`void * __builtin_frob_return_addr (void *addr)` [Built-in Function]
This function does the reverse of `__builtin_extract_return_addr`.

`void * __builtin_frame_address (unsigned int level)` [Built-in Function]
This function is similar to `__builtin_return_address`, but it returns the address of the function frame rather than the return address of the function. Calling `__builtin_frame_address` with a value of `0` yields the frame address of the current function, a value of `1` yields the frame address of the caller of the current function, and so forth.

The frame is the area on the stack that holds local variables and saved registers. The frame address is normally the address of the first word pushed on to the stack by the function. However, the exact definition depends upon the processor and the calling convention. If the processor has a dedicated frame pointer register, and the function has a frame, then `__builtin_frame_address` returns the value of the frame pointer register.

On some machines it may be impossible to determine the frame address of any function other than the current one; in such cases, or when the top of the stack has been reached, this function returns `0` if the first frame pointer is properly initialized by the startup code.

Calling this function with a nonzero argument can have unpredictable effects, including crashing the calling program. As a result, calls that are considered unsafe are diagnosed when the `-Wframe-address` option is in effect. Such calls should only be made in debugging situations.

6.52 Using Vector Instructions through Built-in Functions

On some targets, the instruction set contains SIMD vector instructions which operate on multiple values contained in one large register at the same time. For example, on the x86 the MMX, 3DNow! and SSE extensions can be used this way.

The first step in using these extensions is to provide the necessary data types. This should be done using an appropriate `typedef`:

```
typedef int v4si __attribute__ ((vector_size (16)));
```

The `int` type specifies the *base type*, while the attribute specifies the vector size for the variable, measured in bytes. For example, the declaration above causes the compiler to set the mode for the `v4si` type to be 16 bytes wide and divided into `int` sized units. For a 32-bit `int` this means a vector of 4 units of 4 bytes, and the corresponding mode of `foo` is V4SI.

The `vector_size` attribute is only applicable to integral and floating scalars, although arrays, pointers, and function return values are allowed in conjunction with this construct. Only sizes that are positive power-of-two multiples of the base type size are currently allowed.

All the basic integer types can be used as base types, both as signed and as unsigned: `char`, `short`, `int`, `long`, `long long`. In addition, `float` and `double` can be used to build floating-point vector types.

Specifying a combination that is not valid for the current architecture causes GCC to synthesize the instructions using a narrower mode. For example, if you specify a variable of

type `V4SI` and your architecture does not allow for this specific SIMD type, GCC produces code that uses 4 `SIs`.

The types defined in this manner can be used with a subset of normal C operations. Currently, GCC allows using the following operators on these types: `+, -, *, /, unary minus, ^, |, &, ~, %`.

The operations behave like C++ `valarrays`. Addition is defined as the addition of the corresponding elements of the operands. For example, in the code below, each of the 4 elements in *a* is added to the corresponding 4 elements in *b* and the resulting vector is stored in *c*.

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a, b, c;

c = a + b;
```

Subtraction, multiplication, division, and the logical operations operate in a similar manner. Likewise, the result of using the unary minus or complement operators on a vector type is a vector whose elements are the negative or complemented values of the corresponding elements in the operand.

It is possible to use shifting operators `<<`, `>>` on integer-type vectors. The operation is defined as following: `{a0, a1, ..., an} >> {b0, b1, ..., bn} == {a0 >> b0, a1 >> b1, ..., an >> bn}`. Vector operands must have the same number of elements.

For convenience, it is allowed to use a binary vector operation where one operand is a scalar. In that case the compiler transforms the scalar operand into a vector where each element is the scalar from the operation. The transformation happens only if the scalar could be safely converted to the vector-element type. Consider the following code.

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a, b, c;
long l;

a = b + 1;    /* a = b + {1,1,1,1}; */
a = 2 * b;    /* a = {2,2,2,2} * b; */

a = l + a;    /* Error, cannot convert long to int. */
```

Vectors can be subscripted as if the vector were an array with the same number of elements and base type. Out of bound accesses invoke undefined behavior at run time. Warnings for out of bound accesses for vector subscription can be enabled with `-Warray-bounds`.

Vector comparison is supported with standard comparison operators: `==, !=, <, <=, >, >=`. Comparison operands can be vector expressions of integer-type or real-type. Comparison between integer-type vectors and real-type vectors are not supported. The result of the comparison is a vector of the same width and number of elements as the comparison operands with a signed integral element type.

Vectors are compared element-wise producing 0 when comparison is false and -1 (constant of the appropriate type where all bits are set) otherwise. Consider the following example.

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a = {1,2,3,4};
v4si b = {3,2,1,4};
```

```
v4si c;

c = a >  b;     /* The result would be {0, 0,-1, 0}  */
c = a == b;     /* The result would be {0,-1, 0,-1}  */
```

In C++, the ternary operator `?:` is available. `a?b:c`, where `b` and `c` are vectors of the same type and `a` is an integer vector with the same number of elements of the same size as `b` and `c`, computes all three arguments and creates a vector `{a[0]?b[0]:c[0], a[1]?b[1]:c[1], ...}`. Note that unlike in OpenCL, `a` is thus interpreted as `a != 0` and not `a < 0`. As in the case of binary operations, this syntax is also accepted when one of `b` or `c` is a scalar that is then transformed into a vector. If both `b` and `c` are scalars and the type of `true?b:c` has the same size as the element type of `a`, then `b` and `c` are converted to a vector type whose elements have this type and with the same number of elements as `a`.

In C++, the logic operators `!, &&, ||` are available for vectors. `!v` is equivalent to `v == 0`, `a && b` is equivalent to `a!=0 & b!=0` and `a || b` is equivalent to `a!=0 | b!=0`. For mixed operations between a scalar `s` and a vector `v`, `s && v` is equivalent to `s?v!=0:0` (the evaluation is short-circuit) and `v && s` is equivalent to `v!=0 & (s?-1:0)`.

Vector shuffling is available using functions `__builtin_shuffle (vec, mask)` and `__builtin_shuffle (vec0, vec1, mask)`. Both functions construct a permutation of elements from one or two vectors and return a vector of the same type as the input vector(s). The *mask* is an integral vector with the same width (W) and element count (N) as the output vector.

The elements of the input vectors are numbered in memory ordering of *vec0* beginning at 0 and *vec1* beginning at N. The elements of *mask* are considered modulo N in the single-operand case and modulo $2 * N$ in the two-operand case.

Consider the following example,

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a = {1,2,3,4};
v4si b = {5,6,7,8};
v4si mask1 = {0,1,1,3};
v4si mask2 = {0,4,2,5};
v4si res;

res = __builtin_shuffle (a, mask1);       /* res is {1,2,2,4}  */
res = __builtin_shuffle (a, b, mask2);    /* res is {1,5,3,6}  */
```

Note that `__builtin_shuffle` is intentionally semantically compatible with the OpenCL `shuffle` and `shuffle2` functions.

You can declare variables and use them in function calls and returns, as well as in assignments and some casts. You can specify a vector type as a return type for a function. Vector types can also be used as function arguments. It is possible to cast from one vector type to another, provided they are of the same size (in fact, you can also cast vectors to and from other datatypes of the same size).

You cannot operate between vectors of different lengths or different signedness without a cast.

Vector shuffling is available using the `__builtin_shufflevector (vec1, vec2, index...)` function. *vec1* and *vec2* must be expressions with vector type with a compatible element type. The result of `__builtin_shufflevector` is a vector with the

same element type as *vec1* and *vec2* but that has an element count equal to the number of indices specified.

The *index* arguments are a list of integers that specify the elements indices of the first two vectors that should be extracted and returned in a new vector. These element indices are numbered sequentially starting with the first vector, continuing into the second vector. An index of -1 can be used to indicate that the corresponding element in the returned vector is a don't care and can be freely chosen to optimized the generated code sequence performing the shuffle operation.

Consider the following example,

```
typedef int v4si __attribute__ ((vector_size (16)));
typedef int v8si __attribute__ ((vector_size (32)));

v8si a = {1,-2,3,-4,5,-6,7,-8};
v4si b = __builtin_shufflevector (a, a, 0, 2, 4, 6); /* b is {1,3,5,7} */
v4si c = {-2,-4,-6,-8};
v8si d = __builtin_shufflevector (c, b, 4, 0, 5, 1, 6, 2, 7, 3); /* d is a */
```

Vector conversion is available using the `__builtin_convertvector (vec, vectype)` function. *vec* must be an expression with integral or floating vector type and *vectype* an integral or floating vector type with the same number of elements. The result has *vectype* type and value of a C cast of every element of *vec* to the element type of *vectype*.

Consider the following example,

```
typedef int v4si __attribute__ ((vector_size (16)));
typedef float v4sf __attribute__ ((vector_size (16)));
typedef double v4df __attribute__ ((vector_size (32)));
typedef unsigned long long v4di __attribute__ ((vector_size (32)));

v4si a = {1,-2,3,-4};
v4sf b = {1.5f,-2.5f,3.f,7.f};
v4di c = {1ULL,5ULL,0ULL,10ULL};
v4sf d = __builtin_convertvector (a, v4sf); /* d is {1.f,-2.f,3.f,-4.f} */
/* Equivalent of:
   v4sf d = { (float)a[0], (float)a[1], (float)a[2], (float)a[3] }; */
v4df e = __builtin_convertvector (a, v4df); /* e is {1.,-2.,3.,-4.} */
v4df f = __builtin_convertvector (b, v4df); /* f is {1.5,-2.5,3.,7.} */
v4si g = __builtin_convertvector (f, v4si); /* g is {1,-2,3,7} */
v4si h = __builtin_convertvector (c, v4si); /* h is {1,5,0,10} */
```

Sometimes it is desirable to write code using a mix of generic vector operations (for clarity) and machine-specific vector intrinsics (to access vector instructions that are not exposed via generic built-ins). On x86, intrinsic functions for integer vectors typically use the same vector type `__m128i` irrespective of how they interpret the vector, making it necessary to cast their arguments and return values from/to other vector types. In C, you can make use of a `union` type:

```
#include <immintrin.h>

typedef unsigned char u8x16 __attribute__ ((vector_size (16)));
typedef unsigned int  u32x4 __attribute__ ((vector_size (16)));

typedef union {
        __m128i mm;
        u8x16   u8;
        u32x4   u32;
```

```
} v128;
```

for variables that can be used with both built-in operators and x86 intrinsics:

```
v128 x, y = { 0 };
memcpy (&x, ptr, sizeof x);
y.u8  += 0x80;
x.mm  = _mm_adds_epu8 (x.mm, y.mm);
x.u32 &= 0xffffff;

/* Instead of a variable, a compound literal may be used to pass the
   return value of an intrinsic call to a function expecting the union: */
v128 foo (v128);
x = foo ((v128) {_mm_adds_epu8 (x.mm, y.mm)});
```

6.53 Support for `offsetof`

GCC implements for both C and C++ a syntactic extension to implement the `offsetof` macro.

```
primary:
        "__builtin_offsetof" "(" typename "," offsetof_member_designator ")"

offsetof_member_designator:
          identifier
        | offsetof_member_designator "." identifier
        | offsetof_member_designator "[" expr "]"
```

This extension is sufficient such that

```
#define offsetof(type, member)  __builtin_offsetof (type, member)
```

is a suitable definition of the `offsetof` macro. In C++, *type* may be dependent. In either case, *member* may consist of a single identifier, or a sequence of member accesses and array references.

6.54 Legacy `__sync` Built-in Functions for Atomic Memory Access

The following built-in functions are intended to be compatible with those described in the *Intel Itanium Processor-specific Application Binary Interface*, section 7.4. As such, they depart from normal GCC practice by not using the '`__builtin_`' prefix and also by being overloaded so that they work on multiple types.

The definition given in the Intel documentation allows only for the use of the types `int`, `long`, `long long` or their unsigned counterparts. GCC allows any scalar type that is 1, 2, 4 or 8 bytes in size other than the C type `_Bool` or the C++ type `bool`. Operations on pointer arguments are performed as if the operands were of the `uintptr_t` type. That is, they are not scaled by the size of the type to which the pointer points.

These functions are implemented in terms of the '`__atomic`' builtins (see Section 6.55 [__atomic Builtins], page 733). They should not be used for new code which should use the '`__atomic`' builtins instead.

Not all operations are supported by all target processors. If a particular operation cannot be implemented on the target processor, a warning is generated and a call to an external function is generated. The external function carries the same name as the built-in version, with an additional suffix '`_n`' where *n* is the size of the data type.

In most cases, these built-in functions are considered a *full barrier*. That is, no memory operand is moved across the operation, either forward or backward. Further, instructions are issued as necessary to prevent the processor from speculating loads across the operation and from queuing stores after the operation.

All of the routines are described in the Intel documentation to take "an optional list of variables protected by the memory barrier". It's not clear what is meant by that; it could mean that *only* the listed variables are protected, or it could mean a list of additional variables to be protected. The list is ignored by GCC which treats it as empty. GCC interprets an empty list as meaning that all globally accessible variables should be protected.

`type __sync_fetch_and_add (type *ptr, type value, ...)` [Built-in Function]
`type __sync_fetch_and_sub (type *ptr, type value, ...)` [Built-in Function]
`type __sync_fetch_and_or (type *ptr, type value, ...)` [Built-in Function]
`type __sync_fetch_and_and (type *ptr, type value, ...)` [Built-in Function]
`type __sync_fetch_and_xor (type *ptr, type value, ...)` [Built-in Function]
`type __sync_fetch_and_nand (type *ptr, type value, ...)` [Built-in Function]

These built-in functions perform the operation suggested by the name, and returns the value that had previously been in memory. That is, operations on integer operands have the following semantics. Operations on pointer arguments are performed as if the operands were of the `uintptr_t` type. That is, they are not scaled by the size of the type to which the pointer points.

```
{ tmp = *ptr; *ptr op= value; return tmp; }
{ tmp = *ptr; *ptr = ~(tmp & value); return tmp; }   // nand
```

The object pointed to by the first argument must be of integer or pointer type. It must not be a boolean type.

Note: GCC 4.4 and later implement `__sync_fetch_and_nand` as `*ptr = ~(tmp & value)` instead of `*ptr = ~tmp & value`.

`type __sync_add_and_fetch (type *ptr, type value, ...)` [Built-in Function]
`type __sync_sub_and_fetch (type *ptr, type value, ...)` [Built-in Function]
`type __sync_or_and_fetch (type *ptr, type value, ...)` [Built-in Function]
`type __sync_and_and_fetch (type *ptr, type value, ...)` [Built-in Function]
`type __sync_xor_and_fetch (type *ptr, type value, ...)` [Built-in Function]

type `__sync_nand_and_fetch (`*type* `*ptr, `*type* `value, ...)` [Built-in Function]

These built-in functions perform the operation suggested by the name, and return the new value. That is, operations on integer operands have the following semantics. Operations on pointer operands are performed as if the operand's type were `uintptr_t`.

```
{ *ptr op= value; return *ptr; }
{ *ptr = ~(*ptr & value); return *ptr; }   // nand
```

The same constraints on arguments apply as for the corresponding `__sync_op_and_fetch` built-in functions.

Note: GCC 4.4 and later implement `__sync_nand_and_fetch` as `*ptr = ~(*ptr & value)` instead of `*ptr = ~*ptr & value`.

`bool __sync_bool_compare_and_swap (`*type* `*ptr, `*type* `oldval, `*type* `newval, ...)` [Built-in Function]

type `__sync_val_compare_and_swap (`*type* `*ptr, `*type* `oldval, `*type* `newval, ...)` [Built-in Function]

These built-in functions perform an atomic compare and swap. That is, if the current value of `*`*ptr* is *oldval*, then write *newval* into `*`*ptr*.

The "bool" version returns `true` if the comparison is successful and *newval* is written. The "val" version returns the contents of `*`*ptr* before the operation.

`void __sync_synchronize (...)` [Built-in Function]

This built-in function issues a full memory barrier.

type `__sync_lock_test_and_set (`*type* `*ptr, `*type* `value, ...)` [Built-in Function]

This built-in function, as described by Intel, is not a traditional test-and-set operation, but rather an atomic exchange operation. It writes *value* into `*`*ptr*, and returns the previous contents of `*`*ptr*.

Many targets have only minimal support for such locks, and do not support a full exchange operation. In this case, a target may support reduced functionality here by which the *only* valid value to store is the immediate constant 1. The exact value actually stored in `*`*ptr* is implementation defined.

This built-in function is not a full barrier, but rather an *acquire barrier*. This means that references after the operation cannot move to (or be speculated to) before the operation, but previous memory stores may not be globally visible yet, and previous memory loads may not yet be satisfied.

`void __sync_lock_release (`*type* `*ptr, ...)` [Built-in Function]

This built-in function releases the lock acquired by `__sync_lock_test_and_set`. Normally this means writing the constant 0 to `*`*ptr*.

This built-in function is not a full barrier, but rather a *release barrier*. This means that all previous memory stores are globally visible, and all previous memory loads have been satisfied, but following memory reads are not prevented from being speculated to before the barrier.

6.55 Built-in Functions for Memory Model Aware Atomic Operations

The following built-in functions approximately match the requirements for the C++11 memory model. They are all identified by being prefixed with '`__atomic`' and most are overloaded so that they work with multiple types.

These functions are intended to replace the legacy '`__sync`' builtins. The main difference is that the memory order that is requested is a parameter to the functions. New code should always use the '`__atomic`' builtins rather than the '`__sync`' builtins.

Note that the '`__atomic`' builtins assume that programs will conform to the C++11 memory model. In particular, they assume that programs are free of data races. See the C++11 standard for detailed requirements.

The '`__atomic`' builtins can be used with any integral scalar or pointer type that is 1, 2, 4, or 8 bytes in length. 16-byte integral types are also allowed if '`__int128`' (see Section 6.9 [__int128], page 550) is supported by the architecture.

The four non-arithmetic functions (load, store, exchange, and compare_exchange) all have a generic version as well. This generic version works on any data type. It uses the lock-free built-in function if the specific data type size makes that possible; otherwise, an external call is left to be resolved at run time. This external call is the same format with the addition of a '`size_t`' parameter inserted as the first parameter indicating the size of the object being pointed to. All objects must be the same size.

There are 6 different memory orders that can be specified. These map to the C++11 memory orders with the same names, see the C++11 standard or the GCC wiki on atomic synchronization (`https://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicSync`) for detailed definitions. Individual targets may also support additional memory orders for use on specific architectures. Refer to the target documentation for details of these.

An atomic operation can both constrain code motion and be mapped to hardware instructions for synchronization between threads (e.g., a fence). To which extent this happens is controlled by the memory orders, which are listed here in approximately ascending order of strength. The description of each memory order is only meant to roughly illustrate the effects and is not a specification; see the C++11 memory model for precise semantics.

`__ATOMIC_RELAXED`
: Implies no inter-thread ordering constraints.

`__ATOMIC_CONSUME`
: This is currently implemented using the stronger `__ATOMIC_ACQUIRE` memory order because of a deficiency in C++11's semantics for `memory_order_consume`.

`__ATOMIC_ACQUIRE`
: Creates an inter-thread happens-before constraint from the release (or stronger) semantic store to this acquire load. Can prevent hoisting of code to before the operation.

`__ATOMIC_RELEASE`
: Creates an inter-thread happens-before constraint to acquire (or stronger) semantic loads that read from this release store. Can prevent sinking of code to after the operation.

`__ATOMIC_ACQ_REL`
: Combines the effects of both `__ATOMIC_ACQUIRE` and `__ATOMIC_RELEASE`.

`__ATOMIC_SEQ_CST`
: Enforces total ordering with all other `__ATOMIC_SEQ_CST` operations.

Note that in the C++11 memory model, *fences* (e.g., '`__atomic_thread_fence`') take effect in combination with other atomic operations on specific memory locations (e.g., atomic loads); operations on specific memory locations do not necessarily affect other operations in the same way.

Target architectures are encouraged to provide their own patterns for each of the atomic built-in functions. If no target is provided, the original non-memory model set of '`__sync`' atomic built-in functions are used, along with any required synchronization fences surrounding it in order to achieve the proper behavior. Execution in this case is subject to the same restrictions as those built-in functions.

If there is no pattern or mechanism to provide a lock-free instruction sequence, a call is made to an external routine with the same parameters to be resolved at run time.

When implementing patterns for these built-in functions, the memory order parameter can be ignored as long as the pattern implements the most restrictive `__ATOMIC_SEQ_CST` memory order. Any of the other memory orders execute correctly with this memory order but they may not execute as efficiently as they could with a more appropriate implementation of the relaxed requirements.

Note that the C++11 standard allows for the memory order parameter to be determined at run time rather than at compile time. These built-in functions map any run-time value to `__ATOMIC_SEQ_CST` rather than invoke a runtime library call or inline a switch statement. This is standard compliant, safe, and the simplest approach for now.

The memory order parameter is a signed int, but only the lower 16 bits are reserved for the memory order. The remainder of the signed int is reserved for target use and should be 0. Use of the predefined atomic values ensures proper usage.

type `__atomic_load_n (`*type* `*ptr, int memorder)` [Built-in Function]
: This built-in function implements an atomic load operation. It returns the contents of `*`*ptr*.

  The valid memory order variants are `__ATOMIC_RELAXED`, `__ATOMIC_SEQ_CST`, `__ATOMIC_ACQUIRE`, and `__ATOMIC_CONSUME`.

`void __atomic_load (`*type* `*ptr,` *type* `*ret, int memorder)` [Built-in Function]
: This is the generic version of an atomic load. It returns the contents of `*`*ptr* in `*`*ret*.

`void __atomic_store_n (`*type* `*ptr,` *type* `val, int memorder)` [Built-in Function]
: This built-in function implements an atomic store operation. It writes *val* into `*`*ptr*.

  The valid memory order variants are `__ATOMIC_RELAXED`, `__ATOMIC_SEQ_CST`, and `__ATOMIC_RELEASE`.

`void __atomic_store (`*type* `*ptr,` *type* `*val, int memorder)` [Built-in Function]
: This is the generic version of an atomic store. It stores the value of `*`*val* into `*`*ptr*.

type __atomic_exchange_n (*type* *ptr, *type* val, int memorder) [Built-in Function]

This built-in function implements an atomic exchange operation. It writes *val* into **ptr*, and returns the previous contents of **ptr*.

All memory order variants are valid.

void __atomic_exchange (*type* *ptr, *type* *val, *type* *ret, int memorder) [Built-in Function]

This is the generic version of an atomic exchange. It stores the contents of **val* into **ptr*. The original value of **ptr* is copied into **ret*.

bool __atomic_compare_exchange_n (*type* *ptr, *type* *expected, *type* desired, bool weak, int success_memorder, int failure_memorder) [Built-in Function]

This built-in function implements an atomic compare and exchange operation. This compares the contents of **ptr* with the contents of **expected*. If equal, the operation is a *read-modify-write* operation that writes *desired* into **ptr*. If they are not equal, the operation is a *read* and the current contents of **ptr* are written into **expected*. *weak* is `true` for weak compare_exchange, which may fail spuriously, and `false` for the strong variation, which never fails spuriously. Many targets only offer the strong variation and ignore the parameter. When in doubt, use the strong variation.

If *desired* is written into **ptr* then `true` is returned and memory is affected according to the memory order specified by *success_memorder*. There are no restrictions on what memory order can be used here.

Otherwise, `false` is returned and memory is affected according to *failure_memorder*. This memory order cannot be `__ATOMIC_RELEASE` nor `__ATOMIC_ACQ_REL`. It also cannot be a stronger order than that specified by *success_memorder*.

bool __atomic_compare_exchange (*type* *ptr, *type* *expected, *type* *desired, bool weak, int success_memorder, int failure_memorder) [Built-in Function]

This built-in function implements the generic version of `__atomic_compare_exchange`. The function is virtually identical to `__atomic_compare_exchange_n`, except the desired value is also a pointer.

type __atomic_add_fetch (*type* *ptr, *type* val, int memorder) [Built-in Function]
type __atomic_sub_fetch (*type* *ptr, *type* val, int memorder) [Built-in Function]
type __atomic_and_fetch (*type* *ptr, *type* val, int memorder) [Built-in Function]
type __atomic_xor_fetch (*type* *ptr, *type* val, int memorder) [Built-in Function]
type __atomic_or_fetch (*type* *ptr, *type* val, int memorder) [Built-in Function]

type __atomic_nand_fetch (*type* *ptr, *type* val, int memorder) [Built-in Function]

These built-in functions perform the operation suggested by the name, and return the result of the operation. Operations on pointer arguments are performed as if the operands were of the `uintptr_t` type. That is, they are not scaled by the size of the type to which the pointer points.

```
{ *ptr op= val; return *ptr; }
{ *ptr = ~(*ptr & val); return *ptr; } // nand
```

The object pointed to by the first argument must be of integer or pointer type. It must not be a boolean type. All memory orders are valid.

type __atomic_fetch_add (*type* *ptr, *type* val, int memorder) [Built-in Function]

type __atomic_fetch_sub (*type* *ptr, *type* val, int memorder) [Built-in Function]

type __atomic_fetch_and (*type* *ptr, *type* val, int memorder) [Built-in Function]

type __atomic_fetch_xor (*type* *ptr, *type* val, int memorder) [Built-in Function]

type __atomic_fetch_or (*type* *ptr, *type* val, int memorder) [Built-in Function]

type __atomic_fetch_nand (*type* *ptr, *type* val, int memorder) [Built-in Function]

These built-in functions perform the operation suggested by the name, and return the value that had previously been in **ptr*. Operations on pointer arguments are performed as if the operands were of the `uintptr_t` type. That is, they are not scaled by the size of the type to which the pointer points.

```
{ tmp = *ptr; *ptr op= val; return tmp; }
{ tmp = *ptr; *ptr = ~(*ptr & val); return tmp; } // nand
```

The same constraints on arguments apply as for the corresponding `__atomic_op_fetch` built-in functions. All memory orders are valid.

bool __atomic_test_and_set (void *ptr, int memorder) [Built-in Function]

This built-in function performs an atomic test-and-set operation on the byte at **ptr*. The byte is set to some implementation defined nonzero "set" value and the return value is `true` if and only if the previous contents were "set". It should be only used for operands of type `bool` or `char`. For other types only part of the value may be set.

All memory orders are valid.

void __atomic_clear (bool *ptr, int memorder) [Built-in Function]

This built-in function performs an atomic clear operation on **ptr*. After the operation, **ptr* contains 0. It should be only used for operands of type `bool` or `char` and in conjunction with `__atomic_test_and_set`. For other types it may only clear partially. If the type is not `bool` prefer using `__atomic_store`.

The valid memory order variants are `__ATOMIC_RELAXED`, `__ATOMIC_SEQ_CST`, and `__ATOMIC_RELEASE`.

`void __atomic_thread_fence (int memorder)` [Built-in Function]
This built-in function acts as a synchronization fence between threads based on the specified memory order.

All memory orders are valid.

`void __atomic_signal_fence (int memorder)` [Built-in Function]
This built-in function acts as a synchronization fence between a thread and signal handlers based in the same thread.

All memory orders are valid.

`bool __atomic_always_lock_free (size_t size, void *ptr)` [Built-in Function]
This built-in function returns `true` if objects of *size* bytes always generate lock-free atomic instructions for the target architecture. *size* must resolve to a compile-time constant and the result also resolves to a compile-time constant.

ptr is an optional pointer to the object that may be used to determine alignment. A value of 0 indicates typical alignment should be used. The compiler may also ignore this parameter.

```
if (__atomic_always_lock_free (sizeof (long long), 0))
```

`bool __atomic_is_lock_free (size_t size, void *ptr)` [Built-in Function]
This built-in function returns `true` if objects of *size* bytes always generate lock-free atomic instructions for the target architecture. If the built-in function is not known to be lock-free, a call is made to a runtime routine named `__atomic_is_lock_free`.

ptr is an optional pointer to the object that may be used to determine alignment. A value of 0 indicates typical alignment should be used. The compiler may also ignore this parameter.

6.56 Built-in Functions to Perform Arithmetic with Overflow Checking

The following built-in functions allow performing simple arithmetic operations together with checking whether the operations overflowed.

`bool __builtin_add_overflow (type1 a, type2 b, type3 *res)` [Built-in Function]
`bool __builtin_sadd_overflow (int a, int b, int *res)` [Built-in Function]
`bool __builtin_saddl_overflow (long int a, long int b, long int *res)` [Built-in Function]
`bool __builtin_saddll_overflow (long long int a, long long int b, long long int *res)` [Built-in Function]
`bool __builtin_uadd_overflow (unsigned int a, unsigned int b, unsigned int *res)` [Built-in Function]
`bool __builtin_uaddl_overflow (unsigned long int a, unsigned long int b, unsigned long int *res)` [Built-in Function]

`bool __builtin_uaddll_overflow (unsigned long long int a, unsigned long long int b, unsigned long long int *res)` [Built-in Function]

These built-in functions promote the first two operands into infinite precision signed type and perform addition on those promoted operands. The result is then cast to the type the third pointer argument points to and stored there. If the stored result is equal to the infinite precision result, the built-in functions return `false`, otherwise they return `true`. As the addition is performed in infinite signed precision, these built-in functions have fully defined behavior for all argument values.

The first built-in function allows arbitrary integral types for operands and the result type must be pointer to some integral type other than enumerated or boolean type, the rest of the built-in functions have explicit integer types.

The compiler will attempt to use hardware instructions to implement these built-in functions where possible, like conditional jump on overflow after addition, conditional jump on carry etc.

`bool __builtin_sub_overflow (`*type1* `a, `*type2* `b, `*type3* `*res)` [Built-in Function]

`bool __builtin_ssub_overflow (int a, int b, int *res)` [Built-in Function]

`bool __builtin_ssubl_overflow (long int a, long int b, long int *res)` [Built-in Function]

`bool __builtin_ssubll_overflow (long long int a, long long int b, long long int *res)` [Built-in Function]

`bool __builtin_usub_overflow (unsigned int a, unsigned int b, unsigned int *res)` [Built-in Function]

`bool __builtin_usubl_overflow (unsigned long int a, unsigned long int b, unsigned long int *res)` [Built-in Function]

`bool __builtin_usubll_overflow (unsigned long long int a, unsigned long long int b, unsigned long long int *res)` [Built-in Function]

These built-in functions are similar to the add overflow checking built-in functions above, except they perform subtraction, subtract the second argument from the first one, instead of addition.

`bool __builtin_mul_overflow (`*type1* `a, `*type2* `b, `*type3* `*res)` [Built-in Function]

`bool __builtin_smul_overflow (int a, int b, int *res)` [Built-in Function]

`bool __builtin_smull_overflow (long int a, long int b, long int *res)` [Built-in Function]

`bool __builtin_smulll_overflow (long long int a, long long int b, long long int *res)` [Built-in Function]

`bool __builtin_umul_overflow (unsigned int a, unsigned int b, unsigned int *res)` [Built-in Function]

`bool __builtin_umull_overflow (unsigned long int a, unsigned long int b, unsigned long int *res)` [Built-in Function]

`bool __builtin_umulll_overflow (unsigned long long int a, unsigned long long int b, unsigned long long int *res)` [Built-in Function]

These built-in functions are similar to the add overflow checking built-in functions above, except they perform multiplication, instead of addition.

The following built-in functions allow checking if simple arithmetic operation would overflow.

`bool __builtin_add_overflow_p (type1 a, type2 b, type3 c)` [Built-in Function]

`bool __builtin_sub_overflow_p (type1 a, type2 b, type3 c)` [Built-in Function]

`bool __builtin_mul_overflow_p (type1 a, type2 b, type3 c)` [Built-in Function]

These built-in functions are similar to `__builtin_add_overflow`, `__builtin_sub_overflow`, or `__builtin_mul_overflow`, except that they don't store the result of the arithmetic operation anywhere and the last argument is not a pointer, but some expression with integral type other than enumerated or boolean type.

The built-in functions promote the first two operands into infinite precision signed type and perform addition on those promoted operands. The result is then cast to the type of the third argument. If the cast result is equal to the infinite precision result, the built-in functions return `false`, otherwise they return `true`. The value of the third argument is ignored, just the side effects in the third argument are evaluated, and no integral argument promotions are performed on the last argument. If the third argument is a bit-field, the type used for the result cast has the precision and signedness of the given bit-field, rather than precision and signedness of the underlying type.

For example, the following macro can be used to portably check, at compile-time, whether or not adding two constant integers will overflow, and perform the addition only when it is known to be safe and not to trigger a `-Woverflow` warning.

```
#define INT_ADD_OVERFLOW_P(a, b) \
   __builtin_add_overflow_p (a, b, (__typeof__ ((a) + (b))) 0)

enum {
    A = INT_MAX, B = 3,
    C = INT_ADD_OVERFLOW_P (A, B) ? 0 : A + B,
    D = __builtin_add_overflow_p (1, SCHAR_MAX, (signed char) 0)
};
```

The compiler will attempt to use hardware instructions to implement these built-in functions where possible, like conditional jump on overflow after addition, conditional jump on carry etc.

6.57 x86-Specific Memory Model Extensions for Transactional Memory

The x86 architecture supports additional memory ordering flags to mark critical sections for hardware lock elision. These must be specified in addition to an existing memory order to atomic intrinsics.

`__ATOMIC_HLE_ACQUIRE`
: Start lock elision on a lock variable. Memory order must be `__ATOMIC_ACQUIRE` or stronger.

`__ATOMIC_HLE_RELEASE`
: End lock elision on a lock variable. Memory order must be `__ATOMIC_RELEASE` or stronger.

When a lock acquire fails, it is required for good performance to abort the transaction quickly. This can be done with a `_mm_pause`.

```
#include <immintrin.h> // For _mm_pause

int lockvar;

/* Acquire lock with lock elision */
while (__atomic_exchange_n(&lockvar, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE))
    _mm_pause(); /* Abort failed transaction */
...
/* Free lock with lock elision */
__atomic_store_n(&lockvar, 0, __ATOMIC_RELEASE|__ATOMIC_HLE_RELEASE);
```

6.58 Object Size Checking

6.58.1 Object Size Checking Built-in Functions

GCC implements a limited buffer overflow protection mechanism that can prevent some buffer overflow attacks by determining the sizes of objects into which data is about to be written and preventing the writes when the size isn't sufficient. The built-in functions described below yield the best results when used together and when optimization is enabled. For example, to detect object sizes across function boundaries or to follow pointer assignments through non-trivial control flow they rely on various optimization passes enabled with `-O2`. However, to a limited extent, they can be used without optimization as well.

`size_t __builtin_object_size (const void * ptr, int type)` [Built-in Function]
: is a built-in construct that returns a constant number of bytes from *ptr* to the end of the object *ptr* pointer points to (if known at compile time). To determine the sizes of dynamically allocated objects the function relies on the allocation functions called to obtain the storage to be declared with the `alloc_size` attribute (see Section 6.33.1 [Common Function Attributes], page 569). `__builtin_object_size` never evaluates its arguments for side effects. If there are any side effects in them, it returns `(size_t) -1` for *type* 0 or 1 and `(size_t) 0` for *type* 2 or 3. If there are multiple objects *ptr* can point to and all of them are known at compile time, the returned number is the maximum of remaining byte counts in those objects if *type* & 2 is 0 and minimum if nonzero. If it is not possible to determine which objects *ptr* points to at compile time, `__builtin_object_size` should return `(size_t) -1` for *type* 0 or 1 and `(size_t) 0` for *type* 2 or 3.

  type is an integer constant from 0 to 3. If the least significant bit is clear, objects are whole variables, if it is set, a closest surrounding subobject is considered the object a pointer points to. The second bit determines if maximum or minimum of remaining bytes is computed.

```
struct V { char buf1[10]; int b; char buf2[10]; } var;
char *p = &var.buf1[1], *q = &var.b;

/* Here the object p points to is var.  */
assert (__builtin_object_size (p, 0) == sizeof (var) - 1);
/* The subobject p points to is var.buf1.  */
assert (__builtin_object_size (p, 1) == sizeof (var.buf1) - 1);
/* The object q points to is var.  */
assert (__builtin_object_size (q, 0)
        == (char *) (&var + 1) - (char *) &var.b);
/* The subobject q points to is var.b.  */
assert (__builtin_object_size (q, 1) == sizeof (var.b));
```

`size_t __builtin_dynamic_object_size (const void * ptr, int type)` [Built-in Function]

is similar to `__builtin_object_size` in that it returns a number of bytes from *ptr* to the end of the object *ptr* pointer points to, except that the size returned may not be a constant. This results in successful evaluation of object size estimates in a wider range of use cases and can be more precise than `__builtin_object_size`, but it incurs a performance penalty since it may add a runtime overhead on size computation. Semantics of *type* as well as return values in case it is not possible to determine which objects *ptr* points to at compile time are the same as in the case of `__builtin_object_size`.

6.58.2 Object Size Checking and Source Fortification

Hardening of function calls using the `_FORTIFY_SOURCE` macro is one of the key uses of the object size checking built-in functions. To make implementation of these features more convenient and improve optimization and diagnostics, there are built-in functions added for many common string operation functions, e.g., for `memcpy __builtin___memcpy_chk` built-in is provided. This built-in has an additional last argument, which is the number of bytes remaining in the object the *dest* argument points to or `(size_t) -1` if the size is not known.

The built-in functions are optimized into the normal string functions like `memcpy` if the last argument is `(size_t) -1` or if it is known at compile time that the destination object will not be overflowed. If the compiler can determine at compile time that the object will always be overflowed, it issues a warning.

The intended use can be e.g.

```
#undef memcpy
#define bos0(dest) __builtin_object_size (dest, 0)
#define memcpy(dest, src, n) \
  __builtin___memcpy_chk (dest, src, n, bos0 (dest))

char *volatile p;
char buf[10];
/* It is unknown what object p points to, so this is optimized
   into plain memcpy - no checking is possible.  */
memcpy (p, "abcde", n);
/* Destination is known and length too.  It is known at compile
   time there will be no overflow.  */
memcpy (&buf[5], "abcde", 5);
/* Destination is known, but the length is not known at compile time.
   This will result in __memcpy_chk call that can check for overflow
```

```
   at run time.  */
memcpy (&buf[5], "abcde", n);
/* Destination is known and it is known at compile time there will
   be overflow.  There will be a warning and __memcpy_chk call that
   will abort the program at run time.  */
memcpy (&buf[6], "abcde", 5);
```

Such built-in functions are provided for `memcpy`, `mempcpy`, `memmove`, `memset`, `strcpy`, `stpcpy`, `strncpy`, `strcat` and `strncat`.

6.58.2.1 Formatted Output Function Checking

`int __builtin___sprintf_chk (char *s, int flag, size_t os, const char *fmt, ...)` [Built-in Function]

`int __builtin___snprintf_chk (char *s, size_t maxlen, int flag, size_t os, const char *fmt, ...)` [Built-in Function]

`int __builtin___vsprintf_chk (char *s, int flag, size_t os, const char *fmt, va_list ap)` [Built-in Function]

`int __builtin___vsnprintf_chk (char *s, size_t maxlen, int flag, size_t os, const char *fmt, va_list ap)` [Built-in Function]

The added *flag* argument is passed unchanged to `__sprintf_chk` etc. functions and can contain implementation specific flags on what additional security measures the checking function might take, such as handling `%n` differently.

The *os* argument is the object size *s* points to, like in the other built-in functions. There is a small difference in the behavior though, if *os* is `(size_t) -1`, the built-in functions are optimized into the non-checking functions only if *flag* is 0, otherwise the checking function is called with *os* argument set to `(size_t) -1`.

In addition to this, there are checking built-in functions `__builtin___printf_chk`, `__builtin___vprintf_chk`, `__builtin___fprintf_chk` and `__builtin___vfprintf_chk`. These have just one additional argument, *flag*, right before format string *fmt*. If the compiler is able to optimize them to `fputc` etc. functions, it does, otherwise the checking function is called and the *flag* argument passed to it.

6.59 Other Built-in Functions Provided by GCC

GCC provides a large number of built-in functions other than the ones mentioned above. Some of these are for internal use in the processing of exceptions or variable-length argument lists and are not documented here because they may change from time to time; we do not recommend general use of these functions.

The remaining functions are provided for optimization purposes.

With the exception of built-ins that have library equivalents such as the standard C library functions discussed below, or that expand to library calls, GCC built-in functions are always expanded inline and thus do not have corresponding entry points and their address cannot be obtained. Attempting to use them in an expression other than a function call results in a compile-time error.

GCC includes built-in versions of many of the functions in the standard C library. These functions come in two forms: one whose names start with the `__builtin_` prefix, and the other without. Both forms have the same type (including prototype), the same address

(when their address is taken), and the same meaning as the C library functions even if you specify the `-fno-builtin` option see Section 3.4 [C Dialect Options], page 42). Many of these functions are only optimized in certain cases; if they are not optimized in a particular case, a call to the library function is emitted.

Outside strict ISO C mode (`-ansi`, `-std=c90`, `-std=c99` or `-std=c11`), the functions `_exit`, `alloca`, `bcmp`, `bzero`, `dcgettext`, `dgettext`, `dremf`, `dreml`, `drem`, `exp10f`, `exp10l`, `exp10`, `ffsll`, `ffsl`, `ffs`, `fprintf_unlocked`, `fputs_unlocked`, `gammaf`, `gammal`, `gamma`, `gammaf_r`, `gammal_r`, `gamma_r`, `gettext`, `index`, `isascii`, `j0f`, `j0l`, `j0`, `j1f`, `j1l`, `j1`, `jnf`, `jnl`, `jn`, `lgammaf_r`, `lgammal_r`, `lgamma_r`, `mempcpy`, `pow10f`, `pow10l`, `pow10`, `printf_unlocked`, `rindex`, `roundeven`, `roundevenf`, `roundevenl`, `scalbf`, `scalbl`, `scalb`, `signbit`, `signbitf`, `signbitl`, `signbitd32`, `signbitd64`, `signbitd128`, `significandf`, `significandl`, `significand`, `sincosf`, `sincosl`, `sincos`, `stpcpy`, `stpncpy`, `strcasecmp`, `strdup`, `strfmon`, `strncasecmp`, `strndup`, `strnlen`, `toascii`, `y0f`, `y0l`, `y0`, `y1f`, `y1l`, `y1`, `ynf`, `ynl` and `yn` may be handled as built-in functions. All these functions have corresponding versions prefixed with `__builtin_`, which may be used even in strict C90 mode.

The ISO C99 functions `_Exit`, `acoshf`, `acoshl`, `acosh`, `asinhf`, `asinhl`, `asinh`, `atanhf`, `atanhl`, `atanh`, `cabsf`, `cabsl`, `cabs`, `cacosf`, `cacoshf`, `cacoshl`, `cacosh`, `cacosl`, `cacos`, `cargf`, `cargl`, `carg`, `casinf`, `casinhf`, `casinhl`, `casinh`, `casinl`, `casin`, `catanf`, `catanhf`, `catanhl`, `catanh`, `catanl`, `catan`, `cbrtf`, `cbrtl`, `cbrt`, `ccosf`, `ccoshf`, `ccoshl`, `ccosh`, `ccosl`, `ccos`, `cexpf`, `cexpl`, `cexp`, `cimagf`, `cimagl`, `cimag`, `clogf`, `clogl`, `clog`, `conjf`, `conjl`, `conj`, `copysignf`, `copysignl`, `copysign`, `cpowf`, `cpowl`, `cpow`, `cprojf`, `cprojl`, `cproj`, `crealf`, `creall`, `creal`, `csinf`, `csinhf`, `csinhl`, `csinh`, `csinl`, `csin`, `csqrtf`, `csqrtl`, `csqrt`, `ctanf`, `ctanhf`, `ctanhl`, `ctanh`, `ctanl`, `ctan`, `erfcf`, `erfcl`, `erfc`, `erff`, `erfl`, `erf`, `exp2f`, `exp2l`, `exp2`, `expm1f`, `expm1l`, `expm1`, `fdimf`, `fdiml`, `fdim`, `fmaf`, `fmal`, `fmaxf`, `fmaxl`, `fmax`, `fma`, `fminf`, `fminl`, `fmin`, `hypotf`, `hypotl`, `hypot`, `ilogbf`, `ilogbl`, `ilogb`, `imaxabs`, `isblank`, `iswblank`, `lgammaf`, `lgammal`, `lgamma`, `llabs`, `llrintf`, `llrintl`, `llrint`, `llroundf`, `llroundl`, `llround`, `log1pf`, `log1pl`, `log1p`, `log2f`, `log2l`, `log2`, `logbf`, `logbl`, `logb`, `lrintf`, `lrintl`, `lrint`, `lroundf`, `lroundl`, `lround`, `nearbyintf`, `nearbyintl`, `nearbyint`, `nextafterf`, `nextafterl`, `nextafter`, `nexttowardf`, `nexttowardl`, `nexttoward`, `remainderf`, `remainderl`, `remainder`, `remquof`, `remquol`, `remquo`, `rintf`, `rintl`, `rint`, `roundf`, `roundl`, `round`, `scalblnf`, `scalblnl`, `scalbln`, `scalbnf`, `scalbnl`, `scalbn`, `snprintf`, `tgammaf`, `tgammal`, `tgamma`, `truncf`, `truncl`, `trunc`, `vfscanf`, `vscanf`, `vsnprintf` and `vsscanf` are handled as built-in functions except in strict ISO C90 mode (`-ansi` or `-std=c90`).

There are also built-in versions of the ISO C99 functions `acosf`, `acosl`, `asinf`, `asinl`, `atan2f`, `atan2l`, `atanf`, `atanl`, `ceilf`, `ceill`, `cosf`, `coshf`, `coshl`, `cosl`, `expf`, `expl`, `fabsf`, `fabsl`, `floorf`, `floorl`, `fmodf`, `fmodl`, `frexpf`, `frexpl`, `ldexpf`, `ldexpl`, `log10f`, `log10l`, `logf`, `logl`, `modfl`, `modff`, `powf`, `powl`, `sinf`, `sinhf`, `sinhl`, `sinl`, `sqrtf`, `sqrtl`, `tanf`, `tanhf`, `tanhl` and `tanl` that are recognized in any mode since ISO C90 reserves these names for the purpose to which ISO C99 puts them. All these functions have corresponding versions prefixed with `__builtin_`.

There are also built-in functions `__builtin_fabsf`*n*, `__builtin_fabsf`*n*`x`, `__builtin_copysignf`*n* and `__builtin_copysignf`*n*`x`, corresponding to the TS 18661-3 functions `fabsf`*n*, `fabsf`*n*`x`, `copysignf`*n* and `copysignf`*n*`x`, for supported types `_Float`*n* and `_Float`*n*`x`.

There are also GNU extension functions `clog10`, `clog10f` and `clog10l` which names are reserved by ISO C99 for future use. All these functions have versions prefixed with `__builtin_`.

The ISO C94 functions `iswalnum`, `iswalpha`, `iswcntrl`, `iswdigit`, `iswgraph`, `iswlower`, `iswprint`, `iswpunct`, `iswspace`, `iswupper`, `iswxdigit`, `towlower` and `towupper` are handled as built-in functions except in strict ISO C90 mode (`-ansi` or `-std=c90`).

The ISO C90 functions `abort`, `abs`, `acos`, `asin`, `atan2`, `atan`, `calloc`, `ceil`, `cosh`, `cos`, `exit`, `exp`, `fabs`, `floor`, `fmod`, `fprintf`, `fputs`, `free`, `frexp`, `fscanf`, `isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`, `tolower`, `toupper`, `labs`, `ldexp`, `log10`, `log`, `malloc`, `memchr`, `memcmp`, `memcpy`, `memset`, `modf`, `pow`, `printf`, `putchar`, `puts`, `realloc`, `scanf`, `sinh`, `sin`, `snprintf`, `sprintf`, `sqrt`, `sscanf`, `strcat`, `strchr`, `strcmp`, `strcpy`, `strcspn`, `strlen`, `strncat`, `strncmp`, `strncpy`, `strpbrk`, `strrchr`, `strspn`, `strstr`, `tanh`, `tan`, `vfprintf`, `vprintf` and `vsprintf` are all recognized as built-in functions unless `-fno-builtin` is specified (or `-fno-builtin-`*`function`* is specified for an individual function). All of these functions have corresponding versions prefixed with `__builtin_`.

GCC provides built-in versions of the ISO C99 floating-point comparison macros that avoid raising exceptions for unordered operands. They have the same names as the standard macros (`isgreater`, `isgreaterequal`, `isless`, `islessequal`, `islessgreater`, and `isunordered`) , with `__builtin_` prefixed. We intend for a library implementor to be able to simply `#define` each standard macro to its built-in equivalent. In the same fashion, GCC provides `fpclassify`, `isfinite`, `isinf_sign`, `isnormal` and `signbit` built-ins used with `__builtin_` prefixed. The `isinf` and `isnan` built-in functions appear both with and without the `__builtin_` prefix. With `-ffinite-math-only` option the `isinf` and `isnan` built-in functions will always return 0.

GCC provides built-in versions of the ISO C99 floating-point rounding and exceptions handling functions `fegetround`, `feclearexcept` and `feraiseexcept`. They may not be available for all targets, and because they need close interaction with libc internal values, they may not be available for all target libcs, but in all cases they will gracefully fallback to libc calls. These built-in functions appear both with and without the `__builtin_` prefix.

`void * __builtin_alloca (size_t size)` [Built-in Function]

The `__builtin_alloca` function must be called at block scope. The function allocates an object *size* bytes large on the stack of the calling function. The object is aligned on the default stack alignment boundary for the target determined by the `__BIGGEST_ALIGNMENT__` macro. The `__builtin_alloca` function returns a pointer to the first byte of the allocated object. The lifetime of the allocated object ends just before the calling function returns to its caller. This is so even when `__builtin_alloca` is called within a nested block.

For example, the following function allocates eight objects of `n` bytes each on the stack, storing a pointer to each in consecutive elements of the array `a`. It then passes the array to function `g` which can safely use the storage pointed to by each of the array elements.

```
void f (unsigned n)
{
  void *a [8];
  for (int i = 0; i != 8; ++i)
```

```
    a [i] = __builtin_alloca (n);

  g (a, n);   // safe
}
```

Since the `__builtin_alloca` function doesn't validate its argument it is the responsibility of its caller to make sure the argument doesn't cause it to exceed the stack size limit. The `__builtin_alloca` function is provided to make it possible to allocate on the stack arrays of bytes with an upper bound that may be computed at run time. Since C99 Variable Length Arrays offer similar functionality under a portable, more convenient, and safer interface they are recommended instead, in both C99 and C++ programs where GCC provides them as an extension. See Section 6.20 [Variable Length], page 561, for details.

`void * __builtin_alloca_with_align (size_t size, size_t alignment)` [Built-in Function]

The `__builtin_alloca_with_align` function must be called at block scope. The function allocates an object *size* bytes large on the stack of the calling function. The allocated object is aligned on the boundary specified by the argument *alignment* whose unit is given in bits (not bytes). The *size* argument must be positive and not exceed the stack size limit. The *alignment* argument must be a constant integer expression that evaluates to a power of 2 greater than or equal to `CHAR_BIT` and less than some unspecified maximum. Invocations with other values are rejected with an error indicating the valid bounds. The function returns a pointer to the first byte of the allocated object. The lifetime of the allocated object ends at the end of the block in which the function was called. The allocated storage is released no later than just before the calling function returns to its caller, but may be released at the end of the block in which the function was called.

For example, in the following function the call to `g` is unsafe because when `overalign` is non-zero, the space allocated by `__builtin_alloca_with_align` may have been released at the end of the `if` statement in which it was called.

```
void f (unsigned n, bool overalign)
{
  void *p;
  if (overalign)
    p = __builtin_alloca_with_align (n, 64 /* bits */);
  else
    p = __builtin_alloc (n);

  g (p, n);   // unsafe
}
```

Since the `__builtin_alloca_with_align` function doesn't validate its *size* argument it is the responsibility of its caller to make sure the argument doesn't cause it to exceed the stack size limit. The `__builtin_alloca_with_align` function is provided to make it possible to allocate on the stack overaligned arrays of bytes with an upper bound that may be computed at run time. Since C99 Variable Length Arrays offer the same functionality under a portable, more convenient, and safer interface they are recommended instead, in both C99 and C++ programs where GCC provides them as an extension. See Section 6.20 [Variable Length], page 561, for details.

`void * __builtin_alloca_with_align_and_max (size_t size, size_t alignment, size_t max_size)` [Built-in Function]

Similar to `__builtin_alloca_with_align` but takes an extra argument specifying an upper bound for *size* in case its value cannot be computed at compile time, for use by `-fstack-usage`, `-Wstack-usage` and `-Walloca-larger-than`. *max_size* must be a constant integer expression, it has no effect on code generation and no attempt is made to check its compatibility with *size*.

`bool __builtin_has_attribute (type-or-expression, attribute)` [Built-in Function]

The `__builtin_has_attribute` function evaluates to an integer constant expression equal to `true` if the symbol or type referenced by the *type-or-expression* argument has been declared with the *attribute* referenced by the second argument. For an *type-or-expression* argument that does not reference a symbol, since attributes do not apply to expressions the built-in consider the type of the argument. Neither argument is evaluated. The *type-or-expression* argument is subject to the same restrictions as the argument to `typeof` (see Section 6.7 [Typeof], page 549). The *attribute* argument is an attribute name optionally followed by a comma-separated list of arguments enclosed in parentheses. Both forms of attribute names—with and without double leading and trailing underscores—are recognized. See Section 6.39 [Attribute Syntax], page 661, for details. When no attribute arguments are specified for an attribute that expects one or more arguments the function returns `true` if *type-or-expression* has been declared with the attribute regardless of the attribute argument values. Arguments provided for an attribute that expects some are validated and matched up to the provided number. The function returns `true` if all provided arguments match. For example, the first call to the function below evaluates to `true` because `x` is declared with the `aligned` attribute but the second call evaluates to `false` because `x` is declared `aligned (8)` and not `aligned (4)`.

```
__attribute__ ((aligned (8))) int x;
_Static_assert (__builtin_has_attribute (x, aligned), "aligned");
_Static_assert (!__builtin_has_attribute (x, aligned (4)), "aligned (4)");
```

Due to a limitation the `__builtin_has_attribute` function returns `false` for the `mode` attribute even if the type or variable referenced by the *type-or-expression* argument was declared with one. The function is also not supported with labels, and in C with enumerators.

Note that unlike the `__has_attribute` preprocessor operator which is suitable for use in `#if` preprocessing directives `__builtin_has_attribute` is an intrinsic function that is not recognized in such contexts.

type `__builtin_speculation_safe_value (type val, type failval)` [Built-in Function]

This built-in function can be used to help mitigate against unsafe speculative execution. *type* may be any integral type or any pointer type.

1. If the CPU is not speculatively executing the code, then *val* is returned.
2. If the CPU is executing speculatively then either:

- The function may cause execution to pause until it is known that the code is no-longer being executed speculatively (in which case *val* can be returned, as above); or
- The function may use target-dependent speculation tracking state to cause *failval* to be returned when it is known that speculative execution has incorrectly predicted a conditional branch operation.

The second argument, *failval*, is optional and defaults to zero if omitted.

GCC defines the preprocessor macro `__HAVE_BUILTIN_SPECULATION_SAFE_VALUE` for targets that have been updated to support this builtin.

The built-in function can be used where a variable appears to be used in a safe way, but the CPU, due to speculative execution may temporarily ignore the bounds checks. Consider, for example, the following function:

```
int array[500];
int f (unsigned untrusted_index)
{
  if (untrusted_index < 500)
    return array[untrusted_index];
  return 0;
}
```

If the function is called repeatedly with `untrusted_index` less than the limit of 500, then a branch predictor will learn that the block of code that returns a value stored in `array` will be executed. If the function is subsequently called with an out-of-range value it will still try to execute that block of code first until the CPU determines that the prediction was incorrect (the CPU will unwind any incorrect operations at that point). However, depending on how the result of the function is used, it might be possible to leave traces in the cache that can reveal what was stored at the out-of-bounds location. The built-in function can be used to provide some protection against leaking data in this way by changing the code to:

```
int array[500];
int f (unsigned untrusted_index)
{
  if (untrusted_index < 500)
    return array[__builtin_speculation_safe_value (untrusted_index)];
  return 0;
}
```

The built-in function will either cause execution to stall until the conditional branch has been fully resolved, or it may permit speculative execution to continue, but using 0 instead of `untrusted_value` if that exceeds the limit.

If accessing any memory location is potentially unsafe when speculative execution is incorrect, then the code can be rewritten as

```
int array[500];
int f (unsigned untrusted_index)
{
  if (untrusted_index < 500)
    return *__builtin_speculation_safe_value (&array[untrusted_index], NULL);
  return 0;
}
```

which will cause a `NULL` pointer to be used for the unsafe case.

`int __builtin_types_compatible_p (`*type1*`, `*type2*`)` [Built-in Function]

You can use the built-in function `__builtin_types_compatible_p` to determine whether two types are the same.

This built-in function returns 1 if the unqualified versions of the types *type1* and *type2* (which are types, not expressions) are compatible, 0 otherwise. The result of this built-in function can be used in integer constant expressions.

This built-in function ignores top level qualifiers (e.g., `const`, `volatile`). For example, `int` is equivalent to `const int`.

The type `int[]` and `int[5]` are compatible. On the other hand, `int` and `char *` are not compatible, even if the size of their types, on the particular architecture are the same. Also, the amount of pointer indirection is taken into account when determining similarity. Consequently, `short *` is not similar to `short **`. Furthermore, two types that are typedefed are considered compatible if their underlying types are compatible.

An `enum` type is not considered to be compatible with another `enum` type even if both are compatible with the same integer type; this is what the C standard specifies. For example, `enum {foo, bar}` is not similar to `enum {hot, dog}`.

You typically use this function in code whose execution varies depending on the arguments' types. For example:

```
#define foo(x)                                                  \
  ({                                                           \
    typeof (x) tmp = (x);                                      \
    if (__builtin_types_compatible_p (typeof (x), long double)) \
      tmp = foo_long_double (tmp);                             \
    else if (__builtin_types_compatible_p (typeof (x), double)) \
      tmp = foo_double (tmp);                                  \
    else if (__builtin_types_compatible_p (typeof (x), float))  \
      tmp = foo_float (tmp);                                   \
    else                                                       \
      abort ();                                                \
    tmp;                                                       \
  })
```

Note: This construct is only available for C.

type `__builtin_call_with_static_chain (`*call_exp*`, `*pointer_exp*`)` [Built-in Function]

The *call_exp* expression must be a function call, and the *pointer_exp* expression must be a pointer. The *pointer_exp* is passed to the function call in the target's static chain location. The result of builtin is the result of the function call.

Note: This builtin is only available for C. This builtin can be used to call Go closures from C.

type `__builtin_choose_expr (`*const_exp*`, `*exp1*`, `*exp2*`)` [Built-in Function]

You can use the built-in function `__builtin_choose_expr` to evaluate code depending on the value of a constant expression. This built-in function returns *exp1* if *const_exp*, which is an integer constant expression, is nonzero. Otherwise it returns *exp2*.

This built-in function is analogous to the '`? :`' operator in C, except that the expression returned has its type unaltered by promotion rules. Also, the built-in function

does not evaluate the expression that is not chosen. For example, if *const_exp* evaluates to `true`, *exp2* is not evaluated even if it has side effects.

This built-in function can return an lvalue if the chosen argument is an lvalue.

If *exp1* is returned, the return type is the same as *exp1*'s type. Similarly, if *exp2* is returned, its return type is the same as *exp2*.

Example:

```
#define foo(x)                                                  \
  __builtin_choose_expr (                                       \
    __builtin_types_compatible_p (typeof (x), double),          \
    foo_double (x),                                             \
    __builtin_choose_expr (                                     \
      __builtin_types_compatible_p (typeof (x), float),         \
      foo_float (x),                                            \
      /* The void expression results in a compile-time error    \
         when assigning the result to something.  */            \
      (void)0))
```

Note: This construct is only available for C. Furthermore, the unused expression (*exp1* or *exp2* depending on the value of *const_exp*) may still generate syntax errors. This may change in future revisions.

type `__builtin_tgmath` (*functions, arguments*) [Built-in Function]

The built-in function `__builtin_tgmath`, available only for C and Objective-C, calls a function determined according to the rules of `<tgmath.h>` macros. It is intended to be used in implementations of that header, so that expansions of macros from that header only expand each of their arguments once, to avoid problems when calls to such macros are nested inside the arguments of other calls to such macros; in addition, it results in better diagnostics for invalid calls to `<tgmath.h>` macros than implementations using other GNU C language features. For example, the `pow` type-generic macro might be defined as:

```
#define pow(a, b) __builtin_tgmath (powf, pow, powl, \
                                    cpowf, cpow, cpowl, a, b)
```

The arguments to `__builtin_tgmath` are at least two pointers to functions, followed by the arguments to the type-generic macro (which will be passed as arguments to the selected function). All the pointers to functions must be pointers to prototyped functions, none of which may have variable arguments, and all of which must have the same number of parameters; the number of parameters of the first function determines how many arguments to `__builtin_tgmath` are interpreted as function pointers, and how many as the arguments to the called function.

The types of the specified functions must all be different, but related to each other in the same way as a set of functions that may be selected between by a macro in `<tgmath.h>`. This means that the functions are parameterized by a floating-point type *t*, different for each such function. The function return types may all be the same type, or they may be *t* for each function, or they may be the real type corresponding to *t* for each function (if some of the types *t* are complex). Likewise, for each parameter position, the type of the parameter in that position may always be the same type, or may be *t* for each function (this case must apply for at least one parameter position), or may be the real type corresponding to *t* for each function.

The standard rules for `<tgmath.h>` macros are used to find a common type *u* from the types of the arguments for parameters whose types vary between the functions; complex integer types (a GNU extension) are treated like the complex type corresponding to the real floating type that would be chosen for the corresponding real integer type. If the function return types vary, or are all the same integer type, the function called is the one for which *t* is *u*, and it is an error if there is no such function. If the function return types are all the same floating-point type, the type-generic macro is taken to be one of those from TS 18661 that rounds the result to a narrower type; if there is a function for which *t* is *u*, it is called, and otherwise the first function, if any, for which *t* has at least the range and precision of *u* is called, and it is an error if there is no such function.

`int __builtin_constant_p (`*exp*`)` [Built-in Function]

You can use the built-in function `__builtin_constant_p` to determine if a value is known to be constant at compile time and hence that GCC can perform constant-folding on expressions involving that value. The argument of the function is the value to test. The function returns the integer 1 if the argument is known to be a compile-time constant and 0 if it is not known to be a compile-time constant. A return of 0 does not indicate that the value is *not* a constant, but merely that GCC cannot prove it is a constant with the specified value of the `-O` option.

You typically use this function in an embedded application where memory is a critical resource. If you have some complex calculation, you may want it to be folded if it involves constants, but need to call a function if it does not. For example:

```
#define Scale_Value(X)      \
  (__builtin_constant_p (X) \
  ? ((X) * SCALE + OFFSET) : Scale (X))
```

You may use this built-in function in either a macro or an inline function. However, if you use it in an inlined function and pass an argument of the function as the argument to the built-in, GCC never returns 1 when you call the inline function with a string constant or compound literal (see Section 6.28 [Compound Literals], page 564) and does not return 1 when you pass a constant numeric value to the inline function unless you specify the `-O` option.

You may also use `__builtin_constant_p` in initializers for static data. For instance, you can write

```
static const int table[] = {
   __builtin_constant_p (EXPRESSION) ? (EXPRESSION) : -1,
   /* ... */
};
```

This is an acceptable initializer even if *EXPRESSION* is not a constant expression, including the case where `__builtin_constant_p` returns 1 because *EXPRESSION* can be folded to a constant but *EXPRESSION* contains operands that are not otherwise permitted in a static initializer (for example, `0 && foo ()`). GCC must be more conservative about evaluating the built-in in this case, because it has no opportunity to perform optimization.

`bool __builtin_is_constant_evaluated (void)` [Built-in Function]

The `__builtin_is_constant_evaluated` function is available only in C++. The built-in is intended to be used by implementations of the `std::is_constant_`

`evaluated` C++ function. Programs should make use of the latter function rather than invoking the built-in directly.

The main use case of the built-in is to determine whether a `constexpr` function is being called in a `constexpr` context. A call to the function evaluates to a core constant expression with the value `true` if and only if it occurs within the evaluation of an expression or conversion that is manifestly constant-evaluated as defined in the C++ standard. Manifestly constant-evaluated contexts include constant-expressions, the conditions of `constexpr if` statements, constraint-expressions, and initializers of variables usable in constant expressions. For more details refer to the latest revision of the C++ standard.

`void __builtin_clear_padding (`*ptr*`)` [Built-in Function]

The built-in function `__builtin_clear_padding` function clears padding bits inside of the object representation of object pointed by *ptr*, which has to be a pointer. The value representation of the object is not affected. The type of the object is assumed to be the type the pointer points to. Inside of a union, the only cleared bits are bits that are padding bits for all the union members.

This built-in-function is useful if the padding bits of an object might have indeterminate values and the object representation needs to be bitwise compared to some other object, for example for atomic operations.

For C++, *ptr* argument type should be pointer to trivially-copyable type, unless the argument is address of a variable or parameter, because otherwise it isn't known if the type isn't just a base class whose padding bits are reused or laid out differently in a derived class.

type `__builtin_bit_cast (`*type*`, `*arg*`)` [Built-in Function]

The `__builtin_bit_cast` function is available only in C++. The built-in is intended to be used by implementations of the `std::bit_cast` C++ template function. Programs should make use of the latter function rather than invoking the built-in directly.

This built-in function allows reinterpreting the bits of the *arg* argument as if it had type *type*. *type* and the type of the *arg* argument need to be trivially copyable types with the same size. When manifestly constant-evaluated, it performs extra diagnostics required for `std::bit_cast` and returns a constant expression if *arg* is a constant expression. For more details refer to the latest revision of the C++ standard.

`long __builtin_expect (long `*exp*`, long `*c*`)` [Built-in Function]

You may use `__builtin_expect` to provide the compiler with branch prediction information. In general, you should prefer to use actual profile feedback for this (`-fprofile-arcs`), as programmers are notoriously bad at predicting how their programs actually perform. However, there are applications in which this data is hard to collect.

The return value is the value of *exp*, which should be an integral expression. The semantics of the built-in are that it is expected that *exp* == *c*. For example:

```
if (__builtin_expect (x, 0))
  foo ();
```

indicates that we do not expect to call `foo`, since we expect `x` to be zero. Since you are limited to integral expressions for *exp*, you should use constructions such as

```
if (__builtin_expect (ptr != NULL, 1))
```

```
foo (*ptr);
```

when testing pointer or floating-point values.

For the purposes of branch prediction optimizations, the probability that a `__builtin_expect` expression is `true` is controlled by GCC's `builtin-expect-probability` parameter, which defaults to 90%.

You can also use `__builtin_expect_with_probability` to explicitly assign a probability value to individual expressions. If the built-in is used in a loop construct, the provided probability will influence the expected number of iterations made by loop optimizations.

`long __builtin_expect_with_probability` [Built-in Function]
(long *exp*, long *c*, double *probability*)

This function has the same semantics as `__builtin_expect`, but the caller provides the expected probability that *exp* == *c*. The last argument, *probability*, is a floating-point value in the range 0.0 to 1.0, inclusive. The *probability* argument must be constant floating-point expression.

`void __builtin_trap (void)` [Built-in Function]

This function causes the program to exit abnormally. GCC implements this function by using a target-dependent mechanism (such as intentionally executing an illegal instruction) or by calling `abort`. The mechanism used may vary from release to release so you should not rely on any particular implementation.

`void __builtin_unreachable (void)` [Built-in Function]

If control flow reaches the point of the `__builtin_unreachable`, the program is undefined. It is useful in situations where the compiler cannot deduce the unreachability of the code.

One such case is immediately following an `asm` statement that either never terminates, or one that transfers control elsewhere and never returns. In this example, without the `__builtin_unreachable`, GCC issues a warning that control reaches the end of a non-void function. It also generates code to return after the `asm`.

```
int f (int c, int v)
{
  if (c)
    {
      return v;
    }
  else
    {
      asm("jmp error_handler");
      __builtin_unreachable ();
    }
}
```

Because the `asm` statement unconditionally transfers control out of the function, control never reaches the end of the function body. The `__builtin_unreachable` is in fact unreachable and communicates this fact to the compiler.

Another use for `__builtin_unreachable` is following a call a function that never returns but that is not declared `__attribute__((noreturn))`, as in this example:

```
void function_that_never_returns (void);
```

```
int g (int c)
{
  if (c)
    {
      return 1;
    }
  else
    {
      function_that_never_returns ();
      __builtin_unreachable ();
    }
}
```

type `__builtin_assoc_barrier (`*type expr*`)` [Built-in Function]

This built-in inhibits re-association of the floating-point expression *expr* with expressions consuming the return value of the built-in. The expression *expr* itself can be reordered, and the whole expression *expr* can be reordered with operands after the barrier. The barrier is only relevant when `-fassociative-math` is active, since otherwise floating-point is not treated as associative.

```
float x0 = a + b - b;
float x1 = __builtin_assoc_barrier(a + b) - b;
```

means that, with `-fassociative-math`, `x0` can be optimized to `x0 = a` but `x1` cannot.

`void * __builtin_assume_aligned (const void *`*exp*`,` [Built-in Function]
`size_t` *align*`, ...)`

This function returns its first argument, and allows the compiler to assume that the returned pointer is at least *align* bytes aligned. This built-in can have either two or three arguments, if it has three, the third argument should have integer type, and if it is nonzero means misalignment offset. For example:

```
void *x = __builtin_assume_aligned (arg, 16);
```

means that the compiler can assume `x`, set to `arg`, is at least 16-byte aligned, while:

```
void *x = __builtin_assume_aligned (arg, 32, 8);
```

means that the compiler can assume for `x`, set to `arg`, that `(char *) x - 8` is 32-byte aligned.

`int __builtin_LINE ()` [Built-in Function]

This function is the equivalent of the preprocessor `__LINE__` macro and returns a constant integer expression that evaluates to the line number of the invocation of the built-in. When used as a C++ default argument for a function *F*, it returns the line number of the call to *F*.

`const char * __builtin_FUNCTION ()` [Built-in Function]

This function is the equivalent of the `__FUNCTION__` symbol and returns an address constant pointing to the name of the function from which the built-in was invoked, or the empty string if the invocation is not at function scope. When used as a C++ default argument for a function *F*, it returns the name of *F*'s caller or the empty string if the call was not made at function scope.

`const char * __builtin_FILE ()` [Built-in Function]

This function is the equivalent of the preprocessor `__FILE__` macro and returns an address constant pointing to the file name containing the invocation of the built-in, or the empty string if the invocation is not at function scope. When used as a C++ default argument for a function *F*, it returns the file name of the call to *F* or the empty string if the call was not made at function scope.

For example, in the following, each call to function `foo` will print a line similar to `"file.c:123: foo: message"` with the name of the file and the line number of the `printf` call, the name of the function `foo`, followed by the word `message`.

```
const char*
function (const char *func = __builtin_FUNCTION ())
{
  return func;
}

void foo (void)
{
  printf ("%s:%i: %s: message\n", file (), line (), function ());
}
```

`void __builtin___clear_cache (void *begin, void *end)` [Built-in Function]

This function is used to flush the processor's instruction cache for the region of memory between *begin* inclusive and *end* exclusive. Some targets require that the instruction cache be flushed, after modifying memory containing code, in order to obtain deterministic behavior.

If the target does not require instruction cache flushes, `__builtin___clear_cache` has no effect. Otherwise either instructions are emitted in-line to clear the instruction cache or a call to the `__clear_cache` function in libgcc is made.

`void __builtin_prefetch (const void *addr, ...)` [Built-in Function]

This function is used to minimize cache-miss latency by moving data into a cache before it is accessed. You can insert calls to `__builtin_prefetch` into code for which you know addresses of data in memory that is likely to be accessed soon. If the target supports them, data prefetch instructions are generated. If the prefetch is done early enough before the access then the data will be in the cache by the time it is accessed.

The value of *addr* is the address of the memory to prefetch. There are two optional arguments, *rw* and *locality*. The value of *rw* is a compile-time constant one or zero; one means that the prefetch is preparing for a write to the memory address and zero, the default, means that the prefetch is preparing for a read. The value *locality* must be a compile-time constant integer between zero and three. A value of zero means that the data has no temporal locality, so it need not be left in the cache after the access. A value of three means that the data has a high degree of temporal locality and should be left in all levels of cache possible. Values of one and two mean, respectively, a low or moderate degree of temporal locality. The default is three.

```
for (i = 0; i < n; i++)
  {
    a[i] = a[i] + b[i];
```

```
    __builtin_prefetch (&a[i+j], 1, 1);
    __builtin_prefetch (&b[i+j], 0, 1);
    /* ... */
  }
```

Data prefetch does not generate faults if *addr* is invalid, but the address expression itself must be valid. For example, a prefetch of `p->next` does not fault if `p->next` is not a valid address, but evaluation faults if `p` is not a valid address.

If the target does not support data prefetch, the address expression is evaluated if it includes side effects but no other code is generated and GCC does not issue a warning.

`size_t __builtin_object_size (const void * ptr, int type)` [Built-in Function]

Returns a constant size estimate of an object pointed to by *ptr*. See Section 6.58 [Object Size Checking], page 740, for a detailed description of the function.

`size_t __builtin_dynamic_object_size (const void * ptr, int type)` [Built-in Function]

Similar to `__builtin_object_size` except that the return value need not be a constant. See Section 6.58 [Object Size Checking], page 740, for a detailed description of the function.

`double __builtin_huge_val (void)` [Built-in Function]

Returns a positive infinity, if supported by the floating-point format, else `DBL_MAX`. This function is suitable for implementing the ISO C macro `HUGE_VAL`.

`float __builtin_huge_valf (void)` [Built-in Function]

Similar to `__builtin_huge_val`, except the return type is `float`.

`long double __builtin_huge_vall (void)` [Built-in Function]

Similar to `__builtin_huge_val`, except the return type is `long double`.

`_Floatn __builtin_huge_valfn (void)` [Built-in Function]

Similar to `__builtin_huge_val`, except the return type is `_Floatn`.

`_Floatnx __builtin_huge_valfnx (void)` [Built-in Function]

Similar to `__builtin_huge_val`, except the return type is `_Floatnx`.

`int __builtin_fpclassify (int, int, int, int, int, ...)` [Built-in Function]

This built-in implements the C99 fpclassify functionality. The first five int arguments should be the target library's notion of the possible FP classes and are used for return values. They must be constant values and they must appear in this order: `FP_NAN`, `FP_INFINITE`, `FP_NORMAL`, `FP_SUBNORMAL` and `FP_ZERO`. The ellipsis is for exactly one floating-point value to classify. GCC treats the last argument as type-generic, which means it does not do default promotion from float to double.

`double __builtin_inf (void)` [Built-in Function]

Similar to `__builtin_huge_val`, except a warning is generated if the target floating-point format does not support infinities.

`_Decimal32 __builtin_infd32 (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `_Decimal32`.

`_Decimal64 __builtin_infd64 (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `_Decimal64`.

`_Decimal128 __builtin_infd128 (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `_Decimal128`.

`float __builtin_inff (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `float`. This function is suitable for implementing the ISO C99 macro `INFINITY`.

`long double __builtin_infl (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `long double`.

`_Floatn __builtin_inffn (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `_Floatn`.

`_Floatn __builtin_inffnx (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `_Floatnx`.

`int __builtin_isinf_sign (...)` [Built-in Function]
Similar to `isinf`, except the return value is -1 for an argument of `-Inf` and 1 for an argument of `+Inf`. Note while the parameter list is an ellipsis, this function only accepts exactly one floating-point argument. GCC treats this parameter as type-generic, which means it does not do default promotion from float to double.

`double __builtin_nan (const char *str)` [Built-in Function]
This is an implementation of the ISO C99 function `nan`.

Since ISO C99 defines this function in terms of `strtod`, which we do not implement, a description of the parsing is in order. The string is parsed as by `strtol`; that is, the base is recognized by leading '`0`' or '`0x`' prefixes. The number parsed is placed in the significand such that the least significant bit of the number is at the least significant bit of the significand. The number is truncated to fit the significand field provided. The significand is forced to be a quiet NaN.

This function, if given a string literal all of which would have been consumed by `strtol`, is evaluated early enough that it is considered a compile-time constant.

`_Decimal32 __builtin_nand32 (const char *str)` [Built-in Function]
Similar to `__builtin_nan`, except the return type is `_Decimal32`.

`_Decimal64 __builtin_nand64 (const char *str)` [Built-in Function]
Similar to `__builtin_nan`, except the return type is `_Decimal64`.

`_Decimal128 __builtin_nand128 (const char *str)` [Built-in Function]
Similar to `__builtin_nan`, except the return type is `_Decimal128`.

`float __builtin_nanf (const char *str)` [Built-in Function]
Similar to `__builtin_nan`, except the return type is `float`.

`long double __builtin_nanl (const char *str)` [Built-in Function]
Similar to `__builtin_nan`, except the return type is `long double`.

`_Floatn __builtin_nanfn (const char *str)` [Built-in Function]
Similar to `__builtin_nan`, except the return type is `_Floatn`.

`_Floatnx __builtin_nanfnx (const char *str)` [Built-in Function]
Similar to `__builtin_nan`, except the return type is `_Floatnx`.

`double __builtin_nans (const char *str)` [Built-in Function]
Similar to `__builtin_nan`, except the significand is forced to be a signaling NaN. The `nans` function is proposed by WG14 N965.

`_Decimal32 __builtin_nansd32 (const char *str)` [Built-in Function]
Similar to `__builtin_nans`, except the return type is `_Decimal32`.

`_Decimal64 __builtin_nansd64 (const char *str)` [Built-in Function]
Similar to `__builtin_nans`, except the return type is `_Decimal64`.

`_Decimal128 __builtin_nansd128 (const char *str)` [Built-in Function]
Similar to `__builtin_nans`, except the return type is `_Decimal128`.

`float __builtin_nansf (const char *str)` [Built-in Function]
Similar to `__builtin_nans`, except the return type is `float`.

`long double __builtin_nansl (const char *str)` [Built-in Function]
Similar to `__builtin_nans`, except the return type is `long double`.

`_Floatn __builtin_nansfn (const char *str)` [Built-in Function]
Similar to `__builtin_nans`, except the return type is `_Floatn`.

`_Floatnx __builtin_nansfnx (const char *str)` [Built-in Function]
Similar to `__builtin_nans`, except the return type is `_Floatnx`.

`int __builtin_issignaling (...)` [Built-in Function]
Return non-zero if the argument is a signaling NaN and zero otherwise. Note while the parameter list is an ellipsis, this function only accepts exactly one floating-point argument. GCC treats this parameter as type-generic, which means it does not do default promotion from float to double. This built-in function can work even without the non-default `-fsignaling-nans` option, although if a signaling NaN is computed, stored or passed as argument to some function other than this built-in in the current translation unit, it is safer to use `-fsignaling-nans`. With `-ffinite-math-only` option this built-in function will always return 0.

`int __builtin_ffs (int x)` [Built-in Function]
Returns one plus the index of the least significant 1-bit of x, or if x is zero, returns zero.

`int __builtin_clz (unsigned int x)` [Built-in Function]
Returns the number of leading 0-bits in x, starting at the most significant bit position. If x is 0, the result is undefined.

`int __builtin_ctz (unsigned int x)` [Built-in Function]
Returns the number of trailing 0-bits in *x*, starting at the least significant bit position. If *x* is 0, the result is undefined.

`int __builtin_clrsb (int x)` [Built-in Function]
Returns the number of leading redundant sign bits in *x*, i.e. the number of bits following the most significant bit that are identical to it. There are no special cases for 0 or other values.

`int __builtin_popcount (unsigned int x)` [Built-in Function]
Returns the number of 1-bits in *x*.

`int __builtin_parity (unsigned int x)` [Built-in Function]
Returns the parity of *x*, i.e. the number of 1-bits in *x* modulo 2.

`int __builtin_ffsl (long)` [Built-in Function]
Similar to `__builtin_ffs`, except the argument type is `long`.

`int __builtin_clzl (unsigned long)` [Built-in Function]
Similar to `__builtin_clz`, except the argument type is `unsigned long`.

`int __builtin_ctzl (unsigned long)` [Built-in Function]
Similar to `__builtin_ctz`, except the argument type is `unsigned long`.

`int __builtin_clrsbl (long)` [Built-in Function]
Similar to `__builtin_clrsb`, except the argument type is `long`.

`int __builtin_popcountl (unsigned long)` [Built-in Function]
Similar to `__builtin_popcount`, except the argument type is `unsigned long`.

`int __builtin_parityl (unsigned long)` [Built-in Function]
Similar to `__builtin_parity`, except the argument type is `unsigned long`.

`int __builtin_ffsll (long long)` [Built-in Function]
Similar to `__builtin_ffs`, except the argument type is `long long`.

`int __builtin_clzll (unsigned long long)` [Built-in Function]
Similar to `__builtin_clz`, except the argument type is `unsigned long long`.

`int __builtin_ctzll (unsigned long long)` [Built-in Function]
Similar to `__builtin_ctz`, except the argument type is `unsigned long long`.

`int __builtin_clrsbll (long long)` [Built-in Function]
Similar to `__builtin_clrsb`, except the argument type is `long long`.

`int __builtin_popcountll (unsigned long long)` [Built-in Function]
Similar to `__builtin_popcount`, except the argument type is `unsigned long long`.

`int __builtin_parityll (unsigned long long)` [Built-in Function]
Similar to `__builtin_parity`, except the argument type is `unsigned long long`.

`double __builtin_powi (double, int)` [Built-in Function]
`float __builtin_powif (float, int)` [Built-in Function]
`long double __builtin_powil (long double, int)` [Built-in Function]
Returns the first argument raised to the power of the second. Unlike the `pow` function no guarantees about precision and rounding are made.

`uint16_t __builtin_bswap16 (uint16_t x)` [Built-in Function]
Returns *x* with the order of the bytes reversed; for example, `0xaabb` becomes `0xbbaa`. Byte here always means exactly 8 bits.

`uint32_t __builtin_bswap32 (uint32_t x)` [Built-in Function]
Similar to `__builtin_bswap16`, except the argument and return types are 32-bit.

`uint64_t __builtin_bswap64 (uint64_t x)` [Built-in Function]
Similar to `__builtin_bswap32`, except the argument and return types are 64-bit.

`uint128_t __builtin_bswap128 (uint128_t x)` [Built-in Function]
Similar to `__builtin_bswap64`, except the argument and return types are 128-bit. Only supported on targets when 128-bit types are supported.

`Pmode __builtin_extend_pointer (void * x)` [Built-in Function]
On targets where the user visible pointer size is smaller than the size of an actual hardware address this function returns the extended user pointer. Targets where this is true included ILP32 mode on x86_64 or Aarch64. This function is mainly useful when writing inline assembly code.

`int __builtin_goacc_parlevel_id (int x)` [Built-in Function]
Returns the openacc gang, worker or vector id depending on whether *x* is 0, 1 or 2.

`int __builtin_goacc_parlevel_size (int x)` [Built-in Function]
Returns the openacc gang, worker or vector size depending on whether *x* is 0, 1 or 2.

6.60 Built-in Functions Specific to Particular Target Machines

On some target machines, GCC supports many built-in functions specific to those machines. Generally these generate calls to specific machine instructions, but allow the compiler to schedule those calls.

6.60.1 AArch64 Built-in Functions

These built-in functions are available for the AArch64 family of processors.

```
unsigned int __builtin_aarch64_get_fpcr ();
void __builtin_aarch64_set_fpcr (unsigned int);
unsigned int __builtin_aarch64_get_fpsr ();
void __builtin_aarch64_set_fpsr (unsigned int);

unsigned long long __builtin_aarch64_get_fpcr64 ();
void __builtin_aarch64_set_fpcr64 (unsigned long long);
unsigned long long __builtin_aarch64_get_fpsr64 ();
void __builtin_aarch64_set_fpsr64 (unsigned long long);
```

6.60.2 Alpha Built-in Functions

These built-in functions are available for the Alpha family of processors, depending on the command-line switches used.

The following built-in functions are always available. They all generate the machine instruction that is part of the name.

```
long __builtin_alpha_implver (void);
long __builtin_alpha_rpcc (void);
long __builtin_alpha_amask (long);
long __builtin_alpha_cmpbge (long, long);
long __builtin_alpha_extbl (long, long);
long __builtin_alpha_extwl (long, long);
long __builtin_alpha_extll (long, long);
long __builtin_alpha_extql (long, long);
long __builtin_alpha_extwh (long, long);
long __builtin_alpha_extlh (long, long);
long __builtin_alpha_extqh (long, long);
long __builtin_alpha_insbl (long, long);
long __builtin_alpha_inswl (long, long);
long __builtin_alpha_insll (long, long);
long __builtin_alpha_insql (long, long);
long __builtin_alpha_inswh (long, long);
long __builtin_alpha_inslh (long, long);
long __builtin_alpha_insqh (long, long);
long __builtin_alpha_mskbl (long, long);
long __builtin_alpha_mskwl (long, long);
long __builtin_alpha_mskll (long, long);
long __builtin_alpha_mskql (long, long);
long __builtin_alpha_mskwh (long, long);
long __builtin_alpha_msklh (long, long);
long __builtin_alpha_mskqh (long, long);
long __builtin_alpha_umulh (long, long);
long __builtin_alpha_zap (long, long);
long __builtin_alpha_zapnot (long, long);
```

The following built-in functions are always with `-mmax` or `-mcpu=`*cpu* where *cpu* is `pca56` or later. They all generate the machine instruction that is part of the name.

```
long __builtin_alpha_pklb (long);
long __builtin_alpha_pkwb (long);
long __builtin_alpha_unpkbl (long);
long __builtin_alpha_unpkbw (long);
long __builtin_alpha_minub8 (long, long);
long __builtin_alpha_minsb8 (long, long);
long __builtin_alpha_minuw4 (long, long);
long __builtin_alpha_minsw4 (long, long);
long __builtin_alpha_maxub8 (long, long);
long __builtin_alpha_maxsb8 (long, long);
long __builtin_alpha_maxuw4 (long, long);
long __builtin_alpha_maxsw4 (long, long);
long __builtin_alpha_perr (long, long);
```

The following built-in functions are always with `-mcix` or `-mcpu=`*cpu* where *cpu* is `ev67` or later. They all generate the machine instruction that is part of the name.

```
long __builtin_alpha_cttz (long);
long __builtin_alpha_ctlz (long);
long __builtin_alpha_ctpop (long);
```

The following built-in functions are available on systems that use the OSF/1 PALcode. Normally they invoke the `rduniq` and `wruniq` PAL calls, but when invoked with `-mtls-kernel`, they invoke `rdval` and `wrval`.

```
void *__builtin_thread_pointer (void);
void __builtin_set_thread_pointer (void *);
```

6.60.3 Altera Nios II Built-in Functions

These built-in functions are available for the Altera Nios II family of processors.

The following built-in functions are always available. They all generate the machine instruction that is part of the name.

```
int __builtin_ldbio (volatile const void *);
int __builtin_ldbuio (volatile const void *);
int __builtin_ldhio (volatile const void *);
int __builtin_ldhuio (volatile const void *);
int __builtin_ldwio (volatile const void *);
void __builtin_stbio (volatile void *, int);
void __builtin_sthio (volatile void *, int);
void __builtin_stwio (volatile void *, int);
void __builtin_sync (void);
int __builtin_rdctl (int);
int __builtin_rdprs (int, int);
void __builtin_wrctl (int, int);
void __builtin_flushd (volatile void *);
void __builtin_flushda (volatile void *);
int __builtin_wrpie (int);
void __builtin_eni (int);
int __builtin_ldex (volatile const void *);
int __builtin_stex (volatile void *, int);
int __builtin_ldsex (volatile const void *);
int __builtin_stsex (volatile void *, int);
```

The following built-in functions are always available. They all generate a Nios II Custom Instruction. The name of the function represents the types that the function takes and returns. The letter before the `n` is the return type or void if absent. The `n` represents the first parameter to all the custom instructions, the custom instruction number. The two letters after the `n` represent the up to two parameters to the function.

The letters represent the following data types:

`<no letter>`
: `void` for return type and no parameter for parameter types.

`i`
: `int` for return type and parameter type

`f`
: `float` for return type and parameter type

`p`
: `void *` for return type and parameter type

And the function names are:

```
void __builtin_custom_n (void);
void __builtin_custom_ni (int);
```

```
void __builtin_custom_nf (float);
void __builtin_custom_np (void *);
void __builtin_custom_nii (int, int);
void __builtin_custom_nif (int, float);
void __builtin_custom_nip (int, void *);
void __builtin_custom_nfi (float, int);
void __builtin_custom_nff (float, float);
void __builtin_custom_nfp (float, void *);
void __builtin_custom_npi (void *, int);
void __builtin_custom_npf (void *, float);
void __builtin_custom_npp (void *, void *);
int __builtin_custom_in (void);
int __builtin_custom_ini (int);
int __builtin_custom_inf (float);
int __builtin_custom_inp (void *);
int __builtin_custom_inii (int, int);
int __builtin_custom_inif (int, float);
int __builtin_custom_inip (int, void *);
int __builtin_custom_infi (float, int);
int __builtin_custom_inff (float, float);
int __builtin_custom_infp (float, void *);
int __builtin_custom_inpi (void *, int);
int __builtin_custom_inpf (void *, float);
int __builtin_custom_inpp (void *, void *);
float __builtin_custom_fn (void);
float __builtin_custom_fni (int);
float __builtin_custom_fnf (float);
float __builtin_custom_fnp (void *);
float __builtin_custom_fnii (int, int);
float __builtin_custom_fnif (int, float);
float __builtin_custom_fnip (int, void *);
float __builtin_custom_fnfi (float, int);
float __builtin_custom_fnff (float, float);
float __builtin_custom_fnfp (float, void *);
float __builtin_custom_fnpi (void *, int);
float __builtin_custom_fnpf (void *, float);
float __builtin_custom_fnpp (void *, void *);
void * __builtin_custom_pn (void);
void * __builtin_custom_pni (int);
void * __builtin_custom_pnf (float);
void * __builtin_custom_pnp (void *);
void * __builtin_custom_pnii (int, int);
void * __builtin_custom_pnif (int, float);
void * __builtin_custom_pnip (int, void *);
void * __builtin_custom_pnfi (float, int);
void * __builtin_custom_pnff (float, float);
void * __builtin_custom_pnfp (float, void *);
```

```
void * __builtin_custom_pnpi (void *, int);
void * __builtin_custom_pnpf (void *, float);
void * __builtin_custom_pnpp (void *, void *);
```

6.60.4 ARC Built-in Functions

The following built-in functions are provided for ARC targets. The built-ins generate the corresponding assembly instructions. In the examples given below, the generated code often requires an operand or result to be in a register. Where necessary further code will be generated to ensure this is true, but for brevity this is not described in each case.

Note: Using a built-in to generate an instruction not supported by a target may cause problems. At present the compiler is not guaranteed to detect such misuse, and as a result an internal compiler error may be generated.

`int __builtin_arc_aligned (void *val, int alignval)` [Built-in Function]
Return 1 if *val* is known to have the byte alignment given by *alignval*, otherwise return 0. Note that this is different from

```
__alignof__(*(char *)val) >= alignval
```

because __alignof__ sees only the type of the dereference, whereas __builtin_arc_align uses alignment information from the pointer as well as from the pointed-to type. The information available will depend on optimization level.

`void __builtin_arc_brk (void)` [Built-in Function]
Generates

```
brk
```

`unsigned int __builtin_arc_core_read (unsigned int regno)` [Built-in Function]
The operand is the number of a register to be read. Generates:

```
mov  dest, rregno
```

where the value in *dest* will be the result returned from the built-in.

`void __builtin_arc_core_write (unsigned int regno, unsigned int val)` [Built-in Function]
The first operand is the number of a register to be written, the second operand is a compile time constant to write into that register. Generates:

```
mov  rregno, val
```

`int __builtin_arc_divaw (int a, int b)` [Built-in Function]
Only available if either `-mcpu=ARC700` or `-meA` is set. Generates:

```
divaw  dest, a, b
```

where the value in *dest* will be the result returned from the built-in.

`void __builtin_arc_flag (unsigned int a)` [Built-in Function]
Generates

```
flag  a
```

`unsigned int __builtin_arc_lr (unsigned int auxr)` [Built-in Function]
The operand, *auxv*, is the address of an auxiliary register and must be a compile time constant. Generates:

```
lr  dest, [auxr]
```

Where the value in *dest* will be the result returned from the built-in.

`void __builtin_arc_mul64 (int a, int b)` [Built-in Function]
Only available with `-mmul64`. Generates:

```
mul64  a, b
```

`void __builtin_arc_mulu64 (unsigned int a, unsigned int b)` [Built-in Function]
Only available with `-mmul64`. Generates:

```
mulu64  a, b
```

`void __builtin_arc_nop (void)` [Built-in Function]
Generates:

```
nop
```

`int __builtin_arc_norm (int src)` [Built-in Function]
Only valid if the 'norm' instruction is available through the `-mnorm` option or by default with `-mcpu=ARC700`. Generates:

```
norm  dest, src
```

Where the value in *dest* will be the result returned from the built-in.

`short int __builtin_arc_normw (short int src)` [Built-in Function]
Only valid if the 'normw' instruction is available through the `-mnorm` option or by default with `-mcpu=ARC700`. Generates:

```
normw  dest, src
```

Where the value in *dest* will be the result returned from the built-in.

`void __builtin_arc_rtie (void)` [Built-in Function]
Generates:

```
rtie
```

`void __builtin_arc_sleep (int a` [Built-in Function]
Generates:

```
sleep  a
```

`void __builtin_arc_sr (unsigned int val, unsigned int auxr)` [Built-in Function]
The first argument, *val*, is a compile time constant to be written to the register, the second argument, *auxr*, is the address of an auxiliary register. Generates:

```
sr  val, [auxr]
```

`int __builtin_arc_swap (int src)` [Built-in Function]

Only valid with `-mswap`. Generates:

```
swap  dest, src
```

Where the value in *dest* will be the result returned from the built-in.

`void __builtin_arc_swi (void)` [Built-in Function]

Generates:

```
swi
```

`void __builtin_arc_sync (void)` [Built-in Function]

Only available with `-mcpu=ARC700`. Generates:

```
sync
```

`void __builtin_arc_trap_s (unsigned int c)` [Built-in Function]

Only available with `-mcpu=ARC700`. Generates:

```
trap_s  c
```

`void __builtin_arc_unimp_s (void)` [Built-in Function]

Only available with `-mcpu=ARC700`. Generates:

```
unimp_s
```

The instructions generated by the following builtins are not considered as candidates for scheduling. They are not moved around by the compiler during scheduling, and thus can be expected to appear where they are put in the C code:

```
__builtin_arc_brk()
__builtin_arc_core_read()
__builtin_arc_core_write()
__builtin_arc_flag()
__builtin_arc_lr()
__builtin_arc_sleep()
__builtin_arc_sr()
__builtin_arc_swi()
```

6.60.5 ARC SIMD Built-in Functions

SIMD builtins provided by the compiler can be used to generate the vector instructions. This section describes the available builtins and their usage in programs. With the `-msimd` option, the compiler provides 128-bit vector types, which can be specified using the `vector_size` attribute. The header file `arc-simd.h` can be included to use the following predefined types:

```
typedef int __v4si   __attribute__((vector_size(16)));
typedef short __v8hi __attribute__((vector_size(16)));
```

These types can be used to define 128-bit variables. The built-in functions listed in the following section can be used on these variables to generate the vector operations.

For all builtins, `__builtin_arc_someinsn`, the header file `arc-simd.h` also provides equivalent macros called `_someinsn` that can be used for programming ease and improved readability. The following macros for DMA control are also provided:

```
#define _setup_dma_in_channel_reg _vdiwr
```

```
#define _setup_dma_out_channel_reg _vdowr
```

The following is a complete list of all the SIMD built-ins provided for ARC, grouped by calling signature.

The following take two `__v8hi` arguments and return a `__v8hi` result:

```
__v8hi __builtin_arc_vaddaw (__v8hi, __v8hi);
__v8hi __builtin_arc_vaddw (__v8hi, __v8hi);
__v8hi __builtin_arc_vand (__v8hi, __v8hi);
__v8hi __builtin_arc_vandaw (__v8hi, __v8hi);
__v8hi __builtin_arc_vavb (__v8hi, __v8hi);
__v8hi __builtin_arc_vavrb (__v8hi, __v8hi);
__v8hi __builtin_arc_vbic (__v8hi, __v8hi);
__v8hi __builtin_arc_vbicaw (__v8hi, __v8hi);
__v8hi __builtin_arc_vdifaw (__v8hi, __v8hi);
__v8hi __builtin_arc_vdifw (__v8hi, __v8hi);
__v8hi __builtin_arc_veqw (__v8hi, __v8hi);
__v8hi __builtin_arc_vh264f (__v8hi, __v8hi);
__v8hi __builtin_arc_vh264ft (__v8hi, __v8hi);
__v8hi __builtin_arc_vh264fw (__v8hi, __v8hi);
__v8hi __builtin_arc_vlew (__v8hi, __v8hi);
__v8hi __builtin_arc_vltw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmaxaw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmaxw (__v8hi, __v8hi);
__v8hi __builtin_arc_vminaw (__v8hi, __v8hi);
__v8hi __builtin_arc_vminw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr1aw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr1w (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr2aw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr2w (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr3aw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr3w (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr4aw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr4w (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr5aw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr5w (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr6aw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr6w (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr7aw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmr7w (__v8hi, __v8hi);
__v8hi __builtin_arc_vmrb (__v8hi, __v8hi);
__v8hi __builtin_arc_vmulaw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmulfaw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmulfw (__v8hi, __v8hi);
__v8hi __builtin_arc_vmulw (__v8hi, __v8hi);
__v8hi __builtin_arc_vnew (__v8hi, __v8hi);
__v8hi __builtin_arc_vor (__v8hi, __v8hi);
__v8hi __builtin_arc_vsubaw (__v8hi, __v8hi);
```

```
__v8hi __builtin_arc_vsubw (__v8hi, __v8hi);
__v8hi __builtin_arc_vsummw (__v8hi, __v8hi);
__v8hi __builtin_arc_vvc1f (__v8hi, __v8hi);
__v8hi __builtin_arc_vvc1ft (__v8hi, __v8hi);
__v8hi __builtin_arc_vxor (__v8hi, __v8hi);
__v8hi __builtin_arc_vxoraw (__v8hi, __v8hi);
```

The following take one `__v8hi` and one `int` argument and return a `__v8hi` result:

```
__v8hi __builtin_arc_vbaddw (__v8hi, int);
__v8hi __builtin_arc_vbmaxw (__v8hi, int);
__v8hi __builtin_arc_vbminw (__v8hi, int);
__v8hi __builtin_arc_vbmulaw (__v8hi, int);
__v8hi __builtin_arc_vbmulfw (__v8hi, int);
__v8hi __builtin_arc_vbmulw (__v8hi, int);
__v8hi __builtin_arc_vbrsubw (__v8hi, int);
__v8hi __builtin_arc_vbsubw (__v8hi, int);
```

The following take one `__v8hi` argument and one `int` argument which must be a 3-bit compile time constant indicating a register number I0-I7. They return a `__v8hi` result.

```
__v8hi __builtin_arc_vasrw (__v8hi, const int);
__v8hi __builtin_arc_vsr8 (__v8hi, const int);
__v8hi __builtin_arc_vsr8aw (__v8hi, const int);
```

The following take one `__v8hi` argument and one `int` argument which must be a 6-bit compile time constant. They return a `__v8hi` result.

```
__v8hi __builtin_arc_vasrpwbi (__v8hi, const int);
__v8hi __builtin_arc_vasrrpwbi (__v8hi, const int);
__v8hi __builtin_arc_vasrrwi (__v8hi, const int);
__v8hi __builtin_arc_vasrsrwi (__v8hi, const int);
__v8hi __builtin_arc_vasrwi (__v8hi, const int);
__v8hi __builtin_arc_vsr8awi (__v8hi, const int);
__v8hi __builtin_arc_vsr8i (__v8hi, const int);
```

The following take one `__v8hi` argument and one `int` argument which must be a 8-bit compile time constant. They return a `__v8hi` result.

```
__v8hi __builtin_arc_vd6tapf (__v8hi, const int);
__v8hi __builtin_arc_vmvaw (__v8hi, const int);
__v8hi __builtin_arc_vmvw (__v8hi, const int);
__v8hi __builtin_arc_vmvzw (__v8hi, const int);
```

The following take two `int` arguments, the second of which which must be a 8-bit compile time constant. They return a `__v8hi` result:

```
__v8hi __builtin_arc_vmovaw (int, const int);
__v8hi __builtin_arc_vmovw (int, const int);
__v8hi __builtin_arc_vmovzw (int, const int);
```

The following take a single `__v8hi` argument and return a `__v8hi` result:

```
__v8hi __builtin_arc_vabsaw (__v8hi);
__v8hi __builtin_arc_vabsw (__v8hi);
__v8hi __builtin_arc_vaddsuw (__v8hi);
```

```
__v8hi __builtin_arc_vexch1 (__v8hi);
__v8hi __builtin_arc_vexch2 (__v8hi);
__v8hi __builtin_arc_vexch4 (__v8hi);
__v8hi __builtin_arc_vsignw (__v8hi);
__v8hi __builtin_arc_vupbaw (__v8hi);
__v8hi __builtin_arc_vupbw (__v8hi);
__v8hi __builtin_arc_vupsbaw (__v8hi);
__v8hi __builtin_arc_vupsbw (__v8hi);
```

The following take two `int` arguments and return no result:

```
void __builtin_arc_vdirun (int, int);
void __builtin_arc_vdorun (int, int);
```

The following take two `int` arguments and return no result. The first argument must a 3-bit compile time constant indicating one of the DR0-DR7 DMA setup channels:

```
void __builtin_arc_vdiwr (const int, int);
void __builtin_arc_vdowr (const int, int);
```

The following take an `int` argument and return no result:

```
void __builtin_arc_vendrec (int);
void __builtin_arc_vrec (int);
void __builtin_arc_vrecrun (int);
void __builtin_arc_vrun (int);
```

The following take a `__v8hi` argument and two `int` arguments and return a `__v8hi` result. The second argument must be a 3-bit compile time constants, indicating one the registers I0-I7, and the third argument must be an 8-bit compile time constant.

Note: Although the equivalent hardware instructions do not take an SIMD register as an operand, these builtins overwrite the relevant bits of the `__v8hi` register provided as the first argument with the value loaded from the `[Ib, u8]` location in the SDM.

```
__v8hi __builtin_arc_vld32 (__v8hi, const int, const int);
__v8hi __builtin_arc_vld32wh (__v8hi, const int, const int);
__v8hi __builtin_arc_vld32wl (__v8hi, const int, const int);
__v8hi __builtin_arc_vld64 (__v8hi, const int, const int);
```

The following take two `int` arguments and return a `__v8hi` result. The first argument must be a 3-bit compile time constants, indicating one the registers I0-I7, and the second argument must be an 8-bit compile time constant.

```
__v8hi __builtin_arc_vld128 (const int, const int);
__v8hi __builtin_arc_vld64w (const int, const int);
```

The following take a `__v8hi` argument and two `int` arguments and return no result. The second argument must be a 3-bit compile time constants, indicating one the registers I0-I7, and the third argument must be an 8-bit compile time constant.

```
void __builtin_arc_vst128 (__v8hi, const int, const int);
void __builtin_arc_vst64 (__v8hi, const int, const int);
```

The following take a `__v8hi` argument and three `int` arguments and return no result. The second argument must be a 3-bit compile-time constant, identifying the 16-bit sub-register to be stored, the third argument must be a 3-bit compile time constants, indicating one the registers I0-I7, and the fourth argument must be an 8-bit compile time constant.

```
void __builtin_arc_vst16_n (__v8hi, const int, const int, const int);
void __builtin_arc_vst32_n (__v8hi, const int, const int, const int);
```

6.60.6 ARM iWMMXt Built-in Functions

These built-in functions are available for the ARM family of processors when the `-mcpu=iwmmxt` switch is used:

```
typedef int v2si __attribute__ ((vector_size (8)));
typedef short v4hi __attribute__ ((vector_size (8)));
typedef char v8qi __attribute__ ((vector_size (8)));

int __builtin_arm_getwcgr0 (void);
void __builtin_arm_setwcgr0 (int);
int __builtin_arm_getwcgr1 (void);
void __builtin_arm_setwcgr1 (int);
int __builtin_arm_getwcgr2 (void);
void __builtin_arm_setwcgr2 (int);
int __builtin_arm_getwcgr3 (void);
void __builtin_arm_setwcgr3 (int);
int __builtin_arm_textrmsb (v8qi, int);
int __builtin_arm_textrmsh (v4hi, int);
int __builtin_arm_textrmsw (v2si, int);
int __builtin_arm_textrmub (v8qi, int);
int __builtin_arm_textrmuh (v4hi, int);
int __builtin_arm_textrmuw (v2si, int);
v8qi __builtin_arm_tinsrb (v8qi, int, int);
v4hi __builtin_arm_tinsrh (v4hi, int, int);
v2si __builtin_arm_tinsrw (v2si, int, int);
long long __builtin_arm_tmia (long long, int, int);
long long __builtin_arm_tmiabb (long long, int, int);
long long __builtin_arm_tmiabt (long long, int, int);
long long __builtin_arm_tmiaph (long long, int, int);
long long __builtin_arm_tmiatb (long long, int, int);
long long __builtin_arm_tmiatt (long long, int, int);
int __builtin_arm_tmovmskb (v8qi);
int __builtin_arm_tmovmskh (v4hi);
int __builtin_arm_tmovmskw (v2si);
long long __builtin_arm_waccb (v8qi);
long long __builtin_arm_wacch (v4hi);
long long __builtin_arm_waccw (v2si);
v8qi __builtin_arm_waddb (v8qi, v8qi);
v8qi __builtin_arm_waddbss (v8qi, v8qi);
v8qi __builtin_arm_waddbus (v8qi, v8qi);
v4hi __builtin_arm_waddh (v4hi, v4hi);
v4hi __builtin_arm_waddhss (v4hi, v4hi);
v4hi __builtin_arm_waddhus (v4hi, v4hi);
v2si __builtin_arm_waddw (v2si, v2si);
v2si __builtin_arm_waddwss (v2si, v2si);
v2si __builtin_arm_waddwus (v2si, v2si);
v8qi __builtin_arm_walign (v8qi, v8qi, int);
long long __builtin_arm_wand(long long, long long);
long long __builtin_arm_wandn (long long, long long);
v8qi __builtin_arm_wavg2b (v8qi, v8qi);
v8qi __builtin_arm_wavg2br (v8qi, v8qi);
v4hi __builtin_arm_wavg2h (v4hi, v4hi);
v4hi __builtin_arm_wavg2hr (v4hi, v4hi);
v8qi __builtin_arm_wcmpeqb (v8qi, v8qi);
v4hi __builtin_arm_wcmpeqh (v4hi, v4hi);
```

```
v2si __builtin_arm_wcmpeqw (v2si, v2si);
v8qi __builtin_arm_wcmpgtsb (v8qi, v8qi);
v4hi __builtin_arm_wcmpgtsh (v4hi, v4hi);
v2si __builtin_arm_wcmpgtsw (v2si, v2si);
v8qi __builtin_arm_wcmpgtub (v8qi, v8qi);
v4hi __builtin_arm_wcmpgtuh (v4hi, v4hi);
v2si __builtin_arm_wcmpgtuw (v2si, v2si);
long long __builtin_arm_wmacs (long long, v4hi, v4hi);
long long __builtin_arm_wmacsz (v4hi, v4hi);
long long __builtin_arm_wmacu (long long, v4hi, v4hi);
long long __builtin_arm_wmacuz (v4hi, v4hi);
v4hi __builtin_arm_wmadds (v4hi, v4hi);
v4hi __builtin_arm_wmaddu (v4hi, v4hi);
v8qi __builtin_arm_wmaxsb (v8qi, v8qi);
v4hi __builtin_arm_wmaxsh (v4hi, v4hi);
v2si __builtin_arm_wmaxsw (v2si, v2si);
v8qi __builtin_arm_wmaxub (v8qi, v8qi);
v4hi __builtin_arm_wmaxuh (v4hi, v4hi);
v2si __builtin_arm_wmaxuw (v2si, v2si);
v8qi __builtin_arm_wminsb (v8qi, v8qi);
v4hi __builtin_arm_wminsh (v4hi, v4hi);
v2si __builtin_arm_wminsw (v2si, v2si);
v8qi __builtin_arm_wminub (v8qi, v8qi);
v4hi __builtin_arm_wminuh (v4hi, v4hi);
v2si __builtin_arm_wminuw (v2si, v2si);
v4hi __builtin_arm_wmulsm (v4hi, v4hi);
v4hi __builtin_arm_wmulul (v4hi, v4hi);
v4hi __builtin_arm_wmulum (v4hi, v4hi);
long long __builtin_arm_wor (long long, long long);
v2si __builtin_arm_wpackdss (long long, long long);
v2si __builtin_arm_wpackdus (long long, long long);
v8qi __builtin_arm_wpackhss (v4hi, v4hi);
v8qi __builtin_arm_wpackhus (v4hi, v4hi);
v4hi __builtin_arm_wpackwss (v2si, v2si);
v4hi __builtin_arm_wpackwus (v2si, v2si);
long long __builtin_arm_wrord (long long, long long);
long long __builtin_arm_wrordi (long long, int);
v4hi __builtin_arm_wrorh (v4hi, long long);
v4hi __builtin_arm_wrorhi (v4hi, int);
v2si __builtin_arm_wrorw (v2si, long long);
v2si __builtin_arm_wrorwi (v2si, int);
v2si __builtin_arm_wsadb (v2si, v8qi, v8qi);
v2si __builtin_arm_wsadbz (v8qi, v8qi);
v2si __builtin_arm_wsadh (v2si, v4hi, v4hi);
v2si __builtin_arm_wsadhz (v4hi, v4hi);
v4hi __builtin_arm_wshufh (v4hi, int);
long long __builtin_arm_wslld (long long, long long);
long long __builtin_arm_wslldi (long long, int);
v4hi __builtin_arm_wsllh (v4hi, long long);
v4hi __builtin_arm_wsllhi (v4hi, int);
v2si __builtin_arm_wsllw (v2si, long long);
v2si __builtin_arm_wsllwi (v2si, int);
long long __builtin_arm_wsrad (long long, long long);
long long __builtin_arm_wsradi (long long, int);
v4hi __builtin_arm_wsrah (v4hi, long long);
v4hi __builtin_arm_wsrahi (v4hi, int);
v2si __builtin_arm_wsraw (v2si, long long);
v2si __builtin_arm_wsrawi (v2si, int);
```

```
long long __builtin_arm_wsrld (long long, long long);
long long __builtin_arm_wsrldi (long long, int);
v4hi __builtin_arm_wsrlh (v4hi, long long);
v4hi __builtin_arm_wsrlhi (v4hi, int);
v2si __builtin_arm_wsrlw (v2si, long long);
v2si __builtin_arm_wsrlwi (v2si, int);
v8qi __builtin_arm_wsubb (v8qi, v8qi);
v8qi __builtin_arm_wsubbss (v8qi, v8qi);
v8qi __builtin_arm_wsubbus (v8qi, v8qi);
v4hi __builtin_arm_wsubh (v4hi, v4hi);
v4hi __builtin_arm_wsubhss (v4hi, v4hi);
v4hi __builtin_arm_wsubhus (v4hi, v4hi);
v2si __builtin_arm_wsubw (v2si, v2si);
v2si __builtin_arm_wsubwss (v2si, v2si);
v2si __builtin_arm_wsubwus (v2si, v2si);
v4hi __builtin_arm_wunpckehsb (v8qi);
v2si __builtin_arm_wunpckehsh (v4hi);
long long __builtin_arm_wunpckehsw (v2si);
v4hi __builtin_arm_wunpckehub (v8qi);
v2si __builtin_arm_wunpckehuh (v4hi);
long long __builtin_arm_wunpckehuw (v2si);
v4hi __builtin_arm_wunpckelsb (v8qi);
v2si __builtin_arm_wunpckelsh (v4hi);
long long __builtin_arm_wunpckelsw (v2si);
v4hi __builtin_arm_wunpckelub (v8qi);
v2si __builtin_arm_wunpckeluh (v4hi);
long long __builtin_arm_wunpckeluw (v2si);
v8qi __builtin_arm_wunpckihb (v8qi, v8qi);
v4hi __builtin_arm_wunpckihh (v4hi, v4hi);
v2si __builtin_arm_wunpckihw (v2si, v2si);
v8qi __builtin_arm_wunpckilb (v8qi, v8qi);
v4hi __builtin_arm_wunpckilh (v4hi, v4hi);
v2si __builtin_arm_wunpckilw (v2si, v2si);
long long __builtin_arm_wxor (long long, long long);
long long __builtin_arm_wzero ();
```

6.60.7 ARM C Language Extensions (ACLE)

GCC implements extensions for C as described in the ARM C Language Extensions (ACLE) specification, which can be found at `https://developer.arm.com/documentation/ihi0053/latest/`.

As a part of ACLE, GCC implements extensions for Advanced SIMD as described in the ARM C Language Extensions Specification. The complete list of Advanced SIMD intrinsics can be found at `https://developer.arm.com/documentation/ihi0073/latest/`. The built-in intrinsics for the Advanced SIMD extension are available when NEON is enabled.

Currently, ARM and AArch64 back ends do not support ACLE 2.0 fully. Both back ends support CRC32 intrinsics and the ARM back end supports the Coprocessor intrinsics, all from `arm_acle.h`. The ARM back end's 16-bit floating-point Advanced SIMD intrinsics currently comply to ACLE v1.1. AArch64's back end does not have support for 16-bit floating point Advanced SIMD intrinsics yet.

See Section 3.19.5 [ARM Options], page 329, and Section 3.19.1 [AArch64 Options], page 308, for more information on the availability of extensions.

6.60.8 ARM Floating Point Status and Control Intrinsics

These built-in functions are available for the ARM family of processors with floating-point unit.

```
unsigned int __builtin_arm_get_fpscr ();
void __builtin_arm_set_fpscr (unsigned int);
```

6.60.9 ARM ARMv8-M Security Extensions

GCC implements the ARMv8-M Security Extensions as described in the ARMv8-M Security Extensions: Requirements on Development Tools Engineering Specification, which can be found at `https://developer.arm.com/documentation/ecm0359818/latest/`.

As part of the Security Extensions GCC implements two new function attributes: `cmse_nonsecure_entry` and `cmse_nonsecure_call`.

As part of the Security Extensions GCC implements the intrinsics below. FPTR is used here to mean any function pointer type.

```
cmse_address_info_t cmse_TT (void *);
cmse_address_info_t cmse_TT_fptr (FPTR);
cmse_address_info_t cmse_TTT (void *);
cmse_address_info_t cmse_TTT_fptr (FPTR);
cmse_address_info_t cmse_TTA (void *);
cmse_address_info_t cmse_TTA_fptr (FPTR);
cmse_address_info_t cmse_TTAT (void *);
cmse_address_info_t cmse_TTAT_fptr (FPTR);
void * cmse_check_address_range (void *, size_t, int);
typeof(p) cmse_nsfptr_create (FPTR p);
intptr_t cmse_is_nsfptr (FPTR);
int cmse_nonsecure_caller (void);
```

6.60.10 AVR Built-in Functions

For each built-in function for AVR, there is an equally named, uppercase built-in macro defined. That way users can easily query if or if not a specific built-in is implemented or not. For example, if `__builtin_avr_nop` is available the macro `__BUILTIN_AVR_NOP` is defined to `1` and undefined otherwise.

`void __builtin_avr_nop (void)`
`void __builtin_avr_sei (void)`
`void __builtin_avr_cli (void)`
`void __builtin_avr_sleep (void)`
`void __builtin_avr_wdr (void)`
`unsigned char __builtin_avr_swap (unsigned char)`
`unsigned int __builtin_avr_fmul (unsigned char, unsigned char)`
`int __builtin_avr_fmuls (char, char)`
`int __builtin_avr_fmulsu (char, unsigned char)`

These built-in functions map to the respective machine instruction, i.e. `nop`, `sei`, `cli`, `sleep`, `wdr`, `swap`, `fmul`, `fmuls` resp. `fmulsu`. The three `fmul*` built-ins are implemented as library call if no hardware multiplier is available.

`void __builtin_avr_delay_cycles (unsigned long ticks)`

Delay execution for *ticks* cycles. Note that this built-in does not take into account the effect of interrupts that might increase delay time. *ticks* must be a

compile-time integer constant; delays with a variable number of cycles are not supported.

`char __builtin_avr_flash_segment (const __memx void*)`
: This built-in takes a byte address to the 24-bit [AVR Named Address Spaces], page 556, `__memx` and returns the number of the flash segment (the 64 KiB chunk) where the address points to. Counting starts at `0`. If the address does not point to flash memory, return `-1`.

`uint8_t __builtin_avr_insert_bits (uint32_t map, uint8_t bits, uint8_t val)`
: Insert bits from *bits* into *val* and return the resulting value. The nibbles of *map* determine how the insertion is performed: Let X be the n-th nibble of *map*

  1. If X is `0xf`, then the n-th bit of *val* is returned unaltered.
  2. If X is in the range $0 \ldots 7$, then the n-th result bit is set to the X-th bit of *bits*
  3. If X is in the range $8 \ldots$ `0xe`, then the n-th result bit is undefined.

  One typical use case for this built-in is adjusting input and output values to non-contiguous port layouts. Some examples:

```
// same as val, bits is unused
__builtin_avr_insert_bits (0xffffffff, bits, val);

// same as bits, val is unused
__builtin_avr_insert_bits (0x76543210, bits, val);

// same as rotating bits by 4
__builtin_avr_insert_bits (0x32107654, bits, 0);

// high nibble of result is the high nibble of val
// low nibble of result is the low nibble of bits
__builtin_avr_insert_bits (0xffff3210, bits, val);

// reverse the bit order of bits
__builtin_avr_insert_bits (0x01234567, bits, 0);
```

`void __builtin_avr_nops (unsigned count)`
: Insert *count* `NOP` instructions. The number of instructions must be a compile-time integer constant.

There are many more AVR-specific built-in functions that are used to implement the ISO/IEC TR 18037 "Embedded C" fixed-point functions of section 7.18a.6. You don't need to use these built-ins directly. Instead, use the declarations as supplied by the `stdfix.h` header with GNU-C99:

```
#include <stdfix.h>

// Re-interpret the bit representation of unsigned 16-bit
// integer uval as Q-format 0.16 value.
unsigned fract get_bits (uint_ur_t uval)
{
    return urbits (uval);
}
```

6.60.11 Blackfin Built-in Functions

Currently, there are two Blackfin-specific built-in functions. These are used for generating `CSYNC` and `SSYNC` machine insns without using inline assembly; by using these built-in

functions the compiler can automatically add workarounds for hardware errata involving these instructions. These functions are named as follows:

```
void __builtin_bfin_csync (void);
void __builtin_bfin_ssync (void);
```

6.60.12 BPF Built-in Functions

The following built-in functions are available for eBPF targets.

`unsigned long long __builtin_bpf_load_byte (unsigned long long offset)` [Built-in Function]

Load a byte from the `struct sk_buff` packet data pointed by the register `%r6` and return it.

`unsigned long long __builtin_bpf_load_half (unsigned long long offset)` [Built-in Function]

Load 16 bits from the `struct sk_buff` packet data pointed by the register `%r6` and return it.

`unsigned long long __builtin_bpf_load_word (unsigned long long offset)` [Built-in Function]

Load 32 bits from the `struct sk_buff` packet data pointed by the register `%r6` and return it.

`void * __builtin_preserve_access_index (expr)` [Built-in Function]

BPF Compile Once-Run Everywhere (CO-RE) support. Instruct GCC to generate CO-RE relocation records for any accesses to aggregate data structures (struct, union, array types) in *expr*. This builtin is otherwise transparent, the return value is whatever *expr* evaluates to. It is also overloaded: *expr* may be of any type (not necessarily a pointer), the return type is the same. Has no effect if `-mco-re` is not in effect (either specified or implied).

`unsigned int __builtin_preserve_field_info (expr, unsigned int kind)` [Built-in Function]

BPF Compile Once-Run Everywhere (CO-RE) support. This builtin is used to extract information to aid in struct/union relocations. *expr* is an access to a field of a struct or union. Depending on *kind*, different information is returned to the program. A CO-RE relocation for the access in *expr* with kind *kind* is recorded if `-mco-re` is in effect.

The following values are supported for *kind*:

`FIELD_BYTE_OFFSET = 0`

: The returned value is the offset, in bytes, of the field from the beginning of the containing structure. For bit-fields, this is the byte offset of the containing word.

`FIELD_BYTE_SIZE = 1`

: The returned value is the size, in bytes, of the field. For bit-fields, this is the size in bytes of the containing word.

`FIELD_EXISTENCE = 2`
: The returned value is 1 if the field exists, 0 otherwise. Always 1 at compile time.

`FIELD_SIGNEDNESS = 3`
: The returned value is 1 if the field is signed, 0 otherwise.

`FIELD_LSHIFT_U64 = 4`
`FIELD_RSHIFT_U64 = 5`
: The returned value is the number of bits of left- or right-shifting (respectively) needed in order to recover the original value of the field, after it has been loaded by a read of `FIELD_BYTE_SIZE` bytes into an unsigned 64-bit value. Primarily useful for reading bit-field values from structures that may change between kernel versions.

Note that the return value is a constant which is known at compile time. If the field has a variable offset then `FIELD_BYTE_OFFSET`, `FIELD_LSHIFT_U64`, and `FIELD_RSHIFT_U64` are not supported. Similarly, if the field has a variable size then `FIELD_BYTE_SIZE`, `FIELD_LSHIFT_U64`, and `FIELD_RSHIFT_U64` are not supported.

For example, `__builtin_preserve_field_info` can be used to reliably extract bit-field values from a structure that may change between kernel versions:

```
struct S
{
  short a;
  int x:7;
  int y:5;
};

int
read_y (struct S *arg)
{
  unsigned long long val;
  unsigned int offset
    = __builtin_preserve_field_info (arg->y, FIELD_BYTE_OFFSET);
  unsigned int size
    = __builtin_preserve_field_info (arg->y, FIELD_BYTE_SIZE);

  /* Read size bytes from arg + offset into val.  */
  bpf_probe_read (&val, size, arg + offset);

  val <<= __builtin_preserve_field_info (arg->y, FIELD_LSHIFT_U64);

  if (__builtin_preserve_field_info (arg->y, FIELD_SIGNEDNESS))
    val = ((long long) val
           >> __builtin_preserve_field_info (arg->y, FIELD_RSHIFT_U64));
  else
    val >>= __builtin_preserve_field_info (arg->y, FIELD_RSHIFT_U64);

  return val;
}
```

6.60.13 FR-V Built-in Functions

GCC provides many FR-V-specific built-in functions. In general, these functions are intended to be compatible with those described by *FR-V Family, Softune C/C++ Compiler Manual (V6), Fujitsu Semiconductor*. The two exceptions are `__MDUNPACKH` and `__MBTOHE`, the GCC forms of which pass 128-bit values by pointer rather than by value.

Most of the functions are named after specific FR-V instructions. Such functions are said to be "directly mapped" and are summarized here in tabular form.

6.60.13.1 Argument Types

The arguments to the built-in functions can be divided into three groups: register numbers, compile-time constants and run-time values. In order to make this classification clear at a glance, the arguments and return values are given the following pseudo types:

Pseudo type	Real C type	Constant?	Description
`uh`	`unsigned short`	No	an unsigned halfword
`uw1`	`unsigned int`	No	an unsigned word
`sw1`	`int`	No	a signed word
`uw2`	`unsigned long long`	No	an unsigned doubleword
`sw2`	`long long`	No	a signed doubleword
`const`	`int`	Yes	an integer constant
`acc`	`int`	Yes	an ACC register number
`iacc`	`int`	Yes	an IACC register number

These pseudo types are not defined by GCC, they are simply a notational convenience used in this manual.

Arguments of type `uh`, `uw1`, `sw1`, `uw2` and `sw2` are evaluated at run time. They correspond to register operands in the underlying FR-V instructions.

`const` arguments represent immediate operands in the underlying FR-V instructions. They must be compile-time constants.

`acc` arguments are evaluated at compile time and specify the number of an accumulator register. For example, an `acc` argument of 2 selects the ACC2 register.

`iacc` arguments are similar to `acc` arguments but specify the number of an IACC register. See see Section 6.60.13.5 [Other Built-in Functions], page 779, for more details.

6.60.13.2 Directly-Mapped Integer Functions

The functions listed below map directly to FR-V I-type instructions.

Function prototype	Example usage	Assembly output
`sw1 __ADDSS (sw1, sw1)`	`c = __ADDSS (a, b)`	`ADDSS a,b,c`
`sw1 __SCAN (sw1, sw1)`	`c = __SCAN (a, b)`	`SCAN a,b,c`
`sw1 __SCUTSS (sw1)`	`b = __SCUTSS (a)`	`SCUTSS a,b`
`sw1 __SLASS (sw1, sw1)`	`c = __SLASS (a, b)`	`SLASS a,b,c`
`void __SMASS (sw1, sw1)`	`__SMASS (a, b)`	`SMASS a,b`
`void __SMSSS (sw1, sw1)`	`__SMSSS (a, b)`	`SMSSS a,b`
`void __SMU (sw1, sw1)`	`__SMU (a, b)`	`SMU a,b`
`sw2 __SMUL (sw1, sw1)`	`c = __SMUL (a, b)`	`SMUL a,b,c`
`sw1 __SUBSS (sw1, sw1)`	`c = __SUBSS (a, b)`	`SUBSS a,b,c`
`uw2 __UMUL (uw1, uw1)`	`c = __UMUL (a, b)`	`UMUL a,b,c`

6.60.13.3 Directly-Mapped Media Functions

The functions listed below map directly to FR-V M-type instructions.

Function prototype	Example usage	Assembly output
`uw1 __MABSHS (sw1)`	`b = __MABSHS (a)`	`MABSHS a,b`
`void __MADDACCS (acc, acc)`	`__MADDACCS (b, a)`	`MADDACCS a,b`
`sw1 __MADDHSS (sw1, sw1)`	`c = __MADDHSS (a, b)`	`MADDHSS a,b,c`
`uw1 __MADDHUS (uw1, uw1)`	`c = __MADDHUS (a, b)`	`MADDHUS a,b,c`
`uw1 __MAND (uw1, uw1)`	`c = __MAND (a, b)`	`MAND a,b,c`
`void __MASACCS (acc, acc)`	`__MASACCS (b, a)`	`MASACCS a,b`
`uw1 __MAVEH (uw1, uw1)`	`c = __MAVEH (a, b)`	`MAVEH a,b,c`
`uw2 __MBTOH (uw1)`	`b = __MBTOH (a)`	`MBTOH a,b`
`void __MBTOHE (uw1 *, uw1)`	`__MBTOHE (&b, a)`	`MBTOHE a,b`
`void __MCLRACC (acc)`	`__MCLRACC (a)`	`MCLRACC a`
`void __MCLRACCA (void)`	`__MCLRACCA ()`	`MCLRACCA`
`uw1 __Mcop1 (uw1, uw1)`	`c = __Mcop1 (a, b)`	`Mcop1 a,b,c`
`uw1 __Mcop2 (uw1, uw1)`	`c = __Mcop2 (a, b)`	`Mcop2 a,b,c`
`uw1 __MCPLHI (uw2, const)`	`c = __MCPLHI (a, b)`	`MCPLHI a,#b,c`
`uw1 __MCPLI (uw2, const)`	`c = __MCPLI (a, b)`	`MCPLI a,#b,c`
`void __MCPXIS (acc, sw1, sw1)`	`__MCPXIS (c, a, b)`	`MCPXIS a,b,c`
`void __MCPXIU (acc, uw1, uw1)`	`__MCPXIU (c, a, b)`	`MCPXIU a,b,c`
`void __MCPXRS (acc, sw1, sw1)`	`__MCPXRS (c, a, b)`	`MCPXRS a,b,c`
`void __MCPXRU (acc, uw1, uw1)`	`__MCPXRU (c, a, b)`	`MCPXRU a,b,c`
`uw1 __MCUT (acc, uw1)`	`c = __MCUT (a, b)`	`MCUT a,b,c`
`uw1 __MCUTSS (acc, sw1)`	`c = __MCUTSS (a, b)`	`MCUTSS a,b,c`
`void __MDADDACCS (acc, acc)`	`__MDADDACCS (b, a)`	`MDADDACCS a,b`
`void __MDASACCS (acc, acc)`	`__MDASACCS (b, a)`	`MDASACCS a,b`
`uw2 __MDCUTSSI (acc, const)`	`c = __MDCUTSSI (a, b)`	`MDCUTSSI a,#b,c`
`uw2 __MDPACKH (uw2, uw2)`	`c = __MDPACKH (a, b)`	`MDPACKH a,b,c`
`uw2 __MDROTLI (uw2, const)`	`c = __MDROTLI (a, b)`	`MDROTLI a,#b,c`
`void __MDSUBACCS (acc, acc)`	`__MDSUBACCS (b, a)`	`MDSUBACCS a,b`
`void __MDUNPACKH (uw1 *, uw2)`	`__MDUNPACKH (&b, a)`	`MDUNPACKH a,b`
`uw2 __MEXPDHD (uw1, const)`	`c = __MEXPDHD (a, b)`	`MEXPDHD a,#b,c`
`uw1 __MEXPDHW (uw1, const)`	`c = __MEXPDHW (a, b)`	`MEXPDHW a,#b,c`
`uw1 __MHDSETH (uw1, const)`	`c = __MHDSETH (a, b)`	`MHDSETH a,#b,c`
`sw1 __MHDSETS (const)`	`b = __MHDSETS (a)`	`MHDSETS #a,b`
`uw1 __MHSETHIH (uw1, const)`	`b = __MHSETHIH (b, a)`	`MHSETHIH #a,b`
`sw1 __MHSETHIS (sw1, const)`	`b = __MHSETHIS (b, a)`	`MHSETHIS #a,b`
`uw1 __MHSETLOH (uw1, const)`	`b = __MHSETLOH (b, a)`	`MHSETLOH #a,b`
`sw1 __MHSETLOS (sw1, const)`	`b = __MHSETLOS (b, a)`	`MHSETLOS #a,b`
`uw1 __MHTOB (uw2)`	`b = __MHTOB (a)`	`MHTOB a,b`
`void __MMACHS (acc, sw1, sw1)`	`__MMACHS (c, a, b)`	`MMACHS a,b,c`
`void __MMACHU (acc, uw1, uw1)`	`__MMACHU (c, a, b)`	`MMACHU a,b,c`
`void __MMRDHS (acc, sw1, sw1)`	`__MMRDHS (c, a, b)`	`MMRDHS a,b,c`
`void __MMRDHU (acc, uw1, uw1)`	`__MMRDHU (c, a, b)`	`MMRDHU a,b,c`
`void __MMULHS (acc, sw1, sw1)`	`__MMULHS (c, a, b)`	`MMULHS a,b,c`
`void __MMULHU (acc, uw1, uw1)`	`__MMULHU (c, a, b)`	`MMULHU a,b,c`

void __MMULXHS (acc, sw1, sw1)	__MMULXHS (*c*, *a*, *b*)	MMULXHS *a*,*b*,*c*
void __MMULXHU (acc, uw1, uw1)	__MMULXHU (*c*, *a*, *b*)	MMULXHU *a*,*b*,*c*
uw1 __MNOT (uw1)	*b* = __MNOT (*a*)	MNOT *a*,*b*
uw1 __MOR (uw1, uw1)	*c* = __MOR (*a*, *b*)	MOR *a*,*b*,*c*
uw1 __MPACKH (uh, uh)	*c* = __MPACKH (*a*, *b*)	MPACKH *a*,*b*,*c*
sw2 __MQADDHSS (sw2, sw2)	*c* = __MQADDHSS (*a*, *b*)	MQADDHSS *a*,*b*,*c*
uw2 __MQADDHUS (uw2, uw2)	*c* = __MQADDHUS (*a*, *b*)	MQADDHUS *a*,*b*,*c*
void __MQCPXIS (acc, sw2, sw2)	__MQCPXIS (*c*, *a*, *b*)	MQCPXIS *a*,*b*,*c*
void __MQCPXIU (acc, uw2, uw2)	__MQCPXIU (*c*, *a*, *b*)	MQCPXIU *a*,*b*,*c*
void __MQCPXRS (acc, sw2, sw2)	__MQCPXRS (*c*, *a*, *b*)	MQCPXRS *a*,*b*,*c*
void __MQCPXRU (acc, uw2, uw2)	__MQCPXRU (*c*, *a*, *b*)	MQCPXRU *a*,*b*,*c*
sw2 __MQLCLRHS (sw2, sw2)	*c* = __MQLCLRHS (*a*, *b*)	MQLCLRHS *a*,*b*,*c*
sw2 __MQLMTHS (sw2, sw2)	*c* = __MQLMTHS (*a*, *b*)	MQLMTHS *a*,*b*,*c*
void __MQMACHS (acc, sw2, sw2)	__MQMACHS (*c*, *a*, *b*)	MQMACHS *a*,*b*,*c*
void __MQMACHU (acc, uw2, uw2)	__MQMACHU (*c*, *a*, *b*)	MQMACHU *a*,*b*,*c*
void __MQMACXHS (acc, sw2, sw2)	__MQMACXHS (*c*, *a*, *b*)	MQMACXHS *a*,*b*,*c*
void __MQMULHS (acc, sw2, sw2)	__MQMULHS (*c*, *a*, *b*)	MQMULHS *a*,*b*,*c*
void __MQMULHU (acc, uw2, uw2)	__MQMULHU (*c*, *a*, *b*)	MQMULHU *a*,*b*,*c*
void __MQMULXHS (acc, sw2, sw2)	__MQMULXHS (*c*, *a*, *b*)	MQMULXHS *a*,*b*,*c*
void __MQMULXHU (acc, uw2, uw2)	__MQMULXHU (*c*, *a*, *b*)	MQMULXHU *a*,*b*,*c*
sw2 __MQSATHS (sw2, sw2)	*c* = __MQSATHS (*a*, *b*)	MQSATHS *a*,*b*,*c*
uw2 __MQSLLHI (uw2, int)	*c* = __MQSLLHI (*a*, *b*)	MQSLLHI *a*,*b*,*c*
sw2 __MQSRAHI (sw2, int)	*c* = __MQSRAHI (*a*, *b*)	MQSRAHI *a*,*b*,*c*
sw2 __MQSUBHSS (sw2, sw2)	*c* = __MQSUBHSS (*a*, *b*)	MQSUBHSS *a*,*b*,*c*
uw2 __MQSUBHUS (uw2, uw2)	*c* = __MQSUBHUS (*a*, *b*)	MQSUBHUS *a*,*b*,*c*
void __MQXMACHS (acc, sw2, sw2)	__MQXMACHS (*c*, *a*, *b*)	MQXMACHS *a*,*b*,*c*
void __MQXMACXHS (acc, sw2, sw2)	__MQXMACXHS (*c*, *a*, *b*)	MQXMACXHS *a*,*b*,*c*
uw1 __MRDACC (acc)	*b* = __MRDACC (*a*)	MRDACC *a*,*b*
uw1 __MRDACCG (acc)	*b* = __MRDACCG (*a*)	MRDACCG *a*,*b*
uw1 __MROTLI (uw1, const)	*c* = __MROTLI (*a*, *b*)	MROTLI *a*,#*b*,*c*
uw1 __MROTRI (uw1, const)	*c* = __MROTRI (*a*, *b*)	MROTRI *a*,#*b*,*c*
sw1 __MSATHS (sw1, sw1)	*c* = __MSATHS (*a*, *b*)	MSATHS *a*,*b*,*c*
uw1 __MSATHU (uw1, uw1)	*c* = __MSATHU (*a*, *b*)	MSATHU *a*,*b*,*c*
uw1 __MSLLHI (uw1, const)	*c* = __MSLLHI (*a*, *b*)	MSLLHI *a*,#*b*,*c*
sw1 __MSRAHI (sw1, const)	*c* = __MSRAHI (*a*, *b*)	MSRAHI *a*,#*b*,*c*
uw1 __MSRLHI (uw1, const)	*c* = __MSRLHI (*a*, *b*)	MSRLHI *a*,#*b*,*c*
void __MSUBACCS (acc, acc)	__MSUBACCS (*b*, *a*)	MSUBACCS *a*,*b*
sw1 __MSUBHSS (sw1, sw1)	*c* = __MSUBHSS (*a*, *b*)	MSUBHSS *a*,*b*,*c*
uw1 __MSUBHUS (uw1, uw1)	*c* = __MSUBHUS (*a*, *b*)	MSUBHUS *a*,*b*,*c*
void __MTRAP (void)	__MTRAP ()	MTRAP
uw2 __MUNPACKH (uw1)	*b* = __MUNPACKH (*a*)	MUNPACKH *a*,*b*
uw1 __MWCUT (uw2, uw1)	*c* = __MWCUT (*a*, *b*)	MWCUT *a*,*b*,*c*
void __MWTACC (acc, uw1)	__MWTACC (*b*, *a*)	MWTACC *a*,*b*
void __MWTACCG (acc, uw1)	__MWTACCG (*b*, *a*)	MWTACCG *a*,*b*
uw1 __MXOR (uw1, uw1)	*c* = __MXOR (*a*, *b*)	MXOR *a*,*b*,*c*

6.60.13.4 Raw Read/Write Functions

This sections describes built-in functions related to read and write instructions to access memory. These functions generate `membar` instructions to flush the I/O load and stores where appropriate, as described in Fujitsu's manual described above.

```
unsigned char __builtin_read8 (void *data)
unsigned short __builtin_read16 (void *data)
unsigned long __builtin_read32 (void *data)
unsigned long long __builtin_read64 (void *data)
void __builtin_write8 (void *data, unsigned char datum)
void __builtin_write16 (void *data, unsigned short datum)
void __builtin_write32 (void *data, unsigned long datum)
void __builtin_write64 (void *data, unsigned long long datum)
```

6.60.13.5 Other Built-in Functions

This section describes built-in functions that are not named after a specific FR-V instruction.

`sw2 __IACCreadll (iacc reg)`
: Return the full 64-bit value of IACC0. The *reg* argument is reserved for future expansion and must be 0.

`sw1 __IACCreadl (iacc reg)`
: Return the value of IACC0H if *reg* is 0 and IACC0L if *reg* is 1. Other values of *reg* are rejected as invalid.

`void __IACCsetll (iacc reg, sw2 x)`
: Set the full 64-bit value of IACC0 to *x*. The *reg* argument is reserved for future expansion and must be 0.

`void __IACCsetl (iacc reg, sw1 x)`
: Set IACC0H to *x* if *reg* is 0 and IACC0L to *x* if *reg* is 1. Other values of *reg* are rejected as invalid.

`void __data_prefetch0 (const void *x)`
: Use the `dcpl` instruction to load the contents of address *x* into the data cache.

`void __data_prefetch (const void *x)`
: Use the `nldub` instruction to load the contents of address *x* into the data cache. The instruction is issued in slot I1.

6.60.14 LoongArch Base Built-in Functions

These built-in functions are available for LoongArch.

Data Type Description:

- `imm0_31`, a compile-time constant in range 0 to 31;
- `imm0_16383`, a compile-time constant in range 0 to 16383;
- `imm0_32767`, a compile-time constant in range 0 to 32767;
- `imm_n2048_2047`, a compile-time constant in range -2048 to 2047;

The intrinsics provided are listed below:

```
unsigned int __builtin_loongarch_movfcsr2gr (imm0_31)
void __builtin_loongarch_movgr2fcsr (imm0_31, unsigned int)
void __builtin_loongarch_cacop_d (imm0_31, unsigned long int, imm_n2048_2047)
unsigned int __builtin_loongarch_cpucfg (unsigned int)
void __builtin_loongarch_asrtle_d (long int, long int)
void __builtin_loongarch_asrtgt_d (long int, long int)
long int __builtin_loongarch_lddir_d (long int, imm0_31)
void __builtin_loongarch_ldpte_d (long int, imm0_31)

int __builtin_loongarch_crc_w_b_w (char, int)
int __builtin_loongarch_crc_w_h_w (short, int)
int __builtin_loongarch_crc_w_w_w (int, int)
int __builtin_loongarch_crc_w_d_w (long int, int)
int __builtin_loongarch_crcc_w_b_w (char, int)
int __builtin_loongarch_crcc_w_h_w (short, int)
int __builtin_loongarch_crcc_w_w_w (int, int)
int __builtin_loongarch_crcc_w_d_w (long int, int)

unsigned int __builtin_loongarch_csrrd_w (imm0_16383)
unsigned int __builtin_loongarch_csrwr_w (unsigned int, imm0_16383)
unsigned int __builtin_loongarch_csrxchg_w (unsigned int, unsigned int, imm0_16383)
unsigned long int __builtin_loongarch_csrrd_d (imm0_16383)
unsigned long int __builtin_loongarch_csrwr_d (unsigned long int, imm0_16383)
unsigned long int __builtin_loongarch_csrxchg_d (unsigned long int, unsigned long int, imm0_16383)

unsigned char __builtin_loongarch_iocsrrd_b (unsigned int)
unsigned short __builtin_loongarch_iocsrrd_h (unsigned int)
unsigned int __builtin_loongarch_iocsrrd_w (unsigned int)
unsigned long int __builtin_loongarch_iocsrrd_d (unsigned int)
void __builtin_loongarch_iocsrwr_b (unsigned char, unsigned int)
void __builtin_loongarch_iocsrwr_h (unsigned short, unsigned int)
void __builtin_loongarch_iocsrwr_w (unsigned int, unsigned int)
void __builtin_loongarch_iocsrwr_d (unsigned long int, unsigned int)

void __builtin_loongarch_dbar (imm0_32767)
void __builtin_loongarch_ibar (imm0_32767)

void __builtin_loongarch_syscall (imm0_32767)
void __builtin_loongarch_break (imm0_32767)
```

*Note:*Since the control register is divided into 32-bit and 64-bit, but the access instruction is not distinguished. So GCC renames the control instructions when implementing intrinsics.

Take the csrrd instruction as an example, built-in functions are implemented as follows:

```
__builtin_loongarch_csrrd_w  // When reading the 32-bit control register use.
__builtin_loongarch_csrrd_d  // When reading the 64-bit control register use.
```

For the convenience of use, the built-in functions are encapsulated, the encapsulated functions and `__drdtime_t, __rdtime_t` are defined in the `larchintrin.h`. So if you call the following function you need to include `larchintrin.h`.

```
typedef struct drdtime{
       unsigned long dvalue;
       unsigned long dtimeid;
} __drdtime_t;

typedef struct rdtime{
       unsigned int value;
```

```
       unsigned int timeid;
 } __rdtime_t;
__drdtime_t __rdtime_d (void)
__rdtime_t  __rdtimel_w (void)
__rdtime_t  __rdtimeh_w (void)
unsigned int  __movfcsr2gr (imm0_31)
void __movgr2fcsr (imm0_31, unsigned int)
void __cacop_d (imm0_31, unsigned long, imm_n2048_2047)
unsigned int  __cpucfg (unsigned int)
void __asrtle_d (long int, long int)
void __asrtgt_d (long int, long int)
long int  __lddir_d (long int, imm0_31)
void __ldpte_d (long int, imm0_31)

int  __crc_w_b_w (char, int)
int  __crc_w_h_w (short, int)
int  __crc_w_w_w (int, int)
int  __crc_w_d_w (long int, int)
int  __crcc_w_b_w (char, int)
int  __crcc_w_h_w (short, int)
int  __crcc_w_w_w (int, int)
int  __crcc_w_d_w (long int, int)

unsigned int  __csrrd_w (imm0_16383)
unsigned int  __csrwr_w (unsigned int, imm0_16383)
unsigned int  __csrxchg_w (unsigned int, unsigned int, imm0_16383)
unsigned long  __csrrd_d (imm0_16383)
unsigned long  __csrwr_d (unsigned long, imm0_16383)
unsigned long  __csrxchg_d (unsigned long, unsigned long, imm0_16383)

unsigned char   __iocsrrd_b (unsigned int)
unsigned short  __iocsrrd_h (unsigned int)
unsigned int  __iocsrrd_w (unsigned int)
unsigned long  __iocsrrd_d (unsigned int)
void __iocsrwr_b (unsigned char, unsigned int)
void __iocsrwr_h (unsigned short, unsigned int)
void __iocsrwr_w (unsigned int, unsigned int)
void __iocsrwr_d (unsigned long, unsigned int)

void __dbar (imm0_32767)
void __ibar (imm0_32767)

void __syscall (imm0_32767)
void __break (imm0_32767)
```

6.60.15 MIPS DSP Built-in Functions

The MIPS DSP Application-Specific Extension (ASE) includes new instructions that are designed to improve the performance of DSP and media applications. It provides instructions that operate on packed 8-bit/16-bit integer data, Q7, Q15 and Q31 fractional data.

GCC supports MIPS DSP operations using both the generic vector extensions (see Section 6.52 [Vector Extensions], page 726) and a collection of MIPS-specific built-in functions. Both kinds of support are enabled by the `-mdsp` command-line option.

Revision 2 of the ASE was introduced in the second half of 2006. This revision adds extra instructions to the original ASE, but is otherwise backwards-compatible with it. You can select revision 2 using the command-line option `-mdspr2`; this option implies `-mdsp`.

The SCOUNT and POS bits of the DSP control register are global. The WRDSP, EXTPDP, EXTPDPV and MTHLIP instructions modify the SCOUNT and POS bits. During optimization, the compiler does not delete these instructions and it does not delete calls to functions containing these instructions.

At present, GCC only provides support for operations on 32-bit vectors. The vector type associated with 8-bit integer data is usually called `v4i8`, the vector type associated with Q7 is usually called `v4q7`, the vector type associated with 16-bit integer data is usually called `v2i16`, and the vector type associated with Q15 is usually called `v2q15`. They can be defined in C as follows:

```
typedef signed char v4i8 __attribute__ ((vector_size(4)));
typedef signed char v4q7 __attribute__ ((vector_size(4)));
typedef short v2i16 __attribute__ ((vector_size(4)));
typedef short v2q15 __attribute__ ((vector_size(4)));
```

`v4i8`, `v4q7`, `v2i16` and `v2q15` values are initialized in the same way as aggregates. For example:

```
v4i8 a = {1, 2, 3, 4};
v4i8 b;
b = (v4i8) {5, 6, 7, 8};

v2q15 c = {0x0fcb, 0x3a75};
v2q15 d;
d = (v2q15) {0.1234 * 0x1.0p15, 0.4567 * 0x1.0p15};
```

Note: The CPU's endianness determines the order in which values are packed. On little-endian targets, the first value is the least significant and the last value is the most significant. The opposite order applies to big-endian targets. For example, the code above sets the lowest byte of `a` to `1` on little-endian targets and `4` on big-endian targets.

Note: Q7, Q15 and Q31 values must be initialized with their integer representation. As shown in this example, the integer representation of a Q7 value can be obtained by multiplying the fractional value by `0x1.0p7`. The equivalent for Q15 values is to multiply by `0x1.0p15`. The equivalent for Q31 values is to multiply by `0x1.0p31`.

The table below lists the `v4i8` and `v2q15` operations for which hardware support exists. `a` and `b` are `v4i8` values, and `c` and `d` are `v2q15` values.

C code	**MIPS instruction**
`a + b`	`addu.qb`
`c + d`	`addq.ph`
`a - b`	`subu.qb`
`c - d`	`subq.ph`

The table below lists the `v2i16` operation for which hardware support exists for the DSP ASE REV 2. `e` and `f` are `v2i16` values.

C code	**MIPS instruction**
`e * f`	`mul.ph`

It is easier to describe the DSP built-in functions if we first define the following types:

```
typedef int q31;
typedef int i32;
typedef unsigned int ui32;
typedef long long a64;
```

`q31` and `i32` are actually the same as `int`, but we use `q31` to indicate a Q31 fractional value and `i32` to indicate a 32-bit integer value. Similarly, `a64` is the same as `long long`, but we use `a64` to indicate values that are placed in one of the four DSP accumulators (`$ac0`, `$ac1`, `$ac2` or `$ac3`).

Also, some built-in functions prefer or require immediate numbers as parameters, because the corresponding DSP instructions accept both immediate numbers and register operands, or accept immediate numbers only. The immediate parameters are listed as follows.

```
imm0_3: 0 to 3.
imm0_7: 0 to 7.
imm0_15: 0 to 15.
imm0_31: 0 to 31.
imm0_63: 0 to 63.
imm0_255: 0 to 255.
imm_n32_31: -32 to 31.
imm_n512_511: -512 to 511.
```

The following built-in functions map directly to a particular MIPS DSP instruction. Please refer to the architecture specification for details on what each instruction does.

```
v2q15 __builtin_mips_addq_ph (v2q15, v2q15);
v2q15 __builtin_mips_addq_s_ph (v2q15, v2q15);
q31 __builtin_mips_addq_s_w (q31, q31);
v4i8 __builtin_mips_addu_qb (v4i8, v4i8);
v4i8 __builtin_mips_addu_s_qb (v4i8, v4i8);
v2q15 __builtin_mips_subq_ph (v2q15, v2q15);
v2q15 __builtin_mips_subq_s_ph (v2q15, v2q15);
q31 __builtin_mips_subq_s_w (q31, q31);
v4i8 __builtin_mips_subu_qb (v4i8, v4i8);
v4i8 __builtin_mips_subu_s_qb (v4i8, v4i8);
i32 __builtin_mips_addsc (i32, i32);
i32 __builtin_mips_addwc (i32, i32);
i32 __builtin_mips_modsub (i32, i32);
i32 __builtin_mips_raddu_w_qb (v4i8);
v2q15 __builtin_mips_absq_s_ph (v2q15);
q31 __builtin_mips_absq_s_w (q31);
v4i8 __builtin_mips_precrq_qb_ph (v2q15, v2q15);
v2q15 __builtin_mips_precrq_ph_w (q31, q31);
v2q15 __builtin_mips_precrq_rs_ph_w (q31, q31);
v4i8 __builtin_mips_precrqu_s_qb_ph (v2q15, v2q15);
q31 __builtin_mips_preceq_w_phl (v2q15);
q31 __builtin_mips_preceq_w_phr (v2q15);
v2q15 __builtin_mips_precequ_ph_qbl (v4i8);
v2q15 __builtin_mips_precequ_ph_qbr (v4i8);
v2q15 __builtin_mips_precequ_ph_qbla (v4i8);
v2q15 __builtin_mips_precequ_ph_qbra (v4i8);
v2q15 __builtin_mips_preceu_ph_qbl (v4i8);
v2q15 __builtin_mips_preceu_ph_qbr (v4i8);
v2q15 __builtin_mips_preceu_ph_qbla (v4i8);
v2q15 __builtin_mips_preceu_ph_qbra (v4i8);
v4i8 __builtin_mips_shll_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shll_qb (v4i8, i32);
v2q15 __builtin_mips_shll_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shll_ph (v2q15, i32);
v2q15 __builtin_mips_shll_s_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shll_s_ph (v2q15, i32);
q31 __builtin_mips_shll_s_w (q31, imm0_31);
q31 __builtin_mips_shll_s_w (q31, i32);
```

```
v4i8 __builtin_mips_shrl_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shrl_qb (v4i8, i32);
v2q15 __builtin_mips_shra_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shra_ph (v2q15, i32);
v2q15 __builtin_mips_shra_r_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shra_r_ph (v2q15, i32);
q31 __builtin_mips_shra_r_w (q31, imm0_31);
q31 __builtin_mips_shra_r_w (q31, i32);
v2q15 __builtin_mips_muleu_s_ph_qbl (v4i8, v2q15);
v2q15 __builtin_mips_muleu_s_ph_qbr (v4i8, v2q15);
v2q15 __builtin_mips_mulq_rs_ph (v2q15, v2q15);
q31 __builtin_mips_muleq_s_w_phl (v2q15, v2q15);
q31 __builtin_mips_muleq_s_w_phr (v2q15, v2q15);
a64 __builtin_mips_dpau_h_qbl (a64, v4i8, v4i8);
a64 __builtin_mips_dpau_h_qbr (a64, v4i8, v4i8);
a64 __builtin_mips_dpsu_h_qbl (a64, v4i8, v4i8);
a64 __builtin_mips_dpsu_h_qbr (a64, v4i8, v4i8);
a64 __builtin_mips_dpaq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpaq_sa_l_w (a64, q31, q31);
a64 __builtin_mips_dpsq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsq_sa_l_w (a64, q31, q31);
a64 __builtin_mips_mulsaq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_maq_s_w_phl (a64, v2q15, v2q15);
a64 __builtin_mips_maq_s_w_phr (a64, v2q15, v2q15);
a64 __builtin_mips_maq_sa_w_phl (a64, v2q15, v2q15);
a64 __builtin_mips_maq_sa_w_phr (a64, v2q15, v2q15);
i32 __builtin_mips_bitrev (i32);
i32 __builtin_mips_insv (i32, i32);
v4i8 __builtin_mips_repl_qb (imm0_255);
v4i8 __builtin_mips_repl_qb (i32);
v2q15 __builtin_mips_repl_ph (imm_n512_511);
v2q15 __builtin_mips_repl_ph (i32);
void __builtin_mips_cmpu_eq_qb (v4i8, v4i8);
void __builtin_mips_cmpu_lt_qb (v4i8, v4i8);
void __builtin_mips_cmpu_le_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_eq_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_lt_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_le_qb (v4i8, v4i8);
void __builtin_mips_cmp_eq_ph (v2q15, v2q15);
void __builtin_mips_cmp_lt_ph (v2q15, v2q15);
void __builtin_mips_cmp_le_ph (v2q15, v2q15);
v4i8 __builtin_mips_pick_qb (v4i8, v4i8);
v2q15 __builtin_mips_pick_ph (v2q15, v2q15);
v2q15 __builtin_mips_packrl_ph (v2q15, v2q15);
i32 __builtin_mips_extr_w (a64, imm0_31);
i32 __builtin_mips_extr_w (a64, i32);
i32 __builtin_mips_extr_r_w (a64, imm0_31);
i32 __builtin_mips_extr_s_h (a64, i32);
i32 __builtin_mips_extr_rs_w (a64, imm0_31);
i32 __builtin_mips_extr_rs_w (a64, i32);
i32 __builtin_mips_extr_s_h (a64, imm0_31);
i32 __builtin_mips_extr_r_w (a64, i32);
i32 __builtin_mips_extp (a64, imm0_31);
i32 __builtin_mips_extp (a64, i32);
i32 __builtin_mips_extpdp (a64, imm0_31);
i32 __builtin_mips_extpdp (a64, i32);
a64 __builtin_mips_shilo (a64, imm_n32_31);
a64 __builtin_mips_shilo (a64, i32);
```

```
a64 __builtin_mips_mthlip (a64, i32);
void __builtin_mips_wrdsp (i32, imm0_63);
i32 __builtin_mips_rddsp (imm0_63);
i32 __builtin_mips_lbux (void *, i32);
i32 __builtin_mips_lhx (void *, i32);
i32 __builtin_mips_lwx (void *, i32);
a64 __builtin_mips_ldx (void *, i32); /* MIPS64 only */
i32 __builtin_mips_bposge32 (void);
a64 __builtin_mips_madd (a64, i32, i32);
a64 __builtin_mips_maddu (a64, ui32, ui32);
a64 __builtin_mips_msub (a64, i32, i32);
a64 __builtin_mips_msubu (a64, ui32, ui32);
a64 __builtin_mips_mult (i32, i32);
a64 __builtin_mips_multu (ui32, ui32);
```

The following built-in functions map directly to a particular MIPS DSP REV 2 instruction. Please refer to the architecture specification for details on what each instruction does.

```
v4q7 __builtin_mips_absq_s_qb (v4q7);
v2i16 __builtin_mips_addu_ph (v2i16, v2i16);
v2i16 __builtin_mips_addu_s_ph (v2i16, v2i16);
v4i8 __builtin_mips_adduh_qb (v4i8, v4i8);
v4i8 __builtin_mips_adduh_r_qb (v4i8, v4i8);
i32 __builtin_mips_append (i32, i32, imm0_31);
i32 __builtin_mips_balign (i32, i32, imm0_3);
i32 __builtin_mips_cmpgdu_eq_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgdu_lt_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgdu_le_qb (v4i8, v4i8);
a64 __builtin_mips_dpa_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dps_w_ph (a64, v2i16, v2i16);
v2i16 __builtin_mips_mul_ph (v2i16, v2i16);
v2i16 __builtin_mips_mul_s_ph (v2i16, v2i16);
q31 __builtin_mips_mulq_rs_w (q31, q31);
v2q15 __builtin_mips_mulq_s_ph (v2q15, v2q15);
q31 __builtin_mips_mulq_s_w (q31, q31);
a64 __builtin_mips_mulsa_w_ph (a64, v2i16, v2i16);
v4i8 __builtin_mips_precr_qb_ph (v2i16, v2i16);
v2i16 __builtin_mips_precr_sra_ph_w (i32, i32, imm0_31);
v2i16 __builtin_mips_precr_sra_r_ph_w (i32, i32, imm0_31);
i32 __builtin_mips_prepend (i32, i32, imm0_31);
v4i8 __builtin_mips_shra_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shra_r_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shra_qb (v4i8, i32);
v4i8 __builtin_mips_shra_r_qb (v4i8, i32);
v2i16 __builtin_mips_shrl_ph (v2i16, imm0_15);
v2i16 __builtin_mips_shrl_ph (v2i16, i32);
v2i16 __builtin_mips_subu_ph (v2i16, v2i16);
v2i16 __builtin_mips_subu_s_ph (v2i16, v2i16);
v4i8 __builtin_mips_subuh_qb (v4i8, v4i8);
v4i8 __builtin_mips_subuh_r_qb (v4i8, v4i8);
v2q15 __builtin_mips_addqh_ph (v2q15, v2q15);
v2q15 __builtin_mips_addqh_r_ph (v2q15, v2q15);
q31 __builtin_mips_addqh_w (q31, q31);
q31 __builtin_mips_addqh_r_w (q31, q31);
v2q15 __builtin_mips_subqh_ph (v2q15, v2q15);
v2q15 __builtin_mips_subqh_r_ph (v2q15, v2q15);
q31 __builtin_mips_subqh_w (q31, q31);
q31 __builtin_mips_subqh_r_w (q31, q31);
a64 __builtin_mips_dpax_w_ph (a64, v2i16, v2i16);
```

```
a64 __builtin_mips_dpsx_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dpaqx_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpaqx_sa_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsqx_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsqx_sa_w_ph (a64, v2q15, v2q15);
```

6.60.16 MIPS Paired-Single Support

The MIPS64 architecture includes a number of instructions that operate on pairs of single-precision floating-point values. Each pair is packed into a 64-bit floating-point register, with one element being designated the "upper half" and the other being designated the "lower half".

GCC supports paired-single operations using both the generic vector extensions (see Section 6.52 [Vector Extensions], page 726) and a collection of MIPS-specific built-in functions. Both kinds of support are enabled by the `-mpaired-single` command-line option.

The vector type associated with paired-single values is usually called `v2sf`. It can be defined in C as follows:

```
typedef float v2sf __attribute__ ((vector_size (8)));
```

`v2sf` values are initialized in the same way as aggregates. For example:

```
v2sf a = {1.5, 9.1};
v2sf b;
float e, f;
b = (v2sf) {e, f};
```

Note: The CPU's endianness determines which value is stored in the upper half of a register and which value is stored in the lower half. On little-endian targets, the first value is the lower one and the second value is the upper one. The opposite order applies to big-endian targets. For example, the code above sets the lower half of `a` to `1.5` on little-endian targets and `9.1` on big-endian targets.

6.60.17 MIPS Loongson Built-in Functions

GCC provides intrinsics to access the SIMD instructions provided by the ST Microelectronics Loongson-2E and -2F processors. These intrinsics, available after inclusion of the `loongson.h` header file, operate on the following 64-bit vector types:

- `uint8x8_t`, a vector of eight unsigned 8-bit integers;
- `uint16x4_t`, a vector of four unsigned 16-bit integers;
- `uint32x2_t`, a vector of two unsigned 32-bit integers;
- `int8x8_t`, a vector of eight signed 8-bit integers;
- `int16x4_t`, a vector of four signed 16-bit integers;
- `int32x2_t`, a vector of two signed 32-bit integers.

The intrinsics provided are listed below; each is named after the machine instruction to which it corresponds, with suffixes added as appropriate to distinguish intrinsics that expand to the same machine instruction yet have different argument types. Refer to the architecture documentation for a description of the functionality of each instruction.

```
int16x4_t packsswh (int32x2_t s, int32x2_t t);
int8x8_t packsshb (int16x4_t s, int16x4_t t);
uint8x8_t packushb (uint16x4_t s, uint16x4_t t);
uint32x2_t paddw_u (uint32x2_t s, uint32x2_t t);
```

```
uint16x4_t paddh_u (uint16x4_t s, uint16x4_t t);
uint8x8_t paddb_u (uint8x8_t s, uint8x8_t t);
int32x2_t paddw_s (int32x2_t s, int32x2_t t);
int16x4_t paddh_s (int16x4_t s, int16x4_t t);
int8x8_t paddb_s (int8x8_t s, int8x8_t t);
uint64_t paddd_u (uint64_t s, uint64_t t);
int64_t paddd_s (int64_t s, int64_t t);
int16x4_t paddsh (int16x4_t s, int16x4_t t);
int8x8_t paddsb (int8x8_t s, int8x8_t t);
uint16x4_t paddush (uint16x4_t s, uint16x4_t t);
uint8x8_t paddusb (uint8x8_t s, uint8x8_t t);
uint64_t pandn_ud (uint64_t s, uint64_t t);
uint32x2_t pandn_uw (uint32x2_t s, uint32x2_t t);
uint16x4_t pandn_uh (uint16x4_t s, uint16x4_t t);
uint8x8_t pandn_ub (uint8x8_t s, uint8x8_t t);
int64_t pandn_sd (int64_t s, int64_t t);
int32x2_t pandn_sw (int32x2_t s, int32x2_t t);
int16x4_t pandn_sh (int16x4_t s, int16x4_t t);
int8x8_t pandn_sb (int8x8_t s, int8x8_t t);
uint16x4_t pavgh (uint16x4_t s, uint16x4_t t);
uint8x8_t pavgb (uint8x8_t s, uint8x8_t t);
uint32x2_t pcmpeqw_u (uint32x2_t s, uint32x2_t t);
uint16x4_t pcmpeqh_u (uint16x4_t s, uint16x4_t t);
uint8x8_t pcmpeqb_u (uint8x8_t s, uint8x8_t t);
int32x2_t pcmpeqw_s (int32x2_t s, int32x2_t t);
int16x4_t pcmpeqh_s (int16x4_t s, int16x4_t t);
int8x8_t pcmpeqb_s (int8x8_t s, int8x8_t t);
uint32x2_t pcmpgtw_u (uint32x2_t s, uint32x2_t t);
uint16x4_t pcmpgth_u (uint16x4_t s, uint16x4_t t);
uint8x8_t pcmpgtb_u (uint8x8_t s, uint8x8_t t);
int32x2_t pcmpgtw_s (int32x2_t s, int32x2_t t);
int16x4_t pcmpgth_s (int16x4_t s, int16x4_t t);
int8x8_t pcmpgtb_s (int8x8_t s, int8x8_t t);
uint16x4_t pextrh_u (uint16x4_t s, int field);
int16x4_t pextrh_s (int16x4_t s, int field);
uint16x4_t pinsrh_0_u (uint16x4_t s, uint16x4_t t);
uint16x4_t pinsrh_1_u (uint16x4_t s, uint16x4_t t);
uint16x4_t pinsrh_2_u (uint16x4_t s, uint16x4_t t);
uint16x4_t pinsrh_3_u (uint16x4_t s, uint16x4_t t);
int16x4_t pinsrh_0_s (int16x4_t s, int16x4_t t);
int16x4_t pinsrh_1_s (int16x4_t s, int16x4_t t);
int16x4_t pinsrh_2_s (int16x4_t s, int16x4_t t);
int16x4_t pinsrh_3_s (int16x4_t s, int16x4_t t);
int32x2_t pmaddhw (int16x4_t s, int16x4_t t);
int16x4_t pmaxsh (int16x4_t s, int16x4_t t);
uint8x8_t pmaxub (uint8x8_t s, uint8x8_t t);
int16x4_t pminsh (int16x4_t s, int16x4_t t);
uint8x8_t pminub (uint8x8_t s, uint8x8_t t);
uint8x8_t pmovmskb_u (uint8x8_t s);
int8x8_t pmovmskb_s (int8x8_t s);
uint16x4_t pmulhuh (uint16x4_t s, uint16x4_t t);
int16x4_t pmulhh (int16x4_t s, int16x4_t t);
int16x4_t pmullh (int16x4_t s, int16x4_t t);
int64_t pmuluw (uint32x2_t s, uint32x2_t t);
uint8x8_t pasubub (uint8x8_t s, uint8x8_t t);
uint16x4_t biadd (uint8x8_t s);
uint16x4_t psadbh (uint8x8_t s, uint8x8_t t);
uint16x4_t pshufh_u (uint16x4_t dest, uint16x4_t s, uint8_t order);
```

```
int16x4_t pshufh_s (int16x4_t dest, int16x4_t s, uint8_t order);
uint16x4_t psllh_u (uint16x4_t s, uint8_t amount);
int16x4_t psllh_s (int16x4_t s, uint8_t amount);
uint32x2_t psllw_u (uint32x2_t s, uint8_t amount);
int32x2_t psllw_s (int32x2_t s, uint8_t amount);
uint16x4_t psrlh_u (uint16x4_t s, uint8_t amount);
int16x4_t psrlh_s (int16x4_t s, uint8_t amount);
uint32x2_t psrlw_u (uint32x2_t s, uint8_t amount);
int32x2_t psrlw_s (int32x2_t s, uint8_t amount);
uint16x4_t psrah_u (uint16x4_t s, uint8_t amount);
int16x4_t psrah_s (int16x4_t s, uint8_t amount);
uint32x2_t psraw_u (uint32x2_t s, uint8_t amount);
int32x2_t psraw_s (int32x2_t s, uint8_t amount);
uint32x2_t psubw_u (uint32x2_t s, uint32x2_t t);
uint16x4_t psubh_u (uint16x4_t s, uint16x4_t t);
uint8x8_t psubb_u (uint8x8_t s, uint8x8_t t);
int32x2_t psubw_s (int32x2_t s, int32x2_t t);
int16x4_t psubh_s (int16x4_t s, int16x4_t t);
int8x8_t psubb_s (int8x8_t s, int8x8_t t);
uint64_t psubd_u (uint64_t s, uint64_t t);
int64_t psubd_s (int64_t s, int64_t t);
int16x4_t psubsh (int16x4_t s, int16x4_t t);
int8x8_t psubsb (int8x8_t s, int8x8_t t);
uint16x4_t psubush (uint16x4_t s, uint16x4_t t);
uint8x8_t psubusb (uint8x8_t s, uint8x8_t t);
uint32x2_t punpckhwd_u (uint32x2_t s, uint32x2_t t);
uint16x4_t punpckhhw_u (uint16x4_t s, uint16x4_t t);
uint8x8_t punpckhbh_u (uint8x8_t s, uint8x8_t t);
int32x2_t punpckhwd_s (int32x2_t s, int32x2_t t);
int16x4_t punpckhhw_s (int16x4_t s, int16x4_t t);
int8x8_t punpckhbh_s (int8x8_t s, int8x8_t t);
uint32x2_t punpcklwd_u (uint32x2_t s, uint32x2_t t);
uint16x4_t punpcklhw_u (uint16x4_t s, uint16x4_t t);
uint8x8_t punpcklbh_u (uint8x8_t s, uint8x8_t t);
int32x2_t punpcklwd_s (int32x2_t s, int32x2_t t);
int16x4_t punpcklhw_s (int16x4_t s, int16x4_t t);
int8x8_t punpcklbh_s (int8x8_t s, int8x8_t t);
```

6.60.17.1 Paired-Single Arithmetic

The table below lists the `v2sf` operations for which hardware support exists. `a`, `b` and `c` are `v2sf` values and `x` is an integral value.

C code	MIPS instruction
`a + b`	`add.ps`
`a - b`	`sub.ps`
`-a`	`neg.ps`
`a * b`	`mul.ps`
`a * b + c`	`madd.ps`
`a * b - c`	`msub.ps`
`-(a * b + c)`	`nmadd.ps`
`-(a * b - c)`	`nmsub.ps`
`x ? a : b`	`movn.ps/movz.ps`

Note that the multiply-accumulate instructions can be disabled using the command-line option `-mno-fused-madd`.

6.60.17.2 Paired-Single Built-in Functions

The following paired-single functions map directly to a particular MIPS instruction. Please refer to the architecture specification for details on what each instruction does.

`v2sf __builtin_mips_pll_ps (v2sf, v2sf)`
: Pair lower lower (`pll.ps`).

`v2sf __builtin_mips_pul_ps (v2sf, v2sf)`
: Pair upper lower (`pul.ps`).

`v2sf __builtin_mips_plu_ps (v2sf, v2sf)`
: Pair lower upper (`plu.ps`).

`v2sf __builtin_mips_puu_ps (v2sf, v2sf)`
: Pair upper upper (`puu.ps`).

`v2sf __builtin_mips_cvt_ps_s (float, float)`
: Convert pair to paired single (`cvt.ps.s`).

`float __builtin_mips_cvt_s_pl (v2sf)`
: Convert pair lower to single (`cvt.s.pl`).

`float __builtin_mips_cvt_s_pu (v2sf)`
: Convert pair upper to single (`cvt.s.pu`).

`v2sf __builtin_mips_abs_ps (v2sf)`
: Absolute value (`abs.ps`).

`v2sf __builtin_mips_alnv_ps (v2sf, v2sf, int)`
: Align variable (`alnv.ps`).

  Note: The value of the third parameter must be 0 or 4 modulo 8, otherwise the result is unpredictable. Please read the instruction description for details.

The following multi-instruction functions are also available. In each case, *cond* can be any of the 16 floating-point conditions: `f`, `un`, `eq`, `ueq`, `olt`, `ult`, `ole`, `ule`, `sf`, `ngle`, `seq`, `ngl`, `lt`, `nge`, `le` or `ngt`.

`v2sf __builtin_mips_movt_c_`*cond*`_ps (v2sf a, v2sf b, v2sf c, v2sf d)`
`v2sf __builtin_mips_movf_c_`*cond*`_ps (v2sf a, v2sf b, v2sf c, v2sf d)`
: Conditional move based on floating-point comparison (`c.`*cond*`.ps`, `movt.ps`/`movf.ps`).

  The `movt` functions return the value *x* computed by:

```
c.cond.ps cc,a,b
mov.ps x,c
movt.ps x,d,cc
```

  The `movf` functions are similar but use `movf.ps` instead of `movt.ps`.

`int __builtin_mips_upper_c_`*cond*`_ps (v2sf a, v2sf b)`
`int __builtin_mips_lower_c_`*cond*`_ps (v2sf a, v2sf b)`
: Comparison of two paired-single values (`c.`*cond*`.ps`, `bc1t`/`bc1f`).

  These functions compare *a* and *b* using `c.`*cond*`.ps` and return either the upper or lower half of the result. For example:

```
v2sf a, b;
```

```
if (__builtin_mips_upper_c_eq_ps (a, b))
  upper_halves_are_equal ();
else
  upper_halves_are_unequal ();

if (__builtin_mips_lower_c_eq_ps (a, b))
  lower_halves_are_equal ();
else
  lower_halves_are_unequal ();
```

6.60.17.3 MIPS-3D Built-in Functions

The MIPS-3D Application-Specific Extension (ASE) includes additional paired-single instructions that are designed to improve the performance of 3D graphics operations. Support for these instructions is controlled by the `-mips3d` command-line option.

The functions listed below map directly to a particular MIPS-3D instruction. Please refer to the architecture specification for more details on what each instruction does.

`v2sf __builtin_mips_addr_ps (v2sf, v2sf)`
: Reduction add (`addr.ps`).

`v2sf __builtin_mips_mulr_ps (v2sf, v2sf)`
: Reduction multiply (`mulr.ps`).

`v2sf __builtin_mips_cvt_pw_ps (v2sf)`
: Convert paired single to paired word (`cvt.pw.ps`).

`v2sf __builtin_mips_cvt_ps_pw (v2sf)`
: Convert paired word to paired single (`cvt.ps.pw`).

`float __builtin_mips_recip1_s (float)`
`double __builtin_mips_recip1_d (double)`
`v2sf __builtin_mips_recip1_ps (v2sf)`
: Reduced-precision reciprocal (sequence step 1) (`recip1.`*`fmt`*).

`float __builtin_mips_recip2_s (float, float)`
`double __builtin_mips_recip2_d (double, double)`
`v2sf __builtin_mips_recip2_ps (v2sf, v2sf)`
: Reduced-precision reciprocal (sequence step 2) (`recip2.`*`fmt`*).

`float __builtin_mips_rsqrt1_s (float)`
`double __builtin_mips_rsqrt1_d (double)`
`v2sf __builtin_mips_rsqrt1_ps (v2sf)`
: Reduced-precision reciprocal square root (sequence step 1) (`rsqrt1.`*`fmt`*).

`float __builtin_mips_rsqrt2_s (float, float)`
`double __builtin_mips_rsqrt2_d (double, double)`
`v2sf __builtin_mips_rsqrt2_ps (v2sf, v2sf)`
: Reduced-precision reciprocal square root (sequence step 2) (`rsqrt2.`*`fmt`*).

The following multi-instruction functions are also available. In each case, *cond* can be any of the 16 floating-point conditions: `f`, `un`, `eq`, `ueq`, `olt`, `ult`, `ole`, `ule`, `sf`, `ngle`, `seq`, `ngl`, `lt`, `nge`, `le` or `ngt`.

`int __builtin_mips_cabs_`*`cond`*`_s (float a, float b)`
`int __builtin_mips_cabs_`*`cond`*`_d (double a, double b)`

Absolute comparison of two scalar values (`cabs.`*`cond`*`.`*`fmt`*, `bc1t`/`bc1f`).

These functions compare *a* and *b* using `cabs.`*`cond`*`.s` or `cabs.`*`cond`*`.d` and return the result as a boolean value. For example:

```
float a, b;
if (__builtin_mips_cabs_eq_s (a, b))
  true ();
else
  false ();
```

`int __builtin_mips_upper_cabs_`*`cond`*`_ps (v2sf a, v2sf b)`
`int __builtin_mips_lower_cabs_`*`cond`*`_ps (v2sf a, v2sf b)`

Absolute comparison of two paired-single values (`cabs.`*`cond`*`.ps`, `bc1t`/`bc1f`).

These functions compare *a* and *b* using `cabs.`*`cond`*`.ps` and return either the upper or lower half of the result. For example:

```
v2sf a, b;
if (__builtin_mips_upper_cabs_eq_ps (a, b))
  upper_halves_are_equal ();
else
  upper_halves_are_unequal ();

if (__builtin_mips_lower_cabs_eq_ps (a, b))
  lower_halves_are_equal ();
else
  lower_halves_are_unequal ();
```

`v2sf __builtin_mips_movt_cabs_`*`cond`*`_ps (v2sf a, v2sf b, v2sf c, v2sf d)`
`v2sf __builtin_mips_movf_cabs_`*`cond`*`_ps (v2sf a, v2sf b, v2sf c, v2sf d)`

Conditional move based on absolute comparison (`cabs.`*`cond`*`.ps`, `movt.ps`/`movf.ps`).

The `movt` functions return the value *x* computed by:

```
cabs.cond.ps cc,a,b
mov.ps x,c
movt.ps x,d,cc
```

The `movf` functions are similar but use `movf.ps` instead of `movt.ps`.

`int __builtin_mips_any_c_`*`cond`*`_ps (v2sf a, v2sf b)`
`int __builtin_mips_all_c_`*`cond`*`_ps (v2sf a, v2sf b)`
`int __builtin_mips_any_cabs_`*`cond`*`_ps (v2sf a, v2sf b)`
`int __builtin_mips_all_cabs_`*`cond`*`_ps (v2sf a, v2sf b)`

Comparison of two paired-single values (`c.`*`cond`*`.ps`/`cabs.`*`cond`*`.ps`, `bc1any2t`/`bc1any2f`).

These functions compare *a* and *b* using `c.`*`cond`*`.ps` or `cabs.`*`cond`*`.ps`. The `any` forms return `true` if either result is `true` and the `all` forms return `true` if both results are `true`. For example:

```
v2sf a, b;
if (__builtin_mips_any_c_eq_ps (a, b))
  one_is_true ();
else
  both_are_false ();
```

```
if (__builtin_mips_all_c_eq_ps (a, b))
  both_are_true ();
else
  one_is_false ();
```

`int __builtin_mips_any_c_`*cond*`_4s (v2sf a, v2sf b, v2sf c, v2sf d)`
`int __builtin_mips_all_c_`*cond*`_4s (v2sf a, v2sf b, v2sf c, v2sf d)`
`int __builtin_mips_any_cabs_`*cond*`_4s (v2sf a, v2sf b, v2sf c, v2sf d)`
`int __builtin_mips_all_cabs_`*cond*`_4s (v2sf a, v2sf b, v2sf c, v2sf d)`

Comparison of four paired-single values (`c.`*cond*`.ps`/`cabs.`*cond*`.ps`, `bc1any4t`/`bc1any4f`).

These functions use `c.`*cond*`.ps` or `cabs.`*cond*`.ps` to compare *a* with *b* and to compare *c* with *d*. The `any` forms return `true` if any of the four results are `true` and the `all` forms return `true` if all four results are `true`. For example:

```
v2sf a, b, c, d;
if (__builtin_mips_any_c_eq_4s (a, b, c, d))
  some_are_true ();
else
  all_are_false ();

if (__builtin_mips_all_c_eq_4s (a, b, c, d))
  all_are_true ();
else
  some_are_false ();
```

6.60.18 MIPS SIMD Architecture (MSA) Support

GCC provides intrinsics to access the SIMD instructions provided by the MSA MIPS SIMD Architecture. The interface is made available by including `<msa.h>` and using `-mmsa -mhard-float -mfp64 -mnan=2008`. For each `__builtin_msa_*`, there is a shortened name of the intrinsic, `__msa_*`.

MSA implements 128-bit wide vector registers, operating on 8-, 16-, 32- and 64-bit integer, 16- and 32-bit fixed-point, or 32- and 64-bit floating point data elements. The following vectors typedefs are included in `msa.h`:

- `v16i8`, a vector of sixteen signed 8-bit integers;
- `v16u8`, a vector of sixteen unsigned 8-bit integers;
- `v8i16`, a vector of eight signed 16-bit integers;
- `v8u16`, a vector of eight unsigned 16-bit integers;
- `v4i32`, a vector of four signed 32-bit integers;
- `v4u32`, a vector of four unsigned 32-bit integers;
- `v2i64`, a vector of two signed 64-bit integers;
- `v2u64`, a vector of two unsigned 64-bit integers;
- `v4f32`, a vector of four 32-bit floats;
- `v2f64`, a vector of two 64-bit doubles.

Instructions and corresponding built-ins may have additional restrictions and/or input/output values manipulated:

- `imm0_1`, an integer literal in range 0 to 1;

- `imm0_3`, an integer literal in range 0 to 3;
- `imm0_7`, an integer literal in range 0 to 7;
- `imm0_15`, an integer literal in range 0 to 15;
- `imm0_31`, an integer literal in range 0 to 31;
- `imm0_63`, an integer literal in range 0 to 63;
- `imm0_255`, an integer literal in range 0 to 255;
- `imm_n16_15`, an integer literal in range -16 to 15;
- `imm_n512_511`, an integer literal in range -512 to 511;
- `imm_n1024_1022`, an integer literal in range -512 to 511 left shifted by 1 bit, i.e., -1024, -1022, . . . , 1020, 1022;
- `imm_n2048_2044`, an integer literal in range -512 to 511 left shifted by 2 bits, i.e., -2048, -2044, . . . , 2040, 2044;
- `imm_n4096_4088`, an integer literal in range -512 to 511 left shifted by 3 bits, i.e., -4096, -4088, . . . , 4080, 4088;
- `imm1_4`, an integer literal in range 1 to 4;
- `i32, i64, u32, u64, f32, f64`, defined as follows:

```
{
typedef int i32;
#if __LONG_MAX__ == __LONG_LONG_MAX__
typedef long i64;
#else
typedef long long i64;
#endif

typedef unsigned int u32;
#if __LONG_MAX__ == __LONG_LONG_MAX__
typedef unsigned long u64;
#else
typedef unsigned long long u64;
#endif

typedef double f64;
typedef float f32;
}
```

6.60.18.1 MIPS SIMD Architecture Built-in Functions

The intrinsics provided are listed below; each is named after the machine instruction.

```
v16i8 __builtin_msa_add_a_b (v16i8, v16i8);
v8i16 __builtin_msa_add_a_h (v8i16, v8i16);
v4i32 __builtin_msa_add_a_w (v4i32, v4i32);
v2i64 __builtin_msa_add_a_d (v2i64, v2i64);

v16i8 __builtin_msa_adds_a_b (v16i8, v16i8);
v8i16 __builtin_msa_adds_a_h (v8i16, v8i16);
v4i32 __builtin_msa_adds_a_w (v4i32, v4i32);
v2i64 __builtin_msa_adds_a_d (v2i64, v2i64);

v16i8 __builtin_msa_adds_s_b (v16i8, v16i8);
v8i16 __builtin_msa_adds_s_h (v8i16, v8i16);
v4i32 __builtin_msa_adds_s_w (v4i32, v4i32);
```

```
v2i64 __builtin_msa_adds_s_d (v2i64, v2i64);

v16u8 __builtin_msa_adds_u_b (v16u8, v16u8);
v8u16 __builtin_msa_adds_u_h (v8u16, v8u16);
v4u32 __builtin_msa_adds_u_w (v4u32, v4u32);
v2u64 __builtin_msa_adds_u_d (v2u64, v2u64);

v16i8 __builtin_msa_addv_b (v16i8, v16i8);
v8i16 __builtin_msa_addv_h (v8i16, v8i16);
v4i32 __builtin_msa_addv_w (v4i32, v4i32);
v2i64 __builtin_msa_addv_d (v2i64, v2i64);

v16i8 __builtin_msa_addvi_b (v16i8, imm0_31);
v8i16 __builtin_msa_addvi_h (v8i16, imm0_31);
v4i32 __builtin_msa_addvi_w (v4i32, imm0_31);
v2i64 __builtin_msa_addvi_d (v2i64, imm0_31);

v16u8 __builtin_msa_and_v (v16u8, v16u8);

v16u8 __builtin_msa_andi_b (v16u8, imm0_255);

v16i8 __builtin_msa_asub_s_b (v16i8, v16i8);
v8i16 __builtin_msa_asub_s_h (v8i16, v8i16);
v4i32 __builtin_msa_asub_s_w (v4i32, v4i32);
v2i64 __builtin_msa_asub_s_d (v2i64, v2i64);

v16u8 __builtin_msa_asub_u_b (v16u8, v16u8);
v8u16 __builtin_msa_asub_u_h (v8u16, v8u16);
v4u32 __builtin_msa_asub_u_w (v4u32, v4u32);
v2u64 __builtin_msa_asub_u_d (v2u64, v2u64);

v16i8 __builtin_msa_ave_s_b (v16i8, v16i8);
v8i16 __builtin_msa_ave_s_h (v8i16, v8i16);
v4i32 __builtin_msa_ave_s_w (v4i32, v4i32);
v2i64 __builtin_msa_ave_s_d (v2i64, v2i64);

v16u8 __builtin_msa_ave_u_b (v16u8, v16u8);
v8u16 __builtin_msa_ave_u_h (v8u16, v8u16);
v4u32 __builtin_msa_ave_u_w (v4u32, v4u32);
v2u64 __builtin_msa_ave_u_d (v2u64, v2u64);

v16i8 __builtin_msa_aver_s_b (v16i8, v16i8);
v8i16 __builtin_msa_aver_s_h (v8i16, v8i16);
v4i32 __builtin_msa_aver_s_w (v4i32, v4i32);
v2i64 __builtin_msa_aver_s_d (v2i64, v2i64);

v16u8 __builtin_msa_aver_u_b (v16u8, v16u8);
v8u16 __builtin_msa_aver_u_h (v8u16, v8u16);
v4u32 __builtin_msa_aver_u_w (v4u32, v4u32);
v2u64 __builtin_msa_aver_u_d (v2u64, v2u64);

v16u8 __builtin_msa_bclr_b (v16u8, v16u8);
v8u16 __builtin_msa_bclr_h (v8u16, v8u16);
v4u32 __builtin_msa_bclr_w (v4u32, v4u32);
v2u64 __builtin_msa_bclr_d (v2u64, v2u64);

v16u8 __builtin_msa_bclri_b (v16u8, imm0_7);
v8u16 __builtin_msa_bclri_h (v8u16, imm0_15);
```

```
v4u32 __builtin_msa_bclri_w (v4u32, imm0_31);
v2u64 __builtin_msa_bclri_d (v2u64, imm0_63);

v16u8 __builtin_msa_binsl_b (v16u8, v16u8, v16u8);
v8u16 __builtin_msa_binsl_h (v8u16, v8u16, v8u16);
v4u32 __builtin_msa_binsl_w (v4u32, v4u32, v4u32);
v2u64 __builtin_msa_binsl_d (v2u64, v2u64, v2u64);

v16u8 __builtin_msa_binsli_b (v16u8, v16u8, imm0_7);
v8u16 __builtin_msa_binsli_h (v8u16, v8u16, imm0_15);
v4u32 __builtin_msa_binsli_w (v4u32, v4u32, imm0_31);
v2u64 __builtin_msa_binsli_d (v2u64, v2u64, imm0_63);

v16u8 __builtin_msa_binsr_b (v16u8, v16u8, v16u8);
v8u16 __builtin_msa_binsr_h (v8u16, v8u16, v8u16);
v4u32 __builtin_msa_binsr_w (v4u32, v4u32, v4u32);
v2u64 __builtin_msa_binsr_d (v2u64, v2u64, v2u64);

v16u8 __builtin_msa_binsri_b (v16u8, v16u8, imm0_7);
v8u16 __builtin_msa_binsri_h (v8u16, v8u16, imm0_15);
v4u32 __builtin_msa_binsri_w (v4u32, v4u32, imm0_31);
v2u64 __builtin_msa_binsri_d (v2u64, v2u64, imm0_63);

v16u8 __builtin_msa_bmnz_v (v16u8, v16u8, v16u8);

v16u8 __builtin_msa_bmnzi_b (v16u8, v16u8, imm0_255);

v16u8 __builtin_msa_bmz_v (v16u8, v16u8, v16u8);

v16u8 __builtin_msa_bmzi_b (v16u8, v16u8, imm0_255);

v16u8 __builtin_msa_bneg_b (v16u8, v16u8);
v8u16 __builtin_msa_bneg_h (v8u16, v8u16);
v4u32 __builtin_msa_bneg_w (v4u32, v4u32);
v2u64 __builtin_msa_bneg_d (v2u64, v2u64);

v16u8 __builtin_msa_bnegi_b (v16u8, imm0_7);
v8u16 __builtin_msa_bnegi_h (v8u16, imm0_15);
v4u32 __builtin_msa_bnegi_w (v4u32, imm0_31);
v2u64 __builtin_msa_bnegi_d (v2u64, imm0_63);

i32 __builtin_msa_bnz_b (v16u8);
i32 __builtin_msa_bnz_h (v8u16);
i32 __builtin_msa_bnz_w (v4u32);
i32 __builtin_msa_bnz_d (v2u64);

i32 __builtin_msa_bnz_v (v16u8);

v16u8 __builtin_msa_bsel_v (v16u8, v16u8, v16u8);

v16u8 __builtin_msa_bseli_b (v16u8, v16u8, imm0_255);

v16u8 __builtin_msa_bset_b (v16u8, v16u8);
v8u16 __builtin_msa_bset_h (v8u16, v8u16);
v4u32 __builtin_msa_bset_w (v4u32, v4u32);
v2u64 __builtin_msa_bset_d (v2u64, v2u64);

v16u8 __builtin_msa_bseti_b (v16u8, imm0_7);
```

```
v8u16 __builtin_msa_bseti_h (v8u16, imm0_15);
v4u32 __builtin_msa_bseti_w (v4u32, imm0_31);
v2u64 __builtin_msa_bseti_d (v2u64, imm0_63);

i32 __builtin_msa_bz_b (v16u8);
i32 __builtin_msa_bz_h (v8u16);
i32 __builtin_msa_bz_w (v4u32);
i32 __builtin_msa_bz_d (v2u64);

i32 __builtin_msa_bz_v (v16u8);

v16i8 __builtin_msa_ceq_b (v16i8, v16i8);
v8i16 __builtin_msa_ceq_h (v8i16, v8i16);
v4i32 __builtin_msa_ceq_w (v4i32, v4i32);
v2i64 __builtin_msa_ceq_d (v2i64, v2i64);

v16i8 __builtin_msa_ceqi_b (v16i8, imm_n16_15);
v8i16 __builtin_msa_ceqi_h (v8i16, imm_n16_15);
v4i32 __builtin_msa_ceqi_w (v4i32, imm_n16_15);
v2i64 __builtin_msa_ceqi_d (v2i64, imm_n16_15);

i32 __builtin_msa_cfcmsa (imm0_31);

v16i8 __builtin_msa_cle_s_b (v16i8, v16i8);
v8i16 __builtin_msa_cle_s_h (v8i16, v8i16);
v4i32 __builtin_msa_cle_s_w (v4i32, v4i32);
v2i64 __builtin_msa_cle_s_d (v2i64, v2i64);

v16i8 __builtin_msa_cle_u_b (v16u8, v16u8);
v8i16 __builtin_msa_cle_u_h (v8u16, v8u16);
v4i32 __builtin_msa_cle_u_w (v4u32, v4u32);
v2i64 __builtin_msa_cle_u_d (v2u64, v2u64);

v16i8 __builtin_msa_clei_s_b (v16i8, imm_n16_15);
v8i16 __builtin_msa_clei_s_h (v8i16, imm_n16_15);
v4i32 __builtin_msa_clei_s_w (v4i32, imm_n16_15);
v2i64 __builtin_msa_clei_s_d (v2i64, imm_n16_15);

v16i8 __builtin_msa_clei_u_b (v16u8, imm0_31);
v8i16 __builtin_msa_clei_u_h (v8u16, imm0_31);
v4i32 __builtin_msa_clei_u_w (v4u32, imm0_31);
v2i64 __builtin_msa_clei_u_d (v2u64, imm0_31);

v16i8 __builtin_msa_clt_s_b (v16i8, v16i8);
v8i16 __builtin_msa_clt_s_h (v8i16, v8i16);
v4i32 __builtin_msa_clt_s_w (v4i32, v4i32);
v2i64 __builtin_msa_clt_s_d (v2i64, v2i64);

v16i8 __builtin_msa_clt_u_b (v16u8, v16u8);
v8i16 __builtin_msa_clt_u_h (v8u16, v8u16);
v4i32 __builtin_msa_clt_u_w (v4u32, v4u32);
v2i64 __builtin_msa_clt_u_d (v2u64, v2u64);

v16i8 __builtin_msa_clti_s_b (v16i8, imm_n16_15);
v8i16 __builtin_msa_clti_s_h (v8i16, imm_n16_15);
v4i32 __builtin_msa_clti_s_w (v4i32, imm_n16_15);
v2i64 __builtin_msa_clti_s_d (v2i64, imm_n16_15);
```

```
v16i8 __builtin_msa_clti_u_b (v16u8, imm0_31);
v8i16 __builtin_msa_clti_u_h (v8u16, imm0_31);
v4i32 __builtin_msa_clti_u_w (v4u32, imm0_31);
v2i64 __builtin_msa_clti_u_d (v2u64, imm0_31);

i32 __builtin_msa_copy_s_b (v16i8, imm0_15);
i32 __builtin_msa_copy_s_h (v8i16, imm0_7);
i32 __builtin_msa_copy_s_w (v4i32, imm0_3);
i64 __builtin_msa_copy_s_d (v2i64, imm0_1);

u32 __builtin_msa_copy_u_b (v16i8, imm0_15);
u32 __builtin_msa_copy_u_h (v8i16, imm0_7);
u32 __builtin_msa_copy_u_w (v4i32, imm0_3);
u64 __builtin_msa_copy_u_d (v2i64, imm0_1);

void __builtin_msa_ctcmsa (imm0_31, i32);

v16i8 __builtin_msa_div_s_b (v16i8, v16i8);
v8i16 __builtin_msa_div_s_h (v8i16, v8i16);
v4i32 __builtin_msa_div_s_w (v4i32, v4i32);
v2i64 __builtin_msa_div_s_d (v2i64, v2i64);

v16u8 __builtin_msa_div_u_b (v16u8, v16u8);
v8u16 __builtin_msa_div_u_h (v8u16, v8u16);
v4u32 __builtin_msa_div_u_w (v4u32, v4u32);
v2u64 __builtin_msa_div_u_d (v2u64, v2u64);

v8i16 __builtin_msa_dotp_s_h (v16i8, v16i8);
v4i32 __builtin_msa_dotp_s_w (v8i16, v8i16);
v2i64 __builtin_msa_dotp_s_d (v4i32, v4i32);

v8u16 __builtin_msa_dotp_u_h (v16u8, v16u8);
v4u32 __builtin_msa_dotp_u_w (v8u16, v8u16);
v2u64 __builtin_msa_dotp_u_d (v4u32, v4u32);

v8i16 __builtin_msa_dpadd_s_h (v8i16, v16i8, v16i8);
v4i32 __builtin_msa_dpadd_s_w (v4i32, v8i16, v8i16);
v2i64 __builtin_msa_dpadd_s_d (v2i64, v4i32, v4i32);

v8u16 __builtin_msa_dpadd_u_h (v8u16, v16u8, v16u8);
v4u32 __builtin_msa_dpadd_u_w (v4u32, v8u16, v8u16);
v2u64 __builtin_msa_dpadd_u_d (v2u64, v4u32, v4u32);

v8i16 __builtin_msa_dpsub_s_h (v8i16, v16i8, v16i8);
v4i32 __builtin_msa_dpsub_s_w (v4i32, v8i16, v8i16);
v2i64 __builtin_msa_dpsub_s_d (v2i64, v4i32, v4i32);

v8i16 __builtin_msa_dpsub_u_h (v8i16, v16u8, v16u8);
v4i32 __builtin_msa_dpsub_u_w (v4i32, v8u16, v8u16);
v2i64 __builtin_msa_dpsub_u_d (v2i64, v4u32, v4u32);

v4f32 __builtin_msa_fadd_w (v4f32, v4f32);
v2f64 __builtin_msa_fadd_d (v2f64, v2f64);

v4i32 __builtin_msa_fcaf_w (v4f32, v4f32);
v2i64 __builtin_msa_fcaf_d (v2f64, v2f64);

v4i32 __builtin_msa_fceq_w (v4f32, v4f32);
```

```
v2i64 __builtin_msa_fceq_d (v2f64, v2f64);

v4i32 __builtin_msa_fclass_w (v4f32);
v2i64 __builtin_msa_fclass_d (v2f64);

v4i32 __builtin_msa_fcle_w (v4f32, v4f32);
v2i64 __builtin_msa_fcle_d (v2f64, v2f64);

v4i32 __builtin_msa_fclt_w (v4f32, v4f32);
v2i64 __builtin_msa_fclt_d (v2f64, v2f64);

v4i32 __builtin_msa_fcne_w (v4f32, v4f32);
v2i64 __builtin_msa_fcne_d (v2f64, v2f64);

v4i32 __builtin_msa_fcor_w (v4f32, v4f32);
v2i64 __builtin_msa_fcor_d (v2f64, v2f64);

v4i32 __builtin_msa_fcueq_w (v4f32, v4f32);
v2i64 __builtin_msa_fcueq_d (v2f64, v2f64);

v4i32 __builtin_msa_fcule_w (v4f32, v4f32);
v2i64 __builtin_msa_fcule_d (v2f64, v2f64);

v4i32 __builtin_msa_fcult_w (v4f32, v4f32);
v2i64 __builtin_msa_fcult_d (v2f64, v2f64);

v4i32 __builtin_msa_fcun_w (v4f32, v4f32);
v2i64 __builtin_msa_fcun_d (v2f64, v2f64);

v4i32 __builtin_msa_fcune_w (v4f32, v4f32);
v2i64 __builtin_msa_fcune_d (v2f64, v2f64);

v4f32 __builtin_msa_fdiv_w (v4f32, v4f32);
v2f64 __builtin_msa_fdiv_d (v2f64, v2f64);

v8i16 __builtin_msa_fexdo_h (v4f32, v4f32);
v4f32 __builtin_msa_fexdo_w (v2f64, v2f64);

v4f32 __builtin_msa_fexp2_w (v4f32, v4i32);
v2f64 __builtin_msa_fexp2_d (v2f64, v2i64);

v4f32 __builtin_msa_fexupl_w (v8i16);
v2f64 __builtin_msa_fexupl_d (v4f32);

v4f32 __builtin_msa_fexupr_w (v8i16);
v2f64 __builtin_msa_fexupr_d (v4f32);

v4f32 __builtin_msa_ffint_s_w (v4i32);
v2f64 __builtin_msa_ffint_s_d (v2i64);

v4f32 __builtin_msa_ffint_u_w (v4u32);
v2f64 __builtin_msa_ffint_u_d (v2u64);

v4f32 __builtin_msa_ffql_w (v8i16);
v2f64 __builtin_msa_ffql_d (v4i32);

v4f32 __builtin_msa_ffqr_w (v8i16);
v2f64 __builtin_msa_ffqr_d (v4i32);
```

```
v16i8 __builtin_msa_fill_b (i32);
v8i16 __builtin_msa_fill_h (i32);
v4i32 __builtin_msa_fill_w (i32);
v2i64 __builtin_msa_fill_d (i64);

v4f32 __builtin_msa_flog2_w (v4f32);
v2f64 __builtin_msa_flog2_d (v2f64);

v4f32 __builtin_msa_fmadd_w (v4f32, v4f32, v4f32);
v2f64 __builtin_msa_fmadd_d (v2f64, v2f64, v2f64);

v4f32 __builtin_msa_fmax_w (v4f32, v4f32);
v2f64 __builtin_msa_fmax_d (v2f64, v2f64);

v4f32 __builtin_msa_fmax_a_w (v4f32, v4f32);
v2f64 __builtin_msa_fmax_a_d (v2f64, v2f64);

v4f32 __builtin_msa_fmin_w (v4f32, v4f32);
v2f64 __builtin_msa_fmin_d (v2f64, v2f64);

v4f32 __builtin_msa_fmin_a_w (v4f32, v4f32);
v2f64 __builtin_msa_fmin_a_d (v2f64, v2f64);

v4f32 __builtin_msa_fmsub_w (v4f32, v4f32, v4f32);
v2f64 __builtin_msa_fmsub_d (v2f64, v2f64, v2f64);

v4f32 __builtin_msa_fmul_w (v4f32, v4f32);
v2f64 __builtin_msa_fmul_d (v2f64, v2f64);

v4f32 __builtin_msa_frint_w (v4f32);
v2f64 __builtin_msa_frint_d (v2f64);

v4f32 __builtin_msa_frcp_w (v4f32);
v2f64 __builtin_msa_frcp_d (v2f64);

v4f32 __builtin_msa_frsqrt_w (v4f32);
v2f64 __builtin_msa_frsqrt_d (v2f64);

v4i32 __builtin_msa_fsaf_w (v4f32, v4f32);
v2i64 __builtin_msa_fsaf_d (v2f64, v2f64);

v4i32 __builtin_msa_fseq_w (v4f32, v4f32);
v2i64 __builtin_msa_fseq_d (v2f64, v2f64);

v4i32 __builtin_msa_fsle_w (v4f32, v4f32);
v2i64 __builtin_msa_fsle_d (v2f64, v2f64);

v4i32 __builtin_msa_fslt_w (v4f32, v4f32);
v2i64 __builtin_msa_fslt_d (v2f64, v2f64);

v4i32 __builtin_msa_fsne_w (v4f32, v4f32);
v2i64 __builtin_msa_fsne_d (v2f64, v2f64);

v4i32 __builtin_msa_fsor_w (v4f32, v4f32);
v2i64 __builtin_msa_fsor_d (v2f64, v2f64);

v4f32 __builtin_msa_fsqrt_w (v4f32);
```

```
v2f64 __builtin_msa_fsqrt_d (v2f64);

v4f32 __builtin_msa_fsub_w (v4f32, v4f32);
v2f64 __builtin_msa_fsub_d (v2f64, v2f64);

v4i32 __builtin_msa_fsueq_w (v4f32, v4f32);
v2i64 __builtin_msa_fsueq_d (v2f64, v2f64);

v4i32 __builtin_msa_fsule_w (v4f32, v4f32);
v2i64 __builtin_msa_fsule_d (v2f64, v2f64);

v4i32 __builtin_msa_fsult_w (v4f32, v4f32);
v2i64 __builtin_msa_fsult_d (v2f64, v2f64);

v4i32 __builtin_msa_fsun_w (v4f32, v4f32);
v2i64 __builtin_msa_fsun_d (v2f64, v2f64);

v4i32 __builtin_msa_fsune_w (v4f32, v4f32);
v2i64 __builtin_msa_fsune_d (v2f64, v2f64);

v4i32 __builtin_msa_ftint_s_w (v4f32);
v2i64 __builtin_msa_ftint_s_d (v2f64);

v4u32 __builtin_msa_ftint_u_w (v4f32);
v2u64 __builtin_msa_ftint_u_d (v2f64);

v8i16 __builtin_msa_ftq_h (v4f32, v4f32);
v4i32 __builtin_msa_ftq_w (v2f64, v2f64);

v4i32 __builtin_msa_ftrunc_s_w (v4f32);
v2i64 __builtin_msa_ftrunc_s_d (v2f64);

v4u32 __builtin_msa_ftrunc_u_w (v4f32);
v2u64 __builtin_msa_ftrunc_u_d (v2f64);

v8i16 __builtin_msa_hadd_s_h (v16i8, v16i8);
v4i32 __builtin_msa_hadd_s_w (v8i16, v8i16);
v2i64 __builtin_msa_hadd_s_d (v4i32, v4i32);

v8u16 __builtin_msa_hadd_u_h (v16u8, v16u8);
v4u32 __builtin_msa_hadd_u_w (v8u16, v8u16);
v2u64 __builtin_msa_hadd_u_d (v4u32, v4u32);

v8i16 __builtin_msa_hsub_s_h (v16i8, v16i8);
v4i32 __builtin_msa_hsub_s_w (v8i16, v8i16);
v2i64 __builtin_msa_hsub_s_d (v4i32, v4i32);

v8i16 __builtin_msa_hsub_u_h (v16u8, v16u8);
v4i32 __builtin_msa_hsub_u_w (v8u16, v8u16);
v2i64 __builtin_msa_hsub_u_d (v4u32, v4u32);

v16i8 __builtin_msa_ilvev_b (v16i8, v16i8);
v8i16 __builtin_msa_ilvev_h (v8i16, v8i16);
v4i32 __builtin_msa_ilvev_w (v4i32, v4i32);
v2i64 __builtin_msa_ilvev_d (v2i64, v2i64);

v16i8 __builtin_msa_ilvl_b (v16i8, v16i8);
v8i16 __builtin_msa_ilvl_h (v8i16, v8i16);
```

```
v4i32 __builtin_msa_ilvl_w (v4i32, v4i32);
v2i64 __builtin_msa_ilvl_d (v2i64, v2i64);

v16i8 __builtin_msa_ilvod_b (v16i8, v16i8);
v8i16 __builtin_msa_ilvod_h (v8i16, v8i16);
v4i32 __builtin_msa_ilvod_w (v4i32, v4i32);
v2i64 __builtin_msa_ilvod_d (v2i64, v2i64);

v16i8 __builtin_msa_ilvr_b (v16i8, v16i8);
v8i16 __builtin_msa_ilvr_h (v8i16, v8i16);
v4i32 __builtin_msa_ilvr_w (v4i32, v4i32);
v2i64 __builtin_msa_ilvr_d (v2i64, v2i64);

v16i8 __builtin_msa_insert_b (v16i8, imm0_15, i32);
v8i16 __builtin_msa_insert_h (v8i16, imm0_7, i32);
v4i32 __builtin_msa_insert_w (v4i32, imm0_3, i32);
v2i64 __builtin_msa_insert_d (v2i64, imm0_1, i64);

v16i8 __builtin_msa_insve_b (v16i8, imm0_15, v16i8);
v8i16 __builtin_msa_insve_h (v8i16, imm0_7, v8i16);
v4i32 __builtin_msa_insve_w (v4i32, imm0_3, v4i32);
v2i64 __builtin_msa_insve_d (v2i64, imm0_1, v2i64);

v16i8 __builtin_msa_ld_b (const void *, imm_n512_511);
v8i16 __builtin_msa_ld_h (const void *, imm_n1024_1022);
v4i32 __builtin_msa_ld_w (const void *, imm_n2048_2044);
v2i64 __builtin_msa_ld_d (const void *, imm_n4096_4088);

v16i8 __builtin_msa_ldi_b (imm_n512_511);
v8i16 __builtin_msa_ldi_h (imm_n512_511);
v4i32 __builtin_msa_ldi_w (imm_n512_511);
v2i64 __builtin_msa_ldi_d (imm_n512_511);

v8i16 __builtin_msa_madd_q_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_madd_q_w (v4i32, v4i32, v4i32);

v8i16 __builtin_msa_maddr_q_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_maddr_q_w (v4i32, v4i32, v4i32);

v16i8 __builtin_msa_maddv_b (v16i8, v16i8, v16i8);
v8i16 __builtin_msa_maddv_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_maddv_w (v4i32, v4i32, v4i32);
v2i64 __builtin_msa_maddv_d (v2i64, v2i64, v2i64);

v16i8 __builtin_msa_max_a_b (v16i8, v16i8);
v8i16 __builtin_msa_max_a_h (v8i16, v8i16);
v4i32 __builtin_msa_max_a_w (v4i32, v4i32);
v2i64 __builtin_msa_max_a_d (v2i64, v2i64);

v16i8 __builtin_msa_max_s_b (v16i8, v16i8);
v8i16 __builtin_msa_max_s_h (v8i16, v8i16);
v4i32 __builtin_msa_max_s_w (v4i32, v4i32);
v2i64 __builtin_msa_max_s_d (v2i64, v2i64);

v16u8 __builtin_msa_max_u_b (v16u8, v16u8);
v8u16 __builtin_msa_max_u_h (v8u16, v8u16);
v4u32 __builtin_msa_max_u_w (v4u32, v4u32);
v2u64 __builtin_msa_max_u_d (v2u64, v2u64);
```

```
v16i8 __builtin_msa_maxi_s_b (v16i8, imm_n16_15);
v8i16 __builtin_msa_maxi_s_h (v8i16, imm_n16_15);
v4i32 __builtin_msa_maxi_s_w (v4i32, imm_n16_15);
v2i64 __builtin_msa_maxi_s_d (v2i64, imm_n16_15);

v16u8 __builtin_msa_maxi_u_b (v16u8, imm0_31);
v8u16 __builtin_msa_maxi_u_h (v8u16, imm0_31);
v4u32 __builtin_msa_maxi_u_w (v4u32, imm0_31);
v2u64 __builtin_msa_maxi_u_d (v2u64, imm0_31);

v16i8 __builtin_msa_min_a_b (v16i8, v16i8);
v8i16 __builtin_msa_min_a_h (v8i16, v8i16);
v4i32 __builtin_msa_min_a_w (v4i32, v4i32);
v2i64 __builtin_msa_min_a_d (v2i64, v2i64);

v16i8 __builtin_msa_min_s_b (v16i8, v16i8);
v8i16 __builtin_msa_min_s_h (v8i16, v8i16);
v4i32 __builtin_msa_min_s_w (v4i32, v4i32);
v2i64 __builtin_msa_min_s_d (v2i64, v2i64);

v16u8 __builtin_msa_min_u_b (v16u8, v16u8);
v8u16 __builtin_msa_min_u_h (v8u16, v8u16);
v4u32 __builtin_msa_min_u_w (v4u32, v4u32);
v2u64 __builtin_msa_min_u_d (v2u64, v2u64);

v16i8 __builtin_msa_mini_s_b (v16i8, imm_n16_15);
v8i16 __builtin_msa_mini_s_h (v8i16, imm_n16_15);
v4i32 __builtin_msa_mini_s_w (v4i32, imm_n16_15);
v2i64 __builtin_msa_mini_s_d (v2i64, imm_n16_15);

v16u8 __builtin_msa_mini_u_b (v16u8, imm0_31);
v8u16 __builtin_msa_mini_u_h (v8u16, imm0_31);
v4u32 __builtin_msa_mini_u_w (v4u32, imm0_31);
v2u64 __builtin_msa_mini_u_d (v2u64, imm0_31);

v16i8 __builtin_msa_mod_s_b (v16i8, v16i8);
v8i16 __builtin_msa_mod_s_h (v8i16, v8i16);
v4i32 __builtin_msa_mod_s_w (v4i32, v4i32);
v2i64 __builtin_msa_mod_s_d (v2i64, v2i64);

v16u8 __builtin_msa_mod_u_b (v16u8, v16u8);
v8u16 __builtin_msa_mod_u_h (v8u16, v8u16);
v4u32 __builtin_msa_mod_u_w (v4u32, v4u32);
v2u64 __builtin_msa_mod_u_d (v2u64, v2u64);

v16i8 __builtin_msa_move_v (v16i8);

v8i16 __builtin_msa_msub_q_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_msub_q_w (v4i32, v4i32, v4i32);

v8i16 __builtin_msa_msubr_q_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_msubr_q_w (v4i32, v4i32, v4i32);

v16i8 __builtin_msa_msubv_b (v16i8, v16i8, v16i8);
v8i16 __builtin_msa_msubv_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_msubv_w (v4i32, v4i32, v4i32);
v2i64 __builtin_msa_msubv_d (v2i64, v2i64, v2i64);
```

```
v8i16 __builtin_msa_mul_q_h (v8i16, v8i16);
v4i32 __builtin_msa_mul_q_w (v4i32, v4i32);

v8i16 __builtin_msa_mulr_q_h (v8i16, v8i16);
v4i32 __builtin_msa_mulr_q_w (v4i32, v4i32);

v16i8 __builtin_msa_mulv_b (v16i8, v16i8);
v8i16 __builtin_msa_mulv_h (v8i16, v8i16);
v4i32 __builtin_msa_mulv_w (v4i32, v4i32);
v2i64 __builtin_msa_mulv_d (v2i64, v2i64);

v16i8 __builtin_msa_nloc_b (v16i8);
v8i16 __builtin_msa_nloc_h (v8i16);
v4i32 __builtin_msa_nloc_w (v4i32);
v2i64 __builtin_msa_nloc_d (v2i64);

v16i8 __builtin_msa_nlzc_b (v16i8);
v8i16 __builtin_msa_nlzc_h (v8i16);
v4i32 __builtin_msa_nlzc_w (v4i32);
v2i64 __builtin_msa_nlzc_d (v2i64);

v16u8 __builtin_msa_nor_v (v16u8, v16u8);

v16u8 __builtin_msa_nori_b (v16u8, imm0_255);

v16u8 __builtin_msa_or_v (v16u8, v16u8);

v16u8 __builtin_msa_ori_b (v16u8, imm0_255);

v16i8 __builtin_msa_pckev_b (v16i8, v16i8);
v8i16 __builtin_msa_pckev_h (v8i16, v8i16);
v4i32 __builtin_msa_pckev_w (v4i32, v4i32);
v2i64 __builtin_msa_pckev_d (v2i64, v2i64);

v16i8 __builtin_msa_pckod_b (v16i8, v16i8);
v8i16 __builtin_msa_pckod_h (v8i16, v8i16);
v4i32 __builtin_msa_pckod_w (v4i32, v4i32);
v2i64 __builtin_msa_pckod_d (v2i64, v2i64);

v16i8 __builtin_msa_pcnt_b (v16i8);
v8i16 __builtin_msa_pcnt_h (v8i16);
v4i32 __builtin_msa_pcnt_w (v4i32);
v2i64 __builtin_msa_pcnt_d (v2i64);

v16i8 __builtin_msa_sat_s_b (v16i8, imm0_7);
v8i16 __builtin_msa_sat_s_h (v8i16, imm0_15);
v4i32 __builtin_msa_sat_s_w (v4i32, imm0_31);
v2i64 __builtin_msa_sat_s_d (v2i64, imm0_63);

v16u8 __builtin_msa_sat_u_b (v16u8, imm0_7);
v8u16 __builtin_msa_sat_u_h (v8u16, imm0_15);
v4u32 __builtin_msa_sat_u_w (v4u32, imm0_31);
v2u64 __builtin_msa_sat_u_d (v2u64, imm0_63);

v16i8 __builtin_msa_shf_b (v16i8, imm0_255);
v8i16 __builtin_msa_shf_h (v8i16, imm0_255);
v4i32 __builtin_msa_shf_w (v4i32, imm0_255);
```

```
v16i8 __builtin_msa_sld_b (v16i8, v16i8, i32);
v8i16 __builtin_msa_sld_h (v8i16, v8i16, i32);
v4i32 __builtin_msa_sld_w (v4i32, v4i32, i32);
v2i64 __builtin_msa_sld_d (v2i64, v2i64, i32);

v16i8 __builtin_msa_sldi_b (v16i8, v16i8, imm0_15);
v8i16 __builtin_msa_sldi_h (v8i16, v8i16, imm0_7);
v4i32 __builtin_msa_sldi_w (v4i32, v4i32, imm0_3);
v2i64 __builtin_msa_sldi_d (v2i64, v2i64, imm0_1);

v16i8 __builtin_msa_sll_b (v16i8, v16i8);
v8i16 __builtin_msa_sll_h (v8i16, v8i16);
v4i32 __builtin_msa_sll_w (v4i32, v4i32);
v2i64 __builtin_msa_sll_d (v2i64, v2i64);

v16i8 __builtin_msa_slli_b (v16i8, imm0_7);
v8i16 __builtin_msa_slli_h (v8i16, imm0_15);
v4i32 __builtin_msa_slli_w (v4i32, imm0_31);
v2i64 __builtin_msa_slli_d (v2i64, imm0_63);

v16i8 __builtin_msa_splat_b (v16i8, i32);
v8i16 __builtin_msa_splat_h (v8i16, i32);
v4i32 __builtin_msa_splat_w (v4i32, i32);
v2i64 __builtin_msa_splat_d (v2i64, i32);

v16i8 __builtin_msa_splati_b (v16i8, imm0_15);
v8i16 __builtin_msa_splati_h (v8i16, imm0_7);
v4i32 __builtin_msa_splati_w (v4i32, imm0_3);
v2i64 __builtin_msa_splati_d (v2i64, imm0_1);

v16i8 __builtin_msa_sra_b (v16i8, v16i8);
v8i16 __builtin_msa_sra_h (v8i16, v8i16);
v4i32 __builtin_msa_sra_w (v4i32, v4i32);
v2i64 __builtin_msa_sra_d (v2i64, v2i64);

v16i8 __builtin_msa_srai_b (v16i8, imm0_7);
v8i16 __builtin_msa_srai_h (v8i16, imm0_15);
v4i32 __builtin_msa_srai_w (v4i32, imm0_31);
v2i64 __builtin_msa_srai_d (v2i64, imm0_63);

v16i8 __builtin_msa_srar_b (v16i8, v16i8);
v8i16 __builtin_msa_srar_h (v8i16, v8i16);
v4i32 __builtin_msa_srar_w (v4i32, v4i32);
v2i64 __builtin_msa_srar_d (v2i64, v2i64);

v16i8 __builtin_msa_srari_b (v16i8, imm0_7);
v8i16 __builtin_msa_srari_h (v8i16, imm0_15);
v4i32 __builtin_msa_srari_w (v4i32, imm0_31);
v2i64 __builtin_msa_srari_d (v2i64, imm0_63);

v16i8 __builtin_msa_srl_b (v16i8, v16i8);
v8i16 __builtin_msa_srl_h (v8i16, v8i16);
v4i32 __builtin_msa_srl_w (v4i32, v4i32);
v2i64 __builtin_msa_srl_d (v2i64, v2i64);

v16i8 __builtin_msa_srli_b (v16i8, imm0_7);
v8i16 __builtin_msa_srli_h (v8i16, imm0_15);
```

```
v4i32 __builtin_msa_srli_w (v4i32, imm0_31);
v2i64 __builtin_msa_srli_d (v2i64, imm0_63);

v16i8 __builtin_msa_srlr_b (v16i8, v16i8);
v8i16 __builtin_msa_srlr_h (v8i16, v8i16);
v4i32 __builtin_msa_srlr_w (v4i32, v4i32);
v2i64 __builtin_msa_srlr_d (v2i64, v2i64);

v16i8 __builtin_msa_srlri_b (v16i8, imm0_7);
v8i16 __builtin_msa_srlri_h (v8i16, imm0_15);
v4i32 __builtin_msa_srlri_w (v4i32, imm0_31);
v2i64 __builtin_msa_srlri_d (v2i64, imm0_63);

void __builtin_msa_st_b (v16i8, void *, imm_n512_511);
void __builtin_msa_st_h (v8i16, void *, imm_n1024_1022);
void __builtin_msa_st_w (v4i32, void *, imm_n2048_2044);
void __builtin_msa_st_d (v2i64, void *, imm_n4096_4088);

v16i8 __builtin_msa_subs_s_b (v16i8, v16i8);
v8i16 __builtin_msa_subs_s_h (v8i16, v8i16);
v4i32 __builtin_msa_subs_s_w (v4i32, v4i32);
v2i64 __builtin_msa_subs_s_d (v2i64, v2i64);

v16u8 __builtin_msa_subs_u_b (v16u8, v16u8);
v8u16 __builtin_msa_subs_u_h (v8u16, v8u16);
v4u32 __builtin_msa_subs_u_w (v4u32, v4u32);
v2u64 __builtin_msa_subs_u_d (v2u64, v2u64);

v16u8 __builtin_msa_subsus_u_b (v16u8, v16i8);
v8u16 __builtin_msa_subsus_u_h (v8u16, v8i16);
v4u32 __builtin_msa_subsus_u_w (v4u32, v4i32);
v2u64 __builtin_msa_subsus_u_d (v2u64, v2i64);

v16i8 __builtin_msa_subsuu_s_b (v16u8, v16u8);
v8i16 __builtin_msa_subsuu_s_h (v8u16, v8u16);
v4i32 __builtin_msa_subsuu_s_w (v4u32, v4u32);
v2i64 __builtin_msa_subsuu_s_d (v2u64, v2u64);

v16i8 __builtin_msa_subv_b (v16i8, v16i8);
v8i16 __builtin_msa_subv_h (v8i16, v8i16);
v4i32 __builtin_msa_subv_w (v4i32, v4i32);
v2i64 __builtin_msa_subv_d (v2i64, v2i64);

v16i8 __builtin_msa_subvi_b (v16i8, imm0_31);
v8i16 __builtin_msa_subvi_h (v8i16, imm0_31);
v4i32 __builtin_msa_subvi_w (v4i32, imm0_31);
v2i64 __builtin_msa_subvi_d (v2i64, imm0_31);

v16i8 __builtin_msa_vshf_b (v16i8, v16i8, v16i8);
v8i16 __builtin_msa_vshf_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_vshf_w (v4i32, v4i32, v4i32);
v2i64 __builtin_msa_vshf_d (v2i64, v2i64, v2i64);

v16u8 __builtin_msa_xor_v (v16u8, v16u8);

v16u8 __builtin_msa_xori_b (v16u8, imm0_255);
```

6.60.19 Other MIPS Built-in Functions

GCC provides other MIPS-specific built-in functions:

`void __builtin_mips_cache (int op, const volatile void *addr)`
: Insert a 'cache' instruction with operands *op* and *addr*. GCC defines the preprocessor macro `___GCC_HAVE_BUILTIN_MIPS_CACHE` when this function is available.

`unsigned int __builtin_mips_get_fcsr (void)`
`void __builtin_mips_set_fcsr (unsigned int value)`
: Get and set the contents of the floating-point control and status register (FPU control register 31). These functions are only available in hard-float code but can be called in both MIPS16 and non-MIPS16 contexts.

  `__builtin_mips_set_fcsr` can be used to change any bit of the register except the condition codes, which GCC assumes are preserved.

6.60.20 MSP430 Built-in Functions

GCC provides a couple of special builtin functions to aid in the writing of interrupt handlers in C.

`__bic_SR_register_on_exit (int mask)`
: This clears the indicated bits in the saved copy of the status register currently residing on the stack. This only works inside interrupt handlers and the changes to the status register will only take affect once the handler returns.

`__bis_SR_register_on_exit (int mask)`
: This sets the indicated bits in the saved copy of the status register currently residing on the stack. This only works inside interrupt handlers and the changes to the status register will only take affect once the handler returns.

`__delay_cycles (long long cycles)`
: This inserts an instruction sequence that takes exactly *cycles* cycles (between 0 and about 17E9) to complete. The inserted sequence may use jumps, loops, or no-ops, and does not interfere with any other instructions. Note that *cycles* must be a compile-time constant integer - that is, you must pass a number, not a variable that may be optimized to a constant later. The number of cycles delayed by this builtin is exact.

6.60.21 NDS32 Built-in Functions

These built-in functions are available for the NDS32 target:

`void __builtin_nds32_isync (int *addr)` [Built-in Function]
: Insert an ISYNC instruction into the instruction stream where *addr* is an instruction address for serialization.

`void __builtin_nds32_isb (void)` [Built-in Function]
: Insert an ISB instruction into the instruction stream.

`int __builtin_nds32_mfsr (int sr)` [Built-in Function]
: Return the content of a system register which is mapped by *sr*.

`int __builtin_nds32_mfusr (int usr)` [Built-in Function]
Return the content of a user space register which is mapped by *usr*.

`void __builtin_nds32_mtsr (int value, int sr)` [Built-in Function]
Move the *value* to a system register which is mapped by *sr*.

`void __builtin_nds32_mtusr (int value, int usr)` [Built-in Function]
Move the *value* to a user space register which is mapped by *usr*.

`void __builtin_nds32_setgie_en (void)` [Built-in Function]
Enable global interrupt.

`void __builtin_nds32_setgie_dis (void)` [Built-in Function]
Disable global interrupt.

6.60.22 Basic PowerPC Built-in Functions

This section describes PowerPC built-in functions that do not require the inclusion of any special header files to declare prototypes or provide macro definitions. The sections that follow describe additional PowerPC built-in functions.

6.60.22.1 Basic PowerPC Built-in Functions Available on all Configurations

`void __builtin_cpu_init (void)` [Built-in Function]
This function is a `nop` on the PowerPC platform and is included solely to maintain API compatibility with the x86 builtins.

`int __builtin_cpu_is (const char *cpuname)` [Built-in Function]
This function returns a value of `1` if the run-time CPU is of type *cpuname* and returns `0` otherwise

The `__builtin_cpu_is` function requires GLIBC 2.23 or newer which exports the hardware capability bits. GCC defines the macro `__BUILTIN_CPU_SUPPORTS__` if the `__builtin_cpu_supports` built-in function is fully supported.

If GCC was configured to use a GLIBC before 2.23, the built-in function `__builtin_cpu_is` always returns a 0 and the compiler issues a warning.

The following CPU names can be detected:

‘`power10`’ IBM POWER10 Server CPU.

‘`power9`’ IBM POWER9 Server CPU.

‘`power8`’ IBM POWER8 Server CPU.

‘`power7`’ IBM POWER7 Server CPU.

‘`power6x`’ IBM POWER6 Server CPU (RAW mode).

‘`power6`’ IBM POWER6 Server CPU (Architected mode).

‘`power5+`’ IBM POWER5+ Server CPU.

‘`power5`’ IBM POWER5 Server CPU.

‘`ppc970`’ IBM 970 Server CPU (ie, Apple G5).

'power4' IBM POWER4 Server CPU.

'ppca2' IBM A2 64-bit Embedded CPU

'ppc476' IBM PowerPC 476FP 32-bit Embedded CPU.

'ppc464' IBM PowerPC 464 32-bit Embedded CPU.

'ppc440' PowerPC 440 32-bit Embedded CPU.

'ppc405' PowerPC 405 32-bit Embedded CPU.

'ppc-cell-be'
IBM PowerPC Cell Broadband Engine Architecture CPU.

Here is an example:

```
#ifdef __BUILTIN_CPU_SUPPORTS__
  if (__builtin_cpu_is ("power8"))
    {
       do_power8 (); // POWER8 specific implementation.
    }
  else
#endif
    {
       do_generic (); // Generic implementation.
    }
```

`int __builtin_cpu_supports (const char *feature)` [Built-in Function]
This function returns a value of `1` if the run-time CPU supports the HWCAP feature *feature* and returns `0` otherwise.

The `__builtin_cpu_supports` function requires GLIBC 2.23 or newer which exports the hardware capability bits. GCC defines the macro `__BUILTIN_CPU_SUPPORTS__` if the `__builtin_cpu_supports` built-in function is fully supported.

If GCC was configured to use a GLIBC before 2.23, the built-in function `__builtin_cpu_supports` always returns a 0 and the compiler issues a warning.

The following features can be detected:

'4xxmac' 4xx CPU has a Multiply Accumulator.

'altivec' CPU has a SIMD/Vector Unit.

'arch_2_05'
CPU supports ISA 2.05 (eg, POWER6)

'arch_2_06'
CPU supports ISA 2.06 (eg, POWER7)

'arch_2_07'
CPU supports ISA 2.07 (eg, POWER8)

'arch_3_00'
CPU supports ISA 3.0 (eg, POWER9)

'arch_3_1'
CPU supports ISA 3.1 (eg, POWER10)

'archpmu' CPU supports the set of compatible performance monitoring events.

‘`booke`’ CPU supports the Embedded ISA category.

‘`cellbe`’ CPU has a CELL broadband engine.

‘`darn`’ CPU supports the `darn` (deliver a random number) instruction.

‘`dfp`’ CPU has a decimal floating point unit.

‘`dscr`’ CPU supports the data stream control register.

‘`ebb`’ CPU supports event base branching.

‘`efpdouble`’
CPU has a SPE double precision floating point unit.

‘`efpsingle`’
CPU has a SPE single precision floating point unit.

‘`fpu`’ CPU has a floating point unit.

‘`htm`’ CPU has hardware transaction memory instructions.

‘`htm-nosc`’
Kernel aborts hardware transactions when a syscall is made.

‘`htm-no-suspend`’
CPU supports hardware transaction memory but does not support the `tsuspend.` instruction.

‘`ic_snoop`’
CPU supports icache snooping capabilities.

‘`ieee128`’ CPU supports 128-bit IEEE binary floating point instructions.

‘`isel`’ CPU supports the integer select instruction.

‘`mma`’ CPU supports the matrix-multiply assist instructions.

‘`mmu`’ CPU has a memory management unit.

‘`notb`’ CPU does not have a timebase (eg, 601 and 403gx).

‘`pa6t`’ CPU supports the PA Semi 6T CORE ISA.

‘`power4`’ CPU supports ISA 2.00 (eg, POWER4)

‘`power5`’ CPU supports ISA 2.02 (eg, POWER5)

‘`power5+`’ CPU supports ISA 2.03 (eg, POWER5+)

‘`power6x`’ CPU supports ISA 2.05 (eg, POWER6) extended opcodes mffgpr and mftgpr.

‘`ppc32`’ CPU supports 32-bit mode execution.

‘`ppc601`’ CPU supports the old POWER ISA (eg, 601)

‘`ppc64`’ CPU supports 64-bit mode execution.

‘`ppcle`’ CPU supports a little-endian mode that uses address swizzling.

‘`scv`’ Kernel supports system call vectored.

'`smt`' CPU support simultaneous multi-threading.

'`spe`' CPU has a signal processing extension unit.

'`tar`' CPU supports the target address register.

'`true_le`' CPU supports true little-endian mode.

'`ucache`' CPU has unified I/D cache.

'`vcrypto`' CPU supports the vector cryptography instructions.

'`vsx`' CPU supports the vector-scalar extension.

Here is an example:

```
#ifdef __BUILTIN_CPU_SUPPORTS__
  if (__builtin_cpu_supports ("fpu"))
    {
       asm("fadd %0,%1,%2" : "=d"(dst) : "d"(src1), "d"(src2));
    }
  else
#endif
    {
       dst = __fadd (src1, src2); // Software FP addition function.
    }
```

The following built-in functions are also available on all PowerPC processors:

```
uint64_t __builtin_ppc_get_timebase ();
unsigned long __builtin_ppc_mftb ();
double __builtin_unpack_ibm128 (__ibm128, int);
__ibm128 __builtin_pack_ibm128 (double, double);
double __builtin_mffs (void);
void __builtin_mtfsf (const int, double);
void __builtin_mtfsb0 (const int);
void __builtin_mtfsb1 (const int);
void __builtin_set_fpscr_rn (int);
```

The `__builtin_ppc_get_timebase` and `__builtin_ppc_mftb` functions generate instructions to read the Time Base Register. The `__builtin_ppc_get_timebase` function may generate multiple instructions and always returns the 64 bits of the Time Base Register. The `__builtin_ppc_mftb` function always generates one instruction and returns the Time Base Register value as an unsigned long, throwing away the most significant word on 32-bit environments. The `__builtin_mffs` return the value of the FPSCR register. Note, ISA 3.0 supports the `__builtin_mffsl()` which permits software to read the control and non-sticky status bits in the FSPCR without the higher latency associated with accessing the sticky status bits. The `__builtin_mtfsf` takes a constant 8-bit integer field mask and a double precision floating point argument and generates the `mtfsf` (extended mnemonic) instruction to write new values to selected fields of the FPSCR. The `__builtin_mtfsb0` and `__builtin_mtfsb1` take the bit to change as an argument. The valid bit range is between 0 and 31. The builtins map to the `mtfsb0` and `mtfsb1` instructions which take the argument and add 32. Hence these instructions only modify the FPSCR[32:63] bits by changing the specified bit to a zero or one respectively. The `__builtin_set_fpscr_rn` builtin allows changing both of the floating point rounding mode bits. The argument is a 2-bit value. The argument can either be a `const int` or stored in a variable. The builtin uses the ISA 3.0 instruction `mffscrn` if available,

otherwise it reads the FPSCR, masks the current rounding mode bits out and OR's in the new value.

6.60.22.2 Basic PowerPC Built-in Functions Available on ISA 2.05

The basic built-in functions described in this section are available on the PowerPC family of processors starting with ISA 2.05 or later. Unless specific options are explicitly disabled on the command line, specifying option `-mcpu=power6` has the effect of enabling the `-mpowerpc64`, `-mpowerpc-gpopt`, `-mpowerpc-gfxopt`, `-mmfcrf`, `-mpopcntb`, `-mfprnd`, `-mcmpb`, `-mhard-dfp`, and `-mrecip-precision` options. Specify the `-maltivec` option explicitly in combination with the above options if desired.

The following functions require option `-mcmpb`.

```
unsigned long long __builtin_cmpb (unsigned long long int, unsigned long long int);
unsigned int __builtin_cmpb (unsigned int, unsigned int);
```

The `__builtin_cmpb` function performs a byte-wise compare on the contents of its two arguments, returning the result of the byte-wise comparison as the returned value. For each byte comparison, the corresponding byte of the return value holds 0xff if the input bytes are equal and 0 if the input bytes are not equal. If either of the arguments to this built-in function is wider than 32 bits, the function call expands into the form that expects `unsigned long long int` arguments which is only available on 64-bit targets.

The following built-in functions are available when hardware decimal floating point (`-mhard-dfp`) is available:

```
void __builtin_set_fpscr_drn(int);
_Decimal64 __builtin_ddedpd (int, _Decimal64);
_Decimal128 __builtin_ddedpdq (int, _Decimal128);
_Decimal64 __builtin_denbcd (int, _Decimal64);
_Decimal128 __builtin_denbcdq (int, _Decimal128);
_Decimal64 __builtin_diex (long long, _Decimal64);
_Decimal128 _builtin_diexq (long long, _Decimal128);
_Decimal64 __builtin_dscli (_Decimal64, int);
_Decimal128 __builtin_dscliq (_Decimal128, int);
_Decimal64 __builtin_dscri (_Decimal64, int);
_Decimal128 __builtin_dscriq (_Decimal128, int);
long long __builtin_dxex (_Decimal64);
long long __builtin_dxexq (_Decimal128);
_Decimal128 __builtin_pack_dec128 (unsigned long long, unsigned long long);
unsigned long long __builtin_unpack_dec128 (_Decimal128, int);

The __builtin_set_fpscr_drn builtin allows changing the three decimal
floating point rounding mode bits.  The argument is a 3-bit value.  The
argument can either be a const int or the value can be stored in
a variable.
The builtin uses the ISA 3.0 instruction mffscdrn if available.
Otherwise the builtin reads the FPSCR, masks the current decimal rounding
mode bits out and OR's in the new value.
```

The following functions require `-mhard-float`, `-mpowerpc-gfxopt`, and `-mpopcntb` options.

```
double __builtin_recipdiv (double, double);
float __builtin_recipdivf (float, float);
double __builtin_rsqrt (double);
float __builtin_rsqrtf (float);
```

The `vec_rsqrt`, `__builtin_rsqrt`, and `__builtin_rsqrtf` functions generate multiple instructions to implement the reciprocal sqrt functionality using reciprocal sqrt estimate instructions.

The `__builtin_recipdiv`, and `__builtin_recipdivf` functions generate multiple instructions to implement division using the reciprocal estimate instructions.

The following functions require `-mhard-float` and `-mmultiple` options.

The `__builtin_unpack_longdouble` function takes a `long double` argument and a compile time constant of 0 or 1. If the constant is 0, the first `double` within the `long double` is returned, otherwise the second `double` is returned. The `__builtin_unpack_longdouble` function is only available if `long double` uses the IBM extended double representation.

The `__builtin_pack_longdouble` function takes two `double` arguments and returns a `long double` value that combines the two arguments. The `__builtin_pack_longdouble` function is only available if `long double` uses the IBM extended double representation.

The `__builtin_unpack_ibm128` function takes a `__ibm128` argument and a compile time constant of 0 or 1. If the constant is 0, the first `double` within the `__ibm128` is returned, otherwise the second `double` is returned.

The `__builtin_pack_ibm128` function takes two `double` arguments and returns a `__ibm128` value that combines the two arguments.

Additional built-in functions are available for the 64-bit PowerPC family of processors, for efficient use of 128-bit floating point (`__float128`) values.

6.60.22.3 Basic PowerPC Built-in Functions Available on ISA 2.06

The basic built-in functions described in this section are available on the PowerPC family of processors starting with ISA 2.05 or later. Unless specific options are explicitly disabled on the command line, specifying option `-mcpu=power7` has the effect of enabling all the same options as for `-mcpu=power6` in addition to the `-maltivec`, `-mpopcntd`, and `-mvsx` options.

The following basic built-in functions require `-mpopcntd`:

```
unsigned int __builtin_addg6s (unsigned int, unsigned int);
long long __builtin_bpermd (long long, long long);
unsigned int __builtin_cbcdtd (unsigned int);
unsigned int __builtin_cdtbcd (unsigned int);
long long __builtin_divde (long long, long long);
unsigned long long __builtin_divdeu (unsigned long long, unsigned long long);
int __builtin_divwe (int, int);
unsigned int __builtin_divweu (unsigned int, unsigned int);
vector __int128 __builtin_pack_vector_int128 (long long, long long);
void __builtin_rs6000_speculation_barrier (void);
long long __builtin_unpack_vector_int128 (vector __int128, signed char);
```

Of these, the `__builtin_divde` and `__builtin_divdeu` functions require a 64-bit environment.

The following basic built-in functions, which are also supported on x86 targets, require `-mfloat128`.

```
__float128 __builtin_fabsq (__float128);
__float128 __builtin_copysignq (__float128, __float128);
__float128 __builtin_infq (void);
__float128 __builtin_huge_valq (void);
__float128 __builtin_nanq (void);
```

```
__float128 __builtin_nansq (void);

__float128 __builtin_sqrtf128 (__float128);
__float128 __builtin_fmaf128 (__float128, __float128, __float128);
```

6.60.22.4 Basic PowerPC Built-in Functions Available on ISA 2.07

The basic built-in functions described in this section are available on the PowerPC family of processors starting with ISA 2.07 or later. Unless specific options are explicitly disabled on the command line, specifying option `-mcpu=power8` has the effect of enabling all the same options as for `-mcpu=power7` in addition to the `-mpower8-fusion`, `-mpower8-vector`, `-mcrypto`, `-mhtm`, `-mquad-memory`, and `-mquad-memory-atomic` options.

This section intentionally empty.

6.60.22.5 Basic PowerPC Built-in Functions Available on ISA 3.0

The basic built-in functions described in this section are available on the PowerPC family of processors starting with ISA 3.0 or later. Unless specific options are explicitly disabled on the command line, specifying option `-mcpu=power9` has the effect of enabling all the same options as for `-mcpu=power8` in addition to the `-misel` option.

The following built-in functions are available on Linux 64-bit systems that use the ISA 3.0 instruction set (`-mcpu=power9`):

`__float128 __builtin_addf128_round_to_odd (__float128, __float128)` [Built-in Function]

Perform a 128-bit IEEE floating point add using round to odd as the rounding mode.

`__float128 __builtin_subf128_round_to_odd (__float128, __float128)` [Built-in Function]

Perform a 128-bit IEEE floating point subtract using round to odd as the rounding mode.

`__float128 __builtin_mulf128_round_to_odd (__float128, __float128)` [Built-in Function]

Perform a 128-bit IEEE floating point multiply using round to odd as the rounding mode.

`__float128 __builtin_divf128_round_to_odd (__float128, __float128)` [Built-in Function]

Perform a 128-bit IEEE floating point divide using round to odd as the rounding mode.

`__float128 __builtin_sqrtf128_round_to_odd (__float128)` [Built-in Function]

Perform a 128-bit IEEE floating point square root using round to odd as the rounding mode.

`__float128 __builtin_fmaf128_round_to_odd (__float128, __float128, __float128)` [Built-in Function]

Perform a 128-bit IEEE floating point fused multiply and add operation using round to odd as the rounding mode.

`double __builtin_truncf128_round_to_odd (__float128)` [Built-in Function]
Convert a 128-bit IEEE floating point value to `double` using round to odd as the rounding mode.

The following additional built-in functions are also available for the PowerPC family of processors, starting with ISA 3.0 or later:

`long long __builtin_darn (void)` [Built-in Function]
`long long __builtin_darn_raw (void)` [Built-in Function]
`int __builtin_darn_32 (void)` [Built-in Function]
The `__builtin_darn` and `__builtin_darn_raw` functions require a 64-bit environment supporting ISA 3.0 or later. The `__builtin_darn` function provides a 64-bit conditioned random number. The `__builtin_darn_raw` function provides a 64-bit raw random number. The `__builtin_darn_32` function provides a 32-bit conditioned random number.

The following additional built-in functions are also available for the PowerPC family of processors, starting with ISA 3.0 or later:

```
int __builtin_byte_in_set (unsigned char u, unsigned long long set);
int __builtin_byte_in_range (unsigned char u, unsigned int range);
int __builtin_byte_in_either_range (unsigned char u, unsigned int ranges);

int __builtin_dfp_dtstsfi_lt (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_lt (unsigned int comparison, _Decimal128 value);
int __builtin_dfp_dtstsfi_lt_dd (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_lt_td (unsigned int comparison, _Decimal128 value);

int __builtin_dfp_dtstsfi_gt (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_gt (unsigned int comparison, _Decimal128 value);
int __builtin_dfp_dtstsfi_gt_dd (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_gt_td (unsigned int comparison, _Decimal128 value);

int __builtin_dfp_dtstsfi_eq (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_eq (unsigned int comparison, _Decimal128 value);
int __builtin_dfp_dtstsfi_eq_dd (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_eq_td (unsigned int comparison, _Decimal128 value);

int __builtin_dfp_dtstsfi_ov (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_ov (unsigned int comparison, _Decimal128 value);
int __builtin_dfp_dtstsfi_ov_dd (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_ov_td (unsigned int comparison, _Decimal128 value);

double __builtin_mffsl(void);
```

The `__builtin_byte_in_set` function requires a 64-bit environment supporting ISA 3.0 or later. This function returns a non-zero value if and only if its `u` argument exactly equals one of the eight bytes contained within its 64-bit `set` argument.

The `__builtin_byte_in_range` and `__builtin_byte_in_either_range` require an environment supporting ISA 3.0 or later. For these two functions, the `range` argument is encoded as 4 bytes, organized as `hi_1:lo_1:hi_2:lo_2`. The `__builtin_byte_in_range` function returns a non-zero value if and only if its `u` argument is within the range bounded between `lo_2` and `hi_2` inclusive. The `__builtin_byte_in_either_range` function returns

non-zero if and only if its `u` argument is within either the range bounded between `lo_1` and `hi_1` inclusive or the range bounded between `lo_2` and `hi_2` inclusive.

The `__builtin_dfp_dtstsfi_lt` function returns a non-zero value if and only if the number of signficant digits of its `value` argument is less than its `comparison` argument. The `__builtin_dfp_dtstsfi_lt_dd` and `__builtin_dfp_dtstsfi_lt_td` functions behave similarly, but require that the type of the `value` argument be `__Decimal64` and `__Decimal128` respectively.

The `__builtin_dfp_dtstsfi_gt` function returns a non-zero value if and only if the number of signficant digits of its `value` argument is greater than its `comparison` argument. The `__builtin_dfp_dtstsfi_gt_dd` and `__builtin_dfp_dtstsfi_gt_td` functions behave similarly, but require that the type of the `value` argument be `__Decimal64` and `__Decimal128` respectively.

The `__builtin_dfp_dtstsfi_eq` function returns a non-zero value if and only if the number of signficant digits of its `value` argument equals its `comparison` argument. The `__builtin_dfp_dtstsfi_eq_dd` and `__builtin_dfp_dtstsfi_eq_td` functions behave similarly, but require that the type of the `value` argument be `__Decimal64` and `__Decimal128` respectively.

The `__builtin_dfp_dtstsfi_ov` function returns a non-zero value if and only if its `value` argument has an undefined number of significant digits, such as when `value` is an encoding of `NaN`. The `__builtin_dfp_dtstsfi_ov_dd` and `__builtin_dfp_dtstsfi_ov_td` functions behave similarly, but require that the type of the `value` argument be `__Decimal64` and `__Decimal128` respectively.

The `__builtin_mffsl` uses the ISA 3.0 `mffsl` instruction to read the FPSCR. The instruction is a lower latency version of the `mffs` instruction. If the `mffsl` instruction is not available, then the builtin uses the older `mffs` instruction to read the FPSCR.

6.60.22.6 Basic PowerPC Built-in Functions Available on ISA 3.1

The basic built-in functions described in this section are available on the PowerPC family of processors starting with ISA 3.1. Unless specific options are explicitly disabled on the command line, specifying option `-mcpu=power10` has the effect of enabling all the same options as for `-mcpu=power9`.

The following built-in functions are available on Linux 64-bit systems that use a future architecture instruction set (`-mcpu=power10`):

`unsigned long long __builtin_cfuged (unsigned long long, unsigned long long)` [Built-in Function]

Perform a 64-bit centrifuge operation, as if implemented by the `cfuged` instruction.

`unsigned long long __builtin_cntlzdm (unsigned long long, unsigned long long)` [Built-in Function]

Perform a 64-bit count leading zeros operation under mask, as if implemented by the `cntlzdm` instruction.

`unsigned long long __builtin_cnttzdm (unsigned long long, unsigned long long)` [Built-in Function]

Perform a 64-bit count trailing zeros operation under mask, as if implemented by the `cnttzdm` instruction.

`unsigned long long __builtin_pdepd (unsigned long long, unsigned long long)` [Built-in Function]

Perform a 64-bit parallel bits deposit operation, as if implemented by the `pdepd` instruction.

`unsigned long long __builtin_pextd (unsigned long long, unsigned long long)` [Built-in Function]

Perform a 64-bit parallel bits extract operation, as if implemented by the `pextd` instruction.

`vector signed __int128 vsx_xl_sext (signed long long, signed char *)` [Built-in Function]
`vector signed __int128 vsx_xl_sext (signed long long, signed short *)` [Built-in Function]
`vector signed __int128 vsx_xl_sext (signed long long, signed int *)` [Built-in Function]
`vector signed __int128 vsx_xl_sext (signed long long, signed long long *)` [Built-in Function]
`vector unsigned __int128 vsx_xl_zext (signed long long, unsigned char *)` [Built-in Function]
`vector unsigned __int128 vsx_xl_zext (signed long long, unsigned short *)` [Built-in Function]
`vector unsigned __int128 vsx_xl_zext (signed long long, unsigned int *)` [Built-in Function]
`vector unsigned __int128 vsx_xl_zext (signed long long, unsigned long long *)` [Built-in Function]

Load (and sign extend) to an __int128 vector, as if implemented by the ISA 3.1 `lxvrbx`, `lxvrhx`, `lxvrwx`, and `lxvrdx` instructions.

`void vec_xst_trunc (vector signed __int128, signed long long, signed char *)` [Built-in Function]
`void vec_xst_trunc (vector signed __int128, signed long long, signed short *)` [Built-in Function]
`void vec_xst_trunc (vector signed __int128, signed long long, signed int *)` [Built-in Function]
`void vec_xst_trunc (vector signed __int128, signed long long, signed long long *)` [Built-in Function]
`void vec_xst_trunc (vector unsigned __int128, signed long long, unsigned char *)` [Built-in Function]
`void vec_xst_trunc (vector unsigned __int128, signed long long, unsigned short *)` [Built-in Function]
`void vec_xst_trunc (vector unsigned __int128, signed long long, unsigned int *)` [Built-in Function]
`void vec_xst_trunc (vector unsigned __int128, signed long long, unsigned long long *)` [Built-in Function]

Truncate and store the rightmost element of a vector, as if implemented by the ISA 3.1 `stxvrbx`, `stxvrhx`, `stxvrwx`, and `stxvrdx` instructions.

6.60.23 PowerPC AltiVec/VSX Built-in Functions

GCC provides an interface for the PowerPC family of processors to access the AltiVec operations described in Motorola's AltiVec Programming Interface Manual. The interface is made available by including `<altivec.h>` and using `-maltivec` and `-mabi=altivec`. The interface supports the following vector types.

```
vector unsigned char
vector signed char
vector bool char

vector unsigned short
vector signed short
vector bool short
vector pixel

vector unsigned int
vector signed int
vector bool int
vector float
```

GCC's implementation of the high-level language interface available from C and C++ code differs from Motorola's documentation in several ways.

- A vector constant is a list of constant expressions within curly braces.
- A vector initializer requires no cast if the vector constant is of the same type as the variable it is initializing.
- If `signed` or `unsigned` is omitted, the signedness of the vector type is the default signedness of the base type. The default varies depending on the operating system, so a portable program should always specify the signedness.
- Compiling with `-maltivec` adds keywords `__vector`, `vector`, `__pixel`, `pixel`, `__bool` and `bool`. When compiling ISO C, the context-sensitive substitution of the keywords `vector`, `pixel` and `bool` is disabled. To use them, you must include `<altivec.h>` instead.
- GCC allows using a `typedef` name as the type specifier for a vector type, but only under the following circumstances:
  - When using `__vector` instead of `vector`; for example,

    ```
    typedef signed short int16;
    __vector int16 data;
    ```

  - When using `vector` in keyword-and-predefine mode; for example,

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

    Note that keyword-and-predefine mode is enabled by disabling GNU extensions (e.g., by using `-std=c11`) and including `<altivec.h>`.
- For C, overloaded functions are implemented with macros so the following does not work:

  ```
  vec_add ((vector signed int){1, 2, 3, 4}, foo);
  ```

  Since `vec_add` is a macro, the vector constant in the example is treated as four separate arguments. Wrap the entire argument in parentheses for this to work.

Note: Only the `<altivec.h>` interface is supported. Internally, GCC uses built-in functions to achieve the functionality in the aforementioned header file, but they are not supported and are subject to change without notice.

GCC complies with the Power Vector Intrinsic Programming Reference (PVIPR), which may be found at `https://openpowerfoundation.org/?resource_lib=power-vector-intrinsic-programming-reference`. Chapter 4 of this document fully documents the vector API interfaces that must be provided by compliant compilers. Programmers should preferentially use the interfaces described therein. However, historically GCC has provided additional interfaces for access to vector instructions. These are briefly described below. Where the PVIPR provides a portable interface, other functions in GCC that provide the same capabilities should be considered deprecated.

The PVIPR documents the following overloaded functions:

```
vec_abs                    vec_absd                   vec_abss
vec_add                    vec_addc                   vec_adde
vec_addec                  vec_adds                   vec_all_eq
vec_all_ge                 vec_all_gt                 vec_all_in
vec_all_le                 vec_all_lt                 vec_all_nan
vec_all_ne                 vec_all_nge                vec_all_ngt
vec_all_nle                vec_all_nlt                vec_all_numeric
vec_and                    vec_andc                   vec_any_eq
vec_any_ge                 vec_any_gt                 vec_any_le
vec_any_lt                 vec_any_nan                vec_any_ne
vec_any_nge                vec_any_ngt                vec_any_nle
vec_any_nlt                vec_any_numeric            vec_any_out
vec_avg                    vec_bperm                  vec_ceil
vec_cipher_be              vec_cipherlast_be          vec_cmpb
vec_cmpeq                  vec_cmpge                  vec_cmpgt
vec_cmple                  vec_cmplt                  vec_cmpne
vec_cmpnez                 vec_cntlz                  vec_cntlz_lsbb
vec_cnttz                  vec_cnttz_lsbb             vec_cpsgn
vec_ctf                    vec_cts                    vec_ctu
vec_div                    vec_double                 vec_doublee
vec_doubleh                vec_doublel                vec_doubleo
vec_eqv                    vec_expte                  vec_extract
vec_extract_exp            vec_extract_fp32_from_     vec_extract_fp32_from_
                           shorth                     shortl
vec_extract_sig            vec_extract_4b             vec_first_match_index
vec_first_match_or_eos_    vec_first_mismatch_        vec_first_mismatch_or_
index                      index                      eos_index
vec_float                  vec_float2                 vec_floate
vec_floato                 vec_floor                  vec_gb
vec_insert                 vec_insert_exp             vec_insert4b
vec_ld                     vec_lde                    vec_ldl
vec_loge                   vec_madd                   vec_madds
```

```
vec_max                    vec_mergee              vec_mergeh
vec_mergel                 vec_mergeo              vec_mfvscr
vec_min                    vec_mradds              vec_msub
vec_msum                   vec_msums               vec_mtvscr
vec_mul                    vec_mule                vec_mulo
vec_nabs                   vec_nand                vec_ncipher_be
vec_ncipherlast_be         vec_nearbyint           vec_neg
vec_nmadd                  vec_nmsub               vec_nor
vec_or                     vec_orc                 vec_pack
vec_pack_to_short_fp32     vec_packpx              vec_packs
vec_packsu                 vec_parity_lsbb         vec_perm
vec_permxor                vec_pmsum_be            vec_popcnt
vec_re                     vec_recipdiv            vec_revb
vec_reve                   vec_rint                vec_rl
vec_rlmi                   vec_rlnm                vec_round
vec_rsqrt                  vec_rsqrte              vec_sbox_be
vec_sel                    vec_shasigma_be         vec_signed
vec_signed2                vec_signede             vec_signedo
vec_sl                     vec_sld                 vec_sldw
vec_sll                    vec_slo                 vec_slv
vec_splat                  vec_splat_s8            vec_splat_s16
vec_splat_s32              vec_splat_u8            vec_splat_u16
vec_splat_u32              vec_splats              vec_sqrt
vec_sr                     vec_sra                 vec_srl
vec_sro                    vec_srv                 vec_st
vec_ste                    vec_stl                 vec_sub
vec_subc                   vec_sube                vec_subec
vec_subs                   vec_sum2s               vec_sum4s
vec_sums                   vec_test_data_class     vec_trunc
vec_unpackh                vec_unpackl             vec_unsigned
vec_unsigned2              vec_unsignede           vec_unsignedo
vec_xl                     vec_xl_be               vec_xl_len
vec_xl_len_r               vec_xor                 vec_xst
vec_xst_be                 vec_xst_len             vec_xst_len_r
```

6.60.23.1 PowerPC AltiVec Built-in Functions on ISA 2.05

The following interfaces are supported for the generic and specific AltiVec operations and the AltiVec predicates. In cases where there is a direct mapping between generic and specific operations, only the generic names are shown here, although the specific operations can also be used.

Arguments that are documented as `const int` require literal integral values within the range required for that operation.

Only functions excluded from the PVIPR are listed here.

```
void vec_dss (const int);

void vec_dssall (void);
```

```
void vec_dst (const vector unsigned char *, int, const int);
void vec_dst (const vector signed char *, int, const int);
void vec_dst (const vector bool char *, int, const int);
void vec_dst (const vector unsigned short *, int, const int);
void vec_dst (const vector signed short *, int, const int);
void vec_dst (const vector bool short *, int, const int);
void vec_dst (const vector pixel *, int, const int);
void vec_dst (const vector unsigned int *, int, const int);
void vec_dst (const vector signed int *, int, const int);
void vec_dst (const vector bool int *, int, const int);
void vec_dst (const vector float *, int, const int);
void vec_dst (const unsigned char *, int, const int);
void vec_dst (const signed char *, int, const int);
void vec_dst (const unsigned short *, int, const int);
void vec_dst (const short *, int, const int);
void vec_dst (const unsigned int *, int, const int);
void vec_dst (const int *, int, const int);
void vec_dst (const float *, int, const int);

void vec_dstst (const vector unsigned char *, int, const int);
void vec_dstst (const vector signed char *, int, const int);
void vec_dstst (const vector bool char *, int, const int);
void vec_dstst (const vector unsigned short *, int, const int);
void vec_dstst (const vector signed short *, int, const int);
void vec_dstst (const vector bool short *, int, const int);
void vec_dstst (const vector pixel *, int, const int);
void vec_dstst (const vector unsigned int *, int, const int);
void vec_dstst (const vector signed int *, int, const int);
void vec_dstst (const vector bool int *, int, const int);
void vec_dstst (const vector float *, int, const int);
void vec_dstst (const unsigned char *, int, const int);
void vec_dstst (const signed char *, int, const int);
void vec_dstst (const unsigned short *, int, const int);
void vec_dstst (const short *, int, const int);
void vec_dstst (const unsigned int *, int, const int);
void vec_dstst (const int *, int, const int);
void vec_dstst (const unsigned long *, int, const int);
void vec_dstst (const long *, int, const int);
void vec_dstst (const float *, int, const int);

void vec_dststt (const vector unsigned char *, int, const int);
void vec_dststt (const vector signed char *, int, const int);
void vec_dststt (const vector bool char *, int, const int);
void vec_dststt (const vector unsigned short *, int, const int);
void vec_dststt (const vector signed short *, int, const int);
void vec_dststt (const vector bool short *, int, const int);
void vec_dststt (const vector pixel *, int, const int);
void vec_dststt (const vector unsigned int *, int, const int);
void vec_dststt (const vector signed int *, int, const int);
void vec_dststt (const vector bool int *, int, const int);
void vec_dststt (const vector float *, int, const int);
void vec_dststt (const unsigned char *, int, const int);
void vec_dststt (const signed char *, int, const int);
void vec_dststt (const unsigned short *, int, const int);
void vec_dststt (const short *, int, const int);
void vec_dststt (const unsigned int *, int, const int);
void vec_dststt (const int *, int, const int);
```

```
void vec_dststt (const float *, int, const int);

void vec_dstt (const vector unsigned char *, int, const int);
void vec_dstt (const vector signed char *, int, const int);
void vec_dstt (const vector bool char *, int, const int);
void vec_dstt (const vector unsigned short *, int, const int);
void vec_dstt (const vector signed short *, int, const int);
void vec_dstt (const vector bool short *, int, const int);
void vec_dstt (const vector pixel *, int, const int);
void vec_dstt (const vector unsigned int *, int, const int);
void vec_dstt (const vector signed int *, int, const int);
void vec_dstt (const vector bool int *, int, const int);
void vec_dstt (const vector float *, int, const int);
void vec_dstt (const unsigned char *, int, const int);
void vec_dstt (const signed char *, int, const int);
void vec_dstt (const unsigned short *, int, const int);
void vec_dstt (const short *, int, const int);
void vec_dstt (const unsigned int *, int, const int);
void vec_dstt (const int *, int, const int);
void vec_dstt (const float *, int, const int);

vector signed char vec_lvebx (int, char *);
vector unsigned char vec_lvebx (int, unsigned char *);

vector signed short vec_lvehx (int, short *);
vector unsigned short vec_lvehx (int, unsigned short *);

vector float vec_lvewx (int, float *);
vector signed int vec_lvewx (int, int *);
vector unsigned int vec_lvewx (int, unsigned int *);

vector unsigned char vec_lvsl (int, const unsigned char *);
vector unsigned char vec_lvsl (int, const signed char *);
vector unsigned char vec_lvsl (int, const unsigned short *);
vector unsigned char vec_lvsl (int, const short *);
vector unsigned char vec_lvsl (int, const unsigned int *);
vector unsigned char vec_lvsl (int, const int *);
vector unsigned char vec_lvsl (int, const float *);

vector unsigned char vec_lvsr (int, const unsigned char *);
vector unsigned char vec_lvsr (int, const signed char *);
vector unsigned char vec_lvsr (int, const unsigned short *);
vector unsigned char vec_lvsr (int, const short *);
vector unsigned char vec_lvsr (int, const unsigned int *);
vector unsigned char vec_lvsr (int, const int *);
vector unsigned char vec_lvsr (int, const float *);

void vec_stvebx (vector signed char, int, signed char *);
void vec_stvebx (vector unsigned char, int, unsigned char *);
void vec_stvebx (vector bool char, int, signed char *);
void vec_stvebx (vector bool char, int, unsigned char *);

void vec_stvehx (vector signed short, int, short *);
void vec_stvehx (vector unsigned short, int, unsigned short *);
void vec_stvehx (vector bool short, int, short *);
void vec_stvehx (vector bool short, int, unsigned short *);

void vec_stvewx (vector float, int, float *);
```

```
void vec_stvewx (vector signed int, int, int *);
void vec_stvewx (vector unsigned int, int, unsigned int *);
void vec_stvewx (vector bool int, int, int *);
void vec_stvewx (vector bool int, int, unsigned int *);

vector float vec_vaddfp (vector float, vector float);

vector signed char vec_vaddsbs (vector bool char, vector signed char);
vector signed char vec_vaddsbs (vector signed char, vector bool char);
vector signed char vec_vaddsbs (vector signed char, vector signed char);

vector signed short vec_vaddshs (vector bool short, vector signed short);
vector signed short vec_vaddshs (vector signed short, vector bool short);
vector signed short vec_vaddshs (vector signed short, vector signed short);

vector signed int vec_vaddsws (vector bool int, vector signed int);
vector signed int vec_vaddsws (vector signed int, vector bool int);
vector signed int vec_vaddsws (vector signed int, vector signed int);

vector signed char vec_vaddubm (vector bool char, vector signed char);
vector signed char vec_vaddubm (vector signed char, vector bool char);
vector signed char vec_vaddubm (vector signed char, vector signed char);
vector unsigned char vec_vaddubm (vector bool char, vector unsigned char);
vector unsigned char vec_vaddubm (vector unsigned char, vector bool char);
vector unsigned char vec_vaddubm (vector unsigned char, vector unsigned char);

vector unsigned char vec_vaddubs (vector bool char, vector unsigned char);
vector unsigned char vec_vaddubs (vector unsigned char, vector bool char);
vector unsigned char vec_vaddubs (vector unsigned char, vector unsigned char);

vector signed short vec_vadduhm (vector bool short, vector signed short);
vector signed short vec_vadduhm (vector signed short, vector bool short);
vector signed short vec_vadduhm (vector signed short, vector signed short);
vector unsigned short vec_vadduhm (vector bool short, vector unsigned short);
vector unsigned short vec_vadduhm (vector unsigned short, vector bool short);
vector unsigned short vec_vadduhm (vector unsigned short, vector unsigned short);

vector unsigned short vec_vadduhs (vector bool short, vector unsigned short);
vector unsigned short vec_vadduhs (vector unsigned short, vector bool short);
vector unsigned short vec_vadduhs (vector unsigned short, vector unsigned short);

vector signed int vec_vadduwm (vector bool int, vector signed int);
vector signed int vec_vadduwm (vector signed int, vector bool int);
vector signed int vec_vadduwm (vector signed int, vector signed int);
vector unsigned int vec_vadduwm (vector bool int, vector unsigned int);
vector unsigned int vec_vadduwm (vector unsigned int, vector bool int);
vector unsigned int vec_vadduwm (vector unsigned int, vector unsigned int);

vector unsigned int vec_vadduws (vector bool int, vector unsigned int);
vector unsigned int vec_vadduws (vector unsigned int, vector bool int);
vector unsigned int vec_vadduws (vector unsigned int, vector unsigned int);

vector signed char vec_vavgsb (vector signed char, vector signed char);

vector signed short vec_vavgsh (vector signed short, vector signed short);

vector signed int vec_vavgsw (vector signed int, vector signed int);
```

```
vector unsigned char vec_vavgub (vector unsigned char, vector unsigned char);

vector unsigned short vec_vavguh (vector unsigned short, vector unsigned short);

vector unsigned int vec_vavguw (vector unsigned int, vector unsigned int);

vector float vec_vcfsx (vector signed int, const int);

vector float vec_vcfux (vector unsigned int, const int);

vector bool int vec_vcmpeqfp (vector float, vector float);

vector bool char vec_vcmpequb (vector signed char, vector signed char);
vector bool char vec_vcmpequb (vector unsigned char, vector unsigned char);

vector bool short vec_vcmpequh (vector signed short, vector signed short);
vector bool short vec_vcmpequh (vector unsigned short, vector unsigned short);

vector bool int vec_vcmpequw (vector signed int, vector signed int);
vector bool int vec_vcmpequw (vector unsigned int, vector unsigned int);

vector bool int vec_vcmpgtfp (vector float, vector float);

vector bool char vec_vcmpgtsb (vector signed char, vector signed char);

vector bool short vec_vcmpgtsh (vector signed short, vector signed short);

vector bool int vec_vcmpgtsw (vector signed int, vector signed int);

vector bool char vec_vcmpgtub (vector unsigned char, vector unsigned char);

vector bool short vec_vcmpgtuh (vector unsigned short, vector unsigned short);

vector bool int vec_vcmpgtuw (vector unsigned int, vector unsigned int);

vector float vec_vmaxfp (vector float, vector float);

vector signed char vec_vmaxsb (vector bool char, vector signed char);
vector signed char vec_vmaxsb (vector signed char, vector bool char);
vector signed char vec_vmaxsb (vector signed char, vector signed char);

vector signed short vec_vmaxsh (vector bool short, vector signed short);
vector signed short vec_vmaxsh (vector signed short, vector bool short);
vector signed short vec_vmaxsh (vector signed short, vector signed short);

vector signed int vec_vmaxsw (vector bool int, vector signed int);
vector signed int vec_vmaxsw (vector signed int, vector bool int);
vector signed int vec_vmaxsw (vector signed int, vector signed int);

vector unsigned char vec_vmaxub (vector bool char, vector unsigned char);
vector unsigned char vec_vmaxub (vector unsigned char, vector bool char);
vector unsigned char vec_vmaxub (vector unsigned char, vector unsigned char);

vector unsigned short vec_vmaxuh (vector bool short, vector unsigned short);
vector unsigned short vec_vmaxuh (vector unsigned short, vector bool short);
vector unsigned short vec_vmaxuh (vector unsigned short, vector unsigned short);

vector unsigned int vec_vmaxuw (vector bool int, vector unsigned int);
```

```
vector unsigned int vec_vmaxuw (vector unsigned int, vector bool int);
vector unsigned int vec_vmaxuw (vector unsigned int, vector unsigned int);

vector float vec_vminfp (vector float, vector float);

vector signed char vec_vminsb (vector bool char, vector signed char);
vector signed char vec_vminsb (vector signed char, vector bool char);
vector signed char vec_vminsb (vector signed char, vector signed char);

vector signed short vec_vminsh (vector bool short, vector signed short);
vector signed short vec_vminsh (vector signed short, vector bool short);
vector signed short vec_vminsh (vector signed short, vector signed short);

vector signed int vec_vminsw (vector bool int, vector signed int);
vector signed int vec_vminsw (vector signed int, vector bool int);
vector signed int vec_vminsw (vector signed int, vector signed int);

vector unsigned char vec_vminub (vector bool char, vector unsigned char);
vector unsigned char vec_vminub (vector unsigned char, vector bool char);
vector unsigned char vec_vminub (vector unsigned char, vector unsigned char);

vector unsigned short vec_vminuh (vector bool short, vector unsigned short);
vector unsigned short vec_vminuh (vector unsigned short, vector bool short);
vector unsigned short vec_vminuh (vector unsigned short, vector unsigned short);

vector unsigned int vec_vminuw (vector bool int, vector unsigned int);
vector unsigned int vec_vminuw (vector unsigned int, vector bool int);
vector unsigned int vec_vminuw (vector unsigned int, vector unsigned int);

vector bool char vec_vmrghb (vector bool char, vector bool char);
vector signed char vec_vmrghb (vector signed char, vector signed char);
vector unsigned char vec_vmrghb (vector unsigned char, vector unsigned char);

vector bool short vec_vmrghh (vector bool short, vector bool short);
vector signed short vec_vmrghh (vector signed short, vector signed short);
vector unsigned short vec_vmrghh (vector unsigned short, vector unsigned short);
vector pixel vec_vmrghh (vector pixel, vector pixel);

vector float vec_vmrghw (vector float, vector float);
vector bool int vec_vmrghw (vector bool int, vector bool int);
vector signed int vec_vmrghw (vector signed int, vector signed int);
vector unsigned int vec_vmrghw (vector unsigned int, vector unsigned int);

vector bool char vec_vmrglb (vector bool char, vector bool char);
vector signed char vec_vmrglb (vector signed char, vector signed char);
vector unsigned char vec_vmrglb (vector unsigned char, vector unsigned char);

vector bool short vec_vmrglh (vector bool short, vector bool short);
vector signed short vec_vmrglh (vector signed short, vector signed short);
vector unsigned short vec_vmrglh (vector unsigned short, vector unsigned short);
vector pixel vec_vmrglh (vector pixel, vector pixel);

vector float vec_vmrglw (vector float, vector float);
vector signed int vec_vmrglw (vector signed int, vector signed int);
vector unsigned int vec_vmrglw (vector unsigned int, vector unsigned int);
vector bool int vec_vmrglw (vector bool int, vector bool int);

vector signed int vec_vmsummbm (vector signed char, vector unsigned char,
```

```
                                vector signed int);

vector signed int vec_vmsumshm (vector signed short, vector signed short,
                                vector signed int);

vector signed int vec_vmsumshs (vector signed short, vector signed short,
                                vector signed int);

vector unsigned int vec_vmsumubm (vector unsigned char, vector unsigned char,
                                  vector unsigned int);

vector unsigned int vec_vmsumuhm (vector unsigned short, vector unsigned short,
                                  vector unsigned int);

vector unsigned int vec_vmsumuhs (vector unsigned short, vector unsigned short,
                                  vector unsigned int);

vector signed short vec_vmulesb (vector signed char, vector signed char);

vector signed int vec_vmulesh (vector signed short, vector signed short);

vector unsigned short vec_vmuleub (vector unsigned char, vector unsigned char);

vector unsigned int vec_vmuleuh (vector unsigned short, vector unsigned short);

vector signed short vec_vmulosb (vector signed char, vector signed char);

vector signed int vec_vmulosh (vector signed short, vector signed short);

vector unsigned short vec_vmuloub (vector unsigned char, vector unsigned char);

vector unsigned int vec_vmulouh (vector unsigned short, vector unsigned short);

vector signed char vec_vpkshss (vector signed short, vector signed short);

vector unsigned char vec_vpkshus (vector signed short, vector signed short);

vector signed short vec_vpkswss (vector signed int, vector signed int);

vector unsigned short vec_vpkswus (vector signed int, vector signed int);

vector bool char vec_vpkuhum (vector bool short, vector bool short);
vector signed char vec_vpkuhum (vector signed short, vector signed short);
vector unsigned char vec_vpkuhum (vector unsigned short, vector unsigned short);

vector unsigned char vec_vpkuhus (vector unsigned short, vector unsigned short);

vector bool short vec_vpkuwum (vector bool int, vector bool int);
vector signed short vec_vpkuwum (vector signed int, vector signed int);
vector unsigned short vec_vpkuwum (vector unsigned int, vector unsigned int);

vector unsigned short vec_vpkuwus (vector unsigned int, vector unsigned int);

vector signed char vec_vrlb (vector signed char, vector unsigned char);
vector unsigned char vec_vrlb (vector unsigned char, vector unsigned char);

vector signed short vec_vrlh (vector signed short, vector unsigned short);
vector unsigned short vec_vrlh (vector unsigned short, vector unsigned short);
```

```
vector signed int vec_vrlw (vector signed int, vector unsigned int);
vector unsigned int vec_vrlw (vector unsigned int, vector unsigned int);

vector signed char vec_vslb (vector signed char, vector unsigned char);
vector unsigned char vec_vslb (vector unsigned char, vector unsigned char);

vector signed short vec_vslh (vector signed short, vector unsigned short);
vector unsigned short vec_vslh (vector unsigned short, vector unsigned short);

vector signed int vec_vslw (vector signed int, vector unsigned int);
vector unsigned int vec_vslw (vector unsigned int, vector unsigned int);

vector signed char vec_vspltb (vector signed char, const int);
vector unsigned char vec_vspltb (vector unsigned char, const int);
vector bool char vec_vspltb (vector bool char, const int);

vector bool short vec_vsplth (vector bool short, const int);
vector signed short vec_vsplth (vector signed short, const int);
vector unsigned short vec_vsplth (vector unsigned short, const int);
vector pixel vec_vsplth (vector pixel, const int);

vector float vec_vspltw (vector float, const int);
vector signed int vec_vspltw (vector signed int, const int);
vector unsigned int vec_vspltw (vector unsigned int, const int);
vector bool int vec_vspltw (vector bool int, const int);

vector signed char vec_vsrab (vector signed char, vector unsigned char);
vector unsigned char vec_vsrab (vector unsigned char, vector unsigned char);

vector signed short vec_vsrah (vector signed short, vector unsigned short);
vector unsigned short vec_vsrah (vector unsigned short, vector unsigned short);

vector signed int vec_vsraw (vector signed int, vector unsigned int);
vector unsigned int vec_vsraw (vector unsigned int, vector unsigned int);

vector signed char vec_vsrb (vector signed char, vector unsigned char);
vector unsigned char vec_vsrb (vector unsigned char, vector unsigned char);

vector signed short vec_vsrh (vector signed short, vector unsigned short);
vector unsigned short vec_vsrh (vector unsigned short, vector unsigned short);

vector signed int vec_vsrw (vector signed int, vector unsigned int);
vector unsigned int vec_vsrw (vector unsigned int, vector unsigned int);

vector float vec_vsubfp (vector float, vector float);

vector signed char vec_vsubsbs (vector bool char, vector signed char);
vector signed char vec_vsubsbs (vector signed char, vector bool char);
vector signed char vec_vsubsbs (vector signed char, vector signed char);

vector signed short vec_vsubshs (vector bool short, vector signed short);
vector signed short vec_vsubshs (vector signed short, vector bool short);
vector signed short vec_vsubshs (vector signed short, vector signed short);

vector signed int vec_vsubsws (vector bool int, vector signed int);
vector signed int vec_vsubsws (vector signed int, vector bool int);
vector signed int vec_vsubsws (vector signed int, vector signed int);
```

```
vector signed char vec_vsububm (vector bool char, vector signed char);
vector signed char vec_vsububm (vector signed char, vector bool char);
vector signed char vec_vsububm (vector signed char, vector signed char);
vector unsigned char vec_vsububm (vector bool char, vector unsigned char);
vector unsigned char vec_vsububm (vector unsigned char, vector bool char);
vector unsigned char vec_vsububm (vector unsigned char, vector unsigned char);

vector unsigned char vec_vsububs (vector bool char, vector unsigned char);
vector unsigned char vec_vsububs (vector unsigned char, vector bool char);
vector unsigned char vec_vsububs (vector unsigned char, vector unsigned char);

vector signed short vec_vsubuhm (vector bool short, vector signed short);
vector signed short vec_vsubuhm (vector signed short, vector bool short);
vector signed short vec_vsubuhm (vector signed short, vector signed short);
vector unsigned short vec_vsubuhm (vector bool short, vector unsigned short);
vector unsigned short vec_vsubuhm (vector unsigned short, vector bool short);
vector unsigned short vec_vsubuhm (vector unsigned short, vector unsigned short);

vector unsigned short vec_vsubuhs (vector bool short, vector unsigned short);
vector unsigned short vec_vsubuhs (vector unsigned short, vector bool short);
vector unsigned short vec_vsubuhs (vector unsigned short, vector unsigned short);

vector signed int vec_vsubuwm (vector bool int, vector signed int);
vector signed int vec_vsubuwm (vector signed int, vector bool int);
vector signed int vec_vsubuwm (vector signed int, vector signed int);
vector unsigned int vec_vsubuwm (vector bool int, vector unsigned int);
vector unsigned int vec_vsubuwm (vector unsigned int, vector bool int);
vector unsigned int vec_vsubuwm (vector unsigned int, vector unsigned int);

vector unsigned int vec_vsubuws (vector bool int, vector unsigned int);
vector unsigned int vec_vsubuws (vector unsigned int, vector bool int);
vector unsigned int vec_vsubuws (vector unsigned int, vector unsigned int);

vector signed int vec_vsum4sbs (vector signed char, vector signed int);

vector signed int vec_vsum4shs (vector signed short, vector signed int);

vector unsigned int vec_vsum4ubs (vector unsigned char, vector unsigned int);

vector unsigned int vec_vupkhpx (vector pixel);

vector bool short vec_vupkhsb (vector bool char);
vector signed short vec_vupkhsb (vector signed char);

vector bool int vec_vupkhsh (vector bool short);
vector signed int vec_vupkhsh (vector signed short);

vector unsigned int vec_vupklpx (vector pixel);

vector bool short vec_vupklsb (vector bool char);
vector signed short vec_vupklsb (vector signed char);

vector bool int vec_vupklsh (vector bool short);
vector signed int vec_vupklsh (vector signed short);
```

6.60.23.2 PowerPC AltiVec Built-in Functions Available on ISA 2.06

The AltiVec built-in functions described in this section are available on the PowerPC family of processors starting with ISA 2.06 or later. These are normally enabled by adding `-mvsx` to the command line.

When `-mvsx` is used, the following additional vector types are implemented.

```
vector unsigned __int128
vector signed __int128
vector unsigned long long int
vector signed long long int
vector double
```

The long long types are only implemented for 64-bit code generation.

Only functions excluded from the PVIPR are listed here.

```
void vec_dst (const unsigned long *, int, const int);
void vec_dst (const long *, int, const int);

void vec_dststt (const unsigned long *, int, const int);
void vec_dststt (const long *, int, const int);

void vec_dstt (const unsigned long *, int, const int);
void vec_dstt (const long *, int, const int);

vector unsigned char vec_lvsl (int, const unsigned long *);
vector unsigned char vec_lvsl (int, const long *);

vector unsigned char vec_lvsr (int, const unsigned long *);
vector unsigned char vec_lvsr (int, const long *);

vector unsigned char vec_lvsl (int, const double *);
vector unsigned char vec_lvsr (int, const double *);

vector double vec_vsx_ld (int, const vector double *);
vector double vec_vsx_ld (int, const double *);
vector float vec_vsx_ld (int, const vector float *);
vector float vec_vsx_ld (int, const float *);
vector bool int vec_vsx_ld (int, const vector bool int *);
vector signed int vec_vsx_ld (int, const vector signed int *);
vector signed int vec_vsx_ld (int, const int *);
vector signed int vec_vsx_ld (int, const long *);
vector unsigned int vec_vsx_ld (int, const vector unsigned int *);
vector unsigned int vec_vsx_ld (int, const unsigned int *);
vector unsigned int vec_vsx_ld (int, const unsigned long *);
vector bool short vec_vsx_ld (int, const vector bool short *);
vector pixel vec_vsx_ld (int, const vector pixel *);
vector signed short vec_vsx_ld (int, const vector signed short *);
vector signed short vec_vsx_ld (int, const short *);
vector unsigned short vec_vsx_ld (int, const vector unsigned short *);
vector unsigned short vec_vsx_ld (int, const unsigned short *);
vector bool char vec_vsx_ld (int, const vector bool char *);
vector signed char vec_vsx_ld (int, const vector signed char *);
vector signed char vec_vsx_ld (int, const signed char *);
vector unsigned char vec_vsx_ld (int, const vector unsigned char *);
vector unsigned char vec_vsx_ld (int, const unsigned char *);

void vec_vsx_st (vector double, int, vector double *);
void vec_vsx_st (vector double, int, double *);
```

```
void vec_vsx_st (vector float, int, vector float *);
void vec_vsx_st (vector float, int, float *);
void vec_vsx_st (vector signed int, int, vector signed int *);
void vec_vsx_st (vector signed int, int, int *);
void vec_vsx_st (vector unsigned int, int, vector unsigned int *);
void vec_vsx_st (vector unsigned int, int, unsigned int *);
void vec_vsx_st (vector bool int, int, vector bool int *);
void vec_vsx_st (vector bool int, int, unsigned int *);
void vec_vsx_st (vector bool int, int, int *);
void vec_vsx_st (vector signed short, int, vector signed short *);
void vec_vsx_st (vector signed short, int, short *);
void vec_vsx_st (vector unsigned short, int, vector unsigned short *);
void vec_vsx_st (vector unsigned short, int, unsigned short *);
void vec_vsx_st (vector bool short, int, vector bool short *);
void vec_vsx_st (vector bool short, int, unsigned short *);
void vec_vsx_st (vector pixel, int, vector pixel *);
void vec_vsx_st (vector pixel, int, unsigned short *);
void vec_vsx_st (vector pixel, int, short *);
void vec_vsx_st (vector bool short, int, short *);
void vec_vsx_st (vector signed char, int, vector signed char *);
void vec_vsx_st (vector signed char, int, signed char *);
void vec_vsx_st (vector unsigned char, int, vector unsigned char *);
void vec_vsx_st (vector unsigned char, int, unsigned char *);
void vec_vsx_st (vector bool char, int, vector bool char *);
void vec_vsx_st (vector bool char, int, unsigned char *);
void vec_vsx_st (vector bool char, int, signed char *);

vector double vec_xxpermdi (vector double, vector double, const int);
vector float vec_xxpermdi (vector float, vector float, const int);
vector long long vec_xxpermdi (vector long long, vector long long, const int);
vector unsigned long long vec_xxpermdi (vector unsigned long long,
                                        vector unsigned long long, const int);
vector int vec_xxpermdi (vector int, vector int, const int);
vector unsigned int vec_xxpermdi (vector unsigned int,
                                  vector unsigned int, const int);
vector short vec_xxpermdi (vector short, vector short, const int);
vector unsigned short vec_xxpermdi (vector unsigned short,
                                    vector unsigned short, const int);
vector signed char vec_xxpermdi (vector signed char, vector signed char,
                                 const int);
vector unsigned char vec_xxpermdi (vector unsigned char,
                                   vector unsigned char, const int);

vector double vec_xxsldi (vector double, vector double, int);
vector float vec_xxsldi (vector float, vector float, int);
vector long long vec_xxsldi (vector long long, vector long long, int);
vector unsigned long long vec_xxsldi (vector unsigned long long,
                                      vector unsigned long long, int);
vector int vec_xxsldi (vector int, vector int, int);
vector unsigned int vec_xxsldi (vector unsigned int, vector unsigned int, int);
vector short vec_xxsldi (vector short, vector short, int);
vector unsigned short vec_xxsldi (vector unsigned short,
                                  vector unsigned short, int);
vector signed char vec_xxsldi (vector signed char, vector signed char, int);
vector unsigned char vec_xxsldi (vector unsigned char,
                                 vector unsigned char, int);
```

Note that the '`vec_ld`' and '`vec_st`' built-in functions always generate the AltiVec '`LVX`' and '`STVX`' instructions even if the VSX instruction set is available. The '`vec_vsx_ld`' and

'vec_vsx_st' built-in functions always generate the VSX 'LXVD2X', 'LXVW4X', 'STXVD2X', and 'STXVW4X' instructions.

6.60.23.3 PowerPC AltiVec Built-in Functions Available on ISA 2.07

If the ISA 2.07 additions to the vector/scalar (power8-vector) instruction set are available, the following additional functions are available for both 32-bit and 64-bit targets. For 64-bit targets, you can use *vector long* instead of *vector long long*, *vector bool long* instead of *vector bool long long*, and *vector unsigned long* instead of *vector unsigned long long*.

Only functions excluded from the PVIPR are listed here.

```
vector long long vec_vaddudm (vector long long, vector long long);
vector long long vec_vaddudm (vector bool long long, vector long long);
vector long long vec_vaddudm (vector long long, vector bool long long);
vector unsigned long long vec_vaddudm (vector unsigned long long,
                                       vector unsigned long long);
vector unsigned long long vec_vaddudm (vector bool unsigned long long,
                                       vector unsigned long long);
vector unsigned long long vec_vaddudm (vector unsigned long long,
                                       vector bool unsigned long long);

vector long long vec_vclz (vector long long);
vector unsigned long long vec_vclz (vector unsigned long long);
vector int vec_vclz (vector int);
vector unsigned int vec_vclz (vector int);
vector short vec_vclz (vector short);
vector unsigned short vec_vclz (vector unsigned short);
vector signed char vec_vclz (vector signed char);
vector unsigned char vec_vclz (vector unsigned char);

vector signed char vec_vclzb (vector signed char);
vector unsigned char vec_vclzb (vector unsigned char);

vector long long vec_vclzd (vector long long);
vector unsigned long long vec_vclzd (vector unsigned long long);

vector short vec_vclzh (vector short);
vector unsigned short vec_vclzh (vector unsigned short);

vector int vec_vclzw (vector int);
vector unsigned int vec_vclzw (vector int);

vector signed char vec_vgbbd (vector signed char);
vector unsigned char vec_vgbbd (vector unsigned char);

vector long long vec_vmaxsd (vector long long, vector long long);

vector unsigned long long vec_vmaxud (vector unsigned long long,
                                      unsigned vector long long);

vector long long vec_vminsd (vector long long, vector long long);

vector unsigned long long vec_vminud (vector long long, vector long long);

vector int vec_vpksdss (vector long long, vector long long);
vector unsigned int vec_vpksdss (vector long long, vector long long);
```

```
vector unsigned int vec_vpkudus (vector unsigned long long,
                                 vector unsigned long long);

vector int vec_vpkudum (vector long long, vector long long);
vector unsigned int vec_vpkudum (vector unsigned long long,
                                 vector unsigned long long);
vector bool int vec_vpkudum (vector bool long long, vector bool long long);

vector long long vec_vpopcnt (vector long long);
vector unsigned long long vec_vpopcnt (vector unsigned long long);
vector int vec_vpopcnt (vector int);
vector unsigned int vec_vpopcnt (vector int);
vector short vec_vpopcnt (vector short);
vector unsigned short vec_vpopcnt (vector unsigned short);
vector signed char vec_vpopcnt (vector signed char);
vector unsigned char vec_vpopcnt (vector unsigned char);

vector signed char vec_vpopcntb (vector signed char);
vector unsigned char vec_vpopcntb (vector unsigned char);

vector long long vec_vpopcntd (vector long long);
vector unsigned long long vec_vpopcntd (vector unsigned long long);

vector short vec_vpopcnth (vector short);
vector unsigned short vec_vpopcnth (vector unsigned short);

vector int vec_vpopcntw (vector int);
vector unsigned int vec_vpopcntw (vector int);

vector long long vec_vrld (vector long long, vector unsigned long long);
vector unsigned long long vec_vrld (vector unsigned long long,
                                    vector unsigned long long);

vector long long vec_vsld (vector long long, vector unsigned long long);
vector long long vec_vsld (vector unsigned long long,
                           vector unsigned long long);

vector long long vec_vsrad (vector long long, vector unsigned long long);
vector unsigned long long vec_vsrad (vector unsigned long long,
                                     vector unsigned long long);

vector long long vec_vsrd (vector long long, vector unsigned long long);
vector unsigned long long char vec_vsrd (vector unsigned long long,
                                         vector unsigned long long);

vector long long vec_vsubudm (vector long long, vector long long);
vector long long vec_vsubudm (vector bool long long, vector long long);
vector long long vec_vsubudm (vector long long, vector bool long long);
vector unsigned long long vec_vsubudm (vector unsigned long long,
                                       vector unsigned long long);
vector unsigned long long vec_vsubudm (vector bool long long,
                                       vector unsigned long long);
vector unsigned long long vec_vsubudm (vector unsigned long long,
                                       vector bool long long);

vector long long vec_vupkhsw (vector int);
vector unsigned long long vec_vupkhsw (vector unsigned int);
```

```
vector long long vec_vupklsw (vector int);
vector unsigned long long vec_vupklsw (vector int);
```

If the ISA 2.07 additions to the vector/scalar (power8-vector) instruction set are available, the following additional functions are available for 64-bit targets. New vector types (*vector __int128* and *vector __uint128*) are available to hold the *__int128* and *__uint128* types to use these builtins.

The normal vector extract, and set operations work on *vector __int128* and *vector __uint128* types, but the index value must be 0.

Only functions excluded from the PVIPR are listed here.

```
vector __int128 vec_vaddcuq (vector __int128, vector __int128);
vector __uint128 vec_vaddcuq (vector __uint128, vector __uint128);

vector __int128 vec_vadduqm (vector __int128, vector __int128);
vector __uint128 vec_vadduqm (vector __uint128, vector __uint128);

vector __int128 vec_vaddecuq (vector __int128, vector __int128,
                              vector __int128);
vector __uint128 vec_vaddecuq (vector __uint128, vector __uint128,
                               vector __uint128);

vector __int128 vec_vaddeuqm (vector __int128, vector __int128,
                              vector __int128);
vector __uint128 vec_vaddeuqm (vector __uint128, vector __uint128,
                               vector __uint128);

vector __int128 vec_vsubecuq (vector __int128, vector __int128,
                              vector __int128);
vector __uint128 vec_vsubecuq (vector __uint128, vector __uint128,
                               vector __uint128);

vector __int128 vec_vsubeuqm (vector __int128, vector __int128,
                              vector __int128);
vector __uint128 vec_vsubeuqm (vector __uint128, vector __uint128,
                               vector __uint128);

vector __int128 vec_vsubcuq (vector __int128, vector __int128);
vector __uint128 vec_vsubcuq (vector __uint128, vector __uint128);

__int128 vec_vsubuqm (__int128, __int128);
__uint128 vec_vsubuqm (__uint128, __uint128);

vector __int128 __builtin_bcdadd (vector __int128, vector __int128, const int);
vector unsigned char __builtin_bcdadd (vector unsigned char, vector unsigned char,
                                       const int);
int __builtin_bcdadd_lt (vector __int128, vector __int128, const int);
int __builtin_bcdadd_lt (vector unsigned char, vector unsigned char, const int);
int __builtin_bcdadd_eq (vector __int128, vector __int128, const int);
int __builtin_bcdadd_eq (vector unsigned char, vector unsigned char, const int);
int __builtin_bcdadd_gt (vector __int128, vector __int128, const int);
int __builtin_bcdadd_gt (vector unsigned char, vector unsigned char, const int);
int __builtin_bcdadd_ov (vector __int128, vector __int128, const int);
int __builtin_bcdadd_ov (vector unsigned char, vector unsigned char, const int);

vector __int128 __builtin_bcdsub (vector __int128, vector __int128, const int);
vector unsigned char __builtin_bcdsub (vector unsigned char, vector unsigned char,
                                       const int);
```

```
int __builtin_bcdsub_lt (vector __int128, vector __int128, const int);
int __builtin_bcdsub_lt (vector unsigned char, vector unsigned char, const int);
int __builtin_bcdsub_eq (vector __int128, vector __int128, const int);
int __builtin_bcdsub_eq (vector unsigned char, vector unsigned char, const int);
int __builtin_bcdsub_gt (vector __int128, vector __int128, const int);
int __builtin_bcdsub_gt (vector unsigned char, vector unsigned char, const int);
int __builtin_bcdsub_ov (vector __int128, vector __int128, const int);
int __builtin_bcdsub_ov (vector unsigned char, vector unsigned char, const int);
```

6.60.23.4 PowerPC AltiVec Built-in Functions Available on ISA 3.0

The following additional built-in functions are also available for the PowerPC family of processors, starting with ISA 3.0 (`-mcpu=power9`) or later.

Only instructions excluded from the PVIPR are listed here.

```
unsigned int scalar_extract_exp (double source);
unsigned long long int scalar_extract_exp (__ieee128 source);

unsigned long long int scalar_extract_sig (double source);
unsigned __int128 scalar_extract_sig (__ieee128 source);

double scalar_insert_exp (unsigned long long int significand,
                          unsigned long long int exponent);
double scalar_insert_exp (double significand, unsigned long long int exponent);

ieee_128 scalar_insert_exp (unsigned __int128 significand,
                            unsigned long long int exponent);
ieee_128 scalar_insert_exp (ieee_128 significand, unsigned long long int exponent);

int scalar_cmp_exp_gt (double arg1, double arg2);
int scalar_cmp_exp_lt (double arg1, double arg2);
int scalar_cmp_exp_eq (double arg1, double arg2);
int scalar_cmp_exp_unordered (double arg1, double arg2);

bool scalar_test_data_class (float source, const int condition);
bool scalar_test_data_class (double source, const int condition);
bool scalar_test_data_class (__ieee128 source, const int condition);

bool scalar_test_neg (float source);
bool scalar_test_neg (double source);
bool scalar_test_neg (__ieee128 source);
```

The `scalar_extract_exp` and `scalar_extract_sig` functions require a 64-bit environment supporting ISA 3.0 or later. The `scalar_extract_exp` and `scalar_extract_sig` built-in functions return the significand and the biased exponent value respectively of their `source` arguments. When supplied with a 64-bit `source` argument, the result returned by `scalar_extract_sig` has the `0x0010000000000000` bit set if the function's `source` argument is in normalized form. Otherwise, this bit is set to 0. When supplied with a 128-bit `source` argument, the `0x00010000000000000000000000000000` bit of the result is treated similarly. Note that the sign of the significand is not represented in the result returned from the `scalar_extract_sig` function. Use the `scalar_test_neg` function to test the sign of its `double` argument.

The `scalar_insert_exp` functions require a 64-bit environment supporting ISA 3.0 or later. When supplied with a 64-bit first argument, the `scalar_insert_exp` built-in function returns a double-precision floating point value that is constructed by assembling the values

of its `significand` and `exponent` arguments. The sign of the result is copied from the most significant bit of the `significand` argument. The significand and exponent components of the result are composed of the least significant 11 bits of the `exponent` argument and the least significant 52 bits of the `significand` argument respectively.

When supplied with a 128-bit first argument, the `scalar_insert_exp` built-in function returns a quad-precision ieee floating point value. The sign bit of the result is copied from the most significant bit of the `significand` argument. The significand and exponent components of the result are composed of the least significant 15 bits of the `exponent` argument and the least significant 112 bits of the `significand` argument respectively.

The `scalar_cmp_exp_gt`, `scalar_cmp_exp_lt`, `scalar_cmp_exp_eq`, and `scalar_cmp_exp_unordered` built-in functions return a non-zero value if `arg1` is greater than, less than, equal to, or not comparable to `arg2` respectively. The arguments are not comparable if one or the other equals NaN (not a number).

The `scalar_test_data_class` built-in function returns 1 if any of the condition tests enabled by the value of the `condition` variable are true, and 0 otherwise. The `condition` argument must be a compile-time constant integer with value not exceeding 127. The `condition` argument is encoded as a bitmask with each bit enabling the testing of a different condition, as characterized by the following:

```
0x40    Test for NaN
0x20    Test for +Infinity
0x10    Test for -Infinity
0x08    Test for +Zero
0x04    Test for -Zero
0x02    Test for +Denormal
0x01    Test for -Denormal
```

The `scalar_test_neg` built-in function returns 1 if its `source` argument holds a negative value, 0 otherwise.

The following built-in functions are also available for the PowerPC family of processors, starting with ISA 3.0 or later (`-mcpu=power9`). These string functions are described separately in order to group the descriptions closer to the function prototypes.

Only functions excluded from the PVIPR are listed here.

```
int vec_all_nez (vector signed char, vector signed char);
int vec_all_nez (vector unsigned char, vector unsigned char);
int vec_all_nez (vector signed short, vector signed short);
int vec_all_nez (vector unsigned short, vector unsigned short);
int vec_all_nez (vector signed int, vector signed int);
int vec_all_nez (vector unsigned int, vector unsigned int);

int vec_any_eqz (vector signed char, vector signed char);
int vec_any_eqz (vector unsigned char, vector unsigned char);
int vec_any_eqz (vector signed short, vector signed short);
int vec_any_eqz (vector unsigned short, vector unsigned short);
int vec_any_eqz (vector signed int, vector signed int);
int vec_any_eqz (vector unsigned int, vector unsigned int);

signed char vec_xlx (unsigned int index, vector signed char data);
unsigned char vec_xlx (unsigned int index, vector unsigned char data);
signed short vec_xlx (unsigned int index, vector signed short data);
unsigned short vec_xlx (unsigned int index, vector unsigned short data);
signed int vec_xlx (unsigned int index, vector signed int data);
```

```
unsigned int vec_xlx (unsigned int index, vector unsigned int data);
float vec_xlx (unsigned int index, vector float data);

signed char vec_xrx (unsigned int index, vector signed char data);
unsigned char vec_xrx (unsigned int index, vector unsigned char data);
signed short vec_xrx (unsigned int index, vector signed short data);
unsigned short vec_xrx (unsigned int index, vector unsigned short data);
signed int vec_xrx (unsigned int index, vector signed int data);
unsigned int vec_xrx (unsigned int index, vector unsigned int data);
float vec_xrx (unsigned int index, vector float data);
```

The `vec_all_nez`, `vec_any_eqz`, and `vec_cmpnez` perform pairwise comparisons between the elements at the same positions within their two vector arguments. The `vec_all_nez` function returns a non-zero value if and only if all pairwise comparisons are not equal and no element of either vector argument contains a zero. The `vec_any_eqz` function returns a non-zero value if and only if at least one pairwise comparison is equal or if at least one element of either vector argument contains a zero. The `vec_cmpnez` function returns a vector of the same type as its two arguments, within which each element consists of all ones to denote that either the corresponding elements of the incoming arguments are not equal or that at least one of the corresponding elements contains zero. Otherwise, the element of the returned vector contains all zeros.

The `vec_xlx` and `vec_xrx` functions extract the single element selected by the `index` argument from the vector represented by the `data` argument. The `index` argument always specifies a byte offset, regardless of the size of the vector element. With `vec_xlx`, `index` is the offset of the first byte of the element to be extracted. With `vec_xrx`, `index` represents the last byte of the element to be extracted, measured from the right end of the vector. In other words, the last byte of the element to be extracted is found at position `(15 - index)`. There is no requirement that `index` be a multiple of the vector element size. However, if the size of the vector element added to `index` is greater than 15, the content of the returned value is undefined.

The following functions are also available if the ISA 3.0 instruction set additions (`-mcpu=power9`) are available.

Only functions excluded from the PVIPR are listed here.

```
vector long long vec_vctz (vector long long);
vector unsigned long long vec_vctz (vector unsigned long long);
vector int vec_vctz (vector int);
vector unsigned int vec_vctz (vector int);
vector short vec_vctz (vector short);
vector unsigned short vec_vctz (vector unsigned short);
vector signed char vec_vctz (vector signed char);
vector unsigned char vec_vctz (vector unsigned char);

vector signed char vec_vctzb (vector signed char);
vector unsigned char vec_vctzb (vector unsigned char);

vector long long vec_vctzd (vector long long);
vector unsigned long long vec_vctzd (vector unsigned long long);

vector short vec_vctzh (vector short);
vector unsigned short vec_vctzh (vector unsigned short);

vector int vec_vctzw (vector int);
vector unsigned int vec_vctzw (vector int);
```

```
vector int vec_vprtyb (vector int);
vector unsigned int vec_vprtyb (vector unsigned int);
vector long long vec_vprtyb (vector long long);
vector unsigned long long vec_vprtyb (vector unsigned long long);

vector int vec_vprtybw (vector int);
vector unsigned int vec_vprtybw (vector unsigned int);

vector long long vec_vprtybd (vector long long);
vector unsigned long long vec_vprtybd (vector unsigned long long);
```

On 64-bit targets, if the ISA 3.0 additions (`-mcpu=power9`) are available:

```
vector long vec_vprtyb (vector long);
vector unsigned long vec_vprtyb (vector unsigned long);
vector __int128 vec_vprtyb (vector __int128);
vector __uint128 vec_vprtyb (vector __uint128);

vector long vec_vprtybd (vector long);
vector unsigned long vec_vprtybd (vector unsigned long);

vector __int128 vec_vprtybq (vector __int128);
vector __uint128 vec_vprtybd (vector __uint128);
```

The following built-in functions are available for the PowerPC family of processors, starting with ISA 3.0 or later (`-mcpu=power9`).

Only functions excluded from the PVIPR are listed here.

```
__vector unsigned char
vec_absdb (__vector unsigned char arg1, __vector unsigned char arg2);
__vector unsigned short
vec_absdh (__vector unsigned short arg1, __vector unsigned short arg2);
__vector unsigned int
vec_absdw (__vector unsigned int arg1, __vector unsigned int arg2);
```

The `vec_absd`, `vec_absdb`, `vec_absdh`, and `vec_absdw` built-in functions each computes the absolute differences of the pairs of vector elements supplied in its two vector arguments, placing the absolute differences into the corresponding elements of the vector result.

The following built-in functions are available for the PowerPC family of processors, starting with ISA 3.0 or later (`-mcpu=power9`):

```
vector unsigned int vec_vrlnm (vector unsigned int, vector unsigned int);
vector unsigned long long vec_vrlnm (vector unsigned long long,
                                     vector unsigned long long);
```

The result of `vec_vrlnm` is obtained by rotating each element of the first argument vector left and ANDing it with a mask. The second argument vector contains the mask beginning in bits 11:15, the mask end in bits 19:23, and the shift count in bits 27:31, of each element.

If the cryptographic instructions are enabled (`-mcrypto` or `-mcpu=power8`), the following builtins are enabled.

Only functions excluded from the PVIPR are listed here.

```
vector unsigned long long __builtin_crypto_vsbox (vector unsigned long long);

vector unsigned long long __builtin_crypto_vcipher (vector unsigned long long,
                                                    vector unsigned long long);

vector unsigned long long __builtin_crypto_vcipherlast
```

```
                                    (vector unsigned long long,
                                     vector unsigned long long);

vector unsigned long long __builtin_crypto_vncipher (vector unsigned long long,
                                                     vector unsigned long long);

vector unsigned long long __builtin_crypto_vncipherlast (vector unsigned long long,
                                                         vector unsigned long long);

vector unsigned char __builtin_crypto_vpermxor (vector unsigned char,
                                                vector unsigned char,
                                                vector unsigned char);

vector unsigned short __builtin_crypto_vpermxor (vector unsigned short,
                                                 vector unsigned short,
                                                 vector unsigned short);

vector unsigned int __builtin_crypto_vpermxor (vector unsigned int,
                                               vector unsigned int,
                                               vector unsigned int);

vector unsigned long long __builtin_crypto_vpermxor (vector unsigned long long,
                                                     vector unsigned long long,
                                                     vector unsigned long long);

vector unsigned char __builtin_crypto_vpmsumb (vector unsigned char,
                                               vector unsigned char);

vector unsigned short __builtin_crypto_vpmsumh (vector unsigned short,
                                                vector unsigned short);

vector unsigned int __builtin_crypto_vpmsumw (vector unsigned int,
                                              vector unsigned int);

vector unsigned long long __builtin_crypto_vpmsumd (vector unsigned long long,
                                                    vector unsigned long long);

vector unsigned long long __builtin_crypto_vshasigmad (vector unsigned long long,
                                                       int, int);

vector unsigned int __builtin_crypto_vshasigmaw (vector unsigned int, int, int);
```

The second argument to *__builtin_crypto_vshasigmad* and *__builtin_crypto_vshasigmaw* must be a constant integer that is 0 or 1. The third argument to these built-in functions must be a constant integer in the range of 0 to 15.

The following sign extension builtins are provided:

```
vector signed int vec_signexti (vector signed char a);
vector signed long long vec_signextll (vector signed char a);
vector signed int vec_signexti (vector signed short a);
vector signed long long vec_signextll (vector signed short a);
vector signed long long vec_signextll (vector signed int a);
vector signed long long vec_signextq (vector signed long long a);
```

Each element of the result is produced by sign-extending the element of the input vector that would fall in the least significant portion of the result element. For example, a sign-extension of a vector signed char to a vector signed long long will sign extend the rightmost byte of each doubleword.

6.60.23.5 PowerPC AltiVec Built-in Functions Available on ISA 3.1

The following additional built-in functions are also available for the PowerPC family of processors, starting with ISA 3.1 (`-mcpu=power10`):

```
vector unsigned long long int
vec_cfuge (vector unsigned long long int, vector unsigned long long int);
```

Perform a vector centrifuge operation, as if implemented by the `vcfuged` instruction.

```
vector unsigned long long int
vec_cntlzm (vector unsigned long long int, vector unsigned long long int);
```

Perform a vector count leading zeros under bit mask operation, as if implemented by the `vclzdm` instruction.

```
vector unsigned long long int
vec_cnttzm (vector unsigned long long int, vector unsigned long long int);
```

Perform a vector count trailing zeros under bit mask operation, as if implemented by the `vctzdm` instruction.

```
vector signed char
vec_clrl (vector signed char a, unsigned int n);
vector unsigned char
vec_clrl (vector unsigned char a, unsigned int n);
```

Clear the left-most `(16 - n)` bytes of vector argument `a`, as if implemented by the `vclrlb` instruction on a big-endian target and by the `vclrrb` instruction on a little-endian target. A value of `n` that is greater than 16 is treated as if it equaled 16.

```
vector signed char
vec_clrr (vector signed char a, unsigned int n);
vector unsigned char
vec_clrr (vector unsigned char a, unsigned int n);
```

Clear the right-most `(16 - n)` bytes of vector argument `a`, as if implemented by the `vclrrb` instruction on a big-endian target and by the `vclrlb` instruction on a little-endian target. A value of `n` that is greater than 16 is treated as if it equaled 16.

```
vector unsigned long long int
vec_gnb (vector unsigned __int128, const unsigned char);
```

Perform a 128-bit vector gather operation, as if implemented by the `vgnb` instruction. The second argument must be a literal integer value between 2 and 7 inclusive.

Vector Extract

```
vector unsigned long long int
vec_extractl (vector unsigned char, vector unsigned char, unsigned int);
vector unsigned long long int
vec_extractl (vector unsigned short, vector unsigned short, unsigned int);
vector unsigned long long int
vec_extractl (vector unsigned int, vector unsigned int, unsigned int);
vector unsigned long long int
vec_extractl (vector unsigned long long, vector unsigned long long, unsigned int);
```

Extract an element from two concatenated vectors starting at the given byte index in natural-endian order, and place it zero-extended in doubleword 1 of the result according to natural element order. If the byte index is out of range for the data type, the intrinsic will be rejected. For little-endian, this output will match the placement by the hardware instruction, i.e., dword[0] in RTL notation. For big-endian, an additional instruction is needed to move it from the "left" doubleword to the "right" one. For little-endian, semantics matching the `vextdubvrx`, `vextduhvrx`, `vextduwvrx` instruction will be generated, while

for big-endian, semantics matching the `vextdubvlx`, `vextduhvlx`, `vextduwvlx` instructions will be generated. Note that some fairly anomalous results can be generated if the byte index is not aligned on an element boundary for the element being extracted. This is a limitation of the bi-endian vector programming model is consistent with the limitation on `vec_perm`.

```
vector unsigned long long int
vec_extracth (vector unsigned char, vector unsigned char, unsigned int);
vector unsigned long long int
vec_extracth (vector unsigned short, vector unsigned short,
      unsigned int);
vector unsigned long long int
vec_extracth (vector unsigned int, vector unsigned int, unsigned int);
vector unsigned long long int
vec_extracth (vector unsigned long long, vector unsigned long long,
      unsigned int);
```

Extract an element from two concatenated vectors starting at the given byte index. The index is based on big endian order for a little endian system. Similarly, the index is based on little endian order for a big endian system. The extraced elements are zero-extended and put in doubleword 1 according to natural element order. If the byte index is out of range for the data type, the intrinsic will be rejected. For little-endian, this output will match the placement by the hardware instruction (vextdubvrx, vextduhvrx, vextduwvrx, vextddvrx) i.e., dword[0] in RTL notation. For big-endian, an additional instruction is needed to move it from the "left" doubleword to the "right" one. For little-endian, semantics matching the `vextdubvlx`, `vextduhvlx`, `vextduwvlx` instructions will be generated, while for big-endian, semantics matching the `vextdubvrx`, `vextduhvrx`, `vextduwvrx` instructions will be generated. Note that some fairly anomalous results can be generated if the byte index is not aligned on the element boundary for the element being extracted. This is a limitation of the bi-endian vector programming model consistent with the limitation on `vec_perm`.

```
vector unsigned long long int
vec_pdep (vector unsigned long long int, vector unsigned long long int);
```

Perform a vector parallel bits deposit operation, as if implemented by the `vpdepd` instruction.

Vector Insert

```
vector unsigned char
vec_insertl (unsigned char, vector unsigned char, unsigned int);
vector unsigned short
vec_insertl (unsigned short, vector unsigned short, unsigned int);
vector unsigned int
vec_insertl (unsigned int, vector unsigned int, unsigned int);
vector unsigned long long
vec_insertl (unsigned long long, vector unsigned long long,
      unsigned int);
vector unsigned char
vec_insertl (vector unsigned char, vector unsigned char, unsigned int;
vector unsigned short
vec_insertl (vector unsigned short, vector unsigned short,
      unsigned int);
vector unsigned int
vec_insertl (vector unsigned int, vector unsigned int, unsigned int);
```

Let src be the first argument, when the first argument is a scalar, or the rightmost element of the left doubleword of the first argument, when the first argument is a vector. Insert

the source into the destination at the position given by the third argument, using natural element order in the second argument. The rest of the second argument is unchanged. If the byte index is greater than 14 for halfwords, greater than 12 for words, or greater than 8 for doublewords the result is undefined. For little-endian, the generated code will be semantically equivalent to `vins[bhwd]rx` instructions. Similarly for big-endian it will be semantically equivalent to `vins[bhwd]lx`. Note that some fairly anomalous results can be generated if the byte index is not aligned on an element boundary for the type of element being inserted.

```
vector unsigned char
vec_inserth (unsigned char, vector unsigned char, unsigned int);
vector unsigned short
vec_inserth (unsigned short, vector unsigned short, unsigned int);
vector unsigned int
vec_inserth (unsigned int, vector unsigned int, unsigned int);
vector unsigned long long
vec_inserth (unsigned long long, vector unsigned long long,
      unsigned int);
vector unsigned char
vec_inserth (vector unsigned char, vector unsigned char, unsigned int);
vector unsigned short
vec_inserth (vector unsigned short, vector unsigned short,
      unsigned int);
vector unsigned int
vec_inserth (vector unsigned int, vector unsigned int, unsigned int);
```

Let src be the first argument, when the first argument is a scalar, or the rightmost element of the first argument, when the first argument is a vector. Insert src into the second argument at the position identified by the third argument, using opposite element order in the second argument, and leaving the rest of the second argument unchanged. If the byte index is greater than 14 for halfwords, 12 for words, or 8 for doublewords, the intrinsic will be rejected. Note that the underlying hardware instruction uses the same register for the second argument and the result. For little-endian, the code generation will be semantically equivalent to `vins[bhwd]lx`, while for big-endian it will be semantically equivalent to `vins[bhwd]rx`. Note that some fairly anomalous results can be generated if the byte index is not aligned on an element boundary for the sort of element being inserted.

Vector Replace Element

```
vector signed int vec_replace_elt (vector signed int, signed int,
      const int);
vector unsigned int vec_replace_elt (vector unsigned int,
      unsigned int, const int);
vector float vec_replace_elt (vector float, float, const int);
vector signed long long vec_replace_elt (vector signed long long,
      signed long long, const int);
vector unsigned long long vec_replace_elt (vector unsigned long long,
      unsigned long long, const int);
vector double rec_replace_elt (vector double, double, const int);
```

The third argument (constrained to [0,3]) identifies the natural-endian element number of the first argument that will be replaced by the second argument to produce the result. The other elements of the first argument will remain unchanged in the result.

If it's desirable to insert a word at an unaligned position, use vec_replace_unaligned instead.

Vector Replace Unaligned

```
vector unsigned char vec_replace_unaligned (vector unsigned char,
      signed int, const int);
vector unsigned char vec_replace_unaligned (vector unsigned char,
      unsigned int, const int);
vector unsigned char vec_replace_unaligned (vector unsigned char,
      float, const int);
vector unsigned char vec_replace_unaligned (vector unsigned char,
      signed long long, const int);
vector unsigned char vec_replace_unaligned (vector unsigned char,
      unsigned long long, const int);
vector unsigned char vec_replace_unaligned (vector unsigned char,
      double, const int);
```

The second argument replaces a portion of the first argument to produce the result, with the rest of the first argument unchanged in the result. The third argument identifies the byte index (using left-to-right, or big-endian order) where the high-order byte of the second argument will be placed, with the remaining bytes of the second argument placed naturally "to the right" of the high-order byte.

The programmer is responsible for understanding the endianness issues involved with the first argument and the result.

Vector Shift Left Double Bit Immediate

```
vector signed char vec_sldb (vector signed char, vector signed char,
      const unsigned int);
vector unsigned char vec_sldb (vector unsigned char,
      vector unsigned char, const unsigned int);
vector signed short vec_sldb (vector signed short, vector signed short,
      const unsigned int);
vector unsigned short vec_sldb (vector unsigned short,
      vector unsigned short, const unsigned int);
vector signed int vec_sldb (vector signed int, vector signed int,
      const unsigned int);
vector unsigned int vec_sldb (vector unsigned int, vector unsigned int,
      const unsigned int);
vector signed long long vec_sldb (vector signed long long,
      vector signed long long, const unsigned int);
vector unsigned long long vec_sldb (vector unsigned long long,
      vector unsigned long long, const unsigned int);
```

Shift the combined input vectors left by the amount specified by the low-order three bits of the third argument, and return the leftmost remaining 128 bits. Code using this instruction must be endian-aware.

Vector Shift Right Double Bit Immediate

```
vector signed char vec_srdb (vector signed char, vector signed char,
      const unsigned int);
vector unsigned char vec_srdb (vector unsigned char, vector unsigned char,
      const unsigned int);
vector signed short vec_srdb (vector signed short, vector signed short,
      const unsigned int);
vector unsigned short vec_srdb (vector unsigned short, vector unsigned short,
      const unsigned int);
vector signed int vec_srdb (vector signed int, vector signed int,
      const unsigned int);
vector unsigned int vec_srdb (vector unsigned int, vector unsigned int,
      const unsigned int);
```

```
vector signed long long vec_srdb (vector signed long long,
      vector signed long long, const unsigned int);
vector unsigned long long vec_srdb (vector unsigned long long,
      vector unsigned long long, const unsigned int);
```

Shift the combined input vectors right by the amount specified by the low-order three bits of the third argument, and return the remaining 128 bits. Code using this built-in must be endian-aware.

Vector Splat

```
vector signed int vec_splati (const signed int);
vector float vec_splati (const float);
```

Splat a 32-bit immediate into a vector of words.

```
vector double vec_splatid (const float);
```

Convert a single precision floating-point value to double-precision and splat the result to a vector of double-precision floats.

```
vector signed int vec_splati_ins (vector signed int,
      const unsigned int, const signed int);
vector unsigned int vec_splati_ins (vector unsigned int,
      const unsigned int, const unsigned int);
vector float vec_splati_ins (vector float, const unsigned int,
      const float);
```

Argument 2 must be either 0 or 1. Splat the value of argument 3 into the word identified by argument 2 of each doubleword of argument 1 and return the result. The other words of argument 1 are unchanged.

Vector Blend Variable

```
vector signed char vec_blendv (vector signed char, vector signed char,
      vector unsigned char);
vector unsigned char vec_blendv (vector unsigned char,
      vector unsigned char, vector unsigned char);
vector signed short vec_blendv (vector signed short,
      vector signed short, vector unsigned short);
vector unsigned short vec_blendv (vector unsigned short,
      vector unsigned short, vector unsigned short);
vector signed int vec_blendv (vector signed int, vector signed int,
      vector unsigned int);
vector unsigned int vec_blendv (vector unsigned int,
      vector unsigned int, vector unsigned int);
vector signed long long vec_blendv (vector signed long long,
      vector signed long long, vector unsigned long long);
vector unsigned long long vec_blendv (vector unsigned long long,
      vector unsigned long long, vector unsigned long long);
vector float vec_blendv (vector float, vector float,
      vector unsigned int);
vector double vec_blendv (vector double, vector double,
      vector unsigned long long);
```

Blend the first and second argument vectors according to the sign bits of the corresponding elements of the third argument vector. This is similar to the `vsel` and `xxsel` instructions but for bigger elements.

Vector Permute Extended

```
vector signed char vec_permx (vector signed char, vector signed char,
      vector unsigned char, const int);
vector unsigned char vec_permx (vector unsigned char,
```

```
      vector unsigned char, vector unsigned char, const int);
vector signed short vec_permx (vector signed short,
      vector signed short, vector unsigned char, const int);
vector unsigned short vec_permx (vector unsigned short,
      vector unsigned short, vector unsigned char, const int);
vector signed int vec_permx (vector signed int, vector signed int,
      vector unsigned char, const int);
vector unsigned int vec_permx (vector unsigned int,
      vector unsigned int, vector unsigned char, const int);
vector signed long long vec_permx (vector signed long long,
      vector signed long long, vector unsigned char, const int);
vector unsigned long long vec_permx (vector unsigned long long,
      vector unsigned long long, vector unsigned char, const int);
vector float (vector float, vector float, vector unsigned char,
      const int);
vector double (vector double, vector double, vector unsigned char,
      const int);
```

Perform a partial permute of the first two arguments, which form a 32-byte section of an emulated vector up to 256 bytes wide, using the partial permute control vector in the third argument. The fourth argument (constrained to values of 0-7) identifies which 32-byte section of the emulated vector is contained in the first two arguments.

```
vector unsigned long long int
vec_pext (vector unsigned long long int, vector unsigned long long int);
```

Perform a vector parallel bit extract operation, as if implemented by the `vpextd` instruction.

```
vector unsigned char vec_stril (vector unsigned char);
vector signed char vec_stril (vector signed char);
vector unsigned short vec_stril (vector unsigned short);
vector signed short vec_stril (vector signed short);
```

Isolate the left-most non-zero elements of the incoming vector argument, replacing all elements to the right of the left-most zero element found within the argument with zero. The typical implementation uses the `vstribl` or `vstrihl` instruction on big-endian targets and uses the `vstribr` or `vstrihr` instruction on little-endian targets.

```
int vec_stril_p (vector unsigned char);
int vec_stril_p (vector signed char);
int short vec_stril_p (vector unsigned short);
int vec_stril_p (vector signed short);
```

Return a non-zero value if and only if the argument contains a zero element. The typical implementation uses the `vstribl.` or `vstrihl.` instruction on big-endian targets and uses the `vstribr.` or `vstrihr.` instruction on little-endian targets. Choose this built-in to check for presence of zero element if the same argument is also passed to `vec_stril`.

```
vector unsigned char vec_strir (vector unsigned char);
vector signed char vec_strir (vector signed char);
vector unsigned short vec_strir (vector unsigned short);
vector signed short vec_strir (vector signed short);
```

Isolate the right-most non-zero elements of the incoming vector argument, replacing all elements to the left of the right-most zero element found within the argument with zero. The typical implementation uses the `vstribr` or `vstrihr` instruction on big-endian targets and uses the `vstribl` or `vstrihl` instruction on little-endian targets.

```
int vec_strir_p (vector unsigned char);
int vec_strir_p (vector signed char);
```

```
int short vec_strir_p (vector unsigned short);
int vec_strir_p (vector signed short);
```

Return a non-zero value if and only if the argument contains a zero element. The typical implementation uses the `vstribr.` or `vstrihr.` instruction on big-endian targets and uses the `vstribl.` or `vstrihl.` instruction on little-endian targets. Choose this built-in to check for presence of zero element if the same argument is also passed to `vec_strir`.

```
vector unsigned char
vec_ternarylogic (vector unsigned char, vector unsigned char,
                  vector unsigned char, const unsigned int);
vector unsigned short
vec_ternarylogic (vector unsigned short, vector unsigned short,
                  vector unsigned short, const unsigned int);
vector unsigned int
vec_ternarylogic (vector unsigned int, vector unsigned int,
                  vector unsigned int, const unsigned int);
vector unsigned long long int
vec_ternarylogic (vector unsigned long long int, vector unsigned long long int,
                  vector unsigned long long int, const unsigned int);
vector unsigned __int128
vec_ternarylogic (vector unsigned __int128, vector unsigned __int128,
                  vector unsigned __int128, const unsigned int);
```

Perform a 128-bit vector evaluate operation, as if implemented by the `xxeval` instruction. The fourth argument must be a literal integer value between 0 and 255 inclusive.

```
vector unsigned char vec_genpcvm (vector unsigned char, const int);
vector unsigned short vec_genpcvm (vector unsigned short, const int);
vector unsigned int vec_genpcvm (vector unsigned int, const int);
vector unsigned int vec_genpcvm (vector unsigned long long int,
                                 const int);
```

Vector Integer Multiply/Divide/Modulo

```
vector signed int
vec_mulh (vector signed int a, vector signed int b);
vector unsigned int
vec_mulh (vector unsigned int a, vector unsigned int b);
```

For each integer value `i` from 0 to 3, do the following. The integer value in word element `i` of a is multiplied by the integer value in word element `i` of b. The high-order 32 bits of the 64-bit product are placed into word element `i` of the vector returned.

```
vector signed long long
vec_mulh (vector signed long long a, vector signed long long b);
vector unsigned long long
vec_mulh (vector unsigned long long a, vector unsigned long long b);
```

For each integer value `i` from 0 to 1, do the following. The integer value in doubleword element `i` of a is multiplied by the integer value in doubleword element `i` of b. The high-order 64 bits of the 128-bit product are placed into doubleword element `i` of the vector returned.

```
vector unsigned long long
vec_mul (vector unsigned long long a, vector unsigned long long b);
vector signed long long
vec_mul (vector signed long long a, vector signed long long b);
```

For each integer value `i` from 0 to 1, do the following. The integer value in doubleword element `i` of a is multiplied by the integer value in doubleword element `i` of b. The low-order 64 bits of the 128-bit product are placed into doubleword element `i` of the vector returned.

```
vector signed int
```

```
vec_div (vector signed int a, vector signed int b);
vector unsigned int
vec_div (vector unsigned int a, vector unsigned int b);
```

For each integer value `i` from 0 to 3, do the following. The integer in word element `i` of a is divided by the integer in word element `i` of b. The unique integer quotient is placed into the word element `i` of the vector returned. If an attempt is made to perform any of the divisions <anything> ÷ 0 then the quotient is undefined.

```
vector signed long long
vec_div (vector signed long long a, vector signed long long b);
vector unsigned long long
vec_div (vector unsigned long long a, vector unsigned long long b);
```

For each integer value `i` from 0 to 1, do the following. The integer in doubleword element `i` of a is divided by the integer in doubleword element `i` of b. The unique integer quotient is placed into the doubleword element `i` of the vector returned. If an attempt is made to perform any of the divisions 0x8000_0000_0000_0000 ÷ -1 or <anything> ÷ 0 then the quotient is undefined.

```
vector signed int
vec_dive (vector signed int a, vector signed int b);
vector unsigned int
vec_dive (vector unsigned int a, vector unsigned int b);
```

For each integer value `i` from 0 to 3, do the following. The integer in word element `i` of a is shifted left by 32 bits, then divided by the integer in word element `i` of b. The unique integer quotient is placed into the word element `i` of the vector returned. If the quotient cannot be represented in 32 bits, or if an attempt is made to perform any of the divisions <anything> ÷ 0 then the quotient is undefined.

```
vector signed long long
vec_dive (vector signed long long a, vector signed long long b);
vector unsigned long long
vec_dive (vector unsigned long long a, vector unsigned long long b);
```

For each integer value `i` from 0 to 1, do the following. The integer in doubleword element `i` of a is shifted left by 64 bits, then divided by the integer in doubleword element `i` of b. The unique integer quotient is placed into the doubleword element `i` of the vector returned. If the quotient cannot be represented in 64 bits, or if an attempt is made to perform <anything> ÷ 0 then the quotient is undefined.

```
vector signed int
vec_mod (vector signed int a, vector signed int b);
vector unsigned int
vec_mod (vector unsigned int a, vector unsigned int b);
```

For each integer value `i` from 0 to 3, do the following. The integer in word element `i` of a is divided by the integer in word element `i` of b. The unique integer remainder is placed into the word element `i` of the vector returned. If an attempt is made to perform any of the divisions 0x8000_0000 ÷ -1 or <anything> ÷ 0 then the remainder is undefined.

```
vector signed long long
vec_mod (vector signed long long a, vector signed long long b);
vector unsigned long long
vec_mod (vector unsigned long long a, vector unsigned long long b);
```

For each integer value `i` from 0 to 1, do the following. The integer in doubleword element `i` of a is divided by the integer in doubleword element `i` of b. The unique integer remainder

is placed into the doubleword element `i` of the vector returned. If an attempt is made to perform <anything> ÷ 0 then the remainder is undefined.

Generate PCV from specified Mask size, as if implemented by the `xxgenpcvbm`, `xxgenpcvhm`, `xxgenpcvwm` instructions, where immediate value is either 0, 1, 2 or 3.

```
vector unsigned __int128 vec_rl (vector unsigned __int128 A,
                                 vector unsigned __int128 B);
vector signed __int128 vec_rl (vector signed __int128 A,
                               vector unsigned __int128 B);
```

Result value: Each element of R is obtained by rotating the corresponding element of A left by the number of bits specified by the corresponding element of B.

```
vector unsigned __int128 vec_rlmi (vector unsigned __int128,
                                   vector unsigned __int128,
                                   vector unsigned __int128);
vector signed __int128 vec_rlmi (vector signed __int128,
                                 vector signed __int128,
                                 vector unsigned __int128);
```

Returns the result of rotating the first input and inserting it under mask into the second input. The first bit in the mask, the last bit in the mask are obtained from the two 7-bit fields bits [108:115] and bits [117:123] respectively of the second input. The shift is obtained from the third input in the 7-bit field [125:131] where all bits counted from zero at the left.

```
vector unsigned __int128 vec_rlnm (vector unsigned __int128,
                                   vector unsigned __int128,
                                   vector unsigned __int128);
vector signed __int128 vec_rlnm (vector signed __int128,
                                 vector unsigned __int128,
                                 vector unsigned __int128);
```

Returns the result of rotating the first input and ANDing it with a mask. The first bit in the mask and the last bit in the mask are obtained from the two 7-bit fields bits [117:123] and bits [125:131] respectively of the second input. The shift is obtained from the third input in the 7-bit field bits [125:131] where all bits counted from zero at the left.

```
vector unsigned __int128 vec_sl(vector unsigned __int128 A, vector unsigned __int128 B);
vector signed __int128 vec_sl(vector signed __int128 A, vector unsigned __int128 B);
```

Result value: Each element of R is obtained by shifting the corresponding element of A left by the number of bits specified by the corresponding element of B.

```
vector unsigned __int128 vec_sr(vector unsigned __int128 A, vector unsigned __int128 B);
vector signed __int128 vec_sr(vector signed __int128 A, vector unsigned __int128 B);
```

Result value: Each element of R is obtained by shifting the corresponding element of A right by the number of bits specified by the corresponding element of B.

```
vector unsigned __int128 vec_sra(vector unsigned __int128 A, vector unsigned __int128 B);
vector signed __int128 vec_sra(vector signed __int128 A, vector unsigned __int128 B);
```

Result value: Each element of R is obtained by arithmetic shifting the corresponding element of A right by the number of bits specified by the corresponding element of B.

```
vector unsigned __int128 vec_mule (vector unsigned long long,
                                   vector unsigned long long);
vector signed __int128 vec_mule (vector signed long long,
                                 vector signed long long);
```

Returns a vector containing a 128-bit integer result of multiplying the even doubleword elements of the two inputs.

```
vector unsigned __int128 vec_mulo (vector unsigned long long,
```

```
                                                    vector unsigned long long);
vector signed __int128 vec_mulo (vector signed long long,
                                                  vector signed long long);
```

Returns a vector containing a 128-bit integer result of multiplying the odd doubleword elements of the two inputs.

```
vector unsigned __int128 vec_div (vector unsigned __int128,
                                                    vector unsigned __int128);
vector signed __int128 vec_div (vector signed __int128,
                                                  vector signed __int128);
```

Returns the result of dividing the first operand by the second operand. An attempt to divide any value by zero or to divide the most negative signed 128-bit integer by negative one results in an undefined value.

```
vector unsigned __int128 vec_dive (vector unsigned __int128,
                                                    vector unsigned __int128);
vector signed __int128 vec_dive (vector signed __int128,
                                                  vector signed __int128);
```

The result is produced by shifting the first input left by 128 bits and dividing by the second. If an attempt is made to divide by zero or the result is larger than 128 bits, the result is undefined.

```
vector unsigned __int128 vec_mod (vector unsigned __int128,
                                                    vector unsigned __int128);
vector signed __int128 vec_mod (vector signed __int128,
                                                  vector signed __int128);
```

The result is the modulo result of dividing the first input by the second input.

The following builtins perform 128-bit vector comparisons. The `vec_all_xx`, `vec_any_xx`, and `vec_cmpxx`, where `xx` is one of the operations `eq, ne, gt, lt, ge, le` perform pairwise comparisons between the elements at the same positions within their two vector arguments. The `vec_all_xx`function returns a non-zero value if and only if all pairwise comparisons are true. The `vec_any_xx` function returns a non-zero value if and only if at least one pairwise comparison is true. The `vec_cmpxx`function returns a vector of the same type as its two arguments, within which each element consists of all ones to denote that specified logical comparison of the corresponding elements was true. Otherwise, the element of the returned vector contains all zeros.

```
vector bool __int128 vec_cmpeq (vector signed __int128, vector signed __int128);
vector bool __int128 vec_cmpeq (vector unsigned __int128, vector unsigned __int128);
vector bool __int128 vec_cmpne (vector signed __int128, vector signed __int128);
vector bool __int128 vec_cmpne (vector unsigned __int128, vector unsigned __int128);
vector bool __int128 vec_cmpgt (vector signed __int128, vector signed __int128);
vector bool __int128 vec_cmpgt (vector unsigned __int128, vector unsigned __int128);
vector bool __int128 vec_cmplt (vector signed __int128, vector signed __int128);
vector bool __int128 vec_cmplt (vector unsigned __int128, vector unsigned __int128);
vector bool __int128 vec_cmpge (vector signed __int128, vector signed __int128);
vector bool __int128 vec_cmpge (vector unsigned __int128, vector unsigned __int128);
vector bool __int128 vec_cmple (vector signed __int128, vector signed __int128);
vector bool __int128 vec_cmple (vector unsigned __int128, vector unsigned __int128);

int vec_all_eq (vector signed __int128, vector signed __int128);
int vec_all_eq (vector unsigned __int128, vector unsigned __int128);
int vec_all_ne (vector signed __int128, vector signed __int128);
int vec_all_ne (vector unsigned __int128, vector unsigned __int128);
int vec_all_gt (vector signed __int128, vector signed __int128);
```

```
int vec_all_gt (vector unsigned __int128, vector unsigned __int128);
int vec_all_lt (vector signed __int128, vector signed __int128);
int vec_all_lt (vector unsigned __int128, vector unsigned __int128);
int vec_all_ge (vector signed __int128, vector signed __int128);
int vec_all_ge (vector unsigned __int128, vector unsigned __int128);
int vec_all_le (vector signed __int128, vector signed __int128);
int vec_all_le (vector unsigned __int128, vector unsigned __int128);

int vec_any_eq (vector signed __int128, vector signed __int128);
int vec_any_eq (vector unsigned __int128, vector unsigned __int128);
int vec_any_ne (vector signed __int128, vector signed __int128);
int vec_any_ne (vector unsigned __int128, vector unsigned __int128);
int vec_any_gt (vector signed __int128, vector signed __int128);
int vec_any_gt (vector unsigned __int128, vector unsigned __int128);
int vec_any_lt (vector signed __int128, vector signed __int128);
int vec_any_lt (vector unsigned __int128, vector unsigned __int128);
int vec_any_ge (vector signed __int128, vector signed __int128);
int vec_any_ge (vector unsigned __int128, vector unsigned __int128);
int vec_any_le (vector signed __int128, vector signed __int128);
int vec_any_le (vector unsigned __int128, vector unsigned __int128);
```

6.60.24 PowerPC Hardware Transactional Memory Built-in Functions

GCC provides two interfaces for accessing the Hardware Transactional Memory (HTM) instructions available on some of the PowerPC family of processors (eg, POWER8). The two interfaces come in a low level interface, consisting of built-in functions specific to PowerPC and a higher level interface consisting of inline functions that are common between PowerPC and S/390.

6.60.24.1 PowerPC HTM Low Level Built-in Functions

The following low level built-in functions are available with `-mhtm` or `-mcpu=CPU` where CPU is 'power8' or later. They all generate the machine instruction that is part of the name.

The HTM builtins (with the exception of `__builtin_tbegin`) return the full 4-bit condition register value set by their associated hardware instruction. The header file `htmintrin.h` defines some macros that can be used to decipher the return value. The `__builtin_tbegin` builtin returns a simple `true` or `false` value depending on whether a transaction was successfully started or not. The arguments of the builtins match exactly the type and order of the associated hardware instruction's operands, except for the `__builtin_tcheck` builtin, which does not take any input arguments. Refer to the ISA manual for a description of each instruction's operands.

```
unsigned int __builtin_tbegin (unsigned int);
unsigned int __builtin_tend (unsigned int);

unsigned int __builtin_tabort (unsigned int);
unsigned int __builtin_tabortdc (unsigned int, unsigned int, unsigned int);
unsigned int __builtin_tabortdci (unsigned int, unsigned int, int);
unsigned int __builtin_tabortwc (unsigned int, unsigned int, unsigned int);
unsigned int __builtin_tabortwci (unsigned int, unsigned int, int);

unsigned int __builtin_tcheck (void);
unsigned int __builtin_treclaim (unsigned int);
unsigned int __builtin_trechkpt (void);
unsigned int __builtin_tsr (unsigned int);
```

In addition to the above HTM built-ins, we have added built-ins for some common extended mnemonics of the HTM instructions:

```
unsigned int __builtin_tendall (void);
unsigned int __builtin_tresume (void);
unsigned int __builtin_tsuspend (void);
```

Note that the semantics of the above HTM builtins are required to mimic the locking semantics used for critical sections. Builtins that are used to create a new transaction or restart a suspended transaction must have lock acquisition like semantics while those builtins that end or suspend a transaction must have lock release like semantics. Specifically, this must mimic lock semantics as specified by C++11, for example: Lock acquisition is as-if an execution of __atomic_exchange_n(&globallock,1,__ATOMIC_ACQUIRE) that returns 0, and lock release is as-if an execution of __atomic_store(&globallock,0,__ATOMIC_RELEASE), with globallock being an implicit implementation-defined lock used for all transactions. The HTM instructions associated with with the builtins inherently provide the correct acquisition and release hardware barriers required. However, the compiler must also be prohibited from moving loads and stores across the builtins in a way that would violate their semantics. This has been accomplished by adding memory barriers to the associated HTM instructions (which is a conservative approach to provide acquire and release semantics). Earlier versions of the compiler did not treat the HTM instructions as memory barriers. A `__TM_FENCE__` macro has been added, which can be used to determine whether the current compiler treats HTM instructions as memory barriers or not. This allows the user to explicitly add memory barriers to their code when using an older version of the compiler.

The following set of built-in functions are available to gain access to the HTM specific special purpose registers.

```
unsigned long __builtin_get_texasr (void);
unsigned long __builtin_get_texasru (void);
unsigned long __builtin_get_tfhar (void);
unsigned long __builtin_get_tfiar (void);

void __builtin_set_texasr (unsigned long);
void __builtin_set_texasru (unsigned long);
void __builtin_set_tfhar (unsigned long);
void __builtin_set_tfiar (unsigned long);
```

Example usage of these low level built-in functions may look like:

```
#include <htmintrin.h>

int num_retries = 10;

while (1)
  {
    if (__builtin_tbegin (0))
      {
        /* Transaction State Initiated.  */
        if (is_locked (lock))
          __builtin_tabort (0);
        ... transaction code...
        __builtin_tend (0);
        break;
      }
```

```
    else
      {
        /* Transaction State Failed.  Use locks if the transaction
           failure is "persistent" or we've tried too many times.  */
        if (num_retries-- <= 0
            || _TEXASRU_FAILURE_PERSISTENT (__builtin_get_texasru ()))
          {
            acquire_lock (lock);
            ... non transactional fallback path...
            release_lock (lock);
            break;
          }
      }
  }
```

One final built-in function has been added that returns the value of the 2-bit Transaction State field of the Machine Status Register (MSR) as stored in `CR0`.

```
unsigned long __builtin_ttest (void)
```

This built-in can be used to determine the current transaction state using the following code example:

```
#include <htmintrin.h>

unsigned char tx_state = _HTM_STATE (__builtin_ttest ());

if (tx_state == _HTM_TRANSACTIONAL)
  {
    /* Code to use in transactional state.  */
  }
else if (tx_state == _HTM_NONTRANSACTIONAL)
  {
    /* Code to use in non-transactional state.  */
  }
else if (tx_state == _HTM_SUSPENDED)
  {
    /* Code to use in transaction suspended state.  */
  }
```

6.60.24.2 PowerPC HTM High Level Inline Functions

The following high level HTM interface is made available by including `<htmxlintrin.h>` and using `-mhtm` or `-mcpu=CPU` where CPU is 'power8' or later. This interface is common between PowerPC and S/390, allowing users to write one HTM source implementation that can be compiled and executed on either system.

```
long __TM_simple_begin (void);
long __TM_begin (void* const TM_buff);
long __TM_end (void);
void __TM_abort (void);
void __TM_named_abort (unsigned char const code);
void __TM_resume (void);
void __TM_suspend (void);

long __TM_is_user_abort (void* const TM_buff);
long __TM_is_named_user_abort (void* const TM_buff, unsigned char *code);
long __TM_is_illegal (void* const TM_buff);
long __TM_is_footprint_exceeded (void* const TM_buff);
long __TM_nesting_depth (void* const TM_buff);
long __TM_is_nested_too_deep(void* const TM_buff);
```

```
long __TM_is_conflict(void* const TM_buff);
long __TM_is_failure_persistent(void* const TM_buff);
long __TM_failure_address(void* const TM_buff);
long long __TM_failure_code(void* const TM_buff);
```

Using these common set of HTM inline functions, we can create a more portable version of the HTM example in the previous section that will work on either PowerPC or S/390:

```
#include <htmxlintrin.h>

int num_retries = 10;
TM_buff_type TM_buff;

while (1)
  {
    if (__TM_begin (TM_buff) == _HTM_TBEGIN_STARTED)
      {
        /* Transaction State Initiated.  */
        if (is_locked (lock))
          __TM_abort ();
        ... transaction code...
        __TM_end ();
        break;
      }
    else
      {
        /* Transaction State Failed.  Use locks if the transaction
           failure is "persistent" or we've tried too many times.  */
        if (num_retries-- <= 0
            || __TM_is_failure_persistent (TM_buff))
          {
            acquire_lock (lock);
            ... non transactional fallback path...
            release_lock (lock);
            break;
          }
      }
  }
```

6.60.25 PowerPC Atomic Memory Operation Functions

ISA 3.0 of the PowerPC added new atomic memory operation (amo) instructions. GCC provides support for these instructions in 64-bit environments. All of the functions are declared in the include file `amo.h`.

The functions supported are:

```
#include <amo.h>

uint32_t amo_lwat_add (uint32_t *, uint32_t);
uint32_t amo_lwat_xor (uint32_t *, uint32_t);
uint32_t amo_lwat_ior (uint32_t *, uint32_t);
uint32_t amo_lwat_and (uint32_t *, uint32_t);
uint32_t amo_lwat_umax (uint32_t *, uint32_t);
uint32_t amo_lwat_umin (uint32_t *, uint32_t);
uint32_t amo_lwat_swap (uint32_t *, uint32_t);

int32_t amo_lwat_sadd (int32_t *, int32_t);
int32_t amo_lwat_smax (int32_t *, int32_t);
int32_t amo_lwat_smin (int32_t *, int32_t);
```

```
int32_t amo_lwat_sswap (int32_t *, int32_t);

uint64_t amo_ldat_add (uint64_t *, uint64_t);
uint64_t amo_ldat_xor (uint64_t *, uint64_t);
uint64_t amo_ldat_ior (uint64_t *, uint64_t);
uint64_t amo_ldat_and (uint64_t *, uint64_t);
uint64_t amo_ldat_umax (uint64_t *, uint64_t);
uint64_t amo_ldat_umin (uint64_t *, uint64_t);
uint64_t amo_ldat_swap (uint64_t *, uint64_t);

int64_t amo_ldat_sadd (int64_t *, int64_t);
int64_t amo_ldat_smax (int64_t *, int64_t);
int64_t amo_ldat_smin (int64_t *, int64_t);
int64_t amo_ldat_sswap (int64_t *, int64_t);

void amo_stwat_add (uint32_t *, uint32_t);
void amo_stwat_xor (uint32_t *, uint32_t);
void amo_stwat_ior (uint32_t *, uint32_t);
void amo_stwat_and (uint32_t *, uint32_t);
void amo_stwat_umax (uint32_t *, uint32_t);
void amo_stwat_umin (uint32_t *, uint32_t);

void amo_stwat_sadd (int32_t *, int32_t);
void amo_stwat_smax (int32_t *, int32_t);
void amo_stwat_smin (int32_t *, int32_t);

void amo_stdat_add (uint64_t *, uint64_t);
void amo_stdat_xor (uint64_t *, uint64_t);
void amo_stdat_ior (uint64_t *, uint64_t);
void amo_stdat_and (uint64_t *, uint64_t);
void amo_stdat_umax (uint64_t *, uint64_t);
void amo_stdat_umin (uint64_t *, uint64_t);

void amo_stdat_sadd (int64_t *, int64_t);
void amo_stdat_smax (int64_t *, int64_t);
void amo_stdat_smin (int64_t *, int64_t);
```

6.60.26 PowerPC Matrix-Multiply Assist Built-in Functions

ISA 3.1 of the PowerPC added new Matrix-Multiply Assist (MMA) instructions. GCC provides support for these instructions through the following built-in functions which are enabled with the `-mmma` option. The vec_t type below is defined to be a normal vector unsigned char type. The uint2, uint4 and uint8 parameters are 2-bit, 4-bit and 8-bit unsigned integer constants respectively. The compiler will verify that they are constants and that their values are within range.

The built-in functions supported are:

```
void __builtin_mma_xvi4ger8 (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi8ger4 (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi16ger2 (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi16ger2s (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf16ger2 (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvbf16ger2 (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf32ger (__vector_quad *, vec_t, vec_t);

void __builtin_mma_xvi4ger8pp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi8ger4pp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi8ger4spp(__vector_quad *, vec_t, vec_t);
```

```
void __builtin_mma_xvi16ger2pp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi16ger2spp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf16ger2pp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf16ger2pn (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf16ger2np (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf16ger2nn (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvbf16ger2pp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvbf16ger2pn (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvbf16ger2np (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvbf16ger2nn (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf32gerpp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf32gerpn (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf32gernp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf32gernn (__vector_quad *, vec_t, vec_t);

void __builtin_mma_pmxvi4ger8 (__vector_quad *, vec_t, vec_t, uint4, uint4, uint8);
void __builtin_mma_pmxvi4ger8pp (__vector_quad *, vec_t, vec_t, uint4, uint4, uint8);

void __builtin_mma_pmxvi8ger4 (__vector_quad *, vec_t, vec_t, uint4, uint4, uint4);
void __builtin_mma_pmxvi8ger4pp (__vector_quad *, vec_t, vec_t, uint4, uint4, uint4);
void __builtin_mma_pmxvi8ger4spp(__vector_quad *, vec_t, vec_t, uint4, uint4, uint4);

void __builtin_mma_pmxvi16ger2 (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvi16ger2s (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvf16ger2 (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvbf16ger2 (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);

void __builtin_mma_pmxvi16ger2pp (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvi16ger2spp (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvf16ger2pp (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvf16ger2pn (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvf16ger2np (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvf16ger2nn (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvbf16ger2pp (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvbf16ger2pn (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvbf16ger2np (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvbf16ger2nn (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);

void __builtin_mma_pmxvf32ger (__vector_quad *, vec_t, vec_t, uint4, uint4);
void __builtin_mma_pmxvf32gerpp (__vector_quad *, vec_t, vec_t, uint4, uint4);
void __builtin_mma_pmxvf32gerpn (__vector_quad *, vec_t, vec_t, uint4, uint4);
void __builtin_mma_pmxvf32gernp (__vector_quad *, vec_t, vec_t, uint4, uint4);
void __builtin_mma_pmxvf32gernn (__vector_quad *, vec_t, vec_t, uint4, uint4);

void __builtin_mma_xvf64ger (__vector_quad *, __vector_pair, vec_t);
void __builtin_mma_xvf64gerpp (__vector_quad *, __vector_pair, vec_t);
void __builtin_mma_xvf64gerpn (__vector_quad *, __vector_pair, vec_t);
void __builtin_mma_xvf64gernp (__vector_quad *, __vector_pair, vec_t);
void __builtin_mma_xvf64gernn (__vector_quad *, __vector_pair, vec_t);

void __builtin_mma_pmxvf64ger (__vector_quad *, __vector_pair, vec_t, uint4, uint2);
void __builtin_mma_pmxvf64gerpp (__vector_quad *, __vector_pair, vec_t, uint4, uint2);
void __builtin_mma_pmxvf64gerpn (__vector_quad *, __vector_pair, vec_t, uint4, uint2);
void __builtin_mma_pmxvf64gernp (__vector_quad *, __vector_pair, vec_t, uint4, uint2);
void __builtin_mma_pmxvf64gernn (__vector_quad *, __vector_pair, vec_t, uint4, uint2);

void __builtin_mma_xxmtacc (__vector_quad *);
void __builtin_mma_xxmfacc (__vector_quad *);
```

```
void __builtin_mma_xxsetaccz (__vector_quad *);

void __builtin_mma_build_acc (__vector_quad *, vec_t, vec_t, vec_t, vec_t);
void __builtin_mma_disassemble_acc (void *, __vector_quad *);

void __builtin_vsx_build_pair (__vector_pair *, vec_t, vec_t);
void __builtin_vsx_disassemble_pair (void *, __vector_pair *);

vec_t __builtin_vsx_xvcvspbf16 (vec_t);
vec_t __builtin_vsx_xvcvbf16spn (vec_t);

__vector_pair __builtin_vsx_lxvp (size_t, __vector_pair *);
void __builtin_vsx_stxvp (__vector_pair, size_t, __vector_pair *);
```

6.60.27 PRU Built-in Functions

GCC provides a couple of special builtin functions to aid in utilizing special PRU instructions.

The built-in functions supported are:

`void __delay_cycles (constant long long cycles)` [Built-in Function]
: This inserts an instruction sequence that takes exactly *cycles* cycles (between 0 and 0xffffffff) to complete. The inserted sequence may use jumps, loops, or no-ops, and does not interfere with any other instructions. Note that *cycles* must be a compile-time constant integer - that is, you must pass a number, not a variable that may be optimized to a constant later. The number of cycles delayed by this builtin is exact.

`void __halt (void)` [Built-in Function]
: This inserts a HALT instruction to stop processor execution.

`unsigned int __lmbd (unsigned int wordval, unsigned int bitval)` [Built-in Function]
: This inserts LMBD instruction to calculate the left-most bit with value *bitval* in value *wordval*. Only the least significant bit of *bitval* is taken into account.

6.60.28 RISC-V Built-in Functions

These built-in functions are available for the RISC-V family of processors.

`void * __builtin_thread_pointer (void)` [Built-in Function]
: Returns the value that is currently set in the 'tp' register.

`void __builtin_riscv_pause (void)` [Built-in Function]
: Generates the `pause` (hint) machine instruction. This implies the Xgnuzihintpausestate extension, which redefines the `pause` instruction to change architectural state.

6.60.29 RISC-V Vector Intrinsics

GCC supports vector intrinsics as specified in version 0.11 of the RISC-V vector intrinsic specification, which is available at the following link: `https://github.com/riscv-non-isa/rvv-intrinsic-doc/tree/v0.11.x`. All of these functions are declared in the include file `riscv_vector.h`.

6.60.30 RX Built-in Functions

GCC supports some of the RX instructions which cannot be expressed in the C programming language via the use of built-in functions. The following functions are supported:

`void __builtin_rx_brk (void)` [Built-in Function]
Generates the `brk` machine instruction.

`void __builtin_rx_clrpsw (int)` [Built-in Function]
Generates the `clrpsw` machine instruction to clear the specified bit in the processor status word.

`void __builtin_rx_int (int)` [Built-in Function]
Generates the `int` machine instruction to generate an interrupt with the specified value.

`void __builtin_rx_machi (int, int)` [Built-in Function]
Generates the `machi` machine instruction to add the result of multiplying the top 16 bits of the two arguments into the accumulator.

`void __builtin_rx_maclo (int, int)` [Built-in Function]
Generates the `maclo` machine instruction to add the result of multiplying the bottom 16 bits of the two arguments into the accumulator.

`void __builtin_rx_mulhi (int, int)` [Built-in Function]
Generates the `mulhi` machine instruction to place the result of multiplying the top 16 bits of the two arguments into the accumulator.

`void __builtin_rx_mullo (int, int)` [Built-in Function]
Generates the `mullo` machine instruction to place the result of multiplying the bottom 16 bits of the two arguments into the accumulator.

`int __builtin_rx_mvfachi (void)` [Built-in Function]
Generates the `mvfachi` machine instruction to read the top 32 bits of the accumulator.

`int __builtin_rx_mvfacmi (void)` [Built-in Function]
Generates the `mvfacmi` machine instruction to read the middle 32 bits of the accumulator.

`int __builtin_rx_mvfc (int)` [Built-in Function]
Generates the `mvfc` machine instruction which reads the control register specified in its argument and returns its value.

`void __builtin_rx_mvtachi (int)` [Built-in Function]
Generates the `mvtachi` machine instruction to set the top 32 bits of the accumulator.

`void __builtin_rx_mvtaclo (int)` [Built-in Function]
Generates the `mvtaclo` machine instruction to set the bottom 32 bits of the accumulator.

`void __builtin_rx_mvtc (int reg, int val)` [Built-in Function]
Generates the `mvtc` machine instruction which sets control register number `reg` to `val`.

`void __builtin_rx_mvtipl (int)` [Built-in Function]
Generates the `mvtipl` machine instruction set the interrupt priority level.

`void __builtin_rx_racw (int)` [Built-in Function]
Generates the `racw` machine instruction to round the accumulator according to the specified mode.

`int __builtin_rx_revw (int)` [Built-in Function]
Generates the `revw` machine instruction which swaps the bytes in the argument so that bits 0–7 now occupy bits 8–15 and vice versa, and also bits 16–23 occupy bits 24–31 and vice versa.

`void __builtin_rx_rmpa (void)` [Built-in Function]
Generates the `rmpa` machine instruction which initiates a repeated multiply and accumulate sequence.

`void __builtin_rx_round (float)` [Built-in Function]
Generates the `round` machine instruction which returns the floating-point argument rounded according to the current rounding mode set in the floating-point status word register.

`int __builtin_rx_sat (int)` [Built-in Function]
Generates the `sat` machine instruction which returns the saturated value of the argument.

`void __builtin_rx_setpsw (int)` [Built-in Function]
Generates the `setpsw` machine instruction to set the specified bit in the processor status word.

`void __builtin_rx_wait (void)` [Built-in Function]
Generates the `wait` machine instruction.

6.60.31 S/390 System z Built-in Functions

`int __builtin_tbegin (void*)` [Built-in Function]
Generates the `tbegin` machine instruction starting a non-constrained hardware transaction. If the parameter is non-NULL the memory area is used to store the transaction diagnostic buffer and will be passed as first operand to `tbegin`. This buffer can be defined using the `struct __htm_tdb` C struct defined in `htmintrin.h` and must reside on a double-word boundary. The second tbegin operand is set to `0xff0c`. This enables save/restore of all GPRs and disables aborts for FPR and AR manipulations inside the transaction body. The condition code set by the tbegin instruction is returned as integer value. The tbegin instruction by definition overwrites the content of all FPRs. The compiler will generate code which saves and restores the FPRs. For soft-float code it is recommended to used the `*_nofloat` variant. In order to prevent a TDB from being written it is required to pass a constant zero value as parameter. Passing a zero value through a variable is not sufficient. Although modifications of access registers inside the transaction will not trigger an transaction abort it is not supported to actually modify them. Access registers do not get saved when entering a transaction. They will have undefined state when reaching the abort code.

Macros for the possible return codes of tbegin are defined in the `htmintrin.h` header file:

`_HTM_TBEGIN_STARTED` [Macro]
`tbegin` has been executed as part of normal processing. The transaction body is supposed to be executed.

`_HTM_TBEGIN_INDETERMINATE` [Macro]
The transaction was aborted due to an indeterminate condition which might be persistent.

`_HTM_TBEGIN_TRANSIENT` [Macro]
The transaction aborted due to a transient failure. The transaction should be re-executed in that case.

`_HTM_TBEGIN_PERSISTENT` [Macro]
The transaction aborted due to a persistent failure. Re-execution under same circumstances will not be productive.

`_HTM_FIRST_USER_ABORT_CODE` [Macro]
The `_HTM_FIRST_USER_ABORT_CODE` defined in `htmintrin.h` specifies the first abort code which can be used for `__builtin_tabort`. Values below this threshold are reserved for machine use.

`struct __htm_tdb` [Data type]
The `struct __htm_tdb` defined in `htmintrin.h` describes the structure of the transaction diagnostic block as specified in the Principles of Operation manual chapter 5-91.

`int __builtin_tbegin_nofloat (void*)` [Built-in Function]
Same as `__builtin_tbegin` but without FPR saves and restores. Using this variant in code making use of FPRs will leave the FPRs in undefined state when entering the transaction abort handler code.

`int __builtin_tbegin_retry (void*, int)` [Built-in Function]
In addition to `__builtin_tbegin` a loop for transient failures is generated. If tbegin returns a condition code of 2 the transaction will be retried as often as specified in the second argument. The perform processor assist instruction is used to tell the CPU about the number of fails so far.

`int __builtin_tbegin_retry_nofloat (void*, int)` [Built-in Function]
Same as `__builtin_tbegin_retry` but without FPR saves and restores. Using this variant in code making use of FPRs will leave the FPRs in undefined state when entering the transaction abort handler code.

`void __builtin_tbeginc (void)` [Built-in Function]
Generates the `tbeginc` machine instruction starting a constrained hardware transaction. The second operand is set to `0xff08`.

`int __builtin_tend (void)` [Built-in Function]
Generates the `tend` machine instruction finishing a transaction and making the changes visible to other threads. The condition code generated by tend is returned as integer value.

`void __builtin_tabort (int)` [Built-in Function]
Generates the `tabort` machine instruction with the specified abort code. Abort codes from 0 through 255 are reserved and will result in an error message.

`void __builtin_tx_assist (int)` [Built-in Function]
Generates the `ppa rX,rY,1` machine instruction. Where the integer parameter is loaded into rX and a value of zero is loaded into rY. The integer parameter specifies the number of times the transaction repeatedly aborted.

`int __builtin_tx_nesting_depth (void)` [Built-in Function]
Generates the `etnd` machine instruction. The current nesting depth is returned as integer value. For a nesting depth of 0 the code is not executed as part of an transaction.

`void __builtin_non_tx_store (uint64_t *, uint64_t)` [Built-in Function]
Generates the `ntstg` machine instruction. The second argument is written to the first arguments location. The store operation will not be rolled-back in case of an transaction abort.

6.60.32 SH Built-in Functions

The following built-in functions are supported on the SH1, SH2, SH3 and SH4 families of processors:

`void __builtin_set_thread_pointer (void *ptr)` [Built-in Function]
Sets the '`GBR`' register to the specified value *ptr*. This is usually used by system code that manages threads and execution contexts. The compiler normally does not generate code that modifies the contents of '`GBR`' and thus the value is preserved across function calls. Changing the '`GBR`' value in user code must be done with caution, since the compiler might use '`GBR`' in order to access thread local variables.

`void * __builtin_thread_pointer (void)` [Built-in Function]
Returns the value that is currently set in the '`GBR`' register. Memory loads and stores that use the thread pointer as a base address are turned into '`GBR`' based displacement loads and stores, if possible. For example:

```
struct my_tcb
{
   int a, b, c, d, e;
};

int get_tcb_value (void)
{
  // Generate 'mov.l @(8,gbr),r0' instruction
  return ((my_tcb*)__builtin_thread_pointer ())->c;
}
```

`unsigned int __builtin_sh_get_fpscr (void)` [Built-in Function]
Returns the value that is currently set in the '`FPSCR`' register.

`void __builtin_sh_set_fpscr (unsigned int val)` [Built-in Function]
Sets the '`FPSCR`' register to the specified value *val*, while preserving the current values of the FR, SZ and PR bits.

6.60.33 SPARC VIS Built-in Functions

GCC supports SIMD operations on the SPARC using both the generic vector extensions (see Section 6.52 [Vector Extensions], page 726) as well as built-in functions for the SPARC Visual Instruction Set (VIS). When you use the `-mvis` switch, the VIS extension is exposed as the following built-in functions:

```
typedef int v1si __attribute__ ((vector_size (4)));
typedef int v2si __attribute__ ((vector_size (8)));
typedef short v4hi __attribute__ ((vector_size (8)));
typedef short v2hi __attribute__ ((vector_size (4)));
typedef unsigned char v8qi __attribute__ ((vector_size (8)));
typedef unsigned char v4qi __attribute__ ((vector_size (4)));

void __builtin_vis_write_gsr (int64_t);
int64_t __builtin_vis_read_gsr (void);

void * __builtin_vis_alignaddr (void *, long);
void * __builtin_vis_alignaddrl (void *, long);
int64_t __builtin_vis_faligndatadi (int64_t, int64_t);
v2si __builtin_vis_faligndatav2si (v2si, v2si);
v4hi __builtin_vis_faligndatav4hi (v4si, v4si);
v8qi __builtin_vis_faligndatav8qi (v8qi, v8qi);

v4hi __builtin_vis_fexpand (v4qi);

v4hi __builtin_vis_fmul8x16 (v4qi, v4hi);
v4hi __builtin_vis_fmul8x16au (v4qi, v2hi);
v4hi __builtin_vis_fmul8x16al (v4qi, v2hi);
v4hi __builtin_vis_fmul8sux16 (v8qi, v4hi);
v4hi __builtin_vis_fmul8ulx16 (v8qi, v4hi);
v2si __builtin_vis_fmuld8sux16 (v4qi, v2hi);
v2si __builtin_vis_fmuld8ulx16 (v4qi, v2hi);

v4qi __builtin_vis_fpack16 (v4hi);
v8qi __builtin_vis_fpack32 (v2si, v8qi);
v2hi __builtin_vis_fpackfix (v2si);
v8qi __builtin_vis_fpmerge (v4qi, v4qi);

int64_t __builtin_vis_pdist (v8qi, v8qi, int64_t);

long __builtin_vis_edge8 (void *, void *);
long __builtin_vis_edge8l (void *, void *);
long __builtin_vis_edge16 (void *, void *);
long __builtin_vis_edge16l (void *, void *);
long __builtin_vis_edge32 (void *, void *);
long __builtin_vis_edge32l (void *, void *);

long __builtin_vis_fcmple16 (v4hi, v4hi);
long __builtin_vis_fcmple32 (v2si, v2si);
long __builtin_vis_fcmpne16 (v4hi, v4hi);
long __builtin_vis_fcmpne32 (v2si, v2si);
long __builtin_vis_fcmpgt16 (v4hi, v4hi);
long __builtin_vis_fcmpgt32 (v2si, v2si);
long __builtin_vis_fcmpeq16 (v4hi, v4hi);
long __builtin_vis_fcmpeq32 (v2si, v2si);

v4hi __builtin_vis_fpadd16 (v4hi, v4hi);
v2hi __builtin_vis_fpadd16s (v2hi, v2hi);
```

```
v2si __builtin_vis_fpadd32 (v2si, v2si);
v1si __builtin_vis_fpadd32s (v1si, v1si);
v4hi __builtin_vis_fpsub16 (v4hi, v4hi);
v2hi __builtin_vis_fpsub16s (v2hi, v2hi);
v2si __builtin_vis_fpsub32 (v2si, v2si);
v1si __builtin_vis_fpsub32s (v1si, v1si);

long __builtin_vis_array8 (long, long);
long __builtin_vis_array16 (long, long);
long __builtin_vis_array32 (long, long);
```

When you use the `-mvis2` switch, the VIS version 2.0 built-in functions also become available:

```
long __builtin_vis_bmask (long, long);
int64_t __builtin_vis_bshuffledi (int64_t, int64_t);
v2si __builtin_vis_bshufflev2si (v2si, v2si);
v4hi __builtin_vis_bshufflev2si (v4hi, v4hi);
v8qi __builtin_vis_bshufflev2si (v8qi, v8qi);

long __builtin_vis_edge8n (void *, void *);
long __builtin_vis_edge8ln (void *, void *);
long __builtin_vis_edge16n (void *, void *);
long __builtin_vis_edge16ln (void *, void *);
long __builtin_vis_edge32n (void *, void *);
long __builtin_vis_edge32ln (void *, void *);
```

When you use the `-mvis3` switch, the VIS version 3.0 built-in functions also become available:

```
void __builtin_vis_cmask8 (long);
void __builtin_vis_cmask16 (long);
void __builtin_vis_cmask32 (long);

v4hi __builtin_vis_fchksm16 (v4hi, v4hi);

v4hi __builtin_vis_fsll16 (v4hi, v4hi);
v4hi __builtin_vis_fslas16 (v4hi, v4hi);
v4hi __builtin_vis_fsrl16 (v4hi, v4hi);
v4hi __builtin_vis_fsra16 (v4hi, v4hi);
v2si __builtin_vis_fsll16 (v2si, v2si);
v2si __builtin_vis_fslas16 (v2si, v2si);
v2si __builtin_vis_fsrl16 (v2si, v2si);
v2si __builtin_vis_fsra16 (v2si, v2si);

long __builtin_vis_pdistn (v8qi, v8qi);

v4hi __builtin_vis_fmean16 (v4hi, v4hi);

int64_t __builtin_vis_fpadd64 (int64_t, int64_t);
int64_t __builtin_vis_fpsub64 (int64_t, int64_t);

v4hi __builtin_vis_fpadds16 (v4hi, v4hi);
v2hi __builtin_vis_fpadds16s (v2hi, v2hi);
v4hi __builtin_vis_fpsubs16 (v4hi, v4hi);
v2hi __builtin_vis_fpsubs16s (v2hi, v2hi);
v2si __builtin_vis_fpadds32 (v2si, v2si);
v1si __builtin_vis_fpadds32s (v1si, v1si);
v2si __builtin_vis_fpsubs32 (v2si, v2si);
v1si __builtin_vis_fpsubs32s (v1si, v1si);
```

```
long __builtin_vis_fucmple8 (v8qi, v8qi);
long __builtin_vis_fucmpne8 (v8qi, v8qi);
long __builtin_vis_fucmpgt8 (v8qi, v8qi);
long __builtin_vis_fucmpeq8 (v8qi, v8qi);

float __builtin_vis_fhadds (float, float);
double __builtin_vis_fhaddd (double, double);
float __builtin_vis_fhsubs (float, float);
double __builtin_vis_fhsubd (double, double);
float __builtin_vis_fnhadds (float, float);
double __builtin_vis_fnhaddd (double, double);

int64_t __builtin_vis_umulxhi (int64_t, int64_t);
int64_t __builtin_vis_xmulx (int64_t, int64_t);
int64_t __builtin_vis_xmulxhi (int64_t, int64_t);
```

When you use the `-mvis4` switch, the VIS version 4.0 built-in functions also become available:

```
v8qi __builtin_vis_fpadd8 (v8qi, v8qi);
v8qi __builtin_vis_fpadds8 (v8qi, v8qi);
v8qi __builtin_vis_fpaddus8 (v8qi, v8qi);
v4hi __builtin_vis_fpaddus16 (v4hi, v4hi);

v8qi __builtin_vis_fpsub8 (v8qi, v8qi);
v8qi __builtin_vis_fpsubs8 (v8qi, v8qi);
v8qi __builtin_vis_fpsubus8 (v8qi, v8qi);
v4hi __builtin_vis_fpsubus16 (v4hi, v4hi);

long __builtin_vis_fpcmple8 (v8qi, v8qi);
long __builtin_vis_fpcmpgt8 (v8qi, v8qi);
long __builtin_vis_fpcmpule16 (v4hi, v4hi);
long __builtin_vis_fpcmpugt16 (v4hi, v4hi);
long __builtin_vis_fpcmpule32 (v2si, v2si);
long __builtin_vis_fpcmpugt32 (v2si, v2si);

v8qi __builtin_vis_fpmax8 (v8qi, v8qi);
v4hi __builtin_vis_fpmax16 (v4hi, v4hi);
v2si __builtin_vis_fpmax32 (v2si, v2si);

v8qi __builtin_vis_fpmaxu8 (v8qi, v8qi);
v4hi __builtin_vis_fpmaxu16 (v4hi, v4hi);
v2si __builtin_vis_fpmaxu32 (v2si, v2si);

v8qi __builtin_vis_fpmin8 (v8qi, v8qi);
v4hi __builtin_vis_fpmin16 (v4hi, v4hi);
v2si __builtin_vis_fpmin32 (v2si, v2si);

v8qi __builtin_vis_fpminu8 (v8qi, v8qi);
v4hi __builtin_vis_fpminu16 (v4hi, v4hi);
v2si __builtin_vis_fpminu32 (v2si, v2si);
```

When you use the `-mvis4b` switch, the VIS version 4.0B built-in functions also become available:

```
v8qi __builtin_vis_dictunpack8 (double, int);
v4hi __builtin_vis_dictunpack16 (double, int);
v2si __builtin_vis_dictunpack32 (double, int);
```

```
long __builtin_vis_fpcmple8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpgt8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpeq8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpne8shl (v8qi, v8qi, int);

long __builtin_vis_fpcmple16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpgt16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpeq16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpne16shl (v4hi, v4hi, int);

long __builtin_vis_fpcmple32shl (v2si, v2si, int);
long __builtin_vis_fpcmpgt32shl (v2si, v2si, int);
long __builtin_vis_fpcmpeq32shl (v2si, v2si, int);
long __builtin_vis_fpcmpne32shl (v2si, v2si, int);

long __builtin_vis_fpcmpule8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpugt8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpule16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpugt16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpule32shl (v2si, v2si, int);
long __builtin_vis_fpcmpugt32shl (v2si, v2si, int);

long __builtin_vis_fpcmpde8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpde16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpde32shl (v2si, v2si, int);

long __builtin_vis_fpcmpur8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpur16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpur32shl (v2si, v2si, int);
```

6.60.34 TI C6X Built-in Functions

GCC provides intrinsics to access certain instructions of the TI C6X processors. These intrinsics, listed below, are available after inclusion of the `c6x_intrinsics.h` header file. They map directly to C6X instructions.

```
int _sadd (int, int);
int _ssub (int, int);
int _sadd2 (int, int);
int _ssub2 (int, int);
long long _mpy2 (int, int);
long long _smpy2 (int, int);
int _add4 (int, int);
int _sub4 (int, int);
int _saddu4 (int, int);

int _smpy (int, int);
int _smpyh (int, int);
int _smpyhl (int, int);
int _smpylh (int, int);

int _sshl (int, int);
int _subc (int, int);

int _avg2 (int, int);
int _avgu4 (int, int);

int _clrr (int, int);
int _extr (int, int);
```

```
int _extru (int, int);
int _abs (int);
int _abs2 (int);
```

6.60.35 x86 Built-in Functions

These built-in functions are available for the x86-32 and x86-64 family of computers, depending on the command-line switches used.

If you specify command-line switches such as `-msse`, the compiler could use the extended instruction sets even if the built-ins are not used explicitly in the program. For this reason, applications that perform run-time CPU detection must compile separate files for each supported architecture, using the appropriate flags. In particular, the file containing the CPU detection code should be compiled without these options.

The following machine modes are available for use with MMX built-in functions (see Section 6.52 [Vector Extensions], page 726): `V2SI` for a vector of two 32-bit integers, `V4HI` for a vector of four 16-bit integers, and `V8QI` for a vector of eight 8-bit integers. Some of the built-in functions operate on MMX registers as a whole 64-bit entity, these use `V1DI` as their mode.

If 3DNow! extensions are enabled, `V2SF` is used as a mode for a vector of two 32-bit floating-point values.

If SSE extensions are enabled, `V4SF` is used for a vector of four 32-bit floating-point values. Some instructions use a vector of four 32-bit integers, these use `V4SI`. Finally, some instructions operate on an entire vector register, interpreting it as a 128-bit integer, these use mode `TI`.

The x86-32 and x86-64 family of processors use additional built-in functions for efficient use of `TF` (`__float128`) 128-bit floating point and `TC` 128-bit complex floating-point values.

The following floating-point built-in functions are always available:

`__float128 __builtin_fabsq (__float128 x)` [Built-in Function]
Computes the absolute value of *x*.

`__float128 __builtin_copysignq (__float128 x, __float128 y)` [Built-in Function]
Copies the sign of *y* into *x* and returns the new value of *x*.

`__float128 __builtin_infq (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `__float128`.

`__float128 __builtin_huge_valq (void)` [Built-in Function]
Similar to `__builtin_huge_val`, except the return type is `__float128`.

`__float128 __builtin_nanq (void)` [Built-in Function]
Similar to `__builtin_nan`, except the return type is `__float128`.

`__float128 __builtin_nansq (void)` [Built-in Function]
Similar to `__builtin_nans`, except the return type is `__float128`.

The following built-in function is always available.

`void __builtin_ia32_pause (void)` [Built-in Function]
Generates the `pause` machine instruction with a compiler memory barrier.

The following built-in functions are always available and can be used to check the target platform type.

`void __builtin_cpu_init (void)` [Built-in Function]
This function runs the CPU detection code to check the type of CPU and the features supported. This built-in function needs to be invoked along with the built-in functions to check CPU type and features, `__builtin_cpu_is` and `__builtin_cpu_supports`, only when used in a function that is executed before any constructors are called. The CPU detection code is automatically executed in a very high priority constructor.

For example, this function has to be used in `ifunc` resolvers that check for CPU type using the built-in functions `__builtin_cpu_is` and `__builtin_cpu_supports`, or in constructors on targets that don't support constructor priority.

```
static void (*resolve_memcpy (void)) (void)
{
  // ifunc resolvers fire before constructors, explicitly call the init
  // function.
  __builtin_cpu_init ();
  if (__builtin_cpu_supports ("ssse3"))
    return ssse3_memcpy; // super fast memcpy with ssse3 instructions.
  else
    return default_memcpy;
}

void *memcpy (void *, const void *, size_t)
     __attribute__ ((ifunc ("resolve_memcpy")));
```

`int __builtin_cpu_is (const char *cpuname)` [Built-in Function]
This function returns a positive integer if the run-time CPU is of type *cpuname* and returns `0` otherwise. The following CPU names can be detected:

'`amd`' AMD CPU.

'`intel`' Intel CPU.

'`atom`' Intel Atom CPU.

'`slm`' Intel Silvermont CPU.

'`core2`' Intel Core 2 CPU.

'`corei7`' Intel Core i7 CPU.

'`nehalem`' Intel Core i7 Nehalem CPU.

'`westmere`'
Intel Core i7 Westmere CPU.

'`sandybridge`'
Intel Core i7 Sandy Bridge CPU.

'`ivybridge`'
Intel Core i7 Ivy Bridge CPU.

‘`haswell`’ Intel Core i7 Haswell CPU.

‘`broadwell`’
Intel Core i7 Broadwell CPU.

‘`skylake`’ Intel Core i7 Skylake CPU.

‘`skylake-avx512`’
Intel Core i7 Skylake AVX512 CPU.

‘`cannonlake`’
Intel Core i7 Cannon Lake CPU.

‘`icelake-client`’
Intel Core i7 Ice Lake Client CPU.

‘`icelake-server`’
Intel Core i7 Ice Lake Server CPU.

‘`cascadelake`’
Intel Core i7 Cascadelake CPU.

‘`tigerlake`’
Intel Core i7 Tigerlake CPU.

‘`cooperlake`’
Intel Core i7 Cooperlake CPU.

‘`sapphirerapids`’
Intel Core i7 sapphirerapids CPU.

‘`alderlake`’
Intel Core i7 Alderlake CPU.

‘`rocketlake`’
Intel Core i7 Rocketlake CPU.

‘`graniterapids`’
Intel Core i7 graniterapids CPU.

‘`graniterapids-d`’
Intel Core i7 graniterapids D CPU.

‘`bonnell`’ Intel Atom Bonnell CPU.

‘`silvermont`’
Intel Atom Silvermont CPU.

‘`goldmont`’
Intel Atom Goldmont CPU.

‘`goldmont-plus`’
Intel Atom Goldmont Plus CPU.

‘`tremont`’ Intel Atom Tremont CPU.

‘`sierraforest`’
Intel Atom Sierra Forest CPU.

'grandridge'
: Intel Atom Grand Ridge CPU.

'knl'
: Intel Knights Landing CPU.

'knm'
: Intel Knights Mill CPU.

'lujiazui'
: ZHAOXIN lujiazui CPU.

'amdfam10h'
: AMD Family 10h CPU.

'barcelona'
: AMD Family 10h Barcelona CPU.

'shanghai'
: AMD Family 10h Shanghai CPU.

'istanbul'
: AMD Family 10h Istanbul CPU.

'btver1'
: AMD Family 14h CPU.

'amdfam15h'
: AMD Family 15h CPU.

'bdver1'
: AMD Family 15h Bulldozer version 1.

'bdver2'
: AMD Family 15h Bulldozer version 2.

'bdver3'
: AMD Family 15h Bulldozer version 3.

'bdver4'
: AMD Family 15h Bulldozer version 4.

'btver2'
: AMD Family 16h CPU.

'amdfam17h'
: AMD Family 17h CPU.

'znver1'
: AMD Family 17h Zen version 1.

'znver2'
: AMD Family 17h Zen version 2.

'amdfam19h'
: AMD Family 19h CPU.

'znver3'
: AMD Family 19h Zen version 3.

'znver4'
: AMD Family 19h Zen version 4.

Here is an example:

```
if (__builtin_cpu_is ("corei7"))
  {
     do_corei7 (); // Core i7 specific implementation.
  }
else
  {
     do_generic (); // Generic implementation.
  }
```

`int __builtin_cpu_supports (const char *feature)` [Built-in Function]

This function returns a positive integer if the run-time CPU supports *feature* and returns `0` otherwise. The following features can be detected:

- '`cmov`' CMOV instruction.
- '`mmx`' MMX instructions.
- '`popcnt`' POPCNT instruction.
- '`sse`' SSE instructions.
- '`sse2`' SSE2 instructions.
- '`sse3`' SSE3 instructions.
- '`ssse3`' SSSE3 instructions.
- '`sse4.1`' SSE4.1 instructions.
- '`sse4.2`' SSE4.2 instructions.
- '`avx`' AVX instructions.
- '`avx2`' AVX2 instructions.
- '`sse4a`' SSE4A instructions.
- '`fma4`' FMA4 instructions.
- '`xop`' XOP instructions.
- '`fma`' FMA instructions.
- '`avx512f`' AVX512F instructions.
- '`bmi`' BMI instructions.
- '`bmi2`' BMI2 instructions.
- '`aes`' AES instructions.
- '`pclmul`' PCLMUL instructions.
- '`avx512vl`' AVX512VL instructions.
- '`avx512bw`' AVX512BW instructions.
- '`avx512dq`' AVX512DQ instructions.
- '`avx512cd`' AVX512CD instructions.
- '`avx512er`' AVX512ER instructions.
- '`avx512pf`' AVX512PF instructions.

‘avx512vbmi’
AVX512VBMI instructions.

‘avx512ifma’
AVX512IFMA instructions.

‘avx5124vnniw’
AVX5124VNNIW instructions.

‘avx5124fmaps’
AVX5124FMAPS instructions.

‘avx512vpopcntdq’
AVX512VPOPCNTDQ instructions.

‘avx512vbmi2’
AVX512VBMI2 instructions.

‘gfni’ GFNI instructions.

‘vpclmulqdq’
VPCLMULQDQ instructions.

‘avx512vnni’
AVX512VNNI instructions.

‘avx512bitalg’
AVX512BITALG instructions.

‘x86-64’ Baseline x86-64 microarchitecture level (as defined in x86-64 psABI).

‘x86-64-v2’
x86-64-v2 microarchitecture level.

‘x86-64-v3’
x86-64-v3 microarchitecture level.

‘x86-64-v4’
x86-64-v4 microarchitecture level.

Here is an example:

```
if (__builtin_cpu_supports ("popcnt"))
  {
     asm("popcnt %1,%0" : "=r"(count) : "rm"(n) : "cc");
  }
else
  {
     count = generic_countbits (n); //generic implementation.
  }
```

The following built-in functions are made available by `-mmmx`. All of them generate the machine instruction that is part of the name.

```
v8qi __builtin_ia32_paddb (v8qi, v8qi);
v4hi __builtin_ia32_paddw (v4hi, v4hi);
v2si __builtin_ia32_paddd (v2si, v2si);
v8qi __builtin_ia32_psubb (v8qi, v8qi);
v4hi __builtin_ia32_psubw (v4hi, v4hi);
```

```
v2si __builtin_ia32_psubd (v2si, v2si);
v8qi __builtin_ia32_paddsb (v8qi, v8qi);
v4hi __builtin_ia32_paddsw (v4hi, v4hi);
v8qi __builtin_ia32_psubsb (v8qi, v8qi);
v4hi __builtin_ia32_psubsw (v4hi, v4hi);
v8qi __builtin_ia32_paddusb (v8qi, v8qi);
v4hi __builtin_ia32_paddusw (v4hi, v4hi);
v8qi __builtin_ia32_psubusb (v8qi, v8qi);
v4hi __builtin_ia32_psubusw (v4hi, v4hi);
v4hi __builtin_ia32_pmullw (v4hi, v4hi);
v4hi __builtin_ia32_pmulhw (v4hi, v4hi);
di __builtin_ia32_pand (di, di);
di __builtin_ia32_pandn (di,di);
di __builtin_ia32_por (di, di);
di __builtin_ia32_pxor (di, di);
v8qi __builtin_ia32_pcmpeqb (v8qi, v8qi);
v4hi __builtin_ia32_pcmpeqw (v4hi, v4hi);
v2si __builtin_ia32_pcmpeqd (v2si, v2si);
v8qi __builtin_ia32_pcmpgtb (v8qi, v8qi);
v4hi __builtin_ia32_pcmpgtw (v4hi, v4hi);
v2si __builtin_ia32_pcmpgtd (v2si, v2si);
v8qi __builtin_ia32_punpckhbw (v8qi, v8qi);
v4hi __builtin_ia32_punpckhwd (v4hi, v4hi);
v2si __builtin_ia32_punpckhdq (v2si, v2si);
v8qi __builtin_ia32_punpcklbw (v8qi, v8qi);
v4hi __builtin_ia32_punpcklwd (v4hi, v4hi);
v2si __builtin_ia32_punpckldq (v2si, v2si);
v8qi __builtin_ia32_packsswb (v4hi, v4hi);
v4hi __builtin_ia32_packssdw (v2si, v2si);
v8qi __builtin_ia32_packuswb (v4hi, v4hi);

v4hi __builtin_ia32_psllw (v4hi, v4hi);
v2si __builtin_ia32_pslld (v2si, v2si);
v1di __builtin_ia32_psllq (v1di, v1di);
v4hi __builtin_ia32_psrlw (v4hi, v4hi);
v2si __builtin_ia32_psrld (v2si, v2si);
v1di __builtin_ia32_psrlq (v1di, v1di);
v4hi __builtin_ia32_psraw (v4hi, v4hi);
v2si __builtin_ia32_psrad (v2si, v2si);
v4hi __builtin_ia32_psllwi (v4hi, int);
v2si __builtin_ia32_pslldi (v2si, int);
v1di __builtin_ia32_psllqi (v1di, int);
v4hi __builtin_ia32_psrlwi (v4hi, int);
v2si __builtin_ia32_psrldi (v2si, int);
v1di __builtin_ia32_psrlqi (v1di, int);
v4hi __builtin_ia32_psrawi (v4hi, int);
v2si __builtin_ia32_psradi (v2si, int);
```

The following built-in functions are made available either with `-msse`, or with `-m3dnowa`. All of them generate the machine instruction that is part of the name.

```
v4hi __builtin_ia32_pmulhuw (v4hi, v4hi);
v8qi __builtin_ia32_pavgb (v8qi, v8qi);
v4hi __builtin_ia32_pavgw (v4hi, v4hi);
v1di __builtin_ia32_psadbw (v8qi, v8qi);
v8qi __builtin_ia32_pmaxub (v8qi, v8qi);
v4hi __builtin_ia32_pmaxsw (v4hi, v4hi);
v8qi __builtin_ia32_pminub (v8qi, v8qi);
v4hi __builtin_ia32_pminsw (v4hi, v4hi);
```

```
int __builtin_ia32_pmovmskb (v8qi);
void __builtin_ia32_maskmovq (v8qi, v8qi, char *);
void __builtin_ia32_movntq (di *, di);
void __builtin_ia32_sfence (void);
```

The following built-in functions are available when `-msse` is used. All of them generate the machine instruction that is part of the name.

```
int __builtin_ia32_comieq (v4sf, v4sf);
int __builtin_ia32_comineq (v4sf, v4sf);
int __builtin_ia32_comilt (v4sf, v4sf);
int __builtin_ia32_comile (v4sf, v4sf);
int __builtin_ia32_comigt (v4sf, v4sf);
int __builtin_ia32_comige (v4sf, v4sf);
int __builtin_ia32_ucomieq (v4sf, v4sf);
int __builtin_ia32_ucomineq (v4sf, v4sf);
int __builtin_ia32_ucomilt (v4sf, v4sf);
int __builtin_ia32_ucomile (v4sf, v4sf);
int __builtin_ia32_ucomigt (v4sf, v4sf);
int __builtin_ia32_ucomige (v4sf, v4sf);
v4sf __builtin_ia32_addps (v4sf, v4sf);
v4sf __builtin_ia32_subps (v4sf, v4sf);
v4sf __builtin_ia32_mulps (v4sf, v4sf);
v4sf __builtin_ia32_divps (v4sf, v4sf);
v4sf __builtin_ia32_addss (v4sf, v4sf);
v4sf __builtin_ia32_subss (v4sf, v4sf);
v4sf __builtin_ia32_mulss (v4sf, v4sf);
v4sf __builtin_ia32_divss (v4sf, v4sf);
v4sf __builtin_ia32_cmpeqps (v4sf, v4sf);
v4sf __builtin_ia32_cmpltps (v4sf, v4sf);
v4sf __builtin_ia32_cmpleps (v4sf, v4sf);
v4sf __builtin_ia32_cmpgtps (v4sf, v4sf);
v4sf __builtin_ia32_cmpgeps (v4sf, v4sf);
v4sf __builtin_ia32_cmpunordps (v4sf, v4sf);
v4sf __builtin_ia32_cmpneqps (v4sf, v4sf);
v4sf __builtin_ia32_cmpnltps (v4sf, v4sf);
v4sf __builtin_ia32_cmpnleps (v4sf, v4sf);
v4sf __builtin_ia32_cmpngtps (v4sf, v4sf);
v4sf __builtin_ia32_cmpngeps (v4sf, v4sf);
v4sf __builtin_ia32_cmpordps (v4sf, v4sf);
v4sf __builtin_ia32_cmpeqss (v4sf, v4sf);
v4sf __builtin_ia32_cmpltss (v4sf, v4sf);
v4sf __builtin_ia32_cmpless (v4sf, v4sf);
v4sf __builtin_ia32_cmpunordss (v4sf, v4sf);
v4sf __builtin_ia32_cmpneqss (v4sf, v4sf);
v4sf __builtin_ia32_cmpnltss (v4sf, v4sf);
v4sf __builtin_ia32_cmpnless (v4sf, v4sf);
v4sf __builtin_ia32_cmpordss (v4sf, v4sf);
v4sf __builtin_ia32_maxps (v4sf, v4sf);
v4sf __builtin_ia32_maxss (v4sf, v4sf);
v4sf __builtin_ia32_minps (v4sf, v4sf);
v4sf __builtin_ia32_minss (v4sf, v4sf);
v4sf __builtin_ia32_andps (v4sf, v4sf);
v4sf __builtin_ia32_andnps (v4sf, v4sf);
v4sf __builtin_ia32_orps (v4sf, v4sf);
v4sf __builtin_ia32_xorps (v4sf, v4sf);
v4sf __builtin_ia32_movss (v4sf, v4sf);
v4sf __builtin_ia32_movhlps (v4sf, v4sf);
v4sf __builtin_ia32_movlhps (v4sf, v4sf);
```

```
v4sf __builtin_ia32_unpckhps (v4sf, v4sf);
v4sf __builtin_ia32_unpcklps (v4sf, v4sf);
v4sf __builtin_ia32_cvtpi2ps (v4sf, v2si);
v4sf __builtin_ia32_cvtsi2ss (v4sf, int);
v2si __builtin_ia32_cvtps2pi (v4sf);
int __builtin_ia32_cvtss2si (v4sf);
v2si __builtin_ia32_cvttps2pi (v4sf);
int __builtin_ia32_cvttss2si (v4sf);
v4sf __builtin_ia32_rcpps (v4sf);
v4sf __builtin_ia32_rsqrtps (v4sf);
v4sf __builtin_ia32_sqrtps (v4sf);
v4sf __builtin_ia32_rcpss (v4sf);
v4sf __builtin_ia32_rsqrtss (v4sf);
v4sf __builtin_ia32_sqrtss (v4sf);
v4sf __builtin_ia32_shufps (v4sf, v4sf, int);
void __builtin_ia32_movntps (float *, v4sf);
int __builtin_ia32_movmskps (v4sf);
```

The following built-in functions are available when `-msse` is used.

`v4sf __builtin_ia32_loadups (float *)` [Built-in Function]
Generates the `movups` machine instruction as a load from memory.

`void __builtin_ia32_storeups (float *, v4sf)` [Built-in Function]
Generates the `movups` machine instruction as a store to memory.

`v4sf __builtin_ia32_loadss (float *)` [Built-in Function]
Generates the `movss` machine instruction as a load from memory.

`v4sf __builtin_ia32_loadhps (v4sf, const v2sf *)` [Built-in Function]
Generates the `movhps` machine instruction as a load from memory.

`v4sf __builtin_ia32_loadlps (v4sf, const v2sf *)` [Built-in Function]
Generates the `movlps` machine instruction as a load from memory

`void __builtin_ia32_storehps (v2sf *, v4sf)` [Built-in Function]
Generates the `movhps` machine instruction as a store to memory.

`void __builtin_ia32_storelps (v2sf *, v4sf)` [Built-in Function]
Generates the `movlps` machine instruction as a store to memory.

The following built-in functions are available when `-msse2` is used. All of them generate the machine instruction that is part of the name.

```
int __builtin_ia32_comisdeq (v2df, v2df);
int __builtin_ia32_comisdlt (v2df, v2df);
int __builtin_ia32_comisdle (v2df, v2df);
int __builtin_ia32_comisdgt (v2df, v2df);
int __builtin_ia32_comisdge (v2df, v2df);
int __builtin_ia32_comisdneq (v2df, v2df);
int __builtin_ia32_ucomisdeq (v2df, v2df);
int __builtin_ia32_ucomisdlt (v2df, v2df);
int __builtin_ia32_ucomisdle (v2df, v2df);
int __builtin_ia32_ucomisdgt (v2df, v2df);
int __builtin_ia32_ucomisdge (v2df, v2df);
int __builtin_ia32_ucomisdneq (v2df, v2df);
v2df __builtin_ia32_cmpeqpd (v2df, v2df);
```

```
v2df __builtin_ia32_cmpltpd (v2df, v2df);
v2df __builtin_ia32_cmplepd (v2df, v2df);
v2df __builtin_ia32_cmpgtpd (v2df, v2df);
v2df __builtin_ia32_cmpgepd (v2df, v2df);
v2df __builtin_ia32_cmpunordpd (v2df, v2df);
v2df __builtin_ia32_cmpneqpd (v2df, v2df);
v2df __builtin_ia32_cmpnltpd (v2df, v2df);
v2df __builtin_ia32_cmpnlepd (v2df, v2df);
v2df __builtin_ia32_cmpngtpd (v2df, v2df);
v2df __builtin_ia32_cmpngepd (v2df, v2df);
v2df __builtin_ia32_cmpordpd (v2df, v2df);
v2df __builtin_ia32_cmpeqsd (v2df, v2df);
v2df __builtin_ia32_cmpltsd (v2df, v2df);
v2df __builtin_ia32_cmplesd (v2df, v2df);
v2df __builtin_ia32_cmpunordsd (v2df, v2df);
v2df __builtin_ia32_cmpneqsd (v2df, v2df);
v2df __builtin_ia32_cmpnltsd (v2df, v2df);
v2df __builtin_ia32_cmpnlesd (v2df, v2df);
v2df __builtin_ia32_cmpordsd (v2df, v2df);
v2di __builtin_ia32_paddq (v2di, v2di);
v2di __builtin_ia32_psubq (v2di, v2di);
v2df __builtin_ia32_addpd (v2df, v2df);
v2df __builtin_ia32_subpd (v2df, v2df);
v2df __builtin_ia32_mulpd (v2df, v2df);
v2df __builtin_ia32_divpd (v2df, v2df);
v2df __builtin_ia32_addsd (v2df, v2df);
v2df __builtin_ia32_subsd (v2df, v2df);
v2df __builtin_ia32_mulsd (v2df, v2df);
v2df __builtin_ia32_divsd (v2df, v2df);
v2df __builtin_ia32_minpd (v2df, v2df);
v2df __builtin_ia32_maxpd (v2df, v2df);
v2df __builtin_ia32_minsd (v2df, v2df);
v2df __builtin_ia32_maxsd (v2df, v2df);
v2df __builtin_ia32_andpd (v2df, v2df);
v2df __builtin_ia32_andnpd (v2df, v2df);
v2df __builtin_ia32_orpd (v2df, v2df);
v2df __builtin_ia32_xorpd (v2df, v2df);
v2df __builtin_ia32_movsd (v2df, v2df);
v2df __builtin_ia32_unpckhpd (v2df, v2df);
v2df __builtin_ia32_unpcklpd (v2df, v2df);
v16qi __builtin_ia32_paddb128 (v16qi, v16qi);
v8hi __builtin_ia32_paddw128 (v8hi, v8hi);
v4si __builtin_ia32_paddd128 (v4si, v4si);
v2di __builtin_ia32_paddq128 (v2di, v2di);
v16qi __builtin_ia32_psubb128 (v16qi, v16qi);
v8hi __builtin_ia32_psubw128 (v8hi, v8hi);
v4si __builtin_ia32_psubd128 (v4si, v4si);
v2di __builtin_ia32_psubq128 (v2di, v2di);
v8hi __builtin_ia32_pmullw128 (v8hi, v8hi);
v8hi __builtin_ia32_pmulhw128 (v8hi, v8hi);
v2di __builtin_ia32_pand128 (v2di, v2di);
v2di __builtin_ia32_pandn128 (v2di, v2di);
v2di __builtin_ia32_por128 (v2di, v2di);
v2di __builtin_ia32_pxor128 (v2di, v2di);
v16qi __builtin_ia32_pavgb128 (v16qi, v16qi);
v8hi __builtin_ia32_pavgw128 (v8hi, v8hi);
v16qi __builtin_ia32_pcmpeqb128 (v16qi, v16qi);
v8hi __builtin_ia32_pcmpeqw128 (v8hi, v8hi);
```

```
v4si __builtin_ia32_pcmpeqd128 (v4si, v4si);
v16qi __builtin_ia32_pcmpgtb128 (v16qi, v16qi);
v8hi __builtin_ia32_pcmpgtw128 (v8hi, v8hi);
v4si __builtin_ia32_pcmpgtd128 (v4si, v4si);
v16qi __builtin_ia32_pmaxub128 (v16qi, v16qi);
v8hi __builtin_ia32_pmaxsw128 (v8hi, v8hi);
v16qi __builtin_ia32_pminub128 (v16qi, v16qi);
v8hi __builtin_ia32_pminsw128 (v8hi, v8hi);
v16qi __builtin_ia32_punpckhbw128 (v16qi, v16qi);
v8hi __builtin_ia32_punpckhwd128 (v8hi, v8hi);
v4si __builtin_ia32_punpckhdq128 (v4si, v4si);
v2di __builtin_ia32_punpckhqdq128 (v2di, v2di);
v16qi __builtin_ia32_punpcklbw128 (v16qi, v16qi);
v8hi __builtin_ia32_punpcklwd128 (v8hi, v8hi);
v4si __builtin_ia32_punpckldq128 (v4si, v4si);
v2di __builtin_ia32_punpcklqdq128 (v2di, v2di);
v16qi __builtin_ia32_packsswb128 (v8hi, v8hi);
v8hi __builtin_ia32_packssdw128 (v4si, v4si);
v16qi __builtin_ia32_packuswb128 (v8hi, v8hi);
v8hi __builtin_ia32_pmulhuw128 (v8hi, v8hi);
void __builtin_ia32_maskmovdqu (v16qi, v16qi);
v2df __builtin_ia32_loadupd (double *);
void __builtin_ia32_storeupd (double *, v2df);
v2df __builtin_ia32_loadhpd (v2df, double const *);
v2df __builtin_ia32_loadlpd (v2df, double const *);
int __builtin_ia32_movmskpd (v2df);
int __builtin_ia32_pmovmskb128 (v16qi);
void __builtin_ia32_movnti (int *, int);
void __builtin_ia32_movnti64 (long long int *, long long int);
void __builtin_ia32_movntpd (double *, v2df);
void __builtin_ia32_movntdq (v2df *, v2df);
v4si __builtin_ia32_pshufd (v4si, int);
v8hi __builtin_ia32_pshuflw (v8hi, int);
v8hi __builtin_ia32_pshufhw (v8hi, int);
v2di __builtin_ia32_psadbw128 (v16qi, v16qi);
v2df __builtin_ia32_sqrtpd (v2df);
v2df __builtin_ia32_sqrtsd (v2df);
v2df __builtin_ia32_shufpd (v2df, v2df, int);
v2df __builtin_ia32_cvtdq2pd (v4si);
v4sf __builtin_ia32_cvtdq2ps (v4si);
v4si __builtin_ia32_cvtpd2dq (v2df);
v2si __builtin_ia32_cvtpd2pi (v2df);
v4sf __builtin_ia32_cvtpd2ps (v2df);
v4si __builtin_ia32_cvttpd2dq (v2df);
v2si __builtin_ia32_cvttpd2pi (v2df);
v2df __builtin_ia32_cvtpi2pd (v2si);
int __builtin_ia32_cvtsd2si (v2df);
int __builtin_ia32_cvttsd2si (v2df);
long long __builtin_ia32_cvtsd2si64 (v2df);
long long __builtin_ia32_cvttsd2si64 (v2df);
v4si __builtin_ia32_cvtps2dq (v4sf);
v2df __builtin_ia32_cvtps2pd (v4sf);
v4si __builtin_ia32_cvttps2dq (v4sf);
v2df __builtin_ia32_cvtsi2sd (v2df, int);
v2df __builtin_ia32_cvtsi642sd (v2df, long long);
v4sf __builtin_ia32_cvtsd2ss (v4sf, v2df);
v2df __builtin_ia32_cvtss2sd (v2df, v4sf);
void __builtin_ia32_clflush (const void *);
```

```
void __builtin_ia32_lfence (void);
void __builtin_ia32_mfence (void);
v16qi __builtin_ia32_loaddqu (const char *);
void __builtin_ia32_storedqu (char *, v16qi);
v1di __builtin_ia32_pmuludq (v2si, v2si);
v2di __builtin_ia32_pmuludq128 (v4si, v4si);
v8hi __builtin_ia32_psllw128 (v8hi, v8hi);
v4si __builtin_ia32_pslld128 (v4si, v4si);
v2di __builtin_ia32_psllq128 (v2di, v2di);
v8hi __builtin_ia32_psrlw128 (v8hi, v8hi);
v4si __builtin_ia32_psrld128 (v4si, v4si);
v2di __builtin_ia32_psrlq128 (v2di, v2di);
v8hi __builtin_ia32_psraw128 (v8hi, v8hi);
v4si __builtin_ia32_psrad128 (v4si, v4si);
v2di __builtin_ia32_pslldqi128 (v2di, int);
v8hi __builtin_ia32_psllwi128 (v8hi, int);
v4si __builtin_ia32_pslldi128 (v4si, int);
v2di __builtin_ia32_psllqi128 (v2di, int);
v2di __builtin_ia32_psrldqi128 (v2di, int);
v8hi __builtin_ia32_psrlwi128 (v8hi, int);
v4si __builtin_ia32_psrldi128 (v4si, int);
v2di __builtin_ia32_psrlqi128 (v2di, int);
v8hi __builtin_ia32_psrawi128 (v8hi, int);
v4si __builtin_ia32_psradi128 (v4si, int);
v4si __builtin_ia32_pmaddwd128 (v8hi, v8hi);
v2di __builtin_ia32_movq128 (v2di);
```

The following built-in functions are available when `-msse3` is used. All of them generate the machine instruction that is part of the name.

```
v2df __builtin_ia32_addsubpd (v2df, v2df);
v4sf __builtin_ia32_addsubps (v4sf, v4sf);
v2df __builtin_ia32_haddpd (v2df, v2df);
v4sf __builtin_ia32_haddps (v4sf, v4sf);
v2df __builtin_ia32_hsubpd (v2df, v2df);
v4sf __builtin_ia32_hsubps (v4sf, v4sf);
v16qi __builtin_ia32_lddqu (char const *);
void __builtin_ia32_monitor (void *, unsigned int, unsigned int);
v4sf __builtin_ia32_movshdup (v4sf);
v4sf __builtin_ia32_movsldup (v4sf);
void __builtin_ia32_mwait (unsigned int, unsigned int);
```

The following built-in functions are available when `-mssse3` is used. All of them generate the machine instruction that is part of the name.

```
v2si __builtin_ia32_phaddd (v2si, v2si);
v4hi __builtin_ia32_phaddw (v4hi, v4hi);
v4hi __builtin_ia32_phaddsw (v4hi, v4hi);
v2si __builtin_ia32_phsubd (v2si, v2si);
v4hi __builtin_ia32_phsubw (v4hi, v4hi);
v4hi __builtin_ia32_phsubsw (v4hi, v4hi);
v4hi __builtin_ia32_pmaddubsw (v8qi, v8qi);
v4hi __builtin_ia32_pmulhrsw (v4hi, v4hi);
v8qi __builtin_ia32_pshufb (v8qi, v8qi);
v8qi __builtin_ia32_psignb (v8qi, v8qi);
v2si __builtin_ia32_psignd (v2si, v2si);
v4hi __builtin_ia32_psignw (v4hi, v4hi);
v1di __builtin_ia32_palignr (v1di, v1di, int);
v8qi __builtin_ia32_pabsb (v8qi);
v2si __builtin_ia32_pabsd (v2si);
```