Intel(R) C++ Compiler Reference

Document Number: 307777-004US

Disclaimer and Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.
The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.
Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Table Of Contents

Compiler Reference

This reference for the Intel® C++ Compiler includes the following sections:

- Intel C++ Compiler Pragmas
- Intel Math Library
- Intel C++ Class Libraries

Intel® C++ Compiler Pragmas

Pragmas are directives that provide instructions to the compiler for use in specific cases. For example, you can use the `novector` pragma to specify that a loop should never be vectorized. The keyword `#pragma` is standard in the C++ language, but individual pragmas are machine-specific or operating system-specific, and vary by compiler.

Some pragmas provide the same functionality as do compiler options. Pragmas override behavior specified by compiler options.

The Intel® C++ Compiler pragmas are categorized as follows:

- Intel-Specific Pragmas - pragmas developed or modified by Intel to work specifically with the Intel® C++ Compiler
- Intel Supported Pragmas - pragmas developed by external sources that are supported by the Intel® C++ Compiler for compatibility reasons

Using Pragmas

You enter pragmas into your C++ source code using the following syntax:

```
#pragma <pragma name>
```

Individual Pragma Descriptions

Each pragma description has the following details:

Section	Description
Short Description	Contains a brief description of what the pragma does
Syntax	Contains the pragma syntax
Arguments	Contains a list of the arguments with descriptions
Description	Contains a detailed description of what the pragma does
Example	Contains typical usage example/s
See Also	Contains links to other pragmas or related topics

Intel-Specific Pragmas

The Intel-specific C++ compiler pragmas are listed below. Click on the pragmas for a more detailed description.

Pragma	Description
`alloc_section`	allocates variable in specified section
`distribute_point`	instructs the compiler to prefer loop distribution at the location indicated
intel_omp_task	specifies a unit of work, potentially executed by a different thread
`intel_omp_taskq`	specifies a unit of work, potentially executed by a different thread
`ivdep`	instructs the compiler to ignore assumed vector dependencies
`loop_count`	indicates the loop count is likely to be an integer
`memref_control`	provides a method for controlling load latency at the variable level
`novector`	specifies that the loop should never be vectorized
`optimize`	enables or disables optimizations for specific functions; provides some degree of compatibility with Microsoft's implementation of `optimize` pragma
`optimization_level`	enables control of optimization for a specific function
`parallel`	parallelizes a loop when it is safe to do so
`prefetch/noprefetch`	asserts that the data prefetches are generated or not generated for some memory references
`swp/noswp`	`swp` indicates preference for loops to be software pipelined; `noswp` indicates the loops not to be software pipelined
`unroll/nounroll`	instructs the compiler the number of times to unroll/not to unroll a loop
`unused`	describes variables that are unused (warnings not generated)
`vector`	indicates to the compiler that the loop should be vectorized according to the arguments: always/aligned/unaligned/nontemporal

The following Intel-specific C++ compiler pragmas are also available.

Pragma	Description
point	suggests point for loop distribution
file	SCO pragma; similar to ident pragma; puts an arbitrary string in the .comment section of the object file
intrin_limited_domain	floating-Point pragma; sets the intrin_limited_domain floating-point attribute on or off
linkage	specifies linkage types; linkage type specifies how a function uses a set of registers; HP compatible
count	estimates trip count (vectorization hint)
maintain_errno	floating-Point pragma; sets the maintain_errno floating-point attribute on or off
noparallel	OpenMP pragma; disables auto-parallelization for an immediately following DO loop
options	GCC-compatible (MacOS) pragma; sets the alignment of fields in structures
pointer_size	HP-compatible; controls pointer size allocation for references and pointer/function/array declarations; pragma is DISABLED
compute	software-based speculative pre-computation (SSP) pragma;
speculate	SSP pragma; enables SSP for the code block marked with SPECULATE
thread	SSP pragma;
system_header	GCC-compatible; causes the rest of the code in the current file to be treated as if it came from a system header
tail_dup	turns on/off tail duplication for loops
use_linkage	HP-compatible; associates a specified linkage with a function
working_directory	SCO pragma; similar to file pragma

alloc_section

Allocates variable in specified section. Controls section attribute specification for variables.

Syntax

```
#pragma alloc_section(var, "attribute-list")
```

Arguments

Argument	Description
var	variable that can be used to define a symbol in the section
"attribute-list"	a comma-separated list of attributes; defined values are: 'short' and 'long'.

Description

The `alloc_section` pragma controls section attribute specification for variables. The compiler to decides whether the variable as defined by `var` should go to a "data" or "bss" or "rdata" section.

Example

The following example illustrates how to use the pragma.

Example

```
#pragma alloc_section(var1, "short")
int var1 = 20;
#pragma alloc_section(var2, "short")
extern int var2;
```

distribute point

Instructs the compiler to prefer loop distribution at the location indicated.

Syntax

```
#pragma distribute point
```

Arguments

None

Description

This pragma is used to suggest to the compiler to split large loops into smaller ones; this is particularly useful in cases where optimizations like software-pipelining (SWP) or vectorization cannot take place due to excessive register usage.

Using `distribute point` pragma for a loop distribution strategy enables software pipelining for the new, smaller loops in the IA-64 architecture. By splitting a loop into smaller segments, it is possible to get each smaller loop or at least one of the smaller loops to SWP or vectorize.

- When the pragma is placed inside a loop, the compiler distributes the loop at that point. All loop-carried dependencies are ignored.
- When inside the loop, pragmas cannot be placed within an `if` statement.
- When the pragma is placed outside the loop, the compiler distributes the loop based on an internal heuristic. The compiler determines where to distribute the loops and observes data dependency. If the pragmas are placed inside the loop, the compiler supports multiple instances of the pragma.

Example 1

The following example uses the `distribute point` pragma outside the loop.

Example

```
#define NUM 1024
void loop distribution pragma1(
       double a[NUM], double b[NUM], double c[NUM],
       double x[NUM], double y[NUM], double z[NUM] )
{
  int i;
  // Before distribution or splitting the loop
  #pragma distribute point
  for (i=0; i< NUM; i++) {
    a[i] = a[i] + i;
    b[i] = b[i] + i;
    c[i] = c[i] + i;
    x[i] = x[i] + i;
    y[i] = y[i] + i;
    z[i] = z[i] + i;
  }
}
```

Example 2

The following example uses the `distribute point` pragma inside the loop.

Example

```
#define NUM 1024
void loop distribution pragma2(
       double a[NUM], double b[NUM], double c[NUM],
       double x[NUM], double y[NUM], double z[NUM] )
{
  int i;
  // After distribution or splitting the loop.
  for (i=0; i< NUM; i++) {
    a[i] = a[i] +i;
    b[i] = b[i] +i;
    c[i] = c[i] +i;
    #pragma distribute point
    x[i] = x[i] +i;
    y[i] = y[i] +i;
    z[i] = z[i] +i;
  }
}
```

Example 3

The following example shows how to use the `distribute point` pragma, first outside the loop and then inside the loop.

Using distribute point pragma

```
void dist1(int a[], int b[], int c[], int d[])
{
  #pragma distribute point
// Compiler will automatically decide where to
// distribute. Data dependency is observed.
  for (int i=1; i<1000; i++) {
    b[i] = a[i] + 1;
    c[i] = a[i] + b[i];
    d[i] = c[i] + 1;
  }
}

void dist2(int a[], int b[], int c[], int d[])
{
  for (int i=1; i<1000; i++) {
    b[i] = a[i] + 1;
    #pragma distribute point
// Distribution will start here,
// ignoring all loop-carried dependency.
    c[i] = a[i] + b[i];
    d[i] = c[i] + 1;
  }
}
```

intel_omp_task

Specifies a unit of work, potentially executed by a different thread.

Syntax

```
#pragma intel omp task

[clause[[,]clause]...]

structured-block
```

where `clause` can be any of the following:

- `private( variable-list )`
- `captureprivate( variable-list )`

Arguments

Argument (clause)	Description
`private(variable-list)`	The `private` clause creates a private, default-constructed version for each object in `variable-list` for the `task`. The original object referenced by the variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and has an indeterminate value upon exit from the construct.

captureprivate(variable-list)	The captureprivate clause creates a private, copy-constructed version for each object in variable-list for the task at the time the task is enqueued. The original object referenced by each variable retains its value but must not be modified within the dynamic extent of the task construct.

Description

The task pragma specifies a unit of work, potentially executed by a different thread.

Example

For an example on how to use task pragma see topic Workqueuing Example Function.

See Also

Workqueing Constructs

intel_omp_taskq

Specifies an environment for the while loop in which to enqueue the units of work specified by the enclosed task pragma.

Syntax

```
#pragma intel omp taskq

[clause[[,]clause]...]

structured-block
```

where clause can be any of the following:

- private (*variable-list*)
- firstprivate (*variable-list*)
- lastprivate (*variable-list*)
- reduction (operator : *variable-list*)
- ordered
- nowait

Arguments

Argument (clause)	Description
private(variable-list)	The private clause creates a private, default-constructed version for each object in variable-list for the taskq. It also implies captureprivate on each enclosed task. The original object referenced by each variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic

	extent of the construct, and has an indeterminate value upon exit from the construct.
firstprivate(variable-list)	The `firstprivate` clause creates a private, copy-constructed version for each object in `variable-list` for the `taskq`. It also implies `captureprivate` on each enclosed task. The original object referenced by each variable must not be modified within the dynamic extent of the construct and has an indeterminate value upon exit from the construct.
lastprivate(variable-list)	The `lastprivate` clause creates a private, default-constructed version for each object in `variable-list` for the `taskq`. It also implies `captureprivate` on each enclosed task. The original object referenced by each variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and is copy-assigned the value of the object from the last enclosed task after that task completes execution.
reduction(operator : variable-list)	The `reduction` clause performs a reduction operation with the given operator in enclosed task constructs for each object in `variable-list`. `operator` and `variable-list` are defined the same as in the OpenMP Specifications.
ordered	The `ordered` clause performs ordered constructs in enclosed `task` constructs in original sequential execution order. The `taskq` directive, to which the `ordered` is bound, must have an `ordered` clause present.
nowait	The `nowait` clause removes the implied barrier at the end of the `taskq`. Threads may exit the `taskq` construct before completing all the `task` constructs queued within it.

Description

The `taskq` pragma specifies the environment within which the enclosed units of work (tasks) are to be executed. From among all the threads that encounter a taskq pragma, one is chosen to execute it initially.

Conceptually, the `taskq` pragma causes an empty queue to be created by the chosen thread, and then the code inside the `taskq` block is executed single-threaded. All the other threads wait for work to be enqueued on the conceptual queue.

The `task` pragma specifies a unit of work, potentially executed by a different thread. When a `task` pragma is encountered lexically within a `taskq` block, the code inside the `task` block is conceptually enqueued on the queue associated with the `taskq`. The conceptual queue is disbanded when all work enqueued on it finishes, and when the end of the `taskq` block is reached.

Example

For an example on how to use `taskq` pragma see the topic Workqueuing Example Function.

See Also

Workqueing Constructs

ivdep

Instructs the compiler to ignore assumed vector dependencies.

Syntax

```
#pragma ivdep
```

Arguments

None

Description

The `ivdep` pragma instructs the compiler to ignore assumed vector dependencies. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization. This pragma overrides that decision. Only use this when you know that the assumed loop dependences are safe to ignore.

Note
The proven dependences that prevent vectorization are not ignored, only assumed dependences are ignored.

When an `ivdep` pragma is specified for applications designed to run on IA-64 architectures, the `-ivdep-parallel` (Linux*) or `/Qivdep-parallel` (Windows*) option indicates there is no loop-carried memory dependency in the loop. This technique is useful for some sparse matrix applications. See Example 2.

Example 1

The loop in this example will not vectorize without the `ivdep` pragma, since the value of `k` is not known; vectorization would be illegal if `k<0`.

Example 1

```
void ignore vec dep(int *a, int k, int c, int m)
{
  #pragma ivdep
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```

The pragma binds only the `for` loop contained in current function. This includes a `for` loop contained in a sub-function called by the current function.

Example 2

The following loop requires the parallel option in addition to the `ivdep` pragma to indicate there is no loop-carried dependencies:

Example 2

```
#pragma ivdep
for (i=1; i<n; i++)
{
  e[ix[2][i]] = e[ix[2][i]]+1.0;
  e[ix[3][i]] = e[ix[3][i]]+2.0;
}
```

Example 3

The following loop requires the parallel option in addition to the `ivdep` pragma to ensure there is no loop-carried dependency for the store into `a()`.

Example 3

```
#pragma ivdep
for (j=0; j<n; j++)
{
  a[b[j]] = a[b[j]] + 1;
}
```

See Also

In addition to the `ivdep` pragma, the `vector` pragma can be used to override the efficiency heuristics of the vectorizer. See
`vector` pragma
`novector` pragma

loop count

Specifies the iterations for the `for` loop.

Syntax

The syntax for this pragma supports the following variations:

```
#pragma loop count (n)

#pragma loop count = n

or

#pragma loop count (n1[, n2]...)

#pragma loop count = n1[, n2]...

or

#pragma loop count min(n), max(n), avg(n)

#pragma loop count min=n, max=n, avg=n
```

Arguments

The syntax variations support the following arguments:

Argument	Description
(n) or =n	Non-negative integer value. The compiler will attempt to iterate the next loop the number of times specified in `n`; however, the number of iterations is not guaranteed.
(n1[,n2]...) or *= n1[,n2]...*	Non-negative integer values. The compiler will attempt to iterate the next loop the number of time specified by `n1` or `n2`, or some other unspecified number of times. This behavior allows the compiler some flexibility in attempting to unroll the loop. The number of iterations is not guaranteed.
`min(`*n*`), max(`*n*`), avg(`*n*`)` or `min=`*n*`, max=`*n*`, avg=`*n*	Non-negative integer values. Specify one or more in any order without duplication. The compiler insures the next loop iterates for the specified maximum, minimum, or average number (n1) of times. The specified number of iterations is guaranteed for min and max.

Description

The `loop count` Pragma affects heuristics in vectorization, loop-transformations, and software pipelining (IA-64 architecture). The pragma specifies the minimum, maximum, or average number of iterations for a `for` loop. In addition, a list of commonly occurring values can be specified to help the compiler generate multiple versions and perform complete unrolling.

You can specify more than one pragma for a single loop; however, do not duplicate the pragma.

Example 1

The following example illustrates how to use the pragma to iterate through the loop and enable software pipelining on IA-64 architectures.

Example 1: Using Loop Count (n)

```
void loop count(int a[], int b[])
{
#pragma loop count (10000)
for (int i=0; i<1000; i++)
    a[i] = b[i] + 1.2;
}
```

Example 2

The following example illustrates how to use the pragma to iterate through the loop a minimum of three, a maximum of ten, and average of five times.

Example 2: Using Loop Count min, max, avg

```
#include <stdio.h>
int i;
int main()
{
#pragma loop count min(3), max(10), avg(5)
  for (i=1;i<=15;i++){
    printf("i=%d\n",i);
```

```
}
```

memref_control

Provides a method to control load latency and temporal locality at the variable level.

Syntax

```
#pragma memref_control [name1[:<locality>[:<latency>]],[name2...]
```

Arguments

The following table lists the supported arguments.

Argument	Description
`name1, name2`	Specifies the name of array or pointer. You must specify at least one name; however, you can specify names with associated locality and latency values.
`locality`	An optional integer value that indicates the desired cache level to store data for future access. This will determine the load/store hint (or `prefetch` hint) to be used for this reference. The value can be one of the following: • l1 = `0` • l2 = `1` • l3 = `2` • mem = `3` To use this argument, you must also specify name.
`latency`	An optional integer value that indicates the load (or the latency that has to be overlapped if a prefetch is issued for this address). The value can be one of the following: • l1_latency = `0` • l2_latency = `1` • l3_latency = `2` • mem_latency = `3` To use this argument, you must also specify name and locality.

Description

The `memref_control` pragma is supported on Itanium® processors only. This pragma provides a method for controlling load latency and temporal locality at the variable level. The `memref_control` pragma allows you to specify locality and latency at the array level. For example, using this pragma allows you to control the following:

- The location (cache level) to store data for future access.

- The most appropriate latency value to be used for a load, or the latency that has to be overlapped if a `prefetch` is issued for this reference.

When you specify source-level and the data locality information at a high level for a particular data access, the compiler decides how best to use this information. If the compiler can prefetch profitably for the reference, then it issues a `prefetch` with a distance that covers the specified latency specified and then schedules the corresponding load with a smaller latency. It also uses the hints on the prefetch and load appropriately to keep the data in the specified cache level.

If the compiler cannot compute the address in advance, or decides that the overheads for prefetching are too high, it uses the specified latency to separate the load and its use (in a pipelined loop or a Global Code Scheduler loop). The hint on the load/store will correspond to the cache level passed with the locality argument.

You can use this with the `prefetch` and `noprefetch` to further tune the hints and prefetch strategies. When using the `memref_control` with `noprefetch`, keep the following guidelines in mind:

- Specifying `noprefetch` along with the `memref_control` causes the compiler to not issue prefetches; instead the latency values specified in the `memref_control` is used to schedule the load.
- There is no ordering requirement for using the two pragmas together. Specify the two pragmas in either order as long as both are specified consecutively just before the loop where it is to be applied. Issuing a `prefetch` with one hint and loading it later using a different hint can provide greater control over the hints used for specific architectures.
- `memref_control` is handled differently from the `prefetch` or `noprefetch`. Even if the load cannot be prefetched, the reference can still be loaded using a non-default load latency passed to the `latency` argument.

Example 1

The following example illustrates a case where the address is not known in advance, so prefetching is not possible. The compiler, in this case, schedules the loads of the tab array with an L3 load latency of 15 cycles (inside a software pipelined loop or GCS loop).

Example 1: gather

```
#pragma memref control  tab : l2 : l3 latency
for (i=0; i<n; i++)
{
   x = <generate 64 random bits inline>;
   dum += tab[x&mask]; x>>=6;
   dum += tab[x&mask]; x>>=6;
   dum += tab[x&mask]; x>>=6;
}
```

Example 2

The following example illustrates one way of using `memref_control`, `prefetch`, and `noprefetch` together.

Example 2: sparse matrix

```
   if( size <= 1000 ) {
```

```
#pragma noprefetch cp, vp
#pragma memref control x:l2:l3 latency

#pragma noprefetch yp, bp, rp
#pragma noprefetch xp
  for (iii=0; iii<rag1m0; iii++) {
    if( ip < rag2 ) {
      sum  -= vp[ip]*x[cp[ip]];
      ip++;
    } else {
       xp[i] = sum*yp[i];
       i++;
       sum   = bp[i];
       rag2  = rp[i+1];
    }
  }
  xp[i] = sum*yp[i];
} else {

#pragma prefetch cp, vp
#pragma memref control x:l2:mem latency

#pragma prefetch yp, bp, rp
#pragma noprefetch xp
  for (iii=0; iii<rag1m0; iii++) {
    if( ip < rag2 ) {
      sum  -= vp[ip]*x[cp[ip]];
        ip++;
    } else {
      xp[i] = sum*yp[i];
      i++;
      sum   = bp[i];
       rag2  = rp[i+1];
    }
  }
  xp[i] = sum*yp[i];
}
```

novector

Specifies that the loop should never be vectorized.

Syntax

```
#pragma novector
```

Arguments

None

Description

The `novector` pragma specifies that a particular loop should never be vectorized, even if it is legal to do so. When avoiding vectorization of a loop is desirable (when vectorization results in a performance regression rather than improvement), the `novector` pragma can be used in the source text to disable vectorization of a loop. This behavior is in contrast to the `vector always` pragma.

Example

When you know the trip count (ub - lb) is too low to make vectorization worthwhile, you can use novector to tell the compiler not to vectorize, even if the loop is considered vectorizable.

Example: novector pragma

```
void foo(int lb, int ub)
{
  #pragma novector
  for(j=lb; j<ub; j++)
  {
     a[j]=a[j]+b[j];
  }
}
```

See Also

`vector` pragma

optimize

Enables or disables optimizations for specific functions.

Syntax

```
#pragma optimize("", on|off)
```

Arguments

The compiler ignores first argument values. Valid second arguments for `optimize` are:

Argument	Description
off	disables optimization
on	enables optimization

Description

The `optimize` pragma is used to enable or disable optimizations for specific functions. Specifying `#pragma optimize("", off)` disables optimization until either the compiler finds a matching `#pragma optimize("", on)` statement or until the compiler reaches the end of the translation unit.

Example 1

In the following example, optimizations are disabled for the `alpha()` function but not for `omega()`.

Example 1: Disabling optimization for a single function

```
#pragma optimize("", off)
alpha() {
...
}
#pragma optimize("", on)
omega() {
...
```

```
}
```

Example 2

In the following example, optimizations are disabled for both the `alpha()` and `omega()` functions.

Example 2: Disabling optimization for all functions

```
#pragma optimize("", off)
alpha() {
...
}
omega() {
...
}
```

optimization_level

Controls optimization for one function or all functions after its first occurrence.

Syntax

```
#pragma [intel|GCC] optimization_level n
```

Arguments

Argument	Description
intel|GCC	indicates the interpretation to use
n	an integer value specifying an optimization level; valid values are: • `0` same optimizations as `-O0` • `1` same optimizations as `-O1` • `2` same optimizations as `-O2` • `3` same optimizations as `-O3`

Note

For more information on the optimizations levels, see Enabling Automatic Optimizations.

Description

The `optimization_level` pragma is used to restrict optimization for a specific function while optimizing the remaining application using a different, higher optimization level. For example, if you specify `-O3` (Linux and Mac OS) for the application and specify `#pragma optimization_level 1`, the marked function will be optimized at the `-O1` option level, while the remaining application will be optimized at the higher level.

In general, the pragma optimizes the function at the level specified as *n*; however, certain compiler optimizations, like Inter-procedural Optimization (IPO), are not enabled or disabled during translation unit compilation. For example, if you enable IPO and a specific optimization level, IPO is enabled even for the function targeted by this pragma; however, IPO might not be fully implemented regardless of the optimization level specified at the command line. The reverse is also true.

Scope of optimization restriction

On Linux* and Mac OS*, the scope of the optimization restriction can be affected by arguments passed to the `-pragma-optimization-level` compiler option as explained in the following table.

Syntax	Behavior
`#pragma intel optimization_level n`	Applies pragma only to the next function, using the specified optimization level, regardless of the argument passed to the `-pragma-optimization-level` option.

`#pragma GCC optimization_level n`
 	
or
 	
`#pragma GCC optimization_level reset`
	Applies pragma to all subsequent functions, using the specified optimization level, regardless of the argument passed to the `-pragma-optimization-level` option.
	
	Specifying `reset` reverses the effect of the most recent `#pragma GCC optimization_level` statement, by returning to the optimization level previously specified.

`#pragma optimization level n`	Applies either the intel or GCC interpretation. Interpretation depends on argument passed to the `-pragma-optimization-level` option.

On Windows*, the pragma always uses the `intel` interpretation; the pragma is applied only to the next function.

Using the `intel` interpretation of the pragma

Place the pragma immediately before the function being affected. See Example below.

Using the `GCC` interpretation of the pragma

Place the pragma in any location prior to the functions being affected.

Example

Example: intel interpretation of pragma

```
#pragma intel optimization level 1
gamma() {
...
}
```

parallel

Instructs the compiler to parallelize the loop.

Syntax

```
#pragma parallel
```

Arguments

None

Description

The `parallel` pragma is used to instruct the compiler to parallelize the loop when parallelizing a particular loop is safe and potential aliases can be ignored.

Example

The following example illustrates how to use the pragma...

Example

```
void add(int k, float *a, float *b)
{
  #pragma parallel
  for (int i = 0; i < 10000; i++)
    a[i] = a[i+k] + b[i];
}
```

See Also

Programming for Multithread Platform Consistency

prefetch/noprefetch

Affect the heuristics used in the compiler by asserting that data prefetches are generated.

Syntax

The general syntax for the pragma pair is shown below:

```
#pragma prefetch

#pragma prefetch a,b

#pragma noprefetch
```

Arguments

Argument	Description
a	data to be prefetched
b	data to be prefetched

Description

The prefetch pragma is supported by Itanium® processors only. The pragma asserts that the data prefetches are generated for some memory references. This affects the heuristics used in the compiler.

If the loop includes expression A(j), placing prefetch A in front of the loop, instructs the compiler to insert prefetches for A(j + d) within the loop. d is the number of iterations ahead to prefetch the data and is determined by the compiler.

The prefetch pragma is supported only when option -O3 (Linux*) or /O3 (Windows*) is on. These directives are also supported when you specify options -O1 and -O2 (Linux) or /O1 and /O2 (Windows). Remember that -O2 or /O2 is the default optimization level.

The noprefetch pragma is supported by Itanium® processors only. The pragma asserts that data prefetches are not generated for some memory references. This affects the heuristics used in the compiler.

Example 1

The following example demonstrates how to use the noprefetch and prefetch pragmas together:

Example 1: using noprefetch and prefetch pragmas

```
#pragma noprefetch b
#pragma prefetch a
for(i=0; i<m; i++) {
  a[i]=b[i]+1;
}
```

Example 2

The following is yet another example of how to use the noprefetch and prefetch pragmas:

Example 2: using noprefetch and prefetch pragmas

```
for (i=i0; i!=i1; i+=is) {

float sum = b[i];
int ip = srow[i];
int c = col[ip];

#pragma noprefetch col
#pragma prefetch value:1:80
#pragma prefetch x:1:40

for(; ip<srow[i+1]; c=col[++ip])
  sum -= value[ip] * x[c];
  y[i] = sum;
}
```

Example 3

The following example, which is for IA-64 architecture only, demonstrates how to use the prefetch, noprefetch, and memref_control pragmas together:

Example 3: using noprefetch, prefetch, memref_control pragmas

```
#define SIZE 10000
```

```
int prefetch(int *a, int *b){
  int i, sum = 0;
  #pragma memref control a:l2
  #pragma noprefetch a
  #pragma prefetch b
  for (i = 0; i<SIZE; i++)
    sum += a[i] * b[i];
  return sum;
}
#include <stdio.h>
int main(){
  int i, arr1[SIZE], arr2[SIZE];
  for (i = 0; i<SIZE; i++) {
    arr1[i] = i;
    arr2[i] = i;
  }
  printf("Demonstrating the use of prefetch, noprefetch,\n"
      "and memref control pragma together.\n");
  prefetch(arr1, arr2);
  return 0;
  }
```

See Also

`memref_control` pragma

swp/noswp

Indicates a preference for loops to be software pipelined or not pipelined.

Syntax

The syntax for this pragma pair is shown below:

```
#pragma swp

#pragma noswp
```

Arguments

None

Description

The `swp` pragma indicates a preference for loops to be software pipelined. The pragma does not help data dependency, but overrides heuristics based on profile counts or unequal control flow.

The software pipelining optimization triggered by the `swp` pragma applies instruction scheduling to certain innermost loops, allowing instructions within a loop to be split into different stages, allowing increased instruction-level parallelism.

This strategy can reduce the impact of long-latency operations, resulting in faster loop execution. Loops chosen for software pipelining are always the innermost loops that do not contain procedure calls that are not inlined. Because the optimizer no

longer considers fully unrolled loops as innermost loops, fully unrolling loops can allow an additional loop to become the innermost loop.

The `noswp` pragma is used to instruct the compiler not to software pipeline that loop. This may be advantageous for loops that iterate few times, as pipelining introduces overhead in executing the `prolog` and `epilog` code sequences.

Example

The following example demonstrates one way of using the pragma to instruct the compiler to attempt software pipeling.

Example: using swp pragma

```
void swp(int a[], int b[])
{
  #pragma swp
  for (int i = 0; i < 100; i++)
    if (a[i] == 0)
      b[i] = a[i] + 1;
    else
      b[i] = a[i] * 2;
}
```

See Also

Loop unrolling options
Optimizer Report Generation

unroll/nounroll

Indicates to the compiler to unroll or not to unroll a counted loop.

Syntax

The general syntax for this pragma is shown below:

```
#pragma unroll

#pragma unroll(n)

#pragma nounroll
```

Arguments

Argument	Description
n	is an integer constant from 0 through 255

Description

The `unroll[n]` pragma tells the compiler how many times to unroll a counted loop.

The `unroll` pragma must precede the `FOR` statement for each `FOR` loop it affects. If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted or if it is outside the allowed range, the optimizer assigns the number of times to unroll the loop.

This pragma is supported only when option `-O3` (Linux* and Mac OS*) or `/O3` (Windows*) is used. The `unroll` pragma overrides any setting of loop unrolling from the command line.

Currently, the pragma can be applied only for the innermost loop nest. If applied to the outer loop nests, it is ignored. The compiler generates correct code by comparing *n* and the loop count.

When unrolling a loop increases register pressure and code size it may be necessary to prevent unrolling of a loop. In such cases, use the `nounroll` pragma. The `nounroll` pragma instructs the compiler not to unroll a specified loop.

Example

Example: using unroll pragma

```
void unroll(int a[], int b[], int c[], int d[]){
  #pragma unroll(4)
  for (int i = 1; i < 100; i++) {
    b[i] = a[i] + 1;
    d[i] = c[i] + 1;
  }
}
```

See Also

Loop unrolling options
Optimizer Report Generation
Applying Optimization Strategies

unused

Describes variables that are unused (warnings not generated).

Syntax

```
#pragma unused
```

Arguments

None

Description

The `unused` pragma is implemented for compatibility with Apple* implementation of GCC.

vector

Indicates to the compiler that the loop should be vectorized according to the argument keywords always/aligned/unaligned/nontemporal.

Syntax

```
#pragma vector {always | aligned | unaligned | nontemporal}
```

Arguments

Argument keywords	Description
always	instructs the compiler to override any efficiency heuristic during the decision to vectorize or not, and vectorize non-unit strides or very unaligned memory accesses; controls the vectorization of the subsequent loop in the program
aligned	instructs compiler to use aligned data movement instructions for all array references when vectorizing
unaligned	instructs compiler to use unaligned data movement instructions for all array references when vectorizing
nontemporal	instructs the compiler to implement streaming stores on systems based on IA-32 architecture.

Description

The vector pragma indicates that the loop should be vectorized, if it is legal to do so, ignoring normal heuristic decisions about profitability. The `vector` pragma takes several argument keywords to specify the kind of loop vectorization required. These keywords are aligned, unaligned, always, and nontemporal.

When the `aligned/unaligned` argument keyword is used with this pragma, it indicates that the loop should be vectorized using aligned/unaligned data movement instructions for all array references. Specify only one argument keyword: aligned or unaligned.

Caution

If you specify aligned as an argument, you must be sure that the loop is vectorizable using this instruction. Otherwise, the compiler generates incorrect code.

When the `always` argument keyword is used, the pragma controls the vectorization of the subsequent loop in the program. As the compiler does not apply the `vector` pragma to nested loops, each nested loop needs a preceding pragma statement. Place the pragma before the loop control statement.

Note

The pragma `vector{always|aligned|unaligned}` should be used with care. Overriding the efficiency heuristics of the compiler should only be done if the programmer is absolutely sure the vectorization will improve performance. Furthermore, instructing the compiler to implement all array references with aligned data movement instructions will cause a run-time exception in case some of the access patterns are actually unaligned.

The `vector nontemporal` pragma results in streaming stores on systems based on IA-32 architecture.

Example 1

The loop in the following example uses the `aligned` argument keyword to request that the loop be vectorized with aligned instructions, as the arrays are declared in such a way that the compiler could not normally prove this would be safe to do so.

Example 1: vector aligned pragma

```
void vec aligned(float *a, int m, int c)
{
  int i;
  // Instruct compiler to ignore assumed vector dependencies.
  #pragma vector aligned
  for (i = 0; i < m; i++)
    a[i] = a[i] * c;
  // Alignment unknown but compiler can still align.
  for (i = 0; i < 100; i++)
    a[i] = a[i] + 1.0f;
}
```

Example 2

The following example illustrates how to use the `vector always` pragma.

Example 2: vector always pragma

```
void vec always(int *a, int *b, int m)
{
  #pragma vector always
  for(int i = 0; i <= m; i++)
    a[32*i] = b[99*i];
}
```

Example 3

A float-type loop together with the generated assembly are shown in the following example. For large `N`, significant performance improvements result on Pentium 4 systems over a non-streaming implementation.

Example 3: vector nontemporal pragma

```
#pragma vector nontemporal
for (i = 0; i < N; i++)
  a[i] = 1;
  .B1.2:
movntps XMMWORD PTR  a[eax], xmm0
movntps XMMWORD PTR  a[eax+16], xmm0
add eax, 32
cmp eax, 4096
jl .B1.2
```

Intel Supported Pragmas

Please also see Intel-specific Pragmas. The Intel® C++ Compiler supports the following pragmas:

Pragma	Description
`alloc_text`	names the code section where the specified function definitions are to reside
`auto_inline`	excludes any function defined within the range where `off` is specified from being considered as candidates for automatic inline expansion
`bss_seg`	indicates to the compiler the segment where uninitialized variables are stored in the `.obj` file
`check_stack`	`on` argument indicates that stack checking should be enabled for functions that follow and `off` argument indicates that stack checking should be disabled for functions that follow.
`code_seg`	specifies a code section where functions are to be allocated
`comment`	places a comment record into an object file or executable file
`component`	controls collecting of browse information or dependency information from within source files
`conform`	specifies the run-time behavior of the `/Zc:forScope` compiler option
`const_seg`	specifies the segment where functions are stored in the `.obj` file
`data_seg`	specifies the default section for initialized data
`deprecated`	indicates that a function, type, or any other identifier may not be supported in a future release or indicates that a function, type, or any other identifier should not be used any more
`poison`	labels the identifiers you want removed from your program; an error results when compiling a "poisoned" identifier; `#pragma POISON` is also supported.
`float_control`	specifies floating-point behavior for a function
`fp_contract`	allows or disallows the implementation to contract expressions
`include_directory`	appends the string argument to the list of places to search for #include files; HP compatible pragma
`init_seg`	specifies the section to contain C++ initialization code for

	the translation unit
message	displays the specified string literal to the standard output device
optimize	specifies optimizations to be performed on a function-by-function basis; implemented to partly support Microsoft's implementation of same pragma; click here for Intel's implementation
pointers_to_members	specifies whether a pointer to a class member can be declared before its associated class definition and is used to control the pointer size and the code required to interpret the pointer
pop_macro	sets the value of the macro_name macro to the value on the top of the stack for this macro
push_macro	saves the value of the macro_name macro on the top of the stack for this macro
region/endregion	specifies a code segment in the Microsoft Visual Studio* 2005 Code Editor that expands and contracts by using the outlining feature
section	creates a section in an .obj file. Once a section is defined, it remains valid for the remainder of the compilation
start_map_region	used in conjunction with the stop_map_region pragma
stop_map_region	used in conjunction with the start_map_region pragma
fenv_access	informs an implementation that a program may test status flags or run under a non-default control mode
vtordisp	on argument enables the generation of hidden vtordisp members and off disables them
warning	allows selective modification of the behavior of compiler warning messages
weak	declares symbol you enter to be weak

Intel Math Library

The Intel® C++ Compiler includes a mathematical software library containing highly optimized and very accurate mathematical functions. These functions are commonly used in scientific or graphic applications, as well as other programs that rely heavily on floating-point computations. Support for C99 `_Complex` data types is included by using the `-std=c99` compiler option. The `mathimf.h` header file includes prototypes for the library functions. See Using the Intel Math Library. For a complete list of the functions available, refer to the Function List in this section.

Math Libraries

The math library linked to an application depends on the compilation or linkage options specified.

Library	Description
`libimf.a`	Default static math library.
`libimf.so`	Default shared math library.

Using the Intel Math Library

To use the Intel math library, include the header file, `mathimf.h`, in your program. Here are two example programs that illustrate the use of the math library.

Example Using Real Functions

```
// real math.c

#include <stdio.h>
#include <mathimf.h>

int main() {

float  fp32bits;
double fp64bits;
long double fp80bits;
long double pi by four = 3.141592653589793238/4.0;

// pi/4 radians is about 45 degrees.

fp32bits = (float) pi by four;   // float approximation to pi/4
fp64bits = (double) pi by four;  // double approximation to pi/4
fp80bits = pi by four;           // long double (extended)
approximation to pi/4

// The sin(pi/4) is known to be 1/sqrt(2) or approximately .7071067

printf("When x = %8.8f, sinf(x) = %8.8f \n", fp32bits,
sinf(fp32bits));
printf("When x = %16.16f, sin(x) = %16.16f \n", fp64bits,
sin(fp64bits));
printf("When x = %20.20Lf, sinl(x) = %20.20f \n", fp80bits,
sinl(fp80bits));
```

```
return 0;
}
```

Compiling `real_math.c`:

```
prompt>icc real_math.c
```

The output of `a.out` will look like this:

```
When x = 0.78539816, sinf(x) = 0.70710678
When x = 0.7853981633974483, sin(x) = 0.7071067811865475
When x = 0.78539816339744827900, sinl(x) = 0.70710678118654750275
```

Example Using Complex Functions

```
// complex math.c

#include <stdio.h>
#include <complex.h>

int main()
{

float   Complex c32in,c32out;
double  Complex c64in,c64out;
double pi by four= 3.141592653589793238/4.0;

c64in = 1.0 + I* pi by four;

// Create the double precision complex number 1 + (pi/4) * i
// where I is the imaginary unit.

c32in = (float  Complex) c64in;

// Create the float complex value from the double complex value.

c64out = cexp(c64in);
c32out = cexpf(c32in);

// Call the complex exponential,
// cexp(z) = cexp(x+iy) = e^ (x + i y) = e^x * (cos(y) + i sin(y))

printf("When z = %7.7f + %7.7f i, cexpf(z) = %7.7f + %7.7f i \n"
,crealf(c32in),cimagf(c32in),crealf(c32out),cimagf(c32out));
printf("When z = %12.12f + %12.12f i, cexp(z) = %12.12f + %12.12f i
\n"
,creal(c64in),cimag(c64in),creal(c64out),cimagf(c64out));

return 0;
}
```

```
prompt>icc -std=c99 complex_math.c
```

The output of `a.out` will look like this:

```
When z = 1.0000000 + 0.7853982 i, cexpf(z) = 1.9221154 + 1.9221156 i
When z = 1.000000000000 + 0.785398163397 i, cexp(z) = 1.922115514080 +
1.922115514080 i
```

Note

_Complex data types are supported in C but not in C++ programs.

Exception Conditions

If you call a math function using argument(s) that may produce undefined results, an error number is assigned to the system variable errno. Math function errors are usually domain errors or range errors.

Domain errors result from arguments that are outside the domain of the function. For example, acos is defined only for arguments between -1 and +1 inclusive. Attempting to evaluate acos(-2) or acos(3) results in a domain error, where the return value is QNaN.

Range errors occur when a mathematically valid argument results in a function value that exceeds the range of representable values for the floating-point data type. Attempting to evaluate exp(1000) results in a range error, where the return value is INF.

When domain or range error occurs, the following values are assigned to errno:

- domain error (EDOM): errno = 33
- range error (ERANGE): errno = 34

The following example shows how to read the errno value for an EDOM and ERANGE error.

```
// errno.c

#include <errno.h>
#include <mathimf.h>
#include <stdio.h>

int main(void)
{
   double neg one=-1.0;
   double zero=0.0;

   // The natural log of a negative number is considered a domain
error - EDOM
   printf("log(%e) = %e and errno(EDOM) = %d
\n",neg one,log(neg one),errno);

   // The natural log of zero is considered a range error - ERANGE
   printf("log(%e) = %e and errno(ERANGE) = %d
\n",zero,log(zero),errno);
}
```

The output of errno.c will look like this:

```
log(-1.000000e+00) = nan and errno(EDOM) = 33
log(0.000000e+00) = -inf and errno(ERANGE) = 34
```

For the math functions in this section, a corresponding value for `errno` is listed when applicable.

Other Considerations

Some math functions are inlined automatically by the compiler. The functions actually inlined may vary and may depend on any vectorization or processor-specific compilation options used. For more information, see Criteria for Inline Expansion of Functions.

A change of the default precision control or rounding mode may affect the results returned by some of the mathematical functions. See Floating-point Arithmetic Precision.

It's necessary to include the `-std=c99` compiler option when compiling programs that require support for `_Complex` data types.

Math Function List

The Intel Math Library functions are listed here by function type.

Function Type	Name
Trigonometric Functions	acos
	acosd
	asin
	asind
	atan
	atan2
	atand
	atand2
	cos
	cosd
	cot
	cotd
	sin
	sincos
	sincosd
	sind
	tan
	tand
Hyperbolic Functions	acosh

	asinh
	atanh
	cosh
	sinh
	sinhcosh
	tanh
Exponential Functions	cbrt
	exp
	exp10
	exp2
	expm1
	frexp
	hypot
	invsqrt
	ilogb
	ldexp
	log
	log10
	log1p
	log2
	logb
	pow
	scalb
	scalbln
	scalbn
	sqrt
Special Functions	annuity
	compound
	erf
	erfc
	erfinv
	gamma
	gamma_r
	j0
	j1

	jn
	lgamma
	lgamma_r
	tgamma
	y0
	y1
	yn
Nearest Integer Functions	ceil
	floor
	llrint
	llround
	lrint
	lround
	modf
	nearbyint
	rint
	round
	trunc
Remainder Functions	fmod
	remainder
	remquo
Miscellaneous Functions	copysign
	fabs
	fdim
	finite
	fma
	fmax
	fmin
	fpclassify
	isfinite
	isgreater
	isgreaterequal
	isinf
	isless
	islessequal

	islessgreater
	isnan
	isnormal
	isunordered
	nextafter
	nexttoward
	signbit
	significand
Complex Functions	cabs
	cacos
	cacosh
	carg
	casin
	casinh
	catan
	catanh
	ccos
	cexp
	cexp2
	cimag
	cis
	clog
	clog10
	conj
	ccosh
	cpow
	cproj
	creal
	csin
	csinh
	csqrt
	ctan
	ctanh

Trigonometric Functions

The Intel Math Library supports the following trigonometric functions:

acos

Description: The `acos` function returns the principal value of the inverse cosine of `x` in the range [0, pi] radians for `x` in the interval [-1,1].

errno: `EDOM`, for |x| > 1

Calling interface:
```
double acos(double x);
long double acosl(long double x);
float acosf(float x);
```

acosd

Description: The `acosd` function returns the principal value of the inverse cosine of `x` in the range [0,180] degrees for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1

Calling interface:
```
double acosd(double x);
long double acosdl(long double x);
float acosdf(float x);
```

asin

Description: The `asin` function returns the principal value of the inverse sine of `x` in the range [-pi/2, +pi/2] radians for `x` in the interval [-1,1].

errno: `EDOM`, for |x| > 1

Calling interface:
```
double asin(double x);
long double asinl(long double x);
float asinf(float x);
```

asind

Description: The `asind` function returns the principal value of the inverse sine of `x` in the range [-90,90] degrees for `x` in the interval [-1,1].

errno: `EDOM`, for |x| > 1

Calling interface:
```
double asind(double x);
```

```
long double asindl(long double x);
float asindf(float x);
```

atan

Description: The `atan` function returns the principal value of the inverse tangent of `x` in the range [-pi/2, +pi/2] radians.

Calling interface:
```
double atan(double x);
long double atanl(long double x);
float atanf(float x);
```

atan2

Description: The `atan2` function returns the principal value of the inverse tangent of `y/x` in the range [-pi, +pi] radians.

errno: `EDOM`, for x = 0 and y = 0

Calling interface:
```
double atan2(double y, double x);
long double atan2l(long double y, long double x);
float atan2f(float y, float x);
```

atand

Description: The `atand` function returns the principal value of the inverse tangent of `x` in the range [-90,90] degrees.

Calling interface:
```
double atand(double x);
long double atandl(long double x);
float atandf(float x);
```

atan2d

Description: The `atan2d` function returns the principal value of the inverse tangent of `y/x` in the range [-180, +180] degrees.

errno: `EDOM`, for x = 0 and y = 0.

Calling interface:
```
double atan2d(double x, double y);
long double atan2dl(long double x, long double y);
float atan2df(float x, float y);
```

cos

Description: The `cos` function returns the cosine of `x` measured in radians. This function may be inlined by the Itanium® compiler.

Calling interface:
```
double cos(double x);
long double cosl(long double x);
float cosf(float x);
```

cosd

Description: The `cosd` function returns the cosine of `x` measured in degrees.

Calling interface:
```
double cosd(double x);
long double cosdl(long double x);
float cosdf(float x);
```

cot

Description: The `cot` function returns the cotangent of `x` measured in radians.

errno: ERANGE, for overflow conditions at x = 0.

Calling interface:
```
double cot(double x);
long double cotl(long double x);
float cotf(float x);
```

cotd

Description: The `cotd` function returns the cotangent of `x` measured in degrees.

errno: ERANGE, for overflow conditions at x = 0.

Calling interface:
```
double cotd(double x);
long double cotdl(long double x);
float cotdf(float x);
```

sin

Description: The `sin` function returns the sine of `x` measured in radians. This function may be inlined by the Itanium® compiler.

Calling interface:
```
double sin(double x);
long double sinl(long double x);
float sinf(float x);
```

sincos

Description: The `sincos` function returns both the sine and cosine of `x` measured in radians. This function may be inlined by the Itanium® compiler.

Calling interface:
```
void sincos(double x, double *sinval, double *cosval);
void sincosl(long double x, long double *sinval, long double *cosval);
void sincosf(float x, float *sinval, float *cosval);
```

sincosd

Description: The `sincosd` function returns both the sine and cosine of `x` measured in degrees.

Calling interface:
```
void sincosd(double x, double *sinval, double *cosval);
void sincosdl(long double x, long double *sinval, long double *cosval);
void sincosdf(float x, float *sinval, float *cosval);
```

sind

Description: The `sind` function computes the sine of `x` measured in degrees.

Calling interface:
```
double sind(double x);
long double sindl(long double x);
float sindf(float x);
```

tan

Description: The `tan` function returns the tangent of `x` measured in radians.

Calling interface:
```
double tan(double x);
long double tanl(long double x);
float tanf(float x);
```

tand

Description: The `tand` function returns the tangent of `x` measured in degrees.

errno: `ERANGE`, for overflow conditions

Calling interface:
```
double tand(double x);
long double tandl(long double x);
float tandf(float x);
```

Hyperbolic Functions

The Intel Math Library supports the following hyperbolic functions:

acosh

Description: The `acosh` function returns the inverse hyperbolic cosine of `x`.

errno: `EDOM`, for x < 1

Calling interface:
```
double acosh(double x);
long double acoshl(long double x);
float acoshf(float x);
```

asinh

Description: The `asinh` function returns the inverse hyperbolic sine of `x`.

Calling interface:
```
double asinh(double x);
long double asinhl(long double x);
float asinhf(float x);
```

atanh

Description: The `atanh` function returns the inverse hyperbolic tangent of `x`.

errno: `EDOM`, for x > 1
errno: `ERANGE`, for x = 1

Calling interface:
```
double atanh(double x);
long double atanhl(long double x);
float atanhf(float x);
```

cosh

Description: The `cosh` function returns the hyperbolic cosine of x, $(e^x + e^{-x})/2$.

errno: `ERANGE`, for overflow conditions

Calling interface:
```
double cosh(double x);
long double coshl(long double x);
float coshf(float x);
```

sinh

Description: The `sinh` function returns the hyperbolic sine of `x`, $(e^x - e^{-x})/2$.

errno: `ERANGE`, for overflow conditions

Calling interface:
```
double sinh(double x);
long double sinhl(long double x);
float sinhf(float x);
```

sinhcosh

Description: The `sinhcosh` function returns both the hyperbolic sine and hyperbolic cosine of `x`.

errno: `ERANGE`, for overflow conditions

Calling interface:
```
void sinhcosh(double x, float *sinval, float *cosval);
void sinhcoshl(long double x, long double *sinval, long double *cosval);
void sinhcoshf(float x, float *sinval, float *cosval);
```

tanh

Description: The `tanh` function returns the hyperbolic tangent of `x`, $(e^x - e^{-x}) / (e^x + e^{-x})$.

Calling interface:
```
double tanh(double x);
long double tanhl(long double x);
float tanhf(float x);
```

Exponential Functions

The Intel Math Library supports the following exponential functions:

cbrt

Description: The `cbrt` function returns the cube root of `x`.

Calling interface:
```
double cbrt(double x);
long double cbrtl(long double x);
float cbrtf(float x);
```

exp

Description: The `exp` function returns `e` raised to the `x` power, e^x. This function may be inlined by the Itanium® compiler.

errno: `ERANGE`, for underflow and overflow conditions

Calling interface:
```
double exp(double x);
long double expl(long double x);
float expf(float x);
```

exp10

Description: The `exp10` function returns 10 raised to the `x` power, 10^x.

errno: `ERANGE`, for underflow and overflow conditions

Calling interface:
```
double exp10(double x);
long double exp10l(long double x);
float exp10f(float x);
```

exp2

Description: The `exp2` function returns 2 raised to the `x` power, 2^x.

errno: `ERANGE`, for underflow and overflow conditions

Calling interface:
```
double exp2(double x);
long double exp2l(long double x);
float exp2f(float x);
```

expm1

Description: The `expm1` function returns `e` raised to the `x` power minus 1, ex - 1.

errno: `ERANGE`, for overflow conditions

Calling interface:
```
double expm1(double x);
long double expm1l(long double x);
float expm1f(float x);
```

frexp

Description: The `frexp` function converts a floating-point number `x` into signed normalized fraction in [1/2, 1) multiplied by an integral power of two. The signed normalized fraction is returned, and the integer exponent stored at location `exp`.

Calling interface:
```
double frexp(double x, int *exp);
long double frexpl(long double x, int *exp);
float frexpf(float x, int *exp);
```

hypot

Description: The `hypot` function returns the square root of $(x^2 + y^2)$.

errno: `ERANGE`, for overflow conditions

Calling interface:
```
double hypot(double x, double y);
long double hypotl(long double x, long double y);
float hypotf(float x, float y);
```

ilogb

Description: The `ilogb` function returns the exponent of `x` base two as a signed `int` value.

errno: `ERANGE`, for x = 0

Calling interface:
```
int ilogb(double x);
int ilogbl(long double x);
int ilogbf(float x);
```

invsqrt

Description: The `invsqrt` function returns the inverse square root. This function may be inlined by the Itanium® compiler.

Calling interface:
```
double invsqrt(double x);
long double invsqrtl(long double x);
float invsqrtf(float x);
```

ldexp

Description: The `ldexp` function returns $x*2^{exp}$, where `exp` is an integer value.

errno: `ERANGE`, for underflow and overflow conditions

Calling interface:
```
double ldexp(double x, int exp);
long double ldexpl(long double x, int exp);
float ldexpf(float x, int exp);
```

log

Description: The `log` function returns the natural log of x, ln(x). This function may be inlined by the Itanium® compiler.

errno: `EDOM`, for x < 0
errno: `ERANGE`, for x = 0

Calling interface:
```
double log(double x);
long double logl(long double x);
float logf(float x);
```

log10

Description: The `log10` function returns the base-10 log of x, $\log_{10}(x)$. This function may be inlined by the Itanium® compiler.

errno: `EDOM`, for x < 0
errno: `ERANGE`, for x = 0

Calling interface:
```
double log10(double x);
long double log10l(long double x);
float log10f(float x);
```

log1p

Description: The `log1p` function returns the natural log of (`x+1`), ln(x + 1).

errno: `EDOM`, for x < -1
errno: `ERANGE`, for x = -1

Calling interface:
```
double log1p(double x);
long double log1pl(long double x);
float log1pf(float x);
```

log2

Description: The `log2` function returns the base-2 log of `x`, $\log_2(x)$.

errno: `EDOM`, for x < 0
errno: `ERANGE`, for x = 0

Calling interface:
```
double log2(double x);
long double log2l(long double x);
float log2f(float x);
```

logb

Description: The `logb` function returns the signed exponent of `x`.

errno: `EDOM`, for x = 0

Calling interface:
```
double logb(double x);
long double logbl(long double x);
float logbf(float x);
```

pow

Description: The `pow` function returns `x` raised to the power of `y`, x^y. This function may be inlined by the Itanium® compiler.

errno: `EDOM`, for x = 0 and y < 0
errno: `EDOM`, for x < 0 and y is a non-integer
errno: `ERANGE`, for overflow and underflow conditions

Calling interface:
```
double pow(double x, double y);
long double powl(double x, double y);
float powf(float x, float y);
```

scalb

Description: The `scalb` function returns $x*2^y$, where `y` is a floating-point value.

errno: `ERANGE`, for underflow and overflow conditions

Calling interface:
```
double scalb(double x, double y);
long double scalbl(long double x, long double y);
float scalbf(float x, float y);
```

scalbn

Description: The `scalbn` function returns $x*2^n$, where `n` is an integer value.

errno: `ERANGE`, for underflow and overflow conditions

Calling interface:
```
double scalbn(double x, int n);
long double scalbnl (long double x, int n);
float scalbnf(float x, int n);
```

scalbln

Description: The `scalbln` function returns $x*2^n$, where `n` is a long integer value.

errno: `ERANGE`, for underflow and overflow conditions

Calling interface:
```
double scalbln(double x, long int n);
long double scalblnl (long double x, long int n);
float scalblnf(float x, long int n);
```

sqrt

Description: The `sqrt` function returns the correctly rounded square root.

errno: `EDOM`, for x < 0

Calling interface:
```
double sqrt(double x);
long double sqrtl(long double x);
float sqrtf(float x);
```

Special Functions

The Intel Math Library supports the following special functions:

annuity

Description: The `annuity` function computes the present value factor for an annuity, $(1 - (1+x)^{(-y)}) / x$, where `x` is a rate and `y` is a period.

errno: `ERANGE`, for underflow and overflow conditions

Calling interface:
```
double annuity(double x, double y);
long double annuityl(long double x, long double y);
float annuityf(float x, float y);
```

compound

Description: The `compound` function computes the compound interest factor, $(1+x)^y$, where `x` is a rate and `y` is a period.

errno: `ERANGE`, for underflow and overflow conditions

Calling interface:
```
double compound(double x, double y);
long double compoundl(long double x, long double y);
float compoundf(float x, float y);
```

erf

Description: The `erf` function returns the error function value.

Calling interface:
```
double erf(double x);
long double erfl(long double x);
float erff(float x);
```

erfc

Description: The `erfc` function returns the complementary error function value.

errno: EDOM, for finite or infinite |x| > 1

Calling interface:
```
double erfc(double x);
long double erfcl(long double x);
float erfcf(float x);
```

erfinv

Description: The `erfinv` function returns the value of the inverse error function of x.

errno: EDOM, for finite or infinite |x| > 1

Calling interface:
```
double erfinv(double x);
long double erfinvl(long double x);
float erfinvf(float x);
```

gamma

Description: The `gamma` function returns the value of the logarithm of the absolute value of gamma.

errno: `ERANGE, for overflow conditions when x is a negative integer.`

Calling interface:
```
double gamma(double x);
long double gammal(long double x);
float gammaf(float x);
```

gamma_r

Description: The `gamma_r` function returns the value of the logarithm of the absolute value of gamma. The sign of the gamma function is returned in the integer signgam.

Calling interface:
```
double gamma_r(double x, int *signgam);
long double gammal_r(long double x, int *signgam);
float gammaf_r(float x, int *signgam);
```

j0

Description: Computes the Bessel function (of the first kind) of x with order 0.

Calling interface:
```
double j0(double x);
long double j0l(long double x);
float j0f(float x);
```

j1

Description: Computes the Bessel function (of the first kind) of x with order 1.

Calling interface:
```
double j1(double x);
long double j1l(long double x);
float j1f(float x);
```

jn

Description: Computes the Bessel function (of the first kind) of x with order n.

Calling interface:
```
double jn(int n, double x);
long double jnl(int n, long double x);
float jnf(int n, float x);
```

lgamma

Description: The `lgamma` function returns the value of the logarithm of the absolute value of gamma.

errno: `ERANGE, for overflow conditions, x=0 or negative integers.`

Calling interface:
```
double lgamma(double x);
long double lgammal(long double x);
float lgammaf(float x);
```

lgamma_r

Description: The `lgamma_r` function returns the value of the logarithm of the absolute value of gamma. The sign of the gamma function is returned in the integer signgam.

errno: ERANGE, for overflow conditions, x=0 or negative integers.

Calling interface:
```
double lgamma_r(double x, int *signgam);
long double lgammal_r(long double x, int *signgam);
float lgammaf_r(float x, int *signgam);
```

tgamma

Description: The tgamma function computes the gamma function of x.

errno: EDOM, for x=0 or negative integers.

Calling interface:
```
double tgamma(double x);
long double tgammal(long double x);
float tgammaf(float x);
```

y0

Description: Computes the Bessel function (of the second kind) of x with order 0.

errno: EDOM, for x <= 0

Calling interface:
```
double y0(double x);
long double y0l(long double x);
float y0f(float x);
```

y1

Description: Computes the Bessel function (of the second kind) of x with order 1.

errno: EDOM, for x <= 0

Calling interface:
```
double y1(double x);
long double y1l(long double x);
float y1f(float x);
```

yn

Description: Computes the Bessel function (of the second kind) of x with order n.

errno: EDOM, for x <= 0

Calling interface:
```
double yn(int n, double x);
long double ynl(int n, long double x);
float ynf(int n, float x);
```

Nearest Integer Functions

The Intel Math Library supports the following nearest integer functions:

ceil

Description: The `ceil` function returns the smallest integral value not less than `x` as a floating-point number. This function may be inlined by the Itanium® compiler.

Calling interface:
```
double ceil(double x);
long double ceill(long double x);
float ceilf(float x);
```

floor

Description: The `floor` function returns the largest integral value not greater than `x` as a floating-point value. This function may be inlined by the Itanium® compiler.

Calling interface:
```
double floor(double x);
long double floorl(long double x);
float floorf(float x);
```

llrint

Description: The `llrint` function returns the rounded integer value (according to the current rounding direction) as a `long long int`.

errno: ERANGE, for values too large

Calling interface:
```
long long int llrint(double x);
long long int llrintl(long double x);
long long int llrintf(float x);
```

llround

Description: The `llround` function returns the rounded integer value as a `long long int`.

errno: ERANGE, for values too large

Calling interface:
```
long long int llround(double x);
long long int llroundl(long double x);
long long int llroundf(float x);
```

lrint

Description: The `lrint` function returns the rounded integer value (according to the current rounding direction) as a `long int`.

errno: ERANGE, for values too large

Calling interface:
```
long int lrint(double x);
long int lrintl(long double x);
long int lrintf(float x);
```

lround

Description: The `lround` function returns the rounded integer value as a `long int`. Halfway cases are rounded away from zero.

errno: ERANGE, for values too large

Calling interface:
```
long int lround(double x);
long int lroundl(long double x);
long int lroundf(float x);
```

modf

Description: The `modf` function returns the value of the signed fractional part of `x` and stores the integral part at `*iptr` as a floating-point number.

Calling interface:
```
double modf(double x, double *iptr);
long double modfl(long double x, long double *iptr);
float modff(float x, float *iptr);
```

nearbyint

Description: The `nearbyint` function returns the rounded integral value as a floating-point number, using the current rounding direction.

Calling interface:
```
double nearbyint(double x);
long double nearbyintl(long double x);
float nearbyintf(float x);
```

rint

Description: The `rint` function returns the rounded integral value as a floating-point number, using the current rounding direction.

Calling interface:
```
double rint(double x);
long double rintl(long double x);
float rintf(float x);
```

round

Description: The `round` function returns the nearest integral value as a floating-point number. Halfway cases are rounded away from zero.

Calling interface:
```
double round(double x);
long double roundl(long double x);
float roundf(float x);
```

trunc

Description: The `trunc` function returns the truncated integral value as a floating-point number.

Calling interface:
```
double trunc(double x);
long double truncl(long double x);
float truncf(float x);
```

Remainder Functions

The Intel Math Library supports the following remainder functions:

fmod

Description: The `fmod` function returns the value `x-n*y` for integer `n` such that if `y` is nonzero, the result has the same sign as `x` and magnitude less than the magnitude of `y`.

errno: EDOM, for y = 0

Calling interface:
```
double fmod(double x, double y);
long double fmodl(long double x, long double y);
float fmodf(float x, float y);
```

remainder

Description: The `remainder` function returns the value of `x REM y` as required by the IEEE standard.

Calling interface:
```
double remainder(double x, double y);
```

```
long double remainderl(long double x, long double y);
float remainderf(float x, float y);
```

remquo

Description: The `remquo` function returns the value of `x REM y`. In the object pointed to by `quo` the function stores a value whose sign is the sign of `x/y` and whose magnitude is congruent modulo `2`n of the integral quotient of x/y. N is an implementation-defined integer. For systems based on IA-64 architecture, N is equal to 24. For all other systems, N is equal to 31.

Calling interface:

```
double remquo(double x, double y, int *quo);
long double remquol(long double x, long double y, int *quo);
float remquof(float x, float y, int *quo);
```

Miscellaneous Functions

The Intel Math Library supports the following miscellaneous functions:

copysign

Description: The `copysign` function returns the value with the magnitude of `x` and the sign of `y`.

Calling interface:

```
double copysign(double x, double y);
long double copysignl(long double x, long double y);
float copysignf(float x, float y);
```

fabs

Description: The `fabs` function returns the absolute value of `x`.

Calling interface:

```
double fabs(double x);
long double fabsl(long double x);
float fabsf(float x);
```

fdim

Description: The `fdim` function returns the positive difference value, `x-y` (for `x` > `y`) or +0 (for `x` ≤ `y`).

errno: ERANGE, for values too large

Calling interface:

```
double fdim(double x, double y);
long double fdiml(long double x, long double y);
float fdimf(float x, float y);
```

finite

Description: The `finite` function returns 1 if `x` is not a `NaN` or +/- infinity. Otherwise 0 is returned.

Calling interface:
```
int finite(double x);
int finitel(long double x);
int finitef(float x);
```

fma

Description: The `fma` functions return `(x*y)+z`.

Calling interface:
```
double fma(double x, double y, double z);
long double fmal(long double x, long double y, long double z);
float fmaf(float x, float y, float double z);
```

fmax

Description: The `fmax` function returns the maximum numeric value of its arguments.

Calling interface:
```
double fmax(double x, double y);
long double fmaxl(long double x, long double y);
float fmaxf(float x, float y);
```

fmin

Description: The `fmin` function returns the minimum numeric value of its arguments.

Calling interface:
```
double fmin(double x, double y);
long double fminl(long double x, long double y);
float fminf(float x, float y);
```

fpclassify

Description: The `fpclassify` function returns the value of the number classification macro appropriate to the value of its argument.

Return Value
0 (NaN)
1 (Infinity)
2 (Zero)

3 (Subnormal)
4 (Finite)

Calling interface:
```
double fpclassify(double x);
long double fpclassifyl(long double x);
float fpclassifyf(float x);
```

isfinite

Description: The `isfinite` function returns 1 if `x` is not a `NaN` or +/- infinity. Otherwise 0 is returned.

Calling interface:
```
int isfinite(double x);
int isfinitel(long double x);
int isfinitef(float x);
```

isgreater

Description: The `isgreater` function returns 1 if `x` is greater than `y`. This function does not raise the invalid floating-point exception.

Calling interface:
```
int isgreater(double x, double y);
int isgreaterl(long double x, long double y);
int isgreaterf(float x, float y);
```

isgreaterequal

Description: The `isgreaterequal` function returns 1 if `x` is greater than or equal to `y`. This function does not raise the invalid floating-point exception.

Calling interface:
```
int isgreaterequal(double x, double y);
int isgreaterequall(long double x, long double y);
int isgreaterequalf(float x, float y);
```

isinf

Description: The `isinf` function returns a non-zero value if and only if its argument has an infinite value.

Calling interface:
```
int isinf(double x);
int isinfl(long double x);
int isinff(float x);
```

isless

Description: The `isless` function returns 1 if `x` is less than `y`. This function does not raise the invalid floating-point exception.

Calling interface:
```
int isless(double x, double y);
int islessl(long double x, long double y);
int islessf(float x, float y);
```

islessequal

Description: The `islessequal` function returns 1 if `x` is less than or equal to `y`. This function does not raise the invalid floating-point exception.

Calling interface:
```
int islessequal(double x, double y);
int islessequall(long double x, long double y);
int islessequalf(float x, float y);
```

islessgreater

Description: The `islessgreater` function returns 1 if `x` is less than or greater than `y`. This function does not raise the invalid floating-point exception.

Calling interface:
```
int islessgreater(double x, double y);
int islessgreaterl(long double x, long double y);
int islessgreaterf(float x, float y);
```

isnan

Description: The `isnan` function returns a non-zero value if and only if `x` has a `NaN` value.

Calling interface:
```
int isnan(double x);
int isnanl(long double x);
int isnanf(float x);
```

isnormal

Description: The `isnormal` function returns a non-zero value if and only if `x` is normal.

Calling interface:
```
int isnormal(double x);
int isnormall(long double x);
int isnormalf(float x);
```

isunordered

Description: The `isunordered` function returns 1 if either `x` or `y` is a NaN. This function does not raise the invalid floating-point exception.

Calling interface:
```
int isunordered(double x, double y);
int isunorderedl(long double x, long double y);
int isunorderedf(float x, float y);
```

nextafter

Description: The `nextafter` function returns the next representable value in the specified format after `x` in the direction of `y`.

errno: ERANGE, for overflow and underflow conditions

Calling interface:
```
double nextafter(double x, double y);
long double nextafterl(long double x, long double y);
float nextafterf(float x, float y);
```

nexttoward

Description: The `nexttoward` function returns the next representable value in the specified format after `x` in the direction of `y`. If `x` equals `y`, then the function returns `y` converted to the type of the function.

errno: ERANGE, for overflow and underflow conditions

Calling interface:
```
double nexttoward(double x, long double y);
long double nexttowardl(long double x, long double y);
float nexttowardf(float x, long double y);
```

signbit

Description: The `signbit` function returns a non-zero value if and only if the sign of `x` is negative.

Calling interface:
```
int signbit(double x);
int signbitl(long double x);
int signbitf(float x);
```

significand

Description: The `significand` function returns the significand of `x` in the interval [1,2). For `x` equal to zero, NaN, or +/- infinity, the original `x` is returned.

Calling interface:
```
double significand(double x);
long double significandl(long double x);
float significandf(float x);
```

Complex Functions

The Intel Math Library supports the following complex functions:

cabs

Description: The `cabs` function returns the complex absolute value of `z`.

Calling interface:
```
double cabs(double _Complex z);
long double cabsl(long double _Complex z);
float cabsf(float _Complex z);
```

cacos

Description: The `cacos` function returns the complex inverse cosine of `z`.

Calling interface:
```
double _Complex cacos(double _Complex z);
long double _Complex cacosl(long double _Complex z);
float _Complex cacosf(float _Complex z);
```

cacosh

Description: The `cacosh` function returns the complex inverse hyperbolic cosine of `z`.

Calling interface:
```
double _Complex cacosh(double _Complex z);
long double _Complex cacoshl(long double _Complex z);
float _Complex cacoshf(float _Complex z);
```

carg

Description: The `carg` function returns the value of the argument in the interval [-pi, +pi].

Calling interface:
```
double carg(double _Complex z);
long double cargl(long double _Complex z);
float cargf(float _Complex z);
```

casin

Description: The `casin` function returns the complex inverse sine of `z`.

Calling interface:
```
double _Complex casin(double _Complex z);
long double _Complex casinl(long double _Complex z);
float _Complex casinf(float _Complex z);
```

casinh

Description: The `casinh` function returns the complex inverse hyperbolic sine of `z`.

Calling interface:
```
double _Complex casinh(double _Complex z);
long double _Complex casinhl(long double _Complex z);
float _Complex casinhf(float _Complex z);
```

catan

Description: The `catan` function returns the complex inverse tangent of `z`.

Calling interface:
```
double _Complex catan(double _Complex z);
long double _Complex catanl(long double _Complex z);
float _Complex catanf(float _Complex z);
```

catanh

Description: The `catanh` function returns the complex inverse hyperbolic tangent of `z`.

Calling interface:
```
double _Complex catanh(double _Complex z);
long double _Complex catanhl(long double _Complex z);
float _Complex catanhf(float _Complex z);
```

ccos

Description: The `ccos` function returns the complex cosine of `z`.

Calling interface:
```
double _Complex ccos(double _Complex z);
long double _Complex ccosl(long double _Complex z);
float _Complex ccosf(float _Complex z);
```

ccosh

Description: The `ccosh` function returns the complex hyperbolic cosine of `z`.

Calling interface:
```
double _Complex ccosh(double _Complex z);
long double _Complex ccoshl(long double _Complex z);
float _Complex ccoshf(float _Complex z);
```

cexp

Description: The `cexp` function returns e^z (`e` raised to the power z).

Calling interface:
```
double _Complex cexp(double _Complex z);
long double _Complex cexpl(long double _Complex z);
float _Complex cexpf(float _Complex z);
```

cexp2

Description: The `cexp` function returns 2^z (2 raised to the power z).

Calling interface:
```
double _Complex cexp2(double _Complex z);
long double _Complex cexp2l(long double _Complex z);
float _Complex cexp2f(float _Complex z);
```

cexp10

Description: The `cexp10` function returns 10^z (10 raised to the power z).

Calling interface:
```
double _Complex cexp10(double _Complex z);
long double _Complex cexp10l(long double _Complex z);
float _Complex cexp10f(float _Complex z);
```

cimag

Description: The `cimag` function returns the imaginary part value of `z`.

Calling interface:
```
double cimag(double _Complex z);
long double cimagl(long double _Complex z);
float cimagf(float _Complex z);
```

cis

Description: The `cis` function returns the cosine and sine (as a complex value) of `z` measured in radians.

Calling interface:
```
double _Complex cis(double x);
long double _Complex cisl(long double z);
float _Complex cisf(float z);
```

cisd

Description: The `cisd` function returns the cosine and sine (as a complex value) of `z` measured in degrees.

Calling interface:
```
double _Complex cisd(double x);
long double _Complex cisdl(long double z);
float _Complex cisdf(float z);
```

clog

Description: The `clog` function returns the complex natural logarithm of `z`.

Calling interface:
```
double _Complex clog(double _Complex z);
long double _Complex clogl(long double _Complex z);
float _Complex clogf(float _Complex z);
```

clog2

Description: The `clog2` function returns the complex logarithm base 2 of `z`.

Calling interface:
```
double _Complex clog2(double _Complex z);
long double _Complex clog2l(long double _Complex z);
float _Complex clog2f(float _Complex z);
```

clog10

Description: The `clog10` function returns the complex logarithm base 10 of `z`.

Calling interface:
```
double _Complex clog10(double _Complex z);
long double _Complex clog10l(long double _Complex z);
float _Complex clog10f(float _Complex z);
```

conj

Description: The `conj` function returns the complex conjugate of `z` by reversing the sign of its imaginary part.

Calling interface:
```
double _Complex conj(double _Complex z);
long double _Complex conjl(long double _Complex z);
float _Complex conjf(float _Complex z);
```

cpow

Description: The `cpow` function returns the complex power function, x^y .

Calling interface:
```
double _Complex cpow(double _Complex x, double _Complex y);
long double _Complex cpowl(long double _Complex x, long double _Complex
y);
float _Complex cpowf(float _Complex x, float _Complex y);
```

cproj

Description: The `cproj` function returns a projection of `z` onto the Riemann sphere.

Calling interface:
```
double _Complex cproj(double _Complex z);
long double _Complex cprojl(long double _Complex z);
float _Complex cprojf(float _Complex z);
```

creal

Description: The `creal` function returns the real part of `z`.

Calling interface:
```
double creal(double _Complex z);
long double creall(long double _Complex z);
float crealf(float _Complex z);
```

csin

Description: The `csin` function returns the complex sine of `z`.

Calling interface:
```
double _Complex csin(double _Complex z);
long double _Complex csinl(long double _Complex z);
float _Complex csinf(float _Complex z);
```

csinh

Description: The `csinh` function returns the complex hyperbolic sine of `z`.

Calling interface:
```
double _Complex csinh(double _Complex z);
long double _Complex csinhl(long double _Complex z);
float _Complex csinhf(float _Complex z);
```

csqrt

Description: The `csqrt` function returns the complex square root of `z`.

Calling interface:
```
double _Complex csqrt(double _Complex z);
long double _Complex csqrtl(long double _Complex z);
float _Complex csqrtf(float _Complex z);
```

ctan

Description: The `ctan` function returns the complex tangent of `z`.

Calling interface:

```
double _Complex ctan(double _Complex z);
long double _Complex ctanl(long double _Complex z);
float _Complex ctanf(float _Complex z);
```

ctanh

Description: The `ctanh` function returns the complex hyperbolic tangent of `z`.

Calling interface:

```
double _Complex ctanh(double _Complex z);
long double _Complex ctanhl(long double _Complex z);
float _Complex ctanhf(float _Complex z);
```

C99 Macros

The Intel Math Library and `mathimf.h` header file support the following C99 macros:

```
int fpclassify(x);

int isfinite(x);

int isgreater(x, y);

int isgreaterequal(x, y);

int isinf(x);

int isless(x, y);

int islessequal(x, y);

int islessgreater(x, y);

int isnan(x);

int isnormal(x);

int isunordered(x, y);

int signbit(x);
```

See Also

- Miscellaneous Functions.

Intel C++ Class Libraries

The Intel C++ Class Libraries enable Single-Instruction, Multiple-Data (SIMD) operations. The principle of SIMD operations is to exploit microprocessor architecture through parallel processing. The effect of parallel processing is increased data throughput using fewer clock cycles. The objective is to improve application performance of complex and computation-intensive audio, video, and graphical data bit streams.

Hardware and Software Requirements

The Intel C++ Class Libraries are functions abstracted from the instruction extensions available on Intel processors as specified in the table that follows:

Processor Requirements for Use of Class Libraries

Header File	Extension Set	Available on These Processors
`ivec.h`	MMX™ technology	Intel® Pentium® processor with MMX™ technology, Intel® Pentium® II processor, Intel® Pentium® III processor, Intel® Pentium® 4 processor, Intel® Xeon® processor, and Intel® Itanium® processor
`fvec.h`	Streaming SIMD Extensions	Intel Pentium III processor, Intel Pentium 4 processor, Intel Xeon processor, and Intel Itanium processor
`dvec.h`	Streaming SIMD Extensions 2	Intel Pentium 4 processor and Intel Xeon processors

About the Classes

The Intel C++ Class Libraries for SIMD Operations include:

- Integer vector (`Ivec`) classes
- Floating-point vector (`Fvec`) classes

You can find the definitions for these operations in three header files: `ivec.h`, `fvec.h`, and `dvec.h`. The classes themselves are not partitioned like this. The classes are named according to the underlying type of operation. The header files are partitioned according to architecture:

- `ivec.h` is specific to architectures with MMX(TM) technology
- `fvec.h` is specific to architectures with Streaming SIMD Extensions
- `dvec.h` is specific to architectures with Streaming SIMD Extensions 2

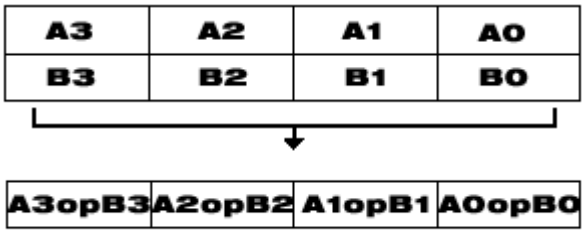
Streaming SIMD Extensions 2 intrinsics cannot be used on IA-64 architecture based systems. The `mmclass.h` header file includes the classes that are usable on the IA-64 architecture.

This documentation is intended for programmers writing code for the Intel architecture, particularly code that would benefit from the use of SIMD instructions. You should be familiar with C++ and the use of C++ classes.

Details about the Libraries

The Intel C++ Class Libraries for SIMD Operations provide a convenient interface to access the underlying instructions for processors as specified in Processor Requirements for Use of Class Libraries. These processor-instruction extensions enable parallel processing using the single instruction-multiple data (SIMD) technique as illustrated in the following figure.

SIMD Data Flow



Performing four operations with a single instruction improves efficiency by a factor of four for that particular instruction.

These new processor instructions can be implemented using assembly inlining, intrinsics, or the C++ SIMD classes. Compare the coding required to add four 32-bit floating-point values, using each of the available interfaces:

Comparison Between Inlining, Intrinsics and Class Libraries

Assembly Inlining	Intrinsics	SIMD Class Libraries
`... __m128 a,b,c; __asm{ movaps xmm0,b movaps xmm1,c addps xmm0,xmm1 movaps a, xmm0 } ...`	`#include <mmintrin.h> ... __m128 a,b,c; a = _mm_add_ps(b,c); ...`	`#include <fvec.h> ... F32vec4 a,b,c; a = b +c; ...`

This table shows an addition of two single-precision floating-point values using assembly inlining, intrinsics, and the libraries. You can see how much easier it is to code with the Intel C++ SIMD Class Libraries. Besides using fewer keystrokes and fewer lines of code, the notation is like the standard notation in C++, making it much easier to implement over other methods.

C++ Classes and SIMD Operations

The use of C++ classes for SIMD operations is based on the concept of operating on arrays, or vectors of data, in parallel. Consider the addition of two vectors, `A` and `B`,

where each vector contains four elements. Using the integer vector (`Ivec`) class, the elements `A[i]` and `B[i]` from each array are summed as shown in the following example.

Typical Method of Adding Elements Using a Loop

```
short a[4], b[4], c[4];
for (i=0; i<4; i++) /* needs four iterations */
c[i] = a[i] + b[i]; /* returns c[0], c[1], c[2], c[3] */
```

The following example shows the same results using one operation with `Ivec` Classes.

SIMD Method of Adding Elements Using Ivec Classes

```
sIs16vec4 ivecA, ivecB, ivec C; /*needs one iteration */
ivecC = ivecA + ivecB; /*returns ivecC0, ivecC1, ivecC2, ivecC3 */
```

Available Classes

The Intel C++ SIMD classes provide parallelism, which is not easily implemented using typical mechanisms of C++. The following table shows how the Intel C++ SIMD classes use the classes and libraries.

SIMD Vector Classes

Instruction Set	Class	Signedness	Data Type	Size	Elements	Header File
MMX(TM) technology	I64vec1	unspecified	__m64	64	1	ivec.h
	I32vec2	unspecified	int	32	2	ivec.h
	Is32vec2	signed	int	32	2	ivec.h
	Iu32vec2	unsigned	int	32	2	ivec.h
	I16vec4	unspecified	short	16	4	ivec.h
	Is16vec4	signed	short	16	4	ivec.h
	Iu16vec4	unsigned	short	16	4	ivec.h
	I8vec8	unspecified	char	8	8	ivec.h
	Is8vec8	signed	char	8	8	ivec.h
	Iu8vec8	unsigned	char	8	8	ivec.h
Streaming SIMD Extensions	F32vec4	signed	float	32	4	fvec.h
	F32vec1	signed	float	32	1	fvec.h
Streaming SIMD Extensions 2	F64vec2	signed	double	64	2	dvec.h

	I128vec1	unspecified	__m128i	128	1	dvec.h
	I64vec2	unspecified	long int	64	4	dvec.h
	Is64vec2	signed	long int	64	4	dvec.h
	Iu64vec2	unsigned	long int	32	4	dvec.h
	I32vec4	unspecified	int	32	4	dvec.h
	Is32vec4	signed	int	32	4	dvec.h
	Iu32vec4	unsigned	int	32	4	dvec.h
	I16vec8	unspecified	int	16	8	dvec.h
	Is16vec8	signed	int	16	8	dvec.h
	Iu16vec8	unsigned	int	16	8	dvec.h
	I8vec16	unspecified	char	8	16	dvec.h
	Is8vec16	signed	char	8	16	dvec.h
	Iu8vec16	unsigned	char	8	16	dvec.h

Most classes contain similar functionality for all data types and are represented by all available intrinsics. However, some capabilities do not translate from one data type to another without suffering from poor performance, and are therefore excluded from individual classes.

Note

Intrinsics that take immediate values and cannot be expressed easily in classes are not implemented.
(For example, `_mm_shuffle_ps`, `_mm_shuffle_pi16`, `_mm_extract_pi16`, `_mm_insert_pi16`).

Access to Classes Using Header Files

The required class header files are installed in the include directory with the Intel® C++ Compiler. To enable the classes, use the `#include` directive in your program file as shown in the table that follows.

Include Directives for Enabling Classes

Instruction Set Extension	Include Directive
MMX Technology	`#include <ivec.h>`
Streaming SIMD Extensions	`#include <fvec.h>`
Streaming SIMD Extensions 2	`#include <dvec.h>`

Each succeeding file from the top down includes the preceding class. You only need to include `fvec.h` if you want to use both the `Ivec` and `Fvec` classes. Similarly, to use all the classes including those for the Streaming SIMD Extensions 2, you need only to include the `dvec.h` file.

Usage Precautions

When using the C++ classes, you should follow some general guidelines. More detailed usage rules for each class are listed in Integer Vector Classes, and Floating-point Vector Classes.

Clear MMX Registers

If you use both the `Ivec` and `Fvec` classes at the same time, your program could mix MMX instructions, called by `Ivec` classes, with Intel x87 architecture floating-point instructions, called by `Fvec` classes. Floating-point instructions exist in the following `Fvec` functions:

- `fvec` constructors
- debug functions (`cout` and element access)
- `rsqrt_nr`

Note

MMX registers are aliased on the floating-point registers, so you should clear the MMX state with the EMMS instruction intrinsic before issuing an x87 floating-point instruction, as in the following example.

`ivecA = ivecA & ivecB;`	Ivec logical operation that uses MMX instructions
`empty ();`	clear state
`cout << f32vec4a;`	F32vec4 operation that uses x87 floating-point instructions

Caution

Failure to clear the MMX registers can result in incorrect execution or poor performance due to an incorrect register state.

Follow EMMS Instruction Guidelines

Intel strongly recommends that you follow the guidelines for using the EMMS instruction. Refer to this topic before coding with the `Ivec` classes.

Capabilities

The fundamental capabilities of each C++ SIMD class include:

- computation
- horizontal data motion
- branch compression/elimination
- caching hints

Understanding each of these capabilities and how they interact is crucial to achieving desired results.

Computation

The SIMD C++ classes contain vertical operator support for most arithmetic operations, including shifting and saturation.

Computation operations include: `+`, `-`, `*`, `/`, reciprocal (`rcp` and `rcp_nr`), square root (`sqrt`), reciprocal square root (`rsqrt` and `rsqrt_nr`).

Operations `rcp` and `rsqrt` are new approximating instructions with very short latencies that produce results with at least 12 bits of accuracy. Operations `rcp_nr` and `rsqrt_nr` use software refining techniques to enhance the accuracy of the approximations, with a minimal impact on performance. (The "`nr`" stands for Newton-Raphson, a mathematical technique for improving performance using an approximate result.)

Horizontal Data Support

The C++ SIMD classes provide horizontal support for some arithmetic operations. The term "horizontal" indicates computation across the elements of one vector, as opposed to the vertical, element-by-element operations on two different vectors.

The `add_horizontal`, `unpack_low` and `pack_sat` functions are examples of horizontal data support. This support enables certain algorithms that cannot exploit the full potential of SIMD instructions.

Shuffle intrinsics are another example of horizontal data flow. Shuffle intrinsics are not expressed in the C++ classes due to their immediate arguments. However, the C++ class implementation enables you to mix shuffle intrinsics with the other C++ functions. For example:

```
F32vec4 fveca, fvecb, fvecd;
fveca += fvecb;
fvecd = _mm_shuffle_ps(fveca,fvecb,0);
```

Typically every instruction with horizontal data flow contains some inefficiency in the implementation. If possible, implement your algorithms without using the horizontal capabilities.

Branch Compression/Elimination

Branching in SIMD architectures can be complicated and expensive, possibly resulting in poor predictability and code expansion. The SIMD C++ classes provide functions to eliminate branches, using logical operations, max and min functions, conditional selects, and compares. Consider the following example:

```
short a[4], b[4], c[4];
for (i=0; i<4; i++)
c[i] = a[i] > b[i] ? a[i] : b[i];
```

This operation is independent of the value of `i`. For each `i`, the result could be either `A` or `B` depending on the actual values. A simple way of removing the branch altogether is to use the `select_gt` function, as follows:

```
Is16vec4 a, b, c
c = select_gt(a, b, a, b)
```

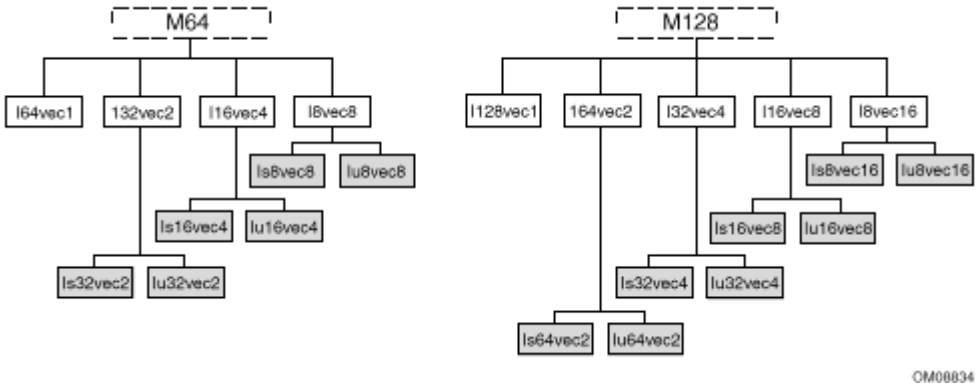
Caching Hints

Streaming SIMD Extensions provide prefetching and streaming hints. Prefetching data can minimize the effects of memory latency. Streaming hints allow you to indicate that certain data should not be cached. This results in higher performance for data that should be cached.

Integer Vector Classes

The `Ivec` classes provide an interface to SIMD processing using integer vectors of various sizes. The class hierarchy is represented in the following figure.

Ivec Class Hierarchy



The `M64` and `M128` classes define the `__m64` and `__m128i` data types from which the rest of the `Ivec` classes are derived. The first generation of child classes are derived based solely on bit sizes of 128, 64, 32, 16, and 8 respectively for the `I128vec1`, `I64vec1`, `I64vec2`, `I32vec2`, `I32vec4`, `I16vec4`, `I16vec8`, `I8vec16`, and `I8vec8` classes. The latter seven of the these classes require specification of signedness and saturation.

Caution

Do not intermix the `M64` and `M128` data types. You will get unexpected behavior if you do.

The signedness is indicated by the `s` and `u` in the class names:

```
Is64vec2
Iu64vec2
Is32vec4
Iu32vec4
Is16vec8
Iu16vec8
Is8vec16
Iu8vec16
Is32vec2
Iu32vec2
Is16vec4
Iu16vec4
Is8vec8
Iu8vec8
```

Terms, Conventions, and Syntax

The following are special terms and syntax used in this chapter to describe functionality of the classes with respect to their associated operations.

Ivec Class Syntax Conventions

The name of each class denotes the data type, signedness, bit size, number of elements using the following generic format:

```
<type><signedness><bits>vec<elements>

{ F | I } { s | u } { 64 | 32 | 16 | 8 } vec { 8 | 4 | 2 | 1 }
```

where

`type`	indicates floating point (`F`) or integer (`I`)
`signedness`	indicates signed (`s`) or unsigned (`u`). For the Ivec class, leaving this field blank indicates an intermediate class. There are no unsigned Fvec classes, therefore for the Fvec classes, this field is blank.
`bits`	specifies the number of bits per element
`elements`	specifies the number of elements

Special Terms and Conventions

The following terms are used to define the functionality and characteristics of the classes and operations defined in this manual.

- **Nearest Common Ancestor** -- This is the intermediate or parent class of two classes of the same size. For example, the nearest common ancestor of `Iu8vec8` and `Is8vec8` is `I8vec8`. Also, the nearest common ancestor between `Iu8vec8` and `I16vec4` is `M64`.
- **Casting** -- Changes the data type from one class to another. When an operation uses different data types as operands, the return value of the operation must be assigned to a single data type. Therefore, one or more of the data types must be converted to a required data type. This conversion is known as a typecast. Sometimes, typecasting is automatic, other times you must use special syntax to explicitly typecast it yourself.
- **Operator Overloading** -- This is the ability to use various operators on the same user-defined data type of a given class. Once you declare a variable, you can add, subtract, multiply, and perform a range of operations. Each family of classes accepts a specified range of operators, and must comply by rules and restrictions regarding typecasting and operator overloading as defined in the header files. The following table shows the notation used in this documention to address typecasting, operator overloading, and other rules.

Class Syntax Notation Conventions

Class Name	Description
I[s|u][N]vec[N]	Any value except I128vec1 nor I64vec1
I64vec1	__m64 data type
I[s|u]64vec2	two 64-bit values of any signedness
I[s|u]32vec4	four 32-bit values of any signedness
I[s|u]8vec16	eight 16-bit values of any signedness
I[s|u]16vec8	sixteen 8-bit values of any signedness
I[s|u]32vec2	two 32-bit values of any signedness
I[s|u]16vec4	four 16-bit values of any signedness
I[s|u]8vec8	eight 8-bit values of any signedness

Rules for Operators

To use operators with the Ivec classes you must use one of the following three syntax conventions:

```
[ Ivec_Class ] R = [ Ivec_Class ] A [ operator ][ Ivec_Class ] B
```

Example 1: `I64vec1 R = I64vec1 A & I64vec1 B;`

```
[ Ivec_Class ] R =[ operator ] ([ Ivec_Class ] A,[ Ivec_Class ] B)
```

Example 2: `I64vec1 R = andnot(I64vec1 A, I64vec1 B);`

```
[ Ivec_Class ] R [ operator ]= [ Ivec_Class ] A
```

Example 3: `I64vec1 R &= I64vec1 A;`

[operator]an operator (for example, &, |, or ^)

[Ivec_Class] an Ivec class

R, A, B variables declared using the pertinent Ivec classes

The table that follows shows automatic and explicit sign and size typecasting. "Explicit" means that it is illegal to mix different types without an explicit typecasting. "Automatic" means that you can mix types freely and the compiler will do the typecasting for you.

Summary of Rules Major Operators

Operators	Sign Typecasting	Size Typecasting	Other Typecasting Requirements
Assignment	N/A	N/A	N/A
Logical	Automatic	Automatic (to left)	Explicit typecasting is required for different types used in non-logical expressions on the right side of the assignment.
Addition and Subtraction	Automatic	Explicit	N/A
Multiplication	Automatic	Explicit	N/A
Shift	Automatic	Explicit	Casting Required to ensure arithmetic shift.
Compare	Automatic	Explicit	Explicit casting is required for signed classes for the less-than or greater-than operations.
Conditional Select	Automatic	Explicit	Explicit casting is required for signed classes for less-than or greater-than operations.

Data Declaration and Initialization

The following table shows literal examples of constructor declarations and data type initialization for all class sizes. All values are initialized with the most significant element on the left and the least significant to the right.

Declaration and Initialization Data Types for Ivec Classes

Operation	Class	Syntax
Declaration	M128	`I128vec1 A; Iu8vec16 A;`
Declaration	M64	`I64vec1 A; Iu8vec16 A;`
`__m128` Initialization	M128	`I128vec1 A(__m128 m); Iu16vec8(__m128 m);`
`__m64` Initialization	M64	`I64vec1 A(__m64 m);Iu8vec8 A(__m64 m);`
`__int64` Initialization	M64	`I64vec1 A = __int64 m; Iu8vec8 A =__int64 m;`
`int i` Initialization	M64	`I64vec1 A = int i; Iu8vec8 A = int i;`

`int` initialization	I32vec2	`I32vec2 A(int A1, int A0);`
		`Is32vec2 A(signed int A1, signed int A0);`
		`Iu32vec2 A(unsigned int A1, unsigned int A0);`

`int` Initialization	I32vec4	`I32vec4 A(short A3, short A2, short A1, short A0);` `Is32vec4 A(signed short A3, ..., signed short`

		A0); Iu32vec4 A(unsigned short A3, ..., unsigned short A0);
short int Initialization	I16vec4	I16vec4 A(short A3, short A2, short A1, short A0); Is16vec4 A(signed short A3, ..., signed short A0); Iu16vec4 A(unsigned short A3, ..., unsigned short A0);
short int Initialization	I16vec8	I16vec8 A(short A7, short A6, ..., short A1, short A0); Is16vec8 A(signed A7, ..., signed short A0); Iu16vec8 A(unsigned short A7, ..., unsigned short A0);
char Initialization	I8vec8	I8vec8 A(char A7, char A6, ..., char A1, char A0); Is8vec8 A(signed char A7, ..., signed char A0); Iu8vec8 A(unsigned char A7, ..., unsigned char A0);
char Initialization	I8vec16	I8vec16 A(char A15, ..., char A0); Is8vec16 A(signed char A15, ..., signed char A0); Iu8vec16 A(unsigned char A15, ..., unsigned char A0);

Assignment Operator

Any `Ivec` object can be assigned to any other `Ivec` object; conversion on assignment from one `Ivec` object to another is automatic.

Assignment Operator Examples

```
Is16vec4 A;

Is8vec8 B;

I64vec1 C;

A = B; /* assign Is8vec8 to Is16vec4 */

B = C; /* assign I64vec1 to Is8vec8 */

B = A & C; /* assign M64 result of '&' to Is8vec8 */
```

Logical Operators

The logical operators use the symbols and intrinsics listed in the following table.

Bitwise Operation	Operator Symbols		Syntax Usage		Corresponding Intrinsic
	Standard	w / assign	Standard	w / assign	
AND	&	&=	R = A & B	R &= A	_mm_and_si64 _mm_and_si128
OR	|	|=	R = A | B	R |= A	_mm_and_si64 _mm_and_si128
XOR	^	^=	R = A^B	R ^= A	_mm_and_si64 _mm_and_si128
ANDNOT	andnot	N/A	R = A andnot B	N/A	_mm_and_si64 _mm_and_si128

Logical Operators and Miscellaneous Exceptions.

A and B converted to M64. Result assigned to `Iu8vec8`.

```
I64vec1 A;

Is8vec8 B;

Iu8vec8 C;

C = A & B;
```

Same size and signedness operators return the nearest common ancestor.

```
I32vec2 R = Is32vec2 A ^ Iu32vec2 B;
```

A&B returns M64, which is cast to `Iu8vec8`.

```
C = Iu8vec8(A&B)+ C;
```

When `A` and `B` are of the same class, they return the same type. When `A` and `B` are of different classes, the return value is the return type of the nearest common ancestor.

The logical operator returns values for combinations of classes, listed in the following tables, apply when `A` and `B` are of different classes.

Ivec Logical Operator Overloading

Return (R)	AND	OR	XOR	NAND	A Operand	B Operand
I64vec1 R	&	|	^	andnot	I[s|u]64vec2 A	I[s|u]64vec2 B
I64vec2 R	&	|	^	andnot	I[s|u]64vec2 A	I[s|u]64vec2 B
I32vec2 R	&	|	^	andnot	I[s|u]32vec2 A	I[s|u]32vec2 B
I32vec4 R	&	|	^	andnot	I[s|u]32vec4 A	I[s|u]32vec4 B
I16vec4 R	&	|	^	andnot	I[s|u]16vec4 A	I[s|u]16vec4 B
I16vec8 R	&	|	^	andnot	I[s|u]16vec8 A	I[s|u]16vec8 B

I8vec8 R	&	|	^	andnot	I[s|u]8vec8 A	I[s|u]8vec8 B
I8vec16 R	&	|	^	andnot	I[s|u]8vec16 A	I[s|u]8vec16 B

For logical operators with assignment, the return value of R is always the same data type as the pre-declared value of R as listed in the table that follows.

Ivec Logical Operator Overloading with Assignment

Return Type	Left Side (R)	AND	OR	XOR	Right Side (Any Ivec Type)
I128vec1	I128vec1 R	&=	|=	^=	I[s|u][N]vec[N] A;
I64vec1	I64vec1 R	&=	|=	^=	I[s|u][N]vec[N] A;
I64vec2	I64vec2 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]32vec4	I[x]32vec4 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]32vec2	I[x]32vec2 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]16vec8	I[x]16vec8 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]16vec4	I[x]16vec4 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]8vec16	I[x]8vec16 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]8vec8	I[x]8vec8 R	&=	|=	^=	I[s|u][N]vec[N] A;

Addition and Subtraction Operators

The addition and subtraction operators return the class of the nearest common ancestor when the right-side operands are of different signs. The following code provides examples of usage and miscellaneous exceptions.

Syntax Usage for Addition and Subtraction Operators

Return nearest common ancestor type, I16vec4.

```
Is16vec4 A;

Iu16vec4 B;

I16vec4 C;

C = A + B;
```

Returns type left-hand operand type.

```
Is16vec4 A;

Iu16vec4 B;
```

```
A += B;

B -= A;
```

Explicitly convert B to `Is16vec4`.

```
Is16vec4 A,C;

Iu32vec24 B;

C = A + C;

C = A + (Is16vec4)B;
```

Addition and Subtraction Operators with Corresponding Intrinsics

Operation	Symbols	Syntax	Corresponding Intrinsics
Addition	+ +=	R = A + B R += A	_mm_add_epi64 _mm_add_epi32 _mm_add_epi16 _mm_add_epi8 _mm_add_pi32 _mm_add_pi16 _mm_add_pi8
Subtraction	- -=	R = A - B R -= A	_mm_sub_epi64 _mm_sub_epi32 _mm_sub_epi16 _mm_sub_epi8 _mm_sub_pi32 _mm_sub_pi16 _mm_sub_pi8

The following table lists addition and subtraction return values for combinations of classes when the right side operands are of different signedness. The two operands must be the same size, otherwise you must explicitly indicate the typecasting.

Addition and Subtraction Operator Overloading

Return Value	Available Operators		Right Side Operands	
R	Add	Sub	A	B
I64vec2 R	+	-	I[s|u]64vec2 A	I[s|u]64vec2 B
I32vec4 R	+	-	I[s|u]32vec4 A	I[s|u]32vec4 B
I32vec2 R	+	-	I[s|u]32vec2 A	I[s|u]32vec2 B
I16vec8 R	+	-	I[s|u]16vec8 A	I[s|u]16vec8 B
I16vec4 R	+	-	I[s|u]16vec4 A	I[s|u]16vec4 B
I8vec8 R	+	-	I[s|u]8vec8 A	I[s|u]8vec8 B
I8vec16 R	+	-	I[s|u]8vec2 A	I[s|u]8vec16 B

The following table shows the return data type values for operands of the addition and subtraction operators with assignment. The left side operand determines the size and signedness of the return value. The right side operand must be the same size as the left operand; otherwise, you must use an explicit typecast.

Addition and Subtraction with Assignment

Return Value (R)	Left Side (R)	Add	Sub	Right Side (A)
I[x]32vec4	I[x]32vec2 R	+=	-=	I[s|u]32vec4 A;
I[x]32vec2 R	I[x]32vec2 R	+=	-=	I[s|u]32vec2 A;
I[x]16vec8	I[x]16vec8	+=	-=	I[s|u]16vec8 A;
I[x]16vec4	I[x]16vec4	+=	-=	I[s|u]16vec4 A;
I[x]8vec16	I[x]8vec16	+=	-=	I[s|u]8vec16 A;
I[x]8vec8	I[x]8vec8	+=	-=	I[s|u]8vec8 A;

Multiplication Operators

The multiplication operators can only accept and return data types from the `I[s|u]16vec4` or `I[s|u]16vec8` classes, as shown in the following example.

Syntax Usage for Multiplication Operators

Explicitly convert B to `Is16vec4`.

```
Is16vec4 A,C;

Iu32vec2 B;

C = A * C;

C = A * (Is16vec4)B;
```

Return nearest common ancestor type, `I16vec4`

```
Is16vec4 A;

Iu16vec4 B;

I16vec4 C;

C = A + B;
```

The `mul_high` and `mul_add` functions take `Is16vec4` data only.

```
Is16vec4 A,B,C,D;

C = mul_high(A,B);
```

```
D = mul_add(A,B);
```

Multiplication Operators with Corresponding Intrinsics

Symbols		Syntax Usage	Intrinsic
*	*=	R = A * B R *= A	_mm_mullo_pi16 _mm_mullo_epi16
mul_high	N/A	R = mul_high(A, B)	_mm_mulhi_pi16 _mm_mulhi_epi16
mul_add	N/A	R = mul_high(A, B)	_mm_madd_pi16 _mm_madd_epi16

The multiplication return operators always return the nearest common ancestor as listed in the table that follows. The two operands must be 16 bits in size, otherwise you must explicitly indicate typecasting.

Multiplication Operator Overloading

R	Mul	A	B
I16vec4 R	*	I[s|u]16vec4 A	I[s|u]16vec4 B
I16vec8 R	*	I[s|u]16vec8 A	I[s|u]16vec8 B
Is16vec4 R	mul_add	Is16vec4 A	Is16vec4 B
Is16vec8	mul_add	Is16vec8 A	Is16vec8 B
Is32vec2 R	mul_high	Is16vec4 A	Is16vec4 B
Is32vec4 R	mul_high	s16vec8 A	Is16vec8 B

The following table shows the return values and data type assignments for operands of the multiplication operators with assignment. All operands must be 16 bytes in size. If the operands are not the right size, you must use an explicit typecast.

Multiplication with Assignment

Return Value (R)	Left Side (R)	Mul	Right Side (A)
I[x]16vec8	I[x]16vec8	*=	I[s|u]16vec8 A;
I[x]16vec4	I[x]16vec4	*=	I[s|u]16vec4 A;

Shift Operators

The right shift argument can be any integer or Ivec value, and is implicitly converted to a M64 data type. The first or left operand of a << can be of any type except I[s|u]8vec[8|16].

Example Syntax Usage for Shift Operators

Automatic size and sign conversion.

```
Is16vec4 A,C;

Iu32vec2 B;

C = A;
```

A&B returns `I16vec4`, which must be cast to `Iu16vec4` to ensure logical shift, not arithmetic shift.

```
Is16vec4 A, C;

Iu16vec4 B, R;

R = (Iu16vec4)(A & B) C;
```

A&B returns `I16vec4`, which must be cast to `Is16vec4` to ensure arithmetic shift, not logical shift.

```
R = (Is16vec4)(A & B) C;
```

Shift Operators with Corresponding Intrinsics

Operation	Symbols	Syntax Usage	Intrinsic
Shift Left	<< &=	R = A << B R &= A	_mm_sll_si64 _mm_slli_si64 _mm_sll_pi32 _mm_slli_pi32 _mm_sll_pi16 _mm_slli_pi16
Shift Right	>>	R = A >> B R >>= A	_mm_srl_si64 _mm_srli_si64 _mm_srl_pi32 _mm_srli_pi32 _mm_srl_pi16 _mm_srli_pi16 _mm_sra_pi32 _mm_srai_pi32 _mm_sra_pi16 _mm_srai_pi16

Right shift operations with signed data types use arithmetic shifts. All unsigned and intermediate classes correspond to logical shifts. The following table shows how the return type is determined by the first argument type.

Shift Operator Overloading

Operation	R	Right Shift		Left Shift		A	B
Logical	I64vec1	>>	>>=	<<	<<=	I64vec1 A;	I64vec1 B;
Logical	I32vec2	>>	>>=	<<	<<=	I32vec2 A	I32vec2 B;
Arithmetic	Is32vec2	>>	>>=	<<	<<=	Is32vec2 A	I[s|u][N]vec[N] B;
Logical	Iu32vec2	>>	>>=	<<	<<=	Iu32vec2 A	I[s|u][N]vec[N] B;
Logical	I16vec4	>>	>>=	<<	<<=	I16vec4 A	I16vec4 B
Arithmetic	Is16vec4	>>	>>=	<<	<<=	Is16vec4 A	I[s|u][N]vec[N] B;
Logical	Iu16vec4	>>	>>=	<<	<<=	Iu16vec4 A	I[s|u][N]vec[N] B;

Comparison Operators

The equality and inequality comparison operands can have mixed signedness, but they must be of the same size. The comparison operators for less-than and greater-than must be of the same sign and size.

Example of Syntax Usage for Comparison Operator

The nearest common ancestor is returned for compare for equal/not-equal operations.

```
Iu8vec8 A;

Is8vec8 B;

I8vec8 C;

C = cmpneq(A,B);
```

Type cast needed for different-sized elements for equal/not-equal comparisons.

```
Iu8vec8 A, C;

Is16vec4 B;

C = cmpeq(A,(Iu8vec8)B);
```

Type cast needed for sign or size differences for less-than and greater-than comparisons.

```
Iu16vec4 A;

Is16vec4 B, C;

C = cmpge((Is16vec4)A,B);
```

```
C = cmpgt(B,C);
```

Inequality Comparison Symbols and Corresponding Intrinsics

Compare For:	Operators	Syntax	Intrinsic	
Equality	cmpeq	R = cmpeq(A, B)	_mm_cmpeq_pi32 _mm_cmpeq_pi16 _mm_cmpeq_pi8	
Inequality	cmpneq	R = cmpneq(A, B)	_mm_cmpeq_pi32 _mm_cmpeq_pi16 _mm_cmpeq_pi8	_mm_andnot_si64
Greater Than	cmpgt	R = cmpgt(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8	
Greater Than or Equal To	cmpge	R = cmpge(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8	_mm_andnot_si64
Less Than	cmplt	R = cmplt(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8	
Less Than or Equal To	cmple	R = cmple(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8	_mm_andnot_si64

Comparison operators have the restriction that the operands must be the size and sign as listed in the Compare Operator Overloading table.

Compare Operator Overloading

R	Comparison	A	B
I32vec2 R	cmpeq cmpne	I[s|u]32vec2 B	I[s|u]32vec2 B
I16vec4 R		I[s|u]16vec4 B	I[s|u]16vec4 B
I8vec8 R		I[s|u]8vec8 B	I[s|u]8vec8 B
I32vec2 R	cmpgt cmpge cmplt cmple	Is32vec2 B	Is32vec2 B
I16vec4 R		Is16vec4 B	Is16vec4 B
I8vec8 R		Is8vec8 B	Is8vec8 B

Conditional Select Operators

For conditional select operands, the third and fourth operands determine the type returned. Third and fourth operands with same size, but different signedness, return the nearest common ancestor data type.

Conditional Select Syntax Usage

Return the nearest common ancestor data type if third and fourth operands are of the same size, but different signs.

```
I16vec4 R = select_neq(Is16vec4, Is16vec4, Is16vec4, Iu16vec4);
```

Conditional Select for Equality

```
R0 := (A0 == B0) ? C0 : D0;

R1 := (A1 == B1) ? C1 : D1;

R2 := (A2 == B2) ? C2 : D2;

R3 := (A3 == B3) ? C3 : D3;
```

Conditional Select for Inequality

```
R0 := (A0 != B0) ? C0 : D0;

R1 := (A1 != B1) ? C1 : D1;

R2 := (A2 != B2) ? C2 : D2;

R3 := (A3 != B3) ? C3 : D3;
```

Conditional Select Symbols and Corresponding Intrinsics

Conditional Select For:	Operators	Syntax	Corresponding Intrinsic	Additional Intrinsic (Applies to All)

Equality	`select_eq`	R = `select_eq`(A, B, C, D)	`_mm_cmpeq_pi32`
			`_mm_cmpeq_pi16`
			`_mm_cmpeq_pi8`
				`_mm_and_si64`
				`_mm_or_si64`
				`_mm_andnot_si64`

Inequality	`select_neq`	R = `select_neq`(A, B, C, D)	`_mm_cmpeq_pi32`
			`_mm_cmpeq_pi16`
			`_mm_cmpeq_pi8`	

Greater Than	`select_gt`	R = `select_gt`(A, B, C, D)	`_mm_cmpgt_pi32`
			`_mm_cmpgt_pi16`
			`_mm_cmpgt_pi8`	

Greater Than or Equal To	`select_ge`	R = `select_gt`(A, B, C, D)	`_mm_cmpge_pi32`
			`_mm_cmpge_pi16`
			`_mm_cmpge_pi8`	

Less Than	`select_lt`	R = `select_lt`(A, B, C, D)	`_mm_cmplt_pi32`
			`_mm_cmplt_pi16`
			`_mm_cmplt_pi8`	

Less Than	`select_le`	R =	`_mm_cmple_pi32` `_mm_cmple_pi16`	

or Equal To		select_le(A, B, C, D)	_mm_cmple_pi8	

All conditional select operands must be of the same size. The return data type is the nearest common ancestor of operands C and D. For conditional select operations using greater-than or less-than operations, the first and second operands must be signed as listed in the table that follows.

Conditional Select Operator Overloading

R	Comparison	A and B	C	D
I32vec2 R	select_eq select_ne	I[s|u]32vec2	I[s|u]32vec2	I[s|u]32vec2
I16vec4 R		I[s|u]16vec4	I[s|u]16vec4	I[s|u]16vec4
I8vec8 R		I[s|u]8vec8	I[s|u]8vec8	I[s|u]8vec8
I32vec2 R	select_gt select_ge select_lt select_le	Is32vec2	Is32vec2	Is32vec2
I16vec4 R		Is16vec4	Is16vec4	Is16vec4
I8vec8 R		Is8vec8	Is8vec8	Is8vec8

The following table shows the mapping of return values from R0 to R7 for any number of elements. The same return value mappings also apply when there are fewer than four return values.

Conditional Select Operator Return Value Mapping

Return Value	A and B Operands								C and D operands
	A0	Available Operators						B0	
R0:=	A0	==	!=	>	>=	<	<=	B0	? C0 : D0;
R1:=	A0	==	!=	>	>=	<	<=	B0	? C1 : D1;
R2:=	A0	==	!=	>	>=	<	<=	B0	? C2 : D2;
R3:=	A0	==	!=	>	>=	<	<=	B0	? C3 : D3;
R4:=	A0	==	!=	>	>=	<	<=	B0	? C4 : D4;
R5:=	A0	==	!=	>	>=	<	<=	B0	? C5 : D5;
R6:=	A0	==	!=	>	>=	<	<=	B0	? C6 : D6;
R7:=	A0	==	!=	>	>=	<	<=	B0	? C7 : D7;

Debug

The debug operations do not map to any compiler intrinsics for MMX(TM) instructions. They are provided for debugging programs only. Use of these operations may result in loss of performance, so you should not use them outside of debugging.

Output

The four 32-bit values of `A` are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is32vec4 A;

cout << Iu32vec4 A;

cout << hex << Iu32vec4 A; /* print in hex format */

"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The two 32-bit values of `A` are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is32vec2 A;

cout << Iu32vec2 A;

cout << hex << Iu32vec2 A; /* print in hex format */

"[1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The eight 16-bit values of `A` are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is16vec8 A;

cout << Iu16vec8 A;

cout << hex << Iu16vec8 A; /* print in hex format */

"[7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The four 16-bit values of `A` are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is16vec4 A;
```

```
cout << Iu16vec4 A;

cout << hex << Iu16vec4 A; /* print in hex format */

"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The sixteen 8-bit values of `A` are placed in the output buffer and printed in the following format (default is decimal):

```
cout << Is8vec16 A; cout << Iu8vec16 A; cout << hex << Iu8vec8 A;

/* print in hex format instead of decimal*/

"[15]:A15 [14]:A14 [13]:A13 [12]:A12 [11]:A11 [10]:A10 [9]:A9 [8]:A8
[7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The eight 8-bit values of `A` are placed in the output buffer and printed in the following format (default is decimal):

```
cout << Is8vec8 A; cout << Iu8vec8 A;cout << hex << Iu8vec8 A;

/* print in hex format instead of decimal*/

"[7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

Element Access Operators

```
int R = Is64vec2 A[i];

unsigned int R = Iu64vec2 A[i];

int R = Is32vec4 A[i];

unsigned int R = Iu32vec4 A[i];

int R = Is32vec2 A[i];

unsigned int R = Iu32vec2 A[i];

short R = Is16vec8 A[i];

unsigned short R = Iu16vec8 A[i];

short R = Is16vec4 A[i];

unsigned short R = Iu16vec4 A[i];
```

```
signed char R = Is8vec16 A[i];

unsigned char R = Iu8vec16 A[i];

signed char R = Is8vec8 A[i];

unsigned char R = Iu8vec8 A[i];
```

Access and read element `i` of `A`. If DEBUG is enabled and the user tries to access an element outside of `A`, a diagnostic message is printed and the program aborts.

Corresponding Intrinsics: none

Element Assignment Operators

```
Is64vec2 A[i] = int R;

Is32vec4 A[i] = int R;

Iu32vec4 A[i] = unsigned int R;

Is32vec2 A[i] = int R;

Iu32vec2 A[i] = unsigned int R;

Is16vec8 A[i] = short R;

Iu16vec8 A[i] = unsigned short R;

Is16vec4 A[i] = short R;

Iu16vec4 A[i] = unsigned short R;

Is8vec16 A[i] = signed char R;

Iu8vec16 A[i] = unsigned char R;

Is8vec8 A[i] = signed char R;

Iu8vec8 A[i] = unsigned char R;
```

Assign `R` to element i of `A`. If `DEBUG` is enabled and the user tries to assign a value to an element outside of `A`, a diagnostic message is printed and the program aborts.

Corresponding Intrinsics: none

Unpack Operators

Interleave the 64-bit value from the high half of `A` with the 64-bit value from the high half of `B`.

```
I364vec2 unpack_high(I64vec2 A, I64vec2 B);

Is64vec2 unpack_high(Is64vec2 A, Is64vec2 B);

Iu64vec2 unpack_high(Iu64vec2 A, Iu64vec2 B);

R0 = A1;
R1 = B1;
```

Corresponding intrinsic: `_mm_unpackhi_epi64`

Interleave the two 32-bit values from the high half of `A` with the two 32-bit values from the high half of `B` .

```
I32vec4 unpack_high(I32vec4 A, I32vec4 B);

Is32vec4 unpack_high(Is32vec4 A, Is32vec4 B);

Iu32vec4 unpack_high(Iu32vec4 A, Iu32vec4 B);

R0 = A1;
R1 = B1;
R2 = A2;
R3 = B2;
```

Corresponding intrinsic: `_mm_unpackhi_epi32`

Interleave the 32-bit value from the high half of `A` with the 32-bit value from the high half of `B`.

```
I32vec2 unpack_high(I32vec2 A, I32vec2 B);

Is32vec2 unpack_high(Is32vec2 A, Is32vec2 B);

Iu32vec2 unpack_high(Iu32vec2 A, Iu32vec2 B);

R0 = A1;
R1 = B1;
```

Corresponding intrinsic: `_mm_unpackhi_pi32`

Interleave the four 16-bit values from the high half of `A` with the two 16-bit values from the high half of `B`.

```
I16vec8 unpack_high(I16vec8 A, I16vec8 B);

Is16vec8 unpack_high(Is16vec8 A, Is16vec8 B);

Iu16vec8 unpack_high(Iu16vec8 A, Iu16vec8 B);

R0 = A2;
R1 = B2;
R2 = A3;
R3 = B3;
```

Corresponding intrinsic: `_mm_unpackhi_epi16`

Interleave the two 16-bit values from the high half of `A` with the two 16-bit values from the high half of `B`.

```
I16vec4 unpack_high(I16vec4 A, I16vec4 B);

Is16vec4 unpack_high(Is16vec4 A, Is16vec4 B);

Iu16vec4 unpack_high(Iu16vec4 A, Iu16vec4 B);

R0 = A2;R1 = B2;
R2 = A3;R3 = B3;
```

Corresponding intrinsic: `_mm_unpackhi_pi16`

Interleave the four 8-bit values from the high half of `A` with the four 8-bit values from the high half of `B`.

```
I8vec8 unpack_high(I8vec8 A, I8vec8 B);

Is8vec8 unpack_high(Is8vec8 A, I8vec8 B);

Iu8vec8 unpack_high(Iu8vec8 A, I8vec8 B);

R0 = A4;
R1 = B4;
R2 = A5;
R3 = B5;
R4 = A6;
R5 = B6;
R6 = A7;
R7 = B7;
```

Corresponding intrinsic: `_mm_unpackhi_pi8`

Interleave the sixteen 8-bit values from the high half of `A` with the four 8-bit values from the high half of `B`.

```
I8vec16 unpack_high(I8vec16 A, I8vec16 B);

Is8vec16 unpack_high(Is8vec16 A, I8vec16 B);

Iu8vec16 unpack_high(Iu8vec16 A, I8vec16 B);

R0 = A8;
R1 = B8;
R2 = A9;
R3 = B9;
R4 = A10;
R5 = B10;
R6 = A11;
R7 = B11;
R8 = A12;
R8 = B12;
R2 = A13;
```

```
R3 = B13;
R4 = A14;
R5 = B14;
R6 = A15;
R7 = B15;
```

Corresponding intrinsic: `_mm_unpackhi_epi16`

Interleave the 32-bit value from the low half of `A` with the 32-bit value from the low half of `B`

```
R0 = A0;
R1 = B0;
```

Corresponding intrinsic: `_mm_unpacklo_epi32`

Interleave the 64-bit value from the low half of `A` with the 64-bit values from the low half of `B`

```
I64vec2 unpack_low(I64vec2 A, I64vec2 B);

Is64vec2 unpack_low(Is64vec2 A, Is64vec2 B);

Iu64vec2 unpack_low(Iu64vec2 A, Iu64vec2 B);

R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```

Corresponding intrinsic: `_mm_unpacklo_epi32`

Interleave the two 32-bit values from the low half of `A` with the two 32-bit values from the low half of `B`

```
I32vec4 unpack_low(I32vec4 A, I32vec4 B);

Is32vec4 unpack_low(Is32vec4 A, Is32vec4 B);

Iu32vec4 unpack_low(Iu32vec4 A, Iu32vec4 B);

R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```

Corresponding intrinsic: `_mm_unpacklo_epi32`

Interleave the 32-bit value from the low half of `A` with the 32-bit value from the low half of `B`.

```
I32vec2 unpack_low(I32vec2 A, I32vec2 B);

Is32vec2 unpack_low(Is32vec2 A, Is32vec2 B);
```

```
Iu32vec2 unpack_low(Iu32vec2 A, Iu32vec2 B);

R0 = A0;
R1 = B0;
```

Corresponding intrinsic: `_mm_unpacklo_pi32`

Interleave the two 16-bit values from the low half of `A` with the two 16-bit values from the low half of `B`.

```
I16vec8 unpack_low(I16vec8 A, I16vec8 B);

Is16vec8 unpack_low(Is16vec8 A, Is16vec8 B);

Iu16vec8 unpack_low(Iu16vec8 A, Iu16vec8 B);

R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
```

Corresponding intrinsic: `_mm_unpacklo_epi16`

Interleave the two 16-bit values from the low half of `A` with the two 16-bit values from the low half of `B`.

```
I16vec4 unpack_low(I16vec4 A, I16vec4 B);

Is16vec4 unpack_low(Is16vec4 A, Is16vec4 B);

Iu16vec4 unpack_low(Iu16vec4 A, Iu16vec4 B);

R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```

Corresponding intrinsic: `_mm_unpacklo_pi16`

Interleave the four 8-bit values from the high low of `A` with the four 8-bit values from the low half of `B`.

```
I8vec16 unpack_low(I8vec16 A, I8vec16 B);

Is8vec16 unpack_low(Is8vec16 A, Is8vec16 B);

Iu8vec16 unpack_low(Iu8vec16 A, Iu8vec16 B);

R0 = A0;
R1 = B0;
R2 = A1;
```

```
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
R8 = A4;
R9 = B4;
R10 = A5;
R11 = B5;
R12 = A6;
R13 = B6;
R14 = A7;
R15 = B7;
```

Corresponding intrinsic: `_mm_unpacklo_epi8`

Interleave the four 8-bit values from the high low of `A` with the four 8-bit values from the low half of `B`.

```
I8vec8 unpack_low(I8vec8 A, I8vec8 B);

Is8vec8 unpack_low(Is8vec8 A, Is8vec8 B);

Iu8vec8 unpack_low(Iu8vec8 A, Iu8vec8 B);

R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
```

Corresponding intrinsic: `_mm_unpacklo_pi8`

Pack Operators

Pack the eight 32-bit values found in `A` and `B` into eight 16-bit values with signed saturation.

```
Is16vec8 pack_sat(Is32vec2 A,Is32vec2 B);
```
Corresponding intrinsic: `_mm_packs_epi32`

Pack the four 32-bit values found in `A` and `B` into eight 16-bit values with signed saturation.

```
Is16vec4 pack_sat(Is32vec2 A,Is32vec2 B);
```
Corresponding intrinsic: `_mm_packs_pi32`

Pack the sixteen 16-bit values found in `A` and `B` into sixteen 8-bit values with signed saturation.

```
Is8vec16 pack_sat(Is16vec4 A,Is16vec4 B);
```
Corresponding intrinsic: `_mm_packs_epi16`

Pack the eight 16-bit values found in `A` and `B` into eight 8-bit values with signed saturation.

```
Is8vec8 pack_sat(Is16vec4 A,Is16vec4 B);
```
Corresponding intrinsic: `_mm_packs_pi16`

Pack the sixteen 16-bit values found in `A` and `B` into sixteen 8-bit values with unsigned saturation .

```
Iu8vec16 packu_sat(Is16vec4 A,Is16vec4 B);
```
Corresponding intrinsic: `_mm_packus_epi16`

Pack the eight 16-bit values found in `A` and `B` into eight 8-bit values with unsigned saturation.

```
Iu8vec8 packu_sat(Is16vec4 A,Is16vec4 B);
```
Corresponding intrinsic: `_mm_packs_pu16`

Clear MMX(TM) Instructions State Operator

Empty the MMX(TM) registers and clear the MMX state. Read the guidelines for using the EMMS instruction intrinsic.

```
void empty(void);
```
Corresponding intrinsic: `_mm_empty`

Integer Functions for Streaming SIMD Extensions

Note

You must include `fvec.h` header file for the following functionality.

Compute the element-wise maximum of the respective signed integer words in `A` and `B`.

```
Is16vec4 simd_max(Is16vec4 A, Is16vec4 B);
```
Corresponding intrinsic: `_mm_max_pi16`

Compute the element-wise minimum of the respective signed integer words in `A` and `B`.

```
Is16vec4 simd_min(Is16vec4 A, Is16vec4 B);
```
Corresponding intrinsic: `_mm_min_pi16`

Compute the element-wise maximum of the respective unsigned bytes in `A` and `B`.

```
Iu8vec8 simd_max(Iu8vec8 A, Iu8vec8 B);
```
Corresponding intrinsic: `_mm_max_pu8`

Compute the element-wise minimum of the respective unsigned bytes in `A` and `B`.

```
Iu8vec8 simd_min(Iu8vec8 A, Iu8vec8 B);
```
Corresponding intrinsic: `_mm_min_pu8`

Create an 8-bit mask from the most significant bits of the bytes in `A`.

```
int move_mask(I8vec8 A);
```
Corresponding intrinsic: `_mm_movemask_pi8`

Conditionally store byte elements of `A` to address `p`. The high bit of each byte in the selector `B` determines whether the corresponding byte in `A` will be stored.

```
void mask_move(I8vec8 A, I8vec8 B, signed char *p);
```
Corresponding intrinsic: `_mm_maskmove_si64`

Store the data in `A` to the address `p` without polluting the caches. `A` can be any `Ivec` type.

```
void store_nta(__m64 *p, M64 A);
```
Corresponding intrinsic: `_mm_stream_pi`

Compute the element-wise average of the respective unsigned 8-bit integers in `A` and `B`.

```
Iu8vec8 simd_avg(Iu8vec8 A, Iu8vec8 B);
```
Corresponding intrinsic: `_mm_avg_pu8`

Compute the element-wise average of the respective unsigned 16-bit integers in `A` and `B`.

```
Iu16vec4 simd_avg(Iu16vec4 A, Iu16vec4 B);
```
Corresponding intrinsic: `_mm_avg_pu16`

Conversions between Fvec and Ivec

Convert the lower double-precision floating-point value of `A` to a 32-bit integer with truncation.

```
int F64vec2ToInt(F64vec42 A);
r := (int)A0;
```

Convert the four floating-point values of `A` to two the two least significant double-precision floating-point values.

```
F64vec2 F32vec4ToF64vec2(F32vec4 A);
r0 := (double)A0;
r1 := (double)A1;
```

Convert the two double-precision floating-point values of `A` to two single-precision floating-point values.

```
F32vec4 F64vec2ToF32vec4(F64vec2 A);
r0 := (float)A0;
r1 := (float)A1;
```

Convert the signed `int` in `B` to a double-precision floating-point value and pass the upper double-precision value from `A` through to the result.

```
F64vec2 InttoF64vec2(F64vec2 A, int B);
r0 := (double)B;
r1 := A1;
```

Convert the lower floating-point value of `A` to a 32-bit integer with truncation.

```
int F32vec4ToInt(F32vec4 A);
r := (int)A0;
```

Convert the two lower floating-point values of `A` to two 32-bit integer with truncation, returning the integers in packed form.

```
Is32vec2 F32vec4ToIs32vec2 (F32vec4 A);
r0 := (int)A0;
r1 := (int)A1;
```

Convert the 32-bit integer value `B` to a floating-point value; the upper three floating-point values are passed through from `A`.

```
F32vec4 IntToF32vec4(F32vec4 A, int B);
r0 := (float)B;
r1 := A1;
r2 := A2;
r3 := A3;
```

Convert the two 32-bit integer values in packed form in `B` to two floating-point values; the upper two floating-point values are passed through from `A`.

```
F32vec4 Is32vec2ToF32vec4(F32vec4 A, Is32vec2 B);
r0 := (float)B0;
r1 := (float)B1;
r2 := A2;
r3 := A3;
```

Floating-point Vector Classes

The floating-point vector classes, `F64vec2`, `F32vec4`, and `F32vec1`, provide an interface to SIMD operations. The class specifications are as follows:

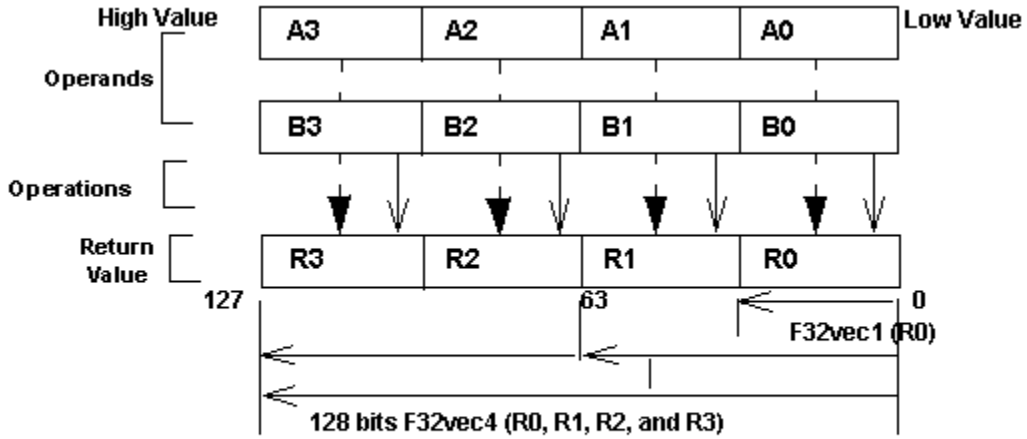
```
F64vec2 A(double x, double y);

F32vec4 A(float z, float y, float x, float w);

F32vec1 B(float w);
```

The packed floating-point input values are represented with the right-most value lowest as shown in the following table.

Single-Precision Floating-point Elements



F32vec4 returns **four** packed **single-precision floating point** values (**R0, R1, R2, and R3**).
F32vec2 returns **one single-precision floating point** value (**R0**).

Fvec Notation Conventions

This reference uses the following conventions for syntax and return values.

Fvec Classes Syntax Notation

Fvec classes use the syntax conventions shown the following examples:

```
[Fvec_Class] R = [Fvec_Class] A [operator][Ivec_Class] B;
```

Example 1: `F64vec2 R = F64vec2 A & F64vec2 B;`

```
[Fvec_Class] R = [operator]([Fvec_Class] A,[Fvec_Class] B);
```

Example 2: `F64vec2 R = andnot(F64vec2 A, F64vec2 B);`

```
[Fvec_Class] R [operator]= [Fvec_Class] A;
```

Example 3: `F64vec2 R &= F64vec2 A;`

where

`[operator]` is an operator (for example, &, |, or ^)

`[Fvec_Class]` is any `Fvec` class (`F64vec2`, `F32vec4`, or `F32vec1`)

R, A, B are declared `Fvec` variables of the type indicated

Return Value Notation

Because the Fvec classes have packed elements, the return values typically follow the conventions presented in the Return Value Convention Notation Mappings table. F32vec4 returns four single-precision, floating-point values (R0, R1, R2, and R3); F64vec2 returns two double-precision, floating-point values, and F32vec1 returns the lowest single-precision floating-point value (R0).

Return Value Convention Notation Mappings

Example 1:	Example 2:	Example 3:	F32vec4	F64vec2	F32vec1
R0 := A0 & B0;	R0 := A0 andnot B0;	R0 &= A0;	x	x	x
R1 := A1 & B1;	R1 := A1 andnot B1;	R1 &= A1;	x	x	N/A
R2 := A2 & B2;	R2 := A2 andnot B2;	R2 &= A2;	x	N/A	N/A
R3 := A3 & B3	R3 := A3 andhot B3;	R3 &= A3;	x	N/A	N/A

Data Alignment

Memory operations using the Streaming SIMD Extensions should be performed on 16-byte-aligned data whenever possible.

F32vec4 and F64vec2 object variables are properly aligned by default. Note that floating point arrays are not automatically aligned. To get 16-byte alignment, you can use the alignment __declspec:

```
__declspec( align(16) ) float A[4];
```

Conversions

All Fvec object variables can be implicitly converted to __m128 data types. For example, the results of computations performed on F32vec4 or F32vec1 object variables can be assigned to __m128 data types.

```
__m128d mm = A & B; /* where A,B are F64vec2 object variables */

__m128 mm = A & B; /* where A,B are F32vec4 object variables */

__m128 mm = A & B; /* where A,B are F32vec1 object variables */
```

Constructors and Initialization

The following table shows how to create and initialize F32vec objects with the Fvec classes.

Constructors and Initialization for Fvec Classes

Example	Intrinsic	Returns
Constructor Declaration		

`F64vec2 A;`
`F32vec4 B;`
`F32vec1 C;`
	N/A	N/A

__m128 Object Initialization		

`F64vec2 A(__m128d mm);`
`F32vec4 B(__m128 mm);`
`F32vec1 C(__m128 mm);`
	N/A	N/A

Double Initialization		

`/* Initializes two doubles. */`
`F64vec2 A(double d0, double d1);`
`F64vec2 A = F64vec2(double d0, double d1);`
	`_mm_set_pd`	A0 := d0;
		A1 := d1;

`F64vec2 A(double d0);`
`/* Initializes both return values`
`with the same double precision value */.`
	`_mm_set1_pd`	A0 := d0;
		A1 := d0;

Float Initialization		

`F32vec4 A(float f3, float f2,`
`float f1, float f0);`
`F32vec4 A = F32vec4(float f3, float f2,`
`float f1, float f0);`
	`_mm_set_ps`	A0 := f0;
		A1 := f1;
		A2 := f2;
		A3 := f3;

`F32vec4 A(float f0);`
`/* Initializes all return values`
`with the same floating point value. */`
	`_mm_set1_ps`	A0 := f0;
		A1 := f0;
		A2 := f0;
		A3 := f0;

`F32vec4 A(double d0);`
`/* Initialize all return values with`
`the same double-precision value. */`
	`_mm_set1_ps(d)`	A0 := d0;
		A1 := d0;
		A2 := d0;
		A3 := d0;

`F32vec1 A(double d0);`
`/* Initializes the lowest value of A`
`with d0 and the other values with 0.*/`
	`_mm_set_ss(d)`	A0 := d0;
		A1 := 0;
		A2 := 0;
		A3 := 0;

`F32vec1 B(float f0);`
`/* Initializes the lowest value of B`
`with f0 and the other values with 0.*/`
	`_mm_set_ss`	B0 := f0;
		B1 := 0;
		B2 := 0;
		B3 := 0;

`F32vec1 B(int I);`
`/* Initializes the lowest value of B`
`with f0, other values are undefined.*/`
	`_mm_cvtsi32_ss`	B0 := f0;
		B1 := {}
		B2 := {}
		B3 := {}

Arithmetic Operators

The following table lists the arithmetic operators of the `Fvec` classes and generic syntax. The operators have been divided into standard and advanced operations, which are described in more detail later in this section.

Fvec Arithmetic Operators

Category	Operation	Operators	Generic Syntax
Standard	Addition	`+` `+=`	`R = A + B;` `R += A;`
	Subtraction	`-` `-=`	`R = A - B;` `R -= A;`
	Multiplication	`*` `*=`	`R = A * B;` `R *= A;`
	Division	`/` `/=`	`R = A / B;` `R /= A;`
Advanced	Square Root	`sqrt`	`R = sqrt(A);`
	Reciprocal (Newton-Raphson)	`rcp` `rcp_nr`	`R = rcp(A);` `R = rcp_nr(A);`
	Reciprocal Square Root (Newton-Raphson)	`rsqrt` `rsqrt_nr`	`R = rsqrt(A);` `R = rsqrt_nr(A);`

Standard Arithmetic Operator Usage

The following two tables show the return values for each class of the standard arithmetic operators, which use the syntax styles described earlier in the Return Value Notation section.

Standard Arithmetic Return Value Mapping

R	A	Operators				B	F32vec4	F64vec2	F32vec1
R0:=	A0	+	-	*	/	B0			
R1:=	A1	+	-	*	/	B1			N/A
R2:=	A2	+	-	*	/	B2		N/A	N/A
R3:=	A3	+	-	*	/	B3		N/A	N/A

Arithmetic with Assignment Return Value Mapping

R	Operators				A	F32vec4	F64vec2	F32vec1
R0:=	+=	-=	*=	/=	A0			
R1:=	+=	-=	*=	/=	A1			N/A
R2:=	+=	-=	*=	/=	A2		N/A	N/A

R3:=	+=	-=	*=	/=	A3		N/A	N/A

This table lists standard arithmetic operator syntax and intrinsics.

Standard Arithmetic Operations for Fvec Classes

Operation	Returns	Example Syntax Usage	Intrinsic
Addition	4 floats	`F32vec4 R = F32vec4 A + F32vec4 B;` `F32vec4 R += F32vec4 A;`	`_mm_add_ps`
	2 doubles	`F64vec2 R = F64vec2 A + F32vec2 B;` `F64vec2 R += F64vec2 A;`	`_mm_add_pd`
	1 float	`F32vec1 R = F32vec1 A + F32vec1 B;` `F32vec1 R += F32vec1 A;`	`_mm_add_ss`
Subtraction	4 floats	`F32vec4 R = F32vec4 A - F32vec4 B;` `F32vec4 R -= F32vec4 A;`	`_mm_sub_ps`
	2 doubles	`F64vec2 R - F64vec2 A + F32vec2 B;` `F64vec2 R -= F64vec2 A;`	`_mm_sub_pd`
	1 float	`F32vec1 R = F32vec1 A - F32vec1 B;` `F32vec1 R -= F32vec1 A;`	`_mm_sub_ss`
Multiplication	4 floats	`F32vec4 R = F32vec4 A * F32vec4 B;` `F32vec4 R *= F32vec4 A;`	`_mm_mul_ps`
	2 doubles	`F64vec2 R = F64vec2 A * F364vec2 B;` `F64vec2 R *= F64vec2 A;`	`_mm_mul_pd`
	1 float	`F32vec1 R = F32vec1 A * F32vec1 B;` `F32vec1 R *= F32vec1 A;`	`_mm_mul_ss`
Division	4 floats	`F32vec4 R = F32vec4 A / F32vec4 B;` `F32vec4 R /= F32vec4 A;`	`_mm_div_ps`
	2 doubles	`F64vec2 R = F64vec2 A / F64vec2 B;` `F64vec2 R /= F64vec2 A;`	`_mm_div_pd`
	1 float	`F32vec1 R = F32vec1 A / F32vec1 B;` `F32vec1 R /= F32vec1 A;`	`_mm_div_ss`

Advanced Arithmetic Operator Usage

The following table shows the return values classes of the advanced arithmetic operators, which use the syntax styles described earlier in the Return Value Notation section.

Advanced Arithmetic Return Value Mapping

R	Operators					A	F32vec4	F64vec2	F32vec1
R0:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A0			
R1:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A1			N/A
R2:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A2		N/A	N/A
R3:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A3		N/A	N/A

f :=	add_horizontal			(A0 + A1 + A2 + A3)				N/A	N/A
d :=	add_horizontal			(A0 + A1)			N/A		N/A

This table shows examples for advanced arithmetic operators.

Advanced Arithmetic Operations for Fvec Classes

Returns	Example Syntax Usage	Intrinsic
Square Root		
4 floats	`F32vec4 R = sqrt(F32vec4 A);`	`_mm_sqrt_ps`
2 doubles	`F64vec2 R = sqrt(F64vec2 A);`	`_mm_sqrt_pd`
1 float	`F32vec1 R = sqrt(F32vec1 A);`	`_mm_sqrt_ss`
Reciprocal		
4 floats	`F32vec4 R = rcp(F32vec4 A);`	`_mm_rcp_ps`
2 doubles	`F64vec2 R = rcp(F64vec2 A);`	`_mm_rcp_pd`
1 float	`F32vec1 R = rcp(F32vec1 A);`	`_mm_rcp_ss`
Reciprocal Square Root		
4 floats	`F32vec4 R = rsqrt(F32vec4 A);`	`_mm_rsqrt_ps`
2 doubles	`F64vec2 R = rsqrt(F64vec2 A);`	`_mm_rsqrt_pd`
1 float	`F32vec1 R = rsqrt(F32vec1 A);`	`_mm_rsqrt_ss`
Reciprocal Newton Raphson		

4 floats	`F32vec4 R = rcp_nr(F32vec4 A);`	`_mm_sub_ps`
		`_mm_add_ps`
		`_mm_mul_ps`
		`_mm_rcp_ps`

2 doubles	`F64vec2 R = rcp_nr(F64vec2 A);`	`_mm_sub_pd`
		`_mm_add_pd`
		`_mm_mul_pd`
		`_mm_rcp_pd`

1 float	`F32vec1 R = rcp_nr(F32vec1 A);`	`_mm_sub_ss`
		`_mm_add_ss`
		`_mm_mul_ss`
		`_mm_rcp_ss`

Reciprocal Square Root Newton Raphson		

4 float	`F32vec4 R = rsqrt_nr(F32vec4 A);`	`_mm_sub_pd`
		`_mm_mul_pd`
		`_mm_rsqrt_ps`

2 doubles	`F64vec2 R = rsqrt_nr(F64vec2 A);`	`_mm_sub_pd`
		`_mm_mul_pd`
		`_mm_rsqrt_pd`

1 float	`F32vec1 R = rsqrt_nr(F32vec1 A);`	`_mm_sub_ss`
		`_mm_mul_ss`
		`_mm_rsqrt_ss`

Horizontal Add		
1 float	`float f = add_horizontal(F32vec4 A);`	`_mm_add_ss` `_mm_shuffle_ss`
1 double	`double d = add_horizontal(F64vec2 A);`	`_mm_add_sd` `_mm_shuffle_sd`

Minimum and Maximum Operators

Compute the minimums of the two double precision floating-point values of `A` and `B`.

```
F64vec2 R = simd_min(F64vec2 A, F64vec2 B)
R0 := min(A0,B0);
R1 := min(A1,B1);
```
Corresponding intrinsic: `_mm_min_pd`

Compute the minimums of the four single precision floating-point values of `A` and `B`.

```
F32vec4 R = simd_min(F32vec4 A, F32vec4 B)
R0 := min(A0,B0);
R1 := min(A1,B1);
R2 := min(A2,B2);
R3 := min(A3,B3);
```
Corresponding intrinsic: `_mm_min_ps`

Compute the minimum of the lowest single precision floating-point values of `A` and `B`.

```
F32vec1 R = simd_min(F32vec1 A, F32vec1 B)
R0 := min(A0,B0);
```
Corresponding intrinsic: `_mm_min_ss`

Compute the maximums of the two double precision floating-point values of `A` and `B`.

```
F64vec2 simd_max(F64vec2 A, F64vec2 B)
R0 := max(A0,B0);
R1 := max(A1,B1);
```
Corresponding intrinsic: `_mm_max_pd`

Compute the maximums of the four single precision floating-point values of `A` and `B`.

```
F32vec4 R = simd_man(F32vec4 A, F32vec4 B)
R0 := max(A0,B0);
R1 := max(A1,B1);
R2 := max(A2,B2);
R3 := max(A3,B3);
```
Corresponding intrinsic: `_mm_max_ps`

Compute the maximum of the lowest single precision floating-point values of `A` and `B`.

```
F32vec1 simd_max(F32vec1 A, F32vec1 B)
R0 := max(A0,B0);
```
Corresponding intrinsic: `_mm_max_ss`

Logical Operators

The following table lists the logical operators of the Fvec classes and generic syntax. The logical operators for `F32vec1` classes use only the lower 32 bits.

Fvec Logical Operators Return Value Mapping

Bitwise Operation	Operators	Generic Syntax
AND	& &=	R = A & B; R &= A;
OR	| |=	R = A | B; R |= A;
XOR	^ ^=	R = A ^ B; R ^= A;
andnot	andnot	R = andnot(A);

The following table lists standard logical operators syntax and corresponding intrinsics. Note that there is no corresponding scalar intrinsic for the `F32vec1` classes, which accesses the lower 32 bits of the packed vector intrinsics.

Logical Operations for Fvec Classes

Operation	Returns	Example Syntax Usage	Intrinsic
AND	4 floats	F32vec4 & = F32vec4 A & F32vec4 B; F32vec4 & &= F32vec4 A;	_mm_and_ps
	2 doubles	F64vec2 R = F64vec2 A & F32vec2 B; F64vec2 R &= F64vec2 A;	_mm_and_pd
	1 float	F32vec1 R = F32vec1 A & F32vec1 B; F32vec1 R &= F32vec1 A;	_mm_and_ps
OR	4 floats	F32vec4 R = F32vec4 A | F32vec4 B; F32vec4 R |= F32vec4 A;	_mm_or_ps
	2 doubles	F64vec2 R = F64vec2 A | F32vec2 B; F64vec2 R |= F64vec2 A;	_mm_or_pd
	1 float	F32vec1 R = F32vec1 A | F32vec1 B; F32vec1 R |= F32vec1 A;	_mm_or_ps
XOR	4 floats	F32vec4 R = F32vec4 A ^ F32vec4 B; F32vec4 R ^= F32vec4 A;	_mm_xor_ps
	2 doubles	F64vec2 R = F64vec2 A ^ F364vec2 B; F64vec2 R ^= F64vec2 A;	_mm_xor_pd
	1 float	F32vec1 R = F32vec1 A ^ F32vec1 B; F32vec1 R ^= F32vec1 A;	_mm_xor_ps
ANDNOT	2 doubles	F64vec2 R = andnot(F64vec2 A, F64vec2 B);	_mm_andnot_pd

Compare Operators

The operators described in this section compare the single precision floating-point values of `A` and `B`. Comparison between objects of any `Fvec` class return the same class being compared.

The following table lists the compare operators for the `Fvec` classes.

Compare Operators and Corresponding Intrinsics

Compare For:	Operators	Syntax
Equality	`cmpeq`	`R = cmpeq(A, B)`
Inequality	`cmpneq`	`R = cmpneq(A, B)`
Greater Than	`cmpgt`	`R = cmpgt(A, B)`
Greater Than or Equal To	`cmpge`	`R = cmpge(A, B)`
Not Greater Than	`cmpngt`	`R = cmpngt(A, B)`
Not Greater Than or Equal To	`cmpnge`	`R = cmpnge(A, B)`
Less Than	`cmplt`	`R = cmplt(A, B)`
Less Than or Equal To	`cmple`	`R = cmple(A, B)`
Not Less Than	`cmpnlt`	`R = cmpnlt(A, B)`
Not Less Than or Equal To	`cmpnle`	`R = cmpnle(A, B)`

The mask is set to `0xffffffff` for each floating-point value where the comparison is true and `0x00000000` where the comparison is false. The following table shows the return values for each class of the compare operators, which use the syntax described earlier in the Return Value Notation section.

Compare Operator Return Value Mapping

R	A0	For Any Operators	B	If True	If False	F32vec4	F64vec2	F32vec1
R0:=	(A1 !(A1	`cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]`	B1) B1)	`0xffffffff`	`0x0000000`	X	X	X
R1:=	(A1 !(A1	`cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]`	B2) B2)	`0xffffffff`	`0x0000000`	X	X	N/A
R2:=	(A1	`cmp[eq |`	B3)	`0xffffffff`	`0x0000000`	X	N/A	N/A

	!(A1	lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B3)					
R3:=	A3	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B3) B3)	0xffffffff	0x0000000	X	N/A	N/A

The following table shows examples for arithmetic operators and intrinsics.

Compare Operations for Fvec Classes

Returns	Example Syntax Usage	Intrinsic
Compare for Equality		
4 floats	F32vec4 R = cmpeq(F32vec4 A);	_mm_cmpeq_ps
2 doubles	F64vec2 R = cmpeq(F64vec2 A);	_mm_cmpeq_pd
1 float	F32vec1 R = cmpeq(F32vec1 A);	_mm_cmpeq_ss
Compare for Inequality		
4 floats	F32vec4 R = cmpneq(F32vec4 A);	_mm_cmpneq_ps
2 doubles	F64vec2 R = cmpneq(F64vec2 A);	_mm_cmpneq_pd
1 float	F32vec1 R = cmpneq(F32vec1 A);	_mm_cmpneq_ss
Compare for Less Than		
4 floats	F32vec4 R = cmplt(F32vec4 A);	_mm_cmplt_ps
2 doubles	F64vec2 R = cmplt(F64vec2 A);	_mm_cmplt_pd
1 float	F32vec1 R = cmplt(F32vec1 A);	_mm_cmplt_ss
Compare for Less Than or Equal		
4 floats	F32vec4 R = cmple(F32vec4 A);	_mm_cmple_ps
2 doubles	F64vec2 R = cmple(F64vec2 A);	_mm_cmple_pd
1 float	F32vec1 R = cmple(F32vec1 A);	_mm_cmple_pd
Compare for Greater Than		
4 floats	F32vec4 R = cmpgt(F32vec4 A);	_mm_cmpgt_ps
2 doubles	F64vec2 R = cmpgt(F32vec42 A);	_mm_cmpgt_pd
1 float	F32vec1 R = cmpgt(F32vec1 A);	_mm_cmpgt_ss
Compare for Greater Than or Equal To		
4 floats	F32vec4 R = cmpge(F32vec4 A);	_mm_cmpge_ps

2 doubles	`F64vec2 R = cmpge(F64vec2 A);`	`_mm_cmpge_pd`
1 float	`F32vec1 R = cmpge(F32vec1 A);`	`_mm_cmpge_ss`
Compare for Not Less Than		
4 floats	`F32vec4 R = cmpnlt(F32vec4 A);`	`_mm_cmpnlt_ps`
2 doubles	`F64vec2 R = cmpnlt(F64vec2 A);`	`_mm_cmpnlt_pd`
1 float	`F32vec1 R = cmpnlt(F32vec1 A);`	`_mm_cmpnlt_ss`
Compare for Not Less Than or Equal		
4 floats	`F32vec4 R = cmpnle(F32vec4 A);`	`_mm_cmpnle_ps`
2 doubles	`F64vec2 R = cmpnle(F64vec2 A);`	`_mm_cmpnle_pd`
1 float	`F32vec1 R = cmpnle(F32vec1 A);`	`_mm_cmpnle_ss`
Compare for Not Greater Than		
4 floats	`F32vec4 R = cmpngt(F32vec4 A);`	`_mm_cmpngt_ps`
2 doubles	`F64vec2 R = cmpngt(F64vec2 A);`	`_mm_cmpngt_pd`
1 float	`F32vec1 R = cmpngt(F32vec1 A);`	`_mm_cmpngt_ss`
Compare for Not Greater Than or Equal		
4 floats	`F32vec4 R = cmpnge(F32vec4 A);`	`_mm_cmpnge_ps`
2 doubles	`F64vec2 R = cmpnge(F64vec2 A);`	`_mm_cmpnge_pd`
1 float	`F32vec1 R = cmpnge(F32vec1 A);`	`_mm_cmpnge_ss`

Conditional Select Operators for Fvec Classes

Each conditional function compares single-precision floating-point values of A and B. The C and D parameters are used for return value. Comparison between objects of any Fvec class returns the same class.

Conditional Select Operators for Fvec Classes

Conditional Select for:	Operators	Syntax
Equality	`select_eq`	`R = select_eq(A, B)`
Inequality	`select_neq`	`R = select_neq(A, B)`
Greater Than	`select_gt`	`R = select_gt(A, B)`
Greater Than or Equal To	`select_ge`	`R = select_ge(A, B)`
Not Greater Than	`select_gt`	`R = select_gt(A, B)`
Not Greater Than or Equal To	`select_ge`	`R = select_ge(A, B)`
Less Than	`select_lt`	`R = select_lt(A, B)`
Less Than or Equal To	`select_le`	`R = select_le(A, B)`

Not Less Than	select_nlt	R = select_nlt(A, B)
Not Less Than or Equal To	select_nle	R = select_nle(A, B)

Conditional Select Operator Usage

For conditional select operators, the return value is stored in C if the comparison is true or in D if false. The following table shows the return values for each class of the conditional select operators, using the Return Value Notation described earlier.

Compare Operator Return Value Mapping

R	A0	Operators	B	C	D	F32vec4	F64vec2	F32vec1
R0:=	(A1 !(A1	select [eq | lt | le | gt | ge] select_[ne | nlt | nle | ngt | nge]	B0) B0)	C0 C0	D0 D0	X	X	X
R1:=	(A2 !(A2	select [eq | lt | le | gt | ge] select_[ne | nlt | nle | ngt | nge]	B1) B1)	C1 C1	D1 D1	X	X	N/A
R2:=	(A2 !(A2	select [eq | lt | le | gt | ge] select_[ne | nlt | nle | ngt | nge]	B2) B2)	C2 C2	D2 D2	X	N/A	N/A
R3:=	(A3 !(A3	select [eq | lt | le | gt | ge] select_[ne | nlt | nle | ngt | nge]	B3) B3)	C3 C3	D3 D3	X	N/A	N/A

The following table shows examples for conditional select operations and corresponding intrinsics.

Conditional Select Operations for Fvec Classes

Returns	Example Syntax Usage	Intrinsic
Compare for Equality		
4 floats	F32vec4 R = select_eq(F32vec4 A);	_mm_cmpeq_ps
2 doubles	F64vec2 R = select_eq(F64vec2 A);	_mm_cmpeq_pd
1 float	F32vec1 R = select_eq(F32vec1 A);	_mm_cmpeq_ss
Compare for Inequality		
4 floats	F32vec4 R = select_neq(F32vec4 A);	_mm_cmpneq_ps
2 doubles	F64vec2 R = select_neq(F64vec2 A);	_mm_cmpneq_pd
1 float	F32vec1 R = select_neq(F32vec1 A);	_mm_cmpneq_ss
Compare for Less Than		
4 floats	F32vec4 R = select_lt(F32vec4 A);	_mm_cmplt_ps

2 doubles	F64vec2 R = select_lt(F64vec2 A);	_mm_cmplt_pd
1 float	F32vec1 R = select_lt(F32vec1 A);	_mm_cmplt_ss
Compare for Less Than or Equal		
4 floats	F32vec4 R = select_le(F32vec4 A);	_mm_cmple_ps
2 doubles	F64vec2 R = select_le(F64vec2 A);	_mm_cmple_pd
1 float	F32vec1 R = select_le(F32vec1 A);	_mm_cmple_ps
Compare for Greater Than		
4 floats	F32vec4 R = select_gt(F32vec4 A);	_mm_cmpgt_ps
2 doubles	F64vec2 R = select_gt(F64vec2 A);	_mm_cmpgt_pd
1 float	F32vec1 R = select_gt(F32vec1 A);	_mm_cmpgt_ss
Compare for Greater Than or Equal To		
4 floats	F32vec1 R = select_ge(F32vec4 A);	_mm_cmpge_ps
2 doubles	F64vec2 R = select_ge(F64vec2 A);	_mm_cmpge_pd
1 float	F32vec1 R = select_ge(F32vec1 A);	_mm_cmpge_ss
Compare for Not Less Than		
4 floats	F32vec1 R = select_nlt(F32vec4 A);	_mm_cmpnlt_ps
2 doubles	F64vec2 R = select_nlt(F64vec2 A);	_mm_cmpnlt_pd
1 float	F32vec1 R = select_nlt(F32vec1 A);	_mm_cmpnlt_ss
Compare for Not Less Than or Equal		
4 floats	F32vec1 R = select_nle(F32vec4 A);	_mm_cmpnle_ps
2 doubles	F64vec2 R = select_nle(F64vec2 A);	_mm_cmpnle_pd
1 float	F32vec1 R = select_nle(F32vec1 A);	_mm_cmpnle_ss
Compare for Not Greater Than		
4 floats	F32vec1 R = select_ngt(F32vec4 A);	_mm_cmpngt_ps
2 doubles	F64vec2 R = select_ngt(F64vec2 A);	_mm_cmpngt_pd
1 float	F32vec1 R = select_ngt(F32vec1 A);	_mm_cmpngt_ss
Compare for Not Greater Than or Equal		
4 floats	F32vec1 R = select_nge(F32vec4 A);	_mm_cmpnge_ps
2 doubles	F64vec2 R = select_nge(F64vec2 A);	_mm_cmpnge_pd
1 float	F32vec1 R = select_nge(F32vec1 A);	_mm_cmpnge_ss

Cacheability Support Operations

Stores (non-temporal) the two double-precision, floating-point values of A. Requires a 16-byte aligned address.

```
void store_nta(double *p, F64vec2 A);
```
Corresponding intrinsic: `_mm_stream_pd`

Stores (non-temporal) the four single-precision, floating-point values of `A`. Requires a 16-byte aligned address.

```
void store_nta(float *p, F32vec4 A);
```
Corresponding intrinsic: `_mm_stream_ps`

Debugging

The debug operations do not map to any compiler intrinsics for MMX(TM) technology or Streaming SIMD Extensions. They are provided for debugging programs only. Use of these operations may result in loss of performance, so you should not use them outside of debugging.

Output Operations

The two single, double-precision floating-point values of `A` are placed in the output buffer and printed in decimal format as follows:

```
cout << F64vec2 A;
"[1]:A1 [0]:A0"
```
Corresponding intrinsics: none

The four, single-precision floating-point values of `A` are placed in the output buffer and printed in decimal format as follows:

```
cout << F32vec4 A;
"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```
Corresponding intrinsics: none

The lowest, single-precision floating-point value of `A` is placed in the output buffer and printed.

```
cout << F32vec1 A;
```
Corresponding intrinsics: none

Element Access Operations

```
double d = F64vec2 A[int i]
```

Read one of the two, double-precision floating-point values of `A` without modifying the corresponding floating-point value. Permitted values of `i` are 0 and 1. For example:

If DEBUG is enabled and `i` is not one of the permitted values (0 or 1), a diagnostic message is printed and the program aborts.

```
double d = F64vec2 A[1];
```
Corresponding intrinsics: none

Read one of the four, single-precision floating-point values of `A` without modifying the corresponding floating point value. Permitted values of `i` are 0, 1, 2, and 3. For example:

```
float f = F32vec4 A[int i]
```

If DEBUG is enabled and `i` is not one of the permitted values (0-3), a diagnostic message is printed and the program aborts.

```
float f = F32vec4 A[2];
```
Corresponding intrinsics: none

Element Assignment Operations

```
F64vec4 A[int i] = double d;
```

Modify one of the two, double-precision floating-point values of `A`. Permitted values of `int i` are 0 and 1. For example:

```
F32vec4 A[1] = double d;
```
F32vec4 A[int i] = float f;

Modify one of the four, single-precision floating-point values of `A`. Permitted values of `int i` are 0, 1, 2, and 3. For example:

If DEBUG is enabled and `int i` is not one of the permitted values (0-3), a diagnostic message is printed and the program aborts.

```
F32vec4 A[3] = float f;
```
Corresponding intrinsics: none.

Load and Store Operators

Loads two, double-precision floating-point values, copying them into the two, floating-point values of `A`. No assumption is made for alignment.

```
void loadu(F64vec2 A, double *p)
```
Corresponding intrinsic: `_mm_loadu_pd`

Stores the two, double-precision floating-point values of `A`. No assumption is made for alignment.

```
void storeu(float *p, F64vec2 A);
```
Corresponding intrinsic: `_mm_storeu_pd`

Loads four, single-precision floating-point values, copying them into the four floating-point values of `A`. No assumption is made for alignment.

```
void loadu(F32vec4 A, double *p)
```
Corresponding intrinsic: `_mm_loadu_ps`

Stores the four, single-precision floating-point values of A. No assumption is made for alignment.

```
void storeu(float *p, F32vec4 A);
```
Corresponding intrinsic: _mm_storeu_ps

Unpack Operators for Fvec Operators

Selects and interleaves the lower, double-precision floating-point values from A and B.

```
F64vec2 R = unpack_low(F64vec2 A, F64vec2 B);
```
Corresponding intrinsic: _mm_unpacklo_pd(a, b)

Selects and interleaves the higher, double-precision floating-point values from A and B.

```
F64vec2 R = unpack_high(F64vec2 A, F64vec2 B);
```
Corresponding intrinsic: _mm_unpackhi_pd(a, b)

Selects and interleaves the lower two, single-precision floating-point values from A and B.

```
F32vec4 R = unpack_low(F32vec4 A, F32vec4 B);
```
Corresponding intrinsic: _mm_unpacklo_ps(a, b)

Selects and interleaves the higher two, single-precision floating-point values from A and B.

```
F32vec4 R = unpack_high(F32vec4 A, F32vec4 B);
```
Corresponding intrinsic: _mm_unpackhi_ps(a, b)

Move Mask Operator

Creates a 2-bit mask from the most significant bits of the two, double-precision floating-point values of A, as follows:

```
int i = move_mask(F64vec2 A)
i := sign(a1)<<1 | sign(a0)<<0
```
Corresponding intrinsic: _mm_movemask_pd

Creates a 4-bit mask from the most significant bits of the four, single-precision floating-point values of A, as follows:

```
int i = move_mask(F32vec4 A)
i := sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 | sign(a0)<<0
```
Corresponding intrinsic: _mm_movemask_ps

Classes Quick Reference

This appendix contains tables listing the class, functionality, and corresponding intrinsics for each class in the Intel C++ Class Libraries for SIMD Operations. The following table lists all Intel C++ Compiler intrinsics that are not implemented in the C++ SIMD classes.

Logical Operators: Corresponding Intrinsics and Classes

Operators	Corresponding Intrinsic	I128vec1, I64vec2, I32vec4, I16vec8, I8vec16	I64vec, I32vec, I16vec, I8vec8	F64vec2	F32vec4	F32vec1
&, &=	_mm_and_[x]	si128	si64	pd	ps	ps
|, |=	_mm_or_[x]	si128	si64	pd	ps	ps
^, ^=	_mm_xor_[x]	si128	si64	pd	ps	ps
Andnot	_mm_andnot_[x]	si128	si64	pd	N/A	N/A

Arithmetic: Corresponding Intrinsics and Classes, Part 1

Operators	Corresponding Intrinsic	I64vec2	I32vec4	I16vec8	I8vec16
+, +=	_mm_add_[x]	epi64	epi32	epi16	epi8
-, -=	_mm_sub_[x]	epi64	epi32	epi16	epi8
*, *=	_mm_mullo_[x]	N/A	N/A	epi16	N/A
/, /=	_mm_div_[x]	N/A	N/A	N/A	N/A
mul_high	_mm_mulhi_[x]	N/A	N/A	epi16	N/A
mul_add	_mm_madd_[x]	N/A	N/A	epi16	N/A
sqrt	_mm_sqrt_[x]	N/A	N/A	N/A	N/A
rcp	_mm_rcp_[x]	N/A	N/A	N/A	N/A
rcp_nr	_mm_rcp_[x] _mm_add_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	N/A
rsqrt	_mm_rsqrt_[x]	N/A	N/A	N/A	N/A
rsqrt_nr	_mm_rsqrt_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	N/A

Arithmetic: Corresponding Intrinsics and Classes, Part 2

Operators	Corresponding Intrinsic	I32vec2	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
+, +=	_mm_add_[x]	pi32	pi16	pi8	pd	ps	ss
-, -=	_mm_sub_[x]	pi32	pi16	pi8	pd	ps	ss
*, *=	_mm_mullo_[x]	N/A	pi16	N/A	pd	ps	ss
/, /=	_mm_div_[x]	N/A	N/A	N/A	pd	ps	ss
mul_high	_mm_mulhi_[x]	N/A	pi16	N/A	N/A	N/A	N/A
mul_add	_mm_madd_[x]	N/A	pi16	N/A	N/A	N/A	N/A
sqrt	_mm_sqrt_[x]	N/A	N/A	N/A	pd	ps	ss
rcp	_mm_rcp_[x]	N/A	N/A	N/A	pd	ps	ss
rcp_nr	_mm_rcp_[x] _mm_add_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	pd	ps	ss
rsqrt	_mm_rsqrt_[x]	N/A	N/A	N/A	pd	ps	ss
rsqrt_nr	_mm_rsqrt_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	pd	ps	ss

Shift Operators: Corresponding Intrinsics and Classes, Part 1

Operators	Corresponding Intrinsic	I128vec1	I64vec2	I32vec4	I16vec8	I8vec16
>>,>>=	_mm_srl_[x] _mm_srli_[x] _mm_sra__[x] _mm_srai_[x]	N/A N/A N/A N/A	epi64 epi64 N/A N/A	epi32 epi32 epi32 epi32	epi16 epi16 epi16 epi16	N/A N/A N/A N/A
<<, <<=	_mm_sll_[x] _mm_slli_[x]	N/A N/A	epi64 epi64	epi32 epi32	epi16 epi16	N/A N/A

Shift Operators: Corresponding Intrinsics and Classes, Part 2

Operators	Corresponding Intrinsic	I64vec1	I32vec2	I16vec4	I8vec8
>>,>>=	_mm_srl_[x] _mm_srli_[x] _mm_sra__[x] _mm_srai_[x]	si64 si64 N/A N/A	pi32 pi32 pi32 pi32	pi16 pi16 pi16 pi16	N/A N/A N/A N/A
<<, <<=	_mm_sll_[x] _mm_slli_[x]	si64 si64	pi32 pi32	pi16 pi16	N/A N/A

Comparison Operators: Corresponding Intrinsics and Classes, Part 1

Operators	Corresponding Intrinsic	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8
cmpeq	_mm_cmpeq_[x]	epi32	epi16	epi8	pi32	pi16	pi8
cmpneq	_mm_cmpeq_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64
cmpgt	_mm_cmpgt_[x]	epi32	epi16	epi8	pi32	pi16	pi8
cmpge	_mm_cmpge_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64
cmplt	_mm_cmplt_[x]	epi32	epi16	epi8	pi32	pi16	pi8
cmple	_mm_cmple_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64
cmpngt	_mm_cmpngt_[x]	epi32	epi16	epi8	pi32	pi16	pi8
cmpnge	_mm_cmpnge_[x]	N/A	N/A	N/A	N/A	N/A	N/A
cmnpnlt	_mm_cmpnlt_[x]	N/A	N/A	N/A	N/A	N/A	N/A
cmpnle	_mm_cmpnle_[x]	N/A	N/A	N/A	N/A	N/A	N/A

* Note that _mm_andnot_[y] intrinsics do not apply to the fvec classes.

Comparison Operators: Corresponding Intrinsics and Classes, Part 2

Operators	Corresponding Intrinsic	F64vec2	F32vec4	F32vec1
cmpeq	_mm_cmpeq_[x]	pd	ps	ss
cmpneq	_mm_cmpeq_[x] _mm_andnot_[y]*	pd	ps	ss
cmpgt	_mm_cmpgt_[x]	pd	ps	ss
cmpge	_mm_cmpge_[x] _mm_andnot_[y]*	pd	ps	ss
cmplt	_mm_cmplt_[x]	pd	ps	ss
cmple	_mm_cmple_[x] _mm_andnot_[y]*	pd	ps	ss
cmpngt	_mm_cmpngt_[x]	pd	ps	ss
cmpnge	_mm_cmpnge_[x]	pd	ps	ss
cmnpnlt	_mm_cmpnlt_[x]	pd	ps	ss
cmpnle	_mm_cmpnle_[x]	pd	ps	ss

Conditional Select Operators: Corresponding Intrinsics and Classes, Part 1

Operators	Corresponding Intrinsic	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8
select_eq	_mm_cmpeq_[x]	epi32	epi16	epi8	pi32	pi16	pi8

	_mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	si128 si128 si128	si128 si128 si128	si128 si128 si128	si64 si64 si64	si64 si64 si64	si64 si64 si64
select_neq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_gt	_mm_cmpgt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_ge	_mm_cmpge_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_lt	_mm_cmplt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_le	_mm_cmple_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_ngt	_mm_cmpgt_[x]	N/A	N/A	N/A	N/A	N/A	N/A
select_nge	_mm_cmpge_[x]	N/A	N/A	N/A	N/A	N/A	N/A
select_nlt	_mm_cmplt_[x]	N/A	N/A	N/A	N/A	N/A	N/A
select_nle	_mm_cmple_[x]	N/A	N/A	N/A	N/A	N/A	N/A

* Note that _mm_andnot_[y] intrinsics do not apply to the fvec classes.

Conditional Select Operators: Corresponding Intrinsics and Classes, Part 2

Operators	Corresponding Intrinsic	F64vec2	F32vec4	F32vec1
select_eq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_neq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_gt	_mm_cmpgt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_ge	_mm_cmpge_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss

select_lt	_mm_cmplt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_le	_mm_cmple_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_ngt	_mm_cmpgt_[x]	pd	ps	ss
select_nge	_mm_cmpge_[x]	pd	ps	ss
select_nlt	_mm_cmplt_[x]	pd	ps	ss
select_nle	_mm_cmple_[x]	pd	ps	ss

Packing and Unpacking Operators: Corresponding Intrinsics and Classes, Part 1

Operators	Corresponding Intrinsic	I64vec2	I32vec4	I16vec8	I8vec16	I32vec2
unpack_high	_mm_unpackhi_[x]	epi64	epi32	epi16	epi8	pi32
unpack_low	_mm_unpacklo_[x]	epi64	epi32	epi16	epi8	pi32
pack_sat	_mm_packs_[x]	N/A	epi32	epi16	N/A	pi32
packu_sat	_mm_packus_[x]	N/A	N/A	epi16	N/A	N/A
sat_add	_mm_adds_[x]	N/A	N/A	epi16	epi8	N/A
sat_sub	_mm_subs_[x]	N/A	N/A	epi16	epi8	N/A

Packing and Unpacking Operators: Corresponding Intrinsics and Classes, Part 2

Operators	Corresponding Intrinsic	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
unpack_high	_mm_unpackhi_[x]	pi16	pi8	pd	ps	N/A
unpack_low	_mm_unpacklo_[x]	pi16	pi8	pd	ps	N/A
pack_sat	_mm_packs_[x]	pi16	N/A	N/A	N/A	N/A
packu_sat	_mm_packus_[x]	pu16	N/A	N/A	N/A	N/A
sat_add	_mm_adds_[x]	pi16	pi8	pd	ps	ss
sat_sub	_mm_subs_[x]	pi16	pi8	pi16	pi8	pd

Conversions Operators: Corresponding Intrinsics and Classes

Operators	Corresponding Intrinsic
F64vec2ToInt	_mm_cvttsd_si32

F32vec4ToF64vec2	_mm_cvtps_pd
F64vec2ToF32vec4	_mm_cvtpd_ps
IntToF64vec2	_mm_cvtsi32_sd
F32vec4ToInt	_mm_cvtt_ss2si
F32vec4ToIs32vec2	_mm_cvttps_pi32
IntToF32vec4	_mm_cvtsi32_ss
Is32vec2ToF32vec4	_mm_cvtpi32_ps

Programming Example

This sample program uses the `F32vec4` class to average the elements of a 20 element floating point array.

```
// Include Streaming SIMD Extension Class Definitions
#include <fvec.h>

// Shuffle any 2 single precision floating point from a
// into low 2 SP FP and shuffle any 2 SP FP from b
// into high 2 SP FP of destination
#define SHUFFLE(a,b,i) (F32vec4) mm shuffle ps(a,b,i)
#include <stdio.h>
#define SIZE 20

// Global variables
float result;
 MM ALIGN16 float array[SIZE];

//*****************************************************
// Function: Add20ArrayElements
// Add all the elements of a 20 element array
//*****************************************************

void Add20ArrayElements (F32vec4 *array, float *result){
   F32vec4 vec0, vec1;
   vec0 =  mm load ps ((float *) array); // Load array's first 4
floats

   //*****************************************************
   // Add all elements of the array, 4 elements at a time
   //*****************************************************

   vec0 += array[1]; // Add elements 5-8
   vec0 += array[2]; // Add elements 9-12
   vec0 += array[3]; // Add elements 13-16
   vec0 += array[4]; // Add elements 17-20

   //*****************************************************
   // There are now 4 partial sums.
   // Add the 2 lowers to the 2 raises,
   // then add those 2 results together
   //*****************************************************

   vec1 = SHUFFLE(vec1, vec0, 0x40);
   vec0 += vec1;
   vec1 = SHUFFLE(vec1, vec0, 0x30);
   vec0 += vec1;
   vec0 = SHUFFLE(vec0, vec0, 2);
    mm store ss (result, vec0); // Store the final sum
}

void main(int argc, char *argv[]){
   int i;
   // Initialize the array
   for (i=0; i < SIZE; i++)
   {
      array[i] = (float) i;
   }

   // Call function to add all array elements
   Add20ArrayElements (array, &result);

   // Print average array element value
   printf ("Average of all array values = %f\n", result/20.);
   printf ("The correct answer is %f\n\n\n", 9.5);
}
```