibility("protected"))) INT;
INT vis_v_ti = 1;
```

C++

Example 7

In this example, the visibility attribute of class `CP` is protected, so the visibility attribute of `CP*` and `CP&` is also protected.

```
class __attribute__((visibility("protected"))) CP {} vis_v_p;
class CP* vis_v_p_p = &vis_v_p;  //protected
class CP& vis_v_lr_p = vis_v_p;  //protected
```

Example 8

In this example, the compiler accepts the default visibility attribute of class `Derived1` because the visibility attribute is explicitly specified for class `Derived1`.

The compiler also accepts the protected visibility attribute of class Derived2 because the visibility attribute is propagated from that of the enclosing class A. Class Derived3 does not have an explicitly specified visibility attribute or an enclosing class, and its visibility attribute is default. The compiler issues a warning message because the visibility attribute of class Derived3 has a lower priority than those of its parent class Base and the nonstatic member function fun().

```
//base class
struct __attribute__((visibility("hidden"))) Base{
   int vis_f_fun(){
     return 0;
   }
};

//Ok
struct __attribute__((visibility("default"))) Derived1: public Base{
   int vis_f_fun(){
      return Base::vis_f_fun();
   };
}vis_v_d;

//Ok
struct __attribute__((visibility("protected"))) A{
   struct Derived2: public Base{
      int vis_f_fun(){
         __attribute__((visibility("protected")))
      };
   }
};

//Warning
struct Derived3: public Base{
  //Warning
  int fun() __attribute__((visibility("protected"))){};
};
```

Example 9

In this example, the visibility attribute of the definition of namespace X does not apply to the extension of namespace X.

```
//namespace definition
namespace X __attribute__((visibility("protected"))){
  int a;  //protected
  int b;  //protected
}
//namespace extension
namespace X {
  int c;  //default
  int d;  //default
}
//equivalent to namespace X
namespace Y {
  int __attribute__((visibility("protected"))) a;  //protected
  int __attribute__((visibility("protected"))) b;  //protected
  int c;  //default
  int d;  //default
}
```

Example 10

In this example, the new and delete operators defined outside of class A are global functions, so the explicitly specified hidden visibility attribute does not take effect. The new and delete operations defined within class A are local ones, so you can specify visibility attributes for them.

```
#include <stddef.h>
//default
void* operator new(size_t) throw (std::bad_alloc) __attribute__((visibility("hidden")))
{
   return 0;
};
void operator delete(void*) throw () __attribute__((visibility("hidden"))){}

class A{
   public:
   //hidden
   void* operator new(size_t) throw (std::bad_alloc) __attribute__((visibility("hidden")))
   {
      return 0;
   };
   void operator delete(void*) throw () __attribute__((visibility("hidden"))){}
};
```

C++ ◄

Propagation rules (C++ only)

Visibility attributes can be propagated from one entity to other entities. The following table lists all the cases for visibility propagation.

Table 16. Propagation of visibility attributes

Original entity	Destination entities	Example

Namespace	Named namespaces that are defined in the original namespace	`namespace A __attribute__((visibility("hidden"))){`
		`   // Namespace B has the hidden visibility attribute,`
		`   // which is propagated from namespace A.`
		`   namespace B{}`
		`   // The unnamed namespace does not have a visibility`
		`   // attribute.`
		`   namespace{}`
		`}`

Namespace	Classes that are defined in the original namespace	`namespace A __attribute__((visibility("hidden"))){`
		`   // Class B has the hidden visibility attribute,`
		`   // which is propagated from namespace A.`
		`   class B;`
		`   // Object x has the hidden visibility attribute,`
		`   // which is propagated from namespace A.`
		`   class{} x;`
		`}`

Namespace	Functions that are defined in the original namespace	`namespace A __attribute__((visibility("hidden"))){`
		`   // Function fun() has the hidden visibility`
		`   // attribute, which is propagated from namespace A.`
		`   void fun(){};`
		`}`

Namespace	Objects that are defined in the original namespace	`namespace A __attribute__((visibility("hidden"))){`
		`   // Variable m has the hidden visibility attribute,`
		`   // which is propagated from namespace A.`
		`   int m;`
		`}`

Class	Member classes	`class __attribute__((visibility("hidden"))) A{`
		`   // Class B has the hidden visibility attribute,`
		`   // which is propagated from class A.`
		`   class B{};`
		`}`

Table 16. Propagation of visibility attributes (continued)

Original entity	**Destination entities**	**Example**

Class	Member functions or static member variables	`class __attribute__((visibility("hidden"))) A{`
		`    // Function fun() has the hidden visibility`
		`    // attribute, which is propagated from class A.`
		`    void fun(){};`
		`    // Static variable m has the hidden visibility`
		`    // attribute, which is propagated from class A.`
		`    static int m;`
		`}`

Template	Template instantiations/ template specifications/ template partial specializations	`template<typename T, typename U>`
		`class  __attribute__((visibility("hidden"))) A{`
		`   public:`
		`   void fun(){};`
		`};`
		
		`// Template instantiation class A<int, char> has the`
		`// hidden visibility attribute, which is propagated`
		`// from template class A(T,U).`
		`class A<int, char>{`
		`   public:`
		`   void fun(){};`
		`};`
		
		`// Template specification`
		`// template<> class A<double, double> has the hidden`
		`// visibility attribute, which is propagated`
		`// from template class A(T,U).`
		`template<> class A<double, double>{`
		`   public:`
		`   void fun(){};`
		`};`
		
		`// Template partial specification`
		`// template<typename T> class A<T, char> has the`
		`// hidden visibility attribute, which is propagated`
		`// from template class A(T,U).`
		`template<typename T> class A<T, char>{`
		`   public:`
		`   void fun(){};`
		`};`

Template argument/ parameter	Template instantiations/ template specifications/ template partial specializations	`template<typename T> void fun1(){}`
		`template<typename T> void fun2(T){}`
		
		`class M __attribute__((visibility("hidden"))){} m;`
		
		`// Template instantiation fun1<M>() has the hidden`
		`// visibility attribute, which is propagated from`
		`// template argument M.`
		`fun1<M>();`
		
		`// Template instantiation fun2<M>(M) has the hidden`
		`// visibility attribute, which is propagated from`
		`// template parameter m.`
		`fun2(m);`
		
		`// Template specification fun1<M>() has the hidden`
		`// visibility attribute, which is propagated from`
		`// template argument M.`
		`template<> void fun1<M>();`

Table 16. *Propagation of visibility attributes (continued)*

Original entity	Destination entities	Example
Inline function	Static local variables	`inline void __attribute__((visibility("hidden"))) fun(){ // Variable m has the hidden visibility attribute, // which is propagated from inline function fun(). static int m = 4; }`
Type	Entities of the original type	`class __attribute__((visibility("hidden"))) A {}; // Object x has the hidden visibility attribute, // which is propagated from class A. class A x;`
Function return type	Function	`class __attribute__((visibility("hidden"))) A{}; // Function fun() has the hidden visibility attribute, // which is propagated from function return type A. A fun();`
Function parameter type	Function	`class __attribute__((visibility("hidden"))) A{}; // Function fun(class A) has the hidden visibility // attribute, which is propagated from function // parameter type A. void fun(class A);`

Specifying visibility attributes using pragma preprocessor directives

You can selectively set visibility attributes for entities by using pairs of the `#pragma GCC visibility push` and `#pragma GCC visibility pop` preprocessor directives throughout your source program.

The compiler supports nested visibility pragma preprocessor directives. If entities are included in several pairs of the nested `#pragma GCC visibility push` and `#pragma GCC visibility pop` directives, the nearest pair of directives takes effect. See Example 1.

You must not specify the visibility pragma directives for header files. Otherwise, your program might exhibit undefined behaviors. See Example 2.

C++ Visibility pragma directives `#pragma GCC visibility push` and `#pragma GCC visibility pop` affect only namespace-scope declarations. Class members and template specializations are not affected. See Example 3 and Example 4. C++

Examples

Example 1

In this example, the function and variables have the visibility attributes that are specified by their nearest pairs of pragma preprocessor directives.

```
#pragma GCC visibility push(default)
namespace ns
{
         void vis_f_fun() {}       //default
#  pragma GCC visibility push(internal)
         int vis_v_i;              //internal
#    pragma GCC visibility push(protected)
         int vis_v_j;              //protected
#      pragma GCC visibility push(hidden)
         int vis_v_k;             //hidden
#      pragma GCC visibility pop
#    pragma GCC visibility pop
#  pragma GCC visibility pop
}
#pragma GCC visibility pop
```

Example 2

In this example, the compiler issues a link error message to indicate that the definition of the `printf()` library function cannot be found.

```
#pragma GCC visibility push(hidden)
#include <stdio.h>
#pragma GCC visibility pop

int main(){
   printf("hello world!");
   return 0;
}
```

C++

Example 3

In this example, the visibility attribute of class members `vis_v_i` and `vis_f_fun()` is hidden. The visibility attribute is propagated from that of the class, but is not affected by the pragma directives.

```
class __attribute__((visibility("hidden"))) A{
#pragma GCC visibility push(protected)
   public:
    static int vis_v_i;
    void vis_f_fun() {}
#pragma GCC visibility pop
} vis_v_a;
```

Example 4

In this example, the visibility attribute of function `vis_f_fun()` is hidden. The visibility attribute is propagated from that of the template specialization or partial specialization, but is not affected by the pragma directives.

```
namespace ns{
   #pragma GCC visibility push(hidden)
   template <typename T, typename U> class TA{
      public:
      void vis_f_fun(){}
   };
   #pragma GCC visibility pop

   #pragma GCC visibility push(protected)
   //The visibility attribute of the template specialization is hidden.
   template <> class TA<char, char>{
      public:
      void vis_f_fun(){}
   };
   #pragma GCC visibility pop

   #pragma GCC visibility push(default)
   //The visibility attribute of the template partial specialization is hidden.
   template <typename T> class TA<T, long>{
```

```
   public:
   void vis_f_fun(){}
};
#pragma GCC visibility pop
```

C++ ◀

Chapter 8. Using vector programming support

In IBM XL C/C++ for Linux on z Systems, V1.2, you can use the Vector Facility for z/Architecture on the Linux distributions that have vector support and run on the IBM z13 models.

To enable vector programming support on the Linux distributions that have vector support, you must specify the **-mzvector** option with the **-march=z13** option or its equivalent; otherwise, the compiler ignores the **-mzvector** option and issues a warning message.

The __VEC__ macro is introduced for vector processing support, which indicates support for vector data types. When the **-mzvector** option is in effect, the value of __VEC__ is 10301.

Related information in the *XL C/C++ Compiler Reference*

-march (-qarch)

-mzvector

Vector data types (IBM extension)

In a declaration context, the keyword `vector` is recognized only when it is used as a type specifier and when vector programming support is enabled.

The `signed` qualifier and the `unsigned` qualifier are optional. If neither is specified, the sign of the type is the same as its underlying plain type. For example, `vector int` is interpreted as `vector signed int`, and `vector char` is interpreted as `vector unsigned char` if **-funsigned-char** is in effect.

Type qualifiers, such as `const`, and storage class specifiers, such as `static`, can be used to refine a vector declaration. Type qualifiers and storage class specifiers can appear in any place within the declaration other than immediately following keyword `vector` or its alternative spelling, `__vector`.

Duplicate type specifiers are ignored in a vector declaration context.

All vector types are aligned on an 8-byte boundary. An aggregate that contains one or more vector types is aligned or, if necessary, padded on an 8-byte boundary so that each member of the vector types is also 8-byte aligned.

The following table lists supported vector data types, the number of elements for each vector data type, and value range for each element.

Table 17. Vector data types

Type	**Interpretation of content**	**Range of element value**
`vector signed char`	16 elements of type `signed char`	From -128 to 127 inclusive
`vector unsigned char`	16 elements of type `unsigned char`	From 0 to 255 inclusive
`vector bool char`[1]	16 elements of type `unsigned char`	0 (FALSE), 255 (TRUE)

Table 17. Vector data types (continued)

Type	Interpretation of content	Range of element value
`vector signed short`	8 elements of type `signed short`	From -32768 to 32767 inclusive
`vector signed short int`		
`vector unsigned short`	8 elements of type `unsigned short`	From 0 to 65535 inclusive
`vector unsigned short int`		
`vector bool short`[1]	8 elements of type `unsigned short`	0 (FALSE), 65535 (TRUE)
`vector bool short int`[1]		
`vector signed int`	4 elements of type `signed int`	From -2^{31} to $(2^{31}-1)$ inclusive
`vector unsigned int`	4 elements of type `unsigned int`	From 0 to $(2^{32}-1)$ inclusive
`vector bool int`[1]	4 elements of type `unsigned int`	0 (FALSE), $2^{32}-1$ (TRUE)
`vector signed long long`	2 elements of type `signed long long`	From -2^{63} to $(2^{63}-1)$ inclusive
`vector unsigned long long`	2 elements of type `unsigned long long`	From 0 to $(2^{64}-1)$ inclusive
`vector bool long long`[1]	2 elements of type `unsigned long long`	0 (FALSE), $2^{64}-1$ (TRUE)
`vector double`	2 elements of type `double`	64-bit IEEE-754 double-precision floating-point values

Note:
1. The keyword `bool` is recognized as a valid type specifier only when preceded by the keyword `vector` or `__vector`. `__vector __bool`, `vector _Bool`, and `__vector _Bool` are equivalent to `vector bool`.

Related information in the *XL C/C++ Compiler Reference*

-mzvector

Vector literals (IBM extension)

A vector literal is a constant expression whose value is interpreted as a vector. The data type of a vector literal is represented by a parenthesized vector type. The value of a vector literal is a set of constant expressions that represent the vector elements and are enclosed in parentheses or braces.

Vector literal syntax

```
►►─(─vector_type─)─┬─(─literal_list─)─┬──────────────────────────►◄
                   └─{─literal_list─}─┘
```

literal_list:

```
   ┌─,──────────────────┐
   ▼                    │
├──constant_expression──┴─────────────────────────────────────────┤
```

The *literal_list* can be either of the following expressions:

- A single expression
  - If the single expression is enclosed with parentheses, all elements of the vector are initialized to the specified value.
  - If the single expression is enclosed with braces, the first element of the vector is initialized to the specified value, and the remaining elements of the vector are initialized to 0.
- A comma-separated list of expressions

  Each element of the vector is initialized to the respectively specified value.
  - If the comma-separated list of expressions is enclosed with parentheses, the number of constant expressions must match the number of elements in the vector.
  - If the comma-separated list of expressions is enclosed with braces, the number of constant expressions can be equal to or less than the number of elements in the vector. If the number of constant expressions is less than the number of elements in the vector, the values of the unspecified elements are 0.

The following table lists the supported vector literals and how the compiler interprets them to determine their values.

Table 18. Vector literals

Syntax	Interpretation
(vector signed char)(*int*)	A list of 16 signed 8-bit quantities that all have the value of the single integer.
(vector signed char)(*int*, ...) (vector signed char){*int*, ...}	A list of 16 signed 8-bit quantities with the values specified by the 16 integers.
(vector unsigned char)(*unsigned int*)	A list of 16 unsigned 8-bit quantities that all have the value of the single integer.
(vector bool char)(*unsigned int*)[1]	
(vector unsigned char)(*unsigned int*, ...) (vector unsigned char){*unsigned int*, ...}	A list of 16 unsigned 8-bit quantities with the values specified by the 16 integers.
(vector bool char)(*unsigned int*, ...)[1] (vector bool char){*unsigned int*, ...}[1]	
(vector signed short)(*int*)	A list of 8 signed 16-bit quantities that all have the value of the single integer.
(vector signed short)(*int*, ...) (vector signed short){*int*, ...}	A list of 8 signed 16-bit quantities with the values specified by the 8 integers.
(vector unsigned short)(*unsigned int*)	A list of 8 unsigned 16-bit quantities that all have the value of the single integer.
(vector bool short)(*unsigned int*)[1]	
(vector unsigned short)(*unsigned int*, ...) (vector unsigned short){*unsigned int*, ...}	A list of 8 unsigned 16-bit quantities with the values specified by the 8 integers.
(vector bool short)(*unsigned int*, ...)[1] (vector bool short){*unsigned int*, ...}[1]	
(vector signed int)(*int*)	A list of 4 signed 32-bit quantities that all have the value of the single integer.

Table 18. Vector literals (continued)

Syntax	Interpretation
(vector signed int)(*int*, ...) (vector signed int){*int*, ...}	A list of 4 signed 32-bit quantities with the values specified by the 4 integers.
(vector unsigned int)(*unsigned int*)	A list of 4 unsigned 32-bit quantities that all have the value of the single integer.
(vector bool int)(*unsigned int*)[1]	
(vector unsigned int)(*unsigned int*, ...) (vector unsigned int){*unsigned int*, ...}	A list of 4 unsigned 32-bit quantities with the values specified by the 4 integers.
(vector bool int)(*unsigned int*, ...)[1] (vector bool int){*unsigned int*, ...}[1]	
(vector signed long long)(*signed long long*)	A list of 2 signed 64-bit quantities that both have the value of the single long long integer.
(vector signed long long)(*signed long long*, ...) (vector signed long long){*signed long long*, ...}	A list of 2 signed 64-bit quantities with the values specified by the 2 long long integers.
(vector unsigned long long)(*unsigned long long*)	A list of 2 unsigned 64-bit quantities that both have the value of the single long long integer.
(vector unsigned long long)(*unsigned long long*, ...) (vector unsigned long long){*unsigned long long*, ...}	A list of 2 unsigned 64-bit quantities with the values specified by the 2 unsigned long long integers.
(vector bool long long)(*unsigned long long*)[1]	A list of 2 boolean 64-bit quantities that both have the value specified by the single unsigned long long integer.
(vector bool long long)(*unsigned long long*, ...)[1] (vector bool long long){*unsigned long long*, ...}[1]	A list of 2 boolean 64-bit quantities with the values specified by the 2 unsigned long long integers.
(vector double)(*double*)	A list of 2 64-bit IEEE-754 double-precision floating-point quantities that both have the value of the single double-precision floating-point value.
(vector double)(*double*, *double*) (vector double){*double*, *double*}	A list of 2 64-bit IEEE-754 double-precision floating-point quantities with the values specified by the 2 double-precision floating-point values.

Note:

1. The value of an element of a vector bool type is FALSE if each bit of the element is set to 0 and TRUE if each bit of the element is set to 1. Otherwise, it is neither TRUE nor FALSE.

You can initialize vector types with vector literals. For example:

```
vector int vi1 = (vector int)(1);    // It initializes all four elements to 1.
vector int vi2 = (vector int){1};    // It initializes the first element to 1
                                     // and all the other elements to 0.
vector int vi3 = (vector int){1, 2}  // It initializes the first element to 1,
                                     // the second element to 2, and the third
                                     // and fourth elements to 0.
```

You can cast vector literals using the cast operator (). Enclosing the vector literal to be cast in parentheses can improve the readability of the code. For example, you can use the following code to cast a `vector signed int` literal to a `vector unsigned char` literal:

```
(vector unsigned char)((vector signed int)(-1, -1, 0, 0))
```

Related information:

"Cast expressions" on page 68

"Initialization of vectors (IBM extension)"

Initialization of vectors (IBM extension)

You can initialize a vector type using a vector literal, an expression of the same vector type, or an initializer list.

In the following code example, a vector literal is used to initialize a vector.

```
vector unsigned int v1 = (vector unsigned int)(10);
```

In the following code example, an expression is used to initialize a vector of the same vector type.

```
vector unsigned int v1 = (vector unsigned int)(10);
vector unsigned int v2;
v2 = v1;
```

You can also initialize a vector type with an initializer list according to the following syntax.

```
►►—vector_type——identifier—=—┬—(—initializer_list—)—┬—;——————————————►◄
                             └—{—initializer_list—}—┘
```

An initializer list that is enclosed in parentheses must have the same number of values as the number of elements of the vector type. The number of values in a braced initializer list must be less than or equal to the number of elements of the vector type. Any element that is not explicitly initialized by the initializer list is initialized to zero.

In the following example, each vector is initialized using a initializer list:

```
vector unsigned int v1 = {1};  // It initializes the first element of v1 to 1
                               // and the remaining three elements to zero.

vector unsigned int v2 = {1, 2};  // It initializes the first element of v2 to 1,
                                  // the second element to 2,
                                  // and the remaining two elements to zero.

vector unsigned int v3 = {1, 2, 3, 4};  // v3 is {1, 2, 3, 4}.
```

Unlike vector literals, the expressions in the initializer list do not have to be constant expressions unless the initialized vector variable has static duration. Thus, the following code is valid:

```
int i=1;
int function() { return 2; }
int main()
{
   vector unsigned int v1 = {i, function()};
   return 0;
}
```

Related information:

"Vector literals (IBM extension)" on page 64

typedef definitions for vector types (IBM extension)

When vector programming support is enabled, you can define your own identifiers for vector types by using `typedef` definitions.

The self-defined type identifiers can be used in place of the original type specifiers, except for declaring other vectors types. The following example illustrates a typical usage of `typedef` definitions with vector types:

```
typedef vector unsigned short vint4;
vint4 v1;
```

Related information in the *XL C/C++ Compiler Reference*

-mzvector

Pointers (IBM extension)

Pointer arithmetic and pointer dereferencing are extended to support vector data types.

You can perform pointer arithmetic on vector data types. For example, if pointer `p` points to a vector, the result of the operation `p+1` is a pointer that points to the next vector.

When you dereference a pointer to a vector data type, the standard behavior is either a load or a copy of the corresponding type.

Expressions and operators (IBM extension)

The C/C++ language is extended to support expressions and operators on vector data types.

Compound literal expressions

A compound literal is a postfix expression that provides an unnamed object whose value is given by an initializer list. You can pass parameters to functions without creating temporary variables.

In the following syntax, *type_name* can be any vector type.

Compound literal syntax

```
►►—(—type_name—)—{—initializer_list—}——————————————————————————►◄
```

Cast expressions

The cast operator, `()`, is extended to support explicit type conversions from one vector data type to another vector data type. The exact same bit pattern is retained from the argument that is cast, and no conversion on the vector element value takes place.

Cast expression syntax for vectors

```
►►—(—vec_data_type—)—vec_expression————————————————————————————►◄
```

You cannot perform casting between scalar types and vector types. To copy data between scalar and vector variables, you can use the vector subscripting operator, `[]`. Alternatively, you can first define a union that consists of a vector variable and an equivalent array of the scalar type and then copy the data using the union.

Unary expressions

Some unary operators that were used with primitive data types are extended to support vector data types.

Unary operators

Vector data types can use these unary operators that are used with primitive data types: &, +, -, and ~.

The following table lists the vector data types that can be used with the address operator (&), unary plus operator (+), unary minus operator (-), and bitwise negation operator (~).

Table 19. Supported vector data types for &, +, -, and ~

Operator	Signed integer vector types	Unsigned integer vector types	Type `vector double`	Bool vector types
&	Yes	Yes	Yes	Yes
+ [1]	Yes	Yes	Yes	No
- [1]	Yes	No	Yes	No
~ [1]	Yes	Yes	No	Yes

Note:

1. Each element in the vector has the operation applied to it. The operator require compatible types to be used as operands unless otherwise stated. The operator is not supported at global scope or for objects that have static duration. There is no constant folding. The operator might not be portable.

Increment operator ++ and decrement operator --

When the increment operator or decrement operator is used with vector data types, each element in the vector has the operation applied to it.

Both operators require compatible types to be used as operands unless otherwise stated. Both operators are not supported at global scope or for objects that have static duration. There is no constant folding. These two operators might not be portable.

The following table lists the vector data types that can be used with the increment operator (++) and decrement operator (--).

Table 20. Supported vector data types for ++ and --

Operator	Integer vector types	Type `vector double`	Bool vector types
++	Yes	Yes	No
--	Yes	Yes	No

The `__alignof__` operator

The `__alignof__` operator is a language extension to C99 and C++03, which returns the alignment of its operand.

When vector programming support is enabled, operator `__alignof__` can accept an operand that is of a vector type. See the following example.

```
vector unsigned int v1 = (vector unsigned int)(10);
vector unsigned int *pv1 = &v1;
__alignof__(v1);   // Alignment of a vector type is 8.
__alignof__(&v1);  // Alignment of the address of a vector is 4.
__alignof__(*pv1); // Alignment of the dereferenced pointer to a vector is 8.
__alignof__(pv1);  // Alignment of the pointer to a vector is 4.
__alignof__(vector signed char); // Alignment of a vector type is 8.
```

When `__attribute__((aligned))` is used to increase the alignment of a variable of vector type, the value that is returned by the `__alignof__` operator is the alignment factor that is specified by `__attribute__((aligned))`.

The `sizeof` operator

The `sizeof` operator yields the size of its operand in bytes.

When vector programming support is enabled, the operand of the `sizeof` operator can be a vector variable, a vector type, or the result of dereferencing a pointer to a vector type. In these cases, the return value of `sizeof` is always 16. For example,

```
vector bool int v1;
vector bool int *pv1 = &v1;
sizeof(v1);             // For a vector type, the result is 16.
sizeof(*pv1);           // For the dereferenced pointer to a vector,
                        // the result is 16.
sizeof(vector bool int); // For a vector type, the result is 16.

sizeof(pv1);            // For the pointer to a vector,
                        // the result is 4 with ILP32 or 8 with ILP64.
sizeof(&v1);            // For the address of a vector,
                        // the result is 4 with ILP32 or 8 with ILP64.
```

Related information in the *XL C/C++ Compiler Reference*

-mzvector

The `typeof` operator

The `typeof` operator returns the type of its argument, which can be an expression or a type. The alternative spelling of the keyword, `__typeof__`, is recommended.

When vector programming support is enabled, the `typeof` operator is extended to accept an operand that is of a vector type.

Related information in the *XL C/C++ Compiler Reference*

-mzvector

The `vec_step` operator

The `vec_step` operator takes an operand that is of a vector type. The operator returns an integer value that represents the amount by which a pointer to a vector element must be incremented to move by 16 bytes, which is the size of a vector.

The following table provides a summary of the increment values by data type.

Table 21. Increment values for `vec_step` by data type

vec_step expresssion	**Value**

`vec_step(vector signed char)`
`vec_step(vector unsigned char)`
`vec_step(vector bool char)`
	16

Table 21. Increment values for `vec_step` by data type (continued)

`vec_step` expresssion	Value

`vec_step(vector signed short)`
`vec_step(vector unsigned short)`
`vec_step(vector bool short)`
	8

`vec_step(vector signed int)`
`vec_step(vector unsigned int)`
`vec_step(vector bool int)`
	4

`vec_step(vector signed long long)`
`vec_step(vector unsigned long long)`
`vec_step(vector bool long long)`
	2

`vec_step(vector double)`	2

Binary expressions

Some binary operators that are used with primitive data types are extended to support vector data types.

The following table provides a summary of the accepted vector data types for each operator.

Table 22. Binary operators

Operator	Integer vector types	Type `vector double`	Bool vector types	Note
=	Yes	Yes	Yes	The operation applies to each element in the first operand and the element of the same index in the second operand.
+	Yes	Yes	Yes	
-	Yes	Yes	Yes	
*	Yes	Yes	No	
/	Yes	Yes	No	
%	Yes	No	No	
&	Yes	Yes	Yes	
^	Yes	Yes	Yes	
|	Yes	Yes	Yes	
<<	Yes	No	No	
>>	Yes	No	No	
[] [1]	Yes	Yes	Yes	
==	Yes	Yes	Yes	Each operation is applied to each set of the elements that are in the same index of the first and the second operands. The final result is obtained by applying the AND operator to the results of each set of the corresponding elements.
!=	Yes	Yes	Yes	
<	Yes	Yes	Yes	
>	Yes	Yes	Yes	
<=	Yes	Yes	Yes	
>=	Yes	Yes	Yes	

Table 22. Binary operators (continued)

Operator	Integer vector types	Type `vector double`	Bool vector types	Note

Notes:

1. The [] operator returns the vector element at the index specified. If the index specified is outside the valid range, the behavior is undefined.
2. All operators require compatible types to be used as operands unless otherwise stated. The operators are not supported at global scope or for objects that have static duration. There is no constant folding. The operators might not be portable.

Assignment operator =

An assignment operator stores a value in the object that is designated by the left operand.

If either side of an assignment expression `a = b` is of a vector type, both sides of the expression, `a` and `b`, must be of the same vector type. Otherwise, the expression is invalid, and the compiler reports an error about the data type inconsistency.

Addition operator +

The addition operator (+) yields the sum of its operands.

The following table lists the accepted vector data types of operands and the result data types:

Table 23. Accepted vector data types for the addition operator and result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
		vector bool char
	vector bool char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector unsigned char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector bool short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector unsigned short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector bool int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long

Table 23. Accepted vector data types for the addition operator and result data types (continued)

Result type	Left operand type	Right operand type
vector double	vector double	vector double

Subtraction operator -

The subtraction operator (-) yields the difference of its operands.

The following table lists the accepted vector data types of operands and the result data types:

Table 24. Accepted vector data types for the subtraction operator and result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
		vector bool char
	vector bool char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector unsigned char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector bool short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector unsigned short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector bool int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector double	vector double	vector double

Multiplication operator *

The multiplication operator (*) yields the product of its operands.

The size of the products is double the size of the vector elements of operands; however, only the least significant half of each product is assigned to the corresponding vector element in the result.

Note: This function is emulated on `vector long long`.

The following table lists the accepted vector data types of operands and the result data types:

Table 25. Accepted vector data types for the multiplication operator and result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector double	vector double	vector double

Division operator /

The division operator (/) yields the algebraic quotient of its operands.

Note: This function is emulated on integer vector values.

The following table lists the accepted vector data types of operands and the result data types:

Table 26. Accepted vector data types for the division operator and result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector double	vector double	vector double

Remainder operator %

The remainder operator (%) yields the remainder from the division of the left operand by the right operand.

The following table lists the accepted vector data types of operands and the result data types:

Table 27. Accepted vector data types for the remainder operator and result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int

Table 27. Accepted vector data types for the remainder operator and result data types (continued)

Result type	Left operand type	Right operand type
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long

Bitwise AND operator &

The bitwise AND operator (&) performs a bitwise AND operation of each bit of its first operand and the corresponding bit of the second operand.

The following table lists the accepted vector data types of operands and the result data types:

Table 28. Accepted vector data types for the bitwise AND operator and result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
		vector bool char
	vector bool char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector unsigned char
vector bool char	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector bool short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector unsigned short
vector bool short	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector bool int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector unsigned int
vector bool int	vector bool int	vector bool int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector bool long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector unsigned long long
vector bool long long	vector bool long long	vector bool long long

Table 28. Accepted vector data types for the bitwise AND operator and result data types (continued)

Result type	Left operand type	Right operand type
vector double	vector bool long long	vector double
	vector double	vector double
		vector bool long long

Bitwise exclusive OR operator ^

The bitwise exclusive OR operator (^) performs a bitwise exclusive OR operation of each bit of its first operand and the corresponding bit of the second operand.

Note: `vector double` does not cause an IEEE exception.

The following table lists the accepted vector data types of operands and the result data types:

Table 29. Accepted vector data types for the bitwise exclusive OR operator and result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
		vector bool char
	vector bool char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector unsigned char
vector bool char	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector bool short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector unsigned short
vector bool short	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector bool int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector unsigned int
vector bool int	vector bool int	vector bool int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector bool long long	vector signed long long

Table 29. Accepted vector data types for the bitwise exclusive OR operator and result data types (continued)

Result type	Left operand type	Right operand type
vector unsigned long long	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector unsigned long long
vector bool long long	vector bool long long	vector bool long long
vector double	vector bool long long	vector double
	vector double	vector double
		vector bool long long

Bitwise inclusive OR operator |

The bitwise inclusive OR operator (`|`) performs a bitwise inclusive OR operation of each bit of its first operand and the corresponding bit of the second operand.

Note: `vector double` does not cause an IEEE exception.

The following table lists the accepted vector data types of operands and the result data types:

Table 30. Accepted vector data types for the bitwise inclusive OR operator and result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
		vector bool char
	vector bool char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector unsigned char
vector bool char	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector bool short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector unsigned short
vector bool short	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector bool int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector unsigned int
vector bool int	vector bool int	vector bool int

Table 30. Accepted vector data types for the bitwise inclusive OR operator and result data types (continued)

Result type	Left operand type	Right operand type
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector bool long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector unsigned long long
vector bool long long	vector bool long long	vector bool long long
vector double	vector bool long long	vector double
	vector double	vector double
		vector bool long long

Bitwise left shift operator <<

The bitwise left shift operator (<<) performs a left shift operation for each element of a vector.

If the right operand is a vector, each element of the result vector is the result of left shifting the corresponding element of the left operand by the number of bits specified by the corresponding element of the right operand. If the right operand is of type `unsigned long`, each element of the result vector is the result of left shifting the number of bits specified by the right operand.

If the value used as the shift bit is greater than the number of bits in the element of the left operand, *n*, the compiler uses the value modulo *n* as the shift bit. The bits that are shifted out are replaced by zeros.

The following table lists the accepted vector data types of operands and the result data types:

Table 31. Accepted vector data types for the bitwise left shift operator and result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector unsigned char
vector unsigned char	vector unsigned char	
vector signed short	vector signed short	vector unsigned short
vector unsigned short	vector unsigned short	
vector signed int	vector signed int	vector unsigned int
vector unsigned int	vector unsigned int	
vector signed long long	vector signed long long	vector unsigned long long
vector unsigned long long	vector unsigned long long	

Table 31. Accepted vector data types for the bitwise left shift operator and result data types (continued)

Result type	Left operand type	Right operand type
vector signed char	vector signed char	unsigned long
vector unsigned char	vector unsigned char	
vector signed short	vector signed short	
vector unsigned short	vector unsigned short	
vector signed int	vector signed int	
vector unsigned int	vector unsigned int	
vector signed long long	vector signed long long	
vector unsigned long long	vector unsigned long long	

Bitwise right shift operator >>

The bitwise right shift operator (>>) performs an algebraic right shift operation for each element of a vector.

If the right operand is a vector, each element of the result vector is the result of right shifting the corresponding element of the left operand by the number of bits specified by the corresponding element of the right operand. If the right operand is of type `unsigned long`, each element of the result vector is the result of right shifting the number of bits specified by the right operand.

If the value used as the shift bit is greater than the number of bits in the element of the left operand, *n*, the compiler uses the value modulo *n* as the shift bit.

If the left operand is an unsigned vector, the bits that are shifted out are replaced by zeros. If the left operand is a signed vector, the bits that are shifted out are replaced by copies of the most significant bit of the element of the left operand.

The following table lists the accepted vector data types of operands and the result data types:

Table 32. Accepted vector data types for the bitwise right shift operator and result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
		vector unsigned char
vector signed short	vector signed short	vector signed short
		vector unsigned short
vector signed int	vector signed int	vector signed int
		vector unsigned int
vector signed long long	vector signed long long	vector signed long long
		vector unsigned long long
vector signed char	vector signed char	unsigned long
vector signed short	vector signed short	
vector signed int	vector signed int	
vector signed long long	vector signed long long	

Table 32. Accepted vector data types for the bitwise right shift operator and result data types (continued)

Result type	Left operand type	Right operand type
vector unsigned char	vector unsigned char	vector unsigned char
vector unsigned short	vector unsigned short	vector unsigned short
vector unsigned int	vector unsigned int	vector unsigned int
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector unsigned char	vector unsigned char	unsigned long
vector unsigned short	vector unsigned short	
vector unsigned int	vector unsigned int	
vector unsigned long long	vector unsigned long long	

Vector subscripting operator []

The subscripting operator ([]) accesses individual elements of a vector, similar to how array elements are accessed.

To access an element of a vector, append a pair of brackets that contains the index of the wanted element to the vector. The index of the first element is 0. The type of the result is the type of the elements that are contained in the vector.

Note: If the specified index is outside of the valid range, the behavior is undefined.

See the following example for the usage of the subscripting operator:

```
vector unsigned int v1 = {1,2,3,4};
unsigned int u1, u2, u3, u4;
u1 = v1[0]; // The value of u1 is 1.
u2 = v1[1]; // The value of u2 is 2.
u3 = v1[2]; // The value of u3 is 3.
u4 = v1[3]; // The value of u4 is 4.
```

Equality operator ==

The equality operator (==) tests whether the corresponding elements of two vectors are equal.

If an element of the left operand is equal to the corresponding element of the right operand, the corresponding element in the result vector is -1. Otherwise, the corresponding element in the result vector is 0.

The following table lists the accepted vector data types of operands and the result data types:

Table 33. Accepted vector data types for the equality operator and result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Inequality operator !=

The inequality operator (!=) tests whether the corresponding elements of two vectors are unequal.

If an element of the left operand is not equal to the corresponding element of the right operand, the corresponding element in the result vector is -1. Otherwise, the corresponding element in the result vector is 0.

The following table lists the accepted vector data types of operands and the result data types:

Table 34. Accepted vector data types for the inequality operator and the result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Relational less than operator <

The relational less than operator (<) tests whether each element of the left operand is less than the corresponding element of the right operand.

If an element of the left operand is less than the corresponding element of the right operand, the corresponding element in the result vector is -1. Otherwise, the corresponding element in the result vector is 0.

Note: If either of the operands is a signed integer vector, a signed comparison is performed.

The following table lists the accepted vector data types of operands and the result data types:

Table 35. Accepted vector data types for the relational less than operator and the result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Relational greater than operator >

The relational greater than operator (>) tests whether each element of the left operand is greater than the corresponding element of the right operand.

If an element of the left operand is greater than the corresponding element of the right operand, the corresponding element in the result vector is -1. Otherwise, the corresponding element in the result vector is 0.

Note: If either of the operands is a signed integer vector, a signed comparison is performed.

The following table lists the accepted vector data types of operands and the result data types:

Table 36. Accepted vector data types for the relational greater than operator and the result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Relational less than or equal to operator <=

The relational less than or equal to operator (<=) tests whether each element of the left operand is less than or equal to the corresponding element of the right operand.

If an element of the left operand is less than or equal to the corresponding element of the right operand, the corresponding element in the result vector is -1. Otherwise, the corresponding element in the result vector is 0.

Note: If either of the operands is a signed integer vector, a signed comparison is performed.

The following table lists the accepted vector data types of operands and the result data types:

Table 37. Accepted vector data types for the relational less than or equal to operator and the result data types

Result type	Left operand type	Right operand type
vector signed char	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Relational greater than or equal to operator >=

The relational greater than or equal to operator (>=) tests whether each element of the left operand is greater than or equal to the corresponding element of the right operand.

If an element of the left operand is greater than or equal to the corresponding element of the right operand, the corresponding element in the result vector is -1. Otherwise, the corresponding element in the result vector is 0.

Note: If either of the operands is a signed integer vector, a signed comparison is performed.

The following table lists the accepted vector data types of operands and the result data types:

Table 38. Accepted vector data types for the relational greater than or equal to operator and the result data types

Result type	**Left operand type**	**Right operand type**
vector signed char	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Extensions to runtime library functions

These runtime library functions are extended to support vector processing.

longjmp
Restores stack environment.

malloc
Allocates storage for variables.

Note: The **new** operator is also extended to support vector processing.

setjmp
Preserves stack environment.

va_arg , va_copy , va_end , and va_start
Accesses function arguments.

Vector built-in functions

You can use vector built-in functions to access and manipulate individual elements of vectors.

You can use vector built-in functions when the following conditions are met:
- The Linux distributions have vector support.
- Both the **-mzvector** option and the **-march=z13** option, or its equivalent, are in effect.
- The `builtins.h` or `vecintrinc.h` header file is included.

In this topic collection, the following pseudo code is used to represent the built-in function syntax:

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

where
- `d` represents the return value of the vector built-in function.
- `a`, `b`, and `c` represent the parameters of the vector built-in function.
- *`vec_builtin_name`* is the name of the vector built-in function.

For example, the syntax for vector built-in function `vector double = vec_xld2(long, double*)` is represented by `d = vec_xld2(a, b)`.

Valid parameter types and the return type of the vector built-in function are provided in the description of each vector built-in function.

Related information in the *XL C/C++ Compiler Reference*

-march (-qarch)

-mzvector

Summary of vector built-in functions

These tables summarize and categorize vector built-in functions.

Arithmetic functions

Table 39. Vector built-in functions for arithmetic

Function name	Short name description	More information
vec_abs	Vector Absolute Value	See details
vec_add_u128	Vector Add Unsigned 128-bit Values	See details
vec_addc	Vector Add Carry	See details
vec_addc_u128	Vector Add Compute Carry Unsigned 128-bit Values	See details
vec_adde_u128	Vector Add with Carry Unsigned 128-bit Values	See details
vec_addec_u128	Vector Add with Carry Compute Carry Unsigned 128-bit Values	See details
vec_andc	Vector AND with Complement	See details
vec_avg	Vector Average	See details
vec_checksum	Vector Checksum	See details
vec_gfmsum	Vector Galois Field Multiply Sum	See details
vec_gfmsum_128	Vector Galois Field Multiply Sum 128-bit Values	See details
vec_gfmsum_accum	Vector Galois Field Multiply Sum and Accumulate	See details
vec_gfmsum_accum_128	Vector Galois Field Multiply Sum and Accumulate 128-bit Values	See details
vec_madd	Vector Multiply Add	See details
vec_max	Vector Maximum	See details
vec_meadd	Vector Multiply and Add Even	See details
vec_mhadd	Vector Multiply and Add High	See details
vec_min	Vector Minimum	See details
vec_mladd	Vector Multiply and Add Low	See details
vec_moadd	Vector Multiply and Add Odd	See details
vec_msub	Vector Multiply Subtract	See details
vec_mule	Vector Multiply Even	See details
vec_mulh	Vector Multiply High	See details
vec_mulo	Vector Multiply Odd	See details
vec_nabs	Vector Negative Absolute	See details
vec_sqrt	Vector Square Root	See details
vec_sub_u128	Vector Subtract Unsigned 128-bit Values	See details
vec_subc	Vector Subtract Compute Borrow	See details
vec_subc_u128	Vector Subtract Compute Borrow Unsigned 128-bit Values	See details
vec_sube_u128	Vector Subtract with Borrow Unsigned 128-bit Values	See details
vec_subec_u128	Vector Subtract with Borrow Compute Borrow Unsigned 128-bit Values	See details
vec_sum_u128	Vector Sum Across Quadword	See details

Table 39. Vector built-in functions for arithmetic (continued)

Function name	Short name description	More information
vec_sum2	Vector Sum Across Doubleword	See details
vec_sum4	Vector Sum Across Word	See details

Comparison functions

Table 40. Vector built-in functions for comparing elements

Function name	Short name description	More information
vec_cmpeq	Vector Compare Equal	See details
vec_cmpeq_idx	Vector Compare Equal Index	See details
vec_cmpeq_idx_cc	Vector Compare Equal Index with Condition Code	See details
vec_cmpeq_or_0_idx	Vector Compare Equal or Zero Index	See details
vec_cmpeq_or_0_idx_cc	Vector Compare Equal or Zero Index with Condition Code	See details
vec_cmpge	Vector Compare Greater Than or Equal	See details
vec_cmpgt	Vector Compare Greater Than	See details
vec_cmple	Vector Compare Less Than or Equal	See details
vec_cmplt	Vector Compare Less Than	See details
vec_cmpne_idx	Vector Compare Not Equal Index	See details
vec_cmpne_idx_cc	Vector Compare Not Equal Index with Condition Code	See details
vec_cmpne_or_0_idx	Vector Compare Not Equal or Zero Index	See details
vec_cmpne_or_0_idx_cc	Vector Compare Not Equal or Zero Index with Condition Code	See details

Ranges comparison functions

Table 41. Vector built-in functions for comparing elements against ranges

Function name	Short name description	More information
vec_cmpnrg	Vector Compare Not in Ranges	See details
vec_cmpnrg_cc	Vector Compare Not in Ranges with Condition Code	See details
vec_cmpnrg_idx	Vector Compare Not in Ranges Index	See details
vec_cmpnrg_idx_cc	Vector Compare Not in Ranges Index with Condition Code	See details
vec_cmpnrg_or_0_idx	Vector Compare Not in Ranges or Zero Index	See details
vec_cmpnrg_or_0_idx_cc	Vector Compare Not in Ranges or Zero Index with Condition Code	See details
vec_cmprg	Vector Compare Ranges	See details
vec_cmprg_cc	Vector Compare Ranges with Condition Code	See details

Table 41. Vector built-in functions for comparing elements against ranges (continued)

Function name	Short name description	More information
vec_cmprg_idx	Vector Compare Ranges Index	See details
vec_cmprg_idx_cc	Vector Compare Ranges Index with Condition Code	See details
vec_cmprg_or_0_idx	Vector Compare Ranges or Zero Index	See details
vec_cmprg_or_0_idx_cc	Vector Compare Ranges or Zero Index with Condition Code	See details

Element searching functions

Table 42. Vector built-in functions for element searching

Function name	Short name description	More information
vec_find_any_eq	Vector Find Any Element Equal	See details
vec_find_any_eq_cc	Vector Find Any Element Equal with Condition Code	See details
vec_find_any_eq_idx	Vector Find Any Element Equal Index	See details
vec_find_any_eq_idx_cc	Vector Find Any Element Equal Index with Condition Code	See details
vec_find_any_eq_or_0_idx	Vector Find Any Element Equal or Zero Index	See details
vec_find_any_eq_or_0_idx_cc	Vector Find Any Element Equal or Zero Index with Condition Code	See details
vec_find_any_ne	Vector Find Any Element Not Equal	See details
vec_find_any_ne_cc	Vector Find Any Element Not Equal with Condition Code	See details
vec_find_any_ne_idx	Vector Find Any Element Not Equal Index	See details
vec_find_any_ne_idx_cc	Vector Find Any Element Not Equal Index with Condition Code	See details
vec_find_any_ne_or_0_idx	Vector Find Any Element Not Equal or Zero Index	See details
vec_find_any_ne_or_0_idx_cc	Vector Find Any Element Not Equal or Zero Index with Condition Code	See details

Gather and scatter functions

Table 43. Vector built-in functions for gathering and scattering elements

Function name	Short name description	More information
vec_extract	Vector Extract	See details
vec_gather_element	Vector Gather Element	See details
vec_insert	Vector Insert	See details
vec_insert_and_zero	Vector Insert and Zero	See details
vec_perm	Vector Permute	See details

Table 43. Vector built-in functions for gathering and scattering elements (continued)

Function name	Short name description	More information
vec_permi	Vector Permute Immediate	See details
vec_promote	Vector Promote	See details
vec_scatter_element	Vector Scatter Element	See details
vec_sel	Vector Select	See details

Mask generation functions

Table 44. Vector built-in functions for mask generation

Function name	Short name description	More information
vec_genmask	Vector Generate Byte Mask	See details
vec_genmasks_8	Vector Generate Mask (Byte)	See details
vec_genmasks_16	Vector Generate Mask (Halfword)	See details
vec_genmasks_32	Vector Generate Mask (Word)	See details
vec_genmasks_64	Vector Generate Mask (Doubleword)	See details

Copy until zero functions

Table 45. Vector built-in functions for copying elements until a zero is encountered

Function name	Short name description	More information
vec_cp_until_zero	Vector Copy Until Zero	See details
vec_cp_until_zero_cc	Vector Copy Until Zero with Condition Code	See details

Load and store functions

Table 46. Vector built-in functions for loading and storing vectors

Function name	Short name description	More information
vec_ld2f	Vector Load 2 Float	See details
vec_load_bndry	Vector Load to Block Boundary	See details
vec_load_len	Vector Load with Length	See details
vec_load_pair	Vector Load Pair	See details
vec_st2f	Vector Store 2 Float	See details
vec_store_len	Vector Store with Length	See details
vec_xld2	Vector Load 2 Doubleword	See details
vec_xlw4	Vector Load 4 Word	See details
vec_xstd2	Vector Store 2 Doubleword	See details
vec_xstw4	Vector Store 4 Word	See details

Logical calculation functions

Table 47. Vector built-in functions for logical calculation

Function name	Short name description	More information
vec_cntlz	Vector Count Leading Zeros	See details
vec_cnttz	Vector Count Trailing Zeros	See details
vec_nor	Vector NOR	See details
vec_popcnt	Vector Population Count	See details

Merge functions

Table 48. Vector built-in functions for merging vectors

Function name	Short name description	More information
vec_mergeh	Vector Merge High	See details
vec_mergel	Vector Merge Low	See details

Pack and unpack functions

Table 49. Vector built-in functions for packing and unpacking

Function name	Short name description	More information
vec_pack	Vector Pack	See details
vec_packs	Vector Pack Saturate	See details
vec_packs_cc	Vector Pack Saturate Condition Code	See details
vec_packsu	Vector Pack Saturated Unsigned	See details
vec_packsu_cc	Vector Pack Saturated Unsigned Condition Code	See details
vec_unpackh	Vector Unpack High Element	See details
vec_unpackl	Vector Unpack Low Element	See details

Replicate functions

Table 50. Vector built-in functions for replicating vector elements

Function name	Short name description	More information
vec_splat	Vector Splat	See details
vec_splat_s8	Vector Splat Signed Byte	See details
vec_splat_s16	Vector Splat Signed Halfword	See details
vec_splat_s32	Vector Splat Signed Word	See details
vec_splat_s64	Vector Splat Signed Doubleword	See details
vec_splat_u8	Vector Splat Unsigned Byte	See details
vec_splat_u16	Vector Splat Unsigned Halfword	See details
vec_splat_u32	Vector Splat Unsigned Word	See details
vec_splat_u64	Vector Splat Doubleword	See details

Table 50. Vector built-in functions for replicating vector elements (continued)

Function name	Short name description	More information
vec_splats	Vector Splats	See details

Rotate and shift functions

Table 51. Vector built-in functions for rotation and shift

Function name	Short name description	More information
vec_rl	Vector Element Rotate Left	See details
vec_rl_mask	Vector Element Rotate and Insert Under Mask	See details
vec_rli	Vector Element Rotate Left Immediate	See details
vec_slb	Vector Shift Left by Byte	See details
vec_sld	Vector Shift Left Double by Byte	See details
vec_sldw	Vector Shift Left Double by Word	See details
vec_sll	Vector Shift Left	See details
vec_srab	Vector Shift Right Arithmetic by Byte	See details
vec_sral	Vector Shift Right Arithmetic	See details
vec_srb	Vector Shift Right by Byte	See details
vec_srl	Vector Shift Right	See details

Rounding and conversion functions

Table 52. Vector built-in functions for rounding and conversion

Function name	Short name description	More information
vec_ceil	Vector Ceiling	See details
vec_ctd	Vector Convert to Double	See details
vec_ctsl	Vector Convert to Signed Long Long	See details
vec_ctul	Vector Convert to Unsigned Long Long	See details
vec_extend_s64	Vector Sign Extend to Doubleword	See details
vec_floor	Vector Floor	See details
vec_round	Vector Round to Nearest	See details
vec_roundc	Vector Round to Current	See details
vec_roundm	Vector Round toward Negative Infinity	See details
vec_roundp	Vector Round toward Positive Infinity	See details
vec_roundz	Vector Round toward Zero	See details
vec_trunc	Vector Truncate	See details

Testing functions

Table 53. Vector built-in functions for testing

Function name	Short name description	More information
vec_fp_test_data_class	Vector Floating-Point Test Data Class	See details
vec_test_mask	Vector Test under Mask	See details

All elements predication functions

Table 54. Vector built-in functions to judge whether all elements meet the search criteria

Function name	Short name description	More information
vec_all_eq	All Elements Equal	See details
vec_all_ge	All Elements Greater Than or Equal	See details
vec_all_gt	All Elements Greater Than	See details
vec_all_le	All Elements Less Than or Equal	See details
vec_all_lt	All Elements Less Than	See details
vec_all_nan	All Elements Not a Number	See details
vec_all_ne	All Elements Not Equal	See details
vec_all_nge	All Elements Not Greater Than or Equal	See details
vec_all_ngt	All Elements Not Greater Than	See details
vec_all_nle	All Elements Not Less Than or Equal	See details
vec_all_nlt	All Elements Not Less Than	See details
vec_all_numeric	All Elements Numeric	See details

Any element predication functions

Table 55. Vector built-in functions to judge whether any element meets the search criteria

Function name	Short name description	More information
vec_any_eq	Any Element Equal	See details
vec_any_ge	Any Element Greater Than or Equal	See details
vec_any_gt	Any Element Greater Than	See details
vec_any_le	Any Element Less Than or Equal	See details
vec_any_lt	Any Element Less Than	See details
vec_any_nan	Any Element Not a Number	See details
vec_any_ne	Any Element Not Equal	See details
vec_any_nge	Any Element Not Greater Than or Equal	See details
vec_any_ngt	Any Element Not Greater Than	See details
vec_any_nle	Any Element Not Less Than or Equal	See details
vec_any_nlt	Any Element Not Less Than	See details
vec_any_numeric	Any Element Numeric	See details

Arithmetic functions

This topic collection describes built-in functions for arithmetic.

vec_abs: Vector Absolute Value

Purpose

Returns a vector that contains the absolute values of the elements of a given vector.

Syntax

```
d = vec_abs(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 56. Return and parameter types for vec_abs

d	a
vector signed char	vector signed char
vector signed short	vector signed short
vector signed int	vector signed int
vector signed long long	vector signed long long
vector double	vector double

Result value

The value of each element of `d` is the absolute value of the corresponding element of `a`.

Note: `vector double` does not cause an IEEE exception.

vec_add_u128: Vector Add Unsigned 128-bit Values

Purpose

Adds two vectors as 128-bit unsigned integers.

Syntax

```
d = vec_add_u128(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 57. Return and parameter types for vec_add_u128

d	a	b
vector unsigned char	vector unsigned char	vector unsigned char

Result value

`d` contains the low 128 bits of `a + b`.

vec_addc: Vector Add Carry

Purpose

Returns a vector that contains the carry produced by adding each set of the corresponding elements of two vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 58. Return and parameter types for vec_addc

d	a	b
vector unsigned char	vector unsigned char	vector unsigned char
vector unsigned short	vector unsigned short	vector unsigned short
vector unsigned int	vector unsigned int	vector unsigned int
vector unsigned long long	vector unsigned long long	vector unsigned long long

Result value

If a carry occurs in adding each set of the corresponding elements of `a` and `b`, the corresponding element of `d` is 1; otherwise, the corresponding element of `d` is 0.

vec_addc_u128: Vector Add Compute Carry Unsigned 128-bit Values

Purpose

Gets the carry produced by adding two vectors as 128-bit unsigned integers.

Syntax

```
d = vec_addc_u128(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 59. Return and parameter types for vec_addc_u128

d	a	b
vector unsigned char	vector unsigned char	vector unsigned char

Result value

`d` contains the carry produced by adding `a` and `b`. If a carry occurs on the addition, bit 127 of `d` is set to 1; otherwise, bit 127 of `d` is set to 0. All other bits of `d` are set to 0.

vec_adde_u128: Vector Add with Carry Unsigned 128-bit Values

Purpose

Adds two vectors as 128-bit unsigned integers with the carry bit from a previous operation.

Syntax

```
d = vec_adde_u128(a, b, c)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 60. Return and parameter types for vec_adde_u128

d	a	b	c
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char

Result value

`d` contains the low 128 bits of `a + b + (c & 1)`.

Note: Only the carry bit, which is bit 127, of `c` is used, and the other bits of `c` are ignored.

vec_addec_u128: Vector Add with Carry Compute Carry Unsigned 128-bit Values

Purpose

Gets the carry produced by adding two vectors as 128-bit unsigned integers and the carry bit from a previous operation.

Syntax

```
d = vec_addec_u128(a, b, c)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 61. Return and parameter types for vec_addec_u128

d	a	b	c
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char

Result value

`d` contains the carry out of `a + b + (c & 1)`. If a carry occurs on this addition, bit 127 of `d` is 1; otherwise, bit 127 of `d` is 0. All other bits of `d` are 0.

Note: Only the carry bit, which is bit 127, of `c` is used, and the other bits of `c` are ignored.

vec_andc: Vector AND with Complement

Purpose

Returns the bitwise AND of the first vector parameter with the bitwise complement of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 62. Return and parameter types for vec_andc

d	a	b
vector signed char	vector signed char	vector signed char
		vector bool char
	vector bool char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector unsigned char
vector bool char	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector bool short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector unsigned short
vector bool short	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector bool int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector unsigned int
vector bool int	vector bool int	vector bool int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector bool long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector unsigned long long
vector bool long long	vector bool long long	vector bool long long

Table 62. Return and parameter types for vec_andc (continued)

d	a	b
vector double	vector bool long long	vector double
	vector double	vector bool long long
		vector double

Result value

d is the bitwise AND of a with the bitwise complement of b.

vec_avg: Vector Average

Purpose

Returns a vector that contains the average values of each set of the corresponding elements of two vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 63. Return and parameter types for vec_avg

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long

Result value

The value of each element of d is the average value of the corresponding elements of a and b.

vec_checksum: Vector Checksum

Purpose

Returns a vector with the 1-indexed element containing a checksum that is computed from the summation of all vector elements of one vector and the 1-indexed element of another vector.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 64. Return and parameter types for vec_checksum

d	a	b
vector unsigned int	vector unsigned int	vector unsigned int

Result value

The value of `d[1]` is the checksum computed from the summation of all vector elements of `a` and `b[1]`. All other vector elements of `d` have a value of 0.

vec_gfmsum: Vector Galois Field Multiply Sum

Purpose

Performs a Galois field multiply sum on each element of two vectors.

The Galois field has an order of two. This multiplication is similar to standard binary multiplication, but it is exclusive ORed instead of adding the shifted multiplicand.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 65. Return and parameter types for vec_gfmsum

d	a	b
vector unsigned short	vector unsigned char	vector unsigned char
vector unsigned int	vector unsigned short	vector unsigned short
vector unsigned long long	vector unsigned int	vector unsigned int

Result value

Each element of `d` is the product in a Galois field of the corresponding elements of `a` and `b`. The resulting even-odd pairs of double element-sized products are exclusive ORed with each other and placed in the corresponding double element-sized elements of `d`.

vec_gfmsum_128: Vector Galois Field Multiply Sum 128-bit Values

Purpose

Performs a Galois field multiply sum on two vectors as 128-bit unsigned integers.

The Galois field has an order of two. This multiplication is similar to standard binary multiplication, but it is exclusive ORed instead of adding the shifted multiplicand.

Syntax

```
d = vec_gfmsum_128(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 66. Return and parameter types for vec_gfmsum_128

d	**a**	**b**
vector unsigned char	vector unsigned long long	vector unsigned long long

Result value

The value of `d` is the product in a Galois field of `a` and `b`. The resulting 128-bit products are exclusive ORed with each other.

vec_gfmsum_accum: Vector Galois Field Multiply Sum and Accumulate

Purpose

Performs a Galois field multiply sum and accumulate on each set of the corresponding elements of given vectors.

The Galois field has an order of two. This multiplication is similar to standard binary multiplication, but it is exclusive ORed instead of adding the shifted multiplicand.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 67. Return and parameter types for vec_gfmsum_accum

d	**a**	**b**	**c**
vector unsigned short	vector unsigned char	vector unsigned char	vector unsigned short
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int
vector unsigned long long	vector unsigned int	vector unsigned int	vector unsigned long long

Result value

Each element of `d` is the product in a Galois field of the corresponding elements of `a` and `b`. The resulting even-odd pairs of double element-sized products are exclusive ORed with each other and exclusive ORed with the corresponding double-wide element of `c`.

vec_gfmsum_accum_128: Vector Galois Field Multiply Sum and Accumulate 128-bit Values

Purpose

Performs a Galois field multiply sum and accumulate on given vectors as 128-bit unsigned integers.

The Galois field has an order of two. This multiplication is similar to standard binary multiplication, but it is exclusive ORed instead of adding the shifted multiplicand.

Syntax

```
d = vec_gfmsum_accum_128(a, b, c)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 68. Return and parameter types for vec_gfmsum_accum_128

d	a	b	c
vector unsigned char	vector unsigned long long	vector unsigned long long	vector unsigned char

Result value

The value of `d` is the product in a Galois field of `a` and `b`. The resulting 128-bit products are exclusive ORed with each other and exclusive ORed with `c`.

vec_madd: Vector Multiply Add

Purpose

Performs a fused multiply-and-add operation on each set of the corresponding elements of given vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 69. Return and parameter types for vec_madd

d	a	b	c
vector double	vector double	vector double	vector double

Result value

The value of each element of `d` is the result of adding the corresponding element of `c` to the product of the corresponding elements of `a` and `b`.

vec_max: Vector Maximum

Purpose

Returns a vector that contains the maximum values of each set of the corresponding elements of two vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 70. Return and parameter types for vec_max

d	**a**	**b**
vector signed char	vector signed char	vector signed char
		vector bool char
	vector bool char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector unsigned char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector bool short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector unsigned short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector bool int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector bool long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector unsigned long long
vector double	vector double	vector double

Result value

The value of each element of `d` is the maximum value of the corresponding elements of `a` and `b`.

Note: This function is emulated on `vector double`.

vec_meadd: Vector Multiply and Add Even

Purpose

Returns a vector that contains double element-sized results of performing a multiply-and-add operation on each set of the corresponding even-indexed elements of given vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 71. Return and parameter types for vec_meadd

d	a	b	c
vector signed short	vector signed char	vector signed char	vector signed short
vector unsigned short	vector unsigned char	vector unsigned char	vector unsigned short
vector signed int	vector signed short	vector signed short	vector signed int
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int
vector signed long long	vector signed int	vector signed int	vector signed long long
vector unsigned long long	vector unsigned int	vector unsigned int	vector unsigned long long

Result value

The value of each element of `d` is the result of adding the corresponding element of `c` to the product of the even-indexed elements of `a` and `b`. Each product is extended to the double size of elements of `a`.

Example

If `a` and `b` are of type `vector singed int` and `c` is of type `vector signed long long`, `d` is obtained as follows:

- The value of `b[0]` is the result of adding `c[0]` to the product of `a[0]` and `b[0]`.
- The value of `b[1]` is the result of adding `c[1]` to the product of `a[2]` and `b[2]`.

Both products are of type `unsigned long long`.

vec_mhadd: Vector Multiply and Add High

Purpose

Returns a vector that contains the most significant half, also known as the high half, of the double element-sized results of performing a multiply-and-add operation on each set of the corresponding elements of given vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 72. Return and parameter types for vec_mhadd

d	a	b	c
vector signed char	vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int

Result value

The value of each element of `d` is the result of adding the corresponding element of `c` to the most significant half of the product of the corresponding elements of `a` and `b`. Each product is extended to the double size of elements of `a`.

vec_min: Vector Minimum

Purpose

Returns a vector that contains the minimum values of each set of the corresponding elements of two vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 73. Return and parameter types for vec_min

d	a	b
vector signed char	vector signed char	vector signed char
		vector bool char
	vector bool char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector unsigned char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector bool short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector unsigned short

Table 73. Return and parameter types for vec_min (continued)

d	a	b
vector signed int	vector signed int	vector signed int
		vector bool int
	vector bool int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector bool long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector unsigned long long
vector double	vector double	vector double

Result value

The value of each element of `d` is the minimum value of the corresponding elements of `a` and `b`.

Note: This function is emulated on `vector double`.

vec_mladd: Vector Multiply and Add Low

Purpose

Returns a vector that contains the least significant half, also known as the low half, of the double element-sized results of performing a multiply-and-add operation on each set of the corresponding elements of given vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 74. Return and parameter types for vec_mladd

d	a	b	c
vector signed char	vector signed char	vector signed char	vector signed char
		vector unsigned char	vector unsigned char
	vector unsigned char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short	vector signed short
		vector unsigned short	vector unsigned short
	vector unsigned short	vector signed short	vector signed short

Table 74. Return and parameter types for vec_mladd (continued)

d	a	b	c
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int	vector signed int
		vector unsigned int	vector unsigned int
	vector unsigned int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int

Result value

The value of each element of `d` is the result of adding the corresponding element of `c` to the least significant half of the product of the corresponding elements of `a` and `b`. Each product is extended to the double size of elements of `a`.

vec_moadd: Vector Multiply and Add Odd

Purpose

Returns a vector that contains double element-sized results of performing a multiply-and-add operation on each set of the corresponding odd-indexed elements of given vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 75. Return and parameter types for vec_moadd

d	a	b	c
vector signed short	vector signed char	vector signed char	vector signed short
vector unsigned short	vector unsigned char	vector unsigned char	vector unsigned short
vector signed int	vector signed short	vector signed short	vector signed int
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int
vector signed long long	vector signed int	vector signed int	vector signed long long
vector unsigned long long	vector unsigned int	vector unsigned int	vector unsigned long long

Result value

The value of each element of `d` is the result of adding the corresponding element of `c` to the product of the odd-indexed elements of `a` and `b`. Each product is extended to the double size of elements of `a`.

vec_msub: Vector Multiply Subtract

Purpose

Performs a multiply-and-subtract operation on each set of the corresponding elements of given vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 76. Return and parameter types for vec_msub

d	a	b	c
vector double	vector double	vector double	vector double

Result value

The value of each element of `d` is the result of subtracting the corresponding element of `c` from the product of the corresponding elements of `a` and `b`.

vec_mule: Vector Multiply Even

Purpose

Returns a vector that contains double element-sized results of performing a multiply operation on each set of the corresponding even-indexed elements of two vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 77. Return and parameter types for vec_mule

d	a	b
vector signed short	vector signed char	vector signed char
vector unsigned short	vector unsigned char	vector unsigned char
vector signed int	vector signed short	vector signed short
vector unsigned int	vector unsigned short	vector unsigned short
vector signed long long	vector signed int	vector signed int
vector unsigned long long	vector unsigned int	vector unsigned int

Result value

The value of each element of `d` is the product of the corresponding even-indexed elements of `a` and `b`. Each product is extended to the double size of elements of `a`.

Example

The following figure illustrates the operation on operands of type `vector signed short` or `vector unsigned short`.

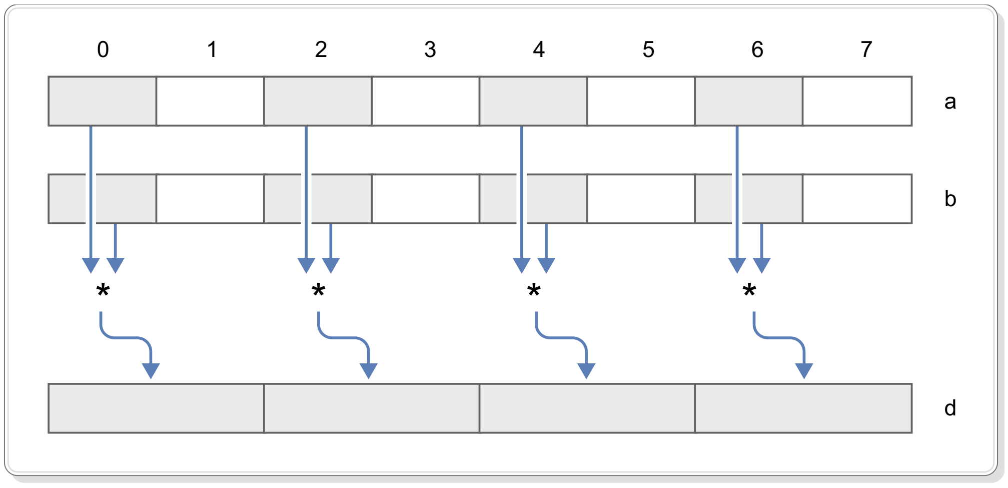


Figure 3. Even multiply of 16-bit integer elements

vec_mulh: Vector Multiply High

Purpose

Returns a vector that contains the most significant half, also known as the high half, of the results of performing a multiply operation on each set of the corresponding elements of two vectors.

Syntax

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

Return and parameter types

Table 78. Return and parameter types for vec_mulh

d	**a**	**b**
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int

Result value

The value of each element of `d` is the most significant half of the product of the corresponding elements of `a` and `b`. Each product is extended to the double size of elements of `a`.

vec_mulo: Vector Multiply Odd

Purpose

Returns a vector that contains double element-sized results of performing a multiply operation on each set of the corresponding odd-indexed elements of two vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 79. Return and parameter types for vec_mulo

d	a	b
vector signed short	vector signed char	vector signed char
vector unsigned short	vector unsigned char	vector unsigned char
vector signed int	vector signed short	vector signed short
vector unsigned int	vector unsigned short	vector unsigned short
vector signed long long	vector signed int	vector signed int
vector unsigned long long	vector unsigned int	vector unsigned int

Result value

The value of each element of `d` is the product of the corresponding odd-indexed elements of `a` and `b`. Each product is extended to the double size of elements of `a`.

Example

The following figure illustrates the operation on operands of type `vector signed short` or `vector unsigned short`.

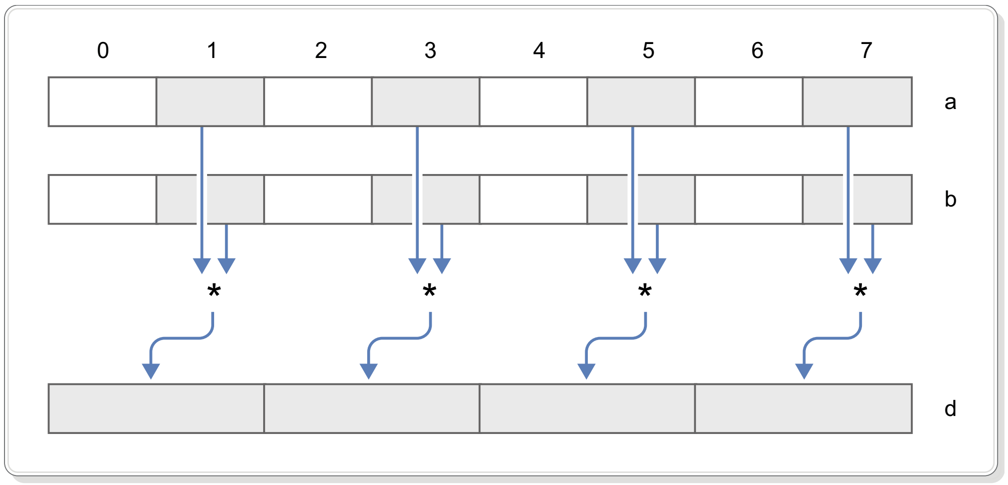


Figure 4. Odd multiply of 16-bit integer elements

vec_nabs: Vector Negative Absolute

Purpose

Returns a vector that contains the results of performing a negative-absolute operation on each element of a given vector.

Syntax

```
d = vec_nabs(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 80. Return and parameter types for vec_nabs

d	**a**
vector double	vector double

Result value

The value of each element of `d` is the negated value of the absolute value of the corresponding element of `a`.

Note: This built-in function does not cause an IEEE exception.

vec_sqrt: Vector Square Root

Purpose

Returns a vector that contains the square roots of the elements of a given vector.

Syntax

```
d = vec_sqrt(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 81. Return and parameter types for vec_sqrt

d	**a**
vector double	vector double

Result value

The value of each element of `d` is the square root of the corresponding element of `a`.

vec_sub_u128: Vector Subtract Unsigned 128-bit Values

Purpose

Subtracts one vector from another vector as 128-bit unsigned integers.

Syntax

```
d = vec_sub_u128(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 82. Return and parameter types for vec_sub_u128

d	a	b
vector unsigned char	vector unsigned char	vector unsigned char

Result value

`d` contains the low 128 bits of `a - b`.

vec_subc: Vector Subtract Compute Borrow

Purpose

Returns a vector that contains the borrow produced by subtracting each element of one vector from the corresponding element of another vector.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 83. Return and parameter types for vec_subc

d	a	b
vector unsigned char	vector unsigned char	vector unsigned char
vector unsigned short	vector unsigned short	vector unsigned short
vector unsigned int	vector unsigned int	vector unsigned int
vector unsigned long long	vector unsigned long long	vector unsigned long long

Result value

If a borrow occurs in subtracting an element of `b` from the corresponding element of `a`, the value of the corresponding element of `d` is 0; otherwise, the value of the corresponding element of `d` is 1.

vec_subc_u128: Vector Subtract Compute Borrow Unsigned 128-bit Values

Purpose

Gets the borrow produced by subtracting one vector from another vector as 128-bit unsigned integers.

Syntax

```
d = vec_subc_u128(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 84. Return and parameter types for vec_subc_u128

d	a	b
vector unsigned char	vector unsigned char	vector unsigned char

Result value

`d` contains the borrow produced by subtracting `b` from `a` as unsigned 128-bit integers. If a borrow occurs, bit 127 of `d` is 1; otherwise, bit 127 of `d` is 0. All other bits of `d` are set to 0.

vec_sube_u128: Vector Subtract with Borrow Unsigned 128-bit Values

Purpose

Returns a vector that contains the result of subtracting one vector from another vector as 128-bit unsigned integers and the borrow bit from a previous operation.

Syntax

```
d = vec_sube_u128(a, b, c)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 85. Return and parameter types for vec_sube_u128

d	a	b	c
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char

Result value

`d` contains the low 128 bits of `a - b - (c & 1)`.

Note: Only the borrow bit, which is bit 127, of `c` is used, and the other bits of `c` are ignored.

vec_subec_u128: Vector Subtract with Borrow Compute Borrow Unsigned 128-bit Values

Purpose

Returns a vector that contains the borrow produced in subtracting one vector from another vector as 128-bit integers and the borrow bit from a previous operation.

Syntax

```
d = vec_subec_u128(a, b, c)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 86. Return and parameter types for vec_subec_u128

d	a	b	c
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char

Result value

`d` contains the borrow out of `a - b - (c & 1)`. If a borrow occurs on this subtraction, bit 127 of `d` is 1; otherwise, bit 127 of `d` 0. All other bits of `d` are 0.

Note: Only the borrow bit of `c` is used, and the other bits of `c` are ignored.

vec_sum_u128: Vector Sum Across Quadword

Purpose

Sums up all the elements of one vector and the rightmost element of another vector.

Syntax

```
d = vec_sum_u128(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 87. Return and parameter types for vec_sum_u128

d	a	b
vector unsigned char	vector unsigned int	vector unsigned int
	vector unsigned long long	vector unsigned long long

Result value

For `vector unsigned int` operands, `d` is obtained as follows as an unsigned 128-bit integer:

```
d = a[0] + a[1] + a[2] + a[3] + b[3]
```

For `vector unsigned long long` operands, `d` is obtained as follows as an unsigned 128-bit integer:

```
d = a[0] + a[1] + b[1]
```

vec_sum2: Vector Sum Across Doubleword

Purpose

Sums up all the elements in each of the doubleword of one vector and the rightmost element of the corresponding doubleword of another vector.

Syntax

```
d = vec_sum2(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 88. Return and parameter types for vec_sum2

d	**a**	**b**
vector unsigned long long	vector unsigned short	vector unsigned short
	vector unsigned int	vector unsigned int

Result value

For `vector unsigned short` operands, `d` is obtained as follows:

```
d[0] = a[0] + a[1] + a[2] + a[3] + b[3]
d[1] = a[4] + a[5] + a[6] + a[7] + b[7]
```

For `vector unsigned int` operands, `d` is obtained as follows:

```
d[0] = a[0] + a[1] + b[1]
d[1] = a[2] + a[3] + b[3]
```

Example

The following figure illustrates the operation on operands of type `vector unsigned int`.

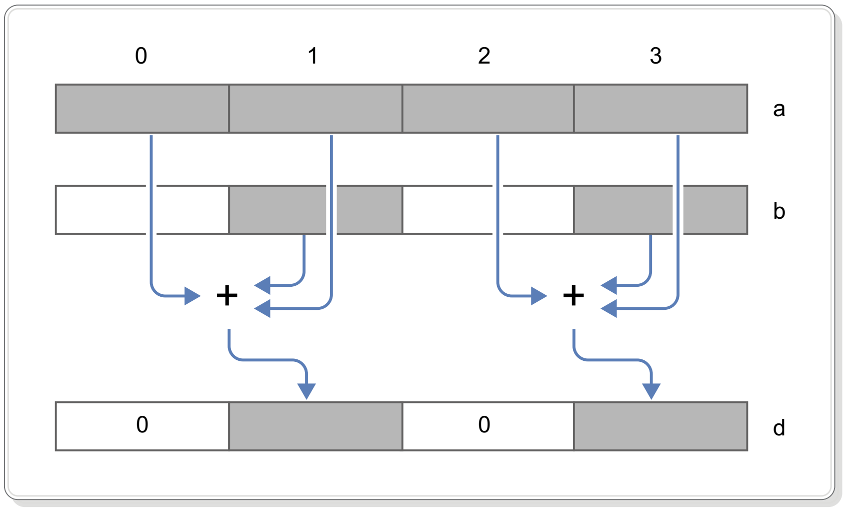


Figure 5. Two saturated sums of 32-bit integer elements

vec_sum4: Vector Sum Across Word

Purpose

Sums up all the elements in each of the word of one vector and the rightmost element of the corresponding word of another vector.

Syntax

```
d = vec_sum4(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 89. Return and parameter types for vec_sum4

d	a	b
vector unsigned int	vector unsigned char	vector unsigned char
	vector unsigned short	vector unsigned short

Result value

For `vector unsigned char` operands, `d` is obtained as follows:

```
d[0] = a[0] + a[1] + a[2] + a[3] + b[3]
d[1] = a[4] + a[5] + a[6] + a[7] + b[7]
d[2] = a[8] + a[9] + a[10] + a[11] + b[11]
d[3] = a[12] + a[13] + a[14] + a[15] + b[15]
```

For `vector unsigned short` operands, `d` is obtained as follows:

```
d[0] = a[0] + a[1] + b[1]
d[1] = a[2] + a[3] + b[3]
d[2] = a[4] + a[5] + b[5]
d[3] = a[6] + a[7] + b[7]
```

Comparison functions

This topic collection describes vector built-in functions that are used to compare elements.

vec_cmpeq: Vector Compare Equal

Purpose

Compares each set of the corresponding elements of two vectors for equality.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 90. Return and parameter types for vec_cmpeq

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
	vector bool char	vector bool char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
	vector bool short	vector bool short

Table 90. Return and parameter types for vec_cmpeq (continued)

d	a	b
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector bool int	vector bool int
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector bool long long	vector bool long long
	vector double	vector double

Result value

For each element of `d`, if the corresponding elements of `a` and `b` are equal, the value of each bit is 1. Otherwise, the value of each bit is 0.

vec_cmpeq_idx: Vector Compare Equal Index

Purpose

Compares each set of the corresponding elements of two vectors for equality and returns the byte index of the eligible element of the lowest position.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 91. Return and parameter types for vec_cmpeq_idx

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
	vector bool int	vector bool int

Result value

If none of the elements of `a` equals the corresponding element of `b`, the result is 16; otherwise, the result is the byte index of the lowest-position element of `a` that is equal to the corresponding element of `b`. The result is placed in byte element seven of `d`, and all other bytes of `d` are set to 0.

Examples

Example 1: No equal corresponding elements

In the following example, none of the elements of a equals the corresponding element of b, so the result is 16. Because the result is placed in byte element seven and elements of d are of type unsigned char, whose size is 1 byte, so the result, 16, is placed in d[7].

```
vector unsigned char a ={'a','a','a','a','a','a','a','a','a','a','a','a','a','a','a','a'};
vector unsigned char b ={'z','z','z','z','z','z','z','z','z','z','z','z','z','z','z','z'};
vector unsigned char d = vec_cmpeq_idx(a, b);
// d is {'0','0','0','0','0','0','0','16','0','0','0','0','0','0','0','0'}.
```

Example 2: One equal corresponding element of type unsigned char found

In the following example, elements of a and b are of 1 byte and a[3] equals b[3], so the result is 3. Similar to example 1, the result, 3, is placed in d[7].

```
vector unsigned char a ={'a','a','a','a','a','a','a','a','a','a','a','a','a','a','a','a'};
vector unsigned char b ={'z','z','z','a','z','z','z','z','z','z','z','z','z','z','z','z'};
vector unsigned char d = vec_cmpeq_idx(a, b);
// d is {'0','0','0','0','0','0','0','3','0','0','0','0','0','0','0','0'}.
```

Example 3: One equal corresponding element of type signed int found

In the following example, elements of a and b are of 4 bytes and a[3] equals b[3], so the byte index of a[3] is 12, the result of (3 - 0) * 4.

Elements of d are of type signed int, whose size is 4 bytes, so byte element seven is found in d[1]. d[1] is assigned the byte index 12.

```
vector signed int a ={1, 2, 3, 4};
vector signed int b ={5, 6, 7, 4};
vector signed int d = vec_cmpeq_idx(a, b);
// d is {0, 12, 0, 0}.
```

Example 4: More than one equal corresponding element found

In the following example, elements of a and b are of 1 byte. a[7] equals b[7], and a[11] equals b[11]. Because byte index 7 is lower than 11, so the result is 7. Similar to example 1, the result, 7, is placed in d[7].

```
vector unsigned char a ={'a','a','a','a','a','a','a','a','a','a','a','a','a','a','a','a'};
vector unsigned char b ={'z','z','z','z','z','z','z','a','z','z','z','a','z','z','z','z'};
vector unsigned char d = vec_cmpeq_idx(a, b);
// d is {'0','0','0','0','0','0','0','7','0','0','0','0','0','0','0','0'}.
```

vec_cmpeq_idx_cc: Vector Compare Equal Index with Condition Code

Purpose

Compares each set of the corresponding elements of two vectors for equality and returns the byte index of the eligible element of the lowest position. It also returns a condition code that indicates whether such an element is found.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 92. Return and parameter types for vec_cmpeq_idx_cc

d	a	b	c
vector signed char	vector signed char	vector signed char	int *
vector unsigned char	vector unsigned char	vector unsigned char	
	vector bool char	vector bool char	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
	vector bool short	vector bool short	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
	vector bool int	vector bool int	

Result value

If none of the elements of a equals the corresponding element of b, the result is 16; otherwise, the result is the byte index of the lowest-position element of a that is equal to the corresponding element of b. The result is placed in byte element seven of d, and all other bytes of d are set to 0.

If at least one element of a equals the corresponding element of b, the value that is referred to by c is set to 1; otherwise, the value that is referred to by c is set to 3.

vec_cmpeq_or_0_idx: Vector Compare Equal or Zero Index

Purpose

Compares each set of the corresponding elements of two vectors for equality and compares each element of one vector against 0. It returns the byte index of the eligible element of the lowest position.

Syntax

```
d = vec_cmpeq_or_0_idx(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 93. Return and parameter types for vec_cmpeq_or_0_idx

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
	vector bool short	vector bool short

Table 93. Return and parameter types for vec_cmpeq_or_0_idx (continued)

d	a	b
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
	vector bool int	vector bool int

Result value

If none of the elements of a equals the corresponding element of b and has a value of 0, the result is 16. Otherwise, the result is the byte index of the lowest-position element of a that is equal to the corresponding element of b or 0. The result is placed in byte element seven of d, and all other bytes of d are set to 0.

vec_cmpeq_or_0_idx_cc: Vector Compare Equal or Zero Index with Condition Code

Purpose

Compares each set of the corresponding elements of two vectors for equality and compares each element of one vector against 0. It returns the byte index of the eligible element of the lowest position. It also returns a condition code that indicates whether such an element is found.

Syntax

```
d = vec_cmpeq_or_0_idx_cc(a, b, c)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 94. Return and parameter types for vec_cmpeq_or_0_idx_cc

d	a	b	c
vector signed char	vector signed char	vector signed char	int *
vector unsigned char	vector unsigned char	vector unsigned char	
	vector bool char	vector bool char	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
	vector bool short	vector bool short	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
	vector bool int	vector bool int	

Result value

If none of the elements of a equals the corresponding element of b and has a value of 0, the result is 16. Otherwise, the result is the byte index of the lowest-position element of a that is equal to the corresponding element of b or 0. The result is placed in byte element seven of d, and all other bytes of d are set to 0.

The value that is referred to by `c` is determined as follows. The comparison is performed starting from `a[0]`.

- 0 if all the corresponding elements of `a` and `b` are unequal before value 0 is found in `a`, or if they happen concurrently.
- 1 if at least one element of `a` equals the corresponding element of `b` and if no elements of `a` have a value of 0.
- 2 if an equality occurs between the corresponding elements of `a` and `b` before value 0 is found in `a`.
- 3 if none of the elements of `a` equals the corresponding element of `b` and if no elements of `a` have a value of 0.

vec_cmpge: Vector Compare Greater Than or Equal

Purpose

Returns a vector that contains the results of a greater-than-or-equal-to comparison between the corresponding elements of two vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 95. Return and parameter types for vec_cmpge

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

Result value

For each element of `d`, if the value of the corresponding element of `a` is greater than or equal to the value of the corresponding element of `b`, the value of each bit is 1. Otherwise, the value of each bit is 0.

Note: This function is emulated on integer vector values.

vec_cmpgt: Vector Compare Greater Than

Purpose

Returns a vector that contains the results of a greater-than comparison between the corresponding elements of two vectors.

Syntax

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

Return and parameter types

Table 96. Return and parameter types for vec_cmpgt

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

Result value

For each element of `d`, if the value of the corresponding element of `a` is greater than the value of the corresponding element of `b`, the value of each bit is 1. Otherwise, the value of each bit is 0.

vec_cmple: Vector Compare Less Than or Equal

Purpose

Returns a vector that contains the results of a less-than-or-equal-to comparison between the corresponding elements of two vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 97. Return and parameter types for vec_cmple

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

Result value

For each element of d, if the value of the corresponding element of a is less than or equal to the value of the corresponding element of b, the value of each bit is 1. Otherwise, the value of each bit is 0.

Note: This function is emulated.

vec_cmplt: Vector Compare Less Than

Purpose

Returns a vector that contains the results of a less-than comparison between the corresponding elements of two vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 98. Return and parameter types for vec_cmplt

d	**a**	**b**
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

Result value

For each element of d, if the value of the corresponding element of a is less than the value of the corresponding element of b, the value of each bit is 1. Otherwise, the value of each bit is 0.

Note: This function is emulated.

vec_cmpne_idx: Vector Compare Not Equal Index

Purpose

Compares each set of the corresponding elements of two vectors for inequality and returns the byte index of the eligible element of the lowest position.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 99. Return and parameter types for vec_cmpne_idx

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
	vector bool int	vector bool int

Result value

If `a` and `b` are equal, the result is 16. Otherwise, the result is the byte index of the lowest-position element of `a` that is unequal to the corresponding element of `b`. The result is placed in byte element seven of `d`, and all other bytes of `d` are set to 0.

vec_cmpne_idx_cc: Vector Compare Not Equal Index with Condition Code

Purpose

Compares each set of the corresponding elements of two vectors for inequality and returns the byte index of the eligible element of the lowest position. It also returns a condition code that indicates whether such an element is found.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 100. Return and parameter types for vec_cmpne_idx_cc

d	a	b	c
vector signed char	vector signed char	vector signed char	int *
vector unsigned char	vector unsigned char	vector unsigned char	
	vector bool char	vector bool char	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
	vector bool short	vector bool short	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
	vector bool int	vector bool int	

Result value

If `a` and `b` are equal, the result is 16. Otherwise, the result is the byte index of the lowest-position element of `a` that is unequal to the corresponding element of `b`. The result is placed in byte element seven of `d`, and all other bytes of `d` are set to 0.

The value that is referred to by `c` is determined as follows:

- 1 if an inequality occurs and that element of `a` is less than the corresponding element of `b`.
- 2 if an inequality occurs and that element of `a` is greater than the corresponding element of `b`.
- 3 if `a` and `b` are equal.

vec_cmpne_or_0_idx: Vector Compare Not Equal or Zero Index

Purpose

Compares each set of the corresponding elements of two vectors for inequality and compares each element of one vector against 0. It returns the byte index of the eligible element of the lowest position.

Syntax

```
d = vec_cmpne_or_0_idx(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 101. Return and parameter types for vec_cmpne_or_0_idx

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
	vector bool short	vector bool short

Table 101. Return and parameter types for vec_cmpne_or_0_idx (continued)

d	a	b
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
	vector bool int	vector bool int

Result value

If a and b are equal and no elements of a have a value of 0, the result is 16. Otherwise, the result is the byte index of the lowest-position element of a that is unequal to the corresponding element of b or that is equal to 0.

vec_cmpne_or_0_idx_cc: Vector Compare Not Equal or Zero Index with Condition Code

Purpose

Compares each set of the corresponding elements of two vectors for inequality and compares each element of one vector against 0. It returns the byte index of the eligible element of the lowest position. It also returns a condition code that indicates whether such an element is found.

Syntax

```
d = vec_cmpne_or_0_idx_cc(a, b, c)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 102. Return and parameter types for vec_cmpne_or_0_idx_cc

d	a	b	c
vector signed char	vector signed char	vector signed char	int *
vector unsigned char	vector unsigned char	vector unsigned char	
	vector bool char	vector bool char	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
	vector bool short	vector bool short	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
	vector bool int	vector bool int	

Result value

If a and b are equal and no elements of a have a value of 0, the result is 16. Otherwise, the result is the byte index of the lowest-position element of a that is unequal to the corresponding element of b or that is equal to 0. The result is placed in byte element seven of d, and all other bytes of d are set to 0.

The value that is referred to by c is determined as follows. The comparison is performed starting from a[0].

- 0 if value 0 is found in `a` and `b` at the same index before an inequality between the corresponding elements of `a` and `b` occurs.
- 1 if an inequality occurs and that element of `a` is less than the corresponding element of `b`, and if prior to the inequality all elements of `a` and `b` are not 0.
- 2 if an inequality occurs and that element of `a` is greater than the corresponding element of `b`, and if prior to the inequality all elements of `a` and `b` are not 0.
- 3 if `a` and `b` are equal and no elements of `a` and `b` have a value of 0.

Range comparison functions

This topic collection describes vector built-in functions that are used to compare elements against ranges.

vec_cmpnrg: Vector Compare Not in Ranges

Purpose

Checks whether the value of each element of a vector is not within any of the specified ranges.

Syntax

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

Usage

This function checks whether the value of each element of `a` is not within any of the ranges that are specified by `b` and `c` as follows:

- Each even-odd element pairs of `b` define values for the limits of the ranges.
- The corresponding even-odd pairs of elements of `c` control which comparison is to be done. The following table lists valid values for elements of `c` of each possible data type for different comparison operations.

Table 103. Element values of parameter `c`

Comparison	**Element value for vector unsigned char**	**Element value for vector unsigned short**	**Element value for vector unsigned int**
Force to FALSE	0	0	0
Equal	0x80	0x8000	0x80000000
Not equal	0x60	0x6000	0x60000000
Greater than	0x20	0x2000	0x20000000
Greater than or equal	0xA0	0xA000	0xA0000000
Less than	0x40	0x4000	0x40000000
Less than and equal	0xC0	0xC000	0xC0000000
Ignore - always TRUE	0xE0	0xE000	0xE0000000

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 104. Return and parameter types for vec_cmpnrg

d	a	b	c
vector bool char	vector unsigned char	vector unsigned char	vector unsigned char
vector bool short	vector unsigned short	vector unsigned short	vector unsigned short
vector bool int	vector unsigned int	vector unsigned int	vector unsigned int

Result value

For each element of `d`, if the value of the corresponding element of `a` is not within any of the specified ranges, the value of each bit is 1. Otherwise, the value of each bit is 0.

Examples

Example 1: Comparing against two ranges

In this example, each element of `a` is checked against the range 10 - 20 exclusive and the range 30 - 40 exclusive.

```
vector unsigned int a = {11, 22, 33, 44};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x20000000,
                         0x40000000};
vector unsigned int d = vec_cmpnrg(a, b, c);
// d = {0, 0xFFFFFFFF, 0, 0xFFFFFFFF}
```

Example 2: Comparing against a single range and a specific value

In this example, each element of `a` is checked against the range 10 - 20 exclusive and the value of 30.

```
vector unsigned int a = {11, 22, 33, 30};
vector unsigned int b = {10, 20, 30, 30};
vector unsigned int c = {0x20000000, 0x40000000, 0x80000000,
                         0x80000000};
vector unsigned int d = vec_cmpnrg(a, b, c);
// d = {0, 0xFFFFFFFF, 0xFFFFFFFF, 0)
```

Example 3: Comparing against a single range

In this example, each element of `a` is checked against the range 10 -20 exclusive.

```
vector unsigned int a = {11, 22, 33, 44};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x00000000,
                         0x00000000};
vector unsigned int d = vec_cmpnrg(a, b, c);
// d = {0, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF}
```

vec_cmpnrg_cc: Vector Compare Not in Ranges with Condition Code

Purpose

Checks whether the value of each element of a vector is not within any of the specified ranges. It also returns a condition code that indicates whether such an element is found.

Syntax

```
d = vec_cmpnrg_cc(a, b, c, e)
```

Usage

This function checks whether the value of each element of `a` is not within any of the ranges that are specified by `b` and `c` as follows:

- Each even-odd element pairs of `b` define values for the limits of the ranges.
- The corresponding even-odd pairs of elements of `c` control which comparison is to be done. The following table lists valid values for elements of `c` of each possible data type for different comparison operations.

Table 105. Element values of parameter c

Comparison	Element value for vector unsigned char	Element value for vector unsigned short	Element value for vector unsigned int
Force to FALSE	0	0	0
Equal	0x80	0x8000	0x80000000
Not equal	0x60	0x6000	0x60000000
Greater than	0x20	0x2000	0x20000000
Greater than or equal	0xA0	0xA000	0xA0000000
Less than	0x40	0x4000	0x40000000
Less than and equal	0xC0	0xC000	0xC0000000
Ignore - always TRUE	0xE0	0xE000	0xE0000000

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 106. Return and parameter types for vec_cmpnrg_cc

d	a	b	c	e
vector bool char	vector unsigned char	vector unsigned char	vector unsigned char	int *
vector bool short	vector unsigned short	vector unsigned short	vector unsigned short	
vector bool int	vector unsigned int	vector unsigned int	vector unsigned int	

Result value

For each element of `d`, if the value of the corresponding element of `a` is not within any of the specified ranges, the value of each bit is 1. Otherwise, the value of each bit is 0.

If the value of at least one element of `a` is not within any of the specified ranges, the value that is referred to by `e` is set to 1. Otherwise, the value that is referred to by `e` is set to 3.

vec_cmpnrg_idx: Vector Compare Not in Ranges Index

Purpose

Returns the byte index of the lowest-position element of a vector whose value is not within any of the specified ranges.

Syntax

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

Usage

This function checks whether the value of each element of `a` is not within any of the ranges that are specified by `b` and `c` as follows:

- Each even-odd element pairs of `b` define values for the limits of the ranges.
- The corresponding even-odd pairs of elements of `c` control which comparison is to be done. The following table lists valid values for elements of `c` of each possible data type for different comparison operations.

Table 107. Element values of parameter c

Comparison	**Element value for vector unsigned char**	**Element value for vector unsigned short**	**Element value for vector unsigned int**
Force to FALSE	0	0	0
Equal	0x80	0x8000	0x80000000
Not equal	0x60	0x6000	0x60000000
Greater than	0x20	0x2000	0x20000000
Greater than or equal	0xA0	0xA000	0xA0000000
Less than	0x40	0x4000	0x40000000
Less than and equal	0xC0	0xC000	0xC0000000
Ignore - always TRUE	0xE0	0xE000	0xE0000000

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 108. Return and parameter types for vec_cmpnrg_idx

d	**a**	**b**	**c**
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int

Result value

The result is the byte index of the lowest-position element of `a` whose value is not within any of the ranges that are specified by `b` and `c`. If the values of all elements of `a` are within the specified ranges, the result is 16.

The result is placed in byte element seven of d, and all other bytes of d are set to 0.

Example

In this example, each element of a is checked against the range 10 - 20 exclusive and the range 30 - 40 exclusive. a[1] and a[3] meet the requirement because they are not within any of the two ranges, and a[1] is the lowest-position eligible element. As a result, 4, which stands for the byte index of a[1], is placed in byte element seven of d.

```
vector unsigned int a = {11, 22, 33, 44};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x20000000,
                         0x40000000};
vector unsigned int d = vec_cmpnrg_idx(a, b, c);
// d = {0, 4, 0, 0)
```

vec_cmpnrg_idx_cc: Vector Compare Not in Ranges Index with Condition Code

Purpose

Returns the byte index of the lowest-position element of a vector whose value is not within any of the specified ranges. It also returns a condition code that indicates whether such an element is found.

Syntax

```
d = vec_cmpnrg_idx_cc(a, b, c, e)
```

Usage

This function checks whether each element of a is not within any of the ranges that are specified by b and c as follows:

- Each even-odd element pairs of b define values for the limits of the ranges.
- The corresponding even-odd pairs of elements of c control which comparison is to be done. The following table lists valid values for elements of c of each possible data type for different comparison operations.

Table 109. Element values of parameter c

Comparison	Element value for vector unsigned char	Element value for vector unsigned short	Element value for vector unsigned int
Force to FALSE	0	0	0
Equal	0x80	0x8000	0x80000000
Not equal	0x60	0x6000	0x60000000
Greater than	0x20	0x2000	0x20000000
Greater than or equal	0xA0	0xA000	0xA0000000
Less than	0x40	0x4000	0x40000000
Less than and equal	0xC0	0xC000	0xC0000000
Ignore - always TRUE	0xE0	0xE000	0xE0000000

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 110. Return and parameter types for vec_cmpnrg_idx_cc

d	a	b	c	e
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	int *
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	

Result value

The result is the byte index of the lowest-position element of `a` whose value is not within any of the ranges that are specified by `b` and `c`. If all elements of `a` are within the specified ranges, the result is 16. The result is placed in byte element seven of `d`, and all other bytes of `d` are set to 0.

If the value of at least one element of `a` is not within any of the specified ranges, the value that is referred to by `e` is set to 1. Otherwise, the value that is referred to by `e` is set to 3.

vec_cmpnrg_or_0_idx: Vector Compare Not in Ranges or Zero Index

Purpose

Returns the byte index of the lowest-position element of a vector whose value is 0 or is not within any of the specified ranges.

Syntax

```
d = vec_cmpnrg_or_0_idx(a, b, c)
```

Usage

This function checks the value of each element of `a` against 0 and the ranges that are specified by `b` and `c` as follows:

- Each even-odd element pairs of `b` define values for the limits of the ranges.
- The corresponding even-odd pairs of elements of `c` control which comparison is to be done. The following table lists valid values for elements of `c` of each possible data type for different comparison operations.

Table 111. Element values of parameter `c`

Comparison	Element value for vector unsigned char	Element value for vector unsigned short	Element value for vector unsigned int
Force to FALSE	0	0	0
Equal	0x80	0x8000	0x80000000
Not equal	0x60	0x6000	0x60000000
Greater than	0x20	0x2000	0x20000000

Table 111. Element values of parameter `c` (continued)

Comparison	Element value for vector unsigned char	Element value for vector unsigned short	Element value for vector unsigned int
Greater than or equal	0xA0	0xA000	0xA0000000
Less than	0x40	0x4000	0x40000000
Less than and equal	0xC0	0xC000	0xC0000000
Ignore - always TRUE	0xE0	0xE000	0xE0000000

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 112. Return and parameter types for vec_cmpnrg_or_0_idx

d	a	b	c
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int

Result value

The result is the byte index of the lowest-position element of `a` whose value is 0 or is not within any of the ranges that are specified by `b` and `c`. If the values of all elements of `a` are within the specified ranges and are not 0, the result is 16.

The result is placed in byte element seven of `d`, and all other bytes of `d` are set to 0.

Example

In this example, each element of `a` is checked not only against 0 but also against the range 10 - 20 exclusive and the range 30 - 40 exclusive. `a[2]` and `a[3]` meet the requirement, and `a[2]` is the lowest-position eligible element. As a result, 8, which stands for the byte index of `a[2]`, is placed in byte element seven of `d`.

```
vector unsigned int a = {11, 33, 0, 22};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x20000000,
                         0x40000000};
vector unsigned int d = vec_cmpnrg_or_0_idx(a, b, c);
// d = {0, 8, 0, 0)
```

vec_cmpnrg_or_0_idx_cc: Vector Compare Not in Ranges or Zero Index with Condition Code

Purpose

Returns the byte index of the lowest-position element of a vector whose value is 0 or is not within any of the specified ranges. It also returns a condition code that indicates whether such an element is found.

Syntax

```
d = vec_cmpnrg_or_0_idx_cc(a, b, c, e)
```

Usage

This function checks the value of each element of `a` against 0 and the ranges that are specified by `b` and `c` as follows:

- Each even-odd element pairs of `b` define values for the limits of the ranges.
- The corresponding even-odd pairs of elements of `c` control which comparison is to be done. The following table lists valid values for elements of `c` of each possible data type for different comparison operations.

Table 113. Element values of parameter `c`

Comparison	Element value for vector unsigned char	Element value for vector unsigned short	Element value for vector unsigned int
Force to FALSE	0	0	0
Equal	0x80	0x8000	0x80000000
Not equal	0x60	0x6000	0x60000000
Greater than	0x20	0x2000	0x20000000
Greater than or equal	0xA0	0xA000	0xA0000000
Less than	0x40	0x4000	0x40000000
Less than and equal	0xC0	0xC000	0xC0000000
Ignore - always TRUE	0xE0	0xE000	0xE0000000

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 114. Return and parameter types for vec_cmpnrg_or_0_idx_cc

d	a	b	c	e
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	int *
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	

Result value

The result is the byte index of the lowest-position element of `a` whose value is 0 or is not within any of the ranges that are specified by `b` and `c`. If the values of all elements of `a` are within the specified ranges and are not 0, the result is 16. The result is placed in byte element seven of `d`, and all other bytes of `d` are set to 0.

The value that is referred to by `e` is determined as follows. The comparison is performed starting from `a[0]`.

- 0 if value 0 is found in `a` before an element whose value is not within any of the specified ranges is found, or if they happen concurrently.
- 1 if no elements of `a` have a value of 0 and if the value of at least one element of `a` is not within any of the specified ranges.
- 2 if value 0 is found in `a` after an element whose value is not within any of the specified ranges is found.
- 3 if no elements of `a` have a value of 0 and if each element of `a` has a value that is within any of the specified ranges.

vec_cmprg: Vector Compare Ranges

Purpose

Checks whether each element of a vector is within any of the specified ranges.

Syntax

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

Usage

You can use this function to check whether each element of `a` is within any of the ranges that are specified by `b` and `c` as follows:

- Each even-odd element pairs of `b` define values for the limits of the ranges.
- The corresponding even-odd pairs of elements of `c` control which comparison is to be done. The following table lists valid values for elements of `c` of each possible data type for different comparison operations.

Table 115. Element values of parameter c

Comparison	Element value for vector unsigned char	Element value for vector unsigned short	Element value for vector unsigned int
Force to FALSE	0	0	0
Equal	0x80	0x8000	0x80000000
Not equal	0x60	0x6000	0x60000000
Greater than	0x20	0x2000	0x20000000
Greater than or equal	0xA0	0xA000	0xA0000000
Less than	0x40	0x4000	0x40000000
Less than and equal	0xC0	0xC000	0xC0000000
Ignore - always TRUE	0xE0	0xE000	0xE0000000

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 116. Return and parameter types for vec_cmprg

d	a	b	c
vector bool char	vector unsigned char	vector unsigned char	vector unsigned char
vector bool short	vector unsigned short	vector unsigned short	vector unsigned short
vector bool int	vector unsigned int	vector unsigned int	vector unsigned int

Result value

For each element of d, if the corresponding element of a is within any of the specified ranges, the value of each bit is 1. Otherwise, the value of each bit is 0.

Examples

Example 1: Comparing against two ranges

In this example, each element of a is checked against the range 10 - 20 exclusive and the range 30 - 40 exclusive.

```
vector unsigned int a = {11, 22, 33, 44};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x20000000,
                         0x40000000};
vector unsigned int d = vec_cmprg(a, b, c);
// d = {0xFFFFFFFF, 0, 0xFFFFFFFF, 0}
```

Example 2: Comparing against a single range and a specific value

In this example, each element of a is checked against the range 10 - 20 exclusive and the value of 30.

```
vector unsigned int a = {11, 22, 33, 30};
vector unsigned int b = {10, 20, 30, 30};
vector unsigned int c = {0x20000000, 0x40000000, 0x80000000,
                         0x80000000};
vector unsigned int d = vec_cmprg(a, b, c);
// d = {0xFFFFFFFF, 0, 0, 0xFFFFFFFF)
```

Example 3: Comparing against a single range

In this example, each element of a is checked against the range 10 -20 exclusive.

```
vector unsigned int a = {11, 22, 33, 44};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x00000000,
                         0x00000000};
vector unsigned int d = vec_cmprg(a, b, c);
// d = {0xFFFFFFFF, 0, 0, 0}
```

vec_cmprg_cc: Vector Compare Ranges with Condition Code Purpose

Checks whether each element of a vector is within any of the specified ranges. It also returns a condition code that indicates whether such an element is found.

Syntax

```
d = vec_cmprg_cc(a, b, c, e)
```

Usage

You can use this function to check whether each element of a is within any of the ranges that are specified by b and c as follows:

- Each even-odd element pairs of b define values for the limits of the ranges.
- The corresponding even-odd pairs of elements of c control which comparison is to be done. The following table lists valid values for elements of c of each possible data type for different comparison operations.

Table 117. Element values of parameter c

Comparison	Element value for vector unsigned char	Element value for vector unsigned short	Element value for vector unsigned int
Force to FALSE	0	0	0
Equal	0x80	0x8000	0x80000000
Not equal	0x60	0x6000	0x60000000
Greater than	0x20	0x2000	0x20000000
Greater than or equal	0xA0	0xA000	0xA0000000
Less than	0x40	0x4000	0x40000000
Less than and equal	0xC0	0xC000	0xC0000000
Ignore - always TRUE	0xE0	0xE000	0xE0000000

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 118. Return and parameter types for vec_cmprg_cc

d	a	b	c	e
vector bool char	vector unsigned char	vector unsigned char	vector unsigned char	int *
vector bool short	vector unsigned short	vector unsigned short	vector unsigned short	
vector bool int	vector unsigned int	vector unsigned int	vector unsigned int	

Result value

For each element of `d`, if the corresponding element of `a` is within any of the specified ranges, the value of each bit is 1. Otherwise, the value of each bit is 0.

If the value of at least one element of `a` is within any of the specified ranges, the value that is referred to by `e` is set to 1. Otherwise, the value that is referred to by `e` is set to 3.

vec_cmprg_idx: Vector Compare Ranges Index

Purpose

Returns the byte index of the lowest-position element of a vector whose value is within any of the specified ranges.

Syntax

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

Usage

This function checks whether each element of `a` is within any of the ranges that are specified by `b` and `c` as follows:

- Each even-odd element pairs of `b` define values for the limits of the ranges.
- The corresponding even-odd pairs of elements of `c` control which comparison is to be done. The following table lists valid values for elements of `c` of each possible data type for different comparison operations.

Table 119. Element values of parameter c

Comparison	Element value for vector unsigned char	Element value for vector unsigned short	Element value for vector unsigned int
Force to FALSE	0	0	0
Equal	0x80	0x8000	0x80000000
Not equal	0x60	0x6000	0x60000000
Greater than	0x20	0x2000	0x20000000
Greater than or equal	0xA0	0xA000	0xA0000000
Less than	0x40	0x4000	0x40000000
Less than and equal	0xC0	0xC000	0xC0000000
Ignore - always TRUE	0xE0	0xE000	0xE0000000

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 120. Return and parameter types for vec_cmprg_idx

d	a	b	c
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int

Result value

The result is the byte index of the lowest-position element of `a` whose value is within any of the ranges that are specified by `b` and `c`. If none of the elements of `a` is within any of the specified ranges, the result is 16.

The result is placed in byte element seven of `d`, and all other bytes are set to 0.

Example

In this example, each element of a is checked against the range 10 - 20 exclusive and the range 30 - 40 exclusive. `a[1]` and `a[3]` meet the requirement, and `a[1]` is the lowest-position eligible element. As a result, 4, which stands for the byte index of `a[1]`, is placed in byte element seven of `d`.

```
vector unsigned int a = {1, 11, 22, 33};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x20000000,
                         0x40000000};
vector unsigned int d = vec_cmprg_idx(a, b, c);
// d = {0, 4, 0, 0)
```

vec_cmprg_idx_cc: Vector Compare Ranges Index with Condition Code

Purpose

Returns the byte index of the lowest-position element of a vector whose value is within any of the specified ranges. It also returns a condition code that indicates whether such an element is found.

Syntax

```
d = vec_cmprg_idx_cc(a, b, c, e)
```

Usage

This function checks whether each element of `a` is within any of the ranges that are specified by `b` and `c` as follows:

- Each even-odd element pairs of `b` define values for the limits of the ranges.
- The corresponding even-odd pairs of elements of `c` control which comparison is to be done. The following table lists valid values for elements of `c` of each possible data type for different comparison operations.

Table 121. Element values of parameter c

Comparison	Element value for vector unsigned char	Element value for vector unsigned short	Element value for vector unsigned int
Force to FALSE	0	0	0
Equal	0x80	0x8000	0x80000000
Not equal	0x60	0x6000	0x60000000
Greater than	0x20	0x2000	0x20000000
Greater than or equal	0xA0	0xA000	0xA0000000
Less than	0x40	0x4000	0x40000000
Less than and equal	0xC0	0xC000	0xC0000000
Ignore - always TRUE	0xE0	0xE000	0xE0000000

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 122. Return and parameter types for vec_cmprg_idx_cc

d	a	b	c	e
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	int *
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	

Result value

The result is the byte index of the lowest-position element of `a` whose value is within any of the ranges that are specified by `b` and `c`. If none of the elements of `a` is within any of the specified ranges, the result is 16. The result is placed in byte element seven of `d`, and all other bytes of `d` are set to 0.

If the value of at least one element of `a` is within any of the specified ranges, the value that is referred to by `e` is set to 1. Otherwise, the value that is referred to by `e` is set to 3.

vec_cmprg_or_0_idx: Vector Compare Ranges or Zero Index

Purpose

Returns the byte index of the lowest-position element of a vector whose value is 0 or is within any of the specified ranges.

Syntax

```
d = vec_cmprg_or_0_idx(a, b, c)
```

Usage

This function checks the value of each element of `a` against value 0 and the ranges that are specified by `b` and `c` as follows:

- Each even-odd element pairs of `b` define values for the limits of the ranges.
- The corresponding even-odd pairs of elements of `c` control which comparison is to be done. The following table lists valid values for elements of `c` of each possible data type for different comparison operations.

Table 123. Element values of parameter c

Comparison	**Element value for vector unsigned char**	**Element value for vector unsigned short**	**Element value for vector unsigned int**
Force to FALSE	0	0	0
Equal	0x80	0x8000	0x80000000
Not equal	0x60	0x6000	0x60000000
Greater than	0x20	0x2000	0x20000000
Greater than or equal	0xA0	0xA000	0xA0000000
Less than	0x40	0x4000	0x40000000
Less than and equal	0xC0	0xC000	0xC0000000
Ignore - always TRUE	0xE0	0xE000	0xE0000000

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 124. Return and parameter types for vec_cmprg_or_0_idx

d	**a**	**b**	**c**
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char

Table 124. Return and parameter types for vec_cmprg_or_0_idx (continued)

d	a	b	c
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int

Result value

The result is the byte index of the lowest-position element of a whose value is 0 or is within any of the ranges that are specified by b and c. If none of the elements of a is within any of the specified ranges and has a value of 0, the result is 16.

The result is placed in byte element seven of d, and all other bytes of d are set to 0.

Example

In this example, each element of a is checked not only against 0 but also against the range 10 - 20 exclusive and the range 30 - 40 exclusive. a[1] and a[3] meet the requirement, and a[1] is the lowest-position element. As a result, 4, which stands for the byte index of a[1], is placed in byte element seven of d.

```
vector unsigned int a = {1, 0, 22, 33};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x20000000,
                         0x40000000};
vector unsigned int d = vec_cmprg_or_0_idx(a, b, c);
// d = {0, 4, 0, 0)
```

vec_cmprg_or_0_idx_cc: Vector Compare Ranges or Zero Index with Condition Code

Purpose

Returns the byte index of the lowest-position element of a vector whose value is 0 or is within any of the specified ranges. It also returns a condition code that indicates whether such an element is found.

Syntax

```
d = vec_cmprg_or_0_idx_cc(a, b, c, e)
```

Usage

This function checks the value of each element of a against value 0 and the ranges that are specified by b and c as follows:

- Each even-odd element pairs of b define values for the limits of the ranges.
- The corresponding even-odd pairs of elements of c control which comparison is to be done. The following table lists valid values for elements of c of each possible data type for different comparison operations.

Table 125. Element values of parameter c

Comparison	Element value for vector unsigned char	Element value for vector unsigned short	Element value for vector unsigned int
Force to FALSE	0	0	0
Equal	0x80	0x8000	0x80000000

Table 125. Element values of parameter c (continued)

Comparison	Element value for vector unsigned char	Element value for vector unsigned short	Element value for vector unsigned int
Not equal	0x60	0x6000	0x60000000
Greater than	0x20	0x2000	0x20000000
Greater than or equal	0xA0	0xA000	0xA0000000
Less than	0x40	0x4000	0x40000000
Less than and equal	0xC0	0xC000	0xC0000000
Ignore - always TRUE	0xE0	0xE000	0xE0000000

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 126. Return and parameter types for vec_cmprg_or_0_idx_cc

d	a	b	c	e
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	int *
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	

Result value

The result is the byte index of the lowest-position element of a whose value is 0 or is within any of the ranges that are specified by b and c. If none of the elements of a is within any of the specified ranges and has a value of 0, the result is 16. The result is placed in byte element seven of d, and all other bytes of d are set to 0.

The value that is referred to by e is determined as follows. The comparison is performed starting from a[0].

- 0 if value 0 is found in a before an element whose value is within any of the specified ranges is found, or if they happen concurrently.
- 1 if no elements of a have a value of 0 and if the value of at least one element of a is within any of the specified ranges.
- 2 if value 0 is found in a after an element whose value is within any of the specified ranges is found.
- 3 if no elements of a have a value of 0 and are within any of the specified ranges.

Element searching functions

This topic collection describes vector built-in functions for element searching.

vec_find_any_eq: Vector Find Any Element Equal

Purpose

Compares each element of one vector with every element of another vector for equality.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 127. Return and parameter types for vec_find_any_eq

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
	vector bool char	vector bool char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
	vector bool short	vector bool short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector bool int	vector bool int

Result value

For each element of `d`, if the corresponding element of `a` equals any element of `b`, the value of each bit is 1. Otherwise, the value of each bit is 0.

Example

In the following example, the value of `a[2]` is found in `b`, so each bit of `d[2]` is set to 1:

```
vector signed int a = {1, -2, 3, -4};
vector signed int b = {-5, 3, -7, 8};
vector bool int d = vec_find_any_eq(a, b);
// d = {0, 0, 0xFFFFFFFF, 0}
```

vec_find_any_eq_cc: Vector Find Any Element Equal with Condition Code

Purpose

Compares each element of one vector with every element of another vector for equality. It also returns a condition code that indicates whether such an element is found.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 128. Return and parameter types for vec_find_any_eq_cc

d	a	b	c
vector bool char	vector signed char	vector signed char	int *
	vector unsigned char	vector unsigned char	
	vector bool char	vector bool char	
vector bool short	vector signed short	vector signed short	
	vector unsigned short	vector unsigned short	
	vector bool short	vector bool short	
vector bool int	vector signed int	vector signed int	
	vector unsigned int	vector unsigned int	
	vector bool int	vector bool int	

Result value

For each element of `d`, if the corresponding element of `a` equals any element of `b`, the value of each bit is 1; otherwise, the value of each bit is 0. If at least one element of `a` finds an element of the same value in `b`, the value that is referred to by `c` is set to 1; otherwise, the value that is referred to by `c` is set to 3.

vec_find_any_eq_idx: Vector Find Any Element Equal Index

Purpose

Compares each element of one vector with every element of another vector for equality and returns the byte index of the eligible element of the lowest position.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 129. Return and parameter types for vec_find_any_eq_idx

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
	vector bool int	vector bool int

Result value

The result is the byte index of the lowest-position element of a that equals any element of b. If none of the elements of a finds an element of the same value in b, the result is 16.

The result is placed in byte element seven of d, and all other bytes of d are set to 0.

Examples

Example 1

In this example, a[2] equals b[1], and the byte index of a[2] is 8. As a result, byte element seven of d is set to 8.

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {5, 3, 7, 8};
vector unsigned int d = vec_find_any_eq_idx(a, b);
// d = {0, 8, 0, 0}
```

Example 2

In this example, no elements of a are found in b, so 16 is returned and placed in byte element seven of d.

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {5, 6, 7, 8};
vector unsigned int d = vec_find_any_eq_idx(a, b);
// d = {0, 16, 0, 0}
```

vec_find_any_eq_idx_cc: Vector Find Any Element Equal Index with Condition Code

Purpose

Compares each element of one vector with every element of another vector for equality and returns the byte index of the eligible element of the lowest position. It also returns a condition code that indicates whether such an element is found.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 130. Return and parameter types for vec_find_any_eq_idx_cc

d	a	b	c
vector signed char	vector signed char	vector signed char	int *
vector unsigned char	vector unsigned char	vector unsigned char	
	vector bool char	vector bool char	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
	vector bool short	vector bool short	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
	vector bool int	vector bool int	

Result value

The result is the byte index of the lowest-position element of `a` that equals any element of `b`. If none of the elements of `a` finds an element of the same value in `b`, the result is 16. The result is placed in byte element seven of `d`, and all other bytes of `d` are set to 0.

If at least one element of `a` finds an element of the same value in `b`, the value that is referred to by `c` is set to 1; otherwise, the value that is referred to by `c` is set to 3.

Examples

Example 1

In this example, `a[2]` equals `b[1]`, so `c` is set to 1. Because the byte index of `a[2]` is 8, byte element seven of `d` is set to 8.

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {5, 3, 7, 8};
int c = 0;
vector unsigned int d = vec_find_any_eq_idx_cc(a, b, &c);
// d = {0, 8, 0, 0}, c = 1
```

Example 2

In this example, no elements of `a` are found in `b`, so byte element seven of `d` is set to 16 and `c` is set to 3.

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {5, 6, 7, 8};
int c = 0;
vector unsigned int d = vec_find_any_eq_idx_cc(a, b, &c);
// d = {0, 16, 0, 0}, c = 3
```

vec_find_any_eq_or_0_idx: Vector Find Any Element Equal or Zero Index

Purpose

Returns the byte index of the lowest-position element of one vector that finds an equal value in another vector or that has a value of 0.

Syntax

```
d = vec_find_any_eq_or_0_idx(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 131. Return and parameter types for vec_find_any_eq_or_0_idx

d	**a**	**b**
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
	vector bool int	vector bool int

Result value

The result is the byte index of the lowest-position element of `a` that equals any element of `b` or that has a value of 0. If all the elements of `a` satisfies neither of these two conditions, the result is 16.

The result is placed in byte element seven of `d`, and all other bytes of `d` are set to 0.

Example

In this example, the value of `a[3]` is found in `b`, and `a[2]` has a value of 0. Because `a[2]`, whose byte index is 8, has lower byte index than `a[3]`, byte element seven of `d` is set to 8.

```
vector unsigned int a = {1, 2, 0, 5};
vector unsigned int b = {5, 6, 7, 8};
vector unsigned int d = vec_find_any_eq_or_0_idx(a, b);
// d = {0, 8, 0, 0}
```

vec_find_any_eq_or_0_idx_cc: Vector Find Any Element Equal or Zero Index with Condition Code

Purpose

Returns the byte index of the lowest-position element of one vector that finds an equal value in another vector or that has a value of 0. It also returns a condition code that indicates whether such an element is found.

Syntax

```
d = vec_find_any_eq_or_0_idx_cc(a, b, c)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 132. Return and parameter types for vec_find_any_eq_or_0_idx_cc

d	a	b	c
vector signed char	vector signed char	vector signed char	int *
vector unsigned char	vector unsigned char	vector unsigned char	
	vector bool char	vector bool char	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
	vector bool short	vector bool short	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
	vector bool int	vector bool int	

Result value

The result is the byte index of the lowest-position element of `a` that equals any element of `b` or that has a value of 0. If all the elements of `a` satisfies neither of these two conditions, the result is 16. The result is placed in byte element seven of `d`, and all other bytes of `d` are set to 0.

The value that is referred to by `c` is determined as follows. The comparison is performed starting from `a[0]`.

- 0 if value 0 is found in `a` before an element that equals any element of `b` is found, or if they happen concurrently.
- 1 if no elements of `a` have a value of 0 and if at least one element of `a` equals any element of `b`.
- 2 if value 0 is found in `a` after an element that equals any element of `b` is found.
- 3 if no elements of `a` have a value of 0 and equal any element of `b`.

vec_find_any_ne: Vector Find Any Element Not Equal

Purpose

Compares each element of one vector with every element of another vector for inequality.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 133. Return and parameter types for vec_find_any_ne

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
	vector bool char	vector bool char

Table 133. Return and parameter types for vec_find_any_ne (continued)

d	a	b
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
	vector bool short	vector bool short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector bool int	vector bool int

Result value

For each element of d, if the corresponding element of a does not equal any element of b, the value of each bit is 1. Otherwise, the value of each bit is 0.

Example

In the following example, the values of a[0], a[1], and a[3] are not found in b, so each bit of the corresponding element of d is set to 1:

```
vector signed int a = {1, -2, 3, -4};
vector signed int b = {-5, 3, -7, 8};
vector bool int d = vec_find_any_ne(a, b);
// d = {0xFFFFFFFF, 0xFFFFFFFF, 0, 0xFFFFFFFF}
```

vec_find_any_ne_cc: Vector Find Any Element Not Equal with Condition Code

Purpose

Compares each element of one vector with every element of another vector for inequality. It also returns a condition code that indicates whether such an element is found.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 134. Return and parameter types for vec_find_any_ne_cc

d	a	b	c
vector bool char	vector signed char	vector signed char	int *
	vector unsigned char	vector unsigned char	
	vector bool char	vector bool char	
vector bool short	vector signed short	vector signed short	
	vector unsigned short	vector unsigned short	
	vector bool short	vector bool short	
vector bool int	vector signed int	vector signed int	
	vector unsigned int	vector unsigned int	
	vector bool int	vector bool int	

Result value

For each element of d, if the corresponding element of a does not equal any element of b, the value of each bit is 1. Otherwise, the value of each bit is 0. If at least one element of a does not find an element of the same value in b, the value that is referred to by c is set to 1. Otherwise, the value that is referred to by c is set to 3.

vec_find_any_ne_idx: Vector Find Any Element Not Equal Index

Purpose

Compares each element of one vector with every element of another vector for inequality and returns the byte index of the eligible element of the lowest position.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 135. Return and parameter types for vec_find_any_ne_idx

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
	vector bool int	vector bool int

Result value

The result is the byte index of the lowest-position element of a that does not equal any element of b. If each element of a has the same value as any element of b, the result is 16.

The result is placed in byte element seven of d, and all other bytes of d are set to 0.

Examples

Example 1

In this example, the value of a[1] is not found in b, and the byte index of a[1] is 4. As a result, byte element seven of d is set to 4.

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {1, 5, 3, 4};
vector unsigned int d = vec_find_any_ne_idx(a, b);
// d = {0, 4, 0, 0}
```

Example 2

In this example, all the values of elements of a are found in b, so 16 is returned and placed in byte element seven of d.

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {1, 2, 3, 4};
vector unsigned int d = vec_find_any_ne_idx(a, b);
// d = {0, 16, 0, 0}
```

vec_find_any_ne_idx_cc: Vector Find Any Element Not Equal Index with Condition Code

Purpose

Compares each element of one vector with every element of another vector for inequality and returns the byte index of the eligible element of the lowest position. It also returns a condition code that indicates whether such an element is found.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 136. Return and parameter types for vec_find_any_ne_idx_cc

d	**a**	**b**	**c**
vector signed char	vector signed char	vector signed char	int *
vector unsigned char	vector unsigned char	vector unsigned char	
	vector bool char	vector bool char	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
	vector bool short	vector bool short	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
	vector bool int	vector bool int	

Result value

The result is the byte index of the lowest-position element of a that does not equal any element of b. If each element of a has the same value as any element of b, the result is 16. The result is placed in byte element seven of d, and all other bytes of d are set to 0.

If at least one element of a does not find an element of the same value in b, the value that is referred to by c is set to 1; otherwise, the value that is referred to by c is set to 3.

Examples

Example 1

In this example, the value of a[1] is not found in b, so c is set to 1. Because the byte index of a[1] is 4, byte element seven of d is set to 4.

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {1, 5, 3, 4};
int c = 0;
vector unsigned int d = vec_find_any_ne_idx_cc(a,b,&c);
// d = {0, 4, 0, 0}, c = 1
```

Example 2

In this example, the value of each element of a is found in b, so byte element seven of d is set to 16 and c is set to 3.

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {1, 2, 3, 4};
int c = 0;
vector unsigned int d = vec_find_any_ne_idx_cc(a,b,&c);
// d = {0, 16, 0, 0}, c = 3
```

vec_find_any_ne_or_0_idx: Vector Find Any Element Not Equal or Zero Index

Purpose

Returns the byte index of the lowest-position element of one vector that cannot find an equal value in another vector or that has a value of 0.

Syntax

```
d = vec_find_any_ne_or_0_idx(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 137. Return and parameter types for vec_find_any_ne_or_0_idx:

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
	vector bool int	vector bool int

Result value

The result is the byte index of the lowest-position element of a that does not equal any element of b or that has a value of 0. If all the elements of a satisfies neither of these two conditions, the result is 16. The result is placed in byte element seven of d, and all other bytes of d are set to 0.

Example

In this example, a[2] has a value of 0, and the value of a[3] is not found in b. Because a[2], whose byte index is 8, has lower byte index than a[3], byte element seven of d is set to 8.

```
vector unsigned int a = {1, 2, 0, 5};
vector unsigned int b = {1, 2, 0, 4};
vector unsigned int d = vec_find_any_ne_or_0_idx(a, b);
// d = {0, 8, 0, 0}
```

vec_find_any_ne_or_0_idx_cc: Vector Find Any Element Not Equal or Zero Index with Condition Code

Purpose

Returns the byte index of the lowest-position element of a that cannot find an equal value in b or that has a value of 0. It also returns a condition code that indicates whether such an element is found.

Syntax

```
d = vec_find_any_ne_or_0_idx_cc(a, b, c)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 138. Return and parameter types for vec_find_any_ne_or_0_idx_cc

d	a	b	c
vector signed char	vector signed char	vector signed char	int *
vector unsigned char	vector unsigned char	vector unsigned char	
	vector bool char	vector bool char	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
	vector bool short	vector bool short	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
	vector bool int	vector bool int	

Result value

The result is the byte index of the lowest-position element of a that does not equal any element of b or that has a value of 0. If all the elements of a satisfies neither of these two conditions, the result is 16. The result is placed in byte element seven of d, and all other bytes of d are set to 0.

The value that is referred to by c is determined as follows:

- 0 if value 0 is found in a before an element that does not equal any element of b is found, or if they happen concurrently.

- 1 if no elements of a have a value of 0 and if at least one element of a does not equal any element of b.
- 2 if value 0 is found in a after an element that does not equal any element of b is found.
- 3 if no elements of a have a value of 0 and each element of a equals a certain element of b.

Gather and scatter functions

This topic collection describes vector built-in functions that are used to gather and scatter elements.

vec_extract: Vector Extract

Purpose

Returns the value of the designated element from a vector.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 139. Return and parameter types for vec_extract

d	a	b
signed char	vector signed char	signed int
unsigned char	vector unsigned char	
	vector bool char	
signed short	vector signed short	
unsigned short	vector unsigned short	
	vector bool short	
signed int	vector signed int	
unsigned int	vector unsigned int	
	vector bool int	
signed long long	vector signed long long	
unsigned long long	vector unsigned long long	
	vector bool long long	
double	vector double	

Result value

d is assigned the value of a[b mod *n*], where *n* is the number of elements in a.

vec_gather_element: Vector Gather Element

Purpose

Returns a copy of a vector with the value of one element replaced by the value obtained by applying the specified offset to a pointer.

Syntax

```
d = vec_gather_element(a, b, c, e)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 140. Return and parameter types for vec_gather_element

d	a	b	c	e
vector signed int	vector signed int	vector unsigned int	const signed int *	unsigned char[1]
vector unsigned int	vector unsigned int		const unsigned int *	
vector bool int	vector bool int			
vector signed long long	vector signed long long	vector unsigned long long	const signed long long *	unsigned char[2]
vector unsigned long long	vector unsigned long long		const unsigned long long *	
vector bool long long	vector bool long long			
vector double	vector double	vector unsigned long long	const double *	

Notes:

1. It must be a literal whose value is in the range 0 - 3 inclusive.
2. It must be a literal whose value is 0 or 1.

Result value

d[e] is assigned the value that is obtained by applying the offset specified by b[e] to pointer c, and other elements of d are assigned the values of the corresponding elements of a.

Example

In the following example, v1[0] is replaced by a1[5] using the offset specified by v3[0], and v2[0] is replaced by a2[5] using the offset specified by v3[0]:

```
unsigned int a1[10] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
unsigned int a2[10] = {20, 21, 22, 23, 24, 25, 26, 27, 28, 29};
vector unsigned int v1, v2 = {1, 2, 3, 4};
vector unsigned int v3 = {5 * sizeof(int), 8 * sizeof(int),
                          9 * sizeof(int), 6 * sizeof(int)};
v1 = vec_gather_element (v1, v3, a1, 0);  // v1 = {15, 2, 3, 4}
v2 = vec_gather_element (v2, v3, a2, 0);  // v2 = {25, 2, 3, 4}
```

The results are the same as the following code example:

```
unsigned int a1[10] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
unsigned int a2[10] = {20, 21, 22, 23, 24, 25, 26, 27, 28, 29};
vector unsigned int v1, v2 = {1, 2, 3, 4};
vector unsigned int v3 = {0, 0, 0, 0};
v1 = vec_gather_element (v1, v3, &a1[5], 0); // v1 = {15, 2, 3, 4};
v2 = vec_gather_element (v2, v3, &a2[5], 0); // v2 = {25, 2, 3, 4};
```

vec_insert: Vector Insert

Purpose

Returns a copy of a vector with the value of the specified element replaced by a given value.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 141. Return and parameter types for vec_insert

d	a	b	c
vector signed char	signed char	vector signed char	signed int
vector unsigned char	unsigned char	vector unsigned char	
		vector bool char	
vector signed short	signed short	vector signed short	
vector unsigned short	unsigned short	vector unsigned short	
		vector bool short	
vector signed int	signed int	vector signed int	
vector unsigned int	unsigned int	vector unsigned int	
		vector bool int	
vector signed long long	signed long long	vector signed long long	
vector unsigned long long	unsigned long long	vector unsigned long long	
		vector bool long long	
vector double	double	vector double	

Result value

`d` is a copy of `b` with the value of `d[c mod` *n*`]` replaced by the value of `a`, where *n* is the number of elements in `b`.

vec_insert_and_zero: Vector Insert and Zero

Purpose

Sets the leftmost doubleword element or the rightmost element of the leftmost doubleword to what is referred to by the specified pointer, and sets the bit positions of all other vector elements to zero.

Syntax

```
d = vec_insert_and_zero(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 142. Return and parameter types for vec_insert_and_zero

d	**a**
vector signed char	const signed char *
vector unsigned char	const unsigned char *
vector signed short	const signed short *
vector unsigned short	const unsigned short *
vector signed int	const signed int *
vector unsigned int	const unsigned int *
vector signed long long	const signed long long *
vector unsigned long long	const unsigned long long *
vector double	const double *

Result value

The leftmost doubleword element or the rightmost element of the leftmost doubleword of `d` is set to the value referred to by `a`. The bit positions of all other elements of `d` are set to 0.

Example

Example 1

```
const signed char temp = 'x';
const signed char * a = &temp;
vector signed char b = vec_insert_and_zero(a);
// b is {'0','0','0','0','0','0','0','x','0','0','0','0','0','0','0','0'}.
// b[7] is the rightmost element of the leftmost doubleword.
```

Example 2

```
const signed short temp = 1;
const signed short * a = &temp;
vector signed short b = vec_insert_and_zero(a);
// b is {0, 0, 0, 1, 0, 0, 0, 0}.
// b[3] is the rightmost element of the leftmost doubleword.
```

Example 3

```
const signed int temp = 1;
const signed int * a = &temp;
vector signed int b = vec_insert_and_zero(a);
// b is {0, 1, 0, 0}.
// b[1] is the rightmost element of the leftmost doubleword.
```

Example 4

```
const double temp = 1;
const double * a = &temp;
vector double b = vec_insert_and_zero(a);
// b is {1, 0}.
// b[0] is the leftmost doubleword element.
```

vec_perm: Vector Permute

Purpose

Returns a vector that contains some elements of two vectors in the order that is specified by a third vector.

Syntax

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

Return and parameter types

Table 143. Return and parameter types for vec_perm

d	a	b	c
vector signed char	vector signed char	vector signed char	vector unsigned char
vector unsigned char	vector unsigned char	vector unsigned char	
vector bool char	vector bool char	vector bool char	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
vector bool short	vector bool short	vector bool short	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
vector bool int	vector bool int	vector bool int	
vector signed long long	vector signed long long	vector signed long long	
vector unsigned long long	vector unsigned long long	vector unsigned long long	
vector bool long long	vector bool long long	vector bool long long	
vector double	vector double	vector signed double	

Result value

`d` contains some elements of `a` and `b` in the order that is specified by `c`.

Note: You can use the vector mask generation built-in functions to generate mask `c`.

Example

In the following figure, the indexes of elements of `a` and `b` are indicated in the corresponding blocks. Each byte of `d` is selected by using the least significant 5 bits of the corresponding byte of `c` as the index.

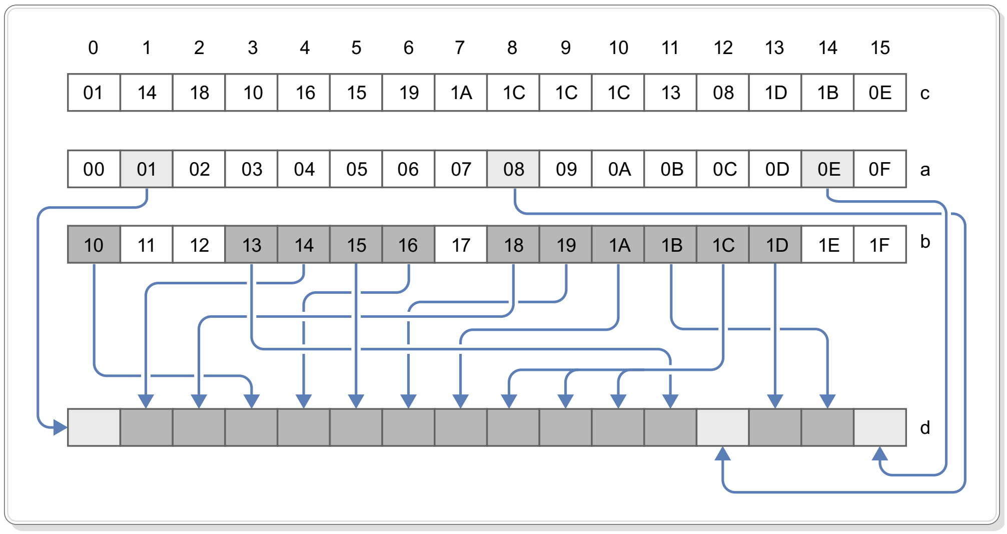

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
01 14 18 10 16 15 19 1A 1C 1C 1C 13 08 1D 1B 0E
c
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
a
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
b
d

Figure 6. Permute 16 8-bit integer elements

Related information:

"Mask generation functions" on page 163

vec_permi: Vector Permute Immediate

Purpose

Returns a vector by permuting and combining two 8-byte elements from two vectors based on the specified value.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 144. Return and parameter types for vec_permi

d	a	b	c
vector signed long long	vector signed long long	vector signed long long	int[1]
vector unsigned long long	vector unsigned long long	vector unsigned long long	
vector bool long long	vector bool long long	vector bool long long	
vector double	vector double	vector double	
Note: 1. It must be a literal whose value is in the range 0 - 3 inclusive.			

Result value

The values of the elements of d are determined by c as follows:

Table 145. Result value of d

c	Value of d[0]	Value of d[1]
0	a[0]	b[0]
1	a[0]	b[1]
2	a[1]	b[0]
3	a[1]	b[1]

vec_promote: Vector Promote

Purpose

Returns a vector with the specified element set to a given value.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 146. Return and parameter types for vec_promote

d	a	b
vector signed char	signed char	signed int
vector unsigned char	unsigned char	
vector signed short	signed short	
vector unsigned short	unsigned short	
vector signed int	signed int	
vector unsigned int	unsigned int	
vector signed long long	signed long long	
vector unsigned long long	unsigned long long	
vector double	double	

Result value

d is set to the value of a[b mod *n*], where *n* is the number of elements in d. All other elements of d are undefined.

vec_scatter_element: Vector Scatter Element

Purpose

Stores the value of a vector element to the location specified by an offset and a pointer.

Syntax

```
d=vec_scatter_element(a, b, c, e)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 147. Return and parameter types for vec_scatter_element

d	a	b	c	e
void	vector signed int	vector unsigned int	signed int *	unsigned long long[1]
	vector unsigned int		unsigned int *	
	vector bool int			
	vector signed long long	vector unsigned long long	signed long long *	unsigned long long[2]
	vector unsigned long long		unsigned long long *	
	vector bool long long			
	vector double	vector unsigned long long	double *	

Notes:

1. It must be a literal whose value is in the range 0 - 3 inclusive.
2. It must be a literal whose value is 0 or 1.

Result value

The value of `a[e]` is stored to the location that is obtained by applying the offset specified by `b[e]` to pointer `c`.

vec_sel: Vector Select

Purpose

Returns a vector that contains the selected values of two vectors according to the value of a third vector.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 148. Return and parameter types for vec_sel

d	a	b	c
vector signed char	vector signed char	vector signed char	vector unsigned char
			vector bool char
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char
			vector bool char
vector bool char	vector bool char	vector bool char	vector unsigned char
			vector bool char

Table 148. Return and parameter types for vec_sel (continued)

d	**a**	**b**	**c**
vector signed short	vector signed short	vector signed short	vector unsigned short
			vector bool short
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short
			vector bool short
vector bool short	vector bool short	vector bool short	vector unsigned short
			vector bool short
vector signed int	vector signed int	vector signed int	vector unsigned int
			vector bool int
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int
			vector bool int
vector bool int	vector bool int	vector bool int	vector unsigned int
			vector bool int
vector signed long long	vector signed long long	vector signed long long	vector unsigned long long
			vector bool long long
vector unsigned long long	vector unsigned long long	vector unsigned long long	vector unsigned long long
			vector bool long long
vector bool long long	vector bool long long	vector bool long long	vector unsigned long long
			vector bool long long
vector double	vector double	vector double	vector unsigned long long
			vector bool long long

Result value

For each bit of c, if the bit is 0, the corresponding bit of d is set to the value of the corresponding bit of a; otherwise, the corresponding bit of d is set to the value of the corresponding bit of b.

Example

In the following example, each bit of c[1] and c[3] has a value of 0, so the corresponding bit of d has the value of the corresponding bit of a. Each bit of c[0] and c[2] does not have a value of 0, so the corresponding bit of d has the value of the corresponding bit of b.

```
vector signed int a = {1, 2, 3, 4};
vector signed int b = {5, 6, 7, 8};
vector unsigned int e = {9, 10, 11, 12};
vector unsigned int f = {9, 9, 11, 11};
vector bool int c = vec_cmpeq(e, f);
// c = {0xFFFFFFFF, 0, 0xFFFFFFFF, 0};
vector signed int d = vec_sel (a, b, c);
// d = {5, 2, 7, 4}
```

Mask generation functions

This topic collection describes vector built-in functions for mask generation.

vec_genmask: Vector Generate Byte Mask

Purpose

Generates byte masks for elements.

Syntax

```
d = vec_genmask(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 149. Return and parameter types for vec_genmask

d	**a**
vector unsigned char	unsigned short[1]
Note: 1. It must be a literal.	

Result value

For each bit in `a`, if the bit is 1, all bit positions in the corresponding byte element of `d` are set to 1. Otherwise, the corresponding byte element of `d` is set to 0.

vec_genmasks_8: Vector Generate Mask (Byte)

Purpose

Generates a bit mask for each byte element.

Syntax

```
d = vec_genmasks_8(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 150. Return and parameter types for vec_genmasks_8

d	**a**	**b**
vector unsigned char	unsigned char[1]	unsigned char[1]
Note: 1. It must be a literal.		

Result value

A bit mask is generated for each byte element of `d`. For each bit mask, the bit positions in the range that starts at the bit position specified by `a` and ends at the bit position specified by `b` are set to 1. All other bit positions are set to 0.

Notes:

- If `a` or `b` is greater than or equal to 8, the compiler uses the value modulo 8.
- If `a` is greater than `b`, the bit mask wraps around to start at the top. For example, each element of the result of `vec_genmasks_8(5,2)` is 0b11100111.

vec_genmasks_16: Vector Generate Mask (Halfword)

Purpose

Generates a bit mask for each halfword element.

Syntax

```
d = vec_genmasks_16(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 151. Return and parameter types for vec_genmasks_16

d	a	b
vector unsigned short	unsigned char[1]	unsigned char[1]
Note: 1. It must be a literal.		

Result value

A bit mask is generated for each halfword element of `d`. For each bit mask, the bit positions in the range that starts at the bit position specified by `a` and ends at the bit position specified by `b` are set to 1. All other bit positions are set to 0.

Notes:

- If `a` or `b` is greater than or equal to 16, the compiler uses the value modulo 16.
- If `a` is greater than `b`, the bit mask wraps around to start at the top.

vec_genmasks_32: Vector Generate Mask (Word)

Purpose

Generates a bit mask for each word element.

Syntax

```
d = vec_genmasks_32(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 152. Return and parameter types for vec_genmasks_32

d	a	b
vector unsigned int	unsigned char[1]	unsigned char[1]
Note: 1. It must be a literal.		

Result value

A bit mask is generated for each word element of `d`. For each bit mask, the bit positions in the range that starts at the bit position specified by `a` and ends at the bit position specified by `b` are set to 1. All other bit positions are set to 0.

Notes:
- If `a` or `b` is greater than or equal to 32, the compiler uses the value modulo 32.
- If `a` is greater than `b`, the bit mask wraps around to start at the top.

vec_genmasks_64: Vector Generate Mask (Doubleword)

Purpose

Generates a bit mask for each doubleword element.

Syntax

```
d = vec_genmasks_64(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 153. Return and parameter types for vec_genmasks_64

d	a	b
vector unsigned long long	unsigned char[1]	unsigned char[1]
Note: 1. It must be a literal.		

Result value

A bit mask is generated for each doubleword element in `d`. For each bit mask, the bit positions in the range that starts at the bit position specified by `a` and ends at the bit position specified by `b` are set to 1. All other bit positions are set to 0.

Notes:
- If `a` or `b` is greater than or equal to 64, the compiler uses the value modulo 64.
- If `a` is greater than `b`, the bit mask wraps around to start at the top.

Copy until zero functions

This topic collection describes vector built-in functions that are used to copy vector elements until a zero element is encountered.

vec_cp_until_zero: Vector Copy Until Zero

Purpose

Copies the elements from a vector until an element that has a value of zero is encountered.

Syntax

```
d = vec_cp_until_zero(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 154. Return and parameter types for vec_cp_until_zero

d	a
vector signed char	vector signed char
vector unsigned char	vector unsigned char
vector bool char	vector bool char
vector signed short	vector signed short
vector unsigned short	vector unsigned short
vector bool short	vector bool short
vector signed int	vector signed int
vector unsigned int	vector unsigned int
vector bool int	vector bool int

Result value

Starting from the first vector element, the vector elements are copied from `a` to `d` until a vector element of `a` that has a value of zero is encountered. The remaining vector elements of `d` are set to zero. If no elements of `a` have a value of zero, the entire vector is copied to `d`.

vec_cp_until_zero_cc: Vector Copy Until Zero with Condition Code

Purpose

Copies the elements from a vector until an element that has a value of zero is encountered. It also returns a condition code that indicates whether all the vector elements are copied.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 155. Return and parameter types for vec_cp_until_zero

d	a	b
vector signed char	vector signed char	int *
vector unsigned char	vector unsigned char	
vector bool char	vector bool char	
vector signed short	vector signed short	
vector unsigned short	vector unsigned short	
vector bool short	vector bool short	
vector signed int	vector signed int	
vector unsigned int	vector unsigned int	
vector bool int	vector bool int	

Result value

Starting from the first vector element, the vector elements are copied from `a` to `d` until a vector element of `a` that has a value of zero is encountered. The remaining vector elements of `d` are set to zero. If no elements of `a` have a value of zero, the entire vector is copied to `d`.

If not all the vector elements are copied because an element of `a` has a value of zero, `c` is set to 0. Otherwise, if all elements of `a` are nonzero, `c` is set to 3.

Load and store functions

This topic collection describes vector built-in functions that are used to load and store vectors.

vec_ld2f: Vector Load 2 Float

Purpose

Loads two consecutive floats, which take eight bytes in total, from the address that is specified by a pointer, and extends them to a vector of type `vector double`.

Note: This function is emulated.

Syntax

```
d = vec_ld2f(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 156. Return and parameter types for vec_ld2f

d	a
vector double	const float *

Result value

The value of `d[0]` is `(double)(*a)`, and the value of `d[1]` is `(double)(*(a+1))`.

vec_load_bndry: Vector Load to Block Boundary

Purpose

Returns a vector that contains the 16 bytes that are loaded starting from a given address unless the specified boundary condition is encountered.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 157. Return and parameter types for vec_load_bndry

d	a	b
vector signed char	const signed char *	unsigned short[1]
vector unsigned char	const unsigned char *	
vector signed short	const signed short *	
vector unsigned short	const unsigned short *	
vector signed int	const signed int *	
vector unsigned int	const unsigned int *	
vector signed long long	const signed long long *	
vector unsigned long long	const unsigned long long *	
vector double	const double *	

Note:

1. It must be a literal whose value is 64, 128, 256, 512, 1024, 2048, or 4096.

Result value

`d` contains the 16 bytes that are loaded starting from `a`. If the specified boundary condition `b` is encountered during the load, the rest of the bytes in `d` are undefined.

vec_load_len: Vector Load with Length

Purpose

Returns a vector with content loaded from a given address from byte 0 to the byte number, which is specified by an integer modulo 16, plus 1.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 158. Return and parameter types for vec_load_len

d	a	b
vector signed char	const signed char *	unsigned int
vector unsigned char	const unsigned char *	
vector signed short	const signed short *	
vector unsigned short	const unsigned short *	
vector signed int	const signed int *	
vector unsigned int	const unsigned int *	
vector signed long long	const signed long long *	
vector unsigned long long	const unsigned long long *	
vector double	const double *	

Result value

`d` contains the content loaded from `*a` from byte 0 to the byte number, which is specified by `b` modulo 16, plus 1. The remaining bytes of `d` are set to 0.

vec_load_pair: Vector Load Pair

Purpose

Returns a vector with the 0-indexed and 1-indexed element set to the specified values.

Note: This function might be emulated.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 159. Return and parameter types for vec_load_pair

d	a	b
vector signed long long	signed long long	signed long long
vector unsigned long long	unsigned long long	unsigned long long

Result value

`d[0]` is set to the value of `a`, and `d[1]` is set to the value of `b`.

vec_st2f: Vector Store 2 Float

Purpose

Rounds a vector of type `vector double` as two floats and stores them, which take eight bytes in total, to the specified location.

Note: This function is emulated.

Syntax

```
d = vec_st2f(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 160. Return and parameter types for vec_st2f

d	**a**	**b**
void	vector double	float *

Result value

The eight bytes starting from address `b` are set to the values that are produced by rounding `a[0]` and `a[1]` to floats.

vec_store_len: Vector Store with Length

Purpose

Stores a number of bytes from a vector to the specified location.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 161. Return and parameter types for vec_store_len

d	**a**	**b**	**c**
void	vector signed char	signed char *	unsigned int
	vector unsigned char	unsigned char *	
	vector signed short	signed short *	
	vector unsigned short	unsigned short *	
	vector signed int	signed int *	
	vector unsigned int	unsigned int *	
	vector signed long long	signed long long *	
	vector unsigned long long	unsigned long long *	
	vector double	double *	

Result value

The `c+1` number of bytes starting from address `b` are set to the values of the corresponding `c+1` bytes of `a`.

Note: If `c` is greater than 15, only 16 bytes are stored.

vec_xld2: Vector Load 2 Doubleword

Purpose

Loads a vector from two 8-byte elements at the memory address that is specified by a displacement and a pointer.

Syntax

```
d = vec_xld2(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 162. Return and parameter types for vec_xld2

d	a	b
vector signed char	long	signed char *
vector unsigned char		unsigned char *
vector signed short		signed short *
vector unsigned short		unsigned short *
vector signed int		signed int *
vector unsigned int		unsigned int *
vector signed long long		signed long long *
vector unsigned long long		unsigned long long *
vector double		double *

Result value

The content of d is loaded from the address that is obtained by adding displacement a and the rvalue of pointer b.

vec_xlw4: Vector Load 4 Word

Purpose

Loads a vector from four 4-byte elements at the memory address that is specified by a displacement and a pointer.

Syntax

```
d = vec_xlw4(a, b)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 163. Return and parameter types for vec_xlw4

d	a	b
vector signed char	long	signed char *
vector unsigned char		unsigned char *
vector signed short		signed short *
vector unsigned short		unsigned short *
vector signed int		signed int *
vector unsigned int		unsigned int *

Result value

The content of d is loaded from the address that is obtained by adding displacement a and the rvalue of pointer b.

vec_xstd2: Vector Store 2 Doubleword

Purpose

Puts a vector as two 8-byte elements to the memory address specified by a displacement and a pointer.

Syntax

```
d = vec_xstd2(a, b, c)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 164. Return and parameter types for vec_xstd2

d	a	b	c
void	vector signed char	long	signed char *
	vector unsigned char		unsigned char *
	vector signed short		signed short *
	vector unsigned short		unsigned short *
	vector signed int		signed int *
	vector unsigned int		unsigned int *
	vector signed long long		signed long long *
	vector unsigned long long		unsigned long long *
	vector double		double *

Result value

This function puts vector a as two 8-byte elements to the memory address that is obtained by adding displacement b and the rvalue of pointer c.

vec_xstw4: Vector Store 4 Word

Purpose

Puts a vector as four 4-byte elements to the memory address specified by a displacement and a pointer.

Syntax

```
d = vec_xstw4(a, b, c)
```

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 165. Return and parameter types for vec_xstw4

d	a	b	c
void	vector signed char	long	signed char *
	vector unsigned char		unsigned char *
	vector signed short		signed short *
	vector unsigned short		unsigned short *
	vector signed int		signed int *
	vector unsigned int		unsigned int *

Result value

This function puts vector `a` as four 4-byte elements to the memory address that is obtained by adding displacement `b` and the rvalue of pointer `c`.

Logical calculation functions

This topic collection describes vector built-in functions for logical calculation.

vec_cntlz: Vector Count Leading Zeros

Purpose

Computes the count of leading zero bits in each element of a vector.

Syntax

```
d = vec_cntlz(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 166. Return and parameter types for vec_cntlz

d	a
vector unsigned char	vector signed char
	vector unsigned char
vector unsigned short	vector signed short
	vector unsigned short

Table 166. Return and parameter types for vec_cntlz (continued)

d	a
vector unsigned int	vector signed int
	vector unsigned int
vector unsigned long long	vector signed long long
	vector unsigned long long

Result value

Each element of `d` is set to the number of leading zero bits in the corresponding element of `a`.

vec_cnttz: Vector Count Trailing Zeros

Purpose

Computes the count of trailing zero bits in each element of a vector.

Syntax

```
d = vec_cnttz(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 167. Return and parameter types for vec_cnttz

d	a
vector unsigned char	vector signed char
	vector unsigned char
vector unsigned short	vector signed short
	vector unsigned short
vector unsigned int	vector signed int
	vector unsigned int
vector unsigned long long	vector signed long long
	vector unsigned long long

Result value

Each element of `d` is set to the number of tailing zero bits in the corresponding element of `a`.

vec_nor: Vector NOR

Purpose

Performs a bitwise NOR operation on two vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 168. Return and parameter types for vec_nor

d	**a**	**b**
vector signed char	vector signed char	vector signed char
		vector bool char
	vector bool char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector unsigned char
vector bool char	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
	vector bool short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector unsigned short
vector bool short	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
	vector bool int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector unsigned int
vector bool int	vector bool int	vector bool int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
	vector bool long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector unsigned long long
vector bool long long	vector bool long long	vector bool long long
vector double	vector bool long long	vector double
	vector double	vector bool long long
		vector double

Result value

`d` is the result of performing a bitwise NOR operation on `a` and `b`.

Note: `vector double` does not cause an IEEE exception.

vec_popcnt: Vector Population Count

Purpose

Computes the population count, that is, the number of bits that are set to one, in each element of a vector.

Syntax

```
d = vec_popcnt(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 169. Return and parameter types for vec_popcnt

d	**a**
vector unsigned char	vector signed char
	vector unsigned char
vector unsigned short	vector signed short
	vector unsigned short
vector unsigned int	vector signed int
	vector unsigned int
vector unsigned long long	vector signed long long
	vector unsigned long long

Result value

Each element of `d` is set to the number of set bits in the corresponding element of `a`.

Note: This function is emulated except for `vector signed char` and `vector unsigned char`.

Merge functions

This topic collection describes vector built-in functions that are used to merge vectors.

vec_mergeh: Vector Merge High

Purpose

Merges the most significant halves, also known as the high halves, of two vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 170. Return and parameter types for vec_mergeh

d	**a**	**b**
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector bool char	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector bool short	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector bool int	vector bool int	vector bool int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector bool long long	vector bool long long	vector bool long long
vector double	vector double	vector double

Result value

The values of the even-numbered elements of `d` are copied, in order, from the elements in the most significant half of `a`. The values of odd-numbered elements of `d` are copied, in order, from the elements in the most significant half of `b`.

Example

The following figure illustrates the operation on operands of type `vector signed int`, `vector unsigned int`, or `vector bool int`.

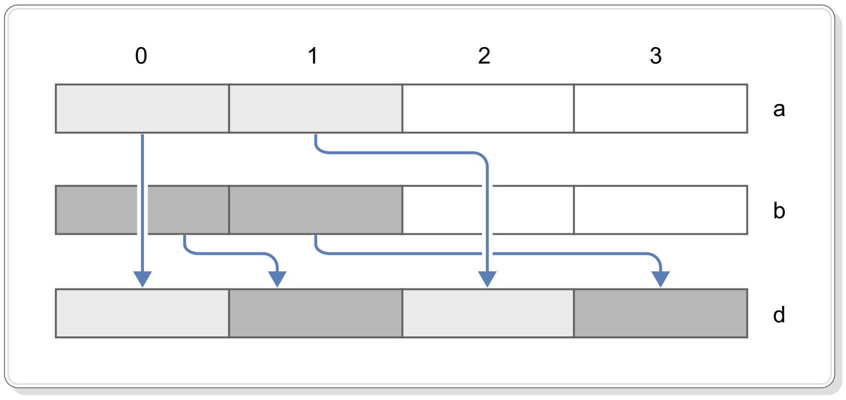


Figure 7. Merge 32-bit high-order elements

vec_mergel: Vector Merge Low

Purpose

Merges the least significant halves, also known as the low halves, of two vectors.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 171. Return and parameter types for vec_mergel

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector bool char	vector bool char	vector bool char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector bool short	vector bool short	vector bool short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector bool int	vector bool int	vector bool int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector bool long long	vector bool long long	vector bool long long
vector double	vector double	vector double

Result value

The values of the even-numbered elements of `d` are copied, in order, from the elements in the least significant half of `a`. The values of the odd-numbered elements of `d` are copied, in order, from the elements in the least significant half of `b`.

Example

The following figure illustrates the operation on operands of type `vector signed int`, `vector unsigned int`, or `vector bool int`.

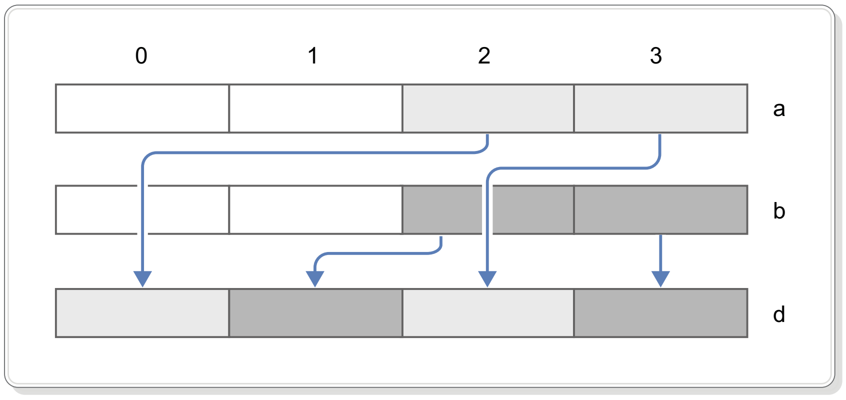


Figure 8. Merge 32-bit low-order elements

Pack and unpack functions

This topic collection describes vector built-in functions for packing and unpacking vector elements.

vec_pack: Vector Pack

Purpose

Packs content from each element of two vectors into the result vector.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 172. Return and parameter types for vec_pack

d	a	b
vector signed char	vector signed short	vector signed short
vector unsigned char	vector unsigned short	vector unsigned short
vector bool char	vector bool short	vector bool short
vector signed short	vector signed int	vector signed int
vector unsigned short	vector unsigned int	vector unsigned int
vector bool short	vector bool int	vector bool int
vector signed int	vector signed long long	vector signed long long
vector unsigned int	vector unsigned long long	vector unsigned long long
vector bool int	vector bool long long	vector bool long long

Result value

The value of each element of `d` is copied from the low half of the corresponding element of the result of concatenating `a` and `b`.

Example

The following figure illustrates the operation on operands of type `vector signed int`, `vector unsigned int`, or `vector bool int`.

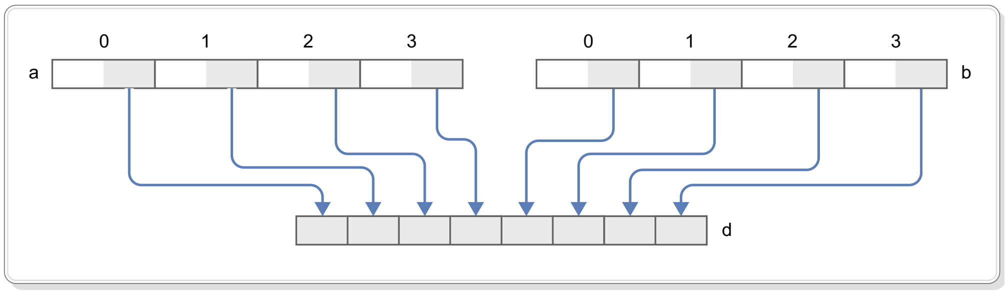


Figure 9. Pack eight 32-bit unsigned integer elements to eight 16-bit unsigned integer elements

vec_packs: Vector Pack Saturate

Purpose

Packs content from each element of two vectors into the result vector using saturated values, which equal the boundary values if overflow occurs.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 173. Return and parameter types for vec_packs

d	a	b
vector signed char	vector signed short	vector signed short
vector unsigned char	vector unsigned short	vector unsigned short
vector signed short	vector signed int	vector signed int
vector unsigned short	vector unsigned int	vector unsigned int
vector signed int	vector signed long long	vector signed long long
vector unsigned int	vector unsigned long long	vector unsigned long long

Result value

The value of each element of `d` is the saturated value of the corresponding element of the result of concatenating `a` and `b`.

Example

The following figure illustrates the operation on operands of type `vector signed int` or `vector unsigned int`.

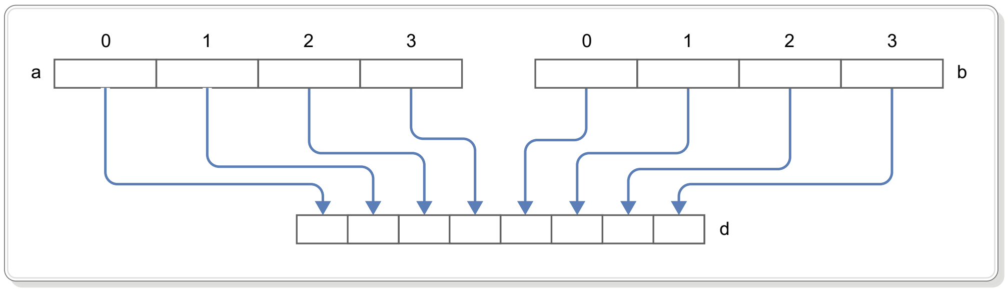


Figure 10. Pack eight 32-bit integer elements to eight 16-bit integer elements

In the following example, value 65535 and -65536 are out of the range of `short int`, so the corresponding elements are set to the boundary values, 32767 and -32768, in the overflow direction respectively.

```
vector signed int a = {65535, -65536, 0, -1};
vector signed short b = vec_packs(a, a);
//b is {32767, -32768, 0, -1, 32767, -32768, 0, -1}.
```

vec_packs_cc: Vector Pack Saturate with Condition Code

Purpose

Packs content from each element of two vectors into the result vector using saturated values, which equal the boundary values if overflow occurs. It also returns a condition code.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 174. Return and parameter types for vec_packs_cc

d	a	b	c
vector signed char	vector signed short	vector signed short	int *
vector unsigned char	vector unsigned short	vector unsigned short	
vector signed short	vector signed int	vector signed int	
vector unsigned short	vector unsigned int	vector unsigned int	
vector signed int	vector signed long long	vector signed long long	
vector unsigned int	vector unsigned long long	vector unsigned long long	

Result value

The value of each element of `d` is the saturated value of the corresponding element of the result of concatenating `a` and `b`. For the signed types, `c` is set to the resulting condition code from the VECTOR PACK SATURATE (VPKS) instruction. For the unsigned types, `c` is set to the resulting condition code from the VECTOR PACK LOGICAL SATURATE (VPKLS) instruction.

vec_packsu: Vector Pack Saturated Unsigned

Purpose

Packs content from each element of two vectors into the result vector using saturated unsigned values, which equal the boundary values if overflow occurs.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 175. Return and parameter types for vec_packsu

d	a	b
vector unsigned char	vector signed short	vector signed short
	vector unsigned short	vector unsigned short

Table 175. Return and parameter types for vec_packsu (continued)

d	a	b
vector unsigned short	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
vector unsigned int	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long

Result value

The value of each element of `d` is the saturated unsigned value of the corresponding element of the result of concatenating `a` and `b`.

Note: This function is emulated on vector signed values.

vec_packsu_cc: Vector Pack Saturated Unsigned with Condition Code

Purpose

Packs content from each element of two vectors into the result vector using saturated unsigned values, which equal the boundary values if overflow occurs. It also returns a condition code.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 176. Return and parameter types for vec_packsu_cc

d	a	b	c
vector unsigned char	vector unsigned short	vector unsigned short	int *
vector unsigned short	vector unsigned int	vector unsigned int	
vector unsigned int	vector unsigned long long	vector unsigned long long	

Result value

The value of each element of `d` is the saturated unsigned value of the corresponding element of the result of concatenating `a` and `b`. `c` is set to the resulting condition code from the VECTOR PACK LOGICAL SATURATE (VPKLS) instruction.

vec_unpackh: Vector Unpack High Element

Purpose

Unpacks with sign extension the most significant half, also known as the high half, of a vector into the result vector whose elements have a larger size.

Syntax

```
d = vec_unpackh(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 177. Return and parameter types for vec_unpackh

d	a
vector signed short	vector signed char
vector unsigned short	vector unsigned char
vector bool short	vector bool char
vector signed int	vector signed short
vector unsigned int	vector unsigned short
vector bool int	vector bool short
vector signed long long	vector signed int
vector unsigned long long	vector unsigned int
vector bool long long	vector bool int

Result value

The value of each element of `d` is the value of the corresponding element in the most significant half of `a`.

Example

The following figure illustrates the operation on operand of type `vector signed short`, `vector unsigned short`, or `vector bool short`.

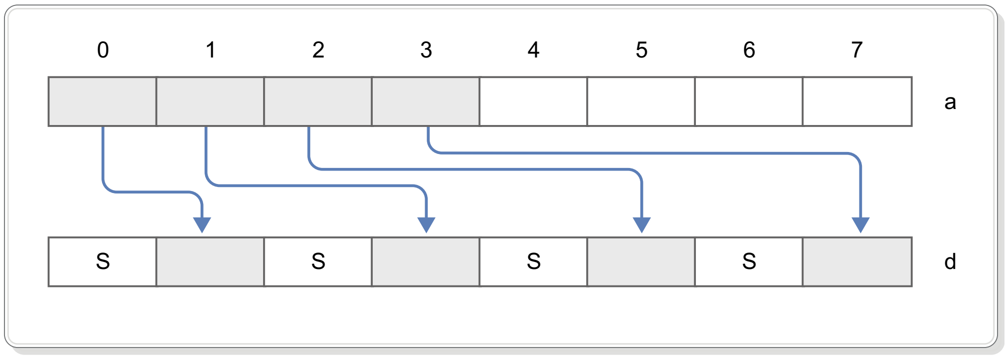


Figure 11. Unpack 16-bit high-order signed integer elements to 32-bit signed integer elements

vec_unpackl: Vector Unpack Low Element
Purpose

Unpacks with sign extension the least significant half, also known as the low half, of a vector into the result vector whose elements have a larger size.

Syntax

```
d = vec_unpackl(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 178. Return and parameter types for vec_unpackl

d	a
vector signed short	vector signed char
vector unsigned short	vector unsigned char
vector bool short	vector bool char
vector signed int	vector signed short
vector unsigned int	vector unsigned short
vector bool int	vector bool short
vector signed long long	vector signed int
vector unsigned long long	vector unsigned int
vector bool long long	vector bool int

Result value

The value of each element of `d` is the value of the corresponding element in the least significant half of `a`.

Example

The following figure illustrates the operation on operand of type `vector signed short`, `vector unsigned short`, or `vector bool short`.

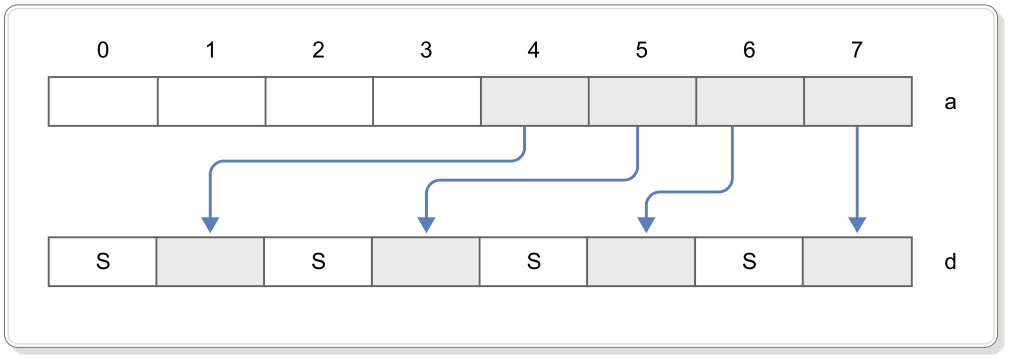


Figure 12. Unpack 16-bit low-order signed integer elements to 32-bit signed integer elements

Replicate functions

This topic collection describes vector built-in functions that are used to replicate vector elements.

vec_splat: Vector Splat

Purpose

Sets all elements of a vector to the value of the designated vector element.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 179. Return and parameter types for vec_splat

d	a	b
vector signed char	vector signed char	unsigned char[1]
vector unsigned char	vector unsigned char	
vector bool char	vector bool char	
vector signed short	vector signed short	unsigned char[2]
vector unsigned short	vector unsigned short	
vector bool short	vector bool short	
vector signed int	vector signed int	unsigned char[3]
vector unsigned int	vector unsigned int	
vector bool int	vector bool int	
vector signed long long	vector signed long long	unsigned char[4]
vector unsigned long long	vector unsigned long long	
vector bool long long	vector bool long long	
vector double	vector double	
Notes:		
1. It must be a literal whose value is in the range 0 - 15 inclusive.		
2. It must be a literal whose value is in the range 0 - 7 inclusive.		
3. It must be a literal whose value is in the range 0 - 3 inclusive.		
4. It must be a literal whose value is 0 or 1.		

Result value

Each element of `d` is set to the value of `a[b]`.

vec_splat_s8: Vector Splat Signed Byte

Purpose

Returns a vector with each of the 16 signed 8-bit elements equal to a given value.

Syntax

```
d = vec_splat_s8(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 180. Return and parameter types for vec_splat_s8

d	a
vector signed char	signed long long[1]
Note:	
1. It must be a literal whose value is in the range from -128 to 127 inclusive.	

Result value

`d` contains 16 signed 8-bit elements of value `a`.

vec_splat_s16: Vector Splat Signed Halfword

Purpose

Returns a vector with each of the eight signed 16-bit elements equal to a given value.

Syntax

```
d = vec_splat_s16(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 181. Return and parameter types for vec_splat_s16

d	a
vector signed short	signed short[1]
Note: 1. It must be a literal.	

Result value

`d` contains eight signed 16-bit elements of value `a`.

vec_splat_s32: Vector Splat Signed Word

Purpose

Returns a vector with each of the four signed 32-bit elements equal to a given value.

Syntax

```
d = vec_splat_s32(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 182. Return and parameter types for vec_splat_s32

d	a
vector signed int	signed short[1]
Note: 1. It must be a literal.	

Result value

`d` contains four signed 32-bit elements of value `a`.

vec_splat_s64: Vector Splat Signed Doubleword

Purpose

Returns a vector with each of the two signed 64-bit elements equal to a given value.

Syntax

```
d = vec_splat_s64(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 183. Return and parameter types for vec_splat_s64

d	a
vector signed long long	signed short[1]
Note: 1. It must be a literal.	

Result value

`d` contains two signed 64-bit elements of value `a`.

vec_splat_u8: Vector Splat Unsigned Byte

Purpose

Returns a vector with each of the 16 unsigned 8-bit elements equal to a given value.

Syntax

```
d = vec_splat_u8(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 184. Return and parameter types for vec_splat_u8

d	a
vector unsigned char	unsigned char[1]
Note: 1. It must be a literal.	

Result value

`d` contains 16 unsigned 8-bit elements of value `a`.

vec_splat_u16: Vector Splat Unsigned Halfword

Purpose

Returns a vector with each of the eight unsigned 16-bit elements equal to a given value.

Syntax

```
d = vec_splat_u16(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 185. Return and parameter types for vec_splat_u16

d	a
vector unsigned short	signed short[1]
Note: 1. It must be a literal.	

Result value

`d` contains eight unsigned 16-bit elements of value `a`.

vec_splat_u32: Vector Splat Unsigned Word

Purpose

Returns a vector with each of the four unsigned 32-bit elements equal to a given value.

Syntax

```
d = vec_splat_u32(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 186. Return and parameter types for vec_splat_u32

d	a
vector unsigned int	signed short[1]
Note: 1. It must be a literal.	

Result value

`d` contains four unsigned 32-bit elements of value `a`.

vec_splat_u64: Vector Splat Doubleword

Purpose

Returns a vector with each of the two unsigned 64-bit elements equal to a given value.

Syntax

```
d = vec_splat_u64(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 187. Return and parameter types for vec_splat_u64

d	a
vector unsigned long long	signed short[1]
Note: 1. It must be a literal.	

Result value

`d` contains two unsigned 64-bit elements of value `a`.

vec_splats: Vector Splats

Purpose

Sets all elements of a vector to a given value.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 188. Return and parameter types for vec_splats

d	a
vector signed char	signed char
vector unsigned char	unsigned char
vector signed short	signed short
vector unsigned short	unsigned short
vector signed int	signed int
vector unsigned int	unsigned int
vector signed long long	signed long long
vector unsigned long long	unsigned long long
vector double	double

Result value

Each element of `d` is set to the value of `a`.

Rotate and shift functions

This topic collection describes vector built-in functions that are used to rotate and shift vector elements.

vec_rl: Vector Element Rotate Left

Purpose

Rotates each element of a vector left by a given number of bits.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 189. Return and parameter types for vec_rl

d	a	b
vector signed char	vector signed char	vector unsigned char
vector unsigned char	vector unsigned char	
vector signed short	vector signed short	vector unsigned short
vector unsigned short	vector unsigned short	
vector signed int	vector signed int	vector unsigned int
vector unsigned int	vector unsigned int	
vector signed long long	vector signed long long	vector unsigned long long
vector unsigned long long	vector unsigned long long	

Result value

Each element of `d` is obtained by rotating the corresponding element of `a` left by the number of bits specified by the corresponding element of `b`. If the value of the element in `b` is out of range, the compiler uses the value modulo the number of bits in the element of `a` as the rotating bit.

vec_rl_mask: Vector Element Rotate and Insert Under Mask

Purpose

Rotates each element of a vector left by a given number of bits and overlays with the vector according to a mask.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 190. Return and parameter types for vec_rl_mask

d	a	b	c
vector signed char	vector signed char	vector unsigned char	unsigned char[1]
vector unsigned char	vector unsigned char		
vector signed short	vector signed short	vector unsigned short	
vector unsigned short	vector unsigned short		
vector signed int	vector signed int	vector unsigned int	
vector unsigned int	vector unsigned int		
vector signed long long	vector signed long long	vector unsigned long long	
vector unsigned long long	vector unsigned long long		

Note:

1. It must be a literal.

Result value

Each bit of `d` is obtained as follows:

- If the corresponding bit mask `b` is 1, it gets the corresponding bit from the intermediate result after rotating each element of vector `a` left by the number of bits specified by `c`. If `c` is out of range, the compiler uses `c` modulo the number of bits in the element of `a` as the rotating bit.
- If the corresponding bit mask `b` is 0, it gets the corresponding bit from `a`.

vec_rli: Vector Element Rotate Left Immediate

Purpose

Rotates each element of a vector left by a given number of bits.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 191. Return and parameter types for vec_rli

d	a	b
vector signed char	vector signed char	unsigned long
vector unsigned char	vector unsigned char	
vector signed short	vector signed short	
vector unsigned short	vector unsigned short	
vector signed int	vector signed int	
vector unsigned int	vector unsigned int	
vector signed long long	vector signed long long	
vector unsigned long long	vector unsigned long long	

Result value

Each element of d is obtained by rotating the corresponding element of a left by the number of bits specified by b. If b is out of range, the compiler uses b modulo the number of bits in the element of a as the rotating bit.

vec_slb: Vector Shift Left by Byte

Purpose

Shifts each element of a vector left by a given number of bytes.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 192. Return and parameter types for vec_slb

d	a	b
vector signed char	vector signed char	vector signed char
		vector unsigned char
vector unsigned char	vector unsigned char	vector signed char
		vector unsigned char
vector signed short	vector signed short	vector signed short
		vector unsigned short
vector unsigned short	vector unsigned short	vector signed short
		vector unsigned short
vector signed int	vector signed int	vector signed int
		vector unsigned int
vector unsigned int	vector unsigned int	vector signed int
		vector unsigned int
vector signed long long	vector signed long long	vector signed long long
		vector unsigned long long
vector unsigned long long	vector unsigned long long	vector signed long long
		vector unsigned long long

Result value

Each element of d is obtained by shifting the corresponding element of a left by the number of bytes specified by the second to fifth bits of byte element seven of b. The bits that are shifted out are replaced by zeros.

vec_sld: Vector Shift Left Double by Byte

Purpose

Shifts two concatenated vectors left by a given number of bytes.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 193. Return and parameter types for vec_sld

d	**a**	**b**	**c**
vector signed char	vector signed char	vector signed char	unsigned long long[1]
vector unsigned char	vector unsigned char	vector unsigned char	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
vector signed long long	vector signed long long	vector signed long long	
vector unsigned long long	vector unsigned long long	vector unsigned long long	
vector double	vector double	vector double	
Note: 1. It must be a literal whose value is in the range 0 - 15 inclusive.			

Result value

d contains the most significant 16 bytes obtained by concatenating a and b and shifting the intermediate result left by the number of bytes specified by c.

Example

The following figure illustrates the operation when the value of c is four.

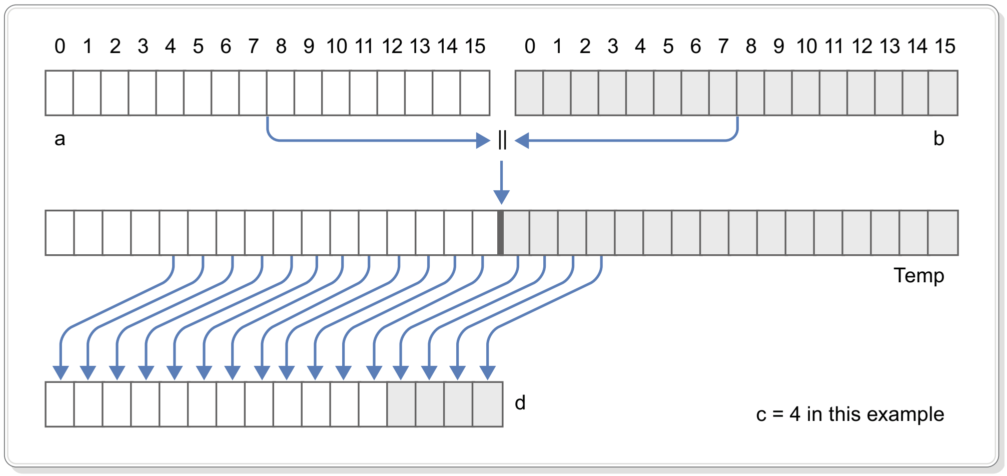


Figure 13. Bitwise conditional select of vector contents

vec_sldw: Vector Shift Left Double by Word Immediate

Purpose

Shifts two concatenated vectors left by multiples of 4 bytes as specified.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 194. Return and parameter types for vec_sldw

d	a	b	c
vector signed char	vector signed char	vector signed char	int[1]
vector unsigned char	vector unsigned char	vector unsigned char	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
vector signed long long	vector signed long long	vector signed long long	
vector unsigned long long	vector unsigned long long	vector unsigned long long	
vector double	vector double	vector double	

Note:
1. It must be a literal whose value is in the range 0 - 3 inclusive.

Result value

`d` contains the four leftmost 4-byte values obtained by concatenating `a` and `b` and shifting the intermediate result left by `c` * 4 bytes.

vec_sll: Vector Shift Left

Purpose

Performs a left shift on each element of a vector by the low-order three bits of all byte elements of another vector.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 195. Return and parameter types for vec_sll

d	**a**	**b**
vector signed char	vector signed char	vector unsigned char
		vector unsigned short
		vector unsigned int
vector unsigned char	vector unsigned char	vector unsigned char
		vector unsigned short
		vector unsigned int
vector bool char	vector bool char	vector unsigned char
		vector unsigned short
		vector unsigned int
vector signed short	vector signed short	vector unsigned char
		vector unsigned short
		vector unsigned int
vector unsigned short	vector unsigned short	vector unsigned char
		vector unsigned short
		vector unsigned int
vector bool short	vector bool short	vector unsigned char
		vector unsigned short
		vector unsigned int
vector signed int	vector signed int	vector unsigned char
		vector unsigned short
		vector unsigned int
vector unsigned int	vector unsigned int	vector unsigned char
		vector unsigned short
		vector unsigned int
vector bool int	vector bool int	vector unsigned char
		vector unsigned short
		vector unsigned int
vector signed long long	vector signed long long	vector unsigned char
		vector unsigned short
		vector unsigned int
vector unsigned long long	vector unsigned long long	vector unsigned char
		vector unsigned short
		vector unsigned int
vector bool long long	vector bool long long	vector unsigned char
		vector unsigned short
		vector unsigned int

Result value

Each element of d is obtained by shifting the corresponding element of a left by the number of bits specified by the low-order three bits of every byte of b. The bits that are shifted out are replaced by zeros.

Note: The low-order three bits of all byte elements in b must be same; otherwise, the result is undefined.

vec_srab: Vector Shift Right Arithmetic by Byte

Purpose

Performs an algebraic right shift on a vector by a given number of bytes.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 196. Return and parameter types for vec_srab

d	a	b
vector signed char	vector signed char	vector signed char
		vector unsigned char
vector unsigned char	vector unsigned char	vector signed char
		vector unsigned char
vector signed short	vector signed short	vector signed short
		vector unsigned short
vector unsigned short	vector unsigned short	vector signed short
		vector unsigned short
vector signed int	vector signed int	vector signed int
		vector unsigned int
vector unsigned int	vector unsigned int	vector signed int
		vector unsigned int
vector signed long long	vector signed long long	vector signed long long
		vector unsigned long long
vector unsigned long long	vector unsigned long long	vector signed long long
		vector unsigned long long
vector double	vector double	vector signed long long
		vector unsigned long long

Result value

Each element of d is obtained by shifting the corresponding element of a right by the number of bytes specified by bit 1 - 4 of byte element seven of b. The bits that are shifted out are replaced by copies of the most significant bit of the corresponding element of a.

vec_sral: Vector Shift Right Arithmetic

Purpose

Performs an algebraic right shift on a vector by a given number of bits.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 197. Return and parameter types for vec_sral

d	a	b
vector signed char	vector signed char	vector unsigned char
		vector unsigned short
		vector unsigned int
vector unsigned char	vector unsigned char	vector unsigned char
		vector unsigned short
		vector unsigned int
vector bool char	vector bool char	vector unsigned char
		vector unsigned short
		vector unsigned int
vector signed short	vector signed short	vector unsigned char
		vector unsigned short
		vector unsigned int
vector unsigned short	vector unsigned short	vector unsigned char
		vector unsigned short
		vector unsigned int
vector bool short	vector bool short	vector unsigned char
		vector unsigned short
		vector unsigned int
vector signed int	vector signed int	vector unsigned char
		vector unsigned short
		vector unsigned int
vector unsigned int	vector unsigned int	vector unsigned char
		vector unsigned short
		vector unsigned int
vector bool int	vector bool int	vector unsigned char
		vector unsigned short
		vector unsigned int
vector signed long long	vector signed long long	vector unsigned char
		vector unsigned short
		vector unsigned int

Table 197. Return and parameter types for vec_sral (continued)

d	a	b
vector unsigned long long	vector unsigned long long	vector unsigned char
		vector unsigned short
		vector unsigned int
vector bool long long	vector bool long long	vector unsigned char
		vector unsigned short
		vector unsigned int

Result value

Each element of `d` is obtained by shifting the corresponding element of `a` right by the number of bits specified by the low-order three bits of every byte of `b`. The bits that are shifted out are replaced by copies of the most significant bit of the corresponding element of `a`.

Note: The low-order three bits of all byte elements of `b` must be same; otherwise, the result is undefined.

vec_srb: Vector Shift Right by Byte

Purpose

Performs a right shift on a vector by a given number of bytes.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 198. Return and parameter types for vec_srb

d	a	b
vector signed char	vector signed char	vector signed char
		vector unsigned char
vector unsigned char	vector unsigned char	vector signed char
		vector unsigned char
vector signed short	vector signed short	vector signed short
		vector unsigned short
vector unsigned short	vector unsigned short	vector signed short
		vector unsigned short
vector signed int	vector signed int	vector signed int
		vector unsigned int
vector unsigned int	vector unsigned int	vector signed int
		vector unsigned int
vector signed long long	vector signed long long	vector signed long long
		vector unsigned long long

Table 198. Return and parameter types for vec_srb (continued)

d	a	b
vector unsigned long long	vector unsigned long long	vector signed long long
		vector unsigned long long
vector double	vector double	vector signed long long
		vector unsigned long long

Result value

Each element of `d` is obtained by shifting the corresponding element of `a` right by the number of bytes specified by bit 1 - 4 of byte element seven of `b`. The bits that are shifted out are replaced by zeros.

vec_srl: Vector Shift Right

Purpose

Performs a right shift on a vector by a given number of bits.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 199. Return and parameter types for vec_srl

d	a	b
vector signed char	vector signed char	vector unsigned char
		vector unsigned short
		vector unsigned int
vector unsigned char	vector unsigned char	vector unsigned char
		vector unsigned short
		vector unsigned int
vector bool char	vector bool char	vector unsigned char
		vector unsigned short
		vector unsigned int
vector signed short	vector signed short	vector unsigned char
		vector unsigned short
		vector unsigned int
vector unsigned short	vector unsigned short	vector unsigned char
		vector unsigned short
		vector unsigned int
vector bool short	vector bool short	vector unsigned char
		vector unsigned short
		vector unsigned int

Table 199. Return and parameter types for vec_srl (continued)

d	a	b
vector signed int	vector signed int	vector unsigned char
		vector unsigned short
		vector unsigned int
vector unsigned int	vector unsigned int	vector unsigned char
		vector unsigned short
		vector unsigned int
vector bool int	vector bool int	vector unsigned char
		vector unsigned short
		vector unsigned int
vector signed long long	vector signed long long	vector unsigned char
		vector unsigned short
		vector unsigned int
vector unsigned long long	vector unsigned long long	vector unsigned char
		vector unsigned short
		vector unsigned int
vector bool long long	vector bool long long	vector unsigned char
		vector unsigned short
		vector unsigned int

Result value

Each element of `d` is obtained by shifting the corresponding element of `a` right by the number of bits specified by the low-order three bits of every byte of `b`. The bits that are shifted out are replaced by zeros.

Note: The low-order three bits of all byte elements in `b` must be same; otherwise, the result is undefined.

Rounding and conversion functions

This topic collection describes vector built-in functions for rounding and conversion.

vec_ceil: Vector Ceiling

Purpose

Returns a vector that contains the smallest representable floating-point integers greater than or equal to the corresponding elements of a given vector.

Syntax

```
d = vec_ceil(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 200. Return and parameter types for vec_ceil

d	a
vector double	vector double

Result value

The value of each element of `d` is the smallest representable floating-point integer that is greater than or equal to the corresponding element of `a`.

Note: `vec_ceil` provides the same functionality as `vec_roundp` except that `vec_ceil` triggers the IEEE-inexact exception.

Related reference:

"vec_roundp: Vector Round toward Positive Infinity" on page 205

vec_ctd: Vector Convert to Double

Purpose

Converts an integer vector to a double-precision floating-point vector.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 201. Return and parameter types for vec_ctd

d	a	b
vector double	vector signed long long	int[1]
	vector unsigned long long	
Note: 1. It must be a literal whose value is in the range 0 - 31 inclusive.		

Result value

Each element of `d` is obtained by converting the corresponding element of `a` from integer to double-precision floating point and dividing the intermediate result by 2 to the power of `b`.

Note: Current BFP rounding mode is used on the conversion.

vec_ctsl: Vector Convert to signed long long

Purpose

Converts a double-precision floating-point vector to a signed long long integer vector.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 202. Return and parameter types for vec_ctsl

d	a	b
vector signed long long	vector double	int[1]
Note: 1. It must be a literal whose value is in the range 0 - 31 inclusive.		

Result value

Each element of d is obtained by multiplying the corresponding element of a by 2 to the power of b and rounding the intermediate result towards 0 into an integer.

vec_ctul: Vector Convert to unsigned long long

Purpose

Converts a double-precision floating-point vector to an unsigned long long integer vector.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 203. Return and parameter types for vec_ctul

d	a	b
vector signed long long	vector double	int[1]
Note: 1. It must be a literal whose value is in the range 0 - 31 inclusive.		

Result value

Each element of d is obtained by multiplying the corresponding element of a by 2 to the power of b and rounding the intermediate result towards 0 into an unsigned integer.

vec_extend_s64: Vector Sign Extend to Doubleword

Purpose

Returns a vector containing sign-extended results of the rightmost element of each doubleword of a given vector.

Syntax

```
d = vec_extend_s64(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 204. Return and parameter types for vec_extend_s64

d	**a**
vector signed long long	vector signed char
	vector signed short
	vector signed int

Result value

`d` contains the results of performing sign extension on the rightmost element of each doubleword of `a`.

Example

If `a` is of type `vector signed char`, `a[7]` and `a[15]` are the rightmost elements of each doubleword of `a`. The value of `d[0]` is the result of performing sign extension on `a[7]`, and the value of `d[1]` is the result of performing sign extension on `a[15]`.

vec_floor: Vector Floor

Purpose

Returns a vector that contains the largest representable floating-point integers less than or equal to the corresponding elements of a given vector.

Syntax

```
d = vec_floor(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 205. Return and parameter types for vec_floor

d	**a**
vector double	vector double

Result value

The value of each element of `d` is the largest representable floating-point integer that is less than or equal to the corresponding element of `a`.

Note: `vec_floor` provides the same functionality as `vec_roundm` except that `vec_floor` triggers the IEEE-inexact exception.

Related reference:

"vec_roundm: Vector Round toward Negative Infinity" on page 205

vec_round: Vector Round to Nearest

Purpose

Rounds the elements of a vector by using the IEEE round-to-nearest rounding mode.

Syntax

```
d = vec_round(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 206. Return and parameter types for vec_round

d	a
vector double	vector double

Result value

Each element of `d` has the value of rounding the corresponding element of `a` to the nearest representable floating-point integer by using the IEEE round-to-nearest rounding mode.

Note: The IEEE-inexact exception is suppressed.

vec_roundc: Vector Round to Current

Purpose

Rounds the double-precision floating-point elements of a vector to integer by using the current rounding mode.

Syntax

```
d = vec_roundc(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 207. Return and parameter types for vec_roundc

d	a
vector double	vector double

Result value

Each element of `d` has the value of rounding the corresponding double-precision floating-point element of `a` to integer by using the current rounding mode.

Note: The IEEE-inexact exception is suppressed.

vec_roundm: Vector Round toward Negative Infinity

Purpose

Returns a vector that contains the largest representable floating-point integers less than or equal to the corresponding elements of a given vector.

Syntax

```
d = vec_roundm(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 208. Return and parameter types for vec_roundm

d	a
vector double	vector double

Result value

The value of each element of `d` is the largest representable floating-point integer that is less than or equal to the corresponding element of `a`.

Note: `vec_roundm` provides the same functionality as `vec_floor` except that `vec_roundm` does not trigger the IEEE-inexact exception.

Related reference:

"vec_floor: Vector Floor" on page 203

vec_roundp: Vector Round toward Positive Infinity

Purpose

Returns a vector that contains the smallest representable floating-point integers greater than or equal to the corresponding elements of a given vector.

Syntax

```
d = vec_roundp(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 209. Return and parameter types for vec_roundp

d	a
vector double	vector double

Result value

The value of each element of `d` is the smallest representable floating-point integer that is greater than or equal to the corresponding element of `a`.

Note: `vec_roundp` provides the same functionality as `vec_ceil`, except that `vec_roundp` does not trigger the IEEE-inexact exception.

Related reference:

"vec_ceil: Vector Ceiling" on page 200

vec_roundz: Vector Round toward Zero

Purpose

Returns a vector that contains the truncated values of the corresponding elements of a given vector.

Syntax

```
d = vec_roundz(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 210. Return and parameter types for vec_roundz

d	**a**
vector double	vector double

Result value

The value of each element of `d` is the result value of truncating the corresponding element of `a` to an integer.

Note: `vec_roundz` provides the same functionality as `vec_trunc` except that `vec_roundz` does not trigger the IEEE-inexact exception.

Related reference:

"vec_trunc: Vector Truncate"

vec_trunc: Vector Truncate

Purpose

Returns a vector that contains the truncated values of the corresponding elements of a given vector.

Syntax

```
d = vec_trunc(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 211. Return and parameter types for vec_trunc

d	**a**
vector double	vector double

Result value

The value of each element of `d` is the result value of truncating the corresponding element of `a` to an integer.

Note: `vec_trunc` provides the same functionality as `vec_roundz` except that `vec_trunc` triggers the IEEE-inexact exception.

Related reference:
"vec_roundz: Vector Round toward Zero" on page 206

Testing functions

This topic collection describes vector built-in functions for testing.

vec_fp_test_data_class: Vector Floating-Point Test Data Class

Purpose

Tests the element class on the vector elements based on the specified condition.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 212. Return and parameter types for vec_fp_test_data_class

d	**a**	**b**	**c**
vector bool long long	vector double	unsigned short[1]	int *
Note: 1. It must be a literal whose value is in the range 0 - 4095 inclusive.			

Result value

This function uses the VECTOR FP TEST DATA CLASS IMMEDIATE (VFTCIDB) instruction to test the BFP element class on the elements of `a`:

- `d` is the first operand in the VFTCIDB instruction.
- `a` is the second operand in the VFTCIDB instruction.
- `b` is the third operand in the VFTCIDB instruction and specifies the condition of the instruction.

The value that is referred to by `c` is set to the condition code set by the VFTCIDB instruction.

vec_test_mask: Vector Test under Mask

Purpose

Performs a zeros or ones test on the selected bits on the masked vector.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 213. Return and parameter types for vec_test_mask

d	a	b
int	vector signed char	vector unsigned char
	vector unsigned char	
	vector signed short	vector unsigned short
	vector unsigned short	
	vector signed int	vector unsigned int
	vector unsigned int	
	vector signed long long	vector unsigned long long
	vector unsigned long long	
	vector double	vector unsigned long long

Result value

`d` is set to the condition code set by the Vector Test Under Mask (VTM) instruction as follows:

- 0 if all the selected bits of `a` under mask `b` are set to zero or if all bits of mask `b` are set to zero.
- 1 if some of the selected bits of `a` under mask `b` are set to zero and others are set to one.
- 3 if all the selected bits of `a` under mask `b` are set to one.

All elements predication functions

This topic collection describes vector built-in functions that are used to compare vector elements to determine whether all elements meet the compare criteria.

vec_all_eq: All Elements Equal

Purpose

Tests whether all sets of the corresponding elements of two vectors are equal.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 214. Return and parameter types for vec_all_eq

d	a	b
int	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Result value

If each element of `a` is equal to the corresponding element of `b`, `d` is 1. Otherwise, `d` is 0.

vec_all_ge: All Elements Greater Than or Equal

Purpose

Tests whether all elements of the first vector parameter are greater than or equal to the corresponding elements of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 215. Return and parameter types for vec_all_ge

d	a	b
int	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Result value

If all elements of `a` are greater than or equal to the corresponding elements of `b`, `d` is 1; otherwise, `d` is 0.

vec_all_gt: All Elements Greater Than

Purpose

Tests whether all elements of the first vector parameter are greater than the corresponding elements of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 216. Return and parameter types for vec_all_gt

d	a	b
int	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Result value

If all elements of a are greater than the corresponding elements of b, d is 1. Otherwise, d is 0.

vec_all_le: All Elements Less Than or Equal

Purpose

Tests whether all elements of the first vector parameter are less than or equal to the corresponding elements of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 217. Return and parameter types for vec_all_le

d	a	b
int	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Result value

If all elements of `a` are less than or equal to the corresponding elements of `b`, `d` is 1. Otherwise, `d` is 0.

vec_all_lt: All Elements Less Than

Purpose

Tests whether all elements of the first vector parameter are less than the corresponding elements of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 218. Return and parameter types for vec_all_lt

d	a	b
int	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Result value

If all elements of `a` are less than the corresponding elements of `b`, `d` is 1. Otherwise, `d` is 0.

vec_all_nan: All Elements Not a Number

Purpose

Tests whether each element of a vector is a `NaN`.

Syntax

```
d = vec_all_nan(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 219. Return and parameter types for vec_all_nan

d	a
int	vector double

Result value

If each element of `a` is a `NaN`, `d` is 1. Otherwise, `d` is 0.

vec_all_ne: All Elements Not Equal

Purpose

Tests whether all sets of the corresponding elements of two vectors are unequal.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 220. Return and parameter types for vec_all_ne

d	a	b
int	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Result value

If each element of `a` is not equal to the corresponding element of `b`, `d` is 1. Otherwise, `d` is 0.

vec_all_nge: All Elements Not Greater Than or Equal

Purpose

Tests whether all elements of the first vector parameter are neither greater than nor equal to the corresponding elements of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 221. Return and parameter types for vec_all_nge

d	a	b
int	vector double	vector double

Result value

If all elements of `a` are neither greater than nor equal to the corresponding elements of `b`, `d` is 1. Otherwise, `d` is 0.

vec_all_ngt: All Elements Not Greater Than

Purpose

Tests whether all elements of the first vector parameter are not greater than the corresponding elements of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 222. Return and parameter types for vec_all_ngt

d	a	b
int	vector double	vector double

Result value

If all elements of `a` are not greater than the corresponding elements of `b`, `d` is 1. Otherwise, `d` is 0.

vec_all_nle: All Elements Not Less Than or Equal

Purpose

Tests whether all elements of the first vector parameter are neither less than nor equal to the corresponding elements of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 223. Return and parameter types for vec_all_nle

d	a	b
int	vector double	vector double

Result value

If all elements of `a` are neither less than nor equal to the corresponding elements of `b`, `d` is 1. Otherwise, `d` is 0.

vec_all_nlt: All Elements Not Less Than

Purpose

Tests whether all elements of the first vector parameter are not less than the corresponding elements of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 224. Return and parameter types for vec_all_nlt

d	a	b
int	vector double	vector double

Result value

If all elements of `a` are not less than the corresponding elements of `b`, `d` is 1. Otherwise, `d` is 0.

vec_all_numeric: All Elements Numeric

Purpose

Tests whether each element of a vector is a numeric, that is, not a `NaN`.

Syntax

```
d = vec_all_numeric(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 225. Return and parameter types for vec_all_numeric

d	a
int	vector double

Result value

If each element of `a` is a numeric, `d` is 1. Otherwise, `d` is 0.

Any element predication functions

This topic collection describes vector built-in functions that are used to compare vector elements to determine whether any element meets the compare criteria.

vec_any_eq: Any Element Equal

Purpose

Tests whether any set of the corresponding elements of two vectors is equal.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 226. Return and parameter types for vec_any_eq

d	a	b
int	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Result value

If any element of `a` is equal to the corresponding element of `b`, `d` is 1. Otherwise, `d` is 0.

vec_any_ge: Any Element Greater Than or Equal

Purpose

Tests whether any element of the first vector parameter is greater than or equal to the corresponding element of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 227. Return and parameter types for vec_any_ge

d	a	b
int	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
	vector double	vector double

Result value

If any element of `a` is greater than or equal to the corresponding element of `b`, `d` is 1. Otherwise, `d` is 0.

vec_any_gt: Any Element Greater Than

Purpose

Tests whether any element of the first vector parameter is greater than the corresponding element of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 228. Return and parameter types for vec_any_gt

d	a	b
int	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
	vector double	vector double

Result value

If any element of `a` is greater than the corresponding element of `b`, `d` is 1. Otherwise, `d` is 0.

vec_any_le: Any Element Less Than or Equal

Purpose

Tests whether any element of the first vector parameter is less than or equal to the corresponding element of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 229. Return and parameter types for vec_any_le

d	a	b
int	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
	vector double	vector double

Result value

If any element of `a` is less than or equal to the corresponding element of `b`, `d` is 1. Otherwise, `d` is 0.

vec_any_lt: Any Element Less Than

Purpose

Tests whether any element of the first vector parameter is less than the corresponding element of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 230. Return and parameter types for vec_any_lt

d	a	b
int	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
	vector double	vector double

Result value

If any element of `a` is less than the corresponding element of `b`, `d` is 1. Otherwise, `d` is 0.

vec_any_nan: Any Element Not a Number

Purpose

Tests whether any element of a vector is a `NaN`.

Syntax

```
d = vec_any_nan(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 231. Return and parameter types for vec_any_nan

d	a
int	vector double

Result value

If any element of `a` is a `NaN`, `d` is 1. Otherwise, `d` is 0.

vec_any_ne: Any Element Not Equal

Purpose

Tests whether any set of the corresponding elements of two vectors is unequal.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 232. Return and parameter types for vec_any_ne

d	a	b
int	vector signed char	vector signed char
		vector bool char
	vector unsigned char	vector unsigned char
		vector bool char
	vector bool char	vector signed char
		vector unsigned char
		vector bool char
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector unsigned short
		vector bool short
	vector bool short	vector signed short
		vector unsigned short
		vector bool short
	vector signed int	vector signed int
		vector bool int
	vector unsigned int	vector unsigned int
		vector bool int
	vector bool int	vector signed int
		vector unsigned int
		vector bool int
	vector signed long long	vector signed long long
		vector bool long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
	vector bool long long	vector signed long long
		vector unsigned long long
		vector bool long long
	vector double	vector double

Result value

If any element of `a` is not equal to the corresponding element of `b`, `d` is 1. Otherwise, `d` is 0.

vec_any_nge: Any Element Not Greater Than or Equal

Purpose

Tests whether any element of the first vector parameter is neither greater than nor equal to the corresponding element of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 233. Return and parameter types for vec_any_nge

d	a	b
int	vector double	vector double

Result value

If any element of `a` is neither greater than nor equal to the corresponding element of `b`, `d` is 1. Otherwise, `d` is 0.

vec_any_ngt: Any Element Not Greater Than

Purpose

Tests whether any element of the first vector parameter is not greater than the corresponding element of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 234. Return and parameter types for vec_any_ngt

d	a	b
int	vector double	vector double

Result value

If any element of `a` is not greater than the corresponding element of `b`, `d` is 1. Otherwise, `d` is 0.

vec_any_nle: Any Element Not Less Than or Equal

Purpose

Tests whether any element of the first vector parameter is neither less than nor equal to the corresponding element of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 235. Return and parameter types for vec_any_nle

d	a	b
int	vector double	vector double

Result value

If any element of `a` is neither less than nor equal to the corresponding element of `b`, `d` is 1. Otherwise, `d` is 0.

vec_any_nlt: Any Element Not Less Than

Purpose

Tests whether any element of the first vector parameter is not less than the corresponding element of the second vector parameter.

Syntax

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

Return and parameter types

The following table describes the types of the return value and function parameters.

Table 236. Return and parameter types for vec_any_nlt

d	a	b
int	vector double	vector double

Result value

If any element of `a` is not less than the corresponding element of `b`, `d` is 1. Otherwise, `d` is 0.

vec_any_numeric: Any Element Numeric

Purpose

Tests whether any element of a vector is a numeric, that is, not a `NaN`.

Syntax

```
d = vec_any_numeric(a)
```

Return and parameter types

The following table describes the types of the return value and function parameter.

Table 237. Return and parameter types for vec_any_numeric

d	a
int	vector double

Result value

If any element of `a` is a numeric, `d` is 1. Otherwise, `d` is 0.

Defining vector built-in functions from the operators

IBM XL C/C++ for Linux on z Systems, V1.2 does not provide the vector built-in functions that perform the same operations as operators, but the XL C/C++ compilers for some other platforms might provide such vector built-in functions. To compile your program that contains these vector built-in functions with IBM XL C/C++ for Linux on z Systems, V1.2, you can use the function-like macros to define these vector built-in functions from operators.

```
#define vec_neg(a) (-(a)) // Vector Negate
#define vec_add(a, b) ((a) + (b)) // Vector Add
#define vec_sub(a, b) ((a) - (b)) // Vector Subtract
#define vec_mul(a, b) ((a) * (b)) // Vector Multiply
#define vec_div(a, b) ((a) / (b)) // Vector Divide
#define vec_and(a, b) ((a) & (b)) // Vector AND
#define vec_or(a, b) ((a) | (b)) // Vector OR
#define vec_xor(a, b) ((a) ^ (b)) // Vector XOR
#define vec_sl(a, b) ((a) << (b)) // Vector Shift Left
#define vec_sr(a, b) ((a) >> (b)) // Vector Shift Right1
#define vec_sra(a, b) ((a) >> (b)) // Vector Shift Right Arithmetic2
```

Notes:

1. The `vec_sr` macro definition can be used on only unsigned vector types so that the correct bits can be inserted on the shifted out bits.
2. The `vec_sra` macro definition can be used only if the first argument is of a signed vector type.

In addition, IBM XL C/C++ for Linux on z Systems, V1.2 does not provide `vec_slo` or `vec_sro`, but you can define them as follows:

```
#define vec_slo(a, b) vec_slb(a, (b) << 64) // Vector Shift Left by Octet
#define vec_sro(a, b) vec_srb(a, (b) << 64) // Vector Shift Right by Octet
```

Related reference:

"vec_slb: Vector Shift Left by Byte" on page 192

"vec_srb: Vector Shift Right by Byte" on page 198

Related information:

"Binary expressions" on page 71

Debug support for vector programming

The **-g** option generates debugging information for vector programming.

The debugging information can be examined by **gdb**, the target debugger on Linux on z Systems. **gdb** version 7.8 or higher provides vector type support. **gdb** does not rely on the consumer library of IBM.

Related information in the *XL C/C++ Compiler Reference*

-g

Chapter 9. Using the high performance libraries

IBM XL C/C++ for Linux on z Systems, V1.2 is shipped with a set of libraries for high-performance mathematical computing:

- The Mathematical Acceleration Subsystem (MASS) is a set of libraries of tuned mathematical intrinsic functions that provide improved performance over the corresponding standard system math library functions. MASS is described in “Using the Mathematical Acceleration Subsystem (MASS) libraries.”
- The Automatically Tuned Linear Algebra Software (ATLAS) libraries contain all the Basic Linear Algebra Subprograms (BLAS) and a subset of the Linear Algebra PACKage (LAPACK) routines. For details, see Using the Automatically Tuned Linear Algebra Software (ATLAS) libraries .

Using the Mathematical Acceleration Subsystem (MASS) libraries

XL C/C++ is shipped with a set of Mathematical Acceleration Subsystem (MASS) libraries for high-performance mathematical computing.

The MASS libraries consist of a set of scalar libraries described in “Using the scalar libraries” and a set of vector libraries described in “Using the vector libraries” on page 234. The scalar and the vector libraries are tuned for IBM zEnterprise® EC12 (zEC12), IBM zEnterprise BC12 (zBC12), and IBM z13™ models. Note that accuracy and exception handling might not be identical in MASS functions and system library functions.

The MASS functions must run with the default rounding mode and floating-point exception trapping settings.

“Compiling and linking a program with MASS” on page 238 describes how to compile and link a program that uses the MASS libraries, and how to selectively use the MASS scalar library functions in conjunction with the regular system libraries.

The following table describes the scalar and the vector libraries that are shipped with the compiler.

Architecture	Scalar libraries	Vector libraries
zEC12 and zBC12	`libmass.zEC12.a`	`libmassv.zEC12.a`
z13	`libmass.z13.a`	`libmassv.z13.a`

Related external information

Mathematical Acceleration Subsystem website, available at http://www.ibm.com/software/awdtools/mass/

Using the scalar libraries

The MASS scalar libraries contain an accelerated set of frequently used math intrinsic functions that provide improved performance over the corresponding standard system library functions. The MASS scalar functions are used when you explicitly link the MASS scalar libraries.

If you want to call the MASS scalar functions, you can take the following steps:

1. Provide the prototypes for the functions by including `math.h` and `mass.h` in your source files.
2. Link the MASS scalar library with your application. For instructions, see "Compiling and linking a program with MASS" on page 238.

The scalar libraries shipped with XL C/C++ are listed below:

libmass.zEC12.a
Contains scalar functions that are tuned for the IBM zEnterprise EC12 and IBM zEnterprise BC12 architectures.

libmass.z13.a
Contains scalar functions that are tuned for the IBM z13 architecture.

The MASS scalar functions accept double-precision parameters and return a double-precision result, or accept single-precision parameters and return a single-precision result, except `sincos` which gives 2 double-precision results. They are summarized in Table 238.

Table 238. MASS scalar functions

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
acos	acosf	Returns the arccosine of x	double acos (double x);	float acosf (float x);
acosh	acoshf	Returns the hyperbolic arccosine of x	double acosh (double x);	float acoshf (float x);
	anint	Returns the rounded integer value of x		float anint (float x);
asin	asinf	Returns the arcsine of x	double asin (double x);	float asinf (float x);
asinh	asinhf	Returns the hyperbolic arcsine of x	double asinh (double x);	float asinhf (float x);
atan2	atan2f	Returns the arctangent of x/y	double atan2 (double x, double y);	float atan2f (float x, float y);
atan	atanf	Returns the arctangent of x	double atan (double x);	float atanf (float x);
atanh	atanhf	Returns the hyperbolic arctangent of x	double atanh (double x);	float atanhf (float x);
cbrt	cbrtf	Returns the cube root of x	double cbrt (double x);	float cbrtf (float x);
copysign	copysignf	Returns x with the sign of y	double copysign (double x,double y);	float copysignf (float x);
cos	cosf	Returns the cosine of x	double cos (double x);	float cosf (float x);
cosh	coshf	Returns the hyperbolic cosine of x	double cosh (double x);	float coshf (float x);
cosisin		Returns a complex number with the real part the cosine of x and the imaginary part the sine of x.	double_Complex cosisin (double);	
dnint		Returns the nearest integer to x (as a double)	double dnint (double x);	

Table 238. MASS scalar functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
erf	erff	Returns the error function of `x`	double erf (double x);	float erff (float x);
erfc	erfcf	Returns the complementary error function of `x`	double erfc (double x);	float erfcf (float x);
exp	expf	Returns the exponential function of `x`	double exp (double x);	float expf (float x);
expm1	expm1f	Returns (the exponential function of `x`) - 1	double expm1 (double x);	float expm1f (float x);
hypot	hypotf	Returns the square root of $x^2 + y^2$	double hypot (double x, double y);	float hypotf (float x, float y);
lgamma	lgammaf	Returns the natural logarithm of the absolute value of the Gamma function of `x`	double lgamma (double x);	float lgammaf (float x);
log	logf	Returns the natural logarithm of `x`	double log (double x);	float logf (float x);
log10	log10f	Returns the base 10 logarithm of `x`	double log10 (double x);	float log10f (float x);
log1p	log1pf	Returns the natural logarithm of (`x` + 1)	double log1p (double x);	float log1pf (float x);
pow	powf	Returns `x` raised to the power `y`	double pow (double x, double y);	float powf (float x, float y);
rint	rintf	Returns the nearest integer to `x` (as a double)	double rint (double x);	float rintf (float x);
rsqrt		Returns the reciprocal of the square root of `x`	double rsqrt (double x);	
sin	sinf	Returns the sine of `x`	double sin (double x);	float sinf (float x);
sincos		Sets `*s` to the sine of `x` and `*c` to the cosine of `x`	void sincos (double x, double* s, double* c);	
sinh	sinhf	Returns the hyperbolic sine of `x`	double sinh (double x);	float sinhf (float x);
sqrt		Returns the square root of `x`	double sqrt (double x);	
tan	tanf	Returns the tangent of `x`	double tan (double x);	float tanf (float x);
tanh	tanhf	Returns the hyperbolic tangent of `x`	double tanh (double x);	float tanhf (float x);

Notes:

- The trigonometric functions (`sin`, `cos`, `tan`) return NaN (Not-a-Number) for large arguments (where the absolute value is greater than 2^{50}pi).
- In some cases, the MASS functions are not as accurate as the ones in the `libm.a` library, and they might handle edge cases differently (`sqrt(Inf)`, for example).

Using the vector libraries

If you want to call any of the MASS vector functions, you can do so by including `massv.h` in your source files and linking your application with the appropriate vector library. Information about linking is provided in "Compiling and linking a program with MASS" on page 238.

The vector libraries shipped with XL C/C++ are listed below:

libmassv.zEC12.a
: Contains vector functions that are tuned for the IBM zEnterprise EC12 and IBM zEnterprise BC12 architectures.

libmassv.z13.a
: Contains vector functions that are tuned for the IBM z13 architecture.

The single-precision and double-precision floating-point functions contained in the vector libraries are summarized in Table 239 on page 235. The integer functions contained in the vector libraries are summarized in Table 240 on page 237. Note that in C and C++ applications, only call by reference is supported, even for scalar arguments.

With the exception of a few functions (described in the following paragraph), all of the floating-point functions in the vector libraries accept three parameters:

- A double-precision (for double-precision functions) or single-precision (for single-precision functions) vector output parameter
- A double-precision (for double-precision functions) or single-precision (for single-precision functions) vector input parameter
- An integer vector-length parameter.

The functions are of the form

`function_name (y,x,n)`

where *y* is the target vector, *x* is the source vector, and *n* is the vector length. The parameters *y* and *x* are assumed to be double-precision for functions with the prefix `v`, and single-precision for functions with the prefix `vs`. As an example, the following code outputs a vector *y* of length 500 whose elements are exp(x[i]), where i=0,...,499:

```
#include <massv.h>

double x[500], y[500];
int n;
n = 500;
...
vexp (y, x, &n);
```

The functions `vdiv`, `vsincos`, `vpow`, and `vatan2` (and their single-precision versions, `vsdiv`, `vssincos`, `vspow`, and `vsatan2`) take four arguments. The functions `vdiv`, `vpow`, and `vatan2` take the arguments (*z*,*x*,*y*,*n*). The function `vdiv` outputs a vector *z* whose elements are x[i]/y[i], where i=0,..,*n–1. The function `vpow` outputs a vector *z* whose elements are $x[i]^{y[i]}$, where i=0,..,*n–1. The function `vatan2` outputs a vector *z* whose elements are atan(x[i]/y[i]), where i=0,..,*n–1. The function `vsincos` takes the arguments (*y*,*z*,*x*,*n*), and outputs two vectors, *y* and *z*, whose elements are sin(x[i]) and cos(x[i]), respectively.

In `vcosisin(y,x,n)` and `vscosisin(y,x,n)`, *x* is a vector of *n* elements and the function outputs a vector *y* of *n* `__Complex` elements of the form (cos(x[i]),sin(x[i])).

Table 239. MASS floating-point vector functions

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vacos	vsacos	Sets `y[i]` to the arc cosine of `x[i]`, for i=0,..,*n-1	void vacos (double y[], double x[], int *n);	void vsacos (float y[], float x[], int *n);
vacosh	vsacosh	Sets `y[i]` to the hyperbolic arc cosine of `x[i]`, for i=0,..,*n-1	void vacosh (double y[], double x[], int *n);	void vsacosh (float y[], float x[], int *n);
vasin	vsasin	Sets `y[i]` to the arc sine of `x[i]`, for i=0,..,*n-1	void vasin (double y[], double x[], int *n);	void vsasin (float y[], float x[], int *n);
vasinh	vsasinh	Sets `y[i]` to the hyperbolic arc sine of `x[i]`, for i=0,..,*n-1	void vasinh (double y[], double x[], int *n);	void vsasinh (float y[], float x[], int *n);
vatan[1]	vsatan[1]	Sets `y[i]` to the arc tangent of `x[i]`, for i=0,..,*n-1	void vatan (double y[], double x[], int *n);	void vsatan (float y[], float x[], int *n);
vatan2	vsatan2	Sets `z[i]` to the arc tangent of `x[i]/y[i]`, for i=0,..,*n-1	void vatan2 (double z[], double x[], double y[], int *n);	void vsatan2 (float z[], float x[], float y[], int *n);
vatanh	vsatanh	Sets `y[i]` to the hyperbolic arc tangent of `x[i]`, for i=0,..,*n-1	void vatanh (double y[], double x[], int *n);	void vsatanh (float y[], float x[], int *n);
vcbrt	vscbrt	Sets `y[i]` to the cube root of `x[i]`, for i=0,..,*n-1	void vcbrt (double y[], double x[], int *n);	void vscbrt (float y[], float x[], int *n);
vcos	vscos	Sets `y[i]` to the cosine of `x[i]`, for i=0,..,*n-1	void vcos (double y[], double x[], int *n);	void vscos (float y[], float x[], int *n);
vcosh	vscosh	Sets `y[i]` to the hyperbolic cosine of `x[i]`, for i=0,..,*n-1	void vcosh (double y[], double x[], int *n);	void vscosh (float y[], float x[], int *n);
vcosisin	vscosisin	Sets the real part of `y[i]` to the cosine of `x[i]` and the imaginary part of `y[i]` to the sine of `x[i]`, for i=0,..,*n-1	void vcosisin (double _Complex y[], double x[], int *n);	void vscosisin (float _Complex y[], float x[], int *n);
vdint		Sets `y[i]` to the integer truncation of `x[i]`, for i=0,..,*n-1	void vdint (double y[], double x[], int *n);	
vdiv	vsdiv	Sets `z[i]` to `x[i]/y[i]`, for i=0,..,*n–1	void vdiv (double z[], double x[], double y[], int *n);	void vsdiv (float z[], float x[], float y[], int *n);
vdnint		Sets `y[i]` to the nearest integer to `x[i]`, for i=0,..,*n-1	void vdnint (double y[], double x[], int *n);	
verf[1]	vserf[1]	Sets `y[i]` to the error function of `x[i]`, for i=0,..,*n-1	void verf (double y[], double x[], int *n)	void vserf (float y[], float x[], int *n)
verfc[1]	vserfc[1]	Sets `y[i]` to the complementary error function of `x[i]`, for i=0,..,*n-1	void verfc (double y[], double x[], int *n)	void vserfc (float y[], float x[], int *n)

Table 239. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vexp	vsexp	Sets `y[i]` to the exponential function of `x[i]`, for i=0,..,*n-1	void vexp (double y[], double x[], int *n);	void vsexp (float y[], float x[], int *n);
vexp2[1]	vsexp2[1]	Sets `y[i]` to 2 raised to the power of `x[i]`, for i=1,..,*n-1	void vexp2 (double y[], double x[], int *n);	void vsexp2 (float y[], float x[], int *n);
vexpm1	vsexpm1	Sets `y[i]` to (the exponential function of `x[i]`)-1, for i=0,..,*n-1	void vexpm1 (double y[], double x[], int *n);	void vsexpm1 (float y[], float x[], int *n);
vexp2m1[1]	vsexp2m1[1]	Sets `y[i]` to (2 raised to the power of `x[i]`) - 1, for i=1,..,*n-1	void vexp2m1 (double y[], double x[], int *n);	void vsexp2m1 (float y[], float x[], int *n);
vhypot[1]	vshypot[1]	Sets `z[i]` to the square root of the sum of the squares of `x[i]` and `y[i]`, for i=0,..,*n-1	void vhypot (double z[], double x[], double y[], int *n);	void vshypot (float z[], float x[], float y[], int *n);
vlgamma[1]	vslgamma[1]	Sets `y[i]` to the natural logarithm of the absolute value of the Gamma function of `x[i]`, for i=0,..,*n-1	void vlgamma (double y[], double x[], int *n);	void vslgamma (float y[], float x[], int *n);
vlog	vslog	Sets `y[i]` to the natural logarithm of `x[i]`, for i=0,..,*n-1	void vlog (double y[], double x[], int *n);	void vslog (float y[], float x[], int *n);
vlog2[1]	vslog2[1]	Sets `y[i]` to the base-2 logarithm of `x[i]`, for i=1,..,*n-1	void vlog2 (double y[], double x[], int *n);	void vslog2 (float y[], float x[], int *n);
vlog10	vslog10	Sets `y[i]` to the base-10 logarithm of `x[i]`, for i=0,..,*n-1	void vlog10 (double y[], double x[], int *n);	void vslog10 (float y[], float x[], int *n);
vlog1p	vslog1p	Sets `y[i]` to the natural logarithm of (`x[i]`+1), for i=0,..,*n-1	void vlog1p (double y[], double x[], int *n);	void vslog1p (float y[], float x[], int *n);
vlog21p[1]	vslog21p[1]	Sets `y[i]` to the base-2 logarithm of (`x[i]`+1), for i=1,..,*n-1	void vlog21p (double y[], double x[], int *n);	void vslog21p (float y[], float x[], int *n);
vpow	vspow	Sets `z[i]` to `x[i]` raised to the power `y[i]`, for i=0,..,*n-1	void vpow (double z[], double x[], double y[], int *n);	void vspow (float z[], float x[], float y[], int *n);
vqdrt	vsqdrt	Sets `y[i]` to the fourth root of `x[i]`, for i=0,..,*n-1	void vqdrt (double y[], double x[], int *n);	void vsqdrt (float y[], float x[], int *n);
vrcbrt	vsrcbrt	Sets `y[i]` to the reciprocal of the cube root of `x[i]`, for i=0,..,*n-1	void vrcbrt (double y[], double x[], int *n);	void vsrcbrt (float y[], float x[], int *n);
vrec	vsrec	Sets `y[i]` to the reciprocal of `x[i]`, for i=0,..,*n-1	void vrec (double y[], double x[], int *n);	void vsrec (float y[], float x[], int *n);
vrqdrt	vsrqdrt	Sets `y[i]` to the reciprocal of the fourth root of `x[i]`, for i=0,..,*n-1	void vrqdrt (double y[], double x[], int *n);	void vsrqdrt (float y[], float x[], int *n);

Table 239. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vrsqrt	vsrsqrt	Sets `y[i]` to the reciprocal of the square root of `x[i]`, for i=0,..,*n-1	void vrsqrt (double y[], double x[], int *n);	void vsrsqrt (float y[], float x[], int *n);
vsin	vssin	Sets `y[i]` to the sine of `x[i]`, for i=0,..,*n-1	void vsin (double y[], double x[], int *n);	void vssin (float y[], float x[], int *n);
vsincos	vssincos	Sets `y[i]` to the sine of `x[i]` and `z[i]` to the cosine of `x[i]`, for i=0,..,*n-1	void vsincos (double y[], double z[], double x[], int *n);	void vssincos (float y[], float z[], float x[], int *n);
vsinh	vssinh	Sets `y[i]` to the hyperbolic sine of `x[i]`, for i=0,..,*n-1	void vsinh (double y[], double x[], int *n);	void vssinh (float y[], float x[], int *n);
vsqrt	vssqrt	Sets `y[i]` to the square root of `x[i]`, for i=0,..,*n-1	void vsqrt (double y[], double x[], int *n);	void vssqrt (float y[], float x[], int *n);
vtan	vstan	Sets `y[i]` to the tangent of `x[i]`, for i=0,..,*n-1	void vtan (double y[], double x[], int *n);	void vstan (float y[], float x[], int *n);
vtanh	vstanh	Sets `y[i]` to the hyperbolic tangent of `x[i]`, for i=0,..,*n-1	void vtanh (double y[], double x[], int *n);	void vstanh (float y[], float x[], int *n);
Note: 1. These functions are available only on z13 architechtures.				

Integer functions are of the form *function_name* (*x*[], **n*), where `x[]` is a vector of 4-byte (for `vpopcnt4`) or 8-byte (for `vpopcnt8`) numeric objects (integral or floating-point), and `*n` is the vector length.

Table 240. MASS integer vector library functions

Function	Description	Prototype
vpopcnt4	Returns the total number of 1 bits in the concatenation of the binary representation of `x[i]`, for i=0,..,*n–1 , where `x` is a vector of 32-bit objects.	unsigned int vpopcnt4 (void *x, int *n)
vpopcnt8	Returns the total number of 1 bits in the concatenation of the binary representation of `x[i]`, for i=0,..,*n–1 , where `x` is a vector of 64-bit objects.	unsigned int vpopcnt8 (void *x, int *n)

Overlap of input and output vectors

In most applications, the MASS vector functions are called with disjoint input and output vectors; that is, the two vectors do not overlap in memory. Another common usage scenario is to call them with the same vector for both input and output parameters (for example, `vsin (y, y, &n)`). Other kinds of overlap (where input and output vectors are neither disjoint nor identical) should be avoided, since they might produce unexpected results:

- For calls to vector functions that take one input and one output vector (for example, `vsin (y, x, &n)`):

The vectors `x[0:n-1]` and `y[0:n-1]` must be either disjoint or identical, or unexpected results might be obtained.

- For calls to vector functions that take two input vectors (for example, `vatan2 (y, x1, x2, &n)`):

  The previous restriction applies to both pairs of vectors `y`,`x1` and `y`,`x2`. That is, `y[0:n-1]` and `x1[0:n-1]` must be either disjoint or identical; and `y[0:n-1]` and `x2[0:n-1]` must be either disjoint or identical.
- For calls to vector functions that take two output vectors (for example, `vsincos (y1, y2, x, &n)`):

  The above restriction applies to both pairs of vectors `y1`,`x` and `y2`,`x`. That is, `y1[0:n-1]` and `x[0:n-1]` must be either disjoint or identical; and `y2[0:n-1]` and `x[0:n-1]` must be either disjoint or identical. Also, the vectors `y1[0:n-1]` and `y2[0:n-1]` must be disjoint.

Compiling and linking a program with MASS

To compile an application that calls the functions in the following MASS libraries, specify the corresponding library names on the **-l** link option.

Table 241. The scalar and vector MASS library

MASS library	Library name
Scalar library	**mass.zEC12**
	mass.z13
Vector library	**massv.zEC12**
	massv.z13

For example, if the MASS libraries are installed in the default directory, you can use one of the following commands:

Link object file progc with scalar library libmass.zEC12.a and vector library libmassv.zEC12.a (31-bit code)

```
xlc progc.c -o progc -lmass.zEC12 -lmassv.zEC12
```

Link object file progc with scalar library libmass.z13.a and vector library libmassv.z13.a (64-bit code)

```
xlc progc.c -o progc -lmass.z13 -lmassv.z13 -q64
```

Using the scalar library with the math system library

If you want to use the scalar library for some functions and the normal math library `libm.a` for other functions, follow this procedure to compile and link your program:

1. Use the **ar** command to extract the object files of the wanted functions from the scalar library. For most functions, the object file name is the function name followed by `.s31.o` for 31-bit or `.s64.o` for 64-bit mode.[1] For example, to extract the object file for the `tan` function in the 31-bit zEC12 architecture, the command would be:
   ```
   ar -x tan.s31.o libmass.zEC12.a
   ```
2. Archive the extracted object files into another library:
   ```
   ar -qv libfasttan.a tan.s31.o
   ranlib libfasttan.a
   ```
3. Create the final executable using **xlc**, specifying **-lfasttan** instead of **-lmass.zEC12**:
   ```
   xlc sample.c -o sample -Ldir_containing_libfasttan -lfasttan
   ```

This links only the `tan` function from MASS (now in `libfasttan.a`) and the remainder of the math functions from the standard system library.

Exceptions:

1. The `sin` and `cos` functions are both contained in the object files sincos.s31.o and sincos.s64.o. The `cosisin` and `sincos` functions are both contained in the object files cosisin.s31.o and cosisin.s64.o.
2. The XL C/C++ `pow` function is contained in the object files dxy.s31.o and dxy.s64.o.

Note: The `cos` and `sin` functions will both be exported if either one is exported. `cosisin` and `sincos` will both be exported if either one is exported.

Using the Automatically Tuned Linear Algebra Software (ATLAS) libraries

IBM XL C/C++ for Linux on z Systems, V1.2 is shipped with a set of Automatically Tuned Linear Algebra Software (ATLAS) libraries for algebra high-performance computing.

The ATLAS libraries contain all the Basic Linear Algebra Subprograms (BLAS) and a subset of the Linear Algebra PACKage (LAPACK) routines with interfaces that are provided for both C and FORTRAN 77 across platforms and architectures. Different versions of the ATLAS libraries, including single-threaded, multithreaded, static, and shared ones, are provided and tuned for the IBM zEnterprise EC12 (zEC12), IBM zEnterprise BC12 (zBC12), and IBM z13 models. To use ATLAS libraries that are tuned for the IBM z13 models, ensure that the compiler runs on a Linux distribution that has vector support. C and C++ calling programs are supported with 31-bit and 64-bit linkage.

To call ATLAS functionality in your program, take the following steps:

1. Include the appropriate header files that contain the prototypes of library functions. For header file names, see “The ATLAS libraries and their header files.”
2. Specify the appropriate compiler option, define the appropriate macros, or both. For the information about the compiler options and macros, see “The required specification to use the ATLAS libraries” on page 242.
3. Link with the appropriate ATLAS libraries. For library names, see “The ATLAS libraries and their header files.”

This topic is intended only as a high-level description of ATLAS and the IBM specific extensions and naming convention. For details about the ATLAS libraries, such as lists of functions that are included in the various libraries, visit the Automatically Tuned Linear Algebra Software website at http://math-atlas.sourceforge.net.

The ATLAS libraries and their header files

The static version of the ATLAS main library, the CBLAS library, the LAPACK library, the Fortran BLAS library, and their shared aggregate library are provided. Include the provided header files in your programs to call ATLAS functionality.

Note: To use ATLAS libraries that are tuned for the IBM z13 models, ensure that the compiler runs on a Linux distribution that has vector support.

The aggregate ATLAS library (shared version)

The shared version of the aggregate ATLAS library contains the corresponding single-threaded or multithreaded LAPACK and BLAS functions and all ATLAS symbols that are needed to support the functions.

Table 242. The shared version of the ATLAS library and its header files

Operation type	Library name for zEC12/zBC12	Library name for z13	Library location	Header files to be used with the library	Header file location

Single-threaded	`libsatlas.zEC12.so`	`libsatlas.z13.so`	`/opt/ibm/atlas/1.2.0/lib` (for 31 bit)
			
			`/opt/ibm/atlas/1.2.0/lib64` (for 64 bit)
				`atlas_*`[1]
				`cblas.h`
				`clapack.h`
				`atlas_*f77*`[1]
					`/opt/ibm/atlas/1.2.0/include`

Multithreaded	`libtatlas.zEC12.so`	`libtatlas.z13.so`			

Note:
1. Replace * with the specific header file name or library file name.

The ATLAS main library (static version)

The static version of the ATLAS main library contains the ATLAS specific variants of the BLAS, CBLAS, and LAPACK routines.

A sample provided interface routine is `ATL_dgemm`.

Table 243. The static version of the ATLAS main library and its header files

Operation type	Library name for zEC12/zBC12	Library name for z13	Library location	Header files to be used with the library	Header file location
Single-threaded	`libatlas.zEC12.a` `libtstatlas.zEC12.a`[1]	`libatlas.z13.a` `libtstatlas.z13.a`[1]	`/opt/ibm/atlas/1.2.0/lib` (for 31 bit) `/opt/ibm/atlas/1.2.0/lib64` (for 64 bit)	`atlas_*`[2]	`/opt/ibm/atlas/1.2.0/include`

Notes:
1. This is a test ATLAS library, which can be used to test functionality that is implemented with ATLAS functions.
2. Replace * with the specific header file name or library file name.

The CBLAS library (static version)

The static version of the BLAS library, which is also known as the CBLAS interface, contains the implementation of the C routines of the BLAS algorithms.

A sample provided interface routine is `cblas_dgemm`.

Table 244. The static version of the CBLAS library and its header files

Operation type	Library name for zEC12/zBC12	Library name for z13	Library location	Header files to be used with the library	Header file location
Single-threaded	`libcblas.zEC12.a`	`libcblas.z13.a`	`/opt/ibm/atlas/1.2.0/lib` (for 31 bit) `/opt/ibm/atlas/1.2.0/lib64` (for 64 bit)	`cblas.h`	`/opt/ibm/atlas/1.2.0/include`
Multithreaded	`libptcblas.zEC12.a`	`libptcblas.z13.a`			

The LAPACK library (static version)

The static version of the LAPACK library, which is also known as the CLAPACK interface, contains the implementation of the C routines of the LAPACK algorithms.

A sample provided interface routine is `clapack_dgesv`.

Table 245. The static version of the LAPACK library and its header files

Operation type	Library name for zEC12/zBC12	Library name for z13	Library location	Header files to be used with the library	Header file location
Single-threaded	`liblapack.zEC12.a`	`liblapack.z13.a`	`/opt/ibm/atlas/1.2.0/lib` (for 31 bit) `/opt/ibm/atlas/1.2.0/lib64` (for 64 bit)	`clapack.h`	`/opt/ibm/atlas/1.2.0/include`
Multithreaded	`libptlapack.zEC12.a`	`libptlapack.z13.a`			

The Fortran BLAS library (static version)

The static version of the Fortran BLAS library, which is also known as the BLAS interface, contains the implementation of the FORTRAN 77 routines of the BLAS algorithms.

A sample provided interface routine is `dgemm_`.

Table 246. The static version of the Fortran BLAS library and its header files

Operation type	Library name for zEC12/zBC12	Library name for z13	Library location	Header files to be used with the library	Header file location
Single-threaded	`libf77blas.zEC12.a` `libf77refblas.zEC12.a`[1]	`libf77blas.z13.a` `libf77refblas.z13.a`[1]	`/opt/ibm/atlas/1.2.0/lib` (for 31 bit) `/opt/ibm/atlas/1.2.0/lib64` (for 64 bit)	`atlas_*f77*`[2]	`/opt/ibm/atlas/1.2.0/include`
Multithreaded	`libptf77blas.zEC12.a`	`libptf77blas.z13.a`			

Table 246. The static version of the Fortran BLAS library and its header files (continued)

Operation type	Library name for zEC12/zBC12	Library name for z13	Library location	Header files to be used with the library	Header file location
Notes: 1. You might need to link this library alongside the single-threaded or multithreaded version of the Fortran BLAS library. 2. Replace * with the specific header file name or library file name.					

Notes:

- For each of the base libraries that are mentioned in this topic, the single-threaded and multithreaded versions contain the same list of functions.
- The libraries archive 31-bit C linkage and 64-bit C linkage objects in different directories. The linker chooses the appropriate libraries based on the specified options, environment variables, or both.

The required specification to use the ATLAS libraries

To force zEC12 optimizations and defaults, you must define ATL_ARCH_IBMzEC12 on the command line. To force z13 optimizations and defaults, you must either define ATL_ARCH_IBMz13 or use the **-march=z13** option, or its equivalent, on the command line. If neither of the above is used, the compiler uses z13 defaults.

To use the ATLAS vector functionality, you must specify the **-mzvector** option with the **-march=z13** option or its equivalent and ensure that the compiler runs on a Linux distribution that has vector support.

You might still be able to compile and link your ATLAS application without specifying the required compiler option or defining the macros. However, the program might generate incorrect results.

Related information:

-march (-qarch)

-mzvector

Example 1: Compiling, linking, and running a simple matrix multiplication ATLAS program

This simple sample achieves a multiplication of two matrices, A and B.

Matrices A and B have elements that are randomly generated with values in the range of 0 to 1. The multiplication is done by using these two ways in the following topics:

- By calling the `dgemm` or `cblas_dgemm` BLAS functionality provided by ATLAS to get Matrix C
- By a manual calculation to get Matrix D

The result matrices C and D are congruent. Each of the following topics provides a method to generate Matrix C using the ATLAS functionality.

- “Method 1” on page 243
- “Method 2” on page 244

- "Method 3" on page 246
- "Method 4" on page 247

Sample output is as follows:

```
Using defaults
Matrix A has 2 rows and 6 columns:
0.446 0.992 0.463 0.006 0.269 0.540
0.753 0.943 0.084 0.753 0.921 0.924

Matrix B has 6 rows and 4 columns:
0.976 0.367 0.013 0.304
0.170 0.980 0.736 0.363
0.387 0.318 0.815 0.111
0.514 0.110 0.125 0.296
0.952 0.053 0.917 0.306
0.095 0.310 0.358 0.574

Matrix C has 2 rows and 4 columns:
1.093 1.465 1.554 0.941
2.279 1.645 2.043 1.616

Matrix D has 2 rows and 4 columns:
1.093 1.465 1.554 0.941
2.279 1.645 2.043 1.616
```

Note: Matrix data is organized in the Fortran way, namely columns major.

Method 1

This program contains a C invocation of the Fortran BLAS function `dgemm_` provided by the ATLAS framework.

Observation: In this sample, the invocation of `dgemm_` has no previously declared prototype; the compiler might issue a warning message. Prototypes can be declared by including the `atlas_f77` header files, but source files might not have these header files specified, which means old source code is written before ATLAS.

`sample1.c` is as follows:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

void init(double* matrix, int row, int column)
{
  for (int j = 0; j < column; j++){
    for (int i = 0; i < row; i++){
      matrix[j*row + i] = ((double)rand())/RAND_MAX;
    }
  }
}

void print(const char * name, const double* matrix, int row, int column)
{
  printf("Matrix %s has %d rows and %d columns:\n", name, row, column);
  for (int i = 0; i < row; i++){
    for (int j = 0; j < column; j++){
      printf("%.3f ", matrix[j*row + i]);
    }
    printf("\n");
  }
  printf("\n");
}

int main(int argc, char * argv[])
```

```
{
  int rowsA, colsB, common;
  int i,j,k;

  if (argc != 4){
    printf("Using defaults\n");
    rowsA = 2; colsB = 4; common = 6;
  }
  else{
    rowsA = atoi(argv[1]); colsB = atoi(argv[2]);common = atoi(argv[3]);
  }

  double A[rowsA * common]; double B[common * colsB];
  double C[rowsA * colsB]; double D[rowsA * colsB];

  char transA = 'N', transB = 'N';
  double one = 1.0, zero = 0.0;

  srand(time(NULL));

  init(A, rowsA, common); init(B, common, colsB);

  dgemm_(&transA, &transB, &rowsA, &colsB, &common, &one, A,
         &rowsA, B, &common, &zero, C, &rowsA);

  for(i=0;i<colsB;i++){
    for(j=0;j<rowsA;j++){
      D[i*rowsA+j]=0;
      for(k=0;k<common;k++){
        D[i*rowsA+j]+=A[k*rowsA+j]*B[k+common*i];
      }
    }
  }

  print("A", A, rowsA, common); print("B", B, common, colsB);
  print("C", C, rowsA, colsB); print("D", D, rowsA, colsB);

  return 0;
}
```

To compile the program for IBM zEnterprise EC12 (zEC12) models, enter:

```
xlc -c -march=zEC12 -DATL_ARCH_IBMzEC12 -o sample1.o sample1.c
```

To compile the program for IBM z13 models, enter:

```
xlc -c -march=z13 -o sample1.o sample1.c
```

, or

```
xlc -c -DATL_ARCH_IBMz13 -o sample1.o sample1.c
```

To link the program for IBM zEnterprise EC12 (zEC12) models in static single-threaded mode, enter:

```
xlc sample1.o -march=zEC12 -lgfortran -lf77blas.zEC12 -latlas.zEC12 -o sample1
```

To link the program for IBM z13 models in static single-threaded mode, enter:

```
xlc sample1.o -march=z13 -lgfortran -lf77blas.z13 -latlas.z13 -o sample1
```

Method 2

This program contains a C++ invocation of the Fortran BLAS function `dgemm_` provided by the ATLAS framework.

Observation: As opposed to method 1, the compiler must be explicitly instructed that the function `dgemm_` has C linkage and thus no mangling needs to be attempted. This can be achieved either as specified, or by including the appropriate header file with the extern "C" designation.

`sample2.C` is as follows:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

extern "C"
{
  int dgemm_(char *, char *, int *, int *, int *, double *, double *, int *,
             double *, int *, double *, double *, int *);
}


void init(double* matrix, int row, int column)
{
  for (int j = 0; j < column; j++){
    for (int i = 0; i < row; i++){
      matrix[j*row + i] = ((double)rand())/RAND_MAX;
    }
  }
}

void print(const char * name, const double* matrix, int row, int column)
{
  printf("Matrix %s has %d rows and %d columns:\n", name, row, column);
  for (int i = 0; i < row; i++){
    for (int j = 0; j < column; j++){
      printf("%.3f ", matrix[j*row + i]);
    }
    printf("\n");
  }
  printf("\n");
}

int main(int argc, char * argv[])
{
  int rowsA, colsB, common;
  int i,j,k;

  if (argc != 4){
    printf("Using defaults\n");
    rowsA = 2; colsB = 4; common = 6;
  }
  else{
    rowsA = atoi(argv[1]); colsB = atoi(argv[2]);common = atoi(argv[3]);
  }

  double A[rowsA * common]; double B[common * colsB];
  double C[rowsA * colsB]; double D[rowsA * colsB];

  char transA = 'N', transB = 'N';
  double one = 1.0, zero = 0.0;

  srand(time(NULL));

  init(A, rowsA, common); init(B, common, colsB);

  dgemm_(&transA, &transB, &rowsA, &colsB, &common, &one, A,
         &rowsA, B, &common, &zero, C, &rowsA);

  for(i=0;i<colsB;i++){
    for(j=0;j<rowsA;j++){
      D[i*rowsA+j]=0;
      for(k=0;k<common;k++){
        D[i*rowsA+j]+=A[k*rowsA+j]*B[k+common*i];
      }
    }
  }

  print("A", A, rowsA, common); print("B", B, common, colsB);
  print("C", C, rowsA, colsB); print("D", D, rowsA, colsB);

  return 0;
}
```

To compile the program for IBM zEnterprise EC12 (zEC12) models, enter:

```
xlC -c -march=zEC12 -DATL_ARCH_IBMzEC12 -o sample2.o sample2.C
```

To compile the program for IBM z13 models, enter:

```
xlC -c -march=z13 -o sample2.o sample2.C
```

, or

```
xlC -c -DATL_ARCH_IBMz13 -o sample2.o sample2.C
```

To link the program for IBM zEnterprise EC12 (zEC12) models in static single-threaded mode, enter:

```
xlC sample2.o -march=zEC12 -lgfortran -lf77blas.zEC12 -latlas.zEC12 -o sample2
```

To link the program for IBM z13 models in static single-threaded mode, enter:

```
xlC sample2.o -march=z13 -lgfortran -lf77blas.z13 -latlas.z13 -o sample2
```

Method 3

This program contains a C invocation of the CBLAS function `cblas_dgemm_` provided by the ATLAS framework.

Observation: This program has same result as if the `dgemm_` function were called.

`sample3.c` is as follows:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <cblas.h>

void init(double* matrix, int row, int column)
{
  for (int j = 0; j < column; j++){
    for (int i = 0; i < row; i++){
      matrix[j*row + i] = ((double)rand())/RAND_MAX;
    }
  }
}

void print(const char * name, const double* matrix, int row, int column)
{
  printf("Matrix %s has %d rows and %d columns:\n", name, row, column);
  for (int i = 0; i < row; i++){
    for (int j = 0; j < column; j++){
      printf("%.3f ", matrix[j*row + i]);
    }
    printf("\n");
  }
  printf("\n");
}

int main(int argc, char * argv[])
```

```
{
  int rowsA, colsB, common;
  int i,j,k;

  if (argc != 4){
    printf("Using defaults\n");
    rowsA = 2; colsB = 4; common = 6;
  }
  else{
    rowsA = atoi(argv[1]); colsB = atoi(argv[2]);common = atoi(argv[3]);
  }

  double A[rowsA * common]; double B[common * colsB];
  double C[rowsA * colsB]; double D[rowsA * colsB];

  enum CBLAS_ORDER order = CblasColMajor;
  enum CBLAS_TRANSPOSE transA = CblasNoTrans;
  enum CBLAS_TRANSPOSE transB = CblasNoTrans;

  double one = 1.0, zero = 0.0;

  srand(time(NULL));

  init(A, rowsA, common); init(B, common, colsB);

  cblas_dgemm(order,transA,transB, rowsA, colsB, common ,1.0,A,
              rowsA ,B, common ,0.0,C, rowsA);

  for(i=0;i<colsB;i++){
    for(j=0;j<rowsA;j++){
      D[i*rowsA+j]=0;
      for(k=0;k<common;k++){
        D[i*rowsA+j]+=A[k*rowsA+j]*B[k+common*i];
      }
    }
  }

  print("A", A, rowsA, common); print("B", B, common, colsB);
  print("C", C, rowsA, colsB); print("D", D, rowsA, colsB);

  return 0;
}
```

To compile the program for IBM zEnterprise EC12 (zEC12) models, enter:

```
xlc -c -march=zEC12 -DATL_ARCH_IBMzEC12 -o sample3.o sample3.c
```

To compile the program for IBM z13 models, enter:

```
xlc -c -march=z13 -o sample3.o sample3.c
```

, or

```
xlc -c -DATL_ARCH_IBMz13 -o sample3.o sample3.c
```

To link the program for IBM zEnterprise EC12 (zEC12) models in static single-threaded mode, enter:

```
xlc sample3.o -march=zEC12 -lcblas.zEC12 -latlas.zEC12 -o sample3
```

To link the program for IBM z13 models in static single-threaded mode, enter:

```
xlc sample3.o -march=z13 -lcblas.z13 -latlas.z13 -o sample3
```

Method 4

This program contains a C++ invocation of the CBLAS function `cblas_dgemm_` provided by the ATLAS framework.

Observation: This program has same result as if the dgemm_ function were called.

sample4.C is as follows:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

extern "C"
{
#include <cblas.h>
}

void init(double* matrix, int row, int column)
{
  for (int j = 0; j < column; j++){
    for (int i = 0; i < row; i++){
      matrix[j*row + i] = ((double)rand())/RAND_MAX;
    }
  }
}

void print(const char * name, const double* matrix, int row, int column)
{
  printf("Matrix %s has %d rows and %d columns:\n", name, row, column);
  for (int i = 0; i < row; i++){
    for (int j = 0; j < column; j++){
      printf("%.3f ", matrix[j*row + i]);
    }
    printf("\n");
  }
  printf("\n");
}

int main(int argc, char * argv[])
{
  int rowsA, colsB, common;
  int i,j,k;

  if (argc != 4){
    printf("Using defaults\n");
    rowsA = 2; colsB = 4; common = 6;
  }
  else{
    rowsA = atoi(argv[1]); colsB = atoi(argv[2]);common = atoi(argv[3]);
  }

  double A[rowsA * common]; double B[common * colsB];
  double C[rowsA * colsB]; double D[rowsA * colsB];

  enum CBLAS_ORDER order = CblasColMajor;
  enum CBLAS_TRANSPOSE transA = CblasNoTrans;
  enum CBLAS_TRANSPOSE transB = CblasNoTrans;

  double one = 1.0, zero = 0.0;

  srand(time(NULL));

  init(A, rowsA, common); init(B, common, colsB);

  cblas_dgemm(order,transA,transB, rowsA, colsB, common ,1.0,A,
              rowsA ,B, common ,0.0,C, rowsA);

  for(i=0;i<colsB;i++){
    for(j=0;j<rowsA;j++){
      D[i*rowsA+j]=0;
      for(k=0;k<common;k++){
        D[i*rowsA+j]+=A[k*rowsA+j]*B[k+common*i];
      }
    }
  }

  print("A", A, rowsA, common); print("B", B, common, colsB);
  print("C", C, rowsA, colsB); print("D", D, rowsA, colsB);

  return 0;
}
```

To compile the program for IBM zEnterprise EC12 (zEC12) models, enter:

```
xlC -c -march=zEC12 -DATL_ARCH_IBMzEC12 -o sample4.o sample4.C
```

To compile the program for IBM z13 models, enter:

```
xlC -c -march=z13 -o sample4.o sample4.C
```

, or

```
xlC -c -DATL_ARCH_IBMz13 -o sample4.o sample4.C
```

To link the program for IBM zEnterprise EC12 (zEC12) models in static single-threaded mode, enter:

```
xlC sample4.o -march=zEC12 -lcblas.zEC12 -latlas.zEC12 -o sample4
```

To link the program for IBM z13 models in static single-threaded mode, enter:

```
xlC sample4.o -march=z13 -lcblas.z13 -latlas.z13 -o sample4
```

Example 2: Compiling, linking, and running a complex ATLAS sample program

In this example, it is assumed that a complex test sample `invtst.c` is shipped with the ATLAS source code. This program combines ATLAS specific, CBLAS, and LAPACK functionality that must be compiled using the invocation commands for C programs.

Examples of the ATLAS specific functions that are called in this sample are `ATL_flushcache`, `ATL_assert`, `ATL_DivBySize`, and `ATL_MulBySize`.

Examples of the CBLAS specific functions that are called in this sample are `cblas_asum`, `cblas_scasum`, `cblas_dzasum`, `cblas_copy`, `cblas_gemm`, `cblas_symm`, and `cblas_hemm`.

The ATLAS header files that are used are `atlas_misc.h`, `atlas_lapack.h`, `cblas.h`, `atlas_cblastypealias.h`, `atlas_tst.h`, `atlas_level3.h`, and `clapack.h`.

To compile `invtst.c` for IBM zEnterprise EC12 (zEC12) models, enter:

```
xlc -c -march=zEC12 -DATL_ARCH_IBMzEC12 -o invtst.o -DL2SIZE=4194304 -DAdd_ \
-DF77_INTEGER=int -DStringSunStyle -DATL_NCPU=20 -DATLCINT -DSREAL \
-DWALL -DATL_CPUMHZ=5564 -DATL_OS_Linux invtst.c
```

To compile `invtst.c` for IBM z13 machine models, enter:

```
xlc -c -march=z13 -o invtst.o -DL2SIZE=4194304 -DAdd_ \
-DF77_INTEGER=int -DStringSunStyle -DATL_NCPU=20 -DATLCINT -DSREAL \
-DWALL -DATL_CPUMHZ=5564 -DATL_OS_Linux invtst.c
```

, or

```
xlc -c -DATL_ARCH_IBMz13 -o invtst.o -DL2SIZE=4194304 -DAdd_ \
-DF77_INTEGER=int -DStringSunStyle -DATL_NCPU=20 -DATLCINT -DSREAL \
-DWALL -DATL_CPUMHZ=5564 -DATL_OS_Linux invtst.c
```

where:

- L2SIZE represents the size of the L2 cache on the target hardware.
- Add_, F77_INTEGER, and StringSunStyle are Fortran macros that outline the inter-language interaction between C and Fortran code on Linux on z Systems.
- ATL_NCPU represents the number of CPUs on the target hardware.
- ATLCINT and SREAL are ATLAS specific macros.
- WALL instructs the ATLAS framework to issue all possible warnings.
- ATL_CPUMHZ represents the speed of the target architecture.
- ATL_OS_Linux instructs ATLAS that a Linux operating system is used for compile operations.

To link the program for IBM zEnterprise EC12 (zEC12) models in static single-threaded mode, enter:

```
xlc invtst.o -march=zEC12 -lgfortran -ltstatlas.zEC12 -llapack.zEC12 -lcblas.zEC12 \
-lf77blas.zEC12 -latlas.zEC12 -lm -o invtst
```

To link the program for IBM z13 models in static single-threaded mode, enter:

```
xlc invtst.o -march=z13 -lgfortran -ltstatlas.z13 -llapack.z13 -lcblas.z13 \
-lf77blas.z13 -latlas.z13 -lm -o invtst
```

When you run the executable, it produces the following output:

```
 NREPS  ORDER   UPLO      N    LDA      TIME     MFLOP        RESID
 =====  =====  =====  =====  =====  ========  ========  ============
   0     Col     GE    100    100     0.027     73.97  8.918092e-03
   0     Col     GE    200    200     0.005   3024.80  6.950735e-03
   0     Col     GE    300    300     0.015   3535.71  8.034554e-03
   0     Col     GE    400    400     0.034   3783.16  9.250009e-03
   0     Col     GE    500    500     0.063   3937.21  7.678587e-03
   0     Col     GE    600    600     0.107   4046.78  9.520883e-03
   0     Col     GE    700    700     0.167   4097.17  8.519278e-03
   0     Col     GE    800    800     0.247   4148.75  8.575264e-03
   0     Col     GE    900    900     0.346   4217.16  1.272196e-02
   0     Col     GE   1000   1000     0.471   4247.66  8.753754e-03
 10 cases: 10 passed, 0 skipped, 0 failed
```

Notices

Programming interfaces: Intended programming interfaces allow the customer to write programs to obtain the services of IBM XL C/C++ for Linux on z Systems.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those

Trademarks

Index

Product Number: 5725-N01

Printed in USA

SC27-5997-01