AltiVec™ Support In MrC[pp]

Revision 1.2
6/6/98

Table Of Contents

Revision History

Revision	Date	Comments
1.0	2/23/98	Initial document.
1.1	16/3/98	All references to VMX changed to AltiVec or vector. All naming conventions changed to use vec_. Rewrite of linkage conventions.
1.2	7/6/98	Fixed a couple of typeos in the Vector Data Types table.

1. Introduction

The PowerPC architecture has been extended to support a set of instructions referred to as the AltiVec™ vector instructions. As a result, MrC (for C) and MrCpp (for C++), referred to here collectively as "MrC[pp]", is extended to support the AltiVec architecture.[1] There are also vector extensions to various libraries and headers that are needed when building MrC[pp] programs that use AltiVec.

2. MrC[pp] AltiVec Compiler Extensions

The following briefly summarizes the areas that are extended or changed to support AltiVec in MrC[pp]. All of these will be described in more detail in subsequent sections.

- Command line option (`-vec[tor] on`) to enable the AltiVec language extensions.
- Predefined macro (`__VEC__`) to indicate the AltiVec extensions are enabled.
- Vector data types.
- Data alignment and dynamic allocation requirements for vector data types.
- Rules for using vector data types in expressions.
- Vector operators to generate the AltiVec instructions.
- Library and header support for AltiVec.
- Functions calls and linkage conventions to allow passing and storing of vector data types.

2.1 Specifying AltiVec on the Command Line

As discussed later, supporting AltiVec requires that the compiler use certain naming conventions which potentially could conflict with existing programs. Therefore the AltiVec extensions are not recognized unless `-vec[tor] on` is specified (thus the default is `-vector off`). The full command line syntax is,

`-vec[tor] on[,[no]vrsave]`	enable AltiVec extensions
`-vec[tor] off`	disable AltiVec extensions (default)

The additional `[no]vrsave` option controls whether function linkage conventions support the VRsave register. VRsave is a AltiVec SPR (special purpose register) used to inform the OS which vector registers need to be saved and reloaded across context switches (e.g., interrupts). The Macintosh system supports use of VRsave. However, the `novrsave` option is provided for contexts in which it is known that the VRsave register is not needed or not supported by the OS.[2]

2.2 Predefined Macros

When `-vector on` is specified on the command line, the compiler will predefine the macro `__VEC__`. `__VEC__` is predefined to have the decimal integer value following the format "vrrnn",

---

™ AltiVec is a registered trademark of Motorola, Inc.

1 The basis for these extensions is defined by the Motorola *AltiVec™ Programming Model* specification. Much of that specification has been incorporated into this document to tailor it specifically for MrC[pp].

2 In general the `noversave` option should never be used. It doesn't affect the environment if the register is maintained whether or not it is supported by the OS. Remember that even if an OS doesn't presently support the handling of VRsave today, it might in the future!

which corresponds to Programming Model version numbering scheme “v.rr.nn”.[3]

2.3 AltiVec Naming Conventions

When AltiVec is enabled all identifiers with the prefix “vec_” are reserved by the compiler for AltiVec extensions. There is nothing prohibiting the user from using identifiers starting with “vec_” in contexts other than what is described here, but this is not recommended.

2.4 Vector Data Types

AltiVec introduces 11 new reserved vector data type names as defined in the following table.

New C/C++ type	Size (bytes)	Interpretation of contents	Values
vector unsigned char	16	16 unsigned char	0...255
vector signed char	16	16 signed char	-128...127
vector bool char	16	16 unsigned char	0 (F), 255 (T)
vector unsigned short	16	8 unsigned short	0...65536
vector signed short	16	8 signed short	-32768...32767
vector bool short	16	8 unsigned short	0 (F), 65535 (T)
vector unsigned long	16	4 unsigned int	$0...2^{32} - 1$
vector signed long	16	4 signed int	$-2^{31} ... 2^{31} - 1$
vector bool long	16	4 unsigned int	0 (F), $2^{32} - 1$ (T)
vector float	16	4 float	IEEE-754 values
vector pixel	16	8 unsigned short	1/5/5/5 pixel

Vector Data Types

In addition to the 11 data types defined above, the type specifier `int` may be combined with `short` or `long` (e.g., `vector unsigned short int`). When multiple simple type specifiers are allowed, they can be freely intermixed in any order. However, the `vector` type specifier must occur first.

Note that although the identifiers `vector` and `pixel` occur as part of the vector data types, they are *not* considered as reserved words except in when used as type specifiers. Similarly `bool` is not treated as a reserved keyword in C except in this context. In C++ however it will be treated as a reserved keyword if the command line option -bool on is specified (which also will then treat `true` and `false` as reserved keywords).

Two reserved keywords are provided as aliases to `vector` and `pixel` . They are `__vector` and `__pixel` respectively. These may always be used in either C or C++.

In this document, the term “vec_data” is defined to mean data that can be any of the above vector data types and “vec_type” is used to represent any of the vector data types.

---

[3] For example, if the current version of the Motorola AltiVec™ *Programming Model* specification is 1.2.1 then `__VEC__` is defined to have the decimal value 10201.

Defining the vector data types as fundamental types permits the compiler to provide stronger type checking and support overloaded operations on these types.

As will be discussed later, all vector operations take the form of overloaded function calls. These overloaded functions are allowed in *both* C and C++. In addition, when vector types appear in C++ member functions, the name mangling rules for function signatures have been extended to support the vector types. See **Appendix C** for further details on C++ vector name type mangling.

2.5 Alignment

A defined data item of any vector data type must always aligned in memory on a 16-byte boundary. A pointer to any vector data type always points to a 16-byte boundary. The compiler is responsible for aligning vector data types on 16-byte boundaries. Given that vector data must be correctly aligned, a program is incorrect if it attempts to dereference a pointer to a vector type if the pointer does not contain a 16-byte aligned address. Note that in the AltiVec architecture an unaligned load/store does not cause an alignment exception. Instead, the low-order bits of the address are quietly ignored.

2.5.1 Alignment of non-vector data

An array of components to be loaded into vector registers need not be aligned, but will have to be accessed with attention to its alignment. Typically, this will be accomplished with the `vec_lvsr()`, `vec_lvsl()`, and `vec_perm()` instructions.

2.5.2 Alignment of vector Data within structures and classes

Structures or classes containing vector types are aligned on 16-byte boundaries and their internal organization padded, if necessary, so that each internal vector type is aligned on a 16-byte boundary regardless of the alignment mode (set via `#pragma align` or `-align` command line option) currently in effect.[4]

2.5.3 Dynamic allocation and alignment

Dynamically allocated space for vector data must be aligned on a 16-byte boundary. There are four ways space is dynamically allocated, two of which are explicitly under user control.

- Space dynamically allocated on the stack for local data allocated by the compiler.
- Space dynamically allocated on the stack by the user.
- Space dynamically allocated on the heap by the user.
- Space dynamically allocated for C++ class objects.

2.5.3.1 Dynamic alignment of compiler allocated local data

In order to guarantee that vector local data is aligned on a 16-byte boundary, the compiler must generate function entry (prolog) code that ensures the function's local data is 16-byte aligned. The additional code generated by the prolog (and function exit epilog) to ensure alignment is only generated if needed, i.e., when a function contains vector locals or has vector parameters that may themselves be on the stack. See section 4 for further details.

---

4 Padding may also occur to align data inherited from parent classes that themselves contain vector data.

2.5.3.2 Space dynamically allocated on the stack by the user (alloca)

When `-alloca` is specified on the command line then the predefined `alloca()` function may be used in a function to dynamically allocate space on the stack. The code generated for `alloca()` will always allocate a multiple of 16 bytes and *always* align the space on a 16-byte boundary on the stack. Therefore the allocated space can be used for whatever purpose, including space for vector data.

2.5.3.3 Space dynamically allocated on the heap by the user (vec_malloc)

Unlike `alloca()`, the standard `malloc()` may be heavily used, many times to allocate relatively small objects. Thus generalizing `malloc()`, `calloc()`, and `realloc()` to always have a multiple-of-16 overhead with 16-byte alignment is not desirable. A different set of variants, called `vec_malloc()`, `vec_calloc()`, and `vec_realloc()` are provided as part of StdCLib and defined in stdlib.h. It is the user's responsibility to use `vec_malloc()`, etc. when the intended use for the allocated space is to contain vector data. In order to free space allocated by these allocators the routine `vec_free()` (also defined in stdlib.h) must be called.

2.5.3.4 Space dynamically allocated for class objects

When the default `operator new` is invoked for a class that contains vector data (either explicitly or implicitly through inheritance) a routine named `vec_new()` is called instead of invoking the `operator new` runtime support routine. Similarly, when the default `operator delete` is called, the compiler substitutes a call to `vec_delete()`. `vec_new()` is implemented by calling `vec_malloc()` and `vec_delete()` calls `vec_free()`.

If an explicit `operator new` (including the placement form of `operator new`) or `operator delete` is declared as a member function, then the user takes responsibility for the allocation. Therefore such implementations must take into account vector alignment if required by calling `vec_new()` (or `vec_malloc()`) and `vec_delete()` (or `vec_free()`) as appropriate.[5]

2.6 Expressions

Most C/C++ operators do not permit any of their arguments to be a vector data type. The normal C/C++ operators are extended to include the operations defined in the following sections.

In the examples in the following sections let `a` and `b` be vector types and `p` be a pointer to a vector type.

2.6.1 sizeof()

`sizeof(a)` and `sizeof(*p)` return 16.

2.6.2 Assignment

If either the left hand side or right hand side of an expression has a vector type, then both sides of the expression must be of the same vector type. Thus, the expression `a = b` is valid and represents assignment only if `a` and `b` are of the same vector type (or if neither is a vector type). Otherwise, the expression is invalid and is reported as an error by the compiler.

---

[5] Internally there are four library routines to support allocation and deallocation of C++ classes: `vec_new()` and `vec_delete()` as discussed above, `__vec_vec_new()` and `__vec_vec_delete()` for arrays of objects (but the latter calls are only generated by the compiler). Like `operator new`, `vec_new()` is defined in new.h.

2.6.3 Address Operator

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

2.6.4 Pointer Arithmetic

The usual pointer arithmetic can be performed on `p`. In particular, `p+1` is a pointer to the next vector element after `p`.

2.6.5 Pointer Dereferencing

If `p` is a pointer to a vector type, `*p` implies either a 128-bit vector load from the address obtained by clearing the low order bits of `p` (equivalent to the instruction `vec_ld(0,p)`) or a 128-bit vector store to that address (equivalent to the instruction `vec_st(0,p)`). If it is desired to mark the data accessed as least-recently-used (LRU), the explicit instruction `vec_ldl(0,p)` or `vec_stl(0,p)` must be used.

Dereferencing a pointer to a non-vector type produces the standard behavior of either a load or a copy of the corresponding type.

Accessing of non-aligned memory must be carried out explicitly by a `vec_ld(int, type *)` operation, a `vec_ldl(int, type *)` operation, a `vec_st(int, type *)` operation or a `vec_stl(int, type *)` operation.

2.6.6 Type Casting

Pointers to non-vector and vector data may be cast back and forth to each other. Casting a pointer to a vector type represents an (unchecked) assertion that the address is 16-byte aligned.

Casts from one vector type to another are provided using the usual C syntax `(vec_type)e`, (e.g., `(vector unsigned char)e`). In all cases the data represented by `e` is converted to the specified vector type without changing the bit pattern.

2.6.7 Vector Constants

Vector constants may used wherever a vector data value is allowed (static/dynamic initialization, parameters, assignments). The compiler generates code which either computes or loads the values into an AltiVec register. They have the following forms:

`(vector unsigned char)(unsigned int)`
`(vector unsigned char)(unsigned int`$_1$`,...,unsigned int`$_{16}$`)`

Represents a vector unsigned char constant consisting of a set of 16 unsigned 8-bit quantities which all have the value specified by a single unsigned integer or as individually specified by 16 unsigned integers.

`(vector signed char)(int)`
`(vector signed char)(int`$_1$`,...,int`$_{16}$`)`

Represents a vector signed char constant consisting of a set of 16 signed 8-bit quantities which all have the value specified by a single integer or as individually specified by 16 integers.

`(vector unsigned short)(unsigned int)`
`(vector unsigned short)(unsigned int`$_1$`,...,unsigned int`$_8$`)`

Represents a vector unsigned short constant consisting of a set of 8 unsigned 16-bit quantities which all have the value specified by a single unsigned integer or as individually specified by

8 unsigned integers.

`(vector signed char)(int)`
`(vector signed char)(int`$_1$`,...,int`$_8$`)`

Represents a vector signed char constant consisting of a set of 8 signed 16-bit quantities which all have the value specified by a single integer or as individually specified by 8 integers.

`(vector unsigned long)(unsigned int)`
`(vector unsigned long)(unsigned int`$_1$`,...,unsigned int`$_4$`)`

Represents a vector unsigned long constant consisting of a set of 4 unsigned 32-bit quantities which all have the value specified by a single unsigned integer or as individually specified by 4 unsigned integers.

`(vector signed long)(int)`
`(vector signed long)(int`$_1$`,...,int`$_4$`)`

Represents a vector signed long constant consisting of a set of 4 signed 32-bit quantities which all have the value specified by a single integer or as individually specified by 4 integers.

`(vector float)(float)`
`(vector float)(float`$_1$`,...,float`$_4$`)`

Represents a vector float constant consisting of a set of 4 32-bit floating-point quantities which all have the value specified by a single float value or as individually specified by 4 float values.

In all of these constants the individual (unsigned) integer(s) or float value(s) may be constant expressions.

2.6.8 Value for adjusting pointers

Given a pointer to a type that is one of the possible vector components, `vec_step(vec_data)` or `vec_step(vec_type)` produces at compile time the integer value to be added to the pointer to cause the pointer to be incremented by 16 bytes. For example, a vector unsigned short data type is considered to contain 8 unsigned 2-byte values. A pointer to unsigned 2-byte values used to stream through an array of unsigned 2-byte values by a full vector at a time should be incremented by `vec_step(vector unsigned short)` which generates the constant 8.

```
vec_step(vector unsigned char)  = 16
vec_step(vector signed char)    = 16
vec_step(vector boolean char)   = 16
vec_step(vector unsigned short) = 8
vec_step(vector signed short)   = 8
vec_step(vector boolean short)  = 8
vec_step(vector unsigned long)  = 4
vec_step(vector signed long)    = 4
vec_step(vector boolean long)   = 4
vec_step(vector float)          = 4
vec_step(vector pixel)          = 8
```

2.7 Operators representing AltiVec operations

The vector operators allow full access to the functionality provided by the AltiVec architecture. The operators are represented in the programming language by language structures which have

function call syntax. The names associated with these operations are all prefixed with "vec_". The appearance of one of these forms can indicate:

- A *generic* (overloaded) AltiVec operation (e.g., `vec_add()`) which generates a vector instruction depending on the argument types.
- A *specific* AltiVec operation (e.g., `vec_addubm()`) which maps directly into a AltiVec machine instruction.
- A predicate (0 or 1) computed from a AltiVec operation (e.g., `vec_all_eq()`).
- A cast, like `(vector signed char)e`, as already discussed in Section 2.6.6.
- Loading of a vector of constant components, as already discussed in section 2.6.7.

Each operator representing a AltiVec operation takes a list of arguments representing the input operands in the order in which they appear in the tables in **Appendix A** and **Appendix B** and returns a result (possibly void).

The permitted operand types for each AltiVec operation, whether specific or generic, are restricted to those in the tables. The programmer may override this constraint by explicitly casting arguments to permissible types.

For a specific operation, the operand types are used to determine whether the operation is acceptable and to determine the type of the result. For example, `vec_addubm(vector signed char, vector signed char)` is acceptable because that represents a reasonable way to do modular addition with signed bytes, while `vec_addubs(vector signed char, vector signed char)` and `addubh(vector signed char, vector signed char)` are not acceptable. The former operation would produce a result in which saturation treated the operands as unsigned, while the latter would produce a result in which adjacent pairs of signed bytes would be treated as signed half words.

For a generic operation, the operand types are used to determine whether the operation is acceptable, to select a particular operation according to the types of the arguments, and to determine the type of the result. For example, `vec_add(vector signed char, vector signed char)` will map onto `vec_addubm()` and return a result of type `vector signed char`, while `vec_add(vector unsigned short, vector unsigned short)` will map onto `vec_adduhm()` and return a result of type `vector unsigned short`.

The AltiVec operations which set condition register CR6 (the "compare dot" instructions) are treated somewhat differently. The programmer does not have access to specific register names. Instead of directly specifying a compare dot instruction, the programmer makes reference to a predicate which returns an integer value derived from the result of a compare dot instruction. As in C, this value may be used directly as a value (1 is true, 0 is false) or as a condition for branching. The predicates all begin with "`vec_all_`" or "`vec_any_`". There are predicates to test the true or false state of any bit which can be set by a compare dot instruction. For example, `vec_all_gt(x,y)` tests the true value of bit 24 of the CR after executing some `vcmpgt.` instruction. To complete the coverage by predicates, additional predicates exercise compare dot instructions with reversed or duplicated arguments. As examples, `vec_all_lt(x,y)` performs a `vcmpgtx.(y,x)`, and `vec_all_nan(x)` is mapped onto `vcmpeqfp.(x,x)`. If the programmer wishes to have both the result of the compare dot instruction as returned in the vector register and the value of CR6, the programmer must specify two instructions.

The tables of permitted generic instructions are documented in **Appendix A**.

The tables of permitted predicates are documented in **Appendix B**.

3. Library and Header Support for AltiVec

The following areas are extended to supported AltiVec:

- Extensions to standard I/O formatting for the vector data types
- Extensions to headers
- Extensions to PPCCRuntime.o
- Extensions to MrCExceptionsLib
- Extensions to StdCLib

3.1 Extensions to Standard I/O Formatting of the Vector Data Types

The conversion specifications in standard I/O output statements (scanf, fprintf, etc.) are extended to support the vector data types. The specifications are described in the following sections; first the forms for output (printf, etc.) and then those for input (scanf, etc.).

3.1.1 Output conversions specifications for printf, etc.

All the output functions that have a format string as one of their arguments (`fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`, `vsprintf`) support vector output conversions that have the following general form:

```
%[<flags>][width][<precision>][<size>]<conversion>
```

where,

```
<flags>          ::= <std-flags> |
                     <c-sep> |
                     [<std-flags>]<num-sep>[<std-flags>]
<std-flags>      ::= <std-flags-char> | <std-flags><std-flags-char>
<std-flags-char> ::= '-' | '+' | '0' | '#' | ' '
<c-sep>          ::= any character including <std-flags-char> except
                     <width> | <precision> | <size> | <conversion>
<num-sep>        ::= any character except <std-flags> | <width> |
                     <precision> | <size> | <conversion>
<width>          ::= <integer> | '*'
<precision>      ::= '.' '*' | '.' [<integer>]
<size>           ::= 'll' | 'L' | 'l' | 'h' | <vector-size>
<vector-size>    ::= 'vl' | 'vh' | 'lv' | 'hv' | 'v'
<conversion>     ::= <char-conv> | <str-conv> | <fp-conv> | <int-conv> |
                     <misc-conv>
<char-conv>      ::= 'c'
<str-conv>       ::= 's' | 'P'
<fp-conv>        ::= 'e' | 'E' | 'f' | 'g' | 'G'
<int-conv>       ::= 'd' | 'i' | 'u' | 'o' | 'p' | 'x' | 'X'
<misc-conv>      ::= 'n' | '%'
```

The extensions to the output conversion specification for vector types are shown in **bold**.

The `<vector-size>` indicates that a single vector value is to be converted. The vector value is displayed in the following general format:

$value_1$ C $value_2$ C $value_3$ C $value_4$ C ... C $value_n$

where C is a separator character defined by the <flags> (<num-sep> or <c-sep>). There are 4, 8, or 16 output values depending on the <vector-size>, each formatted according to the <conversion>.

A <vector-size> of 'vl' or 'lv' consumes one argument and modifies the <int-conv> conversion; it should be of type vector signed long, vector unsigned long, or vector bool long; it is treated as a series of four 4-byte components. A <vector-size> of 'vh' or 'hv' consumes one argument and modifies the <int-conv> conversion; it should be of type vector signed short, vector unsigned short, vector bool short, or vector pixel; it is treated as a series of eight 2-byte components. A <vector-size> of 'v' with <int-conv> or <char-conv> consumes one argument; it should be of type vector signed char, vector unsigned char, or vector bool char; it is treated as a series of sixteen 1-byte components. A <vector-size> of 'v' with <fp-conv> consumes one argument; it should be of type vector float; it is treated as a series of four 4-byte floating-point components. All other combinations of <vector-size> and <conversion> are undefined.

The default value for the separator character is a space unless 'c' conversion is being used. For 'c' conversion the default is to have no separator. Also for 'c' conversion, any of the standard numeric flags characters ('-', '+', '#', ' ') may be used as a separator since these flags are not otherwise used. For numeric conversions the standard flags apply to the conversions and thus may not be specified as a separator flag. Also, only one separator character may be specified in the <flags>.

Examples:

Given the following declarations:

```
vector signed char s8 = (vector signed char)(1,  2,  3,  4,  5,  6,  7,  8,
                                             9, 10, 11, 12, 13, 14, 15, 16);
vector unsigned short u16 = (vector unsigned short)('a', 'b', 'c', 'd',
                                                    'e', 'f', 'g', 'h');
vector signed long s32 = (vector signed long)(1, 2, 3, 12);
vector float f32 = (vector float)(1.1, 2.2, 3.3, 4.4);
```

The following printf statements produce the indicated output:

```
printf("s8  = %vd", s8);         s8  = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
printf("s8  = %,vd", s8);        s8  = 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
printf("u16 = %vhc", u16);       u16 = abcdefgh
printf("s32 = %,2lvd", s32);     s32 =  1, 2, 3,12
printf("f32 = %,5.2vf", f32);    f32 =  1.10, 2.20, 3.30, 4.40
```

3.1.2 Input conversions specifications for scanf, etc.

All the input functions that have a format string as one of their arguments (fscanf, scanf, sscanf) support vector input conversions that have the following general form:

```
%[<flags>][width][<size>]<conversion>
```

where,

```
<flags>          ::=  '*' | <c-sep> ['*'] | ['*'] <c-sep>
<c-sep>          ::=  any character except '*' | <width> | <size> |
                      <conversion>
```

```
<width>          ::= <integer>
<size>           ::= 'll' | 'L' | 'l' | 'h' | <vector-size>
<vector-size>    ::= 'vl' | 'vh' | 'lv' | 'lh' |'v'
<conversion>     ::= <char-conv> | <str-conv> | <fp-conv> | <int-conv> |
                     <misc-conv>
<char-conv>      ::= 'c'
<str-conv>       ::= 's' | 'P' | '[' ['^'] <any characters> ']'
<fp-conv>        ::= 'e' | 'f' | 'g'
<int-conv>       ::= 'd' | 'i' | 'u' | 'o' | 'p' | 'x'
<misc-conv>      ::= 'n' | '%'
```

The extensions to the input conversion specification for vector types are shown in **bold**.

The <vector-size> indicates that a single vector value is to be scanned and converted. The vector data to be scanned is expected to have the following general format:

$$value_1\ C\ value_2\ C\ value_3\ C\ value_4\ C\ ...\ C\ value_n$$

where C is a separator character defined by the <flags> (<c-sep> surrounded by any number of spaces). The number of scanned values is 4, 8, or 16 depending on the <vector-size> with each value scanned according to the <conversion>.

A <vector-size> of 'vl' or 'lv' consumes one argument and modifies the <int-conv> conversion; it should be of type vector signed long * or vector unsigned long * depending on the <int-conv> specification; 4 values are scanned. A <vector-size> of 'vh' or 'hv' consumes one argument and modifies the <int-conv> conversion; it should be of type vector signed * or vector unsigned short * depending on the <int-conv> specification; 8 values are scanned. A <vector-size> of 'v' with <int-conv> or <char-conv> consumes one argument; it should be of type vector signed char * or vector unsigned char * depending on the <int-conv> or <char-conv> specification; 16 values are scanned. A <vector-size> of 'v' with <fp-conv> consumes one argument; it should be of type vector float *; 4 floating-point values are scanned. All other combinations of <vector-size> and <conversion> are undefined.

The default value for the separator character is any number of space unless 'c' conversion is being used. For 'c' conversion the default is to have no separator character.

Examples:

These are equivalent to,

```
sscanf("1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16", "%vd", &s8);
sscanf("1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16", "%,vd", &s8);
sscanf("abcdefgh", "%vhc", &u16);
sscanf("1, 2, 3,12", "%,2lvd", &s32);
sscanf("1.10, 2.20, 3.30, 4.40", "%,5vf", &f32);
```

These set the vector variables as if they were declared as follows:

```
vector signed char s8 = (vector signed char)(1,  2,  3,  4,  5,  6,  7,  8,
                                             9, 10, 11, 12, 13, 14, 15, 16);
vector unsigned short u16 = (vector unsigned short)('a', 'b', 'c', 'd',
                                                    'e', 'f', 'g', 'h');
vector signed long s32 = (vector signed long)(1, 2, 3, 12);
vector float f32 = (vector float)(1.1, 2.2, 3.3, 4.4);
```

3.2 Extensions to the Headers

Four headers are modified to support AltiVec; new.h, stdarg.h, stdlib.h, and setjmp.h. In all these headers, the extensions are only in effect if `-vector on` was specified on the command line. In other words they are all under `#if __VEC__` conditional which is only true when the AltiVec extensions are enabled.

3.2.1 new.h

The C++ header new.h declares `vec_new()`. This is referenced by the compiler when an `operator new` is seen for a class that contains or inherits vector data.

3.2.2 stdarg.h

As discussed under Linkage Conventions section 4, all vector arguments must be aligned on a 16-byte boundary. This requires special treatment when handling a variable argument list using stdarg.h where the argument list can be a mixture of non-vector data (aligned on 4-byte boundaries) and vector data (on 16-byte boundaries).

When not using AltiVec all arguments are aligned on a 4-byte boundary. Thus `va_arg()` is defined to round up the `va_list` pointer using `sizeof(type)` to the next 4-byte boundary. This is not sufficient with AltiVec and its 16-byte alignment requirement. A more complicated adjustment (but still done at compile time) must either round up to the next 4-byte or 16-byte boundary. A compile-time predefined function called `__va_align__(type)` returns a constant 4 or 16 depending on whether its type argument is a non-vector type or a vector type respectively. `va_arg()` is defined in terms of both `__va_align__()` and `sizeof()` in order to correctly handle variable argument lists with vector-typed arguments.

3.2.3 stdlib.h

As discussed in section 2.5.3.3 and section 3.5, `vec_malloc()`, `vec_calloc()` and `vec_realloc()` are provided to ensure 16-byte alignment for dynamically allocated vector data. To free space allocated by these routines, `vec_free()` must be used. The definitions for these routines are defined in stdlib.h. Except for the alignment requirement their behavior and arguments are identical to their non-vector counterparts.

3.2.4 setjmp.h

The definition for `jmp_buf` in setjmp.h must be different when supporting AltiVec in order to save the pertinent AltiVec registers in addition to those saved when AltiVec is not being used. The setjmp.h header defines the larger `jmp_buf` and *redefines* `setjmp()` and `longjmp()` to call an alternate set of library routines in StdCLib that expect this larger jmp_buf. These are called `__vec_setjmp()` and `__vec_longjmp()`.

Since it is the setjmp.h header that determines which form of `jmp_buf` to use and which `setjmp()` and `longjmp()` routines to call, it is up to the user to be consistent with their use. Thus if a `longjmp()` is done from a compilation unit that doesn't otherwise use the AltiVec extensions, it still must specify `-vector on` on the command line in order for the proper `longjmp()` call to be generated. The converse is also true, i.e., doing a `setjmp()` from a compilation unit that doesn't otherwise use the AltiVec extensions to define a `jmp_buf` used by a compilation unit that does. Both compilation units need to be compiled with `-vector on`.

3.3 Extensions to PPCCRuntime.o

When `-opt size` is specified on the compiler command line, any non-leaf functions save their

volatile floating-point registers with the aid of routines supplied in PPCCRuntime.o. Calling a single routine to save registers saves space (hence why it's only used under the `-opt size` option[6]). These routines all have names following the same forms: _savef*N* to save and _restf*N* to restore floating-point registers, where *N* is a number 14 to 31. For example, calling _savef25 will cause floating-point registers fp25 through fp31 to be saved while calling _restf25 will restore them. These calls are generated by the compiler and should never be called by the user.

For AltiVec registers, a similar set of routines are provided in PPCCRuntime.o to save and restore the volatile vector registers. These have the general name _savev*N* and _restv*N*, where *N* is 20 through 31 (see section 4 for a discussion of linkage conventions and the non-volatile vector registers). As with floating-point, these routines are compiler-generated for non-leaf functions compiled with `-opt size` and should not to be called by the user.

3.4 Extensions to MrCExceptionsLib

When exceptions are used in MrCpp `-exceptions on` must be specified on the command line and the program linked with MrCExceptionsLib. Any compilation unit that is also compiled with `-vector on` will invoke a set of different runtime exceptions support routines in MrCExceptionsLib.[7] If both exceptions and AltiVec are used in the program then *all* compilations units should be built with `-vector on` whether or not a specific set of compilation units uses AltiVec. This is necessary to properly restore the vector registers and the VRsave SPR as the stack is unwound from a throw to the appropriate catch clause.

3.5 Extensions to StdCLib

As discussed in section 2.5.3.3, `vec_malloc()`, `vec_calloc()`, `vec_realloc()`, and `vec_free()` are provided to dynamically allocate 16-byte aligned space for vector data. These are located in StdCLib.

AltiVec support for `setjmp()` and `longjmp()`, i.e., the routines `__vec_setjmp()` and `__vec_longjmp()`, are also located in StdCLib.

4. Functions Calls and Linkage Conventions

AltiVec support imposes some additional semantic rules on function calls and their declarations or definitions. There are also differences in the linkage conventions to support the handling of the vector registers, stack frame layout and alignment, and the VRsave special purpose register.

Note that the AltiVec intrinsic operations are not treated as function calls, so these comments do not apply to those operations.

4.1 Additional Function Call Semantics

Any forward reference to a function which includes vector parameters requires a prototype. Vector types as parameters or as a return type are *not* allowed for DTSOM member functions.

4.2 Linkage Conventions

The following sections discuss the modifications to linkage conventions.

---

6 The non-leaf requirement is because if a call was done the function wouldn't be a leaf any longer.

7 Because the vector registers and the VRsave SPR must be restored when a (re)throw is done (using the routines `__vec__eh_throw()` and `__vec__eh_rethrow()`) and space for thrown objects must be allocated or deallocated using 16-byte alignment.

4.2.1 Register Usage Conventions

The register usage conventions for the vector register file are defined as follows:

Registers	Intended use	Behavior across call sites
v0-v1	General use	Volatile (Caller save)
v2-v13	Parameters, general	Volatile (Caller save)
v14-v19	General	Volatile (Caller save)
v20-v31	General	Non-volatile (Callee save)
VRsave	Special, see below	Non-volatile (Callee save)

AltiVec Register Usage Conventions

The special purpose register (SPR) number 256, named `VRsave`, is used to inform the operating system which vector registers need to be saved and reloaded across context switches. Bit `31-`*n* of this register is set to 1 if vector register `v`*n* needs to be saved and restored across a context switch. Otherwise, the operating system may return that register with any value that does not violate security after a context switch. The most significant bit in the 32-bit word is considered to be bit 0.

4.2.2 Function calls with a fixed number of arguments

The first twelve parameters of any non-struct vector data type are placed in consecutive vector registers v2 through v13. Any additional vector-typed parameters are passed through memory on the stack. They appear together, 16-byte aligned, and after any non-vector parameters. If fewer (or no) vector type arguments are passed, the unneeded registers are not loaded and will contain undefined values on entry to the called function.

Non-vector parameters are passed in the same registers as they would be if the vector parameters were not present. Structs that contain vector fields are treated the same as any other struct except that they are 16-byte aligned. This can result in words in the parameter list being skipped for alignment (padding) and left with undefined value.

Vector parameters are *not* shadowed in GPR's. They are not placed in memory unless there are more than 12 vector arguments.

Functions that declare a vector data type as a return value place that return value in register v2.

4.2.3 Function calls with a variable number of arguments

Arguments lists for a function defined with a variable number of arguments are passed differently than those with a fixed number of arguments. *All* arguments are passed in the order specified with vector arguments 16-byte aligned and non-vector arguments 4-byte aligned. All the arguments are put on the stack in the parameter area with the first 8 words shadowed in the GPR's including any "holes" created for alignment.

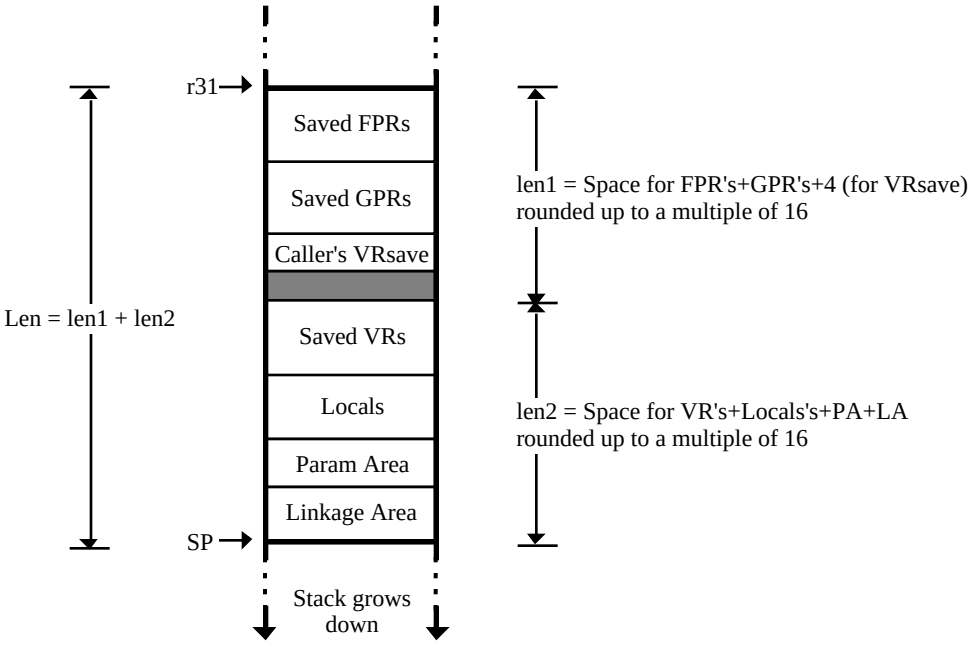
4.3 The Stack Frame

A stack frame for a function having vector local data or using vector registers requires a vector register save area, the VRsave save word, and the alignment padding space to dynamically align the stack to a 16-byte boundary.

The following additional requirements apply to a vector stack frame:

- Before a function changes the value of `VRsave`, it must save the value of `VRsave` at the time of entry to the function in the VRsave save word.
- The alignment padding space is either 0, 4, 8, or 12 bytes long to make the address of the vector register save area (and subsequent stack locations) be 16-byte aligned.
- The code establishing the stack frame dynamically aligns the stack pointer atomically with an `stwux` instruction. The code *always* assumes the stack pointer on entry is aligned on an 8-byte boundary.
- Before a function changes the value in any non-volatile vector register, v*n*, it saves the value in v*n*, in the word in the vector register save area 16*(32-*n*) bytes before the low-addressed end of the alignment padding space.
- Local variables of a vector data type which need to be saved to memory are placed on the stack frame on a 16-byte alignment boundary in the same stack frame region used for local variables of other types.

SP in the figures denotes the stack pointer (general purpose register `r1`) of the called function after it has executed code establishing its stack frame.



Vector Stack Frame Layout

Non-volatile floating point registers (FPR's) and general purpose registers (GPR's) are saved in the frame in the usual way. But when there are non-volatile vector registers (VR's) to be saved or vector locals, then the frame needs additional space for those registers and the caller's VRsave. The prolog code needs to dynamically 16-byte align the frame thus producing a "hole" (illustrated by the shaded area in the above diagram). These is discussed in more detail in the following sections.

4.3.1 Stack Frame Alignment

All vector stack frames must assume that the caller's SP is only 8-byte aligned and therefore each callee is responsible for 16-byte aligning it's own frame. The compiler computes the size of the locals such that it plus the sizes of the parameter area (PA) and linkage areas (LA) will also come out to be a multiple of 16 (len2 in the above diagram). Thus the frame will start on a 16-byte boundary if the area for the saved vector registers also start on a 16-byte boundary.

The upper size of the alignment hole is computed at compile time. It is the minimum number of bytes needed to needed to make the space for the saved FPR's+GPR's+4 (for VRsave) a multiple of 16 (i.e., 4, 8, or 12, len1 in the diagram is the size of this rounded up space). The multiple-of-16 space for the locals+PA+LA (len2) can be dynamically aligned by "sliding" the space (represented by len2) "up" or "down" using the statically computed hole to take up the slack.

The computations for computing the callee's SP are as follows:

- If the static hole is 8 or 12 it is big enough to allow moving the len2 space "up" by 0 or 8 bytes to get that space 16-byte aligned.

  callee's SP = caller's SP - Len + (caller's SP & 8)

- If the static hole is 0 or 4 then it isn't big enough to allow moving the len2 space "up". Therefore it must be moved "down" by 0 or 8 bytes to get it 16-byte aligned.

  callee's SP = caller's SP - Len - (caller's SP & 8)

Because the alignment is dynamic an additional register, r31, must be reserved to allow accessing of the caller's parameters. It will always point to the callee's stack frame (i.e., it is a copy of the caller's SP).[8]

R31 is not always reserved as the caller's frame pointer. If it turns out that r31 is the *only* GPR that needs saving and there is a volatile register between r3 and r10 available, then one of those volatiles will be used in place of r31.

4.3.2 Saving the vector registers (VR's)

If any non-volatile VR's need to be saved on the stack they are saved immediately after the alignment hole. Debuggers can always find these registers if they know the number of saved GPR's and FPR's and whether VRsave is saved or not.

4.3.3 VRsave

VRsave is the AltiVec SPR (256) used to inform the OS which vector registers need to be saved and reloaded across context switches (e.g., interrupts). Bit *i* of this register is set to 1 if vector register *i* needs to be saved and restored across a context switch (the most significant bit in the 32-bit word is considered to correspond to v0). Otherwise, the operating system may return vector register *i* with any value that does not violate security after a context switch.

---

[8] Normally r31 is used as the callee's original frame pointer when `alloca()` is used in order to access locals and the caller's parameters. But as just discussed, when the stack is vector aligned, r31 is used as the caller's frame pointer. Then r30 becomes `alloca()`'s original callee frame pointer. Note that it does not matter whether the stack is vector aligned or not for the space allocated by `alloca()` since it always allocates its stack space on a 16-byte boundary.

When a process is launched, VRsave is set to 0. As functions are called the prolog is responsible for saving the caller's VRsave and OR'ing into VRsave all the bits that correspond to the vector registers (volatile and non-volatile) used by that function. On exit, the epilog code restores the caller's VRsave.

Because VRsave is stored in a fixed place in the stack frame, debuggers can access it if they need to.

4.3.4 Local Variables

Vector locals are 16-byte aligned within the local area and mixed in with all the other locals used by the function. This may incur some wasted space within the local area. The entire local space is also rounded up to a multiple of 16 so that it plus the parameter and linkage area space are a multiple of 16.

Appendix A: Generic and Specific AltiVec Operators

The tables are organized alphabetically by generic operation name and define the permitted generic and specific AltiVec operations. Each table describes a single generic AltiVec operation. Each line shows a valid set of argument types for that generic AltiVec operation, the result type for that set of argument types, and the specific AltiVec instruction generated for that set of arguments. For example, `vec_add(vector unsigned char, vector unsigned char)` maps to "`vaddubm`".

In some tables a Note column is shown. If there is no Note column it is permissible to use a specific AltiVec operator formed by prefixing "`vec_`" to the name of the operation in the Maps To column with that line's set of argument types. For example, `vec_vaddubm(vector unsigned char, vector unsigned char)` has the same effect as `vec_add(vector unsigned char, vector unsigned char)`.

In the few cases in which a Note column is shown, it will have a "N" to indicate that the specific AltiVec instruction is not permitted for that generic operation because that set of argument types has been chosen to produce a different result type.

Any operation which is not explicitly permitted by these tables is prohibited and will cause a compilation error. Casts may be used, if necessary, to use operators in bizarre ways.

A.1 vec_add(arg1, arg2)

Each element of the result is the sum of the corresponding elements of **arg1** and **arg2**. The arithmetic is modular for integer types.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vaddubm
vector unsigned char	vector unsigned char	vector bool char	vaddubm
vector unsigned char	vector bool char	vector unsigned char	vaddubm
vector signed char	vector signed char	vector signed char	vaddubm
vector signed char	vector signed char	vector bool char	vaddubm
vector signed char	vector bool char	vector signed char	vaddubm
vector unsigned short	vector unsigned short	vector unsigned short	vadduhm
vector unsigned short	vector unsigned short	vector bool short	vadduhm
vector unsigned short	vector bool short	vector unsigned short	vadduhm
vector signed short	vector signed short	vector signed short	vadduhm
vector signed short	vector signed short	vector bool short	vadduhm
vector signed short	vector bool short	vector signed short	vadduhm
vector unsigned long	vector unsigned long	vector unsigned long	vadduwm
vector unsigned long	vector unsigned long	vector bool long	vadduwm
vector unsigned long	vector bool long	vector unsigned long	vadduwm
vector signed long	vector signed long	vector signed long	vadduwm
vector signed long	vector signed long	vector bool long	vadduwm
vector signed long	vector bool long	vector signed long	vadduwm
vector float	vector float	vector float	vaddfp

A.2 vec_addc(arg1, arg2)

Each element of the result is the carry produced by adding the corresponding elements of **arg1** and **arg2**. A carry gives a value of 1; no carry gives a value of 0.

Result	arg1	arg2	Maps To
vector unsigned long	vector unsigned long	vector unsigned long	vaddcuw

A.3 vec_adds(arg1, arg2)

Each element of the result is the saturated sum of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vaddubs
vector unsigned char	vector unsigned char	vector bool char	vaddubs
vector unsigned char	vector bool char	vector unsigned char	vaddubs
vector signed char	vector signed char	vector signed char	vaddsbs
vector signed char	vector signed char	vector bool char	vaddsbs
vector signed char	vector bool char	vector signed char	vaddsbs
vector unsigned short	vector unsigned short	vector unsigned short	vadduhs
vector unsigned short	vector unsigned short	vector bool short	vadduhs
vector unsigned short	vector bool short	vector unsigned short	vadduhs
vector signed short	vector signed short	vector signed short	vaddshs
vector signed short	vector signed short	vector bool short	vaddshs
vector signed short	vector bool short	vector signed short	vaddshs
vector unsigned long	vector unsigned long	vector unsigned long	vadduws
vector unsigned long	vector unsigned long	vector bool long	vadduws
vector unsigned long	vector bool long	vector unsigned long	vadduws
vector signed long	vector signed long	vector signed long	vaddsws
vector signed long	vector signed long	vector bool long	vaddsws
vector signed long	vector bool long	vector signed long	vaddsws

A.4 vec_and(arg1, arg2)

Each element of the result is the logical AND of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vand
vector unsigned char	vector unsigned char	vector bool char	vand
vector unsigned char	vector bool char	vector unsigned char	vand
vector signed char	vector signed char	vector signed char	vand
vector signed char	vector signed char	vector bool char	vand
vector signed char	vector bool char	vector signed char	vand
vector bool char	vector bool char	vector bool char	vand
vector unsigned short	vector unsigned short	vector unsigned short	vand
vector unsigned short	vector unsigned short	vector bool short	vand
vector unsigned short	vector bool short	vector unsigned short	vand
vector signed short	vector signed short	vector signed short	vand
vector signed short	vector signed short	vector bool short	vand
vector signed short	vector bool short	vector signed short	vand
vector bool short	vector bool short	vector bool short	vand

vector unsigned long	vector unsigned long	vector unsigned long	vand
vector unsigned long	vector unsigned long	vector bool long	vand
vector unsigned long	vector bool long	vector unsigned long	vand
vector signed long	vector signed long	vector signed long	vand
vector signed long	vector signed long	vector bool long	vand
vector signed long	vector bool long	vector signed long	vand
vector bool long	vector bool long	vector bool long	vand
vector float	vector bool long	vector float	vand
vector float	vector float	vector bool long	vand
vector float	vector float	vector float	vand

A.5 vec_andc(arg1, arg2)

Each element of the result is the logical AND of the corresponding element of **arg1** and the one's complement of the corresponding element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vandc
vector unsigned char	vector unsigned char	vector bool char	vandc
vector unsigned char	vector bool char	vector unsigned char	vandc
vector signed char	vector signed char	vector signed char	vandc
vector signed char	vector signed char	vector bool char	vandc
vector signed char	vector bool char	vector signed char	vandc
vector bool char	vector bool char	vector bool char	vandc
vector unsigned short	vector unsigned short	vector unsigned short	vandc
vector unsigned short	vector unsigned short	vector bool short	vandc
vector unsigned short	vector bool short	vector unsigned short	vandc
vector signed short	vector signed short	vector signed short	vandc
vector signed short	vector signed short	vector bool short	vandc
vector signed short	vector bool short	vector signed short	vandc
vector bool short	vector bool short	vector bool short	vandc
vector unsigned long	vector unsigned long	vector unsigned long	vandc
vector unsigned long	vector unsigned long	vector bool long	vandc
vector unsigned long	vector bool long	vector unsigned long	vandc
vector signed long	vector signed long	vector signed long	vandc
vector signed long	vector signed long	vector bool long	vandc
vector signed long	vector bool long	vector signed long	vandc
vector bool long	vector bool long	vector bool long	vandc
vector float	vector bool long	vector float	vandc
vector float	vector float	vector bool long	vandc
vector float	vector float	vector float	vandc

A.6 vec_avg(arg1, arg2)

Each element of the result is the average of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vavgub
vector signed char	vector signed char	vector signed char	vavgsb
vector unsigned short	vector unsigned short	vector unsigned short	vavguh

vector signed short	vector signed short	vector signed short	vavgsh
vector unsigned long	vector unsigned long	vector unsigned long	vavguw
vector signed long	vector signed long	vector signed long	vavgsw

A.7 vec_ceil(arg1)

Each element of the result is the largest representable floating point integer not less than the corresponding element of **arg1**.

Result	arg1	Maps To
vector float	vector float	vrfip

A.8 vec_cmpb(arg1, arg2)

Each element of the result is 0 if the corresponding element of **arg1** is greater than or equal to the negative of the corresponding element of **arg2** and less than or equal to the corresponding element of **arg2**. If the corresponding element of **arg2** is not negative, each element of the result will be negative if the corresponding element of **arg1** is greater than the corresponding element of **arg2** and positive if the corresponding element of **arg1** is less than the negative of the corresponding element of **arg1**.

Result	arg1	arg2	Maps To
vector signed long	vector float	vector float	vcmpbfp

A.9 vec_cmpeq(arg1, arg2)

Each element of the result is TRUE if the corresponding element of **arg1** is equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
vector bool char	vector unsigned char	vector unsigned char	vcmpequb
vector bool char	vector signed char	vector signed char	vcmpequb
vector bool short	vector unsigned short	vector unsigned short	vcmpequh
vector bool short	vector signed short	vector signed short	vcmpequh
vector bool long	vector unsigned long	vector unsigned long	vcmpequw
vector bool long	vector signed long	vector signed long	vcmpequw
vector bool long	vector float	vector float	vcmpeqfp

A.10 vec_cmpge(arg1, arg2)

Each element of the result is TRUE if the corresponding element of **arg1** is greater than or equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
vector bool long	vector float	vector float	vcmpgefp

A.11 vec_cmpgt(arg1, arg2)

Each element of the result is TRUE if the corresponding element of **arg1** is greater than the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
vector bool char	vector unsigned char	vector unsigned char	vcmpgtub
vector bool char	vector signed char	vector signed char	vcmpgtsb
vector bool short	vector unsigned short	vector unsigned short	vcmpgtuh
vector bool short	vector signed short	vector signed short	vcmpgtsh
vector bool long	vector unsigned long	vector unsigned long	vcmpgtuw
vector bool long	vector signed long	vector signed long	vcmpgtsw
vector bool long	vector float	vector float	vcmpgtfp

A.12 vec_ctf(arg1, arg2)

Each element of the result is the closest floating-point representation of the number obtained by dividing the corresponding element of **arg1** by 2 to the power of **arg2**.

Result	arg1	arg2	Maps To
vector float	vector unsigned long	5-bit unsigned literal	vcfux
vector float	vector signed long	5-bit unsigned literal	vcfsx

A.13 vec_cts(arg1, arg2)

Each element of the result is the saturated signed value obtained after truncating the number obtained by multiplying the corresponding element of **arg1** by 2 to the power of **arg2**.

Result	arg1	arg2	Maps To
vector signed long	vector float	5-bit unsigned literal	vctsxs

A.14 vec_ctu(arg1, arg2)

Each element of the result is the saturated unsigned value obtained after truncating the number obtained by multiplying the corresponding element of **arg1** by 2 to the power of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned long	vector float	immed_u5	vctuxs

A.15 vec_dss(arg1)

Each operation stops cache touches for the data stream associated with tag **arg1**.

Result	arg1	Maps To
void	2-bit unsigned literal	dss

A.16 vec_dssall(arg1)

The operation stops cache touches for all data streams.

Result	arg1	Maps To
void	void	dssall

A.17 vec_dst(arg1, arg2, arg3)

Each operation initiates cache touches for loads for the data stream associated with tag **arg3** at the address **arg1** using the data block in **arg2**. The **arg1** may also be a pointer to a const-qualified type.

Result	arg1	arg2	arg3	Maps To
void	vector unsigned char *	int	2-bit unsigned literal	dst
void	vector signed char *	int	2-bit unsigned literal	dst
void	vector bool char *	int	2-bit unsigned literal	dst
void	vector unsigned short *	int	2-bit unsigned literal	dst
void	vector signed short *	int	2-bit unsigned literal	dst
void	vector bool short *	int	2-bit unsigned literal	dst
void	vector pixel *	int	2-bit unsigned literal	dst
void	vector unsigned long *	int	2-bit unsigned literal	dst
void	vector signed long *	int	2-bit unsigned literal	dst
void	vector bool long *	int	2-bit unsigned literal	dst
void	vector float *	int	2-bit unsigned literal	dst
void	unsigned char *	int	2-bit unsigned literal	dst
void	signed char *	int	2-bit unsigned literal	dst
void	unsigned short *	int	2-bit unsigned literal	dst
void	short *	int	2-bit unsigned literal	dst
void	unsigned int *	int	2-bit unsigned literal	dst
void	int *	int	2-bit unsigned literal	dst
void	unsigned long *	int	2-bit unsigned literal	dst
void	long *	int	2-bit unsigned literal	dst
void	float *	int	2-bit unsigned literal	dst

A.18 vec_dstst(arg1, arg2, arg3)

Each operation initiates cache touches for stores for the data stream associated with tag **arg3** at the address **arg1** using the data block in **arg2**. The **arg1** may also be a pointer to a const-qualified type.

Result	arg1	arg2	arg3	Maps To
void	vector unsigned char *	int	2-bit unsigned literal	dstst
void	vector signed char *	int	2-bit unsigned literal	dstst
void	vector bool char *	int	2-bit unsigned literal	dstst
void	vector unsigned short *	int	2-bit unsigned literal	dstst
void	vector signed short *	int	2-bit unsigned literal	dstst
void	vector bool short *	int	2-bit unsigned literal	dstst
void	vector pixel *	int	2-bit unsigned literal	dstst
void	vector unsigned long *	int	2-bit unsigned literal	dstst
void	vector signed long *	int	2-bit unsigned literal	dstst

void	vector bool long *	int	2-bit unsigned literal	dstst
void	vector float *	int	2-bit unsigned literal	dstst
void	unsigned char *	int	2-bit unsigned literal	dstst
void	signed char *	int	2-bit unsigned literal	dstst
void	unsigned short *	int	2-bit unsigned literal	dstst
void	short *	int	2-bit unsigned literal	dstst
void	unsigned int *	int	2-bit unsigned literal	dstst
void	int *	int	2-bit unsigned literal	dstst
void	unsigned long *	int	2-bit unsigned literal	dstst
void	long *	int	2-bit unsigned literal	dstst
void	float *	int	2-bit unsigned literal	dstst

A.19 vec_dststt(arg1, arg2, arg3)

Each operation initiates cache touches for transient stores for the data stream associated with tag **arg3** at the address **arg1** using the data block in **arg2**. The **arg1** may also be a pointer to a const-qualified type.

Result	arg1	arg2	arg3	Maps To
void	vector unsigned char *	int	2-bit unsigned literal	dststt
void	vector signed char *	int	2-bit unsigned literal	dststt
void	vector bool char *	int	2-bit unsigned literal	dststt
void	vector unsigned short *	int	2-bit unsigned literal	dststt
void	vector signed short *	int	2-bit unsigned literal	dststt
void	vector bool short *	int	2-bit unsigned literal	dststt
void	vector pixel *	int	2-bit unsigned literal	dststt
void	vector unsigned long *	int	2-bit unsigned literal	dststt
void	vector signed long *	int	2-bit unsigned literal	dststt
void	vector bool long *	int	2-bit unsigned literal	dststt
void	vector float *	int	2-bit unsigned literal	dststt
void	unsigned char *	int	2-bit unsigned literal	dststt
void	signed char *	int	2-bit unsigned literal	dststt
void	unsigned short *	int	2-bit unsigned literal	dststt
void	short *	int	2-bit unsigned literal	dststt
void	unsigned int *	int	2-bit unsigned literal	dststt
void	int *	int	2-bit unsigned literal	dststt
void	unsigned long *	int	2-bit unsigned literal	dststt
void	long *	int	2-bit unsigned literal	dststt
void	float *	int	2-bit unsigned literal	dststt

A.20 vec_dstt(arg1, arg2, arg3)

Each operation initiates cache touches for transient loads for the data stream associated with tag **arg3** at the address **arg1** using the data block in **arg2**. The **arg1** may also be a pointer to a const-qualified type.

Result	arg1	arg2	arg3	Maps To
void	vector unsigned char *	int	2-bit unsigned literal	dstt
void	vector signed char *	int	2-bit unsigned literal	dstt
void	vector bool char *	int	2-bit unsigned literal	dstt

void	vector unsigned short *	int	2-bit unsigned literal	dstt
void	vector signed short *	int	2-bit unsigned literal	dstt
void	vector bool short *	int	2-bit unsigned literal	dstt
void	vector pixel *	int	2-bit unsigned literal	dstt
void	vector unsigned long *	int	2-bit unsigned literal	dstt
void	vector signed long *	int	2-bit unsigned literal	dstt
void	vector bool long *	int	2-bit unsigned literal	dstt
void	vector float *	int	2-bit unsigned literal	dstt
void	unsigned char *	int	2-bit unsigned literal	dstt
void	signed char *	int	2-bit unsigned literal	dstt
void	unsigned short *	int	2-bit unsigned literal	dstt
void	short *	int	2-bit unsigned literal	dstt
void	unsigned int *	int	2-bit unsigned literal	dstt
void	int *	int	2-bit unsigned literal	dstt
void	unsigned long *	int	2-bit unsigned literal	dstt
void	long *	int	2-bit unsigned literal	dstt
void	float *	int	2-bit unsigned literal	dstt

A.21 vec_expte(arg1)

Each element of the result is an estimate of 2 raised to the corresponding element of **arg1**.

Result	arg1	Maps To
vector float	vector float	vexptefp

A.22 vec_floor(arg1)

Each element of the result is the largest representable floating point integer not greater than **arg1**.

Result	arg1	Maps To
vector float	vector float	vrfim

A.23 vec_ld(arg1, arg2)

Each operation performs a 16-byte load at a 16-byte aligned address. **arg1** is taken to be an integer value, while **arg2** is a pointer. Note that the sum of **arg1** and **arg2** is truncated, if necessary, to give 16-byte alignment; loading unaligned data into a vector register typically requires a permutation of the results of two loads. Note that this load is the one which will be generated for a loading dereference of a pointer to a vector type. The **arg2** may also be a pointer to a const-qualified type.

Result	arg1	arg2	Maps To
vector unsigned char	int	vector unsigned char *	lvx
vector unsigned char	int	unsigned char *	lvx
vector signed char	int	vector signed char *	lvx
vector signed char	int	signed char *	lvx
vector bool char	int	vector bool char *	lvx
vector unsigned short	int	vector unsigned short *	lvx
vector unsigned short	int	unsigned short *	lvx
vector signed short	int	vector signed short *	lvx

vector signed short	int	short *	lvx
vector bool short	int	vector bool short *	lvx
vector pixel	int	vector pixel *	lvx
vector unsigned long	int	vector unsigned long *	lvx
vector unsigned long	int	unsigned int *	lvx
vector unsigned long	int	unsigned long *	lvx
vector signed long	int	vector signed long *	lvx
vector signed long	int	int *	lvx
vector signed long	int	long *	lvx
vector bool long	int	vector bool long *	lvx
vector float	int	vector float *	lvx
vector float	int	float *	lvx

A.24 vec_lde(arg1, arg2)

Each operation loads a single element into the position in the vector register corresponding to its address, leaving the remaining elements of the register undefined. **arg1** is taken to be an integer value, while **arg2** is a pointer. The **arg2** may also be a pointer to a const-qualified type.

Result	arg1	arg2	Maps To
vector unsigned char	int	unsigned char *	lvebx
vector signed char	int	signed char *	lvebx
vector unsigned short	int	unsigned short *	lvehx
vector signed short	int	short *	lvehx
vector unsigned long	int	unsigned int *	lvewx
vector unsigned long	int	unsigned long *	lvewx
vector signed long	int	int *	lvewx
vector signed long	int	long *	lvewx
vector float	int	float *	lvewx

A.25 vec_ldl(arg1, arg2)

Each operation performs a 16-byte load at a 16-byte aligned address. **arg1** is taken to be an integer value, while **arg2** is a pointer. Note that the sum of **arg1** and **arg2** is truncated, if necessary, to give 16-byte alignment; loading unaligned data into a vector register typically requires a permutation of the results of two loads. These operations mark the cache line as least-recently-used. The **arg2** may also be a pointer to a const-qualified type.

Result	arg1	arg2	Maps To
vector unsigned char	int	vector unsigned char *	lvxl
vector unsigned char	int	unsigned char *	lvxl
vector signed char	int	vector signed char *	lvxl
vector signed char	int	signed char *	lvxl
vector bool char	int	vector bool char *	lvxl
vector unsigned short	int	vector unsigned short *	lvxl
vector unsigned short	int	unsigned short *	lvxl
vector signed short	int	vector signed short *	lvxl
vector signed short	int	short *	lvxl
vector bool short	int	vector bool short *	lvxl
vector pixel	int	vector pixel *	lvxl

vector unsigned long	int	vector unsigned long *	lvxl
vector unsigned long	int	unsigned int *	lvxl
vector unsigned long	int	unsigned long *	lvxl
vector signed long	int	vector signed long *	lvxl
vector signed long	int	int *	lvxl
vector signed long	int	long *	lvxl
vector bool long	int	vector bool long *	lvxl
vector float	int	vector float *	lvxl
vector float	int	float *	lvxl

A.26 vec_loge(arg1)

Each element of the result is an estimate of the logarithm to base 2 of the corresponding element of **arg1**.

Result	arg1	Maps To
vector float	vector float	vlogefp

A.27 vec_lvsl(arg1, arg2)

Each operation generates a permutations useful for aligning data from an unaligned address. The **arg2** may also be a pointer to a const or volatile qualified type.

Result	arg1	arg2	Maps To
vector unsigned char	int	unsigned char *	lvsl
vector unsigned char	int	signed char *	lvsl
vector unsigned char	int	unsigned short *	lvsl
vector unsigned char	int	short *	lvsl
vector unsigned char	int	unsigned int *	lvsl
vector unsigned char	int	unsigned long *	lvsl
vector unsigned char	int	int *	lvsl
vector unsigned char	int	long *	lvsl
vector unsigned char	int	float *	lvsl

A.28 vec_lvsr(arg1, arg2)

Each operation generates a permutations useful for aligning data from an unaligned address. The **arg2** may also be a pointer to a const or volatile qualified type.

Result	arg1	arg2	Maps To
vector unsigned char	int	unsigned char *	lvsr
vector unsigned char	int	signed char *	lvsr
vector unsigned char	int	unsigned short *	lvsr
vector unsigned char	int	short *	lvsr
vector unsigned char	int	unsigned int *	lvsr
vector unsigned char	int	unsigned long *	lvsr
vector unsigned char	int	int *	lvsr
vector unsigned char	int	long *	lvsr
vector unsigned char	int	float *	lvsr

A.29 vec_madd(arg1, arg2, arg3)

Each element of the result is the sum of the corresponding element of **arg3** and the product of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	arg3	Maps To
vector float	vector float	vector float	vector float	vmaddfp

A.30 vec_madds(arg1, arg2, arg3)

Each element of the result is the 16-bit saturated sum of the corresponding element of **arg3** and the high-order 17 bits of the product of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	arg3	Maps To
vector signed short	vector signed short	vector signed short	vector signed short	vmhaddshs

A.31 vec_max(arg1, arg2)

Each element of the result is the larger of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vmaxub
vector unsigned char	vector unsigned char	vector bool char	vmaxub
vector unsigned char	vector bool char	vector unsigned char	vmaxub
vector signed char	vector signed char	vector signed char	vmaxsb
vector signed char	vector signed char	vector bool char	vmaxsb
vector signed char	vector bool char	vector signed char	vmaxsb
vector unsigned short	vector unsigned short	vector unsigned short	vmaxuh
vector unsigned short	vector unsigned short	vector bool short	vmaxuh
vector unsigned short	vector bool short	vector unsigned short	vmaxuh
vector signed short	vector signed short	vector signed short	vmaxsh
vector signed short	vector signed short	vector bool short	vmaxsh
vector signed short	vector bool short	vector signed short	vmaxsh
vector unsigned long	vector unsigned long	vector unsigned long	vmaxuw
vector unsigned long	vector unsigned long	vector bool long	vmaxuw
vector unsigned long	vector bool long	vector unsigned long	vmaxuw
vector signed long	vector signed long	vector signed long	vmaxsw
vector signed long	vector signed long	vector bool long	vmaxsw
vector signed long	vector bool long	vector signed long	vmaxsw
vector float	vector float	vector float	vmaxfp

A.32 vec_mergeh(arg1, arg2)

The even elements of the result are obtained left-to-right from the high elements of **arg1.** The odd elements of the result are obtained left-to-right from the high elements of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vmrghb
vector signed char	vector signed char	vector signed char	vmrghb
vector bool char	vector bool char	vector bool char	vmrghb
vector unsigned short	vector unsigned short	vector unsigned short	vmrghh

vector signed short	vector signed short	vector signed short	vmrghh
vector bool short	vector bool short	vector bool short	vmrghh
vector pixel	vector pixel	vector pixel	vmrghh
vector unsigned long	vector unsigned long	vector unsigned long	vmrghw
vector signed long	vector signed long	vector signed long	vmrghw
vector bool long	vector bool long	vector bool long	vmrghw
vector float	vector float	vector float	vmrghw

A.33 vec_mergel(arg1, arg2)

The even elements of the result are obtained left-to-right from the low elements of **arg1.** The odd elements of the result are obtained left-to-right from the low elements of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vmrglb
vector signed char	vector signed char	vector signed char	vmrglb
vector bool char	vector bool char	vector bool char	vmrglb
vector unsigned short	vector unsigned short	vector unsigned short	vmrglh
vector signed short	vector signed short	vector signed short	vmrglh
vector bool short	vector bool short	vector bool short	vmrglh
vector pixel	vector pixel	vector pixel	vmrglh
vector unsigned long	vector unsigned long	vector unsigned long	vmrglw
vector signed long	vector signed long	vector signed long	vmrglw
vector bool long	vector bool long	vector bool long	vmrglw
vector float	vector float	vector float	vmrglw

A.34 vec_mfvscr(void)

The first six elements of the result are 0. The seventh element of the result contains the high-order 16 bits of the VSCR (including NJ). The eighth element of the result contains the low-order 16 bits of the VSCR (including SAT).

Result	Maps To
vector unsigned short	mfvscr

A.35 vec_min(arg1, arg2)

Each element of the result is the smaller of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vminub
vector unsigned char	vector unsigned char	vector bool char	vminub
vector unsigned char	vector bool char	vector unsigned char	vminub
vector signed char	vector signed char	vector signed char	vminsb
vector signed char	vector signed char	vector bool char	vminsb
vector signed char	vector bool char	vector signed char	vminsb
vector unsigned short	vector unsigned short	vector unsigned short	vminuh
vector unsigned short	vector unsigned short	vector bool short	vminuh
vector unsigned short	vector bool short	vector unsigned short	vminuh
vector signed short	vector signed short	vector signed short	vminsh

vector signed short	vector signed short	vector bool short	vminsh
vector signed short	vector bool short	vector signed short	vminsh
vector unsigned long	vector unsigned long	vector unsigned long	vminuw
vector unsigned long	vector unsigned long	vector bool long	vminuw
vector unsigned long	vector bool long	vector unsigned long	vminuw
vector signed long	vector signed long	vector signed long	vminsw
vector signed long	vector signed long	vector bool long	vminsw
vector signed long	vector bool long	vector signed long	vminsw
vector float	vector float	vector float	vminfp

A.36 vec_mladd(arg1, arg2, arg3)

Each element of the result is the low-order 16 bits of the sum of the corresponding element of **arg3** and the product of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	arg3	Maps To
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	vmladduhm
vector signed short	vector unsigned short	vector signed short	vector signed short	vmladduhm
vector signed short	vector signed short	vector unsigned short	vector unsigned short	vmladduhm
vector signed short	vector signed short	vector signed short	vector signed short	vmladduhm

A.37 vec_mradds(arg1, arg2, arg3)

Each element of the result is the 16-bit saturated sum of the corresponding element of **arg3** and the high-order 17 bits of the rounded product of the corresponding elements of **arg1** and **arg2**. Note that **arg2** is unsigned, while **arg1** is signed for the variant which maps to vmsumbm.

Result	arg1	arg2	arg3	Maps To
vector signed short	vector signed short	vector signed short	vector signed short	vmhraddshs

A.38 vec_msum(arg1, arg2, arg3)

Each element of the result is the sum of the corresponding element of **arg3** and the products of the elements of **arg1** and **arg2** which overlap the positions of that element of **arg3**. The sum is performed with 32-bit modular addition.

Result	arg1	arg2	arg3	Maps To
vector unsigned long	vector unsigned char	vector unsigned char	vector unsigned long	vmsumubm
vector unsigned long	vector unsigned short	vector unsigned short	vector unsigned long	vmsumuhm
vector signed long	vector unsigned char	vector signed char	vector signed long	vmsummbm
vector signed long	vector signed short	vector signed short	vector signed long	vmsumshm

A.39 vec_msums(arg1, arg2, arg3)

Each element of the result is the sum of the corresponding element of **arg3** and the products of the elements of **arg1** and **arg2** which overlap the positions of that element of **arg3**. The sum is performed with 32-bit saturating addition.

Result	arg1	arg2	arg3	Maps To
vector unsigned long	vector unsigned short	vector unsigned short	vector unsigned long	vmsumuhs
vector signed long	vector signed short	vector signed short	vector signed long	vmsumshs

A.40 vec_mtvscr(arg1)

The VSCR is set by the elements in **arg1** which occupy the last 32 bits.

Result	arg1	Maps To
void	vector unsigned char	mtvscr
void	vector signed char	mtvscr
void	vector bool char	mtvscr
void	vector unsigned short	mtvscr
void	vector signed short	mtvscr
void	vector bool short	mtvscr
void	vector pixel	mtvscr
void	vector unsigned long	mtvscr
void	vector signed long	mtvscr
void	vector bool long	mtvscr

A.41 vec_mule(arg1, arg2)

Each element of the result is the product of the corresponding high half-width elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned short	vector unsigned char	vector unsigned char	vmuleub
vector signed short	vector signed char	vector signed char	vmulesb
vector unsigned long	vector unsigned short	vector unsigned short	vmuleuh
vector signed long	vector signed short	vector signed short	vmulesh

A.42 vec_mulo(arg1, arg2)

Each element of the result is the product of the corresponding low half-width elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned short	vector unsigned char	vector unsigned char	vmuloub
vector signed short	vector signed char	vector signed char	vmulosb
vector unsigned long	vector unsigned short	vector unsigned short	vmulouh
vector signed long	vector signed short	vector signed short	vmulosh

A.43 vec_nmsub(arg1, arg2, arg3)

Each element of the result is the negative of the difference of the corresponding element of **arg3** and the product of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	arg3	Maps To
vector float	vector float	vector float	vector float	vnmsubfp

A.44 vec_nor(arg1, arg2)

Each element of the result is the logical NOR of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vnor
vector signed char	vector signed char	vector signed char	vnor
vector bool char	vector bool char	vector bool char	vnor
vector unsigned short	vector unsigned short	vector unsigned short	vnor
vector signed short	vector signed short	vector signed short	vnor
vector bool short	vector bool short	vector bool short	vnor
vector unsigned long	vector unsigned long	vector unsigned long	vnor
vector signed long	vector signed long	vector signed long	vnor
vector bool long	vector bool long	vector bool long	vnor
vector float	vector float	vector float	vnor

A.45 vec_or(arg1, arg2)

Each element of the result is the logical OR of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vor
vector unsigned char	vector unsigned char	vector bool char	vor
vector unsigned char	vector bool char	vector unsigned char	vor
vector signed char	vector signed char	vector signed char	vor
vector signed char	vector signed char	vector bool char	vor
vector signed char	vector bool char	vector signed char	vor
vector bool char	vector bool char	vector bool char	vor
vector unsigned short	vector unsigned short	vector unsigned short	vor
vector unsigned short	vector unsigned short	vector bool short	vor
vector unsigned short	vector bool short	vector unsigned short	vor
vector signed short	vector signed short	vector signed short	vor
vector signed short	vector signed short	vector bool short	vor
vector signed short	vector bool short	vector signed short	vor
vector bool short	vector bool short	vector bool short	vor
vector unsigned long	vector unsigned long	vector unsigned long	vor
vector unsigned long	vector unsigned long	vector bool long	vor
vector unsigned long	vector bool long	vector unsigned long	vor
vector signed long	vector signed long	vector signed long	vor
vector signed long	vector signed long	vector bool long	vor
vector signed long	vector bool long	vector signed long	vor
vector bool long	vector bool long	vector bool long	vor
vector float	vector bool long	vector float	vor
vector float	vector float	vector bool long	vor
vector float	vector float	vector float	vor

A.46 vec_pack(arg1, arg2)

Each high element of the result is the truncation of the corresponding wider element of **arg1**. Each low element of the result is the truncation of the corresponding wider element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned short	vector unsigned short	vpkuhum
vector signed char	vector signed short	vector signed short	vpkuhum
vector bool char	vector bool short	vector bool short	vpkuhum
vector unsigned short	vector unsigned long	vector unsigned long	vpkuwum
vector signed short	vector signed long	vector signed long	vpkuwum
vector bool short	vector bool long	vector bool long	vpkuwum

A.47 vec_packpx(arg1, arg2)

Each high element of the result is the packed pixel from the corresponding wider element of **arg1**. Each low element of the result is the packed pixel from the corresponding wider element of **arg2**.

Result	arg1	arg2	Maps To
vector pixel	vector unsigned long	vector unsigned long	vpkpx

A.48 vec_packs(arg1, arg2)

Each high element of the result is the saturated value of the corresponding wider element of **arg1**. Each low element of the result is the saturated value of the corresponding wider element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned short	vector unsigned short	vpkuhus
vector signed char	vector signed short	vector signed short	vpkshss
vector unsigned short	vector unsigned long	vector unsigned long	vpkuwus
vector signed short	vector signed long	vector signed long	vpkswss

A.49 vec_packsu(arg1, arg2)

Each high element of the result is the saturated value of the corresponding wider element of **arg1**. Each low element of the result is the saturated value of the corresponding wider element of **arg2**. The result elements are all unsigned. Note, it is necessary to use the generic name for the two variants with specific operations `vec_vpkuhus` and `vec_vpkuwus` since these are used for two variants of the `vec_packs` generic operation.

Result	arg1	arg2	Maps To	Note
vector unsigned char	vector unsigned short	vector unsigned short	vpkuhus	N
vector unsigned char	vector signed short	vector signed short	vpkshus	
vector unsigned short	vector unsigned long	vector unsigned long	vpkuwus	N
vector unsigned short	vector signed long	vector signed long	vpkswus	

A.50 vec_perm(arg1, arg2, arg3)

Each element of the result is selected independently by indexing the catenated bytes of **arg1** and **arg2** by the corresponding element of **arg3**.

Result	arg1	arg2	arg3	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	vperm
vector signed char	vector signed char	vector signed char	vector unsigned char	vperm
vector bool char	vector bool char	vector bool char	vector unsigned char	vperm
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned char	vperm
vector signed short	vector signed short	vector signed short	vector unsigned char	vperm
vector bool short	vector bool short	vector bool short	vector unsigned char	vperm
vector pixel	vector pixel	vector pixel	vector unsigned char	vperm
vector unsigned long	vector unsigned long	vector unsigned long	vector unsigned char	vperm
vector signed long	vector signed long	vector signed long	vector unsigned char	vperm
vector bool long	vector bool long	vector bool long	vector unsigned char	vperm
vector float	vector float	vector float	vector unsigned char	vperm

A.51 vec_re(arg1)

Each element of the result is an estimate of the reciprocal the corresponding element of **arg1**.

Result	arg1	Maps To
vector float	vector float	vrefp

A.52 vec_rl(arg1, arg2)

Each element of the result is the result of rotating left the corresponding element of **arg1** by the number of bits in the corresponding element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vrlb
vector signed char	vector signed char	vector unsigned char	vrlb
vector unsigned short	vector unsigned short	vector unsigned short	vrlh
vector signed short	vector signed short	vector unsigned short	vrlh
vector unsigned long	vector unsigned long	vector unsigned long	vrlw
vector signed long	vector signed long	vector unsigned long	vrlw

A.53 vec_round(arg1)

Each element of the result is the nearest representable floating point integer to **arg1**, using IEEE round-to-nearest rounding.

Result	arg1	Maps To
vector float	vector float	vrfin

A.54 vec_rsqrte(arg1)

Each element of the result is an estimate of the reciprocal square root of the corresponding element of **arg1**.

Result	arg1	Maps To
vector float	vector float	vrsqrtefp

A.55 vec_sel(arg1, arg2, arg3)

Each bit of the result is the corresponding bit of **arg1** if the corresponding bit of **arg3** is 0. Otherwise, it is the corresponding bit of **arg2**.

Result	arg1	arg2	arg3	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	vsel
vector unsigned char	vector unsigned char	vector unsigned char	vector bool char	vsel
vector signed char	vector signed char	vector signed char	vector unsigned char	vsel
vector signed char	vector signed char	vector signed char	vector bool char	vsel
vector bool char	vector bool char	vector bool char	vector unsigned char	vsel
vector bool char	vector bool char	vector bool char	vector bool char	vsel
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	vsel
vector unsigned short	vector unsigned short	vector unsigned short	vector bool short	vsel
vector signed short	vector signed short	vector signed short	vector unsigned short	vsel
vector signed short	vector signed short	vector signed short	vector bool short	vsel
vector bool short	vector bool short	vector bool short	vector unsigned short	vsel
vector bool short	vector bool short	vector bool short	vector bool short	vsel
vector unsigned long	vector unsigned long	vector unsigned long	vector unsigned long	vsel
vector unsigned long	vector unsigned long	vector unsigned long	vector bool long	vsel
vector signed long	vector signed long	vector signed long	vector unsigned long	vsel
vector signed long	vector signed long	vector signed long	vector bool long	vsel
vector bool long	vector bool long	vector bool long	vector unsigned long	vsel
vector bool long	vector bool long	vector bool long	vector bool long	vsel
vector float	vector float	vector float	vector unsigned long	vsel
vector float	vector float	vector float	vector bool long	vsel

A.56 vec_sl(arg1, arg2)

Each element of the result is the result of shifting the corresponding element of **arg1** left by the number of bits of the corresponding element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vslb
vector signed char	vector signed char	vector unsigned char	vslb
vector unsigned short	vector unsigned short	vector unsigned short	vslh
vector signed short	vector signed short	vector unsigned short	vslh
vector unsigned long	vector unsigned long	vector unsigned long	vslw
vector signed long	vector signed long	vector unsigned long	vslw

A.57 vec_sld(arg1, arg2, arg3)

The result is obtained by selecting the top 16 bytes obtained by shifting left (unsigned) by the value of **arg3** bytes a 32-byte quantity formed by catenating **arg1** with **arg2**.

Result	arg1	arg2	arg3	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	4-bit unsigned literal	vsldoi
vector signed char	vector signed char	vector signed char	4-bit unsigned literal	vsldoi
vector unsigned short	vector unsigned short	vector unsigned short	4-bit unsigned literal	vsldoi
vector signed short	vector signed short	vector signed short	4-bit unsigned literal	vsldoi
vector pixel	vector pixel	vector pixel	4-bit unsigned literal	vsldoi
vector unsigned long	vector unsigned long	vector unsigned long	4-bit unsigned literal	vsldoi
vector signed long	vector signed long	vector signed long	4-bit unsigned literal	vsldoi
vector float	vector float	vector float	4-bit unsigned literal	vsldoi

A.58 vec_sll(arg1, arg2)

The result is obtained by shifting **arg1** left by a number of bits specified by the last 3 bits of the last element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsl
vector unsigned char	vector unsigned char	vector unsigned short	vsl
vector unsigned char	vector unsigned char	vector unsigned long	vsl
vector signed char	vector signed char	vector unsigned char	vsl
vector signed char	vector signed char	vector unsigned short	vsl
vector signed char	vector signed char	vector unsigned long	vsl
vector bool char	vector bool char	vector unsigned char	vsl
vector bool char	vector bool char	vector unsigned short	vsl
vector bool char	vector bool char	vector unsigned long	vsl
vector unsigned short	vector unsigned short	vector unsigned char	vsl
vector unsigned short	vector unsigned short	vector unsigned short	vsl
vector unsigned short	vector unsigned short	vector unsigned long	vsl
vector signed short	vector signed short	vector unsigned char	vsl
vector signed short	vector signed short	vector unsigned short	vsl
vector signed short	vector signed short	vector unsigned long	vsl
vector bool short	vector bool short	vector unsigned char	vsl
vector bool short	vector bool short	vector unsigned short	vsl
vector bool short	vector bool short	vector unsigned long	vsl
vector pixel	vector pixel	vector unsigned char	vsl
vector pixel	vector pixel	vector unsigned short	vsl
vector pixel	vector pixel	vector unsigned long	vsl
vector unsigned long	vector unsigned long	vector unsigned char	vsl
vector unsigned long	vector unsigned long	vector unsigned short	vsl
vector unsigned long	vector unsigned long	vector unsigned long	vsl
vector signed long	vector signed long	vector unsigned char	vsl
vector signed long	vector signed long	vector unsigned short	vsl
vector signed long	vector signed long	vector unsigned long	vsl
vector bool long	vector bool long	vector unsigned char	vsl
vector bool long	vector bool long	vector unsigned short	vsl
vector bool long	vector bool long	vector unsigned long	vsl

A.59 vec_slo(arg1, arg2)

The result is obtained by shifting **arg1** left by a number of bytes specified by shifting the value of the last element of **arg2** by 3 bits.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vslo
vector unsigned char	vector unsigned char	vector signed char	vslo
vector signed char	vector signed char	vector unsigned char	vslo
vector signed char	vector signed char	vector signed char	vslo
vector unsigned short	vector unsigned short	vector unsigned char	vslo
vector unsigned short	vector unsigned short	vector signed char	vslo
vector signed short	vector signed short	vector unsigned char	vslo
vector signed short	vector signed short	vector signed char	vslo
vector pixel	vector pixel	vector unsigned char	vslo
vector pixel	vector pixel	vector signed char	vslo
vector unsigned long	vector unsigned long	vector unsigned char	vslo
vector unsigned long	vector unsigned long	vector signed char	vslo
vector signed long	vector signed long	vector unsigned char	vslo
vector signed long	vector signed long	vector signed char	vslo
vector float	vector float	vector unsigned char	vslo
vector float	vector float	vector signed char	vslo

A.60 vec_splat(arg1, arg2)

Each element of the result is component **arg2** of **arg1**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	5-bit unsigned literal	vspltb
vector signed char	vector signed char	5-bit unsigned literal	vspltb
vector bool char	vector bool char	5-bit unsigned literal	vspltb
vector unsigned short	vector unsigned short	5-bit unsigned literal	vsplth
vector signed short	vector signed short	5-bit unsigned literal	vsplth
vector bool short	vector bool short	5-bit unsigned literal	vsplth
vector pixel	vector pixel	5-bit unsigned literal	vsplth
vector unsigned long	vector unsigned long	5-bit unsigned literal	vspltw
vector signed long	vector signed long	5-bit unsigned literal	vspltw
vector bool long	vector bool long	5-bit unsigned literal	vspltw
vector float	vector float	5-bit unsigned literal	vspltw

A.61 vec_splat_s8(arg1)

Each element of the result is the value obtained by sign-extending **arg1**. Note that this permits values ranging from -16 to +15 only.

Result	arg1	Maps To
vector signed char	5-bit signed literal	vspltisb

A.62 vec_splat_s16(arg1)

Each element of the result is the value obtained by sign-extending **arg1**. Note that this permits values ranging from -16 to +15 only.

Result	arg1	Maps To
vector signed short	5-bit signed literal	vspltish

A.63 vec_splat_s32(arg1)

Each element of the result is the value obtained by sign-extending **arg1**. Note that this permits values ranging from -16 to +15 only.

Result	arg1	Maps To
vector signed long	5-bit signed literal	vspltisw

A.64 vec_splat_u8(arg1)

Each element of the result is the value obtained by sign-extending **arg1** and casting it to an unsigned char value. Note that this permits values ranging from -16 to +15 with the negative values interpreted as lying interval from 240 to 255 since the result is a vector unsigned char. Values 240 to 255 are also permitted for **arg1** and equivalent to -16 to -1 respectively. Also note, it is necessary to use the generic name since the specific operation `vec_vspltisb` is used for the `vec_splat_s8` generic operation.

Result	arg1	Maps To	Note
vector unsigned char	5-bit signed literal	vspltisb	N

A.65 vec_splat_u16(arg1)

Each element of the result is the value obtained by sign-extending **arg1**. Note that this permits values ranging from -16 to +15 with the negative values interpreted as lying interval from 65520 to 65535 since the result is a vector unsigned short. Values 65520 to 65535 are also permitted for **arg1** and equivalent to -16 to -1 respectively. Also note, it is necessary to use the generic name since the specific operation `vec_vspltish` is used for the `vec_splat_s16` generic operation.

Result	arg1	Maps To	Note
vector unsigned short	5-bit signed literal	vspltish	N

A.66 vec_splat_u32(arg1)

Each element of the result is the value obtained by sign-extending **arg1**. Note that this permits values ranging from -16 to +15 with the negative values interpreted as lying interval from 4294967280 to 4294967295 since the result is a vector unsigned long. Values 4294967280 to 4294967295 are also permitted for **arg1** and equivalent to -16 to -1 respectively. Also note, it is necessary to use the generic name since the specific operation `vec_vspltisw` is used for the `vec_splat_s32` generic operation.

Result	arg1	Maps To	Note
vector unsigned long	5-bit signed literal	vspltisw	N

A.67 vec_sr(arg1, arg2)

Each element of the result is the result of shifting the corresponding element of **arg1** right by the number of bits of the corresponding element of **arg2**. Zero bits are shifted in from the left for both signed and unsigned argument types.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsrb
vector signed char	vector signed char	vector unsigned char	vsrb
vector unsigned short	vector unsigned short	vector unsigned short	vsrh
vector signed short	vector signed short	vector unsigned short	vsrh
vector unsigned long	vector unsigned long	vector unsigned long	vsrw
vector signed long	vector signed long	vector unsigned long	vsrw

A.68 vec_sra(arg1, arg2)

Each element of the result is the result of shifting the corresponding element of **arg1** right by the number of bits of the corresponding element of **arg2**. Copies of the sign bit are shifted in from the left for both signed and unsigned argument types.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsrab
vector signed char	vector signed char	vector unsigned char	vsrab
vector unsigned short	vector unsigned short	vector unsigned short	vsrah
vector signed short	vector signed short	vector unsigned short	vsrah
vector unsigned long	vector unsigned long	vector unsigned long	vsraw
vector signed long	vector signed long	vector unsigned long	vsraw

A.69 vec_srl(arg1, arg2)

The result is obtained by shifting **arg1** right by a number of bits specified by the last 3 bits of the last element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsr
vector unsigned char	vector unsigned char	vector unsigned short	vsr
vector unsigned char	vector unsigned char	vector unsigned long	vsr
vector signed char	vector signed char	vector unsigned char	vsr
vector signed char	vector signed char	vector unsigned short	vsr
vector signed char	vector signed char	vector unsigned long	vsr
vector bool char	vector bool char	vector unsigned char	vsr
vector bool char	vector bool char	vector unsigned short	vsr
vector bool char	vector bool char	vector unsigned long	vsr
vector unsigned short	vector unsigned short	vector unsigned char	vsr
vector unsigned short	vector unsigned short	vector unsigned short	vsr
vector unsigned short	vector unsigned short	vector unsigned long	vsr
vector signed short	vector signed short	vector unsigned char	vsr
vector signed short	vector signed short	vector unsigned short	vsr
vector signed short	vector signed short	vector unsigned long	vsr
vector bool short	vector bool short	vector unsigned char	vsr
vector bool short	vector bool short	vector unsigned short	vsr

vector bool short	vector bool short	vector unsigned long	vsr
vector pixel	vector pixel	vector unsigned char	vsr
vector pixel	vector pixel	vector unsigned short	vsr
vector pixel	vector pixel	vector unsigned long	vsr
vector unsigned long	vector unsigned long	vector unsigned char	vsr
vector unsigned long	vector unsigned long	vector unsigned short	vsr
vector unsigned long	vector unsigned long	vector unsigned long	vsr
vector signed long	vector signed long	vector unsigned char	vsr
vector signed long	vector signed long	vector unsigned short	vsr
vector signed long	vector signed long	vector unsigned long	vsr
vector bool long	vector bool long	vector unsigned char	vsr
vector bool long	vector bool long	vector unsigned short	vsr
vector bool long	vector bool long	vector unsigned long	vsr

A.70 vec_sro(arg1, arg2)

The result is obtained by shifting (unsigned) **arg1** right by a number of bytes specified by shifting the value of the last element of **arg2** by 3 bits.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsro
vector unsigned char	vector unsigned char	vector signed char	vsro
vector signed char	vector signed char	vector unsigned char	vsro
vector signed char	vector signed char	vector signed char	vsro
vector unsigned short	vector unsigned short	vector unsigned char	vsro
vector unsigned short	vector unsigned short	vector signed char	vsro
vector signed short	vector signed short	vector unsigned char	vsro
vector signed short	vector signed short	vector signed char	vsro
vector pixel	vector pixel	vector unsigned char	vsro
vector pixel	vector pixel	vector signed char	vsro
vector unsigned long	vector unsigned long	vector unsigned char	vsro
vector unsigned long	vector unsigned long	vector signed char	vsro
vector signed long	vector signed long	vector unsigned char	vsro
vector signed long	vector signed long	vector signed char	vsro
vector float	vector float	vector unsigned char	vsro
vector float	vector float	vector signed char	vsro

A.71 vec_st(arg1, arg2, arg3)

The 16-byte value of **arg1** is stored at a 16-byte aligned address formed by truncating the last four bits of the sum of **arg2** and **arg3**. **arg2** is taken to be an integer value, while **arg3** is a pointer. Note that this is not, by itself, an acceptable way to store aligned data to unaligned addresses. Note that this store is the one which will be generated for a storing dereference of a pointer to a vector type.

Result	arg1	arg2	arg3	Maps To
void	vector unsigned char	int	vector unsigned char *	stvx
void	vector unsigned char	int	unsigned char *	stvx
void	vector signed char	int	vector signed char *	stvx
void	vector signed char	int	signed char *	stvx

void	vector unsigned short	int	vector unsigned short *	stvx
void	vector unsigned short	int	unsigned short *	stvx
void	vector signed short	int	vector signed short *	stvx
void	vector signed short	int	short *	stvx
void	vector unsigned long	int	vector unsigned long *	stvx
void	vector unsigned long	int	unsigned int *	stvx
void	vector unsigned long	int	unsigned long *	stvx
void	vector signed long	int	vector signed long *	stvx
void	vector signed long	int	int *	stvx
void	vector signed long	int	long *	stvx
void	vector float	int	vector float *	stvx
void	vector float	int	float *	stvx

A.72 vec_ste(arg1, arg2, arg3)

A single element of **arg1** is stored at the address formed by truncating the last 0 (char), 1 (short) or 2 (int, float) bits of the sum of **arg2** and **arg3**. The element stored is the one whose position in the register matches the position of the adjusted address relative to 16-byte alignment. Note that if you don't know the alignment of the sum of **arg2** and **arg3**, you won't know which element is stored.

Result	arg1	arg2	arg3	Maps To
void	vector unsigned char	int	unsigned char *	stvebx
void	vector signed char	int	signed char *	stvebx
void	vector unsigned short	int	unsigned short *	stvehx
void	vector signed short	int	short *	stvehx
void	vector unsigned long	int	unsigned int *	stvewx
void	vector unsigned long	int	unsigned long *	stvewx
void	vector signed long	int	int *	stvewx
void	vector signed long	int	long *	stvewx
void	vector float	int	float *	stvewx

A.73 vec_stl(arg1, arg2, arg3)

The 16-byte value of **arg1** is stored at a 16-byte aligned address formed by truncating the last four bits of the sum of **arg2** and **arg3**. **arg2** is taken to be an integer value, while **arg3** is a pointer. Note that this is not, by itself, an acceptable way to store aligned data to unaligned addresses. The cache line stored into is marked LRU.

Result	arg1	arg2	arg3	Maps To
void	vector unsigned char	int	vector unsigned char *	stvxl
void	vector unsigned char	int	unsigned char *	stvxl
void	vector signed char	int	vector signed char *	stvxl
void	vector signed char	int	signed char *	stvxl
void	vector unsigned short	int	vector unsigned short *	stvxl
void	vector unsigned short	int	unsigned short *	stvxl
void	vector signed short	int	vector signed short *	stvxl
void	vector signed short	int	short *	stvxl
void	vector unsigned long	int	vector unsigned long *	stvxl
void	vector unsigned long	int	unsigned int *	stvxl

void	vector unsigned long	int	unsigned long *	stvxl
void	vector signed long	int	vector signed long *	stvxl
void	vector signed long	int	int *	stvxl
void	vector signed long	int	long *	stvxl
void	vector float	int	vector float *	stvxl
void	vector float	int	float *	stvxl

A.74 vec_sub(arg1, arg2)

Each element of the result is the difference between the corresponding elements of **arg1** and **arg2**. The arithmetic is modular for integer types.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsububm
vector unsigned char	vector unsigned char	vector bool char	vsububm
vector unsigned char	vector bool char	vector unsigned char	vsububm
vector signed char	vector signed char	vector signed char	vsububm
vector signed char	vector signed char	vector bool char	vsububm
vector signed char	vector bool char	vector signed char	vsububm
vector unsigned short	vector unsigned short	vector unsigned short	vsubuhm
vector unsigned short	vector unsigned short	vector bool short	vsubuhm
vector unsigned short	vector bool short	vector unsigned short	vsubuhm
vector signed short	vector signed short	vector signed short	vsubuhm
vector signed short	vector signed short	vector bool short	vsubuhm
vector signed short	vector bool short	vector signed short	vsubuhm
vector unsigned long	vector unsigned long	vector unsigned long	vsubuwm
vector unsigned long	vector unsigned long	vector bool long	vsubuwm
vector unsigned long	vector bool long	vector unsigned long	vsubuwm
vector signed long	vector signed long	vector signed long	vsubuwm
vector signed long	vector signed long	vector bool long	vsubuwm
vector signed long	vector bool long	vector signed long	vsubuwm
vector float	vector float	vector float	vsubfp

A.75 vec_subc(arg1, arg2)

Each element of the result is the value of the carry generated by subtracting the corresponding elements of **arg1** and **arg2**. The value is 0 if a borrow occurred and 1 if no borrow occurred.

Result	arg1	arg2	Maps To
vector unsigned long	vector unsigned long	vector unsigned long	vsubcuw

A.76 vec_subs(arg1, arg2)

Each element of the result is the saturated difference between the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsububs
vector unsigned char	vector unsigned char	vector bool char	vsububs
vector unsigned char	vector bool char	vector unsigned char	vsububs

vector signed char	vector signed char	vector signed char	vsubsbs
vector signed char	vector signed char	vector bool char	vsubsbs
vector signed char	vector bool char	vector signed char	vsubsbs
vector unsigned short	vector unsigned short	vector unsigned short	vsubuhs
vector unsigned short	vector unsigned short	vector bool short	vsubuhs
vector unsigned short	vector bool short	vector unsigned short	vsubuhs
vector signed short	vector signed short	vector signed short	vsubshs
vector signed short	vector signed short	vector bool short	vsubshs
vector signed short	vector bool short	vector signed short	vsubshs
vector unsigned long	vector unsigned long	vector unsigned long	vsubuws
vector unsigned long	vector unsigned long	vector bool long	vsubuws
vector unsigned long	vector bool long	vector unsigned long	vsubuws
vector signed long	vector signed long	vector signed long	vsubsws
vector signed long	vector signed long	vector bool long	vsubsws
vector signed long	vector bool long	vector signed long	vsubsws

A.77 vec_sum4s(arg1, arg2)

Each element of the result is the 32-bit saturated sum of the corresponding element in **arg2** and all elements in **arg1** with positions overlapping those of that element.

Result	arg1	arg2	Maps To
vector unsigned long	vector unsigned char	vector unsigned long	vsum4ubs
vector signed long	vector signed char	vector signed long	vsum4sbs
vector signed long	vector signed short	vector signed long	vsum4shs

A.78 vec_sum2s(arg1, arg2)

The first and third elements of the result are 0. The second element of the result is the 32-bit saturated sum of the first two elements of **arg1** and the second element of **arg2**. The fourth element of the result is the 32-bit saturated sum of the last two elements of **arg1** and the fourth element of **arg2**.

Result	arg1	arg2	Maps To
vector signed long	vector signed long	vector signed long	vsum2sws

A.79 vec_sums(arg1, arg2)

The first three elements of the result are 0. The fourth element of the result is the 32-bit saturated sum of all elements of **arg1** and the fourth element of **arg2**.

Result	arg1	arg2	Maps To
vector signed long	vector signed long	vector signed long	vsumsws

A.80 vec_trunc(arg1)

Each element of the result is the value of the corresponding element of **arg1** truncated to an integral value.

Result	arg1	Maps To
vector float	vector float	vrfiz

A.81 vec_unpackh(arg1)

Each element of the result is the result of extending the corresponding half-width high element of **arg1**.

Result	arg1	Maps To
vector signed short	vector signed char	vupkhsb
vector bool short	vector bool char	vupkhsb
vector unsigned long	vector pixel	vupkhpx
vector signed long	vector signed short	vupkhsh
vector bool long	vector bool short	vupkhsh

A.82 vec_unpackl(arg1)

Each element of the result is the result of extending the corresponding half-width low element of **arg1**.

Result	arg1	Maps To
vector signed short	vector signed char	vupklsb
vector bool short	vector bool char	vupklsb
vector unsigned long	vector pixel	vupklpx
vector signed long	vector signed short	vupklsh
vector bool long	vector bool short	vupklsh

A.83 vec_unpack2sh(arg1, arg2)

These operations form signed double-size elements by catenating each high element of **arg1** with the corresponding high element of **arg2**. If **arg1** is a vector of 0's, this effectively is a signed unpack of the unsigned value **arg2**. Note, it is necessary to use the generic name since the specific operations `vec_vmrghb` and `vec_vmrghh` are also used for the `vec_unpack2uh` generic operation.

Result	arg1	arg2	Maps To	Note
vector signed short	vector unsigned char	vector unsigned char	vmrghb	N
vector signed long	vector unsigned short	vector unsigned short	vmrghh	N

A.84 vec_unpack2sl(arg1, arg2)

These operations form signed double-size elements by catenating each low element of **arg1** with the corresponding low element of **arg2**. If **arg1** is a vector of 0's, this effectively is a signed unpack of the unsigned value **arg2**. Note, it is necessary to use the generic name since the specific operations `vec_vmrglb` and vec_vmrglh are also used for the vec_unpack2ul generic operation.

Result	arg1	arg2	Maps To	Note
vector signed short	vector unsigned char	vector unsigned char	vmrglb	N
vector signed long	vector unsigned short	vector unsigned short	vmrglh	N

A.85 vec_unpack2uh(arg1, arg2)

These operations form unsigned double-size elements by catenating each high element of **arg1** with the corresponding high element of **arg2**. If **arg1** is a vector of 0's, this effectively is an unpack of **arg2**. Note, it is necessary to use the generic name since the specific operations `vec_vmrghb` and `vec_vmrghh` are also used for the `vec_unpack2sh` generic operation.

Result	arg1	arg2	Maps To	Note
vector unsigned short	vector unsigned char	vector unsigned char	vmrghb	N
vector unsigned long	vector unsigned short	vector unsigned short	vmrghh	N

A.86 vec_unpack2ul(arg1, arg2)

These operations form unsigned double-size elements by catenating each low element of **arg1** with the corresponding low element of **arg2**. If **arg1** is a vector of 0's, this effectively is an unpack of **arg2**. Note, it is necessary to use the generic name since the specific operations `vec_vmrglb` and vec_vmrglh are also used for the vec_unpack2sl generic operation.

Result	arg1	arg2	Maps To	Note
vector unsigned short	vector unsigned char	vector unsigned char	vmrglb	N
vector unsigned long	vector unsigned short	vector unsigned short	vmrglh	N

A.87 vec_xor(arg1, arg2)

Each element of the result is the logical XOR of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vxor
vector unsigned char	vector unsigned char	vector bool char	vxor
vector unsigned char	vector bool char	vector unsigned char	vxor
vector signed char	vector signed char	vector signed char	vxor
vector signed char	vector signed char	vector bool char	vxor
vector signed char	vector bool char	vector signed char	vxor
vector bool char	vector bool char	vector bool char	vxor
vector unsigned short	vector unsigned short	vector unsigned short	vxor
vector unsigned short	vector unsigned short	vector bool short	vxor
vector unsigned short	vector bool short	vector unsigned short	vxor
vector signed short	vector signed short	vector signed short	vxor
vector signed short	vector signed short	vector bool short	vxor
vector signed short	vector bool short	vector signed short	vxor
vector bool short	vector bool short	vector bool short	vxor
vector unsigned long	vector unsigned long	vector unsigned long	vxor
vector unsigned long	vector unsigned long	vector bool long	vxor
vector unsigned long	vector bool long	vector unsigned long	vxor
vector signed long	vector signed long	vector signed long	vxor
vector signed long	vector signed long	vector bool long	vxor

vector signed long	vector bool long	vector signed long	vxor
vector bool long	vector bool long	vector bool long	vxor
vector float	vector bool long	vector float	vxor
vector float	vector float	vector bool long	vxor
vector float	vector float	vector float	vxor

Appendix B: AltiVec Predicates

The predicates are organized alphabetically by predicate name. Each table describes a single generic predicate. Each line shows a valid set of argument types for that predicate, and the specific AltiVec instruction generated for that set of arguments. For example, `vec_any_lt(vector unsigned char, vector unsigned char)` will use the instruction "`vcmpgtb.`".

The Notes column for predicates always indicates "N" to show that the specific AltiVec instruction cannot be used by itself. The entry "R" indicates that the operands will be reversed in invoking the instruction, while the entry "D" indicates that the same operand will be used twice.

B.1 vec_all_eq(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector unsigned char	vector unsigned char	vcmpequb.	N
int	vector unsigned char	vector bool char	vcmpequb.	N
int	vector signed char	vector signed char	vcmpequb.	N
int	vector signed char	vector bool char	vcmpequb.	N
int	vector bool char	vector unsigned char	vcmpequb.	N
int	vector bool char	vector signed char	vcmpequb.	N
int	vector unsigned short	vector unsigned short	vcmpequh.	N
int	vector unsigned short	vector bool short	vcmpequh.	N
int	vector signed short	vector signed short	vcmpequh.	N
int	vector signed short	vector bool short	vcmpequh.	N
int	vector bool short	vector unsigned short	vcmpequh.	N
int	vector bool short	vector signed short	vcmpequh.	N
int	vector unsigned long	vector unsigned long	vcmpequw.	N
int	vector unsigned long	vector bool long	vcmpequw.	N
int	vector signed long	vector signed long	vcmpequw.	N
int	vector signed long	vector bool long	vcmpequw.	N
int	vector bool long	vector unsigned long	vcmpequw.	N
int	vector bool long	vector signed long	vcmpequw.	N
int	vector float	vector float	vcmpeqfp.	N

B.2 vec_all_ge(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is greater than or equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector unsigned char	vector unsigned char	vcmpgtub.	NR
int	vector unsigned char	vector bool char	vcmpgtub.	NR
int	vector signed char	vector signed char	vcmpgtsb.	NR
int	vector signed char	vector bool char	vcmpgtsb.	NR
int	vector bool char	vector unsigned char	vcmpgtub.	NR
int	vector bool char	vector signed char	vcmpgtsb.	NR
int	vector unsigned short	vector unsigned short	vcmpgtuh.	NR
int	vector unsigned short	vector bool short	vcmpgtuh.	NR

int	vector signed short	vector signed short	vcmpgtsh.	NR
int	vector signed short	vector bool short	vcmpgtsh.	NR
int	vector bool short	vector unsigned short	vcmpgtuh.	NR
int	vector bool short	vector signed short	vcmpgtsh.	NR
int	vector unsigned long	vector unsigned long	vcmpgtuw.	NR
int	vector unsigned long	vector bool long	vcmpgtuw.	NR
int	vector signed long	vector signed long	vcmpgtsw.	NR
int	vector signed long	vector bool long	vcmpgtsw.	NR
int	vector bool long	vector unsigned long	vcmpgtuw.	NR
int	vector bool long	vector signed long	vcmpgtsw.	NR
int	vector float	vector float	vcmpgefp.	N

B.3 vec_all_gt(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is greater than the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector unsigned char	vector unsigned char	vcmpgtub.	N
int	vector unsigned char	vector bool char	vcmpgtub.	N
int	vector signed char	vector signed char	vcmpgtsb.	N
int	vector signed char	vector bool char	vcmpgtsb.	N
int	vector bool char	vector unsigned char	vcmpgtub.	N
int	vector bool char	vector signed char	vcmpgtsb.	N
int	vector unsigned short	vector unsigned short	vcmpgtuh.	N
int	vector unsigned short	vector bool short	vcmpgtuh.	N
int	vector signed short	vector signed short	vcmpgtsh.	N
int	vector signed short	vector bool short	vcmpgtsh.	N
int	vector bool short	vector unsigned short	vcmpgtuh.	N
int	vector bool short	vector signed short	vcmpgtsh.	N
int	vector unsigned long	vector unsigned long	vcmpgtuw.	N
int	vector unsigned long	vector bool long	vcmpgtuw.	N
int	vector signed long	vector signed long	vcmpgtsw.	N
int	vector signed long	vector bool long	vcmpgtsw.	N
int	vector bool long	vector unsigned long	vcmpgtuw.	N
int	vector bool long	vector signed long	vcmpgtsw.	N
int	vector float	vector float	vcmpgtfp.	N

B.4 vec_all_in(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is less than or equal to the corresponding element of **arg2** and greater than or equal to the negative of the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector float	vector float	vcmpbfp.	N

B.5 vec_all_le(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is less than or equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector unsigned char	vector unsigned char	vcmpgtub.	N
int	vector unsigned char	vector bool char	vcmpgtub.	N
int	vector signed char	vector signed char	vcmpgtsb.	N
int	vector signed char	vector bool char	vcmpgtsb.	N
int	vector bool char	vector unsigned char	vcmpgtub.	N
int	vector bool char	vector signed char	vcmpgtsb.	N
int	vector unsigned short	vector unsigned short	vcmpgtuh.	N
int	vector unsigned short	vector bool short	vcmpgtuh.	N
int	vector signed short	vector signed short	vcmpgtsh.	N
int	vector signed short	vector bool short	vcmpgtsh.	N
int	vector bool short	vector unsigned short	vcmpgtuh.	N
int	vector bool short	vector signed short	vcmpgtsh.	N
int	vector unsigned long	vector unsigned long	vcmpgtuw.	N
int	vector unsigned long	vector bool long	vcmpgtuw.	N
int	vector signed long	vector signed long	vcmpgtsw.	N
int	vector signed long	vector bool long	vcmpgtsw.	N
int	vector bool long	vector unsigned long	vcmpgtuw.	N
int	vector bool long	vector signed long	vcmpgtsw.	N
int	vector float	vector float	vcmpgefp.	NR

B.6 vec_all_lt(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is less than the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector unsigned char	vector unsigned char	vcmpgtub.	NR
int	vector unsigned char	vector bool char	vcmpgtub.	NR
int	vector signed char	vector signed char	vcmpgtsb.	NR
int	vector signed char	vector bool char	vcmpgtsb.	NR
int	vector bool char	vector unsigned char	vcmpgtub.	NR
int	vector bool char	vector signed char	vcmpgtsb.	NR
int	vector unsigned short	vector unsigned short	vcmpgtuh.	NR
int	vector unsigned short	vector bool short	vcmpgtuh.	NR
int	vector signed short	vector signed short	vcmpgtsh.	NR
int	vector signed short	vector bool short	vcmpgtsh.	NR
int	vector bool short	vector unsigned short	vcmpgtuh.	NR
int	vector bool short	vector signed short	vcmpgtsh.	NR
int	vector unsigned long	vector unsigned long	vcmpgtuw.	NR
int	vector unsigned long	vector bool long	vcmpgtuw.	NR
int	vector signed long	vector signed long	vcmpgtsw.	NR
int	vector signed long	vector bool long	vcmpgtsw.	NR
int	vector bool long	vector unsigned long	vcmpgtuw.	NR
int	vector bool long	vector signed long	vcmpgtsw.	NR
int	vector float	vector float	vcmpgtfp.	NR

B.7 vec_all_nan(arg1)

Each predicate returns 1 if each element of **arg1** is a NaN. Otherwise, it returns 0.

Result	arg1	Maps To	Note
int	vector float	vcmpeqfp.	ND

B.8 vec_all_ne(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is not equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector unsigned char	vector unsigned char	vcmpequb.	N
int	vector unsigned char	vector bool char	vcmpequb.	N
int	vector signed char	vector signed char	vcmpequb.	N
int	vector signed char	vector bool char	vcmpequb.	N
int	vector bool char	vector unsigned char	vcmpequb.	N
int	vector bool char	vector signed char	vcmpequb.	N
int	vector unsigned short	vector unsigned short	vcmpequh.	N
int	vector unsigned short	vector bool short	vcmpequh.	N
int	vector signed short	vector signed short	vcmpequh.	N
int	vector signed short	vector bool short	vcmpequh.	N
int	vector bool short	vector unsigned short	vcmpequh.	N
int	vector bool short	vector signed short	vcmpequh.	N
int	vector unsigned long	vector unsigned long	vcmpequw.	N
int	vector unsigned long	vector bool long	vcmpequw.	N
int	vector signed long	vector signed long	vcmpequw.	N
int	vector signed long	vector bool long	vcmpequw.	N
int	vector bool long	vector unsigned long	vcmpequw.	N
int	vector bool long	vector signed long	vcmpequw.	N
int	vector float	vector float	vcmpeqfp.	N

B.9 vec_all_nge(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is not greater than or equal to the corresponding element of **arg2**. Otherwise, it returns 0. Not greater than or equal can mean either less than or that one of the elements is a NaN.

Result	arg1	arg2	Maps To	Note
int	vector float	vector float	vcmpgefp.	N

B.10 vec_all_ngt(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is not greater than the corresponding element of **arg2**. Otherwise, it returns 0. Not greater than or equal can mean either less than or equal to or that one of the elements is a NaN.

Result	arg1	arg2	Maps To	Note

int	vector float	vector float	vcmpgtfp.	N

B.11 vec_all_nle(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is not less than or equal to the corresponding element of **arg2**. Otherwise, it returns 0. Not greater than or equal can mean either greater than or that one of the elements is a NaN.

Result	arg1	arg2	Maps To	Note
int	vector float	vector float	vcmpgefp.	NR

B.12 vec_all_nlt(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is not less than the corresponding element of **arg2**. Otherwise, it returns 0. Not greater than or equal can mean either greater than or equal to or that one of the elements is a NaN.

Result	arg1	arg2	Maps To	Note
int	vector float	vector float	vcmpgtfp.	NR

B.13 vec_all_numeric(arg1)

Each predicate returns 1 if each element of **arg1** is numeric (not a NaN). Otherwise, it returns 0.

Result	arg1	Maps To	Note
int	vector float	vcmpeqfp.	ND

B.14 vec_any_eq(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector unsigned char	vector unsigned char	vcmpequb.	N
int	vector unsigned char	vector bool char	vcmpequb.	N
int	vector signed char	vector signed char	vcmpequb.	N
int	vector signed char	vector bool char	vcmpequb.	N
int	vector bool char	vector unsigned char	vcmpequb.	N
int	vector bool char	vector signed char	vcmpequb.	N
int	vector unsigned short	vector unsigned short	vcmpequh.	N
int	vector unsigned short	vector bool short	vcmpequh.	N
int	vector signed short	vector signed short	vcmpequh.	N
int	vector signed short	vector bool short	vcmpequh.	N
int	vector bool short	vector unsigned short	vcmpequh.	N
int	vector bool short	vector signed short	vcmpequh.	N
int	vector unsigned long	vector unsigned long	vcmpequw.	N
int	vector unsigned long	vector bool long	vcmpequw.	N
int	vector signed long	vector signed long	vcmpequw.	N
int	vector signed long	vector bool long	vcmpequw.	N
int	vector bool long	vector unsigned long	vcmpequw.	N

int	vector bool long	vector signed long	vcmpequw.	N
int	vector float	vector float	vcmpeqfp.	N

B.15 vec_any_ge(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is greater than or equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector unsigned char	vector unsigned char	vcmpgtub.	NR
int	vector unsigned char	vector bool char	vcmpgtub.	NR
int	vector signed char	vector signed char	vcmpgtsb.	NR
int	vector signed char	vector bool char	vcmpgtsb.	NR
int	vector bool char	vector unsigned char	vcmpgtub.	NR
int	vector bool char	vector signed char	vcmpgtsb.	NR
int	vector unsigned short	vector unsigned short	vcmpgtuh.	NR
int	vector unsigned short	vector bool short	vcmpgtuh.	NR
int	vector signed short	vector signed short	vcmpgtsh.	NR
int	vector signed short	vector bool short	vcmpgtsh.	NR
int	vector bool short	vector unsigned short	vcmpgtuh.	NR
int	vector bool short	vector signed short	vcmpgtsh.	NR
int	vector unsigned long	vector unsigned long	vcmpgtuw.	NR
int	vector unsigned long	vector bool long	vcmpgtuw.	NR
int	vector signed long	vector signed long	vcmpgtsw.	NR
int	vector signed long	vector bool long	vcmpgtsw.	NR
int	vector bool long	vector unsigned long	vcmpgtuw.	NR
int	vector bool long	vector signed long	vcmpgtsw.	NR
int	vector float	vector float	vcmpgefp.	N

B.16 vec_any_gt(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is greater than the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector unsigned char	vector unsigned char	vcmpgtub.	N
int	vector unsigned char	vector bool char	vcmpgtub.	N
int	vector signed char	vector signed char	vcmpgtsb.	N
int	vector signed char	vector bool char	vcmpgtsb.	N
int	vector bool char	vector unsigned char	vcmpgtub.	N
int	vector bool char	vector signed char	vcmpgtsb.	N
int	vector unsigned short	vector unsigned short	vcmpgtuh.	N
int	vector unsigned short	vector bool short	vcmpgtuh.	N
int	vector signed short	vector signed short	vcmpgtsh.	N
int	vector signed short	vector bool short	vcmpgtsh.	N
int	vector bool short	vector unsigned short	vcmpgtuh.	N
int	vector bool short	vector signed short	vcmpgtsh.	N
int	vector unsigned long	vector unsigned long	vcmpgtuw.	N
int	vector unsigned long	vector bool long	vcmpgtuw.	N
int	vector signed long	vector signed long	vcmpgtsw.	N

int	vector signed long	vector bool long	vcmpgtsw.	N
int	vector bool long	vector unsigned long	vcmpgtuw.	N
int	vector bool long	vector signed long	vcmpgtsw.	N
int	vector float	vector float	vcmpgtfp.	N

B.17 vec_any_le(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is less than or equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector unsigned char	vector unsigned char	vcmpgtub.	N
int	vector unsigned char	vector bool char	vcmpgtub.	N
int	vector signed char	vector signed char	vcmpgtsb.	N
int	vector signed char	vector bool char	vcmpgtsb.	N
int	vector bool char	vector unsigned char	vcmpgtub.	N
int	vector bool char	vector signed char	vcmpgtsb.	N
int	vector unsigned short	vector unsigned short	vcmpgtuh.	N
int	vector unsigned short	vector bool short	vcmpgtuh.	N
int	vector signed short	vector signed short	vcmpgtsh.	N
int	vector signed short	vector bool short	vcmpgtsh.	N
int	vector bool short	vector unsigned short	vcmpgtuh.	N
int	vector bool short	vector signed short	vcmpgtsh.	N
int	vector unsigned long	vector unsigned long	vcmpgtuw.	N
int	vector unsigned long	vector bool long	vcmpgtuw.	N
int	vector signed long	vector signed long	vcmpgtsw.	N
int	vector signed long	vector bool long	vcmpgtsw.	N
int	vector bool long	vector unsigned long	vcmpgtuw.	N
int	vector bool long	vector signed long	vcmpgtsw.	N
int	vector float	vector float	vcmpgefp.	NR

B.18 vec_any_lt(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is less than the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector unsigned char	vector unsigned char	vcmpgtub.	NR
int	vector unsigned char	vector bool char	vcmpgtub.	NR
int	vector signed char	vector signed char	vcmpgtsb.	NR
int	vector signed char	vector bool char	vcmpgtsb.	NR
int	vector bool char	vector unsigned char	vcmpgtub.	NR
int	vector bool char	vector signed char	vcmpgtsb.	NR
int	vector unsigned short	vector unsigned short	vcmpgtuh.	NR
int	vector unsigned short	vector bool short	vcmpgtuh.	NR
int	vector signed short	vector signed short	vcmpgtsh.	NR
int	vector signed short	vector bool short	vcmpgtsh.	NR
int	vector bool short	vector unsigned short	vcmpgtuh.	NR
int	vector bool short	vector signed short	vcmpgtsh.	NR
int	vector unsigned long	vector unsigned long	vcmpgtuw.	NR

int	vector unsigned long	vector bool long	vcmpgtuw.	NR
int	vector signed long	vector signed long	vcmpgtsw.	NR
int	vector signed long	vector bool long	vcmpgtsw.	NR
int	vector bool long	vector unsigned long	vcmpgtuw.	NR
int	vector bool long	vector signed long	vcmpgtsw.	NR
int	vector float	vector float	vcmpgtfp.	NR

B.19 vec_any_nan(arg1)

Each predicate returns 1 if at least one element of **arg1** is a NaN. Otherwise, it returns 0.

Result	arg1	Maps To	Note
int	vector float	vcmpeqfp.	ND

B.20 vec_any_ne(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is not equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To	Note
int	vector unsigned char	vector unsigned char	vcmpequb.	N
int	vector unsigned char	vector bool char	vcmpequb.	N
int	vector signed char	vector signed char	vcmpequb.	N
int	vector signed char	vector bool char	vcmpequb.	N
int	vector bool char	vector unsigned char	vcmpequb.	N
int	vector bool char	vector signed char	vcmpequb.	N
int	vector unsigned short	vector unsigned short	vcmpequh.	N
int	vector unsigned short	vector bool short	vcmpequh.	N
int	vector signed short	vector signed short	vcmpequh.	N
int	vector signed short	vector bool short	vcmpequh.	N
int	vector bool short	vector unsigned short	vcmpequh.	N
int	vector bool short	vector signed short	vcmpequh.	N
int	vector unsigned long	vector unsigned long	vcmpequw.	N
int	vector unsigned long	vector bool long	vcmpequw.	N
int	vector signed long	vector signed long	vcmpequw.	N
int	vector signed long	vector bool long	vcmpequw.	N
int	vector bool long	vector unsigned long	vcmpequw.	N
int	vector bool long	vector signed long	vcmpequw.	N
int	vector float	vector float	vcmpeqfp.	N

B.21 vec_any_nge(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is not greater than or equal to the corresponding element of **arg2**. Otherwise, it returns 0. Not greater than or equal can mean either less than or that one of the elements is a NaN.

Result	arg1	arg2	Maps To	Note
int	vector float	vector float	vcmpgefp.	N

B.22 vec_any_ngt(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is not greater than the corresponding element of **arg2**. Otherwise, it returns 0. Not greater than can mean either less than or equal to or that one of the elements is a NaN.

Result	arg1	arg2	Maps To	Note
int	vector float	vector float	vcmpgtfp.	N

B.23 vec_any_nle(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is not less than or equal to the corresponding element of **arg2**. Otherwise, it returns 0. Not less than or equal can mean either greater than or that one of the elements is a NaN.

Result	arg1	arg2	Maps To	Note
int	vector float	vector float	vcmpgefp.	NR

B.24 vec_any_nlt(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is not less than the corresponding element of **arg2**. Otherwise, it returns 0. Not less than can mean either greater than or equal to or that one of the elements is a NaN.

Result	arg1	arg2	Maps To	Note
int	vector float	vector float	vcmpgtfp.	NR

B.25 vec_any_numeric(arg1)

Each predicate returns 1 if at least one element of **arg1** is numeric (not a NaN). Otherwise, it returns 0.

Result	arg1	Maps To	Note
int	vector float	vcmpeqfp.	ND

B.26 vec_any_out(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is not less than or equal to the corresponding element of **arg2** or not greater than or equal to the negative of the corresponding element of **arg2**. Otherwise, it returns 0. Not less than or equal can mean greater than or that either argument is a NaN. Not greater than or equal can mean less than or that either argument is a NaN.

Result	arg1	arg2	Maps To	Note
int	vector float	vector float	vcmpbfp.	N

Appendix C: C++ Name Mangling of the Vector Data Types

The MrCpp C++ name mangling rules have been extended to support all the new vector data types. The following table defines the basic type mangling strings used when one of the vector types appears in a C++ function signature.

Vector Data Type	Type Mangling String
vector unsigned char	`XUc`
vector signed char	`Xc`
vector bool char	`XC`
vector unsigned short	`XUs`
vector signed short	`Xs`
vector bool short	`XS`
vector unsigned long	`XUi`
vector signed long	`Xi`
vector bool long	`Xl`
vector float	`Xf`
vector pixel	`Xp`

C++ Mangling conventions of the basic vector data types

Examples:

```
vector unsigned char example1(vector unsigned char)
        example1__FXUc

void example2(vector signed char *)
        example2__FPXc

vector unsigned char example3(vector unsigned char, vector signed char *)
        example3__FXUcPXc

vector float *example4(vector signed char *, vector unsigned short[][10],
                       vector bool char)
        example4__FPXcPA10XUsXC
```