Intel® C++ Compiler for Linux* Reference

Document number: 307777-001

Disclaimer and Legal Information

The information in this manual is subject to change without notice and Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. This document and the software described in it are furnished under license and may only be used or copied in accordance with the terms of the license. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. The information in this document is provided in connection with Intel products and should not be construed as a commitment by Intel Corporation.

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. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

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 software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

Intel, the Intel logo, Intel SpeedStep, Intel NetBurst, Intel NetStructure, MMX, i386, i486, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2,Celeron, Intel Centrino, Intel Xeon, Intel XScale, Itanium, Pentium, Pentium II Xeon, Pentium III Xeon, Pentium M, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Table Of Contents

Introduction to Compiler Reference

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

- Compiler Limits
- Key Files
- Predefined Macros
- Intel® Math Library
- Intel® C++ Intrinsics Reference
- Intel® C++ Class Libraries

Compiler Limits

The following table shows the size or number of each item that the compiler can process. All capacities shown in the table are tested values; the actual number can be greater than the number shown.

Item	Tested Values
Control structure nesting (block nesting)	512
Conditional compilation nesting	512
Declarator modifiers	512
Parenthesis nesting levels	512
Significant characters, internal identifier	2048
External identifier name length	64K
Number of external identifiers/file	128K
Number of identifiers in a single block	2048
Number of macros simultaneously defined	128K
Number of parameters to a function call	512
Number of parameters per macro	512
Number of characters in a string	128K
Bytes in an object	512K
Include file nesting depth	512
Case labels in a switch	32K

Item	Tested Values
Members in one structure or union	32K
Enumeration constants in one enumeration	8192
Levels of structure nesting	320
Size of arrays (IA-32 only)	2 GB

Key Files Summary for IA-32 Compiler

The following tables list and briefly describe files that are installed for use by the IA-32 version of the compiler.

/bin Files

File	Description
`codecov`	Code-coverage tool
`iccvars.sh` `iccvars.csh`	Batch file to set environment variables
`icc` `icpc`	Scripts that check for license file and call compiler driver
`iccbin` `icpcbin`	Compiler drivers
`mcpcom`	Intel® C++ Compiler
`profmerge`	Utility used for Profile Guided Optimizations
`proforder`	Utility used for Profile Guided Optimizations
`tselect`	Test-prioritization tool
`xiar`	Tool used for Interprocedural Optimizations
`xild`	Tool used for Interprocedural Optimizations

/include Files

File	Description
dvec.h	SSE 2 intrinsics for Class Libraries
emm_func.h	Header file for SSE2 intrinsics (used by emmintrin.h)
emmintrin.h	Principal header file for SSE2 intrinsics
float.h	IEEE 754 version of standard float.h
fvec.h	SSE intrinsics for Class Libraries
iso646.h	Standard header file
ivec.h	MMX(TM) instructions intrinsics for Class Libraries
limits.h	Standard header file
mathf.h	Principal header file for legacy Intel Math Library
mathimf.h	Principal header file for current Intel Math Library
mmintrin.h	Intrinsics for MMX instructions
omp.h	Principal header file OpenMP*
omp_lib.h	Header file for OpenMP
pgouser.h	For use in the instrumentation compilation phase of profile-guided optimizations
pmmintrin.h	Principal header file SSE3 intrinsics
proto.h	
sse2mmx.h	Principal header file for Streaming SIMD Extensions 2 intrinsics
stdarg.h	Replacement header for standard stdarg.h
stdbool.h	Defines _Bool keyword
stddef.h	Standard header file
syslimits.h	

`varargs.h`	Replacement header for standard `varargs.h`
`xarg.h`	Header file used by `stdargs.h` and `varargs.h`
`xmm_func.h.h`	Header file for Streaming SIMD Extensions
`xmm_utils.h`	Utilities for Streaming SIMD Extensions
`xmmintrin.h`	Principal header file for Streaming SIMD Extensions intrinsics

/lib Files

Library	Description
`libguide.a` `libguide.so`	For OpenMP* implementation
`libguide_stats.a` `libguide_stats.so`	OpenMP static library for the parallelizer tool with performance statistics and profile information
`libompstub.a`	Library that resolves references to OpenMP subroutines when OpenMP is not in use
`libsvml.a`	Short vector math library
`libirc.a`	Intel support library for PGO and CPU dispatch
`libimf.a`	Intel math library
`libimf.so`	Intel math library

`libcprts.a`
`libcprts.so`
`libcprts.so.3`
	Dinkumware* C++ Library

`libunwind.a`
`libunwind.so`
`libunwind.so.3`
	Unwinder library

`libcxa.a`
`libcxa.so`
`libcxa.so.3`
	Intel run time support for C++ features

`libcxaguard.a`
`libcxaguard.so`
`libcxaguard.so.3`
	Used for interoperability support with the `-cxxlib-gcc` option.
	See gcc Interoperability.

Key Files Summary for Itanium® Compiler

The following tables list and briefly describe files that are installed for use by the Itanium® compiler.

/bin Files

File	Description
`codecov`	Code-coverage tool
`iccvars.sh`	Batch file to set environment variables
`icc.cfg`	Configuration file for use from command line
`icc` `icpc`	Scripts that check for license file and call compiler driver
`iccbin` `icpcbin`	Compiler drivers
`mcpcom`	Intel® C++ Compiler
`profmerge`	Utility used for Profile Guided Optimizations
`proforder`	Utility used for Profile Guided Optimizations
`tselect`	Test-prioritization tool
`xiar`	Tool used for Interprocedural Optimizations
`xild`	Tool used for Interprocedural Optimizations

/include Files

File	Description
`emmintrin.h`	Principal header file for SSE2 intrinsics
`float.h`	IEEE 754 version of standard `float.h`
`fvec.h`	SSE intrinsics for Class Libraries
`ia64intrin.h`	
`ia64regs.h`	Standard header file
`iso646.h`	Standard header file

ivec.h	MMX(TM) instructions intrinsics for Class Libraries
limits.h	Standard header file
mathimf.h	Principal header file for current Intel Math Library
mmintrin.h	Intrinsics for MMX instructions
omp.h	Principal header file OpenMP*
pgouser.h	For use in the instrumentation compilation phase of profile-guided optimizations
proto.h	
sse2mmx.h	Principal header file for Streaming SIMD Extensions 2 intrinsics
stdarg.h	Replacement header for standard stdarg.h
stdbool.h	Defines _Bool keyword
stddef.h	Standard header file
syslimits.h	
varargs.h	Replacement header for standard varargs.h
xarg.h	Header file used by stdargs.h and varargs.h
xmmintrin.h	Principal header file for Streaming SIMD Extensions intrinsics

/lib Files

File	Description
libcprts.a	C++ standard language library
libcxa.so	C++ language library indicating I/O data location
libirc.a	Intel-specific library (optimizations)
libm.a	Math library
libguide.a	OpenMP library
libguide.so	Shared OpenMP library

`libmofl.a`	Multiple Object Format Library, used by the Intel assembler
`libmofl.so`	Shared Multiple Object Format Library, used by the Intel assembler
`libunwinder.a`	Unwinder library
`libintrins.a`	Intrinsic functions library

Remarks, Warnings, and Errors

This topic describes compiler remarks, warnings, and errors. The Intel® C++ Compiler displays these messages, along with the erroneous source line, on the standard output.

Remarks

Remark messages report common but sometimes unconventional use of C or C++. The compiler does not print or display remarks unless you specify the `-w2` option. Remarks do not stop translation or linking. Remarks do not interfere with any output files. The following are some representative remark messages:

- `function declared implicitly`
- `type qualifiers are meaningless in this declaration`
- `controlling expression is constant`

Warnings

Warning messages report legal but questionable use of the C or C++. The compiler displays warnings by default. You can suppress all warning messages with the `-w` compiler option. Warnings do not stop translation or linking. Warnings do not interfere with any output files. The following are some representative warning messages:

- `declaration does not declare anything`
- `pointless comparison of unsigned integer with zero`
- `possible use of = where == was intended`

Additional Warnings

This version of Intel C++ Compiler includes support for the following options:

Option	Result
`-W[no-]missing-prototypes`	Warn for missing prototypes
`-W[no-]pointer-arith`	Warn for questionable pointer arithmetic
`-W[no-]uninitialized`	Warn if a variable is used before being initialized
`-W[no-]deprecated`	Display warnings related to deprecated features
`-W[no-]abi`	Warn if generated code is not C++ ABI compliant
`-W[no-]unused-function`	Warn if declared function is not used
`-W[no-]unknown-pragmas`	Warn if an unknown `#pragma` directive is used
`-W[no-]main`	Warn if return type of main is not expected
`-W[no-]comment[s]`	Warn when /* appears in the middle of a /* */ comment
`-W[no-]return-type`	Warn when a function uses the default `int` return type Warn when a return statement is used in a void function

Errors

These messages report syntactic or semantic misuse of C or C++. The compiler always displays error messages. Errors suppress object code for the module containing the error and prevent linking, but they allow parsing to continue to detect other possible errors. Some representative error messages are:

- `missing closing quote`
- `expression must have arithmetic type`
- `expected a ";"`

Option Summary

Use the following compiler options to control remarks, warnings, and errors:

Option	Result
`-w`	Suppress all warnings
`-w0`	Display only errors
`-w1`	Display only errors and warnings
`-w2`	Display errors, warnings, and remarks
`-Wbrief`	Display brief one-line diagnostics
`-Wcheck`	Enable more strict diagnostics

ANSI Standard Predefined Macros

The ANSI/ISO standard for the C language requires that certain predefined macros be supplied with conforming compilers. The following table lists the macros that the Intel C++ Compiler supplies in accordance with this standard:

The compiler includes predefined macros in addition to those required by the standard.

Macro	Value
`__DATE__`	The date of compilation as a string literal in the form `Mmm dd yyyy`.
`__FILE__`	A string literal representing the name of the file being compiled.
`__LINE__`	The current line number as a decimal constant.
`__STDC__`	The name `__STDC__` is defined when compiling a C translation unit.
`__STDC_HOSTED__`	1
`__TIME__`	The time of compilation as a string literal in the form `hh:mm:ss`.

See Also

Additional Predefined Macros

Additional Predefined Macros

The Intel® C++ Compiler supports the predefined macros listed in the following table. The compiler also includes predefined macros specified by the ISO/ANSI standard.

Macro Name	Value	Architecture
`__BASE_FILE__`	Name of source file	All
`__cplusplus`	1	All
`__EDG__`	1	All
`__EDG_VERSION__`	303	All
`__ELF__`	1	All
`__EXCEPTIONS`	Defined when `-fno-exceptions` is not used.	IA-32 only
`__GNUC__`	2 - if gcc version is less than 3.2 3 - if gcc version is 3.2, 3.3, or 3.4	All
`__gnu_linux__`	1	All
`__GNUC_MINOR__`	95 - if gcc version is less than 3.2 2 - if gcc version is 3.2 3 - if gcc version is 3.3 4 - if gcc version is 3.4	All
`__GNUC_PATCHLEVEL__`	0	All
`__GXX_ABI_VERSION`	102	All
`__i386`	1	IA-32 only
`__i386__`	1	IA-32 only
`i386`	1	IA-32 only
`__ia64`	1	Itanium® architecture only

Macro Name	Value	Architecture
`__ia64__`	1	Itanium architecture only
`ia64`	1	Itanium architecture only
`__INTEL_COMPILER`	900	All
`__INTEL_COMPILER_BUILD_DATE`	`YYYYMMDD`	All
`__INTEL_CXXLIB_ICC`	1 when `-cxxlib-icc` option is specified during compilation.	All
`__INTEL_RTTI__`	1 when `-fno-rtti` is not specified.	All
`__INTEL_STRICT_ANSI__`	1 when `-strict-ansi` is specified.	All
`_INTEGRAL_MAX_BITS`	64	Itanium architecture only
`__itanium__`	1	Itanium architecture only
`__linux`	1	All
`__linux__`	1	All
`linux`	1	All
`__LONG_DOUBLE_SIZE__`	80	IA-32 only
`__LONG_MAX__`	9223372036854775807L	Itanium architecture only
`__lp64`	1	Itanium architecture only

Macro Name	Value	Architecture
__LP64__	1	Itanium architecture only
_LP64	1	Itanium architecture only
__NO_INLINE__	1	All
__NO_MATH_INLINES	1	All
__NO_STRING_INLINES	1	All
__OPTIMIZE__	1	All
__PIC__	1 when -fPIC is used.	All
__pic__	1 when -fPIC is used.	All
_PGO_INSTRUMENT	1 when -prof-gen[x] is used.	All
__PTRDIFF_TYPE__	int on IA-32 long on Itanium architecture	All
__REGISTER_PREFIX__	(no value)	All
__SIGNED_CHARS__	1	All
__SIZE_TYPE__	unsigned on IA-32 unsigned long on Itanium architecture	All
__unix	1	All
__unix__	1	All
unix	1	All
__USER_LABEL_PREFIX__	(no value)	All

Macro Name	Value	Architecture
`__VERSION__`	"Intel(R) C++ gcc 3.0 mode"	All
`__WCHAR_T`	1	All
`__WCHAR_TYPE__`	`long int` on IA-32 `int` on Itanium architecture	All
`__WINT_TYPE__`	`unsigned int`	All

See Also

- -D Compiler Option
- -U Compiler Option
- ANSI Standrard Predefined Macros

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 `-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 = 3.1415926535897932384/4.0;

// 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= 3.14159265358979238/4.0;

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 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 `-c99` compiler option when compiling programs that require support for `_Complex` data types.

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, $e^x - 1$.

`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 frexpd(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.

Calling interface:

`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

```
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`: ERANGE, for underflow conditions

Calling interface:

```
double erfc(double x);
long double erfcl(long double x);
float erfcf(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^{31} (for IA-32 and Intel® EM64T) or congruent modulo 2^{24} (for Itanium®-based systems) of the integral quotient of `x/y`, where n is an implementation-defined integer greater than or equal to 3.

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 e^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 2^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 10^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 z);
long double _Complex cisl(long double z);
float _Complex cisf(float z);
```

CISD

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

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

The Intel® Pentium® 4 processor and other Intel processors have instructions to enable development of optimized multimedia applications. The instructions are implemented through extensions to previously implemented instructions. This technology uses the single instruction, multiple data (SIMD) technique. By processing data elements in parallel, applications with media-rich bit streams are able to significantly improve performance using SIMD instructions. The Intel® Itanium® processor also supports these instructions.

The most direct way to use these instructions is to inline the assembly language instructions into your source code. However, this can be time-consuming and tedious, and assembly language inline programming is not supported on all compilers. Instead, Intel provides easy implementation through the use of API extension sets referred to as intrinsics.

Intrinsics are special coding extensions that allow using the syntax of C function calls and C variables instead of hardware registers. Using these intrinsics frees programmers from having to program in assembly language and manage registers. In addition, the compiler optimizes the instruction scheduling so that executables run faster.

In addition, the native intrinsics for the Itanium processor give programmers access to Itanium instructions that cannot be generated using the standard constructs of the C and C++ languages. The Intel® C++ Compiler also supports general purpose intrinsics that work across all IA-32 and Itanium-based platforms.

For more information on intrinsics, please refer to the following publications:

Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual, Intel Corporation, doc. number 243191

Intrinsics Availability on Intel Processors

Processors:	MMX(TM) Technology Intrinsics	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium Processor Instructions
Itanium Processor	X	X	N/A	X
Pentium 4 Processor	X	X	X	N/A
Pentium III Processor	X	X	N/A	N/A
Pentium II Processor	X	N/A	N/A	N/A
Pentium with MMX Technology	X	N/A	N/A	N/A
Pentium Pro Processor	N/A	N/A	N/A	N/A
Pentium Processor	N/A	N/A	N/A	N/A

Processors:	MMX(TM) Technology Intrinsics	Streaming SIMD Extensions	Streaming SIMD Extensions 2
Pentium 4 Processor	X	X	X

Pentium III Processor	X	X	N/A
Pentium II Processor	X	N/A	N/A
Pentium with MMX Technology	X	N/A	N/A
Pentium Pro Processor	N/A	N/A	N/A
Pentium Processor	N/A	N/A	N/A

Benefits of Using Intrinsics

The major benefit of using intrinsics is that you now have access to key features that are not available using conventional coding practices. Intrinsics enable you to code with the syntax of C function calls and variables instead of assembly language. Most MMX(TM) technology, Streaming SIMD Extensions, and Streaming SIMD Extensions 2 intrinsics have a corresponding C intrinsic that implements that instruction directly. This frees you from managing registers and enables the compiler to optimize the instruction scheduling.

The MMX technology and Streaming SIMD Extension instructions use the following new features:

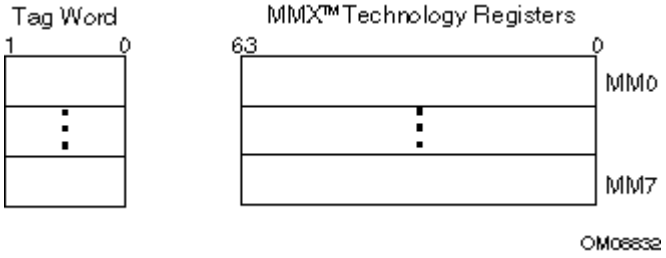
- new Registers--Enable packed data of up to 128 bits in length for optimal SIMD processing
- new Data Types--Enable packing of up to 16 elements of data in one register

The Streaming SIMD Extensions 2 intrinsics are defined only for IA-32, not for Itanium®-based systems. Streaming SIMD Extensions 2 operate on 128 bit quantities - 2 64-bit double precision floating point values. The Itanium architecture does not support parallel double precision computation, so Streaming SIMD Extensions 2 are not implemented on Itanium-based systems.

New Registers

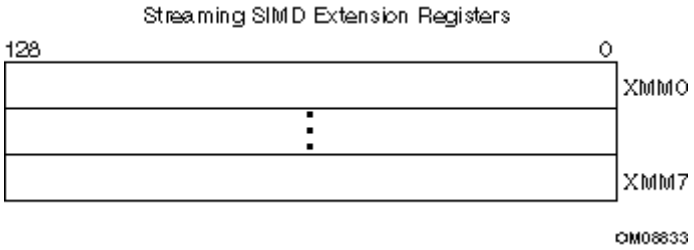
A key feature provided by the architecture of the processors are new register sets. The MMX instructions use eight 64-bit registers (`mm0` to `mm7`) which are aliased on the floating-point stack registers.

MMX(TM) Technology Registers



Streaming SIMD Extensions Registers

The Streaming SIMD Extensions use eight 128-bit registers (`xmm0` to `xmm7`).



These new data registers enable the processing of data elements in parallel. Because each register can hold more than one data element, the processor can process more than one data element simultaneously. This processing capability is also known as single-instruction multiple data processing (SIMD).

For each computational and data manipulation instruction in the new extension sets, there is a corresponding C intrinsic that implements that instruction directly. This frees you from managing registers and assembly programming. Further, the compiler optimizes the instruction scheduling so that your executable runs faster.

Note

The `MM` and `XMM` registers are the SIMD registers used by the IA-32 platforms to implement MMX technology and Streaming SIMD Extensions/Streaming SIMD Extensions 2 intrinsics. On the Itanium-based platforms, the MMX and Streaming SIMD Extension intrinsics use the 64-bit general registers and the 64-bit significand of the 80-bit floating-point register.

Data Types

Intrinsic functions use four new C data types as operands, representing the new registers that are used as the operands to these intrinsic functions. The following table shows the data type availability marked with "X".

New Data Types Available

New Data Type	MMX(TM) Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2
`__m64`	X	X	X
`__m128`	N/A	X	X
`__m128d`	N/A	N/A	X
`__m128i`	N/A	N/A	X

__m64 Data Type

The `__m64` data type is used to represent the contents of an MMX register, which is the register that is used by the MMX technology intrinsics. The `__m64` data type can hold eight 8-bit values, four 16-bit values, two 32-bit values, or one 64-bit value.

__m128 Data Types

The `__m128` data type is used to represent the contents of a Streaming SIMD Extension register used by the Streaming SIMD Extension intrinsics. The `__m128` data type can hold four 32-bit floating values.

The `__m128d` data type can hold two 64-bit floating-point values.

The `__m128i` data type can hold sixteen 8-bit, eight 16-bit, four 32-bit, or two 64-bit integer values.

The compiler aligns `__m128` local and global data to 16-byte boundaries on the stack. To align `integer`, `float`, or `double` arrays, you can use the declspec statement.

New Data Types Usage Guidelines

Since these new data types are not basic ANSI C data types, you must observe the following usage restrictions:

- Use new data types only on either side of an assignment, as a return value, or as a parameter. You cannot use it with other arithmetic expressions (+, -, etc).
- Use new data types as objects in aggregates, such as unions to access the byte elements and structures.

- Use new data types only with the respective intrinsics described in this documentation. The new data types are supported on both sides of an assignment statement: as parameters to a function call, and as a return value from a function call.

Naming and Usage Syntax

Most of the intrinsic names use a notational convention as follows:

`_mm_<intrin_op>_<suffix>`

`<intrin_op>`	Indicates the intrinsics basic operation; for example, `add` for addition and `sub` for subtraction.
`<suffix>`	Denotes the type of data operated on by the instruction. The first one or two letters of each suffix denotes whether the data is packed (`p`), extended packed (`ep`), or scalar (`s`). The remaining letters denote the type: • `s` single-precision floating point • `d` double-precision floating point • `i128` signed 128-bit integer • `i64` signed 64-bit integer • `u64` unsigned 64-bit integer • `i32` signed 32-bit integer • `u32` unsigned 32-bit integer • `i16` signed 16-bit integer • `u16` unsigned 16-bit integer • `i8` signed 8-bit integer • `u8` unsigned 8-bit integer

A number appended to a variable name indicates the element of a packed object. For example, `r0` is the lowest word of `r`. Some intrinsics are "composites" because they require more than one instruction to implement them.

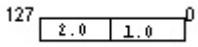
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
double a[2] = {1.0, 2.0};
__m128d t = _mm_load_pd(a);
```

The result is the same as either of the following:

```
__m128d t = _mm_set_pd(2.0, 1.0);
__m128d t = _mm_setr_pd(1.0, 2.0);
```

In other words, the xmm register that holds the value t will look as follows:



The "scalar" element is 1.0. Due to the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals).

Intrinsic Syntax

To use an intrinsic in your code, insert a line with the following syntax:

```
data_type intrinsic_name (parameters)
```

Where,

data_type	Is the return data type, which can be either void, int, __m64, __m128, __m128d, __m128i, __int64. Intrinsics that can be implemented across all IA may return other data types as well, as indicated in the intrinsic syntax definitions.
intrinsic_name	Is the name of the intrinsic, which behaves like a function that you can use in your C++ code instead of inlining the actual instruction.
parameters	Represents the parameters required by each intrinsic.

Intrinsics For All IA

The intrinsics in this section function across all IA-32 and Itanium®-based platforms. They are offered as a convenience to the programmer. They are grouped as follows:

- Integer Arithmetic Related
- Floating-Point Related
- String and Block Copy Related
- Miscellaneous

Integer Arithmetic Related

Intrinsic	Description
`int abs(int)`	Returns the absolute value of an integer.
`long labs(long)`	Returns the absolute value of a long integer.
`unsigned long _lrotl(unsigned long value, int shift)`	Rotates bits left for an unsigned long integer.
`unsigned long _lrotr(unsigned long value, int shift)`	Rotates bits right for an unsigned long integer.
`unsigned int __rotl(unsigned int value, int shift)`	Rotates bits left for an unsigned integer.
`unsigned int __rotr(unsigned int value, int shift)`	Rotates bits right for an unsigned integer.

Note

Passing a constant shift value in the rotate intrinsics results in higher performance.

Floating-point Related

Intrinsic	Description
`double fabs(double)`	Returns the absolute value of a floating-point value.
`double log(double)`	Returns the natural logarithm ln(x), x>0, with double precision.
`float logf(float)`	Returns the natural logarithm ln(x), x>0, with single precision.
`double log10(double)`	Returns the base 10 logarithm log10(x), x>0, with double precision.
`float log10f(float)`	Returns the base 10 logarithm log10(x), x>0, with single precision.
`double exp(double)`	Returns the exponential function with double precision.

Intrinsic	Description
`float expf(float)`	Returns the exponential function with single precision.
`double pow(double, double)`	Returns the value of x to the power y with double precision.
`float powf(float, float)`	Returns the value of x to the power y with single precision.
`double sin(double)`	Returns the sine of x with double precision.
`float sinf(float)`	Returns the sine of x with single precision.
`double cos(double)`	Returns the cosine of x with double precision.
`float cosf(float)`	Returns the cosine of x with single precision.
`double tan(double)`	Returns the tangent of x with double precision.
`float tanf(float)`	Returns the tangent of x with single precision.
`double acos(double)`	Returns the arccosine of x with double precision
`float acosf(float)`	Returns the arccosine of x with single precision
`double acosh(double)`	Compute the inverse hyperbolic cosine of the argument with double precision.
`float acoshf(float)`	Compute the inverse hyperbolic cosine of the argument with single precision.
`double asin(double)`	Compute arc sine of the argument with double precision.
`float asinf(float)`	Compute arc sine of the argument with single precision.

Intrinsic	Description
`double asinh(double)`	Compute inverse hyperbolic sine of the argument with double precision.
`float asinhf(float)`	Compute inverse hyperbolic sine of the argument with single precision.
`double atan(double)`	Compute arc tangent of the argument with double precision.
`float atanf(float)`	Compute arc tangent of the argument with single precision.
`double atanh(double)`	Compute inverse hyperbolic tangent of the argument with double precision.
`float atanhf(float)`	Compute inverse hyperbolic tangent of the argument with single precision.
`float cabs(double)**`	Computes absolute value of complex number.
`double ceil(double)`	Computes smallest integral value of double precision argument not less than the argument.
`float ceilf(float)`	Computes smallest integral value of single precision argument not less than the argument.
`double cosh(double)`	Computes the hyperbolic cosine of double precison argument.
`float coshf(float)`	Computes the hyperbolic cosine of single precison argument.
`float fabsf(float)`	Computes absolute value of single precision argument.
`double floor(double)`	Computes the largest integral value of the double precision argument not greater than the argument.
`float floorf(float)`	Computes the largest integral value of the single precision argument not greater than the argument.

Intrinsic	Description
`double fmod(double)`	Computes the floating-point remainder of the division of the first argument by the second argument with double precison.
`float fmodf(float)`	Computes the floating-point remainder of the division of the first argument by the second argument with single precison.
`double hypot(double, double)`	Computes the length of the hypotenuse of a right angled triangle with double precision.
`float hypotf(float)`	Computes the length of the hypotenuse of a right angled triangle with single precision.
`double rint(double)`	Computes the integral value represented as double using the IEEE rounding mode.
`float rintf(float)`	Computes the integral value represented with single precision using the IEEE rounding mode.
`double sinh(double)`	Computes the hyperbolic sine of the double precision argument.
`float sinhf(float)`	Computes the hyperbolic sine of the single precision argument.
`float sqrtf(float)`	Computes the square root of the single precision argument.
`double tanh(double)`	Computes the hyperbolic tangent of the double precision argument.
`float tanhf(float)`	Computes the hyperbolic tangent of the single precision argument.

* Not implemented on Itanium®-based systems.

** `double` in this case is a complex number made up of two single precision (32-bit floating point) elements (real and imaginary parts).

String and Block Copy Related

The following are not implemented as intrinsics on Itanium®-based platforms.

Intrinsic	Description
`char *_strset(char *, _int32)`	Sets all characters in a string to a fixed value.
`void *memcmp(const void *cs, const void *ct, size_t n)`	Compares two regions of memory. Return <0 if `cs<ct`, 0 if `cs=ct`, or >0 if `cs>ct`.
`void *memcpy(void *s, const void *ct, size_t n)`	Copies from memory. Returns `s`.
`void *memset(void * s, int c, size_t n)`	Sets memory to a fixed value. Returns `s`.
`char *strcat(char * s, const char * ct)`	Appends to a string. Returns `s`.
`int *strcmp(const char *, const char *)`	Compares two strings. Return <0 if `cs<ct`, 0 if `cs=ct`, or >0 if `cs>ct`.
`char *strcpy(char * s, const char * ct)`	Copies a string. Returns `s`.

Intrinsic	Description
`size_t strlen(const char * cs)`	Returns the length of string `cs`.
`int strncmp(char *, char *, int)`	Compare two strings, but only specified number of characters.
`int strncpy(char *, char *, int)`	Copies a string, but only specified number of characters.

Miscellaneous Intrinsics

The intrinsic functions listed here are supported on all architectures, except where noted.

Intrinsic	Description
`_abnormal_termination(void)`	Can be invoked only by termination handlers. Returns `TRUE` if the termination handler is invoked as a result of a premature exit of the corresponding try-finally region.
`void *_alloca(int)`	Allocates the buffers.
`extern int _bit_scan_forward(int x)`	Returns the bit index of the least significant set bit of x. If x is 0, the result is undefined.
`extern int _bit_scan_reverse(int)`	Returns the bit index of the most significant set bit of x. If x is 0, the result is undefined.

Intrinsic	Description
`extern int _bswap(int)`	Reverses the byte order of x. Bits 0-7 are swapped with bits 24-31, and bits 8-15 are swapped with bits 16-23.
`_exception_code(void)`	Returns the exception code.
`_exception_info(void)`	Returns the exception information.
`void _enable()`	Enables the interrupt.
`void _disable()`	Disables the interrupt.
`int _in_byte(int)`	Intrinsic that maps to the IA-32 instruction `IN`. Transfer data byte from port specified by argument.
`int _in_dword(int)`	Intrinsic that maps to the IA-32 instruction `IN`. Transfer double word from port specified by argument.
`int _in_word(int)`	Intrinsic that maps to the IA-32 instruction `IN`. Transfer word from port specified by argument.
`int _inp(int)`	Same as `_in_byte`
`int _inpd(int)`	Same as `_in_dword`
`int _inpw(int)`	Same as `_in_word`
`int _out_byte(int, int)`	Intrinsic that maps to the IA-32 instruction `OUT`. Transfer data byte in second argument to port specified by first argument.

Intrinsic	Description
`int _out_dword(int, int)`	Intrinsic that maps to the IA-32 instruction `OUT`. Transfer double word in second argument to port specified by first argument.
`int _out_word(int, int)`	Intrinsic that maps to the IA-32 instruction `OUT`. Transfer word in second argument to port specified by first argument.
`int _outp(int, int)`	Same as `_out_byte`
`int _outpd(int, int)`	Same as `_out_dword`
`int _outpw(int, int)`	Same as `_out_word`
`extern int _popcnt32(int x)`	Returns the number of set bits in x.
`extern __int64 _rdtsc(void)`	Returns the current value of the processor's 64-bit time stamp counter. (not supported on Itanium® architercture)
`extern __int64 _rdpmc(int p)`	Returns the current value of the 40-bit performance monitoring counter specified by p.
`int _setjmp(jmp_buf)*`	A fast version of `setjmp()`, which bypasses the termination handling. Saves the callee-save registers, stack pointer and return address.

Support for MMX(TM) Technology

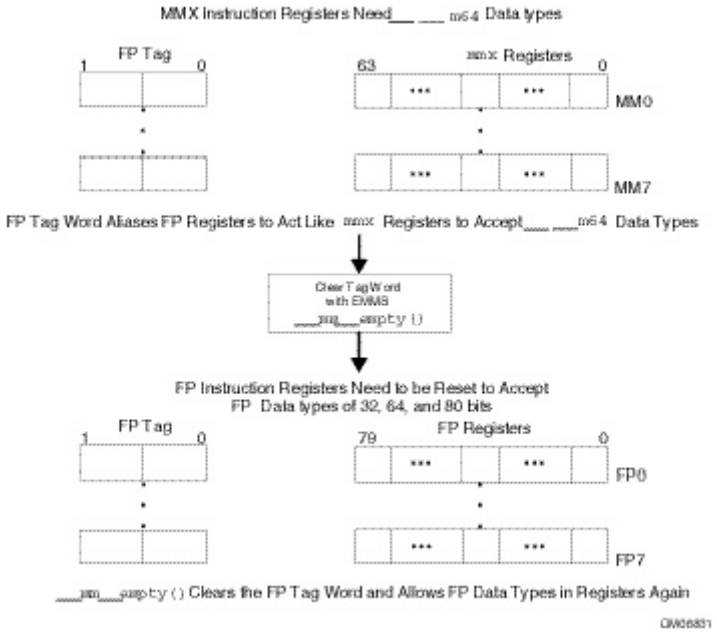
MMX(TM) technology is an extension to the Intel architecture (IA) instruction set. The MMX instruction set adds 57 opcodes and a 64-bit quadword data type, and eight 64-bit registers. Each of the eight registers can be directly addressed using the register names `mm0` to `mm7`.

The prototypes for MMX technology intrinsics are in the `mmintrin.h` header file.

The EMMS Instruction: Why You Need It

Using `EMMS` is like emptying a container to accommodate new content. For instance, MMX(TM) instructions automatically enable an FP tag word in the register to enable use of the `__m64` data type. This resets the FP register set to alias it as the MMX register set. To enable the FP register set again, reset the register state with the `EMMS` instruction or via the `_mm_empty()` intrinsic.

Why You Need EMMS to Reset After an MMX(TM) Instruction



Caution

Failure to empty the multimedia state after using an MMX instruction and before using a floating-point instruction can result in unexpected execution or poor performance.

EMMS Usage Guidelines

The guidelines when to use EMMS are:

- Do not use on Itanium®-based systems. There are no special registers (or overlay) for the MMX(TM) instructions or Streaming SIMD Extensions on Itanium-based systems even though the intrinsics are supported.
- Use `_mm_empty()` after an MMX instruction if the next instruction is a floating-point (FP) instruction -- for example, before calculations on `float`, `double` or `long double`. You must be aware of all situations when your code generates an MMX instruction with the Intel® C++ Compiler, i.e.:
  - when using an MMX technology intrinsic
  - when using Streaming SIMD Extension integer intrinsics that use the `__m64` data type
  - when referencing an `__m64` data type variable
  - when using an MMX instruction through inline assembly
- Do not use `_mm_empty()` before an MMX instruction, since using `_mm_empty()` before an MMX instruction incurs an operation with no benefit (no-op).
- Use different functions for operations that use FP instructions and those that use MMX instructions. This eliminates the need to empty the multimedia state within the body of a critical loop.
- Use `_mm_empty()` during runtime initialization of `__m64` and FP data types. This ensures resetting the register between data type transitions.
- See the "Correct Usage" coding example.

Incorrect Usage	Correct Usage
`__m64 x = _m_paddd(y, z);` `float f = init();`	`__m64 x = _m_paddd(y, z);` `float f = (_mm_empty(), init());`

For more documentation on EMMS, visit the http://developer.intel.com Web site.

MMX(TM) Technology General Support Intrinsics

The prototypes for MMX(TM) technology intrinsics are in the `mmintrin.h` header file.

Intrinsic Name	Alternate Name	Corresponding Instruction	Operation	Signed	Saturation
_m_empty	_mm_empty	EMMS	Empty MM state	--	--
_m_from_int	_mm_cvtsi32_si64	MOVD	Convert from int	--	--
_m_to_int	_mm_cvtsi64_si32	MOVD	Convert from int	--	--
_m_from_int64	_mm_cvtsi64_m64	MOVQ	Convert from int	--	--
_m_to_int64	_mm_cvtm64_si64	MOVQ	Convert from int	--	--
_m_packsswb	_mm_packs_pi16	PACKSSWB	Pack	Yes	Yes
_m_packssdw	_mm_packs_pi32	PACKSSDW	Pack	Yes	Yes
_m_packuswb	_mm_packs_pu16	PACKUSWB	Pack	No	Yes
_m_punpckhbw	_mm_unpackhi_pi8	PUNPCKHBW	Interleave	--	--
_m_punpckhwd	_mm_unpackhi_pi16	PUNPCKHWD	Interleave	--	--
_m_punpckhdq	_mm_unpackhi_pi32	PUNPCKHDQ	Interleave	--	--
_m_punpcklbw	_mm_unpacklo_pi8	PUNPCKLBW	Interleave	--	--
_m_punpcklwd	_mm_unpacklo_pi16	PUNPCKLWD	Interleave	--	--
_m_punpckldq	_mm_unpacklo_pi32	PUNPCKLDQ	Interleave	--	--

`void _m_empty(void)`

Empty the multimedia state.

`__m64 _m_from_int(int i)`

Convert the integer object `i` to a 64-bit `__m64` object. The integer value is zero-extended to 64 bits.

`int _m_to_int(__m64 m)`

Convert the lower 32 bits of the `__m64` object `m` to an integer.

`__m64 _mm_cvtsi64_m64(__int64 i)`

Move the 64-bit integer object `i` to a `__mm64` object

`__int64 _mm_cvtm64_si64(__m64 m)`

Move the `__m64` object `m` to a 64-bit integer

`__m64 _m_packsswb(__m64 m1, __m64 m2)`

Pack the four 16-bit values from `m1` into the lower four 8-bit values of the result with signed saturation, and pack the four 16-bit values from `m2` into the upper four 8-bit values of the result with signed saturation.

`__m64 _m_packssdw(__m64 m1, __m64 m2)`

Pack the two 32-bit values from `m1` into the lower two 16-bit values of the result with signed saturation, and pack the two 32-bit values from `m2` into the upper two 16-bit values of the result with signed saturation.

`__m64 _m_packuswb(__m64 m1, __m64 m2)`

Pack the four 16-bit values from `m1` into the lower four 8-bit values of the result with unsigned saturation, and pack the four 16-bit values from `m2` into the upper four 8-bit values of the result with unsigned saturation.

`__m64 _m_punpckhbw(__m64 m1, __m64 m2)`

Interleave the four 8-bit values from the high half of `m1` with the four values from the high half of `m2`. The interleaving begins with the data from `m1`.

`__m64 _m_punpckhwd(__m64 m1, __m64 m2)`

Interleave the two 16-bit values from the high half of `m1` with the two values from the high half of `m2`. The interleaving begins with the data from `m1`.

`__m64 _m_punpckhdq(__m64 m1, __m64 m2)`

Interleave the 32-bit value from the high half of `m1` with the 32-bit value from the high half of `m2`. The interleaving begins with the data from `m1`.

`__m64 _m_punpcklbw(__m64 m1, __m64 m2)`

Interleave the four 8-bit values from the low half of `m1` with the four values from the low half of `m2`. The interleaving begins with the data from `m1`.

`__m64 _m_punpcklwd(__m64 m1, __m64 m2)`

Interleave the two 16-bit values from the low half of `m1` with the two values from the low half of `m2`. The interleaving begins with the data from `m1`.

`__m64 _m_punpckldq(__m64 m1, __m64 m2)`

Interleave the 32-bit value from the low half of `m1` with the 32-bit value from the low half of `m2`. The interleaving begins with the data from `m1`.

MMX(TM) Technology Packed Arithmetic Intrinsics

The prototypes for MMX(TM) technology intrinsics are in the `mmintrin.h` header file.

Packed Arithmetic Intrinsics, Part 1

Intrinsic Name	Alternate Name	Corresponding Instruction	Operation	Signed
_m_paddb	_mm_add_pi8	PADDB	Addition	--
_m_paddw	_mm_add_pi16	PADDW	Addition	--
_m_paddd	_mm_add_pi32	PADDD	Addition	--
_m_paddsb	_mm_adds_pi8	PADDSB	Addition	Yes
_m_paddsw	_mm_adds_pi16	PADDSW	Addition	Yes
_m_paddusb	_mm_adds_pu8	PADDUSB	Addition	No
_m_paddusw	_mm_adds_pu16	PADDUSW	Addition	No
_m_psubb	_mm_sub_pi8	PSUBB	Subtraction	--
_m_psubw	_mm_sub_pi16	PSUBW	Subtraction	--
_m_psubd	_mm_sub_pi32	PSUBD	Subtraction	--
_m_psubsb	_mm_subs_pi8	PSUBSB	Subtraction	Yes
_m_psubsw	_mm_subs_pi16	PSUBSW	Subtraction	Yes
_m_psubusb	_mm_subs_pu8	PSUBUSB	Subtraction	No
_m_psubusw	_mm_subs_pu16	PSUBUSW	Subtraction	No
_m_pmaddwd	_mm_madd_pi16	PMADDWD	Multiplication	--
_m_pmulhw	_mm_mulhi_pi16	PMULHW	Multiplication	Yes
_m_pmullw	_mm_mullo_pi16	PMULLW	Multiplication	--

Packed Arithmetic Intrinsics, Part 2

Intrinsic Name	Alternate Name	Corresponding Instruction	Argument Values/Bits	Result Values/Bits
_m_paddb	_mm_add_pi8	PADDB	8/8	8/8
_m_paddw	_mm_add_pi16	PADDW	4/16	4/16
_m_paddd	_mm_add_pi32	PADDD	2/32	2/32
_m_paddsb	_mm_adds_pi8	PADDSB	8/8	8/8
_m_paddsw	_mm_adds_pi16	PADDSW	4/16	4/16
_m_paddusb	_mm_adds_pu8	PADDUSB	8/8	8/8
_m_paddusw	_mm_adds_pu16	PADDUSW	4/16	4/16
_m_psubb	_mm_sub_pi8	PSUBB	8/8	8/8
_m_psubw	_mm_sub_pi16	PSUBW	4/16	4/16
_m_psubd	_mm_sub_pi32	PSUBD	2/32	2/32
_m_psubsb	_mm_subs_pi8	PSUBSB	8/8	8/8
_m_psubsw	_mm_subs_pi16	PSUBSW	4/16	4/16
_m_psubusb	_mm_subs_pu8	PSUBUSB	8/8	8/8
_m_psubusw	_mm_subs_pu16	PSUBUSW	4/16	4/16
_m_pmaddwd	_mm_madd_pi16	PMADDWD	4/16	2/32
_m_pmulhw	_mm_mulhi_pi16	PMULHW	4/16	4/16 (high)
_m_pmullw	_mm_mullo_pi16	PMULLW	4/16	4/16 (low)

`__m64 _m_paddb(__m64 m1, __m64 m2)`

Add the eight 8-bit values in `m1` to the eight 8-bit values in `m2`.

`__m64 _m_paddw(__m64 m1, __m64 m2)`

Add the four 16-bit values in `m1` to the four 16-bit values in `m2`.

`__m64 _m_paddd(__m64 m1, __m64 m2)`

Add the two 32-bit values in `m1` to the two 32-bit values in `m2`.

```
__m64 _m_paddsb(__m64 m1, __m64 m2)
```

Add the eight signed 8-bit values in `m1` to the eight signed 8-bit values in `m2` using saturating arithmetic.

```
__m64 _m_paddsw(__m64 m1, __m64 m2)
```

Add the four signed 16-bit values in `m1` to the four signed 16-bit values in `m2` using saturating arithmetic.

```
__m64 _m_paddusb(__m64 m1, __m64 m2)
```

Add the eight unsigned 8-bit values in `m1` to the eight unsigned 8-bit values in `m2` and using saturating arithmetic.

```
__m64 _m_paddusw(__m64 m1, __m64 m2)
```

Add the four unsigned 16-bit values in `m1` to the four unsigned 16-bit values in `m2` using saturating arithmetic.

```
__m64 _m_psubb(__m64 m1, __m64 m2)
```

Subtract the eight 8-bit values in `m2` from the eight 8-bit values in `m1`.

```
__m64 _m_psubw(__m64 m1, __m64 m2)
```

Subtract the four 16-bit values in `m2` from the four 16-bit values in `m1`.

```
__m64 _m_psubd(__m64 m1, __m64 m2)
```

Subtract the two 32-bit values in `m2` from the two 32-bit values in `m1`.

```
__m64 _m_psubsb(__m64 m1, __m64 m2)
```

Subtract the eight signed 8-bit values in `m2` from the eight signed 8-bit values in `m1` using saturating arithmetic.

```
__m64 _m_psubsw(__m64 m1, __m64 m2)
```

Subtract the four signed 16-bit values in `m2` from the four signed 16-bit values in `m1` using saturating arithmetic.

```
__m64 _m_psubusb(__m64 m1, __m64 m2)
```

Subtract the eight unsigned 8-bit values in `m2` from the eight unsigned 8-bit values in `m1` using saturating arithmetic.

```
__m64 _m_psubusw(__m64 m1, __m64 m2)
```

Subtract the four unsigned 16-bit values in `m2` from the four unsigned 16-bit values in `m1` using saturating arithmetic.

`__m64 _m_pmaddwd(__m64 m1, __m64 m2)`

Multiply four 16-bit values in `m1` by four 16-bit values in `m2` producing four 32-bit intermediate results, which are then summed by pairs to produce two 32-bit results.

`__m64 _m_pmulhw(__m64 m1, __m64 m2)`

Multiply four signed 16-bit values in `m1` by four signed 16-bit values in `m2` and produce the high 16 bits of the four results.

`__m64 _m_pmullw(__m64 m1, __m64 m2)`

Multiply four 16-bit values in `m1` by four 16-bit values in `m2` and produce the low 16 bits of the four results.

MMX(TM) Technology Shift Intrinsics

The prototypes for MMX(TM) technology intrinsics are in the `mmintrin.h` header file.

Intrinsic Name	Alternate Name	Shift Direction	Shift Type	Corresponding Instruction
`_m_psllw`	`_mm_sll_pi16`	left	Logical	`PSLLW`
`_m_psllwi`	`_mm_slli_pi16`	left	Logical	`PSLLWI`
`_m_pslld`	`_mm_sll_pi32`	left	Logical	`PSLLD`
`_m_pslldi`	`_mm_slli_pi32`	left	Logical	`PSLLDI`
`_m_psllq`	`_mm_sll_si64`	left	Logical	`PSLLQ`
`_m_psllqi`	`_mm_slli_si64`	left	Logical	`PSLLQI`
`_m_psraw`	`_mm_sra_pi16`	right	Arithmetic	`PSRAW`
`_m_psrawi`	`_mm_srai_pi16`	right	Arithmetic	`PSRAWI`
`_m_psrad`	`_mm_sra_pi32`	right	Arithmetic	`PSRAD`
`_m_psradi`	`_mm_srai_pi32`	right	Arithmetic	`PSRADI`
`_m_psrlw`	`_mm_srl_pi16`	right	Logical	`PSRLW`
`_m_psrlwi`	`_mm_srli_pi16`	right	Logical	`PSRLWI`
`_m_psrld`	`_mm_srl_pi32`	right	Logical	`PSRLD`
`_m_psrldi`	`_mm_srli_pi32`	right	Logical	`PSRLDI`

Intrinsic Name	Alternate Name	Shift Direction	Shift Type	Corresponding Instruction
_m_psrlq	_mm_srl_si64	right	Logical	PSRLQ
_m_psrlqi	_mm_srli_si64	right	Logical	PSRLQI

```
__m64 _m_psllw(__m64 m, __m64 count)
```

Shift four 16-bit values in `m` left the amount specified by `count` while shifting in zeros.

```
__m64 _m_psllwi(__m64 m, int count)
```

Shift four 16-bit values in `m` left the amount specified by `count` while shifting in zeros. For the best performance, `count` should be a constant.

```
__m64 _m_pslld(__m64 m, __m64 count)
```

Shift two 32-bit values in `m` left the amount specified by `count` while shifting in zeros.

```
__m64 _m_pslldi(__m64 m, int count)
```

Shift two 32-bit values in `m` left the amount specified by `count` while shifting in zeros. For the best performance, `count` should be a constant.

```
__m64 _m_psllq(__m64 m, __m64 count)
```

Shift the 64-bit value in `m` left the amount specified by `count` while shifting in zeros.

```
__m64 _m_psllqi(__m64 m, int count)
```

Shift the 64-bit value in `m` left the amount specified by `count` while shifting in zeros. For the best performance, `count` should be a constant.

```
__m64 _m_psraw(__m64 m, __m64 count)
```

Shift four 16-bit values in `m` right the amount specified by `count` while shifting in the sign bit.

```
__m64 _m_psrawi(__m64 m, int count)
```

Shift four 16-bit values in `m` right the amount specified by `count` while shifting in the sign bit. For the best performance, `count` should be a constant.

```
__m64 _m_psrad(__m64 m, __m64 count)
```

Shift two 32-bit values in `m` right the amount specified by `count` while shifting in the sign bit.

```
__m64 _m_psradi(__m64 m, int count)
```

Shift two 32-bit values in `m` right the amount specified by `count` while shifting in the sign bit. For the best performance, `count` should be a constant.

```
__m64 _m_psrlw(__m64 m, __m64 count)
```

Shift four 16-bit values in `m` right the amount specified by `count` while shifting in zeros.

```
__m64 _m_psrlwi(__m64 m, int count)
```

Shift four 16-bit values in `m` right the amount specified by `count` while shifting in zeros. For the best performance, `count` should be a constant.

```
__m64 _m_psrld(__m64 m, __m64 count)
```

Shift two 32-bit values in `m` right the amount specified by `count` while shifting in zeros.

```
__m64 _m_psrldi(__m64 m, int count)
```

Shift two 32-bit values in `m` right the amount specified by `count` while shifting in zeros. For the best performance, `count` should be a constant.

```
__m64 _m_psrlq(__m64 m, __m64 count)
```

Shift the 64-bit value in `m` right the amount specified by `count` while shifting in zeros.

```
__m64 _m_psrlqi(__m64 m, int count)
```

Shift the 64-bit value in `m` right the amount specified by `count` while shifting in zeros. For the best performance, `count` should be a constant.

MMX(TM) Technology Logical Intrinsics

The prototypes for MMX(TM) technology intrinsics are in the `mmintrin.h` header file.

Intrinsic Name	Alternate Name	Operation	Corresponding Instruction
`_m_pand`	`_mm_and_si64`	Bitwise AND	`PAND`
`_m_pandn`	`_mm_andnot_si64`	Logical NOT	`PANDN`
`_m_por`	`_mm_or_si64`	Bitwise OR	`POR`
`_m_pxor`	`_mm_xor_si64`	Bitwise Exclusive OR	`PXOR`

`__m64 _m_pand(__m64 m1, __m64 m2)`

Perform a bitwise AND of the 64-bit value in `m1` with the 64-bit value in `m2`.

`__m64 _m_pandn(__m64 m1, __m64 m2)`

Perform a logical NOT on the 64-bit value in `m1` and use the result in a bitwise AND with the 64-bit value in `m2`.

`__m64 _m_por(__m64 m1, __m64 m2)`

Perform a bitwise OR of the 64-bit value in `m1` with the 64-bit value in `m2`.

`__m64 _m_pxor(__m64 m1, __m64 m2)`

Perform a bitwise XOR of the 64-bit value in `m1` with the 64-bit value in `m2`.

MMX(TM) Technology Compare Intrinsics

The prototypes for MMX(TM) technology intrinsics are in the `mmintrin.h` header file.

Intrinsic Name	Alternate Name	Comparison	Number of Elements	Element Bit Size	Corresponding Instruction
`_m_pcmpeqb`	`_mm_cmpeq_pi8`	Equal	8	8	`PCMPEQB`
`_m_pcmpeqw`	`_mm_cmpeq_pi16`	Equal	4	16	`PCMPEQW`
`_m_pcmpeqd`	`_mm_cmpeq_pi32`	Equal	2	32	`PCMPEQD`
`_m_pcmpgtb`	`_mm_cmpgt_pi8`	Greater Than	8	8	`PCMPGTB`
`_m_pcmpgtw`	`_mm_cmpgt_pi16`	Greater Than	4	16	`PCMPGTW`
`_m_pcmpgtd`	`_mm_cmpgt_pi32`	Greater Than	2	32	`PCMPGTD`

`__m64 _m_pcmpeqb(__m64 m1, __m64 m2)`

If the respective 8-bit values in `m1` are equal to the respective 8-bit values in `m2` set the respective 8-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _m_pcmpeqw(__m64 m1, __m64 m2)`

If the respective 16-bit values in `m1` are equal to the respective 16-bit values in `m2` set the respective 16-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _m_pcmpeqd(__m64 m1, __m64 m2)`

If the respective 32-bit values in `m1` are equal to the respective 32-bit values in `m2` set the respective 32-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _m_pcmpgtb(__m64 m1, __m64 m2)`

If the respective 8-bit values in `m1` are greater than the respective 8-bit values in `m2` set the respective 8-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _m_pcmpgtw(__m64 m1, __m64 m2)`

If the respective 16-bit values in `m1` are greater than the respective 16-bit values in `m2` set the respective 16-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _m_pcmpgtd(__m64 m1, __m64 m2)`

If the respective 32-bit values in `m1` are greater than the respective 32-bit values in `m2` set the respective 32-bit resulting values to all ones, otherwise set them all to zeros.

MMX(TM) Technology Set Intrinsics

The prototypes for MMX(TM) technology intrinsics are in the `mmintrin.h` header file.

Intrinsic Name	Operation	Number of Elements	Element Bit Size	Signed	Reverse Order
`_mm_setzero_si64`	set to zero	1	64	No	No
`_mm_set_pi32`	set integer values	2	32	No	No
`_mm_set_pi16`	set integer values	4	16	No	No
`_mm_set_pi8`	set integer values	8	8	No	No
`_mm_set1_pi32`	set integer values	2	32	Yes	No
`_mm_set1_pi16`	set integer values	4	16	Yes	No

Intrinsic Name	Operation	Number of Elements	Element Bit Size	Signed	Reverse Order
_mm_set1_pi8	set integer values	8	8	Yes	No
_mm_setr_pi32	set integer values	2	32	No	Yes
_mm_setr_pi16	set integer values	4	16	No	Yes
_mm_setr_pi8	set integer values	8	8	No	Yes

Note

In the following descriptions regarding the bits of the MMX register, bit 0 is the least significant and bit 63 is the most significant.

```
__m64 _mm_setzero_si64()
```

PXOR
Sets the 64-bit value to zero.

```
r := 0x0
```

```
__m64 _mm_set_pi32(int i1, int i0)
```

(composite) Sets the 2 signed 32-bit integer values.

```
r0 := i0
r1 := i1
```

```
__m64 _mm_set_pi16(short s3, short s2, short s1, short s0)
```

(composite) Sets the 4 signed 16-bit integer values.

```
r0 := w0
r1 := w1
r2 := w2
r3 := w3
```

```
__m64 _mm_set_pi8(char b7, char b6, char b5, char b4, char
b3, char b2, char b1, char b0)
```

(composite) Sets the 8 signed 8-bit integer values.

```
r0 := b0
r1 := b1
...
r7 := b7
```

```
__m64 _mm_set1_pi32(int i)
```

Sets the 2 signed 32-bit integer values to `i`.

```
r0 := i
r1 := i
```

```
__m64 _mm_set1_pi16(short s)
```

(composite) Sets the 4 signed 16-bit integer values to `w`.

```
r0 := w
r1 := w
r2 := w
r3 := w
```

```
__m64 _mm_set1_pi8(char b)
```

(composite) Sets the 8 signed 8-bit integer values to b

r0 := b
r1 := b
...
r7 := b

```
__m64 _mm_setr_pi32(int i1, int i0)
```

(composite) Sets the 2 signed 32-bit integer values in reverse order.

```
r0 := i0
r1 := i1
```

```
__m64 _mm_setr_pi16(short s3, short s2, short s1, short s0)
```

(composite) Sets the 4 signed 16-bit integer values in reverse order.

```
r0 := w0
r1 := w1
r2 := w2
r3 := w3
```

```
__m64 _mm_setr_pi8(char b7, char b6, char b5, char b4, char
b3, char b2, char b1, char b0)
```

(composite) Sets the 8 signed 8-bit integer values in reverse order.

```
r0 := b0
r1 := b1
...
r7 := b7
```

MMX(TM) Technology Intrinsics on Itanium® Architecture

MMX(TM) technology intrinsics provide access to the MMX technology instruction set on Itanium®-based systems. To provide source compatibility with the IA-32 architecture, these intrinsics are equivalent both in name and functionality to the set of IA-32-based MMX intrinsics.

Some intrinsics have more than one name. When one intrinsic has two names, both names generate the same instructions, but the first is preferred as it conforms to a newer naming standard.

The prototypes for MMX technology intrinsics are in the `mmintrin.h` header file.

Data Types

The C data type `__m64` is used when using MMX technology intrinsics. It can hold eight 8-bit values, four 16-bit values, two 32-bit values, or one 64-bit value.

The `__m64` data type is not a basic ANSI C data type. Therefore, observe the following usage restrictions:

- Use the new data type only on the left-hand side of an assignment, as a return value, or as a parameter. You cannot use it with other arithmetic expressions (" + ", " - ", and so on).
- Use the new data type as objects in aggregates, such as unions, to access the byte elements and structures; the address of an `__m64` object may be taken.
- Use new data types only with the respective intrinsics described in this documentation.

For complete details of the hardware instructions, see the Intel® Architecture MMX Technology Programmer's Reference Manual. For descriptions of data types, see the Intel® Architecture Software Developer's Manual, Volume 2.

Streaming SIMD Extensions

This section describes the C++ language-level features supporting the Streaming SIMD Extensions (SSE) in the Intel® C++ Compiler. These topics explain the following features of the intrinsics:

- Floating Point Intrinsics
- Arithmetic Operation Intrinsics
- Logical Operation Intrinsics
- Comparison Intrinsics
- Conversion Intrinsics
- Load Operations
- Set Operations
- Store Operations
- Cacheability Support
- Integer Intrinsics
- Memory and Initialization Intrinsics
- Miscellaneous Intrinsics
- Using Streaming SIMD Extensions on Itanium® Architecture

The prototypes for SSE intrinsics are in the `xmmintrin.h` header file.

Note

You can also use the single `ia32intrin.h` header file for any IA-32 intrinsics.

Floating-point Intrinsics for Streaming SIMD Extensions

You should be familiar with the hardware features provided by the Streaming SIMD Extensions (SSE) when writing programs with the intrinsics. The following are four important issues to keep in mind:

- Certain intrinsics, such as `_mm_loadr_ps` and `_mm_cmpgt_ss`, are not directly supported by the instruction set. While these intrinsics are convenient programming aids, be mindful that they may consist of more than one machine-language instruction.
- Floating-point data loaded or stored as `__m128` objects must be generally 16-byte-aligned.
- Some intrinsics require that their argument be immediates, that is, constant integers (literals), due to the nature of the instruction.
- The result of arithmetic operations acting on two `NaN` (Not a Number) arguments is undefined. Therefore, FP operations using `NaN` arguments will not match the expected behavior of the corresponding assembly instructions.

Arithmetic Operations for Streaming SIMD Extensions

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

Intrinsic	Instruction	Operation	R0	R1	R2	R3
`_mm_add_ss`	`ADDSS`	Addition	`a0 [op] b0`	`a1`	`a2`	`a3`
`_mm_add_ps`	`ADDPS`	Addition	`a0 [op] b0`	`a1 [op] b1`	`a2 [op] b2`	`a3 [op] b3`
`_mm_sub_ss`	`SUBSS`	Subtraction	`a0 [op] b0`	`a1`	`a2`	`a3`
`_mm_sub_ps`	`SUBPS`	Subtraction	`a0 [op] b0`	`a1 [op] b1`	`a2 [op] b2`	`a3 [op] b3`
`_mm_mul_ss`	`MULSS`	Multiplication	`a0 [op] b0`	`a1`	`a2`	`a3`
`_mm_mul_ps`	`MULPS`	Multiplication	`a0 [op] b0`	`a1 [op] b1`	`a2 [op] b2`	`a3 [op] b3`
`_mm_div_ss`	`DIVSS`	Division	`a0 [op] b0`	`a1`	`a2`	`a3`
`_mm_div_ps`	`DIVPS`	Division	`a0 [op] b0`	`a1 [op] b1`	`a2 [op] b2`	`a3 [op] b3`
`_mm_sqrt_ss`	`SQRTSS`	Squared Root	`[op] a0`	`a1`	`a2`	`a3`
`_mm_sqrt_ps`	`SQRTPS`	Squared Root	`[op] a0`	`[op] b1`	`[op] b2`	`[op] b3`
`_mm_rcp_ss`	`RCPSS`	Reciprocal	`[op] a0`	`a1`	`a2`	`a3`

Intrinsic	Instruction	Operation	R0	R1	R2	R3
_mm_rcp_ps	RCPPS	Reciprocal	[op] a0	[op] b1	[op] b2	[op] b3
_mm_rsqrt_ss	RSQRTSS	Reciprocal Square Root	[op] a0	a1	a2	a3
_mm_rsqrt_ps	RSQRTPS	Reciprocal Squared Root	[op] a0	[op] b1	[op] b2	[op] b3
_mm_min_ss	MINSS	Computes Minimum	[op](a0,b0)	a1	a2	a3
_mm_min_ps	MINPS	Computes Minimum	[op](a0,b0)	[op] (a1, b1)	[op] (a2, b2)	[op] (a3, b3)
_mm_max_ss	MAXSS	Computes Maximum	[op](a0,b0)	a1	a2	a3
_mm_max_ps	MAXPS	Computes Maximum	[op](a0,b0)	[op] (a1, b1)	[op] (a2, b2)	[op] (a3, b3)

```
__m128 _mm_add_ss(__m128 a, __m128 b)
```

Adds the lower SP FP (single-precision, floating-point) values of a and b; the upper 3 SP FP values are passed through from a.

```
r0 := a0 + b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_add_ps(__m128 a, __m128 b)
```

Adds the four SP FP values of a and b.

```
r0 := a0 + b0
r1 := a1 + b1
r2 := a2 + b2
r3 := a3 + b3
```

```
__m128 _mm_sub_ss(__m128 a, __m128 b)
```

Subtracts the lower SP FP values of a and b. The upper 3 SP FP values are passed through from a.

```
r0 := a0 - b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_sub_ps(__m128 a, __m128 b)
```

Subtracts the four SP FP values of a and b.

```
r0 := a0 - b0
r1 := a1 - b1
r2 := a2 - b2
r3 := a3 - b3
```

```
__m128 _mm_mul_ss(__m128 a, __m128 b)
```

Multiplies the lower SP FP values of a and b; the upper 3 SP FP values are passed through from a.

```
r0 := a0 * b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_mul_ps(__m128 a, __m128 b)
```

Multiplies the four SP FP values of a and b.

```
r0 := a0 * b0
r1 := a1 * b1
r2 := a2 * b2
r3 := a3 * b3
```

```
__m128 _mm_div_ss(__m128 a, __m128 b )
```

Divides the lower SP FP values of a and b; the upper 3 SP FP values are passed through from a.

```
r0 := a0 / b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_div_ps(__m128 a, __m128 b)
```

Divides the four SP FP values of a and b.

```
r0 := a0 / b0
r1 := a1 / b1
r2 := a2 / b2
r3 := a3 / b3
```

```
__m128 _mm_sqrt_ss(__m128 a)
```

Computes the square root of the lower SP FP value of a ; the upper 3 SP FP values are passed through.

```
r0 := sqrt(a0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_sqrt_ps(__m128 a)
```

Computes the square roots of the four SP FP values of a.

```
r0 := sqrt(a0)
r1 := sqrt(a1)
r2 := sqrt(a2)
r3 := sqrt(a3)
```

```
__m128 _mm_rcp_ss(__m128 a)
```

Computes the approximation of the reciprocal of the lower SP FP value of a; the upper 3 SP FP values are passed through.

```
r0 := recip(a0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_rcp_ps(__m128 a)
```

Computes the approximations of reciprocals of the four SP FP values of a.

```
r0 := recip(a0)
r1 := recip(a1)
r2 := recip(a2)
r3 := recip(a3)
```

```
__m128 _mm_rsqrt_ss(__m128 a)
```

Computes the approximation of the reciprocal of the square root of the lower SP FP value of a; the upper 3 SP FP values are passed through.

```
r0 := recip(sqrt(a0))
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_rsqrt_ps(__m128 a)
```

Computes the approximations of the reciprocals of the square roots of the four SP FP values of a.

```
r0 := recip(sqrt(a0))
r1 := recip(sqrt(a1))
r2 := recip(sqrt(a2))
r3 := recip(sqrt(a3))
```

```
__m128 _mm_min_ss(__m128 a, __m128 b)
```

Computes the minimum of the lower SP FP values of a and b; the upper 3 SP FP values are passed through from a.

```
r0 := min(a0, b0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_min_ps(__m128 a, __m128 b)
```

Computes the minimum of the four SP FP values of a and b.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

```
__m128 _mm_max_ss(__m128 a, __m128 b)
```

Computes the maximum of the lower SP FP values of a and b; the upper 3 SP FP values are passed through from a.

```
r0 := max(a0, b0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_max_ps(__m128 a, __m128 b)
```

Computes the maximum of the four SP FP values of a and b.

```
r0 := max(a0, b0)
r1 := max(a1, b1)
r2 := max(a2, b2)
r3 := max(a3, b3)
```

Logical Operations for Streaming SIMD Extensions

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

Intrinsic Name	Operation	Corresponding Instruction
`_mm_and_ps`	Bitwise AND	ANDPS
`_mm_andnot_ps`	Logical NOT	ANDNPS
`_mm_or_ps`	Bitwise OR	ORPS
`_mm_xor_ps`	Bitwise Exclusive OR	XORPS

```
__m128 _mm_and_ps(__m128 a, __m128 b)
```

Computes the bitwise And of the four SP FP values of a and b.

```
r0 := a0 & b0
r1 := a1 & b1
r2 := a2 & b2
r3 := a3 & b3
```

```
__m128 _mm_andnot_ps(__m128 a, __m128 b)
```

Computes the bitwise AND-NOT of the four SP FP values of a and b.

```
r0 := ~a0 & b0
r1 := ~a1 & b1
r2 := ~a2 & b2
r3 := ~a3 & b3
```

```
__m128 _mm_or_ps(__m128 a, __m128 b)
```

Computes the bitwise OR of the four SP FP values of a and b.

```
r0 := a0 | b0
r1 := a1 | b1
r2 := a2 | b2
r3 := a3 | b3
```

```
__m128 _mm_xor_ps(__m128 a, __m128 b)
```

Computes bitwise XOR (exclusive-or) of the four SP FP values of a and b.

```
r0 := a0 ^ b0
r1 := a1 ^ b1
r2 := a2 ^ b2
r3 := a3 ^ b3
```

Comparisons for Streaming SIMD Extensions

Each comparison intrinsic performs a comparison of a and b. For the packed form, the four SP FP values of a and b are compared, and a 128-bit mask is returned. For the scalar form, the lower SP FP values of a and b are compared, and a 32-bit mask is returned; the upper three SP FP values are passed through from a. The mask is set to `0xffffffff` for each element where the comparison is true and `0x0` where the comparison is false.

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

Intrinsic Name	Comparison	Corresponding Instruction
`_mm_cmpeq_ss`	Equal	`CMPEQSS`
`_mm_cmpeq_ps`	Equal	`CMPEQPS`
`_mm_cmplt_ss`	Less Than	`CMPLTSS`
`_mm_cmplt_ps`	Less Than	`CMPLTPS`
`_mm_cmple_ss`	Less Than or Equal	`CMPLESS`
`_mm_cmple_ps`	Less Than or Equal	`CMPLEPS`
`_mm_cmpgt_ss`	Greater Than	`CMPLTSS`
`_mm_cmpgt_ps`	Greater Than	`CMPLTPS`
`_mm_cmpge_ss`	Greater Than or Equal	`CMPLESS`
`_mm_cmpge_ps`	Greater Than or Equal	`CMPLEPS`
`_mm_cmpneq_ss`	Not Equal	`CMPNEQSS`
`_mm_cmpneq_ps`	Not Equal	`CMPNEQPS`
`_mm_cmpnlt_ss`	Not Less Than	`CMPNLTSS`

Intrinsic Name	Comparison	Corresponding Instruction
_mm_cmpnlt_ps	Not Less Than	CMPNLTPS
_mm_cmpnle_ss	Not Less Than or Equal	CMPNLESS
_mm_cmpnle_ps	Not Less Than or Equal	CMPNLEPS
_mm_cmpngt_ss	Not Greater Than	CMPNLTSS
_mm_cmpngt_ps	Not Greater Than	CMPNLTPS
_mm_cmpnge_ss	Not Greater Than or Equal	CMPNLESS
_mm_cmpnge_ps	Not Greater Than or Equal	CMPNLEPS
_mm_cmpord_ss	Ordered	CMPORDSS
_mm_cmpord_ps	Ordered	CMPORDPS
_mm_cmpunord_ss	Unordered	CMPUNORDSS
_mm_cmpunord_ps	Unordered	CMPUNORDPS
_mm_comieq_ss	Equal	COMISS
_mm_comilt_ps	Less Than	COMISS
_mm_comile_ss	Less Than or Equal	COMISS
_mm_comigt_ss	Greater Than	COMISS
_mm_comige_ss	Greater Than or Equal	COMISS
_mm_comineq_ss	Not Equal	COMISS
_mm_ucomieq_ss	Equal	UCOMISS
_mm_ucomilt_ss	Less Than	UCOMISS
_mm_ucomile_ss	Less Than or Equal	UCOMISS
_mm_ucomigt_ss	Greater Than	UCOMISS
_mm_ucomige_ss	Greater Than or Equal	UCOMISS
_mm_ucomineq_ss	Not Equal	UCOMISS

```
__m128 _mm_cmpeq_ss(__m128 a, __m128 b)
```

Compare for equality.
```
r0 := (a0 == b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpeq_ps(__m128 a, __m128 b)
```

Compare for equality.
```
r0 := (a0 == b0) ? 0xffffffff : 0x0
r1 := (a1 == b1) ? 0xffffffff : 0x0
r2 := (a2 == b2) ? 0xffffffff : 0x0
r3 := (a3 == b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmplt_ss(__m128 a, __m128 b)
```

Compare for less-than.
```
r0 := (a0 < b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmplt_ps(__m128 a, __m128 b)
```

Compare for less-than.
```
r0 := (a0 < b0) ? 0xffffffff : 0x0
r1 := (a1 < b1) ? 0xffffffff : 0x0
r2 := (a2 < b2) ? 0xffffffff : 0x0
r3 := (a3 < b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmple_ss(__m128 a, __m128 b)
```

Compare for less-than-or-equal.
```
r0 := (a0 <= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmple_ps(__m128 a, __m128 b)
```

Compare for less-than-or-equal.
```
r0 := (a0 <= b0) ? 0xffffffff : 0x0
r1 := (a1 <= b1) ? 0xffffffff : 0x0
r2 := (a2 <= b2) ? 0xffffffff : 0x0
r3 := (a3 <= b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpgt_ss(__m128 a, __m128 b)
```

Compare for greater-than.
```
r0 := (a0 > b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpgt_ps(__m128 a, __m128 b)
```

Compare for greater-than.

```
r0 := (a0 > b0) ? 0xffffffff : 0x0
r1 := (a1 > b1) ? 0xffffffff : 0x0
r2 := (a2 > b2) ? 0xffffffff : 0x0
r3 := (a3 > b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpge_ss(__m128 a, __m128 b)
```

Compare for greater-than-or-equal.

```
r0 := (a0 >= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpge_ps(__m128 a, __m128 b)
```

Compare for greater-than-or-equal.

```
r0 := (a0 >= b0) ? 0xffffffff : 0x0
r1 := (a1 >= b1) ? 0xffffffff : 0x0
r2 := (a2 >= b2) ? 0xffffffff : 0x0
r3 := (a3 >= b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpneq_ss(__m128 a, __m128 b)
```

Compare for inequality.

```
r0 := (a0 != b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpneq_ps(__m128 a, __m128 b)
```

Compare for inequality.

```
r0 := (a0 != b0) ? 0xffffffff : 0x0
r1 := (a1 != b1) ? 0xffffffff : 0x0
r2 := (a2 != b2) ? 0xffffffff : 0x0
r3 := (a3 != b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)
```

Compare for not-less-than.

```
r0 := !(a0 < b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)
```

Compare for not-less-than.

```
r0 := !(a0 < b0) ? 0xffffffff : 0x0
r1 := !(a1 < b1) ? 0xffffffff : 0x0
r2 := !(a2 < b2) ? 0xffffffff : 0x0
r3 := !(a3 < b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpnle_ss(__m128 a, __m128 b)
```

Compare for not-less-than-or-equal.

```
r0 := !(a0 <= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpnle_ps(__m128 a, __m128 b)
```

Compare for not-less-than-or-equal.

```
r0 := !(a0 <= b0) ? 0xffffffff : 0x0
r1 := !(a1 <= b1) ? 0xffffffff : 0x0
r2 := !(a2 <= b2) ? 0xffffffff : 0x0
r3 := !(a3 <= b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpngt_ss(__m128 a, __m128 b)
```

Compare for not-greater-than.

```
r0 := !(a0 > b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpngt_ps(__m128 a, __m128 b)
```

Compare for not-greater-than.

```
r0 := !(a0 > b0) ? 0xffffffff : 0x0
r1 := !(a1 > b1) ? 0xffffffff : 0x0
r2 := !(a2 > b2) ? 0xffffffff : 0x0
r3 := !(a3 > b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpnge_ss(__m128 a, __m128 b)
```

Compare for not-greater-than-or-equal.

```
r0 := !(a0 >= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpnge_ps(__m128 a, __m128 b)
```

Compare for not-greater-than-or-equal.

```
r0 := !(a0 >= b0) ? 0xffffffff : 0x0
r1 := !(a1 >= b1) ? 0xffffffff : 0x0
r2 := !(a2 >= b2) ? 0xffffffff : 0x0
r3 := !(a3 >= b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpord_ss(__m128 a, __m128 b)
```

Compare for ordered.

```
r0 := (a0 ord? b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpord_ps(__m128 a, __m128 b)
```

Compare for ordered.

```
r0 := (a0 ord? b0) ? 0xffffffff : 0x0
r1 := (a1 ord? b1) ? 0xffffffff : 0x0
r2 := (a2 ord? b2) ? 0xffffffff : 0x0
r3 := (a3 ord? b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpunord_ss(__m128 a, __m128 b)
```

Compare for unordered.

```
r0 := (a0 unord? b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpunord_ps(__m128 a, __m128 b)
```

Compare for unordered.

```
r0 := (a0 unord? b0) ? 0xffffffff : 0x0
r1 := (a1 unord? b1) ? 0xffffffff : 0x0
r2 := (a2 unord? b2) ? 0xffffffff : 0x0
r3 := (a3 unord? b3) ? 0xffffffff : 0x0
```

```
int _mm_comieq_ss(__m128 a, __m128 b)
```

Compares the lower SP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.

```
r := (a0 == b0) ? 0x1 : 0x0
```

```
int _mm_comilt_ss(__m128 a, __m128 b)
```

Compares the lower SP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 < b0) ? 0x1 : 0x0
```

```
int _mm_comile_ss(__m128 a, __m128 b)
```

Compares the lower SP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 <= b0) ? 0x1 : 0x0
```

```
int _mm_comigt_ss(__m128 a, __m128 b)
```

Compares the lower SP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.

```
r := (a0 > b0) ? 0x1 : 0x0
```

```
int _mm_comige_ss(__m128 a, __m128 b)
```

Compares the lower SP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 >= b0) ? 0x1 : 0x0
```

```
int _mm_comineq_ss(__m128 a, __m128 b)
```

Compares the lower SP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.

```
r := (a0 != b0) ? 0x1 : 0x0
```

```
int _mm_ucomieq_ss(__m128 a, __m128 b)
```

Compares the lower SP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.

```
r := (a0 == b0) ? 0x1 : 0x0
```

```
int _mm_ucomilt_ss(__m128 a, __m128 b)
```

Compares the lower SP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 < b0) ? 0x1 : 0x0
```

```
int _mm_ucomile_ss(__m128 a, __m128 b)
```

Compares the lower SP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 <= b0) ? 0x1 : 0x0
```

```
int _mm_ucomigt_ss(__m128 a, __m128 b)
```

Compares the lower SP FP value of a and b for a greater than b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 > b0) ? 0x1 : 0x0
```

```
int _mm_ucomige_ss(__m128 a, __m128 b)
```

Compares the lower SP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 >= b0) ? 0x1 : 0x0
```

```
int _mm_ucomineq_ss(__m128 a, __m128 b)
```

Compares the lower SP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.

```
r := (a0 != b0) ? 0x1 : 0x0
```

Conversion Operations for Streaming SIMD Extensions

The conversions operations are listed in the following table followed by a description of each intrinsic with the most recent mnemonic naming convention. The alternate name is provided in case you have used these intrinsics before.

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

Intrinsic Name	Alternate Name	Corresponding Instruction
_mm_cvt_ss2si	_mm_cvtss_si32	CVTSS2SI
	_mm_cvtss_si64	CVTSS2SI
_mm_cvt_ps2pi	_mm_cvtps_pi32	CVTPS2PI
_mm_cvtt_ss2si	_mm_cvttss_si32	CVTTSS2SI
	_mm_cvttss_si64	CVTTSS2SI
_mm_cvtt_ps2pi	_mm_cvttps_pi32	CVTTPS2PI
_mm_cvt_si2ss	_mm_cvtsi32_ss	CVTSI2SS
	_mm_cvtsi64_ss	CVTSI2SS
_mm_cvt_pi2ps	_mm_cvtpi32_ps	CVTTPS2PI
_mm_cvtpi16_ps		composite
_mm_cvtpu16_ps		composite
_mm_cvtpi8_ps		composite
_mm_cvtpu8_ps		composite
_mm_cvtpi32x2_ps		composite
_mm_cvtps_pi16		composite
_mm_cvtps_pi8		composite

```
int _mm_cvt_ss2si(__m128 a)
```

Convert the lower SP FP value of a to a 32-bit integer according to the current rounding mode.

```
r := (int)a0
```

```
__int64 _mm_cvtss_si64(__m128 a)
```

Convert the lower SP FP value of a to a 64-bit signed integer according to the current rounding mode.

```
r := (__int64)a0
```

```
__m64 _mm_cvt_ps2pi(__m128 a)
```

Convert the two lower SP FP values of a to two 32-bit integers according to the current rounding mode, returning the integers in packed form.

```
r0 := (int)a0
r1 := (int)a1
```

```
int _mm_cvtt_ss2si(__m128 a)
```

Convert the lower SP FP value of a to a 32-bit integer with truncation.

```
r := (int)a0
```

```
__int64 _mm_cvttss_si64(__m128 a)
```

Convert the lower SP FP value of a to a 64-bit signed integer with truncation.

```
r:=(__int64)a0
```

```
__m64 _mm_cvtt_ps2pi(__m128 a)
```

Convert the two lower SP FP values of a to two 32-bit integer with truncation, returning the integers in packed form.

```
r0 := (int)a0
r1 := (int)a1
```

```
__m128 _mm_cvt_si2ss(__m128, int)
```

Convert the 32-bit integer value b to an SP FP value; the upper three SP FP values are passed through from a.

```
r0 := (float)b
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cvtsi64_ss(__m128 a, __int64 b)
```

Convert the signed 64-bit integer value b to an SP FP value; the upper three SP FP values are passed through from a.

```
r0 := (float)b
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cvt_pi2ps(__m128, __m64)
```

Convert the two 32-bit integer values in packed form in b to two SP FP values; the upper two SP FP values are passed through from a.

```
r0 := (float)b0
r1 := (float)b1
r2 := a2
r3 := a3
```

```
__inline __m128 _mm_cvtpi16_ps(__m64 a)
```

Convert the four 16-bit signed integer values in a to four single precision FP values.

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

```
__inline __m128 _mm_cvtpu16_ps(__m64 a)
```

Convert the four 16-bit unsigned integer values in a to four single precision FP values.

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

```
__inline __m128 _mm_cvtpi8_ps(__m64 a)
```

Convert the lower four 8-bit signed integer values in a to four single precision FP values.

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

```
__inline __m128 _mm_cvtpu8_ps(__m64 a)
```

Convert the lower four 8-bit unsigned integer values in a to four single precision FP values.

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

```
__inline __m128 _mm_cvtpi32x2_ps(__m64 a, __m64 b)
```

Convert the two 32-bit signed integer values in a and the two 32-bit signed integer values in b to four single precision FP values.

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)b0
r3 := (float)b1
```

```
__inline __m64 _mm_cvtps_pi16(__m128 a)
```

Convert the four single precision FP values in a to four signed 16-bit integer values.

```
r0 := (short)a0
r1 := (short)a1
r2 := (short)a2
r3 := (short)a3
```

```
__inline __m64 _mm_cvtps_pi8(__m128 a)
```

Convert the four single precision FP values in a to the lower four signed 8-bit integer values of the result.

```
r0 := (char)a0
r1 := (char)a1
r2 := (char)a2
r3 := (char)a3
```

Load Operations for Streaming SIMD Extensions

See summary table in Summary of Memory and Initialization topic.

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

```
__m128 _mm_load_ss(float * p )
```

Loads an SP FP value into the low word and clears the upper three words.

```
r0 := *p
r1 := 0.0 ; r2 := 0.0 ; r3 := 0.0
```

```
__m128 _mm_load_ps1(float * p )
```

Loads a single SP FP value, copying it into all four words.

```
r0 := *p
r1 := *p
r2 := *p
r3 := *p
```

```
__m128 _mm_load_ps(float * p )
```

Loads four SP FP values. The address must be 16-byte-aligned.

```
r0 := p[0]
r1 := p[1]
r2 := p[2]
r3 := p[3]
```

```
__m128 _mm_loadu_ps(float * p)
```

Loads four SP FP values. The address need not be 16-byte-aligned.

```
r0 := p[0]
r1 := p[1]
r2 := p[2]
r3 := p[3]
```

```
__m128 _mm_loadr_ps(float * p)
```

Loads four SP FP values in reverse order. The address must be 16-byte-aligned.

```
r0 := p[3]
r1 := p[2]
r2 := p[1]
r3 := p[0]
```

Set Operations for Streaming SIMD Extensions

See summary table in Summary of Memory and Initialization topic.

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

```
__m128 _mm_set_ss(float w )
```

Sets the low word of an SP FP value to `w` and clears the upper three words.

```
r0 := w
r1 := r2 := r3 := 0.0
```

```
__m128 _mm_set_ps1(float w )
```

Sets the four SP FP values to `w`.

```
r0 := r1 := r2 := r3 := w
```

```
__m128 _mm_set_ps(float z, float y, float x, float w )
```

Sets the four SP FP values to the four inputs.

```
r0 := w
r1 := x
r2 := y
r3 := z
```

```
__m128 _mm_setr_ps(float z, float y, float x, float w )
```

Sets the four SP FP values to the four inputs in reverse order.

```
r0 := z
r1 := y
r2 := x
r3 := w
```

```
__m128 _mm_setzero_ps(void)
```

Clears the four SP FP values.

```
r0 := r1 := r2 := r3 := 0.0
```

Store Operations for Streaming SIMD Extensions

See summary table in Summary of Memory and Initialization topic.

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

```
void _mm_store_ss(float * p, __m128 a)
```

Stores the lower SP FP value.

```
*p := a0
```

```
void _mm_store_ps1(float * p, __m128 a )
```

Stores the lower SP FP value across four words.

```
p[0] := a0
p[1] := a0
p[2] := a0
p[3] := a0
```

```
void _mm_store_ps(float *p, __m128 a)
```

Stores four SP FP values. The address must be 16-byte-aligned.

```
p[0] := a0
p[1] := a1
p[2] := a2
p[3] := a3
```

```
void _mm_storeu_ps(float *p, __m128 a)
```

Stores four SP FP values. The address need not be 16-byte-aligned.

```
p[0] := a0
p[1] := a1
p[2] := a2
p[3] := a3
```

```
void _mm_storer_ps(float * p, __m128 a )
```

Stores four SP FP values in reverse order. The address must be 16-byte-aligned.

```
p[0] := a3
p[1] := a2
p[2] := a1
p[3] := a0
```

```
__m128 _mm_move_ss( __m128 a, __m128 b)
```

Sets the low word to the SP FP value of `b`. The upper 3 SP FP values are passed through from `a`.

```
r0 := b0
r1 := a1
r2 := a2
r3 := a3
```

Cacheability Support Using Streaming SIMD Extensions

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

`void _mm_pause(void)`

The execution of the next instruction is delayed an implementation specific amount of time. The instruction does not modify the architectural state. This intrinsic provides especially significant performance gain.

PAUSE Intrinsic

The PAUSE intrinsic is used in spin-wait loops with the processors implementing dynamic execution (especially out-of-order execution). In the spin-wait loop, PAUSE improves the speed at which the code detects the release of the lock. For dynamic scheduling, the PAUSE instruction reduces the penalty of exiting from the spin-loop.

Example of loop with the PAUSE instruction:

```
spin_loop:pause
cmp eax, A
jne spin_loop
```

In this example, the program spins until memory location A matches the value in register eax. The code sequence that follows shows a test-and-test-and-set. In this example, the spin occurs only after the attempt to get a lock has failed.

```
get_lock: mov eax, 1
xchg eax, A ; Try to get lock
cmp eax, 0 ; Test if successful
jne spin_loop
```

Critical Section

```
// critical_section code
mov A, 0 ; Release lock
jmp continue
spin_loop: pause;
// spin-loop hint
cmp 0, A ;
// check lock availability
jne spin_loop
jmp get_lock
// continue: other code
```

Note that the first branch is predicted to fall-through to the critical section in anticipation of successfully gaining access to the lock. It is highly recommended

that all spin-wait loops include the PAUSE instruction. Since PAUSE is backwards compatible to all existing IA-32 processor generations, a test for processor type (a CPUID test) is not needed. All legacy processors will execute PAUSE as a NOP, but in processors which use the PAUSE as a hint there can be significant performance benefit.

Integer Intrinsics Using Streaming SIMD Extensions

The integer intrinsics are listed in the following table followed by a description of each intrinsic with the most recent mnemonic naming convention.

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

Intrinsic Name	Alternate Name	Operation	Corresponding Instruction
`_m_pextrw`	`_mm_extract_pi16`	Extract on of four words	PEXTRW
`_m_pinsrw`	`_mm_insert_pi16`	Insert a word	PINSRW
`_m_pmaxsw`	`_mm_max_pi16`	Compute the maximum	PMAXSW
`_m_pmaxub`	`_mm_max_pu8`	Compute the maximum, unsigned	PMAXUB
`_m_pminsw`	`_mm_min_pi16`	Compute the minimum	PMINSW
`_m_pminub`	`_mm_min_pu8`	Compute the minimum, unsigned	PMINUB
`_m_pmovmskb`	`_mm_movemask_pi8`	Create an eight-bit mask	PMOVMSKB
`_m_pmulhuw`	`_mm_mulhi_pu16`	Multiply, return high bits	PMULHUW
`_m_pshufw`	`_mm_shuffle_pi16`	Return a combination of four words	PSHUFW
`_m_maskmovq`	`_mm_maskmove_si64`	Conditional Store	MASKMOVQ

Intrinsic Name	Alternate Name	Operation	Corresponding Instruction
_m_pavgb	_mm_avg_pu8	Compute rounded average	PAVGB
_m_pavgw	_mm_avg_pu16	Compute rounded average	PAVGW
_m_psadbw	_mm_sad_pu8	Compute sum of absolute differences	PSADBW

For these intrinsics you need to empty the multimedia state for the mmx register. See The EMMS Instruction: Why You Need It and When to Use It topic for more details.

```
int _m_pextrw(__m64 a, int n)
```

Extracts one of the four words of a. The selector n must be an immediate.

```
r := (n==0) ? a0 : ( (n==1) ? a1 : ( (n==2) ? a2 : a3 ) )
```

```
__m64 _m_pinsrw(__m64 a, int d, int n)
```

Inserts word d into one of four words of a. The selector n must be an immediate.

```
r0 := (n==0) ? d : a0;
r1 := (n==1) ? d : a1;
r2 := (n==2) ? d : a2;
r3 := (n==3) ? d : a3;
```

```
__m64 _m_pmaxsw(__m64 a, __m64 b)
```

Computes the element-wise maximum of the words in a and b.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

```
__m64 _m_pmaxub(__m64 a, __m64 b)
```

Computes the element-wise maximum of the unsigned bytes in a and b.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

```
__m64 _m_pminsw(__m64 a, __m64 b)
```

Computes the element-wise minimum of the words in a and b.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

```
__m64 _m_pminub(__m64 a, __m64 b)
```

Computes the element-wise minimum of the unsigned bytes in a and b.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

```
int _m_pmovmskb(__m64 a)
```

Creates an 8-bit mask from the most significant bits of the bytes in a.

```
r := sign(a7)<<7 | sign(a6)<<6 |... | sign(a0)
```

```
__m64 _m_pmulhuw(__m64 a, __m64 b)
```

Multiplies the unsigned words in a and b, returning the upper 16 bits of the 32-bit intermediate results.

```
r0 := hiword(a0 * b0)
r1 := hiword(a1 * b1)
r2 := hiword(a2 * b2)
r3 := hiword(a3 * b3)
```

```
__m64 _m_pshufw(__m64 a, int n)
```

Returns a combination of the four words of a. The selector n must be an immediate.

```
r0 := word (n&0x3) of a
r1 := word ((n>>2)&0x3) of a
r2 := word ((n>>4)&0x3) of a
r3 := word ((n>>6)&0x3) of a
```

```
void _m_maskmovq(__m64 d, __m64 n, char *p)
```

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

```
if (sign(n0)) p[0] := d0
if (sign(n1)) p[1] := d1
...
if (sign(n7)) p[7] := d7
```

```
__m64 _m_pavgb(__m64 a, __m64 b)
```

Computes the (rounded) averages of the unsigned bytes in a and b.

```
t = (unsigned short)a0 + (unsigned short)b0
r0 = (t >> 1) | (t & 0x01)
...
t = (unsigned short)a7 + (unsigned short)b7
r7 = (unsigned char)((t >> 1) | (t & 0x01))
```

```
__m64 _m_pavgw(__m64 a, __m64 b)
```

Computes the (rounded) averages of the unsigned words in a and b.

```
t = (unsigned int)a0 + (unsigned int)b0
r0 = (t >> 1) | (t & 0x01)
...
t = (unsigned word)a7 + (unsigned word)b7
r7 = (unsigned short)((t >> 1) | (t & 0x01))
```

```
__m64 _m_psadbw(__m64 a, __m64 b)
```

Computes the sum of the absolute differences of the unsigned bytes in a and b, returning he value in the lower word. The upper three words are cleared.

```
r0 = abs(a0-b0) +... + abs(a7-b7)
r1 = r2 = r3 = 0
```

Memory and Initialization Using Streaming SIMD Extensions

This section describes the `load`, `set`, and `store` operations, which let you load and store data into memory. The `load` and `set` operations are similar in that both initialize `__m128` data. However, the `set` operations take a float argument and are intended for initialization with constants, whereas the `load` operations take a floating point argument and are intended to mimic the instructions for loading data from memory. The `store` operation assigns the initialized data to the address.

The intrinsics are listed in the following table. Syntax and a brief description are contained the following topics.

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

Intrinsic Name	Alternate Name	Operation	Corresponding Instruction
`_mm_load_ss`		Load the low value and clear the three high values	`MOVSS`
`_mm_load_ps1`	`_mm_load1_ps`	Load one value into all four words	`MOVSS + Shuffling`
`_mm_load_ps`		Load four values, address aligned	`MOVAPS`
`_mm_loadu_ps`		Load four values, address unaligned	`MOVUPS`
`_mm_loadr_ps`		Load four values, in reverse order	`MOVAPS + Shuffling`
`_mm_set_ss`		Set the low value and clear the three high values	Composite
`_mm_set_ps1`	`_mm_set1_ps`	Set all four words with the same value	Composite
`_mm_set_ps`		Set four values, address aligned	Composite
`_mm_setr_ps`		Set four values, in reverse order	Composite
`_mm_setzero_ps`		Clear all four values	Composite
`_mm_store_ss`		Store the low value	`MOVSS`
`_mm_store_ps1`	`_mm_store1_ps`	Store the low value across all four words. The address must be 16-byte aligned.	`Shuffling + MOVSS`

Intrinsic Name	Alternate Name	Operation	Corresponding Instruction
_mm_store_ps		Store four values, address aligned	MOVAPS
_mm_storeu_ps		Store four values, address unaligned	MOVUPS
_mm_storer_ps		Store four values, in reverse order	MOVAPS + Shuffling
_mm_move_ss		Set the low word, and pass in three high values	MOVSS
_mm_getcsr		Return register contents	STMXCSR
_mm_setcsr		Control Register	LDMXCSR
_mm_prefetch			
_mm_stream_pi			
_mm_stream_ps			
_mm_sfence			
_mm_cvtss_f32			

```
__m128 _mm_load_ss(float const*a)
```

Loads an SP FP value into the low word and clears the upper three words.

```
r0 := *a
r1 := 0.0 ; r2 := 0.0 ; r3 := 0.0
```

```
__m128 _mm_load_ps1(float const*a)
```

Loads a single SP FP value, copying it into all four words.

```
r0 := *a
r1 := *a
r2 := *a
r3 := *a
```

```
__m128 _mm_load_ps(float const*a)
```

Loads four SP FP values. The address must be 16-byte-aligned.

```
r0 := a[0]
r1 := a[1]
r2 := a[2]
r3 := a[3]
```

```
__m128 _mm_loadu_ps(float const*a)
```

Loads four SP FP values. The address need not be 16-byte-aligned.

```
r0 := a[0]
r1 := a[1]
r2 := a[2]
r3 := a[3]
```

```
__m128 _mm_loadr_ps(float const*a)
```

Loads four SP FP values in reverse order. The address must be 16-byte-aligned.

```
r0 := a[3]
r1 := a[2]
r2 := a[1]
r3 := a[0]
```

```
__m128 _mm_set_ss(float a)
```

Sets the low word of an SP FP value to a and clears the upper three words.

```
r0 := c
r1 := r2 := r3 := 0.0
```

```
__m128 _mm_set_ps1(float a)
```

Sets the four SP FP values to a.

```
r0 := r1 := r2 := r3 := a
```

```
__m128 _mm_set_ps(float a, float b, float c, float d)
```

Sets the four SP FP values to the four inputs.

```
r0 := a
r1 := b
r2 := c
r3 := d
```

```
__m128 _mm_setr_ps(float a, float b, float c, float d)
```

Sets the four SP FP values to the four inputs in reverse order.

```
r0 := d
r1 := c
r2 := b
r3 := a
```

```
__m128 _mm_setzero_ps(void)
```

Clears the four SP FP values.

```
r0 := r1 := r2 := r3 := 0.0
```

```
void _mm_store_ss(float *v, __m128 a)
```

Stores the lower SP FP value.

```
*v := a0
```

```
void _mm_store_ps1(float *v, __m128 a)
```

Stores the lower SP FP value across four words.

```
v[0] := a0
v[1] := a0
v[2] := a0
v[3] := a0
```

```
void _mm_store_ps(float *v, __m128 a)
```

Stores four SP FP values. The address must be 16-byte-aligned.

```
v[0] := a0
v[1] := a1
v[2] := a2
v[3] := a3
```

```
void _mm_storeu_ps(float *v, __m128 a)
```

Stores four SP FP values. The address need not be 16-byte-aligned.

```
v[0] := a0
v[1] := a1
v[2] := a2
v[3] := a3
```

```
void _mm_storer_ps(float *v, __m128 a)
```

Stores four SP FP values in reverse order. The address must be 16-byte-aligned.

```
v[0] := a3
v[1] := a2
v[2] := a1
v[3] := a0
```

```
__m128 _mm_move_ss(__m128 a, __m128 b)
```

Sets the low word to the SP FP value of b. The upper 3 SP FP values are passed through from a.

```
r0 := b0
r1 := a1
r2 := a2
r3 := a3
```

```
unsigned int _mm_getcsr(void)
```

Returns the contents of the control register.

`void _mm_setcsr(unsigned int i)`

Sets the control register to the value specified.

`void _mm_prefetch(char const*a, int sel)`

(uses `PREFETCH`) Loads one cache line of data from address a to a location "closer" to the processor. The value `sel` specifies the type of prefetch operation: the constants `_MM_HINT_T0`, `_MM_HINT_T1`, `_MM_HINT_T2`, and `_MM_HINT_NTA` should be used for IA-32, corresponding to the type of `prefetch` instruction. The constants `_MM_HINT_T1`, `_MM_HINT_NT1`, `_MM_HINT_NT2`, and `_MM_HINT_NTA` should be used for Itanium®-based systems.

`void _mm_stream_pi(__m64 *p, __m64 a)`

(uses `MOVNTQ`) Stores the data in a to the address p without polluting the caches. This intrinsic requires you to empty the multimedia state for the `mmx` register. See The EMMS Instruction: Why You Need It and When to Use It topic.

`void _mm_stream_ps(float *p, __m128 a)`

(see `MOVNTPS`) Stores the data in a to the address p without polluting the caches. The address must be 16-byte-aligned.

`void _mm_sfence(void)`

(uses `SFENCE`) Guarantees that every preceding store is globally visible before any subsequent store.

`float _mm_cvtss_f32(__m128 a)`

This intrinsic extracts a single precision floating point value from the first vector element of an `__m128`. It does so in the most effecient manner possible in the context used. This intrinsic doesn't map to any specific SSE instruction.

Miscellaneous Intrinsics Using Streaming SIMD Extensions

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

Intrinsic Name	Operation	Corresponding Instruction
`_mm_shuffle_ps`	Shuffle	`SHUFPS`
`_mm_unpackhi_ps`	Unpack High	`UNPCKHPS`
`_mm_unpacklo_ps`	Unpack Low	`UNPCKLPS`

Intrinsic Name	Operation	Corresponding Instruction
_mm_loadh_pi	Load High	MOVHPS reg, mem
_mm_storeh_pi	Store High	MOVHPS mem, reg
_mm_movehl_ps	Move High to Low	MOVHLPS
_mm_movelh_ps	Move Low to High	MOVLHPS
_mm_loadl_pi	Load Low	MOVLPS reg, mem
_mm_storel_pi	Store Low	MOVLPS mem, reg
_mm_movemask_ps	Create four-bit mask	MOVMSKPS

```
__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int
imm8)
```

Selects four specific SP FP values from a and b, based on the mask imm8. The mask must be an immediate. See Macro Function for Shuffle Using Streaming SIMD Extensions for a description of the shuffle semantics.

```
__m128 _mm_unpackhi_ps(__m128 a, __m128 b)
```

Selects and interleaves the upper two SP FP values from a and b.

```
r0 := a2
r1 := b2
r2 := a3
r3 := b3
```

```
__m128 _mm_unpacklo_ps(__m128 a, __m128 b)
```

Selects and interleaves the lower two SP FP values from a and b.

```
r0 := a0
r1 := b0
r2 := a1
r3 := b1
```

```
__m128 _mm_loadh_pi(__m128, __m64 const *p)
```

Sets the upper two SP FP values with 64 bits of data loaded from the address p.

```
r0 := a0
r1 := a1
r2 := *p0
r3 := *p1
```

```
void _mm_storeh_pi(__m64 *p, __m128 a)
```

Stores the upper two SP FP values to the address p.

```
*p0 := a2
*p1 := a3
```

```
__m128 _mm_movehl_ps(__m128 a, __m128 b)
```

Moves the upper 2 SP FP values of b to the lower 2 SP FP values of the result. The upper 2 SP FP values of a are passed through to the result.

```
r3 := a3
r2 := a2
r1 := b3
r0 := b2
```

```
__m128 _mm_movelh_ps(__m128 a, __m128 b)
```

Moves the lower 2 SP FP values of b to the upper 2 SP FP values of the result. The lower 2 SP FP values of a are passed through to the result.

```
r3 := b1
r2 := b0
r1 := a1
r0 := a0
```

```
__m128 _mm_loadl_pi(__m128 a, __m64 const *p)
```

Sets the lower two SP FP values with 64 bits of data loaded from the address p; the upper two values are passed through from a.

```
r0 := *p0
r1 := *p1
r2 := a2
r3 := a3
```

```
void _mm_storel_pi(__m64 *p, __m128 a)
```

Stores the lower two SP FP values of a to the address p.

```
*p0 := a0
*p1 := a1
```

```
int _mm_movemask_ps(__m128 a)
```

Creates a 4-bit mask from the most significant bits of the four SP FP values.

```
r := sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 | sign(a0)
```

Using Streaming SIMD Extensions on Itanium® Architecture

The Streaming SIMD Extensions (SSE) intrinsics provide access to Itanium® instructions for Streaming SIMD Extensions. To provide source compatibility with the IA-32 architecture, these intrinsics are equivalent both in name and functionality to the set of IA-32-based SSE intrinsics.

To write programs with the intrinsics, you should be familiar with the hardware features provided by SSE. Keep the following issues in mind:

- Certain intrinsics are provided only for compatibility with previously-defined IA-32 intrinsics. Using them on Itanium-based systems probably leads to performance degradation.
- Floating-point (FP) data loaded stored as `__m128` objects must be 16-byte-aligned.
- Some intrinsics require that their arguments be immediates -- that is, constant integers (literals), due to the nature of the instruction.

Data Types

The new data type `__m128` is used with the SSE intrinsics. It represents a 128-bit quantity composed of four single-precision FP values. This corresponds to the 128-bit IA-32 Streaming SIMD Extensions register.

The compiler aligns `__m128` local data to 16-byte boundaries on the stack. Global data of these types is also 16 byte-aligned. To align `integer`, `float`, or `double` arrays, you can use the `declspec` alignment.

Because Itanium instructions treat the SSE registers in the same way whether you are using packed or scalar data, there is no `__m32` data type to represent scalar data. For scalar operations, use the `__m128` objects and the "scalar" forms of the intrinsics; the compiler and the processor implement these operations with 32-bit memory references. But, for better performance the packed form should be substituting for the scalar form whenever possible.

The address of a `__m128` object may be taken.

For more information, see Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual, Intel Corporation, doc. number 243191.

Implementation on Itanium-based systems

SSE intrinsics are defined for the `__m128` data type, a 128-bit quantity consisting of four single-precision FP values. SIMD instructions for Itanium-based systems operate on 64-bit FP register quantities containing two single-precision floating-point values. Thus, each `__m128` operand is actually a pair of FP registers and

therefore each intrinsic corresponds to at least one pair of Itanium instructions operating on the pair of FP register operands.

Compatibility versus Performance

Many of the SSE intrinsics for Itanium-based systems were created for compatibility with existing IA-32 intrinsics and not for performance. In some situations, intrinsic usage that improved performance on IA-32 will not do so on Itanium-based systems. One reason for this is that some intrinsics map nicely into the IA-32 instruction set but not into the Itanium instruction set. Thus, it is important to differentiate between intrinsics which were implemented for a performance advantage on Itanium-based systems, and those implemented simply to provide compatibility with existing IA-32 code.

The following intrinsics are likely to reduce performance and should only be used to initially port legacy code or in non-critical code sections:

- Any SSE scalar intrinsic (`_ss variety`) - use packed (`_ps`) version if possible
- `comi` and `ucomi` SSE comparisons - these correspond to IA-32 `COMISS` and `UCOMISS` instructions only. A sequence of Itanium instructions are required to implement these.
- Conversions in general are multi-instruction operations. These are particularly expensive: `_mm_cvtpi16_ps`, `_mm_cvtpu16_ps`, `_mm_cvtpi8_ps`, `_mm_cvtpu8_ps`, `_mm_cvtpi32x2_ps`, `_mm_cvtps_pi16`, `_mm_cvtps_pi8`
- SSE utility intrinsic `_mm_movemask_ps`

If the inaccuracy is acceptable, the SIMD reciprocal and reciprocal square root approximation intrinsics (`rcp` and `rsqrt`) are much faster than the true `div` and `sqrt` intrinsics.

Macro Function for Shuffle Using Streaming SIMD Extensions

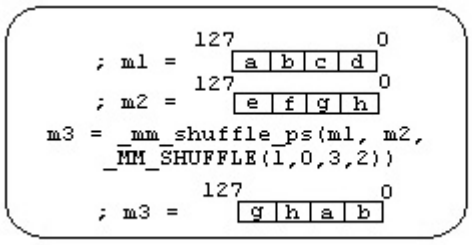
The Streaming SIMD Extensions (SSE) provide a macro function to help create constants that describe shuffle operations. The macro takes four small integers (in the range of 0 to 3) and combines them into an 8-bit immediate value used by the SHUFPS instruction.

Shuffle Function Macro

```
  _MM_SHUFFLE(z,y,x,w)
/* expands to the following value */
 (z<<6) | (y<<4) | (x<<2)| w
```

You can view the four integers as selectors for choosing which two words from the first input operand and which two words from the second are to be put into the result word.

View of Original and Result Words with Shuffle Function Macro



Macro Functions to Read and Write the Control Registers

The following macro functions enable you to read and write bits to and from the control register. For details, see Set Operations. For Itanium®-based systems, these macros do not allow you to access all of the bits of the FPSR. See the descriptions for the `getfpsr()` and `setfpsr()` intrinsics in the Native Intrinsics for Itanium Instructions topic.

Exception State Macros	Macro Arguments
`_MM_SET_EXCEPTION_STATE(x)`	`_MM_EXCEPT_INVALID`
`_MM_GET_EXCEPTION_STATE()`	`_MM_EXCEPT_DIV_ZERO`
	`_MM_EXCEPT_DENORM`
Macro Definitions Write to and read from the sixth-least significant control register bit, respectively.	`_MM_EXCEPT_OVERFLOW`
	`_MM_EXCEPT_UNDERFLOW`
	`_MM_EXCEPT_INEXACT`

The following example tests for a divide-by-zero exception.

Exception State Macros with _MM_EXCEPT_DIV_ZERO

```
if (_MM_GET_EXCEPTION_STATE(x) & _MM_EXCEPT_DIV_ZERO) {
    /* Exception has occurred */
}
```

Exception Mask Macros	Macro Arguments
_MM_SET_EXCEPTION_MASK(x)	_MM_MASK_INVALID
_MM_GET_EXCEPTION_MASK ()	_MM_MASK_DIV_ZERO
	_MM_MASK_DENORM
Macro Definitions Write to and read from the seventh through twelfth control register bits, respectively. **Note**: All six exception mask bits are always affected. Bits not set explicitly are cleared.	_MM_MASK_OVERFLOW
	_MM_MASK_UNDERFLOW
	_MM_MASK_INEXACT

The following example masks the overflow and underflow exceptions and unmasks all other exceptions.

Exception Mask with _MM_MASK_OVERFLOW and _MM_MASK_UNDERFLOW	
_MM_SET_EXCEPTION_MASK(MM_MASK_OVERFLOW | _MM_MASK_UNDERFLOW)	
Rounding Mode	**Macro Arguments**
_MM_SET_ROUNDING_MODE(x)	_MM_ROUND_NEAREST
_MM_GET_ROUNDING_MODE()	_MM_ROUND_DOWN
Macro Definition Write to and read from bits thirteen and fourteen of the control register.	_MM_ROUND_UP
	_MM_ROUND_TOWARD_ZERO

The following example tests the rounding mode for round toward zero.

Rounding Mode with _MM_ROUND_TOWARD_ZERO

```
if (_MM_GET_ROUNDING_MODE() == _MM_ROUND_TOWARD_ZERO) {
/* Rounding mode is round toward zero */
}
```

Flush-to-Zero Mode	Macro Arguments
`_MM_SET_FLUSH_ZERO_MODE(x)`	`_MM_FLUSH_ZERO_ON`
`_MM_GET_FLUSH_ZERO_MODE()`	`_MM_FLUSH_ZERO_OFF`
Macro Definition Write to and read from bit fifteen of the control register.	

The following example disables flush-to-zero mode.

Flush-to-Zero Mode with _MM_FLUSH_ZERO_OFF
`_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_OFF)`

Macro Function for Matrix Transposition

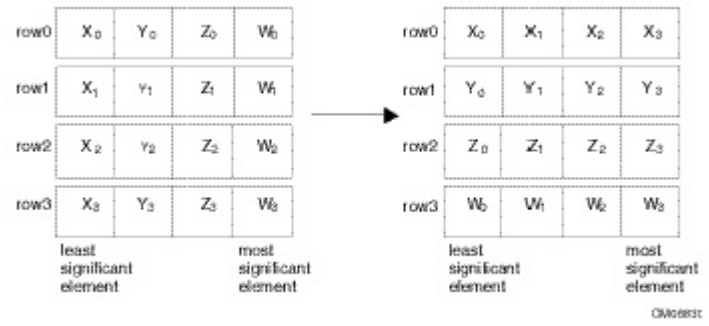
The Streaming SIMD Extensions (SSE) also provide the following macro function to transpose a 4 by 4 matrix of single precision floating point values.

`_MM_TRANSPOSE4_PS(row0, row1, row2, row3)`

The arguments `row0`, `row1`, `row2`, and `row3` are `__m128` values whose elements form the corresponding rows of a 4 by 4 matrix. The matrix transposition is returned in arguments `row0`, `row1`, `row2`, and `row3` where `row0` now holds column 0 of the original matrix, `row1` now holds column 1 of the original matrix, and so on.

The transposition function of this macro is illustrated in the "Matrix Transposition Using the `_MM_TRANSPOSE4_PS`" figure.

Matrix Transposition Using _MM_TRANSPOSE4_PS Macro



Streaming SIMD Extensions 2

This section describes the C++ language-level features supporting the Intel® Pentium® 4 processor Streaming SIMD Extensions 2 (SSE2) in the Intel® C++ Compiler, which are divided into two categories:

- Floating-Point Intrinsics -- describes the arithmetic, logical, compare, conversion, memory, and initialization intrinsics for the double-precision floating-point data type (`__m128d`).
- Integer Intrinsics -- describes the arithmetic, logical, compare, conversion, memory, and initialization intrinsics for the extended-precision integer data type (`__m128i`).

Note

The Pentium 4 processor SSE2 intrinsics are defined only for IA-32 platforms, not Itanium®-based platforms. Pentium 4 processor SSE2 operate on 128 bit quantities -- 2 64-bit double precision floating point values. The Itanium processor does not support parallel double precision computation, so Pentium 4 processor SSE2 are not implemented on Itanium-based systems.

For more details, refer to the *Pentium® 4 processor Streaming SIMD Extensions 2 External Architecture Specification (EAS)* and other Pentium 4 processor manuals available for download from the developer.intel.com web site. You should be familiar with the hardware features provided by the StSE2 when writing programs with the intrinsics. The following are three important issues to keep in mind:

- Certain intrinsics, such as `_mm_loadr_pd` and `_mm_cmpgt_sd`, are not directly supported by the instruction set. While these intrinsics are convenient programming aids, be mindful of their implementation cost.
- Data loaded or stored as `__m128d` objects must be generally 16-byte-aligned.
- Some intrinsics require that their argument be immediates, that is, constant integers (literals), due to the nature of the instruction.

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

Note

You can also use the single `ia32intrin.h` header file for any IA-32 intrinsics.

Floating-point Arithmetic Operations for Streaming SIMD Extensions 2

The arithmetic operations for the Streaming SIMD Extensions 2 (SSE2) are listed in the following table. The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

Intrinsic Name	Corresponding Instruction	Operation	R0 Value	R1 Value
`_mm_add_sd`	`ADDSD`	Addition	`a0 [op] b0`	`a1`
`_mm_add_pd`	`ADDPD`	Addition	`a0 [op] b0`	`a1 [op] b1`
`_mm_sub_sd`	`SUBSD`	Subtraction	`a0 [op] b0`	`a1`
`_mm_sub_pd`	`SUBPD`	Subtraction	`a0 [op] b0`	`a1 [op] b1`
`_mm_mul_sd`	`MULSD`	Multiplication	`a0 [op] b0`	`a1`
`_mm_mul_pd`	`MULPD`	Multiplication	`a0 [op] b0`	`a1 [op] b1`
`_mm_div_sd`	`DIVSD`	Division	`a0 [op] b0`	`a1`
`_mm_div_pd`	`DIVPD`	Division	`a0 [op] b0`	`a1 [op] b1`
`_mm_sqrt_sd`	`SQRTSD`	Computes Square Root	`a0 [op] b0`	`a1`
`_mm_sqrt_pd`	`SQRTPD`	Computes Square Root	`a0 [op] b0`	`a1 [op] b1`
`_mm_min_sd`	`MINSD`	Computes Minimum	`a0 [op] b0`	`a1`
`_mm_min_pd`	`MINPD`	Computes Minimum	`a0 [op] b0`	`a1 [op] b1`
`_mm_max_sd`	`MAXSD`	Computes Maximum	`a0 [op] b0`	`a1`

Intrinsic Name	Corresponding Instruction	Operation	R0 Value	R1 Value
_mm_max_pd	MAXPD	Computes Maximum	a0 [op] b0	a1 [op] b1

```
__m128d _mm_add_sd(__m128d a, __m128d b)
```

Adds the lower DP FP (double-precision, floating-point) values of a and b ; the upper DP FP value is passed through from a.

```
r0 := a0 + b0
r1 := a1
```

```
__m128d _mm_add_pd(__m128d a, __m128d b)
```

Adds the two DP FP values of a and b.

```
r0 := a0 + b0
r1 := a1 + b1
```

```
__m128d _mm_sub_sd(__m128d a, __m128d b)
```

Subtracts the lower DP FP value of b from a. The upper DP FP value is passed through from a.

```
r0 := a0 - b0
r1 := a1
```

```
__m128d _mm_sub_pd(__m128d a, __m128d b)
```

Subtracts the two DP FP values of b from a.

```
r0 := a0 - b0
r1 := a1 - b1
```

```
__m128d _mm_mul_sd(__m128d a, __m128d b)
```

Multiplies the lower DP FP values of a and b. The upper DP FP is passed through from a.

```
r0 := a0 * b0
r1 := a1
```

```
__m128d _mm_mul_pd(__m128d a, __m128d b)
```

Multiplies the two DP FP values of a and b.

```
r0 := a0 * b0
r1 := a1 * b1
```

```
__m128d _mm_div_sd(__m128d a, __m128d b)
```

Divides the lower DP FP values of a and b. The upper DP FP value is passed through from a.

```
r0 := a0 / b0
r1 := a1
```

```
__m128d _mm_div_pd(__m128d a, __m128d b)
```

Divides the two DP FP values of a and b.

```
r0 := a0 / b0
r1 := a1 / b1
```

```
__m128d _mm_sqrt_sd(__m128d a, __m128d b)
```

Computes the square root of the lower DP FP value of b. The upper DP FP value is passed through from a.

```
r0 := sqrt(b0)
r1 := a1
```

```
__m128d _mm_sqrt_pd(__m128d a)
```

Computes the square roots of the two DP FP values of a.

```
r0 := sqrt(a0)
r1 := sqrt(a1)
```

```
__m128d _mm_min_sd(__m128d a, __m128d b)
```

Computes the minimum of the lower DP FP values of a and b. The upper DP FP value is passed through from a.

```
r0 := min (a0, b0)
r1 := a1
```

```
__m128d _mm_min_pd(__m128d a, __m128d b)
```

Computes the minima of the two DP FP values of a and b.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
```

```
__m128d _mm_max_sd(__m128d a, __m128d b)
```

Computes the maximum of the lower DP FP values of a and b. The upper DP FP value is passed through from a.

```
r0 := max (a0, b0)
r1 := a1
```

```
__m128d _mm_max_pd(__m128d a, __m128d b)
```

Computes the maxima of the two DP FP values of a and b.

```
r0 := max(a0, b0)
r1 := max(a1, b1)
```

Floating-point Logical Operations for Streaming SIMD Extensions 2

The prototypes for Streaming SIMD Extensions 2 (SSE2) intrinsics are in the `emmintrin.h` header file.

```
__m128d _mm_and_pd(__m128d a, __m128d b)
```

(uses `ANDPD`) Computes the bitwise `AND` of the two DP FP values of a and b.

```
r0 := a0 & b0
r1 := a1 & b1
```

```
__m128d _mm_andnot_pd(__m128d a, __m128d b)
```

(uses `ANDNPD`) Computes the bitwise `AND` of the 128-bit value in b and the bitwise `NOT` of the 128-bit value in a.

```
r0 := (~a0) & b0
r1 := (~a1) & b1
```

```
__m128d _mm_or_pd(__m128d a, __m128d b)
```

(uses `ORPD`) Computes the bitwise `OR` of the two DP FP values of a and b.

```
r0 := a0 | b0
r1 := a1 | b1
```

```
__m128d _mm_xor_pd(__m128d a, __m128d b)
```

(uses `XORPD`) Computes the bitwise `XOR` of the two DP FP values of a and b.

```
r0 := a0 ^ b0
r1 := a1 ^ b1
```

Floating-point Comparison Operations for Streaming SIMD Extensions 2

Each comparison intrinsic performs a comparison of a and b. For the packed form, the two DP FP values of a and b are compared, and a 128-bit mask is returned. For the scalar form, the lower DP FP values of a and b are compared, and a 64-bit mask is returned; the upper DP FP value is passed through from a. The mask is set to `0xffffffffffffffff` for each element where the comparison is true and 0x0 where the comparison is false. The `r` following the instruction name indicates that the operands to the instruction are reversed in the actual implementation. The comparison intrinsics for the Streaming SIMD Extensions 2 (SSE2) are listed in the following table followed by detailed descriptions.

The prototypes for SSE2 intrinsics are in the emmintrin.h header file.

Intrinsic Name	Corresponding Instruction	Compare For:
_mm_cmpeq_pd	CMPEQPD	Equality
_mm_cmplt_pd	CMPLTPD	Less Than
_mm_cmple_pd	CMPLEPD	Less Than or Equal
_mm_cmpgt_pd	CMPLTPDr	Greater Than
_mm_cmpge_pd	CMPLEPDr	Greater Than or Equal
_mm_cmpord_pd	CMPORDPD	Ordered
_mm_cmpunord_pd	CMPUNORDPD	Unordered
_mm_cmpneq_pd	CMPNEQPD	Inequality
_mm_cmpnlt_pd	CMPNLTPD	Not Less Than
_mm_cmpnle_pd	CMPNLEPD	Not Less Than or Equal
_mm_cmpngt_pd	CMPNLTPDr	Not Greater Than
_mm_cmpnge_pd	CMPLEPDr	Not Greater Than or Equal
_mm_cmpeq_sd	CMPEQSD	Equality
_mm_cmplt_sd	CMPLTSD	Less Than
_mm_cmple_sd	CMPLESD	Less Than or Equal
_mm_cmpgt_sd	CMPLTSDr	Greater Than
_mm_cmpge_sd	CMPLESDr	Greater Than or Equal
_mm_cmpord_sd	CMPORDSD	Ordered
_mm_cmpunord_sd	CMPUNORDSD	Unordered
_mm_cmpneq_sd	CMPNEQSD	Inequality
_mm_cmpnlt_sd	CMPNLTSD	Not Less Than
_mm_cmpnle_sd	CMPNLESD	Not Less Than or Equal
_mm_cmpngt_sd	CMPNLTSDr	Not Greater Than

Intrinsic Name	Corresponding Instruction	Compare For:
_mm_cmpnge_sd	CMPNLESDR	Not Greater Than or Equal
_mm_comieq_sd	COMISD	Equality
_mm_comilt_sd	COMISD	Less Than
_mm_comile_sd	COMISD	Less Than or Equal
_mm_comigt_sd	COMISD	Greater Than
_mm_comige_sd	COMISD	Greater Than or Equal
_mm_comineq_sd	COMISD	Not Equal
_mm_ucomieq_sd	UCOMISD	Equality
_mm_ucomilt_sd	UCOMISD	Less Than
_mm_ucomile_sd	UCOMISD	Less Than or Equal
_mm_ucomigt_sd	UCOMISD	Greater Than
_mm_ucomige_sd	UCOMISD	Greater Than or Equal
_mm_ucomineq_sd	UCOMISD	Not Equal

```
__m128d _mm_cmpeq_pd(__m128d a, __m128d b)
```

Compares the two DP FP values of a and b for equality.

```
r0 := (a0 == b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 == b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmplt_pd(__m128d a, __m128d b)
```

Compares the two DP FP values of a and b for a less than b.

```
r0 := (a0 < b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 < b1) ? 0xffffffffffffffff : 0x0
```

```
___m128d _mm_cmple_pd(__m128d a, __m128d b)
```

Compares the two DP FP values of a and b for a less than or equal to b.

```
r0 := (a0 <= b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 <= b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpgt_pd(__m128d a, __m128d b)
```

Compares the two DP FP values of a and b for a greater than b.

```
r0 := (a0 > b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 > b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpge_pd(__m128d a, __m128d b)
```

Compares the two DP FP values of a and b for a greater than or equal to b.

```
r0 := (a0 >= b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 >= b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpord_pd(__m128d a, __m128d b)
```

Compares the two DP FP values of a and b for ordered.

```
r0 := (a0 ord b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 ord b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpunord_pd(__m128d a, __m128d b)
```

Compares the two DP FP values of a and b for unordered.

```
r0 := (a0 unord b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 unord b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpneq_pd ( __m128d a, __m128d b)
```

Compares the two DP FP values of a and b for inequality.

```
r0 := (a0 != b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 != b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpnlt_pd(__m128d a, __m128d b)
```

Compares the two DP FP values of a and b for a not less than b.

```
r0 := !(a0 < b0) ? 0xffffffffffffffff : 0x0
r1 := !(a1 < b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpnle_pd(__m128d a, __m128d b)
```

Compares the two DP FP values of a and b for a not less than or equal to b.

```
r0 := !(a0 <= b0) ? 0xffffffffffffffff : 0x0
r1 := !(a1 <= b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpngt_pd(__m128d a, __m128d b)
```

Compares the two DP FP values of a and b for a not greater than b.

```
r0 := !(a0 > b0) ? 0xffffffffffffffff : 0x0
r1 := !(a1 > b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpnge_pd(__m128d a, __m128d b)
```

Compares the two DP FP values of a and b for a not greater than or equal to b.

```
r0 := !(a0 >= b0) ? 0xffffffffffffffff : 0x0
r1 := !(a1 >= b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpeq_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for equality. The upper DP FP value is passed through from a.

```
r0 := (a0 == b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmplt_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a less than b. The upper DP FP value is passed through from a.

```
r0 := (a0 < b0) ? 0xffffffffffffffff : 0x0
r1 := i1
```

```
__m128d _mm_cmple_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a less than or equal to b. The upper DP FP value is passed through from a.

```
r0 := (a0 <= b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpgt_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a greater than b. The upper DP FP value is passed through from a.

```
r0 := (a0 > b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpge_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a greater than or equal to b. The upper DP FP value is passed through from a.

```
r0 := (a0 >= b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpord_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for ordered. The upper DP FP value is passed through from a.

```
r0 := (a0 ord b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpunord_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for unordered. The upper DP FP value is passed through from a.

```
r0 := (a0 unord b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpneq_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for inequality. The upper DP FP value is passed through from a.

```
r0 := (a0 != b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpnlt_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a not less than b. The upper DP FP value is passed through from a.

```
r0 := !(a0 < b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpnle_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a not less than or equal to b. The upper DP FP value is passed through from a.

```
r0 := !(a0 <= b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpngt_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a not greater than b. The upper DP FP value is passed through from a.

```
r0 := !(a0 > b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpnge_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a not greater than or equal to b. The upper DP FP value is passed through from a.

```
r0 := !(a0 >= b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
int _mm_comieq_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.

```
r := (a0 == b0) ? 0x1 : 0x0
```

```
int _mm_comilt_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 < b0) ? 0x1 : 0x0
```

```
int _mm_comile_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 <= b0) ? 0x1 : 0x0
```

```
int _mm_comigt_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.

```
r := (a0 > b0) ? 0x1 : 0x0
```

```
int _mm_comige_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 >= b0) ? 0x1 : 0x0
```

```
int _mm_comineq_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.

```
r := (a0 != b0) ? 0x1 : 0x0
```

```
int _mm_ucomieq_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.

```
r := (a0 == b0) ? 0x1 : 0x0
```

```
int _mm_ucomilt_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 < b0) ? 0x1 : 0x0
```

```
int _mm_ucomile_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 <= b0) ? 0x1 : 0x0
```

```
int _mm_ucomigt_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.

```
r := (a0 > b0) ? 0x1 : 0x0
```

```
int _mm_ucomige_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.

```
r := (a0 >= b0) ? 0x1 : 0x0
```

```
int _mm_ucomineq_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.

```
r := (a0 != b0) ? 0x1 : 0x0
```

Floating-point Conversion Operations for Streaming SIMD Extensions 2

Each conversion intrinsic takes one data type and performs a conversion to a different type. Some conversions such as `_mm_cvtpd_ps` result in a loss of precision. The rounding mode used in such cases is determined by the value in the MXCSR register. The default rounding mode is round-to-nearest. Note that the rounding mode used by the C and C++ languages when performing a type conversion is to truncate. The `_mm_cvttpd_epi32` and `_mm_cvttsd_si32` intrinsics use the truncate rounding mode regardless of the mode specified by the `MXCSR` register.

The conversion-operation intrinsics for Streaming SIMD Extensions 2 (SSE2) are listed in the following table followed by detailed descriptions.

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

Intrinsic Name	Corresponding Instruction	Return Type	Parameters
`_mm_cvtpd_ps`	`CVTPD2PS`	`__m128`	`(__m128d a)`
`_mm_cvtps_pd`	`CVTPS2PD`	`__m128d`	`(__m128 a)`
`_mm_cvtepi32_pd`	`CVTDQ2PD`	`__m128d`	`(__m128i a)`
`_mm_cvtpd_epi32`	`CVTPD2DQ`	`__m128i`	`(__m128d a)`
`_mm_cvtsd_si32`	`CVTSD2SI`	`int`	`(__m128d a)`
`_mm_cvtsd_ss`	`CVTSD2SS`	`__m128`	`(__m128 a, __m128d b)`
`_mm_cvtsi32_sd`	`CVTSI2SD`	`__m128d`	`(__m128d a, int b)`
`_mm_cvtss_sd`	`CVTSS2SD`	`__m128d`	`(__m128d a, __m128 b)`
`_mm_cvttpd_epi32`	`CVTTPD2DQ`	`__m128i`	`(__m128d a)`
`_mm_cvttsd_si32`	`CVTTSD2SI`	`int`	`(__m128d a)`
`_mm_cvtpd_pi32`	`CVTPD2PI`	`__m64`	`(__m128d a)`
`_mm_cvttpd_pi32`	`CVTTPD2PI`	`__m64`	`(__m128d a)`
`_mm_cvtpi32_pd`	`CVTPI2PD`	`__m128d`	`(__m64 a)`

Intrinsic Name	Corresponding Instruction	Return Type	Parameters
_mm_cvtsd_f64	None	double	(__m128d a)

```
__m128 _mm_cvtpd_ps(__m128d a)
```

Converts the two DP FP values of a to SP FP values.

```
r0 := (float) a0
r1 := (float) a1
r2 := 0.0 ; r3 := 0.0
```

```
__m128d _mm_cvtps_pd(__m128 a)
```

Converts the lower two SP FP values of a to DP FP values.

```
r0 := (double) a0
r1 := (double) a1
```

```
__m128d _mm_cvtepi32_pd(__m128i a)
```

Converts the lower two signed 32-bit integer values of a to DP FP values.

```
r0 := (double) a0
r1 := (double) a1
```

```
__m128i _mm_cvtpd_epi32(__m128d a)
```

Converts the two DP FP values of a to 32-bit signed integer values.

```
r0 := (int) a0
r1 := (int) a1
r2 := 0x0 ; r3 := 0x0
```

```
int _mm_cvtsd_si32(__m128d a)
```

Converts the lower DP FP value of a to a 32-bit signed integer value.

```
r := (int) a0
```

```
__m128 _mm_cvtsd_ss(__m128 a, __m128d b)
```

Converts the lower DP FP value of b to an SP FP value. The upper SP FP values in a are passed through.

```
r0 := (float) b0
r1 := a1; r2 := a2 ; r3 := a3
```

```
__m128d _mm_cvtsi32_sd(__m128d a, int b)
```

Converts the signed integer value in b to a DP FP value. The upper DP FP value in a is passed through.

```
r0 := (double) b
r1 := a1
```

```
__m128d _mm_cvtss_sd(__m128d a, __m128 b)
```

Converts the lower SP FP value of b to a DP FP value. The upper value DP FP value in a is passed through.

```
r0 := (double) b0
r1 := a1
```

```
__m128i _mm_cvttpd_epi32(__m128d a)
```

Converts the two DP FP values of a to 32-bit signed integers using truncate.

```
r0 := (int) a0
r1 := (int) a1
r2 := 0x0 ; r3 := 0x0
```

```
int _mm_cvttsd_si32(__m128d a)
```

Converts the lower DP FP value of a to a 32-bit signed integer using truncate.

```
r := (int) a0
```

```
__m64 _mm_cvtpd_pi32(__m128d a)
```

Converts the two DP FP values of a to 32-bit signed integer values.

```
r0 := (int) a0
r1 := (int) a1
```

```
__m64 _mm_cvttpd_pi32(__m128d a)
```

Converts the two DP FP values of a to 32-bit signed integer values using truncate.

```
r0 := (int) a0
r1 := (int) a1
```

```
__m128d _mm_cvtpi32_pd(__m64 a)
```

Converts the two 32-bit signed integer values of a to DP FP values.

```
r0 := (double) a0
r1 := (double) a1
```

```
_mm_cvtsd_f64(__m128d a)
```

This intrinsic extracts a double precision floating point value from the first vector element of an `__m128d`. It does so in the most efficient manner possible in the context used. This intrinsic does not map to any specific SSE2 instruction.

Floating-point Memory and Initialization Operations for Streaming SIMD Extensions 2

This section describes the `load`, `set`, and `store` operations, which let you load and store data into memory. The `load` and `set` operations are similar in that both initialize `__m128d` data. However, the `set` operations take a double argument and are intended for initialization with constants, while the `load` operations take a double pointer argument and are intended to mimic the

instructions for loading data from memory. The `store` operation assigns the initialized data to the address.

Note

There is no intrinsic for move operations. To move data from one register to another, a simple assignment, `A = B`, suffices, where `A` and `B` are the source and target registers for the move operation.

The prototypes for Streaming SIMD Extensions 2 (SSE2) intrinsics are in the `emmintrin.h` header file.

Floating-point Load Operations for Streaming SIMD Extensions 2

The following `load` operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

```
__m128d _mm_load_pd(double const*dp)
```

(uses `MOVAPD`) Loads two DP FP values. The address p must be 16-byte aligned.

```
r0 := p[0]
r1 := p[1]
```

```
__m128d _mm_load1_pd(double const*dp)
```

(uses `MOVSD` + shuffling) Loads a single DP FP value, copying to both elements. The address p need not be 16-byte aligned.

```
r0 := *p
r1 := *p
```

```
__m128d _mm_loadr_pd(double const*dp)
```

(uses `MOVAPD` + shuffling) Loads two DP FP values in reverse order. The address p must be 16-byte aligned.

```
r0 := p[1]
r1 := p[0]
```

```
__m128d _mm_loadu_pd(double const*dp)
```

(uses `MOVUPD`) Loads two DP FP values. The address p need not be 16-byte aligned.

```
r0 := p[0]
r1 := p[1]
```

```
__m128d _mm_load_sd(double const*dp)
```

(uses MOVSD) Loads a DP FP value. The upper DP FP is set to zero. The address p need not be 16-byte aligned.

```
r0 := *p
r1 := 0.0
```

```
__m128d _mm_loadh_pd(__m128d a, double const*dp)
```

(uses MOVHPD) Loads a DP FP value as the upper DP FP value of the result. The lower DP FP value is passed through from a. The address p need not be 16-byte aligned.

```
r0 := a0
r1 := *p
```

```
__m128d _mm_loadl_pd(__m128d a, double const*dp)
```

(uses MOVLPD) Loads a DP FP value as the lower DP FP value of the result. The upper DP FP value is passed through from a. The address p need not be 16-byte aligned.

```
r0 := *p
r1 := a1
```

Floating-point Set Operations for Streaming SIMD Extensions 2

The following set operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

The prototypes for SSE2 intrinsics are in the emmintrin.h header file.

```
__m128d _mm_set_sd(double w)
```

(composite) Sets the lower DP FP value to w and sets the upper DP FP value to zero.

```
r0 := w
r1 := 0.0
```

```
__m128d _mm_set1_pd(double w)
```

(composite) Sets the 2 DP FP values to w.

```
r0 := w
r1 := w
```

```
__m128d _mm_set_pd(double w, double x)
```

(composite) Sets the lower DP FP value to x and sets the upper DP FP value to w.

```
r0 := x
r1 := w
```

```
__m128d _mm_setr_pd(double w, double x)
```

(composite) Sets the lower DP FP value to w and sets the upper DP FP value to x.

```
r0 := w
r1 := x
```

```
__m128d _mm_setzero_pd(void)
```

(uses XORPD) Sets the 2 DP FP values to zero.

```
r0 := 0.0
r1 := 0.0
```

```
__m128d _mm_move_sd( __m128d a, __m128d b)
```

(uses MOVSD) Sets the lower DP FP value to the lower DP FP value of b. The upper DP FP value is passed through from a.

```
r0 := b0
r1 := a1
```

Floating-point Store Operations for Streaming SIMD Extensions 2

The following `store` operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

```
void _mm_store_sd(double *dp, __m128d a)
```

(uses MOVSD) Stores the lower DP FP value of a. The address dp need not be 16-byte aligned.

```
*dp := a0
```

```
void _mm_store1_pd(double *dp, __m128d a)
```

(uses MOVAPD + shuffling) Stores the lower DP FP value of a twice. The address dp must be 16-byte aligned.

```
dp[0] := a0
dp[1] := a0
```

```
void _mm_store_pd(double *dp, __m128d a)
```

(uses MOVAPD) Stores two DP FP values. The address dp must be 16-byte aligned.

```
dp[0] := a0
dp[1] := a1
```

```
void _mm_storeu_pd(double *dp, __m128d a)
```

(uses MOVUPD) Stores two DP FP values. The address dp need not be 16-byte aligned.

```
dp[0] := a0
dp[1] := a1
```

```
void _mm_storer_pd(double *dp, __m128d a)
```

(uses MOVAPD + shuffling) Stores two DP FP values in reverse order. The address dp must be 16-byte aligned.

```
dp[0] := a1
dp[1] := a0
```

```
void _mm_storeh_pd(double *dp, __m128d a)
```

(uses MOVHPD) Stores the upper DP FP value of a.

```
*dp := a1
```

```
void _mm_storel_pd(double *dp, __m128d a)
```

(uses MOVLPD) Stores the lower DP FP value of a.

```
*dp := a0
```

Integer Arithmetic Operations for Streaming SIMD Extensions 2

The integer arithmetic operations for Streaming SIMD Extensions 2 (SSE2) are listed in the following table followed by their descriptions. The packed arithmetic intrinsics for SSE2 are listed in the Floating-point Arithmetic Operations topic.

The prototypes for SSE2 intrinsics are in the emmintrin.h header file.

Intrinsic	Instruction	Operation
_mm_add_epi8	PADDB	Addition
_mm_add_epi16	PADDW	Addition
_mm_add_epi32	PADDD	Addition
_mm_add_si64	PADDQ	Addition
_mm_add_epi64	PADDQ	Addition
_mm_adds_epi8	PADDSB	Addition
_mm_adds_epi16	PADDSW	Addition
_mm_adds_epu8	PADDUSB	Addition

Intrinsic	Instruction	Operation
_mm_adds_epu16	PADDUSW	Addition
_mm_avg_epu8	PAVGB	Computes Average
_mm_avg_epu16	PAVGW	Computes Average
_mm_madd_epi16	PMADDWD	Multiplication/Addition
_mm_max_epi16	PMAXSW	Computes Maxima
_mm_max_epu8	PMAXUB	Computes Maxima
_mm_min_epi16	PMINSW	Computes Minima
_mm_min_epu8	PMINUB	Computes Minima
_mm_mulhi_epi16	PMULHW	Multiplication
_mm_mulhi_epu16	PMULHUW	Multiplication
_mm_mullo_epi16	PMULLW	Multiplication
_mm_mul_su32	PMULUDQ	Multiplication
_mm_mul_epu32	PMULUDQ	Multiplication
_mm_sad_epu8	PSADBW	Computes Difference/Adds
_mm_sub_epi8	PSUBB	Subtraction
_mm_sub_epi16	PSUBW	Subtraction
_mm_sub_epi32	PSUBD	Subtraction
_mm_sub_si64	PSUBQ	Subtraction
_mm_sub_epi64	PSUBQ	Subtraction
_mm_subs_epi8	PSUBSB	Subtraction
_mm_subs_epi16	PSUBSW	Subtraction
_mm_subs_epu8	PSUBUSB	Subtraction
_mm_subs_epu16	PSUBUSW	Subtraction

```
__mm128i _mm_add_epi8(__m128i a, __m128i b)
```

Adds the 16 signed or unsigned 8-bit integers in a to the 16 signed or unsigned 8-bit integers in b.

```
r0 := a0 + b0
r1 := a1 + b1
...
r15 := a15 + b15
```

```
__mm128i _mm_add_epi16(__m128i a, __m128i b)
```

Adds the 8 signed or unsigned 16-bit integers in a to the 8 signed or unsigned 16-bit integers in b.

```
r0 := a0 + b0
r1 := a1 + b1
...
r7 := a7 + b7
```

```
__m128i _mm_add_epi32(__m128i a, __m128i b)
```

Adds the 4 signed or unsigned 32-bit integers in a to the 4 signed or unsigned 32-bit integers in b.

```
r0 := a0 + b0
r1 := a1 + b1
r2 := a2 + b2
r3 := a3 + b3
```

```
__m64 _mm_add_si64(__m64 a, __m64 b)
```

Adds the signed or unsigned 64-bit integer a to the signed or unsigned 64-bit integer b.

```
r := a + b
```

```
__m128i _mm_add_epi64(__m128i a, __m128i b)
```

Adds the 2 signed or unsigned 64-bit integers in a to the 2 signed or unsigned 64-bit integers in b.

```
r0 := a0 + b0
r1 := a1 + b1
```

```
__m128i _mm_adds_epi8(__m128i a, __m128i b)
```

Adds the 16 signed 8-bit integers in a to the 16 signed 8-bit integers in b using saturating arithmetic.

```
r0 := SignedSaturate(a0 + b0)
r1 := SignedSaturate(a1 + b1)
...
r15 := SignedSaturate(a15 + b15)
```

```
__m128i _mm_adds_epi16(__m128i a, __m128i b)
```

Adds the 8 signed 16-bit integers in a to the 8 signed 16-bit integers in b using saturating arithmetic.

```
r0 := SignedSaturate(a0 + b0)
r1 := SignedSaturate(a1 + b1)
...
r7 := SignedSaturate(a7 + b7)
```

```
__m128i _mm_adds_epu8(__m128i a, __m128i b)
```

Adds the 16 unsigned 8-bit integers in a to the 16 unsigned 8-bit integers in b using saturating arithmetic.

```
r0 := UnsignedSaturate(a0 + b0)
r1 := UnsignedSaturate(a1 + b1)
...
r15 := UnsignedSaturate(a15 + b15)
```

```
__m128i _mm_adds_epu16(__m128i a, __m128i b)
```

Adds the 8 unsigned 16-bit integers in a to the 8 unsigned 16-bit integers in b using saturating arithmetic.

```
r0 := UnsignedSaturate(a0 + b0)
r1 := UnsignedSaturate(a1 + b1)
...
r15 := UnsignedSaturate(a7 + b7)
```

```
__m128i _mm_avg_epu8(__m128i a, __m128i b)
```

Computes the average of the 16 unsigned 8-bit integers in a and the 16 unsigned 8-bit integers in b and rounds.

```
r0 := (a0 + b0) / 2
r1 := (a1 + b1) / 2
...
r15 := (a15 + b15) / 2
```

```
__m128i _mm_avg_epu16(__m128i a, __m128i b)
```

Computes the average of the 8 unsigned 16-bit integers in a and the 8 unsigned 16-bit integers in b and rounds.

```
r0 := (a0 + b0) / 2
r1 := (a1 + b1) / 2
...
r7 := (a7 + b7) / 2
```

```
__m128i _mm_madd_epi16(__m128i a, __m128i b)
```

Multiplies the 8 signed 16-bit integers from a by the 8 signed 16-bit integers from b. Adds the signed 32-bit integer results pairwise and packs the 4 signed 32-bit integer results.

```
r0 := (a0 * b0) + (a1 * b1)
r1 := (a2 * b2) + (a3 * b3)
r2 := (a4 * b4) + (a5 * b5)
r3 := (a6 * b6) + (a7 * b7)
```

```
__m128i _mm_max_epi16(__m128i a, __m128i b)
```

Computes the pairwise maxima of the 8 signed 16-bit integers from a and the 8 signed 16-bit integers from b.

```
r0 := max(a0, b0)
r1 := max(a1, b1)
...
r7 := max(a7, b7)
```

```
__m128i _mm_max_epu8(__m128i a, __m128i b)
```

Computes the pairwise maxima of the 16 unsigned 8-bit integers from a and the 16 unsigned 8-bit integers from b.

```
r0 := max(a0, b0)
r1 := max(a1, b1)
...
r15 := max(a15, b15)
```

```
__m128i _mm_min_epi16(__m128i a, __m128i b)
```

Computes the pairwise minima of the 8 signed 16-bit integers from a and the 8 signed 16-bit integers from b.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

```
__m128i _mm_min_epu8(__m128i a, __m128i b)
```

Computes the pairwise minima of the 16 unsigned 8-bit integers from a and the 16 unsigned 8-bit integers from b.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r15 := min(a15, b15)
```

```
__m128i _mm_mulhi_epi16(__m128i a, __m128i b)
```

Multiplies the 8 signed 16-bit integers from a by the 8 signed 16-bit integers from b. Packs the upper 16-bits of the 8 signed 32-bit results.

```
r0 := (a0 * b0)[31:16]
r1 := (a1 * b1)[31:16]
...
r7 := (a7 * b7)[31:16]
```

```
__m128i _mm_mulhi_epu16(__m128i a, __m128i b)
```

Multiplies the 8 unsigned 16-bit integers from a by the 8 unsigned 16-bit integers from b. Packs the upper 16-bits of the 8 unsigned 32-bit results.

```
r0 := (a0 * b0)[31:16]
r1 := (a1 * b1)[31:16]
...
r7 := (a7 * b7)[31:16]
```

```
__m128i_mm_mullo_epi16(__m128i a, __m128i b)
```

Multiplies the 8 signed or unsigned 16-bit integers from a by the 8 signed or unsigned 16-bit integers from b. Packs the lower 16-bits of the 8 signed or unsigned 32-bit results.

```
r0 := (a0 * b0)[15:0]
r1 := (a1 * b1)[15:0]
...
r7 := (a7 * b7)[15:0]
```

```
__m64 _mm_mul_su32(__m64 a, __m64 b)
```

Multiplies the lower 32-bit integer from a by the lower 32-bit integer from b, and returns the 64-bit integer result.

```
r := a0 * b0
```

```
__m128i _mm_mul_epu32(__m128i a, __m128i b)
```

Multiplies 2 unsigned 32-bit integers from a by 2 unsigned 32-bit integers from b. Packs the 2 unsigned 64-bit integer results.

```
r0 := a0 * b0
r1 := a2 * b2
```

```
__m128i _mm_sad_epu8(__m128i a, __m128i b)
```

Computes the absolute difference of the 16 unsigned 8-bit integers from a and the 16 unsigned 8-bit integers from b. Sums the upper 8 differences and lower 8 differences, and packs the resulting 2 unsigned 16-bit integers into the upper and lower 64-bit elements.

```
r0 := abs(a0 - b0) + abs(a1 - b1) +...+ abs(a7 - b7)
r1 := 0x0 ; r2 := 0x0 ; r3 := 0x0
r4 := abs(a8 - b8) + abs(a9 - b9) +...+ abs(a15 - b15)
r5 := 0x0 ; r6 := 0x0 ; r7 := 0x0
```

```
__m128i _mm_sub_epi8(__m128i a, __m128i b)
```

Subtracts the 16 signed or unsigned 8-bit integers of b from the 16 signed or unsigned 8-bit integers of a.

```
r0 := a0 - b0
r1 := a1 - b1
...
r15 := a15 - b15
```

```
__m128i_mm_sub_epi16(__m128i a, __m128i b)
```

Subtracts the 8 signed or unsigned 16-bit integers of b from the 8 signed or unsigned 16-bit integers of a.

```
r0 := a0 - b0
r1 := a1 - b1
...
r7 := a7 - b7
```

```
__m128i _mm_sub_epi32(__m128i a, __m128i b)
```

Subtracts the 4 signed or unsigned 32-bit integers of b from the 4 signed or unsigned 32-bit integers of a.

```
r0 := a0 - b0
r1 := a1 - b1
r2 := a2 - b2
r3 := a3 - b3
```

```
__m64 _mm_sub_si64 (__m64 a, __m64 b)
```

Subtracts the signed or unsigned 64-bit integer b from the signed or unsigned 64-bit integer a.

```
r := a - b
```

```
__m128i _mm_sub_epi64(__m128i a, __m128i b)
```

Subtracts the 2 signed or unsigned 64-bit integers in b from the 2 signed or unsigned 64-bit integers in a.

```
r0 := a0 - b0
r1 := a1 - b1
```

```
__m128i _mm_subs_epi8(__m128i a, __m128i b)
```

Subtracts the 16 signed 8-bit integers of b from the 16 signed 8-bit integers of a using saturating arithmetic.

```
r0 := SignedSaturate(a0 - b0)
r1 := SignedSaturate(a1 - b1)
...
r15 := SignedSaturate(a15 - b15)
```

```
__m128i _mm_subs_epi16(__m128i a, __m128i b)
```

Subtracts the 8 signed 16-bit integers of b from the 8 signed 16-bit integers of a using saturating arithmetic.

```
r0 := SignedSaturate(a0 - b0)
r1 := SignedSaturate(a1 - b1)
...
r7 := SignedSaturate(a7 - b7)
```

```
__m128i _mm_subs_epu8(__m128i a, __m128i b)
```

Subtracts the 16 unsigned 8-bit integers of b from the 16 unsigned 8-bit integers of a using saturating arithmetic.

```
r0 := UnsignedSaturate(a0 - b0)
r1 := UnsignedSaturate(a1 - b1)
...
r15 := UnsignedSaturate(a15 - b15)
```

```
__m128i _mm_subs_epu16(__m128i a, __m128i b)
```

Subtracts the 8 unsigned 16-bit integers of b from the 8 unsigned 16-bit integers of a using saturating arithmetic.

```
r0 := UnsignedSaturate(a0 - b0)
r1 := UnsignedSaturate(a1 - b1)
...
r7 := UnsignedSaturate(a7 - b7)
```

Integer Logical Operations for Streaming SIMD Extensions 2

The following four logical-operation intrinsics and their respective instructions are functional as part of Streaming SIMD Extensions 2 (SSE2).

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

```
__m128i _mm_and_si128(__m128i a, __m128i b)
```

(uses PAND) Computes the bitwise AND of the 128-bit value in a and the 128-bit value in b.

```
r := a & b
```

```
__m128i _mm_andnot_si128(__m128i a, __m128i b)
```

(uses PANDN) Computes the bitwise AND of the 128-bit value in b and the bitwise NOT of the 128-bit value in a.

```
r := (~a) & b
```

```
__m128i _mm_or_si128(__m128i a, __m128i b)
```

(uses POR) Computes the bitwise OR of the 128-bit value in a and the 128-bit value in b.

```
r := a | b
```

`__m128i _mm_xor_si128(__m128i a, __m128i b)`

(uses PXOR) Computes the bitwise XOR of the 128-bit value in a and the 128-bit value in b.
`r := a ^ b`

Integer Shift Operations for Streaming SIMD Extensions 2

The shift-operation intrinsics for Streaming SIMD Extensions 2 (SSE2) and the description for each are listed in the following table.

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

Intrinsic	Shift Direction	Shift Type	Corresponding Instruction
`_mm_slli_si128`	Left	Logical	PSLLDQ
`_mm_slli_epi16`	Left	Logical	PSLLW
`_mm_sll_epi16`	Left	Logical	PSLLW
`_mm_slli_epi32`	Left	Logical	PSLLD
`_mm_sll_epi32`	Left	Logical	PSLLD
`_mm_slli_epi64`	Left	Logical	PSLLQ
`_mm_sll_epi64`	Left	Logical	PSLLQ
`_mm_srai_epi16`	Right	Arithmetic	PSRAW
`_mm_sra_epi16`	Right	Arithmetic	PSRAW
`_mm_srai_epi32`	Right	Arithmetic	PSRAD
`_mm_sra_epi32`	Right	Arithmetic	PSRAD
`_mm_srli_si128`	Right	Logical	PSRLDQ
`_mm_srli_epi16`	Right	Logical	PSRLW
`_mm_srl_epi16`	Right	Logical	PSRLW
`_mm_srli_epi32`	Right	Logical	PSRLD
`_mm_srl_epi32`	Right	Logical	PSRLD
`_mm_srli_epi64`	Right	Logical	PSRLQ

Intrinsic	Shift Direction	Shift Type	Corresponding Instruction
_mm_srl_epi64	Right	Logical	PSRLQ

```
__m128i _mm_slli_si128(__m128i a, int imm)
```

Shifts the 128-bit value in a left by imm bytes while shifting in zeros. imm must be an immediate.

```
r := a << (imm * 8)
```

```
__m128i _mm_slli_epi16(__m128i a, int count)
```

Shifts the 8 signed or unsigned 16-bit integers in a left by count bits while shifting in zeros.

```
r0 := a0 << count
r1 := a1 << count
...
r7 := a7 << count
```

```
__m128i _mm_sll_epi16(__m128i a, __m128i count)
```

Shifts the 8 signed or unsigned 16-bit integers in a left by count bits while shifting in zeros.

```
r0 := a0 << count
r1 := a1 << count
...
r7 := a7 << count
```

```
__m128i _mm_slli_epi32(__m128i a, int count)
```

Shifts the 4 signed or unsigned 32-bit integers in a left by count bits while shifting in zeros.

```
r0 := a0 << count
r1 := a1 << count
r2 := a2 << count
r3 := a3 << count
```

```
__m128i _mm_sll_epi32(__m128i a, __m128i count)
```

Shifts the 4 signed or unsigned 32-bit integers in a left by count bits while shifting in zeros.

```
r0 := a0 << count
r1 := a1 << count
r2 := a2 << count
r3 := a3 << count
```

```
__m128i _mm_slli_epi64(__m128i a, int count)
```

Shifts the 2 signed or unsigned 64-bit integers in a left by `count` bits while shifting in zeros.

```
r0 := a0 << count
r1 := a1 << count
```

```
__m128i _mm_sll_epi64(__m128i a, __m128i count)
```

Shifts the 2 signed or unsigned 64-bit integers in a left by `count` bits while shifting in zeros.

```
r0 := a0 << count
r1 := a1 << count
```

```
__m128i _mm_srai_epi16(__m128i a, int count)
```

Shifts the 8 signed 16-bit integers in a right by `count` bits while shifting in the sign bit.

```
r0 := a0 >> count
r1 := a1 >> count
...
r7 := a7 >> count
```

```
__m128i _mm_sra_epi16(__m128i a, __m128i count)
```

Shifts the 8 signed 16-bit integers in a right by `count` bits while shifting in the sign bit.

```
r0 := a0 >> count
r1 := a1 >> count
...
r7 := a7 >> count
```

```
__m128i _mm_srai_epi32(__m128i a, int count)
```

Shifts the 4 signed 32-bit integers in a right by `count` bits while shifting in the sign bit.

```
r0 := a0 >> count
r1 := a1 >> count
r2 := a2 >> count
r3 := a3 >> count
```

```
__m128i _mm_sra_epi32(__m128i a, __m128i count)
```

Shifts the 4 signed 32-bit integers in a right by `count` bits while shifting in the sign bit.

```
r0 := a0 >> count
r1 := a1 >> count
r2 := a2 >> count
r3 := i3 >> count
```

```
__m128i _mm_srli_si128(__m128i a, int imm)
```

Shifts the 128-bit value in a right by `imm` bytes while shifting in zeros. `imm` must be an immediate.

```
r := srl(a, imm*8)
```

```
__m128i _mm_srli_epi16(__m128i a, int count)
```

Shifts the 8 signed or unsigned 16-bit integers in a right by `count` bits while shifting in zeros.

```
r0 := srl(a0, count)
r1 := srl(a1, count)
...
r7 := srl(a7, count)
```

```
__m128i _mm_srl_epi16(__m128i a, __m128i count)
```

Shifts the 8 signed or unsigned 16-bit integers in a right by `count` bits while shifting in zeros.

```
r0 := srl(a0, count)
r1 := srl(a1, count)
...
r7 := srl(a7, count)
```

```
__m128i _mm_srli_epi32(__m128i a, int count)
```

Shifts the 4 signed or unsigned 32-bit integers in a right by `count` bits while shifting in zeros.

```
r0 := srl(a0, count)
r1 := srl(a1, count)
r2 := srl(a2, count)
r3 := srl(a3, count)
```

```
__m128i _mm_srl_epi32(__m128i a, __m128i count)
```

Shifts the 4 signed or unsigned 32-bit integers in a right by `count` bits while shifting in zeros.

```
r0 := srl(a0, count)
r1 := srl(a1, count)
r2 := srl(a2, count)
r3 := srl(a3, count)
```

```
__m128i _mm_srli_epi64(__m128i a, int count)
```

Shifts the 2 signed or unsigned 64-bit integers in a right by `count` bits while shifting in zeros.

```
r0 := srl(a0, count)
r1 := srl(a1, count)
```

```
__m128i _mm_srl_epi64(__m128i a, __m128i count)
```

Shifts the 2 signed or unsigned 64-bit integers in a right by `count` bits while shifting in zeros.

```
r0 := srl(a0, count)
r1 := srl(a1, count)
```

Integer Comparison Operations for Streaming SIMD Extensions 2

The comparison intrinsics for Streaming SIMD Extensions 2 (SSE2) and descriptions for each are listed in the following table.

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

Intrinsic Name	Instruction	Comparison	Elements	Size of Elements
`_mm_cmpeq_epi8`	`PCMPEQB`	Equality	16	8
`_mm_cmpeq_epi16`	`PCMPEQW`	Equality	8	16
`_mm_cmpeq_epi32`	`PCMPEQD`	Equality	4	32
`_mm_cmpgt_epi8`	`PCMPGTB`	Greater Than	16	8
`_mm_cmpgt_epi16`	`PCMPGTW`	Greater Than	8	16
`_mm_cmpgt_epi32`	`PCMPGTD`	Greater Than	4	32
`_mm_cmplt_epi8`	`PCMPGTBr`	Less Than	16	8
`_mm_cmplt_epi16`	`PCMPGTWr`	Less Than	8	16
`_mm_cmplt_epi32`	`PCMPGTDr`	Less Than	4	32

```
__m128i _mm_cmpeq_epi8(__m128i a, __m128i b)
```

Compares the 16 signed or unsigned 8-bit integers in a and the 16 signed or unsigned 8-bit integers in b for equality.

```
r0 := (a0 == b0) ? 0xff : 0x0
r1 := (a1 == b1) ? 0xff : 0x0
...
r15 := (a15 == b15) ? 0xff : 0x0
```

```
__m128i _mm_cmpeq_epi16(__m128i a, __m128i b)
```

Compares the 8 signed or unsigned 16-bit integers in a and the 8 signed or unsigned 16-bit integers in b for equality.

```
r0 := (a0 == b0) ? 0xffff : 0x0
r1 := (a1 == b1) ? 0xffff : 0x0
...
r7 := (a7 == b7) ? 0xffff : 0x0
```

```
__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)
```

Compares the 4 signed or unsigned 32-bit integers in a and the 4 signed or unsigned 32-bit integers in b for equality.

```
r0 := (a0 == b0) ? 0xffffffff : 0x0
r1 := (a1 == b1) ? 0xffffffff : 0x0
r2 := (a2 == b2) ? 0xffffffff : 0x0
r3 := (a3 == b3) ? 0xffffffff : 0x0
```

```
__m128i _mm_cmpgt_epi8(__m128i a, __m128i b)
```

Compares the 16 signed 8-bit integers in a and the 16 signed 8-bit integers in b for greater than.

```
r0 := (a0 > b0) ? 0xff : 0x0
r1 := (a1 > b1) ? 0xff : 0x0
...
r15 := (a15 > b15) ? 0xff : 0x0
```

```
__m128i _mm_cmpgt_epi16(__m128i a, __m128i b)
```

Compares the 8 signed 16-bit integers in a and the 8 signed 16-bit integers in b for greater than.

```
r0 := (a0 > b0) ? 0xffff : 0x0
r1 := (a1 > b1) ? 0xffff : 0x0
...
r7 := (a7 > b7) ? 0xffff : 0x0
```

```
__m128i _mm_cmpgt_epi32(__m128i a, __m128i b)
```

Compares the 4 signed 32-bit integers in a and the 4 signed 32-bit integers in b for greater than.

```
r0 := (a0 > b0) ? 0xffff : 0x0
r1 := (a1 > b1) ? 0xffff : 0x0
r2 := (a2 > b2) ? 0xffff : 0x0
r3 := (a3 > b3) ? 0xffff : 0x0
```

```
__m128i _mm_cmplt_epi8( __m128i a, __m128i b)
```

Compares the 16 signed 8-bit integers in a and the 16 signed 8-bit integers in b for less than.

```
r0 := (a0 < b0) ? 0xff : 0x0
r1 := (a1 < b1) ? 0xff : 0x0
...
r15 := (a15 < b15) ? 0xff : 0x0
```

```
__m128i _mm_cmplt_epi16( __m128i a, __m128i b)
```

Compares the 8 signed 16-bit integers in a and the 8 signed 16-bit integers in b for less than.

```
r0 := (a0 < b0) ? 0xffff : 0x0
r1 := (a1 < b1) ? 0xffff : 0x0
...
r7 := (a7 < b7) ? 0xffff : 0x0
```

```
__m128i _mm_cmplt_epi32( __m128i a, __m128i b)
```

Compares the 4 signed 32-bit integers in a and the 4 signed 32-bit integers in b for less than.

```
r0 := (a0 < b0) ? 0xffff : 0x0
r1 := (a1 < b1) ? 0xffff : 0x0
r2 := (a2 < b2) ? 0xffff : 0x0
r3 := (a3 < b3) ? 0xffff : 0x0
```

Integer Conversions Operations for Streaming SIMD Extensions 2

The following two conversion intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

```
__m128i _mm_cvtsi32_si128(int a)
```

(uses `MOVD`) Moves 32-bit integer a to the least significant 32 bits of an `__m128i` object. Copies the sign bit of a into the upper 96 bits of the `__m128i` object.

```
r0 := a
r1 := 0x0 ; r2 := 0x0 ; r3 := 0x0
```

```
__m128i _mm_cvtsi64_si128(__int64 a)
```

(uses `MOVQ`) Moves 64-bit integer a to the lower 64 bits of an `__m128i` object, zeroing the upper bits.

```
r0 := a
r1 := 0x0
```

```
int _mm_cvtsi128_si32(__m128i a)
```

(uses MOVD) Moves the least significant 32 bits of a to a 32-bit integer.

```
r := a0
```

```
__int64 _mm_cvtsi128_si64(__m128i a)
```

(uses MOVQ) Moves the lower 64 bits of a to a 64-bit integer.

```
r := a0
```

```
__m128d _mm_cvtsi64_sd(__m128d a, __int64 b)
```

(uses CVTSI2SD) Converts the signed 64-bit integer value in b to a DP FP value. The upper DP FP value in a is passed through.

```
r0 := (double)b
```

```
r1 := a1
```

```
__int64 _mm_cvtsd_si64(__m128d a)
```

(uses CVTSD2SI) Converts the lower DP FP value of a to a 64-bit signed integer value according to the current rounding mode.

```
r := (__int64) a0
```

```
__int64 _mm_cvttsd_si64(__m128d a)
```

(uses CVTTSD2SI) Converts the lower DP FP value of a to a 64-bit signed integer value using truncation.

```
r := (__int64) a0
```

```
__m128 _mm_cvtepi32_ps(__m128i a)
```

Converts the 4 signed 32-bit integer values of a to SP FP values.

```
r0 := (float) a0
r1 := (float) a1
r2 := (float) a2
r3 := (float) a3
```

```
__m128i _mm_cvtps_epi32(__m128 a)
```

Converts the 4 SP FP values of a to signed 32-bit integer values.

```
r0 := (int) a0
r1 := (int) a1
r2 := (int) a2
r3 := (int) a3
```

```
__m128i _mm_cvttps_epi32(__m128 a)
```

Converts the 4 SP FP values of a to signed 32 bit integer values using truncate.

```
r0 := (int) a0
r1 := (int) a1
r2 := (int) a2
r3 := (int) a3
```

Integer Memory and Initialization Operations for Streaming SIMD Extensions 2

The integer `load`, `set`, and `store` intrinsics and their respective instructions provide memory and initialization operations for the Streaming SIMD Extensions 2 (SSE2).

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

- Load Operations
- Set Operations
- Store Operations

Integer Load Operations for Streaming SIMD Extensions 2

The following `load` operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

```
__m128i _mm_load_si128(__m128i const*p)
```

(uses `MOVDQA`) Loads 128-bit value. Address `p` must be 16-byte aligned.

```
r := *p
```

```
__m128i _mm_loadu_si128(__m128i const*p)
```

(uses `MOVDQU`) Loads 128-bit value. Address `p` not need be 16-byte aligned.

```
r := *p
```

```
__m128i _mm_loadl_epi64(__m128i const*p)
```

(uses `MOVQ`) Load the lower 64 bits of the value pointed to by `p` into the lower 64 bits of the result, zeroing the upper 64 bits of the result.

```
r0:= *p[63:0]
r1:=0x0
```

Integer Set Operations for SSE2

The following `set` operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

```
__m128i _mm_set_epi64(__m64 q1, __m64 q0)
```

Sets the 2 64-bit integer values.

```
r0 := q0
r1 := q1
```

```
__m128i _mm_set_epi32(int i3, int i2, int i1, int i0)
```

Sets the 4 signed 32-bit integer values.

```
r0 := i0
r1 := i1
r2 := i2
r3 := i3
```

```
__m128i _mm_set_epi16(short w7, short w6, short w5, short
w4, short w3, short w2, short w1, short w0)
```

Sets the 8 signed 16-bit integer values.

```
r0 := w0
r1 := w1
...
r7 := w7
```

```
__m128i _mm_set_epi8(char b15, char b14, char b13, char
b12, char b11, char b10, char b9, char b8, char b7, char
b6, char b5, char b4, char b3, char b2, char b1, char b0)
```

Sets the 16 signed 8-bit integer values.

```
r0 := b0
r1 := b1
...
r15 := b15
```

```
__m128i _mm_set1_epi64(__m64 q)
```

Sets the 2 64-bit integer values to `q`.

```
r0 := q
r1 := q
```

```
__m128i _mm_set1_epi32(int i)
```

Sets the 4 signed 32-bit integer values to `i`.

```
r0 := i
r1 := i
r2 := i
r3 := i
```

```
__m128i _mm_set1_epi16(short w)
```

Sets the 8 signed 16-bit integer values to w.

```
r0 := w
r1 := w
...
r7 := w
```

```
__m128i _mm_set1_epi8(char b)
```

Sets the 16 signed 8-bit integer values to b.

```
r0 := b
r1 := b
...
r15 := b
```

```
__m128i _mm_setr_epi64(__m64 q0, __m64 q1)
```

Sets the 2 64-bit integer values in reverse order.

```
r0 := q0
r1 := q1
```

```
__m128i _mm_setr_epi32(int i0, int i1, int i2, int i3)
```

Sets the 4 signed 32-bit integer values in reverse order.

```
r0 := i0
r1 := i1
r2 := i2
r3 := i3
```

```
__m128i _mm_setr_epi16(short w0, short w1, short w2, short
w3, short w4, short w5, short w6, short w7)
```

Sets the 8 signed 16-bit integer values in reverse order.

```
r0 := w0
r1 := w1
...
r7 := w7
```

```
__m128i _mm_setr_epi8(char b15, char b14, char b13, char
b12, char b11, char b10, char b9, char b8, char b7, char
b6, char b5, char b4, char b3, char b2, char b1, char b0)
```

Sets the 16 signed 8-bit integer values in reverse order.

```
r0 := b0
r1 := b1
...
r15 := b15
```

```
__m128i _mm_setzero_si128()
```

Sets the 128-bit value to zero.

```
r := 0x0
```

Integer Store Operations for Streaming SIMD Extensions 2

The following `store` operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

```
void _mm_store_si128(__m128i *p, __m128i b)
```

(uses `MOVDQA`) Stores 128-bit value. Address `p` must be 16 byte aligned.

```
*p := a
```

```
void _mm_storeu_si128(__m128i *p, __m128i b)
```

(uses `MOVDQU`) Stores 128-bit value. Address `p` need not be 16-byte aligned.

```
*p := a
```

```
void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p)
```

(uses `MASKMOVDQU`) Conditionally store byte elements of `d` to address `p`. The high bit of each byte in the selector `n` determines whether the corresponding byte in `d` will be stored. Address `p` need not be 16-byte aligned.

```
if (n0[7]) p[0] := d0
if (n1[7]) p[1] := d1
...
if (n15[7]) p[15] := d15
```

```
void _mm_storel_epi64(__m128i *p, __m128i q)
```

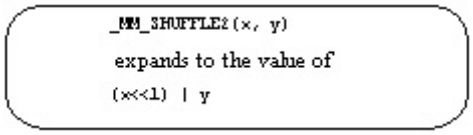
(uses `MOVQ`) Stores the lower 64 bits of the value pointed to by `p`.

```
*p[63:0]:=a0
```

Macro Function for Shuffle

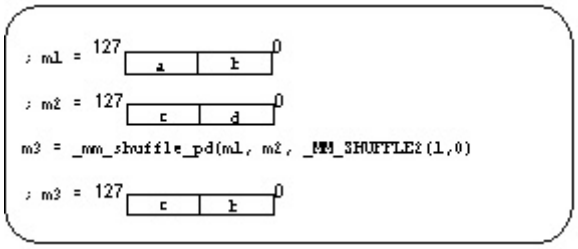
The Streaming SIMD Extensions 2 (SSE2) provide a macro function to help create constants that describe shuffle operations. The macro takes two small integers (in the range of 0 to 1) and combines them into an 2-bit immediate value used by the `SHUFPD` instruction. See the following example.

Shuffle Function Macro



You can view the two integers as selectors for choosing which two words from the first input operand and which two words from the second are to be put into the result word.

View of Original and Result Words with Shuffle Function Macro



Cacheability Support Operations for Streaming SIMD Extensions 2

The prototypes for Streaming SIMD Extensions 2 (SSE2) intrinsics are in the `emmintrin.h` header file.

```
void _mm_stream_pd(double *p, __m128d a)
```

(uses `MOVNTPD`) Stores the data in a to the address p without polluting caches. The address p must be 16-byte aligned. If the cache line containing address p is already in the cache, the cache will be updated.

```
p[0] := a0
p[1] := a1
```

```
void _mm_stream_si128(__m128i *p, __m128i a)
```

Stores the data in a to the address p without polluting the caches. If the cache line containing address p is already in the cache, the cache will be updated. Address p must be 16-byte aligned.

```
*p := a
```

```
void _mm_stream_si32(int *p, int a)
```

Stores the data in a to the address p without polluting the caches. If the cache line containing address p is already in the cache, the cache will be updated.

```
*p := a
```

```
void _mm_clflush(void const*p)
```

Cache line containing p is flushed and invalidated from all caches in the coherency domain.

```
void _mm_lfence(void)
```

Guarantees that every load instruction that precedes, in program order, the load fence instruction is globally visible before any load instruction which follows the fence in program order.

`void _mm_mfence(void)`

Guarantees that every memory access that precedes, in program order, the memory fence instruction is globally visible before any memory instruction which follows the fence in program order.

`void _mm_pause(void)`

The execution of the next instruction is delayed an implementation specific amount of time. The instruction does not modify the architectural state. This intrinsic provides especially significant performance gain.

PAUSE Intrinsic

The PAUSE intrinsic is used in spin-wait loops with the processors implementing dynamic execution (especially out-of-order execution). In the spin-wait loop, PAUSE improves the speed at which the code detects the release of the lock. For dynamic scheduling, the PAUSE instruction reduces the penalty of exiting from the spin-loop.

Example of loop with the PAUSE instruction:

```
spin_loop:pause
cmp eax, A
jne spin_loop
```

In this example, the program spins until memory location A matches the value in register `eax`. The code sequence that follows shows a test-and-test-and-set. In this example, the spin occurs only after the attempt to get a lock has failed.

```
get_lock: mov eax, 1
xchg eax, A ; Try to get lock
cmp eax, 0 ; Test if successful
jne spin_loop
critical_section code
mov A, 0 ; Release lock
jmp continue
spin_loop: pause ; Spin-loop hint
cmp 0, A ; Check lock availability
jne spin_loop
jmp get_lock
continue:
```

Note that the first branch is predicted to fall-through to the critical section in anticipation of successfully gaining access to the lock. It is highly recommended that all spin-wait loops include the PAUSE instruction. Since PAUSE is backwards compatible to all existing IA-32 processor generations, a test for processor type (a CPUID test) is not needed. All legacy processors will execute PAUSE as a NOP,

but in processors which use the PAUSE as a hint there can be significant performance benefit.

Miscellaneous Operations for Streaming SIMD Extensions 2

The miscellaneous intrinsics for Streaming SIMD Extensions 2 (SSE2) are listed in the following table followed by their descriptions.

The prototypes for SSE2 intrinsics are in the `emmintrin.h` header file.

Intrinsic	Corresponding Instruction	Operation
_mm_packs_epi16	PACKSSWB	Packed Saturation
_mm_packs_epi32	PACKSSDW	Packed Saturation
_mm_packus_epi16	PACKUSWB	Packed Saturation
_mm_extract_epi16	PEXTRW	Extraction
_mm_insert_epi16	PINSRW	Insertion
_mm_movemask_epi8	PMOVMSKB	Mask Creation
_mm_shuffle_epi32	PSHUFD	Shuffle
_mm_shufflehi_epi16	PSHUFHW	Shuffle
_mm_shufflelo_epi16	PSHUFLW	Shuffle
_mm_unpackhi_epi8	PUNPCKHBW	Interleave
_mm_unpackhi_epi16	PUNPCKHWD	Interleave
_mm_unpackhi_epi32	PUNPCKHDQ	Interleave
_mm_unpackhi_epi64	PUNPCKHQDQ	Interleave
_mm_unpacklo_epi8	PUNPCKLBW	Interleave
_mm_unpacklo_epi16	PUNPCKLWD	Interleave
_mm_unpacklo_epi32	PUNPCKLDQ	Interleave
_mm_unpacklo_epi64	PUNPCKLQDQ	Interleave
_mm_movepi64_pi64	MOVDQ2Q	move

Intrinsic	Corresponding Instruction	Operation
_m128i_mm_movpi64_epi64	MOVQ2DQ	move
_mm_move_epi64	MOVQ	move

```
__m128i _mm_packs_epi16(__m128i a, __m128i b)
```

Packs the 16 signed 16-bit integers from a and b into 8-bit integers and saturates.

```
r0 := SignedSaturate(a0)
r1 := SignedSaturate(a1)
...
r7 := SignedSaturate(a7)
r8 := SignedSaturate(b0)
r9 := SignedSaturate(b1)
...
r15 := SignedSaturate(b7)
```

```
__m128i _mm_packs_epi32(__m128i a, __m128i b)
```

Packs the 8 signed 32-bit integers from a and b into signed 16-bit integers and saturates.

```
r0 := SignedSaturate(a0)
r1 := SignedSaturate(a1)
r2 := SignedSaturate(a2)
r3 := SignedSaturate(a3)
r4 := SignedSaturate(b0)
r5 := SignedSaturate(b1)
r6 := SignedSaturate(b2)
r7 := SignedSaturate(b3)
```

```
__m128i _mm_packus_epi16(__m128i a, __m128i b)
```

Packs the 16 signed 16-bit integers from a and b into 8-bit unsigned integers and saturates.

```
r0 := UnsignedSaturate(a0)
r1 := UnsignedSaturate(a1)
...
r7 := UnsignedSaturate(a7)
r8 := UnsignedSaturate(b0)
r9 := UnsignedSaturate(b1)
...
r15 := UnsignedSaturate(b7)
```

```
int _mm_extract_epi16(__m128i a, int imm)
```

Extracts the selected signed or unsigned 16-bit integer from a and zero extends. The selector `imm` must be an immediate.

```
r := (imm == 0) ? a0 :
( (imm == 1) ? a1 :
...
(imm == 7) ? a7 )
```

```
__m128i _mm_insert_epi16(__m128i a, int b, int imm)
```

Inserts the least significant 16 bits of b into the selected 16-bit integer of a. The selector `imm` must be an immediate.

```
r0 := (imm == 0) ? b : a0;
r1 := (imm == 1) ? b : a1;
...
r7 := (imm == 7) ? b : a7;
```

```
int _mm_movemask_epi8(__m128i a)
```

Creates a 16-bit mask from the most significant bits of the 16 signed or unsigned 8-bit integers in a and zero extends the upper bits.

```
r := a15[7] << 15 |
a14[7] << 14 |
...
a1[7] << 1 |
a0[7]
```

```
__m128i _mm_shuffle_epi32(__m128i a, int imm)
```

Shuffles the 4 signed or unsigned 32-bit integers in a as specified by `imm`. The shuffle value, `imm`, must be an immediate. See Macro Function for Shuffle for a description of shuffle semantics.

```
__m128i _mm_shufflehi_epi16(__m128i a, int imm)
```

Shuffles the upper 4 signed or unsigned 16-bit integers in a as specified by `imm`. The shuffle value, `imm`, must be an immediate. See Macro Function for Shuffle for a description of shuffle semantics.

```
__m128i _mm_shufflelo_epi16(__m128i a, int imm)
```

Shuffles the lower 4 signed or unsigned 16-bit integers in a as specified by `imm`. The shuffle value, `imm`, must be an immediate. See Macro Function for Shuffle for a description of shuffle semantics.

```
__m128i _mm_unpackhi_epi8(__m128i a, __m128i b)
```

Interleaves the upper 8 signed or unsigned 8-bit integers in a with the upper 8 signed or unsigned 8-bit integers in b.

```
r0 := a8 ; r1 := b8
r2 := a9 ; r3 := b9
...
r14 := a15 ; r15 := b15
```

```
__m128i _mm_unpackhi_epi16(__m128i a, __m128i b)
```

Interleaves the upper 4 signed or unsigned 16-bit integers in a with the upper 4 signed or unsigned 16-bit integers in b.

```
r0 := a4 ; r1 := b4
r2 := a5 ; r3 := b5
r4 := a6 ; r5 := b6
r6 := a7 ; r7 := b7
```

```
__m128i _mm_unpackhi_epi32(__m128i a, __m128i b)
```

Interleaves the upper 2 signed or unsigned 32-bit integers in a with the upper 2 signed or unsigned 32-bit integers in b.

```
r0 := a2 ; r1 := b2
r2 := a3 ; r3 := b3
```

```
__m128i _mm_unpackhi_epi64(__m128i a, __m128i b)
```

Interleaves the upper signed or unsigned 64-bit integer in a with the upper signed or unsigned 64-bit integer in b.

```
r0 := a1 ; r1 := b1
```

```
__m128i _mm_unpacklo_epi8(__m128i a, __m128i b)
```

Interleaves the lower 8 signed or unsigned 8-bit integers in a with the lower 8 signed or unsigned 8-bit integers in b.

```
r0 := a0 ; r1 := b0
r2 := a1 ; r3 := b1
...
r14 := a7 ; r15 := b7
```

```
__m128i _mm_unpacklo_epi16(__m128i a, __m128i b)
```

Interleaves the lower 4 signed or unsigned 16-bit integers in a with the lower 4 signed or unsigned 16-bit integers in b.

```
r0 := a0 ; r1 := b0
r2 := a1 ; r3 := b1
r4 := a2 ; r5 := b2
r6 := a3 ; r7 := b3
```

```
__m128i _mm_unpacklo_epi32(__m128i a, __m128i b)
```

Interleaves the lower 2 signed or unsigned 32-bit integers in a with the lower 2 signed or unsigned 32-bit integers in b.

```
r0 := a0 ; r1 := b0
r2 := a1 ; r3 := b1
```

```
__m128i _mm_unpacklo_epi64(__m128i a, __m128i b)
```

Interleaves the lower signed or unsigned 64-bit integer in a with the lower signed or unsigned 64-bit integer in b.

```
r0 := a0 ; r1 := b0
```

```
__m64 _mm_movepi64_pi64(__m128i a)
```

Returns the lower 64 bits of a as an __m64 type.

```
r0 := a0 ;
```

```
__128i _mm_movpi64_pi64(__m64 a)
```

Moves the 64 bits of a to the lower 64 bits of the result, zeroing the upper bits.

```
r0 := a0 ; r1 := 0X0 ;
```

```
__128i _mm_move_epi64(__128i a)
```

Moves the lower 64 bits of a to the lower 64 bits of the result, zeroing the upper bits.

```
r0 := a0 ; r1 := 0X0 ;
```

Additional Miscellaneous Intrinsics

The prototypes for Streaming SIMD Extensions 2 (SSE2) intrinsics are in the emmintrin.h header file.

```
__m128d _mm_unpackhi_pd(__m128d a, __m128d b)
```

(uses UNPCKHPD) Interleaves the upper DP FP values of a and b.

```
r0 := a1
r1 := b1
```

```
__m128d _mm_unpacklo_pd(__m128d a, __m128d b)
```

(uses UNPCKLPD) Interleaves the lower DP FP values of a and b.

```
r0 := a0
r1 := b0
```

```
int _mm_movemask_pd(__m128d a)
```

(uses MOVMSKPD) Creates a two-bit mask from the sign bits of the two DP FP values of a.

```
r := sign(a1) << 1 | sign(a0)
```

```
__m128d _mm_shuffle_pd(__m128d a, __m128d b, int i)
```

(uses SHUFPD) Selects two specific DP FP values from a and b, based on the mask `i`. The mask must be an immediate. See Macro Function for Shuffle for a description of the shuffle semantics.

Intrinsics for Casting Support

This version of the Intel C++ Compiler supports casting between various SP, DP, and INT vector types. These intrinsics do not convert values; they just change the type.

```
extern __m128  _mm_castpd_ps(__m128d in);
extern __m128i _mm_castpd_si128(__m128d in);
extern __m128d _mm_castps_pd(__m128 in);
extern __m128i _mm_castps_si128(__m128 in);
extern __m128  _mm_castsi128_ps(__m128i in);
extern __m128d _mm_castsi128_pd(__m128i in);
```

Streaming SIMD Extensions 3

The Intel® C++ intrinsics listed in this section are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3). They will not function correctly on other IA-32 processors. New SSE3 intrinsics include:

- Floating-point Vector Intrinsics
- Integer Vector Intrinsics
- Miscellaneous Intrinsics
- Macro Functions

The prototypes for these intrinsics are in the `pmmintrin.h` header file.

Note

You can also use the single `ia32intrin.h` header file for any IA-32 intrinsics.

Floating-point Vector Intrinsics for Streaming SIMD Extensions 3

The floating-point intrinsics listed here are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3).

The prototypes for these intrinsics are in the `pmmintrin.h` header file.

Single-precision Floating-point Vector Intrinsics

```
extern __m128 _mm_addsub_ps(__m128 a, __m128 b);
```

Subtracts even vector elements while adding odd vector elements.

```
r0 := a0 - b0;
r1 := a1 + b1;
r2 := a2 - b2;
r3 := a3 + b3;
```

```
extern __m128 _mm_hadd_ps(__m128 a, __m128 b);
```

Adds adjacent vector elements.

```
r0 := a0 + a1;
r1 := a2 + a3;
r2 := b0 + b1;
r3 := b2 + b3;
```

```
extern __m128 _mm_hsub_ps(__m128 a, __m128 b);
```

Subtracts adjacent vector elements.

```
r0 := a0 - a1;
r1 := a2 - a3;
r2 := b0 - b1;
r3 := b2 - b3;
```

```
extern __m128 _mm_movehdup_ps(__m128 a);
```

Duplicates odd vector elements into even vector elements.

```
r0 := a1;
r1 := a1;
r2 := a3;
r3 := a3;
```

```
extern __m128 _mm_moveldup_ps(__m128 a);
```

Duplicates even vector elements into odd vector elements.

```
r0 := a0;
r1 := a0;
r2 := a2;
r3 := a2;
```

Double-precision Floating-point Vector Intrinsics

```
extern __m128d _mm_addsub_pd(__m128d a, __m128d b);
```

Adds upper vector element while subtracting lower vector element.

```
r0 := a0 - b0;
r1 := a1 + b1;
```

```
extern __m128d _mm_hadd_pd(__m128d a, __m128d b);
```

Adds adjacent vector elements.

```
r0 := a0 + a1;
r1 := b0 + b1;
```

```
extern __m128d _mm_hsub_pd(__m128d a, __m128d b);
```

Subtracts adjacent vector elements.

```
r0 := a0 - a1;
r1 := b0 - b1;
```

```
extern __m128d _mm_loaddup_pd(double const * dp);
```

Duplicates a double value into upper and lower vector elements.

```
r0 := *dp;
r1 := *dp;
```

```
extern __m128d _mm_movedup_pd(__m128d a);
```

Duplicates lower vector element into upper vector element.

```
r0 := a0;
r1 := a0;
```

Integer Vector Intrinsics for Streaming SIMD Extensions 3

The integer vector intrinsic listed here is designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3).

The prototypes for these intrinsics are in the `pmmintrin.h` header file.

```
extern __m128i _mm_lddqu_si128(__m128i const *p);
```

Loads an unaligned 128-bit value. This differs from `movdqu` in that it can provide higher performance in some cases. However, it also may provide lower performance than `movdqu` if the memory value being read was just previously written.

```
r := *p;
```

Macro Functions for Streaming SIMD Extensions 3

The macro function intrinsics listed here are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3).

The prototypes for these intrinsics are in the `pmmintrin.h` header file.

```
_MM_SET_DENORMALS_ZERO_MODE(x)
```

Macro arguments: one of `__MM_DENORMALS_ZERO_ON`, `_MM_DENORMALS_ZERO_OFF`
This causes "`denormals are zero`" mode to be turned on or off by setting the appropriate bit of the control register.

`_MM_GET_DENORMALS_ZERO_MODE()`

No arguments. This returns the current value of the denormals are zero mode bit of the control register.

Miscellaneous Intrinsics for Streaming SIMD Extensions 3

The miscellaneous intrinsics listed here are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3).

The prototypes for these intrinsics are in the `pmmintrin.h` header file.

```
extern void _mm_monitor(void const *p, unsigned extensions,
unsigned hints);
```

Generates the `MONITOR` instruction. This sets up an address range for the monitor hardware using `p` to provide the logical address, and will be passed to the monitor instruction in register `eax`. The extensions parameter contains optional extensions to the monitor hardware which will be passed in `ecx`. The hints parameter will contain hints to the monitor hardware, which will be passed in `edx`. A non-zero value for extensions will cause a general protection fault.

```
extern void _mm_mwait(unsigned extensions, unsigned hints);
```

Generates the `MWAIT` instruction. This instruction is a hint that allows the processor to stop execution and enter an implementation-dependent optimized state until occurrence of a class of events. In future processor designs extensions and hints parameters may be used to convey additional information to the processor. All non-zero values of extensions and hints are reserved. A non-zero value for extensions will cause a general protection fault.

Intrinsics for Itanium® Instructions

This section lists and describes the native intrinsics for Itanium® instructions. These intrinsics cannot be used on the IA-32 architecture. The intrinsics for Itanium instructions give programmers access to Itanium instructions that cannot be generated using the standard constructs of the C and C++ languages.

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Native Intrinsics for Itanium® Instructions

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Integer Operations

Intrinsic	Corresponding Instruction
`__int64 _m64_dep_mr(__int64 r, __int64 s, const int pos, const int len)`	`dep` (Deposit)
`__int64 _m64_dep_mi(const int v, __int64 s, const int p, const int len)`	`dep` (Deposit)
`__int64 _m64_dep_zr(__int64 s, const int pos, const int len)`	`dep.z` (Deposit)
`__int64 _m64_dep_zi(const int v, const int pos, const int len)`	`dep.z` (Deposit)
`__int64 _m64_extr(__int64 r, const int pos, const int len)`	`extr` (Extract)
`__int64 _m64_extru(__int64 r, const int pos, const int len)`	`extr.u` (Extract)
`__int64 _m64_xmal(__int64 a, __int64 b, __int64 c)`	`xma.l` (Fixed-point multiply add using the low 64 bits of the 128-bit result. The result is signed.)
`__int64 _m64_xmalu(__int64 a, __int64 b, __int64 c)`	`xma.lu` (Fixed-point multiply add using the low 64 bits of the 128-bit result. The result is unsigned.)
`__int64 _m64_xmah(__int64 a, __int64 b, __int64 c)`	`xma.h` (Fixed-point multiply add using the high 64 bits of the 128-bit result. The result is signed.)
`__int64 _m64_xmahu(__int64 a, __int64 b, __int64 c)`	`xma.hu` (Fixed-point multiply add using the high 64 bits of the 128-bit result. The result is unsigned.)
`__int64 _m64_popcnt(__int64 a)`	`popcnt` (Population count)

Intrinsic	Corresponding Instruction
`__int64 _m64_shladd(__int64 a, const int count, __int64 b)`	`shladd` (Shift left and add)
`__int64 _m64_shrp(__int64 a, __int64 b, const int count)`	`shrp` (Shift right pair)

FSR Operations

Intrinsic	Description
`void _fsetc(int amask, int omask)`	Sets the control bits of `FPSR.sf0`. Maps to the `fsetc.sf0 r, r` instruction. There is no corresponding instruction to read the control bits. Use `_mm_getfpsr()`.
`void _fclrf(void)`	Clears the floating point status flags (the 6-bit flags of `FPSR.sf0`). Maps to the `fclrf.sf0` instruction.

`__int64 _m64_dep_mr(__int64 r, __int64 s, const int pos, const int len)`

The right-justified 64-bit value `r` is deposited into the value in `s` at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `pos` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

`__int64 _m64_dep_mi(const int v, __int64 s, const int p, const int len)`

The sign-extended value v (either all 1s or all 0s) is deposited into the value in `s` at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position p and extends to the left (toward the most significant bit) the number of bits specified by `len`.

`__int64 _m64_dep_zr(__int64 s, const int pos, const int len)`

The right-justified 64-bit value `s` is deposited into a 64-bit field of all zeros at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `pos` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

```
__int64 _m64_dep_zi(const int v, const int pos, const int
len)
```

The sign-extended value v (either all 1s or all 0s) is deposited into a 64-bit field of all zeros at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `pos` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

```
__int64 _m64_extr(__int64 r, const int pos, const int len)
```

A field is extracted from the 64-bit value `r` and is returned right-justified and sign extended. The extracted field begins at position `pos` and extends `len` bits to the left. The sign is taken from the most significant bit of the extracted field.

```
__int64 _m64_extru(__int64 r, const int pos, const int len)
```

A field is extracted from the 64-bit value `r` and is returned right-justified and zero extended. The extracted field begins at position `pos` and extends `len` bits to the left.

```
__int64 _m64_xmal(__int64 a, __int64 b, __int64 c)
```

The 64-bit values a and b are treated as signed integers and multiplied to produce a full 128-bit signed result. The 64-bit value c is zero-extended and added to the product. The least significant 64 bits of the sum are then returned.

```
__int64 _m64_xmalu(__int64 a, __int64 b, __int64 c)
```

The 64-bit values a and b are treated as signed integers and multiplied to produce a full 128-bit unsigned result. The 64-bit value c is zero-extended and added to the product. The least significant 64 bits of the sum are then returned.

```
__int64 _m64_xmah(__int64 a, __int64 b, __int64 c)
```

The 64-bit values a and b are treated as signed integers and multiplied to produce a full 128-bit signed result. The 64-bit value c is zero-extended and added to the product. The most significant 64 bits of the sum are then returned.

```
__int64 _m64_xmahu(__int64 a, __int64 b, __int64 c)
```

The 64-bit values a and b are treated as unsigned integers and multiplied to produce a full 128-bit unsigned result. The 64-bit value c is zero-extended and added to the product. The most significant 64 bits of the sum are then returned.

```
__int64 _m64_popcnt(__int64 a)
```

The number of bits in the 64-bit integer a that have the value 1 are counted, and the resulting sum is returned.

```
__int64 _m64_shladd(__int64 a, const int count, __int64 b)
```

a is shifted to the left by `count` bits and then added to b. The result is returned.

```
__int64 _m64_shrp(__int64 a, __int64 b, const int count)
```

a and b are concatenated to form a 128-bit value and shifted to the right `count` bits. The least significant 64 bits of the result are returned.

Lock and Atomic Operation Related Intrinsics

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Intrinsic	Description
`unsigned __int64 _InterlockedExchange8(volatile unsigned char *Target, unsigned __int64 value)`	Map to the `xchg1` instruction. Atomically write the least significant byte of its 2nd argument to address specified by its 1st argument.
`unsigned __int64 _InterlockedCompareExchange8_rel(volatile unsigned char *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Compare and exchange atomically the least significant byte at the address specified by its 1st argument. Maps to the `cmpxchg1.rel` instruction with appropriate setup.
`unsigned __int64 _InterlockedCompareExchange8_acq(volatile unsigned char *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Same as the previous intrinsic, but using `acquire` semantic.
`unsigned __int64 _InterlockedExchange16(volatile unsigned short *Target, unsigned __int64 value)`	Map to the `xchg2` instruction. Atomically write the least significant word of its 2nd argument to address specified by its 1st argument.
`unsigned __int64 _InterlockedCompareExchange16_rel(volatile unsigned short *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Compare and exchange atomically the least significant word at the address specified by its 1st argument. Maps to the `cmpxchg2.rel` instruction with appropriate setup.

Intrinsic	Description
`unsigned __int64 _InterlockedCompareExchange16_acq(volatile unsigned short *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Same as the previous intrinsic, but using `acquire` semantic.
`int _InterlockedIncrement(volatile int *addend`	Atomically increment by one the value specified by its argument. Maps to the `fetchadd4` instruction.
`int _InterlockedDecrement(volatile int *addend`	Atomically decrement by one the value specified by its argument. Maps to the `fetchadd4` instruction.
`int _InterlockedExchange(volatile int *Target, long value`	Do an exchange operation atomically. Maps to the `xchg4` instruction.
`int _InterlockedCompareExchange(volatile int *Destination, int Exchange, int Comparand`	Do a compare and exchange operation atomically. Maps to the `cmpxchg4` instruction with appropriate setup.
`int _InterlockedExchangeAdd(volatile int *addend, int increment`	Use compare and exchange to do an atomic add of the increment value to the addend. Maps to a loop with the `cmpxchg4` instruction to guarantee atomicity.
`int _InterlockedAdd(volatile int *addend, int increment)`	Same as the previous intrinsic, but returns new value, not the original one.
`void * _InterlockedCompareExchangePointer(void * volatile *Destination, void *Exchange, void *Comparand)`	Map the `exch8` instruction; Atomically compare and exchange the pointer value specified by its first argument (all arguments are pointers)
`unsigned __int64`	Atomically exchange the 32-

Intrinsic	Description
`_InterlockedExchangeU(volatile unsigned int *Target, unsigned __int64 value)`	bit quantity specified by the 1st argument. Maps to the `xchg4` instruction.
`unsigned __int64 _InterlockedCompareExchange_rel(volatile unsigned int *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Maps to the `cmpxchg4.rel` instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 64-bit pointer).
`unsigned __int64 _InterlockedCompareExchange_acq(volatile unsigned int *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Same as the previous intrinsic, but map the `cmpxchg4.acq` instruction.
`void _ReleaseSpinLock(volatile int *x)`	Release spin lock.
`__int64 _InterlockedIncrement64(volatile __int64 *addend)`	Increment by one the value specified by its argument. Maps to the `fetchadd` instruction.
`__int64 _InterlockedDecrement64(volatile __int64 *addend)`	Decrement by one the value specified by its argument. Maps to the `fetchadd` instruction.
`__int64 _InterlockedExchange64(volatile __int64 *Target, __int64 value)`	Do an exchange operation atomically. Maps to the `xchg` instruction.
`unsigned __int64 _InterlockedExchangeU64(volatile unsigned __int64 *Target, unsigned __int64 value)`	Same as `InterlockedExchange64` (for unsigned quantities).
`unsigned __int64 _InterlockedCompareExchange64_rel(volatile unsigned __int64 *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Maps to the `cmpxchg.rel` instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 64-bit pointer).

Intrinsic	Description
`unsigned __int64 _InterlockedCompareExchange64_acq(volatile unsigned __int64 *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Maps to the `cmpxchg.acq` instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 64-bit pointer).
`__int64 _InterlockedCompareExchange64(volatile __int64 *Destination, __int64 Exchange, __int64 Comparand)`	Same as the previous intrinsic for signed quantities.
`__int64 _InterlockedExchangeAdd64(volatile __int64 *addend, __int64 increment)`	Use compare and exchange to do an atomic add of the increment value to the addend. Maps to a loop with the `cmpxchg` instruction to guarantee atomicity
`__int64 _InterlockedAdd64(volatile __int64 *addend, __int64 increment);`	Same as the previous intrinsic, but returns the new value, not the original value. See Note.

Note

`_InterlockedSub64` is provided as a macro definition based on `_InterlockedAdd64`.

`#define _InterlockedSub64(target, incr) _InterlockedAdd64((target),(-(incr)))`.

Uses `cmpxchg` to do an atomic sub of the `incr` value to the `target`. Maps to a loop with the `cmpxchg` instruction to guarantee atomicity.

Load and Store

You can use the load and store intrinsic to force the strict memory access ordering of specific data objects. This intended use is for the case when the user suppresses the strict memory access ordering by using the `-serialize-volatile-` option.

Intrinsic	Prototype	Description
`__st1_rel`	`void __st1_rel(void *dst, const char value);`	Generates an `st1.rel` instruction.
`__st2_rel`	`void __st2_rel(void *dst, const short value);`	Generates an `st2.rel` instruction.
`__st4_rel`	`void __st4_rel(void *dst, const int value);`	Generates an `st4.rel` instruction.
`__st8_rel`	`void __st8_rel(void *dst, const __int64 value);`	Generates an `st8.rel` instruction.
`__ld1_acq`	`unsigned char __ld1_acq(void *src);`	Generates an `ld1.acq` instruction.
`__ld2_acq`	`unsigned short __ld2_acq(void *src);`	Generates an `ld2.acq` instruction.
`__ld4_acq`	`unsigned int __ld4_acq(void *src);`	Generates an `ld4.acq` instruction.
`__ld8_acq`	`unsigned __int64 __ld8_acq(void *src);`	Generates an `ld8.acq` instruction.

Operating System Related Intrinsics

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Intrinsic	Description
`unsigned __int64 __getReg(const int whichReg)`	Gets the value from a hardware register based on the index passed in. Produces a corresponding `mov = r` instruction. Provides access to the following registers: See Register Names for getReg() and setReg().

Intrinsic	Description
`void __setReg(const int whichReg, unsigned __int64 value)`	Sets the value for a hardware register based on the index passed in. Produces a corresponding `mov = r` instruction. See Register Names for getReg() and setReg().
`unsigned __int64 __getIndReg(const int whichIndReg, __int64 index)`	Return the value of an indexed register. The index is the 2nd argument; the register file is the first argument.
`void __setIndReg(const int whichIndReg, __int64 index, unsigned __int64 value)`	Copy a value in an indexed register. The index is the 2nd argument; the register file is the first argument.
`void *__ptr64 _rdteb(void)`	Gets `TEB` address. The `TEB` address is kept in `r13` and maps to the move `r=tp` instruction
`void __isrlz(void)`	Executes the serialize instruction. Maps to the `srlz.i` instruction.
`void __dsrlz(void)`	Serializes the data. Maps to the `srlz.d` instruction.
`unsigned __int64 __fetchadd4_acq(unsigned int *addend, const int increment)`	Map the `fetchadd4.acq` instruction.
`unsigned __int64 __fetchadd4_rel(unsigned int *addend, const int increment)`	Map the `fetchadd4.rel` instruction.
`unsigned __int64 __fetchadd8_acq(unsigned __int64 *addend, const int increment)`	Map the `fetchadd8.acq` instruction.
`unsigned __int64 __fetchadd8_rel(unsigned __int64 *addend, const int increment)`	Map the `fetchadd8.rel` instruction.

Intrinsic	Description
`void __fwb(void)`	Flushes the write buffers. Maps to the `fwb` instruction.
`void __ldfs(const int whichFloatReg, void *src)`	Map the `ldfs` instruction. Load a single precision value to the specified register.
`void __ldfd(const int whichFloatReg, void *src)`	Map the `ldfd` instruction. Load a double precision value to the specified register.
`void __ldfe(const int whichFloatReg, void *src)`	Map the `ldfe` instruction. Load an extended precision value to the specified register.
`void __ldf8(const int whichFloatReg, void *src)`	Map the `ldf8` instruction.
`void __ldf_fill(const int whichFloatReg, void *src)`	Map the `ldf.fill` instruction.
`void __stfs(void *dst, const int whichFloatReg)`	Map the `sfts` instruction.
`void __stfd(void *dst, const int whichFloatReg)`	Map the `stfd` instruction.
`void __stfe(void *dst, const int whichFloatReg)`	Map the `stfe` instruction.
`void __stf8(void *dst, const int whichFloatReg)`	Map the `stf8` instruction.
`void __stf_spill(void *dst, const int whichFloatReg)`	Map the `stf.spill` instruction.
`void __mf(void)`	Executes a memory fence instruction. Maps to the `mf` instruction.
`void __mfa(void)`	Executes a memory fence, acceptance form instruction. Maps to the `mf.a` instruction.
`void __synci(void)`	Enables memory synchronization. Maps to the `sync.i` instruction.

Intrinsic	Description
`void __thash(__int64)`	Generates a translation hash entry address. Maps to the `thash r = r` instruction.
`void __ttag(__int64)`	Generates a translation hash entry tag. Maps to the `ttag r=r` instruction.
`void __itcd(__int64 pa)`	Insert an entry into the data translation cache (Map `itc.d` instruction).
`void __itci(__int64 pa)`	Insert an entry into the instruction translation cache (Map `itc.i`).
`void __itrd(__int64 whichTransReg, __int64 pa)`	Map the `itr.d` instruction.
`void __itri(__int64 whichTransReg, __int64 pa)`	Map the `itr.i` instruction.
`void __ptce(__int64 va)`	Map the `ptc.e` instruction.
`void __ptcl(__int64 va, __int64 pagesz)`	Purges the local translation cache. Maps to the `ptc.l r, r` instruction.
`void __ptcg(__int64 va, __int64 pagesz)`	Purges the global translation cache. Maps to the `ptc.g r, r` instruction.
`void __ptcga(__int64 va, __int64 pagesz)`	Purges the global translation cache and ALAT. Maps to the `ptc.ga r, r` instruction.
`void __ptri(__int64 va, __int64 pagesz)`	Purges the translation register. Maps to the `ptr.i r, r` instruction.
`void __ptrd(__int64 va, __int64 pagesz)`	Purges the translation register. Maps to the `ptr.d r, r` instruction.
`__int64 __tpa(__int64 va)`	Map the `tpa` instruction.
`void __invalat(void)`	Invalidates ALAT. Maps to the `invala` instruction.
`void __invala (void)`	Same as `void __invalat(void)`

Intrinsic	Description
`void __invala_gr(const int whichGeneralReg)`	`whichGeneralReg` = 0-127
`void __invala_fr(const int whichFloatReg)`	`whichFloatReg` = 0-127
`void __break(const int)`	Generates a break instruction with an immediate.
`void __nop(const int)`	Generate a `nop` instruction.
`void __debugbreak(void)`	Generates a Debug Break Instruction fault.
`void __fc(__int64)`	Flushes a cache line associated with the address given by the argument. Maps to the `fc` instruction.
`void __sum(int mask)`	Sets the user mask bits of PSR. Maps to the `sum imm24` instruction.
`void __rum(int mask)`	Resets the user mask.
`__int64 _ReturnAddress(void)`	Get the caller's address.
`void __lfetch(int lfhint, void *y)`	Generate the `lfetch.lfhint` instruction. The value of the first argument specifies the hint type.
`void __lfetch_fault(int lfhint, void *y)`	Generate the `lfetch.fault.lfhint` instruction. The value of the first argument specifies the hint type.
`void __lfetch_excl(int lfhint, void *y)`	Generate the `lfetch.excl.lfhint` instruction. The value {0|1|2|3} of the first argument specifies the hint type.
`void __lfetch_fault_excl(int lfhint, void *y)`	Generate the `lfetch.fault.excl.lfhint` instruction. The value of the first argument specifies the hint type.
`unsigned int __cacheSize(unsigned int`	`__cacheSize(n)` returns the size in bytes of the cache at level n. 1

Intrinsic	Description
`cacheLevel)`	represents the first-level cache. 0 is returned for a non-existent cache level. For example, an application may query the cache size and use it to select block sizes in algorithms that operate on matrices.
`void __memory_barrier(void)`	Creates a barrier across which the compiler will not schedule any data access instruction. The compiler may allocate local data in registers across a memory barrier, but not global data.
`void __ssm(int mask)`	Sets the system mask. Maps to the `ssm imm24` instruction.
`void __rsm(int mask)`	Resets the system mask bits of PSR. Maps to the `rsm imm24` instruction.

Conversion Intrinsics

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Intrinsic	Description
`__int64 _m_to_int64(__m64 a)`	Convert a of type `__m64` to type `__int64`. Translates to `nop` since both types reside in the same register on Itanium-based systems.
`__m64 _m_from_int64(__int64 a)`	Convert a of type `__int64` to type `__m64`. Translates to `nop` since both types reside in the same register on Itanium-based systems.
`__int64 __round_double_to_int64(double d)`	Convert its double precision argument to a signed integer.
`unsigned __int64 __getf_exp(double d)`	Map the `getf.exp` instruction and return the 16-bit exponent and the sign of its operand.

Register Names for getReg() and setReg()

The prototypes for getReg() and setReg() intrinsics are in the `ia64regs.h` header file.

Name	whichReg
_IA64_REG_IP	1016
_IA64_REG_PSR	1019
_IA64_REG_PSR_L	1019

General Integer Registers

Name	whichReg
_IA64_REG_GP	1025
_IA64_REG_SP	1036
_IA64_REG_TP	1037

Application Registers

Name	whichReg
_IA64_REG_AR_KR0	3072
_IA64_REG_AR_KR1	3073
_IA64_REG_AR_KR2	3074
_IA64_REG_AR_KR3	3075
_IA64_REG_AR_KR4	3076
_IA64_REG_AR_KR5	3077
_IA64_REG_AR_KR6	3078
_IA64_REG_AR_KR7	3079
_IA64_REG_AR_RSC	3088
_IA64_REG_AR_BSP	3089

Name	whichReg
_IA64_REG_AR_BSPSTORE	3090
_IA64_REG_AR_RNAT	3091
_IA64_REG_AR_FCR	3093
_IA64_REG_AR_EFLAG	3096
_IA64_REG_AR_CSD	3097
_IA64_REG_AR_SSD	3098
_IA64_REG_AR_CFLAG	3099
_IA64_REG_AR_FSR	3100
_IA64_REG_AR_FIR	3101
_IA64_REG_AR_FDR	3102
_IA64_REG_AR_CCV	3104
_IA64_REG_AR_UNAT	3108
_IA64_REG_AR_FPSR	3112
_IA64_REG_AR_ITC	3116
_IA64_REG_AR_PFS	3136
_IA64_REG_AR_LC	3137
_IA64_REG_AR_EC	3138

Control Registers

Name	whichReg
_IA64_REG_CR_DCR	4096
_IA64_REG_CR_ITM	4097
_IA64_REG_CR_IVA	4098
_IA64_REG_CR_PTA	4104
_IA64_REG_CR_IPSR	4112

Name	whichReg
_IA64_REG_CR_ISR	4113
_IA64_REG_CR_IIP	4115
_IA64_REG_CR_IFA	4116
_IA64_REG_CR_ITIR	4117
_IA64_REG_CR_IIPA	4118
_IA64_REG_CR_IFS	4119
_IA64_REG_CR_IIM	4120
_IA64_REG_CR_IHA	4121
_IA64_REG_CR_LID	4160
_IA64_REG_CR_IVR	4161 *
_IA64_REG_CR_TPR	4162
_IA64_REG_CR_EOI	4163
_IA64_REG_CR_IRR0	4164 *
_IA64_REG_CR_IRR1	4165 *
_IA64_REG_CR_IRR2	4166 *
_IA64_REG_CR_IRR3	4167 *
_IA64_REG_CR_ITV	4168
_IA64_REG_CR_PMV	4169
_IA64_REG_CR_CMCV	4170
_IA64_REG_CR_LRR0	4176
_IA64_REG_CR_LRR1	4177

- `getReg` only

Indirect Registers for getIndReg() and setIndReg()

Name	whichReg
_IA64_REG_INDR_CPUID	9000 *
_IA64_REG_INDR_DBR	9001
_IA64_REG_INDR_IBR	9002
_IA64_REG_INDR_PKR	9003
_IA64_REG_INDR_PMC	9004
_IA64_REG_INDR_PMD	9005
_IA64_REG_INDR_RR	9006
_IA64_REG_INDR_RESERVED	9007

* getIndReg only

Multimedia Additions

The prototypes for these intrinsics are in the ia64intrin.h header file.

Intrinsic	Corresponding Instruction
__int64 _m64_czx1l(__m64 a)	czx1.l (Compute Zero Index)
__int64 _m64_czx1r(__m64 a)	czx1.r (Compute Zero Index)
__int64 _m64_czx2l(__m64 a)	czx2.l (Compute Zero Index)
__int64 _m64_czx2r(__m64 a)	czx2.r (Compute Zero Index)
__m64 _m64_mix1l(__m64 a, __m64 b)	mix1.l (Mix)
__m64 _m64_mix1r(__m64 a, __m64 b)	mix1.r (Mix)
__m64 _m64_mix2l(__m64 a, __m64 b)	mix2.l (Mix)
__m64 _m64_mix2r(__m64 a, __m64 b)	mix2.r (Mix)
__m64 _m64_mix4l(__m64 a, __m64 b)	mix4.l (Mix)

Intrinsic	Corresponding Instruction
`__m64 _m64_mix4r(__m64 a, __m64 b)`	`mix4.r` (Mix)
`__m64 _m64_mux1(__m64 a, const int n)`	`mux1` (Mux)
`__m64 _m64_mux2(__m64 a, const int n)`	`mux2` (Mux)
`__m64 _m64_padd1uus(__m64 a, __m64 b)`	`padd1.uus` (Parallel add)
`__m64 _m64_padd2uus(__m64 a, __m64 b)`	`padd2.uus` (Parallel add)
`__m64 _m64_pavg1_nraz(__m64 a, __m64 b)`	`pavg1` (Parallel average)
`__m64 _m64_pavg2_nraz(__m64 a, __m64 b)`	`pavg2` (Parallel average)
`__m64 _m64_pavgsub1(__m64 a, __m64 b)`	`pavgsub1` (Parallel average subtract)
`__m64 _m64_pavgsub2(__m64 a, __m64 b)`	`pavgsub2` (Parallel average subtract)
`__m64 _m64_pmpy2r(__m64 a, __m64 b)`	`pmpy2.r` (Parallel multiply)
`__m64 _m64_pmpy2l(__m64 a, __m64 b)`	`pmpy2.l` (Parallel multiply)
`__m64 _m64_pmpyshr2(__m64 a, __m64 b, const int count)`	`pmpyshr2` (Parallel multiply and shift right)
`__m64 _m64_pmpyshr2u(__m64 a, __m64 b, const int count)`	`pmpyshr2.u` (Parallel multiply and shift right)
`__m64 _m64_pshladd2(__m64 a, const int count, __m64 b)`	`pshladd2` (Parallel shift left and add)
`__m64 _m64_pshradd2(__m64 a, const int count, __m64 b)`	`pshradd2` (Parallel shift right and add)
`__m64 _m64_psub1uus(__m64 a, __m64 b)`	`psub1.uus` (Parallel subtract)

Intrinsic	Corresponding Instruction
`__m64 _m64_psub2uus(__m64 a, __m64 b)`	`psub2.uus` (Parallel subtract)

`__int64 _m64_czx1l(__m64 a)`

The 64-bit value a is scanned for a zero element from the most significant element to the least significant element, and the index of the first zero element is returned. The element width is 8 bits, so the range of the result is from 0 - 7. If no zero element is found, the default result is 8.

`__int64 _m64_czx1r(__m64 a)`

The 64-bit value a is scanned for a zero element from the least significant element to the most significant element, and the index of the first zero element is returned. The element width is 8 bits, so the range of the result is from 0 - 7. If no zero element is found, the default result is 8.

`__int64 _m64_czx2l(__m64 a)`

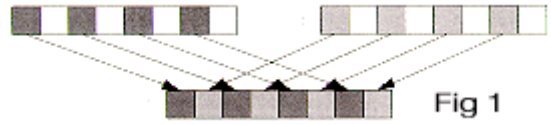
The 64-bit value a is scanned for a zero element from the most significant element to the least significant element, and the index of the first zero element is returned. The element width is 16 bits, so the range of the result is from 0 - 3. If no zero element is found, the default result is 4.

`__int64 _m64_czx2r(__m64 a)`

The 64-bit value a is scanned for a zero element from the least significant element to the most significant element, and the index of the first zero element is returned. The element width is 16 bits, so the range of the result is from 0 - 3. If no zero element is found, the default result is 4.

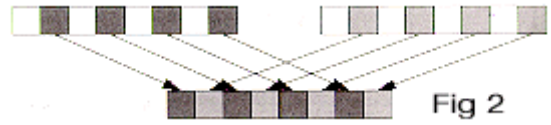
`__m64 _m64_mix1l(__m64 a, __m64 b)`

Interleave 64-bit quantities a and b in 1-byte groups, starting from the left, as shown in Figure 1, and return the result.



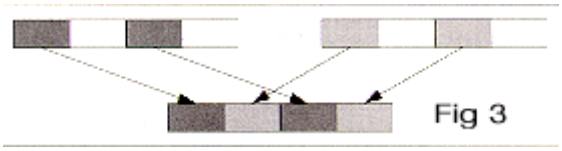
`__m64 _m64_mix1r(__m64 a, __m64 b)`

Interleave 64-bit quantities a and b in 1-byte groups, starting from the right, as shown in Figure 2, and return the result.



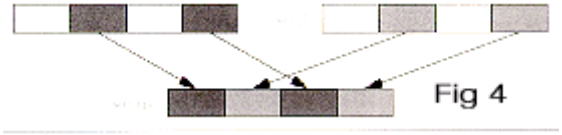
```
__m64 _m64_mix2l(__m64 a, __m64 b)
```

Interleave 64-bit quantities a and b in 2-byte groups, starting from the left, as shown in Figure 3, and return the result.




```
__m64 _m64_mix2r(__m64 a, __m64 b)
```

Interleave 64-bit quantities a and b in 2-byte groups, starting from the right, as shown in Figure 4, and return the result.




```
__m64 _m64_mix4l(__m64 a, __m64 b)
```

Interleave 64-bit quantities a and b in 4-byte groups, starting from the left, as shown in Figure 5, and return the result.


```
__m64 _m64_mix4r(__m64 a, __m64 b)
```

Interleave 64-bit quantities a and b in 4-byte groups, starting from the right, as shown in Figure 6, and return the result.


```
__m64 _m64_mux1(__m64 a, const int n)
```

Based on the value of n, a permutation is performed on a as shown in Figure 7, and the result is returned. Table 1 shows the possible values of n.

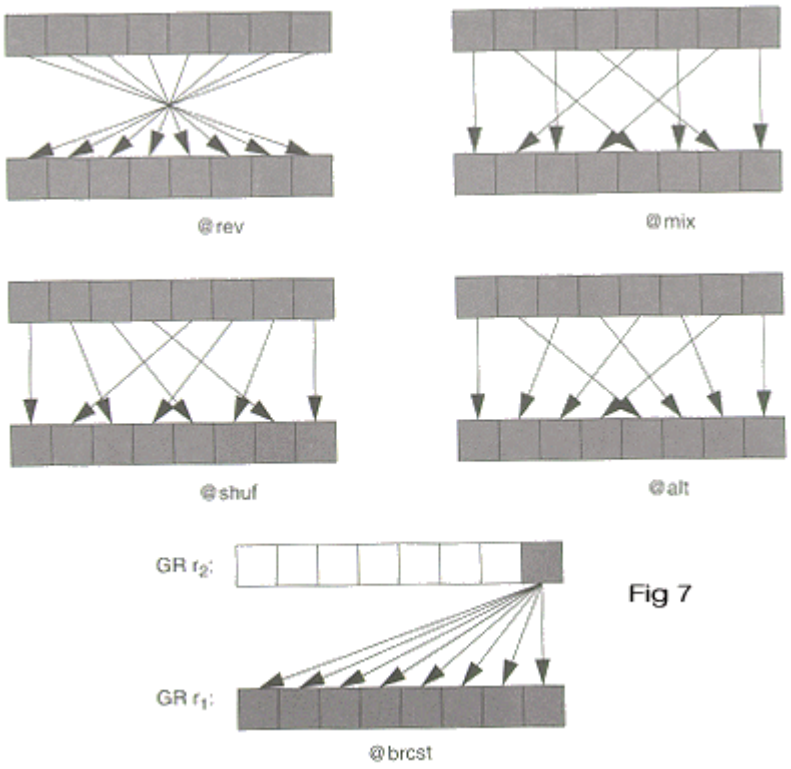


Fig 7

Table 1. Values of n for m64_mux1 Operation

	n
`@brcst`	`0`
`@mix`	`8`
`@shuf`	`9`
`@alt`	`0xA`
`@rev`	`0xB`

```
__m64 _m64_mux2(__m64 a, const int n)
```

Based on the value of n, a permutation is performed on a as shown in Figure 8, and the result is returned.

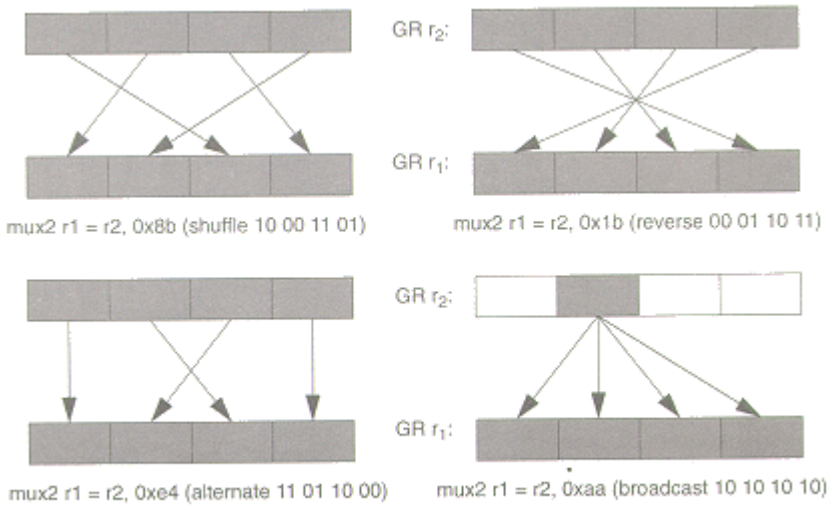


Fig 8

```
__m64 _m64_pavgsub1(__m64 a, __m64 b)
```

The unsigned data elements (bytes) of b are subtracted from the unsigned data elements (bytes) of a and the results of the subtraction are then each independently shifted to the right by one position. The high-order bits of each element are filled with the borrow bits of the subtraction.

```
__m64 _m64_pavgsub2(__m64 a, __m64 b)
```

The unsigned data elements (double bytes) of b are subtracted from the unsigned data elements (double bytes) of a and the results of the subtraction are then each independently shifted to the right by one position. The high-order bits of each element are filled with the borrow bits of the subtraction.

```
__m64 _m64_pmpy2l(__m64 a, __m64 b)
```

Two signed 16-bit data elements of a, starting with the most significant data element, are multiplied by the corresponding two signed 16-bit data elements of b, and the two 32-bit results are returned as shown in Figure 9.

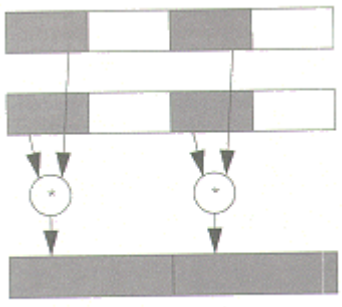


Fig 9

```
__m64 _m64_pmpy2r(__m64 a, __m64 b)
```

Two signed 16-bit data elements of a, starting with the least significant data element, are multiplied by the corresponding two signed 16-bit data elements of b, and the two 32-bit results are returned as shown in Figure 10.

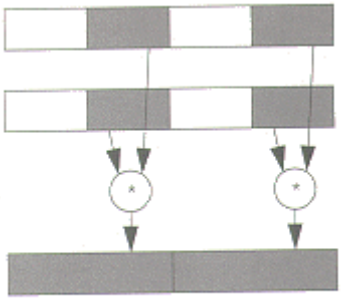


Fig 10

```
__m64 _m64_pmpyshr2(__m64 a, __m64 b, const int count)
```

The four signed 16-bit data elements of a are multiplied by the corresponding signed 16-bit data elements of b, yielding four 32-bit products. Each product is then shifted to the right count bits and the least significant 16 bits of each shifted product form 4 16-bit results, which are returned as one 64-bit word.

```
__m64 _m64_pmpyshr2u(__m64 a, __m64 b, const int count)
```

The four unsigned 16-bit data elements of a are multiplied by the corresponding unsigned 16-bit data elements of b, yielding four 32-bit products. Each product is then shifted to the right count bits and the least significant 16 bits of each shifted product form 4 16-bit results, which are returned as one 64-bit word.

```
__m64 _m64_pshladd2(__m64 a, const int count, __m64 b)
```

a is shifted to the left by count bits and then is added to b. The upper 32 bits of the result are forced to 0, and then bits [31:30] of b are copied to bits [62:61] of the result. The result is returned.

```
__m64 _m64_pshradd2(__m64 a, const int count, __m64 b)
```

The four signed 16-bit data elements of a are each independently shifted to the right by `count` bits (the high order bits of each element are filled with the initial value of the sign bits of the data elements in a); they are then added to the four signed 16-bit data elements of b. The result is returned.

```
__m64 _m64_padd1uus(__m64 a, __m64 b)
```

a is added to b as eight separate byte-wide elements. The elements of a are treated as unsigned, while the elements of b are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_padd2uus(__m64 a, __m64 b)
```

a is added to b as four separate 16-bit wide elements. The elements of a are treated as unsigned, while the elements of b are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_psub1uus(__m64 a, __m64 b)
```

a is subtracted from b as eight separate byte-wide elements. The elements of a are treated as unsigned, while the elements of b are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_psub2uus(__m64 a, __m64 b)
```

a is subtracted from b as four separate 16-bit wide elements. The elements of a are treated as unsigned, while the elements of b are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_pavg1_nraz(__m64 a, __m64 b)
```

The unsigned byte-wide data elements of a are added to the unsigned byte-wide data elements of b and the results of each add are then independently shifted to the right by one position. The high-order bits of each element are filled with the carry bits of the sums.

```
__m64 _m64_pavg2_nraz(__m64 a, __m64 b)
```

The unsigned 16-bit wide data elements of a are added to the unsigned 16-bit wide data elements of b and the results of each add are then independently shifted to the right by one position. The high-order bits of each element are filled with the carry bits of the sums.

Synchronization Primitives

The synchronization primitive intrinsics provide a variety of operations. Besides performing these operations, each intrinsic has two key properties:

- the function performed is guaranteed to be atomic
- associated with each intrinsic are certain memory barrier properties that restrict the movement of memory references to visible data across the intrinsic operation by either the compiler or the processor

For the following intrinsics, `<type>` is either a 32-bit or 64-bit integer.

Atomic Fetch-and-op Operations

```
<type> __sync_fetch_and_add(<type> *ptr,<type> val)
<type> __sync_fetch_and_and(<type> *ptr,<type> val)
<type> __sync_fetch_and_nand(<type> *ptr,<type> val)
<type> __sync_fetch_and_or(<type> *ptr,<type> val)
<type> __sync_fetch_and_sub(<type> *ptr,<type> val)
<type> __sync_fetch_and_xor(<type> *ptr,<type> val)
```

Atomic Op-and-fetch Operations

```
<type> __sync_add_and_fetch(<type> *ptr,<type> val)
<type> __sync_sub_and_fetch(<type> *ptr,<type> val)
<type> __sync_or_and_fetch(<type> *ptr,<type> val)
<type> __sync_and_and_fetch(<type> *ptr,<type> val)
<type> __sync_nand_and_fetch(<type> *ptr,<type> val)
<type> __sync_xor_and_fetch(<type> *ptr,<type> val)
```

Atomic Compare-and-swap Operations

```
<type> __sync_val_compare_and_swap(<type> *ptr, <type>
old_val, <type> new_val)
int __sync_bool_compare_and_swap(<type> *ptr, <type>
old_val, <type> new_val)
```

Atomic Synchronize Operation

```
void __sync_synchronize (void);
```

Atomic Lock-test-and-set Operation

```
<type> __sync_lock_test_and_set(<type> *ptr,<type> val)
```

Atomic Lock-release Operation

```
void __sync_lock_release(<type> *ptr)
```

Miscellaneous Intrinsics

```
void* __get_return_address(unsigned int level);
```

This intrinsic yields the return address of the current function. The `level` argument must be a constant value. A value of 0 yields the return address of the current function. Any other value yields a zero return address. On Linux systems, this intrinsic is synonymous with `__builtin_return_address`. The name and the argument are provided for compatibility with gcc*.

```
void __set_return_address(void* addr);
```

This intrinsic overwrites the default return address of the current function with the address indicated by its argument. On return from the current invocation, program execution continues at the address provided.

```
void* __get_frame_address(unsigned int level);
```

This intrinsic returns the frame address of the current function. The `level` argument must be a constant value. A value of 0 yields the frame address of the current function. Any other value yields a zero return value. On Linux systems, this intrinsic is synonymous with `__builtin_frame_address`. The name and the argument are provided for compatibility with gcc.

Data Alignment, Memory Allocation Intrinsics, and Inline Assembly

This section describes features that support usage of the intrinsics. The following topics are described:

- Alignment Support
- Allocating and Freeing Aligned Memory Blocks

Alignment Support

To improve intrinsics performance, you need to align data. For example, when you are using the Streaming SIMD Extensions, you should align data to 16 bytes in memory operations to improve performance. Specifically, you must align `__m128` objects as addresses passed to the `_mm_load` and `_mm_store` intrinsics. If you want to declare arrays of floats and treat them as `__m128` objects by casting, you need to ensure that the float arrays are properly aligned.

Use `__declspec(align)` to direct the compiler to align data more strictly than it otherwise does on both IA-32 and Itanium®-based systems. For example, a data object of type int is allocated at a byte address which is a multiple of 4 by default (the size of an int). However, by using `__declspec(align)`, you can direct the compiler to instead use an address which is a multiple of 8, 16, or 32 with the following restrictions on IA-32:

- 32-byte addresses must be statically allocated
- 16-byte addresses can be locally or statically allocated

You can use this data alignment support as an advantage in optimizing cache line usage. By clustering small objects that are commonly used together into a `struct`, and forcing the `struct` to be allocated at the beginning of a cache line, you can effectively guarantee that each object is loaded into the cache as soon as any one is accessed, resulting in a significant performance benefit.

The syntax of this extended-attribute is as follows:

`align(`*n*`)`

where *n* is an integral power of 2, less than or equal to 32. The value specified is the requested alignment.

Caution

In this release, `__declspec(align(8))` does not function correctly. Use `__declspec(align(16))` instead.

Note

If a value is specified that is less than the alignment of the affected data type, it has no effect. In other words, data is aligned to the maximum of its own alignment or the alignment specified with `__declspec(align)`.

You can request alignments for individual variables, whether of static or automatic storage duration. (Global and static variables have static storage duration; local variables have automatic storage duration by default.) You cannot adjust the alignment of a parameter, nor a field of a `struct` or `class`. You can, however, increase the alignment of a `struct` (or `union` or `class`), in which case every object of that type is affected.

As an example, suppose that a function uses local variables `i` and `j` as subscripts into a 2-dimensional array. They might be declared as follows:

```
int i, j;
```

These variables are commonly used together. But they can fall in different cache lines, which could be detrimental to performance. You can instead declare them as follows:

```
__declspec(align(8)) struct { int i, j; } sub;
```

The compiler now ensures that they are allocated in the same cache line. In C++, you can omit the `struct` variable name (written as sub in the previous example). In C, however, it is required, and you must write references to `i` and `j` as `sub.i` and `sub.j`.

If you use many functions with such subscript pairs, it is more convenient to declare and use a `struct` type for them, as in the following example:

```
typedef struct __declspec(align(8)) { int i, j; } Sub;
```

By placing the `__declspec(align)` after the keyword `struct`, you are requesting the appropriate alignment for all objects of that type. However, that allocation of parameters is unaffected by `__declspec(align)`. (If necessary, you can assign the value of a parameter to a local variable with the appropriate alignment.)

You can also force alignment of global variables, such as arrays:

```
__declspec(align(16)) float array[1000];
```

Allocating and Freeing Aligned Memory Blocks

Use the `_mm_malloc` and `_mm_free` intrinsics to allocate and free aligned blocks of memory. These intrinsics are based on `malloc` and `free`, which are in the `libirc.a` library. You need to include `malloc.h`. The syntax for these intrinsics is as follows:

```
void* _mm_malloc (int size, int align)
```

```
void _mm_free (void *p)
```

The `_mm_malloc` routine takes an extra parameter, which is the alignment constraint. This constraint must be a power of two. The pointer that is returned from `_mm_malloc` is guaranteed to be aligned on the specified boundary.

Note

Memory that is allocated using `_mm_malloc` must be freed using `_mm_free` . Calling `free` on memory allocated with `_mm_malloc` or calling `_mm_free` on memory allocated with `malloc` will cause unpredictable behavior.

Inline Assembly

By default, the compiler inlines a number of standard C, C++, and math library functions. This usually results in faster execution of your program.

Sometimes inline expansion of library functions can cause unexpected results. The inlined library functions do not set the `errno` variable. So, in code that relies upon the setting of the `errno` variable, you should use the `-nolib_inline` option, which turns off inline expansion of library functions. Also, if one of your functions has the same name as one of the compiler's supplied library functions, the compiler assumes that it is one of the latter and replaces the call with the inlined version. Consequently, if the program defines a function with the same name as one of the known library routines, you must use the `-nolib_inline` option to ensure that the program's function is the one used.

Note

Automatic inline expansion of library functions is not related to the inline expansion that the compiler does during interprocedural optimizations. For example, the following command compiles the program sum.c without expanding the library functions, but with inline expansion from interprocedural optimizations (IPO):

```
prompt>icpc -ip -nolib_inline sum.cpp
```

For details on IPO, see Interprocedural Optimizations.

MASM* Style Inline Assembly

The Intel® C++ Compiler supports MASM style inline assembly with the `-use_msasm` option. See your MASM documentation for the proper syntax.

GNU*-like Style Inline Assembly (IA-32 only)

The Intel® C++ Compiler supports GNU-like style inline assembly. The syntax is as follows:

```
asm-keyword [ volatile-keyword ] ( asm-template [ asm-interface ] ) ;
```

Caution

Under the `-use_msasm` compilation flag, Gnu asm aliases will only work if you use the `__asm__` keyword, they will not work correctly if you use the alternate `__asm` or `asm` keywords.

Syntax Element	Description
`asm-keyword`	`asm` statements begin with the keyword `asm`. Alternatively, either `__asm` or `__asm__` may be used for compatibility. See **Caution** statement.
`volatile-keyword`	If the optional keyword `volatile` is given, the `asm` is volatile. Two `volatile asm` statements will never be moved past each other, and a reference to a `volatile` variable will not be moved relative to a volatile `asm`. Alternate keywords `__volatile` and `__volatile__` may be used for compatibility.
`asm-template`	The `asm-template` is a C language ASCII string which specifies how to output the assembly code for an instruction. Most of the template is a fixed string; everything but the substitution-directives, if any, is passed through to the assembler. The syntax for a substitution directive is a % followed by one or two characters. The supported substitution directives are specified in a subsequent section.

Syntax Element	Description
asm-interface	The asm-interface consists of three parts: 1. an optional output-list 2. an optional input-list 3. an optional clobber-list These are separated by colon (:) characters. If the output-list is missing, but an input-list is given, the input list may be preceded by two colons (::)to take the place of the missing output-list. If the asm-interface is omitted altogether, the asm statement is considered volatile regardless of whether a volatile-keyword was specified.
output-list	An output-list consists of one or more output-specs separated by commas. For the purposes of substitution in the asm-template, each output-spec is numbered. The first operand in the output-list is numbered 0, the second is 1, and so on. Numbering is continuous through the output-list and into the input-list. The total number of operands is limited to 10 (i.e. 0-9).
input-list	Similar to an output-list, an input-list consists of one or more input-specs separated by commas. For the purposes of substitution in the asm-template, each input-spec is numbered, with the numbers continuing from those in the output-list.
clobber-list	A clobber-list tells the compiler that the asm uses or changes a specific machine register that is either coded directly into the asm or is changed implicitly by the assembly instruction. The clobber-list is a comma-separated list of clobber-specs.
input-spec	The input-specs tell the compiler about expressions whose values may be needed by the inserted assembly instruction. In order to describe fully the input requirements of the asm, you can list input-specs that are not actually referenced in the asm-template.

Syntax Element	Description
`clobber-spec`	Each `clobber-spec` specifies the name of a single machine register that is clobbered. The register name may optionally be preceded by a %. The following are the valid register names: eax, ebx, ecx, edx, esi, edi, ebp, esp, ax, bx, cx, dx, si, di, bp, sp, al, bl, cl, dl, ah, bh, ch, dh, st, st(1) - st(7), mm0 - mm7, xmm0 - xmm7, and cc. It is also legal to specify "memory" in a `clobber-spec`. This prevents the compiler from keeping data cached in registers across the `asm` statement.

Intrinsics Cross-processor Implementation

This section provides a series of tables that compare intrinsics performance across architectures. Before implementing intrinsics across architectures, please note the following.

- Instrinsics may generate code that does not run on all IA processors. Therefore the programmer is responsible for using `CPUID` to detect the processor and generating the appropriate code.
- Implement intrinsics by processor family, not by specific processor. The guiding principle for which family -- IA-32 or Itanium® processors -- the intrinsic is implemented on is performance, not compatibility. Where there is added performance on both families, the intrinsic will be identical.

Intrinsics For Implementation Across All IA

The following intrinsics provide significant performance gain over a non-intrinsic-based code equivalent.

`int abs(int)`
`long labs(long)`
`unsigned long __lrotl(unsigned long value, int shift)`
`unsigned long __lrotr(unsigned long value, int shift)`
`unsigned int __rotl(unsigned int value, int shift)`
`unsigned int __rotr(unsigned int value, int shift)`
`__int64 __i64_rotl(__int64 value, int shift)`

`__int64 __i64_rotr(__int64 value, int shift)`
`double fabs(double)`
`double log(double)`
`float logf(float)`
`double log10(double)`
`float log10f(float)`
`double exp(double)`
`float expf(float)`
`double pow(double, double)`
`float powf(float, float)`
`double sin(double)`
`float sinf(float)`
`double cos(double)`
`float cosf(float)`
`double tan(double)`
`float tanf(float)`
`double acos(double)`
`float acosf(float)`
`double acosh(double)`
`float acoshf(float)`
`double asin(double)`
`float asinf(float)`
`double asinh(double)`
`float asinhf(float)`
`double atan(double)`

float atanf(float)
double atanh(double)
float atanhf(float)
float cabs(double)*
double ceil(double)
float ceilf(float)
double cosh(double)
float coshf(float)
float fabsf(float)
double floor(double)
float floorf(float)
double fmod(double)
float fmodf(float)
double hypot(double, double)
float hypotf(float)
double rint(double)
float rintf(float)
double sinh(double)
float sinhf(float)
float sqrtf(float)
double tanh(double)
float tanhf(float)
char *_strset(char *, _int32)
void *memcmp(const void *cs, const void *ct, size_t n)
void *memcpy(void *s, const void *ct, size_t n)

void *memset(void * s, int c, size_t n)
char *Strcat(char * s, const char * ct)
int *strcmp(const char *, const char *)
char *strcpy(char * s, const char * ct)
size_t strlen(const char * cs)
int strncmp(char *, char *, int)
int strncpy(char *, char *, int)
void *__alloca(int)
int _setjmp(jmp_buf)
_exception_code(void)
_exception_info(void)
_abnormal_termination(void)
void _enable()
void _disable()
int _bswap(int)
int _in_byte(int)
int _in_dword(int)
int _in_word(int)
int _inp(int)
int _inpd(int)
int _inpw(int)
int _out_byte(int, int)
int _out_dword(int, int)
int _out_word(int, int)
int _outp(int, int)

`int _outpd(int, int)`
`int _outpw(int, int)`

MMX(TM) Technology Intrinsics Implementation

Key to the table entries

- A = Expected to give significant performance gain over non-intrinsic-based code equivalent.
- B = Non-intrinsic-based source code would be better; the intrinsic's implementation may map directly to native instructions, but they offer no significant performance gain.
- C = Requires contorted implementation for particular microarchitecture. Will result in very poor performance if used.

Intrinsic Name	Alternate Name	Across All IA	MMX(TM) Technology Streaming SIMD Extensions Streaming SIMD Extensions 2	Itanium® Architecture
`_m_empty`	`_mm_empty`	N/A	A	B
`_m_from_int`	`_mm_cvtsi32_si64`	N/A	A	A
`_m_to_int`	`_mm_cvtsi64_si32`	N/A	A	A
`_m_packsswb`	`_mm_packs_pi16`	N/A	A	A
`_m_packssdw`	`_mm_packs_pi32`	N/A	A	A
`_m_packuswb`	`_mm_packs_pu16`	N/A	A	A
`_m_punpckhbw`	`_mm_unpackhi_pi8`	N/A	A	A
`_m_punpckhwd`	`_mm_unpackhi_pi16`	N/A	A	A
`_m_punpckhdq`	`_mm_unpackhi_pi32`	N/A	A	A

_m_punpcklbw	_mm_unpacklo_pi8	N/A	A	A
_m_punpcklwd	_mm_unpacklo_pi16	N/A	A	A
_m_punpckldq	_mm_unpacklo_pi32	N/A	A	A
_m_paddb	_mm_add_pi8	N/A	A	A
_m_paddw	_mm_add_pi16	N/A	A	A
_m_paddd	_mm_add_pi32	N/A	A	A
_m_paddsb	_mm_adds_pi8	N/A	A	A
_m_paddsw	_mm_adds_pi16	N/A	A	A
_m_paddusb	_mm_adds_pu8	N/A	A	A
_m_paddusw	_mm_adds_pu16	N/A	A	A
_m_psubb	_mm_sub_pi8	N/A	A	A
_m_psubw	_mm_sub_pi16	N/A	A	A
_m_psubd	_mm_sub_pi32	N/A	A	A
_m_psubsb	_mm_subs_pi8	N/A	A	A
_m_psubsw	_mm_subs_pi16	N/A	A	A
_m_psubusb	_mm_subs_pu8	N/A	A	A
_m_psubusw	_mm_subs_pu16	N/A	A	A
_m_pmaddwd	_mm_madd_pi16	N/A	A	C
_m_pmulhw	_mm_mulhi_pi16	N/A	A	A
_m_pmullw	_mm_mullo_pi16	N/A	A	A
_m_psllw	_mm_sll_pi16	N/A	A	A
_m_psllwi	_mm_slli_pi16	N/A	A	A
_m_pslld	_mm_sll_pi32	N/A	A	A
_m_pslldi	_mm_slli_pi32	N/A	A	A
_m_psllq	_mm_sll_si64	N/A	A	A

_m_psllqi	_mm_slli_si64	N/A	A	A
_m_psraw	_mm_sra_pi16	N/A	A	A
_m_psrawi	_mm_srai_pi16	N/A	A	A
_m_psrad	_mm_sra_pi32	N/A	A	A
_m_psradi	_mm_srai_pi32	N/A	A	A
_m_psrlw	_mm_srl_pi16	N/A	A	A
_m_psrlwi	_mm_srli_pi16	N/A	A	A
_m_psrld	_mm_srl_pi32	N/A	A	A
_m_psrldi	_mm_srli_pi32	N/A	A	A
_m_psrlq	_mm_srl_si64	N/A	A	A
_m_psrlqi	_mm_srli_si64	N/A	A	A
_m_pand	_mm_and_si64	N/A	A	A
_m_pandn	_mm_andnot_si64	N/A	A	A
_m_por	_mm_or_si64	N/A	A	A
_m_pxor	_mm_xor_si64	N/A	A	A
_m_pcmpeqb	_mm_cmpeq_pi8	N/A	A	A
_m_pcmpeqw	_mm_cmpeq_pi16	N/A	A	A
_m_pcmpeqd	_mm_cmpeq_pi32	N/A	A	A
_m_pcmpgtb	_mm_cmpgt_pi8	N/A	A	A
_m_pcmpgtw	_mm_cmpgt_pi16	N/A	A	A
_m_pcmpgtd	_mm_cmpgt_pi32	N/A	A	A
_mm_setzero_si64		N/A	A	A
_mm_set_pi32		N/A	A	A
_mm_set_pi16		N/A	A	C
_mm_set_pi8		N/A	A	C

_mm_set1_pi32		N/A	A	A
_mm_set1_pi16		N/A	A	A
_mm_set1_pi8		N/A	A	A
_mm_setr_pi32		N/A	A	A
_mm_setr_pi16		N/A	A	C
_mm_setr_pi8		N/A	A	C

_mm_empty is implemented in Itanium instructions as a NOP for source compatibility only.

Streaming SIMD Extensions Intrinsics Implementation

Regular Streaming SIMD Extensions intrinsics work on 4 32-bit single precision values. On Itanium®-based systems basic operations like add or compare will require two SIMD instructions. All can be executed in the same cycle so the throughput is one basic Streaming SIMD Extensions operation per cycle or 4 32-bit single precision operations per cycle.

Key to the table entries

- A = Expected to give significant performance gain over non-intrinsic-based code equivalent.
- B = Non-intrinsic-based source code would be better; the intrinsic's implementation may map directly to native instructions but they offer no significant performance gain.
- C = Requires contorted implementation for particular microarchitecture. Will result in very poor performance if used.

Intrinsic Name	Alternate Name	Across All IA	MMX(TM Technology	Streaming SIMD Extensions Streaming SIMD Extensions 2	Itanium® Architectu
_mm_add_ss		N/A	N/A	B	B
_mm_add_ps		N/A	N/A	A	A
_mm_sub_ss		N/A	N/A	B	B

_mm_sub_ps		N/A	N/A	A	A
_mm_mul_ss		N/A	N/A	B	B
_mm_mul_ps		N/A	N/A	A	A
_mm_div_ss		N/A	N/A	B	B
_mm_div_ps		N/A	N/A	A	A
_mm_sqrt_ss		N/A	N/A	B	B
_mm_sqrt_ps		N/A	N/A	A	A
_mm_rcp_ss		N/A	N/A	B	B
_mm_rcp_ps		N/A	N/A	A	A
_mm_rsqrt_ss		N/A	N/A	B	B
_mm_rsqrt_ps		N/A	N/A	A	A
_mm_min_ss		N/A	N/A	B	B
_mm_min_ps		N/A	N/A	A	A
_mm_max_ss		N/A	N/A	B	B
_mm_max_ps		N/A	N/A	A	A
_mm_and_ps		N/A	N/A	A	A
_mm_andnot_ps		N/A	N/A	A	A
_mm_or_ps		N/A	N/A	A	A
_mm_xor_ps		N/A	N/A	A	A
_mm_cmpeq_ss		N/A	N/A	B	B
_mm_cmpeq_ps		N/A	N/A	A	A
_mm_cmplt_ss		N/A	N/A	B	B
_mm_cmplt_ps		N/A	N/A	A	A
_mm_cmple_ss		N/A	N/A	B	B
_mm_cmple_ps		N/A	N/A	A	A

_mm_cmpgt_ss		N/A	N/A	B	B
_mm_cmpgt_ps		N/A	N/A	A	A
_mm_cmpge_ss		N/A	N/A	B	B
_mm_cmpge_ps		N/A	N/A	A	A
_mm_cmpneq_ss		N/A	N/A	B	B
_mm_cmpneq_ps		N/A	N/A	A	A
_mm_cmpnlt_ss		N/A	N/A	B	B
_mm_cmpnlt_ps		N/A	N/A	A	A
_mm_cmpnle_ss		N/A	N/A	B	B
_mm_cmpnle_ps		N/A	N/A	A	A
_mm_cmpngt_ss		N/A	N/A	B	B
_mm_cmpngt_ps		N/A	N/A	A	A
_mm_cmpnge_ss		N/A	N/A	B	B
_mm_cmpnge_ps		N/A	N/A	A	A
_mm_cmpord_ss		N/A	N/A	B	B
_mm_cmpord_ps		N/A	N/A	A	A
_mm_cmpunord_ss		N/A	N/A	B	B
_mm_cmpunord_ps		N/A	N/A	A	A
_mm_comieq_ss		N/A	N/A	B	B
_mm_comilt_ss		N/A	N/A	B	B
_mm_comile_ss		N/A	N/A	B	B
_mm_comigt_ss		N/A	N/A	B	B
_mm_comige_ss		N/A	N/A	B	B
_mm_comineq_ss		N/A	N/A	B	B
_mm_ucomieq_ss		N/A	N/A	B	B

_mm_ucomilt_ss		N/A	N/A	B	B
_mm_ucomile_ss		N/A	N/A	B	B
_mm_ucomigt_ss		N/A	N/A	B	B
_mm_ucomige_ss		N/A	N/A	B	B
_mm_ucomineq_ss		N/A	N/A	B	B
_mm_cvt_ss2si	_mm_cvtss_si32	N/A	N/A	A	B
_mm_cvt_ps2pi	_mm_cvtps_pi32	N/A	N/A	A	A
_mm_cvtt_ss2si	_mm_cvttss_si32	N/A	N/A	A	B
_mm_cvtt_ps2pi	_mm_cvttps_pi32	N/A	N/A	A	A
_mm_cvt_si2ss	_mm_cvtsi32_ss	N/A	N/A	A	B
_mm_cvt_pi2ps	_mm_cvtpi32_ps	N/A	N/A	A	C
_mm_cvtpi16_ps		N/A	N/A	A	C
_mm_cvtpu16_ps		N/A	N/A	A	C
_mm_cvtpi8_ps		N/A	N/A	A	C
_mm_cvtpu8_ps		N/A	N/A	A	C
_mm_cvtpi32x2_ps		N/A	N/A	A	C
_mm_cvtps_pi16		N/A	N/A	A	C
_mm_cvtps_pi8		N/A	N/A	A	C
_mm_move_ss		N/A	N/A	A	A
_mm_shuffle_ps		N/A	N/A	A	A
_mm_unpackhi_ps		N/A	N/A	A	A
_mm_unpacklo_ps		N/A	N/A	A	A
_mm_movehl_ps		N/A	N/A	A	A
_mm_movelh_ps		N/A	N/A	A	A
_mm_movemask_ps		N/A	N/A	A	C

_mm_getcsr		N/A	N/A	A	A
_mm_setcsr		N/A	N/A	A	A
_mm_loadh_pi		N/A	N/A	A	A
_mm_loadl_pi		N/A	N/A	A	A
_mm_load_ss		N/A	N/A	A	B
_mm_load_ps1	_mm_load1_ps	N/A	N/A	A	A
_mm_load_ps		N/A	N/A	A	A
_mm_loadu_ps		N/A	N/A	A	A
_mm_loadr_ps		N/A	N/A	A	A
_mm_storeh_pi		N/A	N/A	A	A
_mm_storel_pi		N/A	N/A	A	A
_mm_store_ss		N/A	N/A	A	A
_mm_store_ps		N/A	N/A	A	A
_mm_store_ps1	_mm_store1_ps	N/A	N/A	A	A
_mm_storeu_ps		N/A	N/A	A	A
_mm_storer_ps		N/A	N/A	A	A
_mm_set_ss		N/A	N/A	A	A
_mm_set_ps1	_mm_set1_ps	N/A	N/A	A	A
_mm_set_ps		N/A	N/A	A	A
_mm_setr_ps		N/A	N/A	A	A
_mm_setzero_ps		N/A	N/A	A	A
_mm_prefetch		N/A	N/A	A	A
_mm_stream_pi		N/A	N/A	A	A
_mm_stream_ps		N/A	N/A	A	A
_mm_sfence		N/A	N/A	A	A

_m_pextrw	_mm_extract_pi16	N/A	N/A	A	A
_m_pinsrw	_mm_insert_pi16	N/A	N/A	A	A
_m_pmaxsw	_mm_max_pi16	N/A	N/A	A	A
_m_pmaxub	_mm_max_pu8	N/A	N/A	A	A
_m_pminsw	_mm_min_pi16	N/A	N/A	A	A
_m_pminub	_mm_min_pu8	N/A	N/A	A	A
_m_pmovmskb	_mm_movemask_pi8	N/A	N/A	A	C
_m_pmulhuw	_mm_mulhi_pu16	N/A	N/A	A	A
_m_pshufw	_mm_shuffle_pi16	N/A	N/A	A	A
_m_maskmovq	_mm_maskmove_si64	N/A	N/A	A	C
_m_pavgb	_mm_avg_pu8	N/A	N/A	A	A
_m_pavgw	_mm_avg_pu16	N/A	N/A	A	A
_m_psadbw	_mm_sad_pu8	N/A	N/A	A	A

Streaming SIMD Extensions 2 Intrinsics Implementation

Streaming SIMD Extensions 2 operate on 128-bit quantities with 64-bit double precision floating-point values. The Intel® Itanium® processor does not support parallel double precision computation, so Streaming SIMD Extensions 2 are not implemented on Itanium-based systems.

Key to the table entries

- A = Expected to give significant performance gain over non-intrinsic-based code equivalent.
- B = Non-intrinsic-based source code would be better; the intrinsic's implementation may map directly to native instructions, but they offer no significant performance gain.
- C = Requires contorted implementation for particular microarchitecture. Will result in very poor performance if used.

Intrinsic	Across All IA	MMX(TM) Technology	Streaming SIMD Extenions	Streaming SIMD Extensions 2	Itanium® Architecture
_mm_add_sd	N/A	N/A	N/A	A	N/A
_mm_add_pd	N/A	N/A	N/A	A	N/A
_mm_sub_sd	N/A	N/A	N/A	A	N/A
_mm_sub_pd	N/A	N/A	N/A	A	N/A
_mm_mul_sd	N/A	N/A	N/A	A	N/A
_mm_mul_pd	N/A	N/A	N/A	A	N/A
_mm_sqrt_sd	N/A	N/A	N/A	A	N/A
_mm_sqrt_pd	N/A	N/A	N/A	A	N/A
_mm_div_sd	N/A	N/A	N/A	A	N/A
_mm_div_pd	N/A	N/A	N/A	A	N/A
_mm_min_sd	N/A	N/A	N/A	A	N/A
_mm_min_pd	N/A	N/A	N/A	A	N/A
_mm_max_sd	N/A	N/A	N/A	A	N/A
_mm_max_pd	N/A	N/A	N/A	A	N/A
_mm_and_pd	N/A	N/A	N/A	A	N/A
_mm_andnot_pd	N/A	N/A	N/A	A	N/A
_mm_or_pd	N/A	N/A	N/A	A	N/A
_mm_xor_pd	N/A	N/A	N/A	A	N/A
_mm_cmpeq_sd	N/A	N/A	N/A	A	N/A
_mm_cmpeq_pd	N/A	N/A	N/A	A	N/A
_mm_cmplt_sd	N/A	N/A	N/A	A	N/A
_mm_cmplt_pd	N/A	N/A	N/A	A	N/A

_mm_cmple_sd	N/A	N/A	N/A	A	N/A
_mm_cmple_pd	N/A	N/A	N/A	A	N/A
_mm_cmpgt_sd	N/A	N/A	N/A	A	N/A
_mm_cmpgt_pd	N/A	N/A	N/A	A	N/A
_mm_cmpge_sd	N/A	N/A	N/A	A	N/A
_mm_cmpge_pd	N/A	N/A	N/A	A	N/A
_mm_cmpneq_sd	N/A	N/A	N/A	A	N/A
_mm_cmpneq_pd	N/A	N/A	N/A	A	N/A
_mm_cmpnlt_sd	N/A	N/A	N/A	A	N/A
_mm_cmpnlt_pd	N/A	N/A	N/A	A	N/A
_mm_cmpnle_sd	N/A	N/A	N/A	A	N/A
_mm_cmpnle_pd	N/A	N/A	N/A	A	N/A
_mm_cmpngt_sd	N/A	N/A	N/A	A	N/A
_mm_cmpngt_pd	N/A	N/A	N/A	A	N/A
_mm_cmpnge_sd	N/A	N/A	N/A	A	N/A
_mm_cmpnge_pd	N/A	N/A	N/A	A	N/A
_mm_cmpord_pd	N/A	N/A	N/A	A	N/A
_mm_cmpord_sd	N/A	N/A	N/A	A	N/A
_mm_cmpunord_pd	N/A	N/A	N/A	A	N/A
_mm_cmpunord_sd	N/A	N/A	N/A	A	N/A
_mm_comieq_sd	N/A	N/A	N/A	A	N/A
_mm_comilt_sd	N/A	N/A	N/A	A	N/A
_mm_comile_sd	N/A	N/A	N/A	A	N/A
_mm_comigt_sd	N/A	N/A	N/A	A	N/A
_mm_comige_sd	N/A	N/A	N/A	A	N/A

_mm_comineq_sd	N/A	N/A	N/A	A	N/A
_mm_ucomieq_sd	N/A	N/A	N/A	A	N/A
_mm_ucomilt_sd	N/A	N/A	N/A	A	N/A
_mm_ucomile_sd	N/A	N/A	N/A	A	N/A
_mm_ucomigt_sd	N/A	N/A	N/A	A	N/A
_mm_ucomige_sd	N/A	N/A	N/A	A	N/A
_mm_ucomineq_sd	N/A	N/A	N/A	A	N/A
_mm_cvtepi32_pd	N/A	N/A	N/A	A	N/A
_mm_cvtpd_epi32	N/A	N/A	N/A	A	N/A
_mm_cvttpd_epi32	N/A	N/A	N/A	A	N/A
_mm_cvtepi32_ps	N/A	N/A	N/A	A	N/A
_mm_cvtps_epi32	N/A	N/A	N/A	A	N/A
_mm_cvttps_epi32	N/A	N/A	N/A	A	N/A
_mm_cvtpd_ps	N/A	N/A	N/A	A	N/A
_mm_cvtps_pd	N/A	N/A	N/A	A	N/A
_mm_cvtsd_ss	N/A	N/A	N/A	A	N/A
_mm_cvtss_sd	N/A	N/A	N/A	A	N/A
_mm_cvtsd_si32	N/A	N/A	N/A	A	N/A
_mm_cvttsd_si32	N/A	N/A	N/A	A	N/A
_mm_cvtsi32_sd	N/A	N/A	N/A	A	N/A
_mm_cvtpd_pi32	N/A	N/A	N/A	A	N/A
_mm_cvttpd_pi32	N/A	N/A	N/A	A	N/A
_mm_cvtpi32_pd	N/A	N/A	N/A	A	N/A
_mm_unpackhi_pd	N/A	N/A	N/A	A	N/A
_mm_unpacklo_pd	N/A	N/A	N/A	A	N/A

_mm_unpacklo_pd	N/A	N/A	N/A	A	N/A
_mm_shuffle_pd	N/A	N/A	N/A	A	N/A
_mm_load_pd	N/A	N/A	N/A	A	N/A
_mm_load1_pd	N/A	N/A	N/A	A	N/A
_mm_loadr_pd	N/A	N/A	N/A	A	N/A
_mm_loadu_pd	N/A	N/A	N/A	A	N/A
_mm_load_sd	N/A	N/A	N/A	A	N/A
_mm_loadh_pd	N/A	N/A	N/A	A	N/A
_mm_loadl_pd	N/A	N/A	N/A	A	N/A
_mm_set_sd	N/A	N/A	N/A	A	N/A
_mm_set1_pd	N/A	N/A	N/A	A	N/A
_mm_set_pd	N/A	N/A	N/A	A	N/A
_mm_setr_pd	N/A	N/A	N/A	A	N/A
_mm_setzero_pd	N/A	N/A	N/A	A	N/A
_mm_move_sd	N/A	N/A	N/A	A	N/A
_mm_store_sd	N/A	N/A	N/A	A	N/A
_mm_store1_pd	N/A	N/A	N/A	A	N/A
_mm_store_pd	N/A	N/A	N/A	A	N/A
_mm_storeu_pd	N/A	N/A	N/A	A	N/A
_mm_storer_pd	N/A	N/A	N/A	A	N/A
_mm_storeh_pd	N/A	N/A	N/A	A	N/A
_mm_storel_pd	N/A	N/A	N/A	A	N/A
_mm_add_epi8	N/A	N/A	N/A	A	N/A
_mm_add_epi16	N/A	N/A	N/A	A	N/A
_mm_add_epi32	N/A	N/A	N/A	A	N/A

_mm_add_si64	N/A	N/A	N/A	A	N/A
_mm_add_epi64	N/A	N/A	N/A	A	N/A
_mm_adds_epi8	N/A	N/A	N/A	A	N/A
_mm_adds_epi16	N/A	N/A	N/A	A	N/A
_mm_adds_epu8	N/A	N/A	N/A	A	N/A
_mm_adds_epu16	N/A	N/A	N/A	A	N/A
_mm_avg_epu8	N/A	N/A	N/A	A	N/A
_mm_avg_epu16	N/A	N/A	N/A	A	N/A
_mm_madd_epi16	N/A	N/A	N/A	A	N/A
_mm_max_epi16	N/A	N/A	N/A	A	N/A
_mm_max_epu8	N/A	N/A	N/A	A	N/A
_mm_min_epi16	N/A	N/A	N/A	A	N/A
_mm_min_epu8	N/A	N/A	N/A	A	N/A
_mm_mulhi_epi16	N/A	N/A	N/A	A	N/A
_mm_mulhi_epu16	N/A	N/A	N/A	A	N/A
_mm_mullo_epi16	N/A	N/A	N/A	A	N/A
_mm_mul_su32	N/A	N/A	N/A	A	N/A
_mm_mul_epu32	N/A	N/A	N/A	A	N/A
_mm_sad_epu8	N/A	N/A	N/A	A	N/A
_mm_sub_epi8	N/A	N/A	N/A	A	N/A
_mm_sub_epi16	N/A	N/A	N/A	A	N/A
_mm_sub_epi32	N/A	N/A	N/A	A	N/A
_mm_sub_si64	N/A	N/A	N/A	A	N/A
_mm_sub_epi64	N/A	N/A	N/A	A	N/A
_mm_subs_epi8	N/A	N/A	N/A	A	N/A

_mm_subs_epi16	N/A	N/A	N/A	A	N/A
_mm_subs_epu8	N/A	N/A	N/A	A	N/A
_mm_subs_epu16	N/A	N/A	N/A	A	N/A
_mm_and_si128	N/A	N/A	N/A	A	N/A
_mm_andnot_si128	N/A	N/A	N/A	A	N/A
_mm_or_si128	N/A	N/A	N/A	A	N/A
_mm_xor_si128	N/A	N/A	N/A	A	N/A
_mm_slli_si128	N/A	N/A	N/A	A	N/A
_mm_slli_epi16	N/A	N/A	N/A	A	N/A
_mm_sll_epi16	N/A	N/A	N/A	A	N/A
_mm_slli_epi32	N/A	N/A	N/A	A	N/A
_mm_sll_epi32	N/A	N/A	N/A	A	N/A
_mm_slli_epi64	N/A	N/A	N/A	A	N/A
_mm_sll_epi64	N/A	N/A	N/A	A	N/A
_mm_srai_epi16	N/A	N/A	N/A	A	N/A
_mm_sra_epi16	N/A	N/A	N/A	A	N/A
_mm_srai_epi32	N/A	N/A	N/A	A	N/A
_mm_sra_epi32	N/A	N/A	N/A	A	N/A
_mm_srli_si128	N/A	N/A	N/A	A	N/A
_mm_srli_epi16	N/A	N/A	N/A	A	N/A
_mm_srl_epi16	N/A	N/A	N/A	A	N/A
_mm_srli_epi32	N/A	N/A	N/A	A	N/A
_mm_srl_epi32	N/A	N/A	N/A	A	N/A
_mm_srli_epi64	N/A	N/A	N/A	A	N/A
_mm_srl_epi64	N/A	N/A	N/A	A	N/A

_mm_cmpeq_epi8	N/A	N/A	N/A	A	N/A
_mm_cmpeq_epi16	N/A	N/A	N/A	A	N/A
_mm_cmpeq_epi32	N/A	N/A	N/A	A	N/A
_mm_cmpgt_epi8	N/A	N/A	N/A	A	N/A
_mm_cmpgt_epi16	N/A	N/A	N/A	A	N/A
_mm_cmpgt_epi32	N/A	N/A	N/A	A	N/A
_mm_cmplt_epi8	N/A	N/A	N/A	A	N/A
_mm_cmplt_epi16	N/A	N/A	N/A	A	N/A
_mm_cmplt_epi32	N/A	N/A	N/A	A	N/A
_mm_cvtsi32_si128	N/A	N/A	N/A	A	N/A
_mm_cvtsi128_si32	N/A	N/A	N/A	A	N/A
_mm_packs_epi16	N/A	N/A	N/A	A	N/A
_mm_packs_epi32	N/A	N/A	N/A	A	N/A
_mm_packus_epi16	N/A	N/A	N/A	A	N/A
_mm_extract_epi16	N/A	N/A	N/A	A	N/A
_mm_insert_epi16	N/A	N/A	N/A	A	N/A
_mm_movemask_epi8	N/A	N/A	N/A	A	N/A
_mm_shuffle_epi32	N/A	N/A	N/A	A	N/A
_mm_shufflehi_epi16	N/A	N/A	N/A	A	N/A
_mm_shufflelo_epi16	N/A	N/A	N/A	A	N/A
_mm_unpackhi_epi8	N/A	N/A	N/A	A	N/A
_mm_unpackhi_epi16	N/A	N/A	N/A	A	N/A
_mm_unpackhi_epi32	N/A	N/A	N/A	A	N/A
_mm_unpackhi_epi64	N/A	N/A	N/A	A	N/A
_mm_unpacklo_epi8	N/A	N/A	N/A	A	N/A

_mm_unpacklo_epi16	N/A	N/A	N/A	A	N/A
_mm_unpacklo_epi32	N/A	N/A	N/A	A	N/A
_mm_unpacklo_epi64	N/A	N/A	N/A	A	N/A
_mm_move_epi64	N/A	N/A	N/A	A	N/A
_mm_movpi64_epi64	N/A	N/A	N/A	A	N/A
_mm_movepi64_pi64	N/A	N/A	N/A	A	N/A
_mm_load_si128	N/A	N/A	N/A	A	N/A
_mm_loadu_si128	N/A	N/A	N/A	A	N/A
_mm_loadl_epi64	N/A	N/A	N/A	A	N/A
_mm_set_epi64	N/A	N/A	N/A	A	N/A
_mm_set_epi32	N/A	N/A	N/A	A	N/A
_mm_set_epi16	N/A	N/A	N/A	A	N/A
_mm_set_epi8	N/A	N/A	N/A	A	N/A
_mm_set1_epi64	N/A	N/A	N/A	A	N/A
_mm_set1_epi32	N/A	N/A	N/A	A	N/A
_mm_set1_epi16	N/A	N/A	N/A	A	N/A
_mm_set1_epi8	N/A	N/A	N/A	A	N/A
_mm_setr_epi64	N/A	N/A	N/A	A	N/A
_mm_setr_epi32	N/A	N/A	N/A	A	N/A
_mm_setr_epi16	N/A	N/A	N/A	A	N/A
_mm_setr_epi8	N/A	N/A	N/A	A	N/A
_mm_setzero_si128	N/A	N/A	N/A	A	N/A
_mm_store_si128	N/A	N/A	N/A	A	N/A
_mm_storeu_si128	N/A	N/A	N/A	A	N/A
_mm_storel_epi64	N/A	N/A	N/A	A	N/A

_mm_maskmoveu_si128	N/A	N/A	N/A	A	N/A
_mm_stream_pd	N/A	N/A	N/A	A	N/A
_mm_stream_si128	N/A	N/A	N/A	A	N/A
_mm_clflush	N/A	N/A	N/A	A	N/A
_mm_lfence	N/A	N/A	N/A	A	N/A
_mm_mfence	N/A	N/A	N/A	A	N/A
_mm_stream_si32	N/A	N/A	N/A	A	N/A
_mm_pause	N/A	N/A	N/A	A	N/A

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

You must have the Intel® C++ Compiler version 4.0 or higher installed on your system to use the class libraries. 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(TM) technology	Pentium® with MMX technology, Pentium II, Pentium III, Pentium 4, Intel® Xeon(TM), and Itanium® processors
`fvec.h`	Streaming SIMD Extensions	Pentium III, Pentium 4, Intel Xeon, and Itanium processors
`dvec.h`	Streaming SIMD Extensions 2	Pentium 4 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

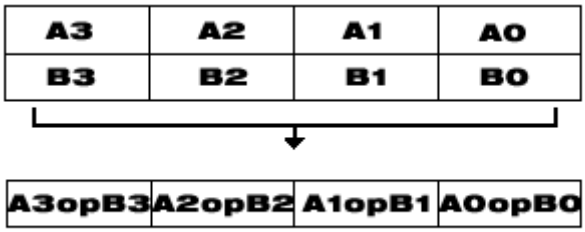
Streaming SIMD Extensions 2 intrinsics cannot be used on Itanium®-based systems. The `mmclass.h` header file includes the classes that are usable on the Itanium architecuture.

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 (available for IA-32- and Itanium®-based systems)	I64vec1	unspecified	__m64	64	1	ivec.h
	I32vec2	unspecified	int	32	2	ivec.h
	Is32vec2	signed	int	32	2	ivec.h

Instruction Set	Class	Signedness	Data Type	Size	Elements	Header File
	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 (available for IA-32 and Itanium-based systems)	F32vec4	signed	float	32	4	fvec.h
	F32vec1	signed	float	32	1	fvec.h
Streaming SIMD Extensions 2 (available for IA-32)	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

Instruction Set	Class	Signedness	Data Type	Size	Elements	Header File
	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)
```

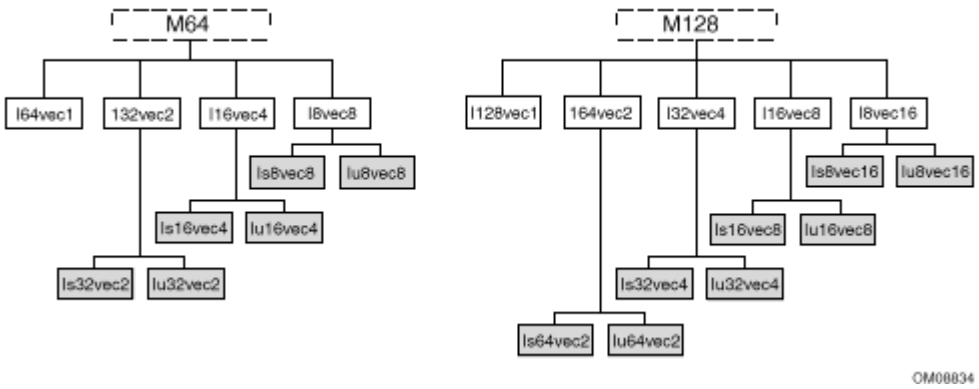
Caching Hints

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`, `164vec2`, `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.

Operators	Sign Typecasting	Size Typecasting	Other Typecasting Requirements
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);`

Operation	Class	Syntax

`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;

Return Type	Left Side (R)	AND	OR	XOR	Right Side (Any Ivec Type)
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 or Equal To	`select_le`	R = `select_le`(A, B, C, D)	`_mm_cmple_pi32`
			`_mm_cmple_pi16`
			`_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 Intrinsics 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:

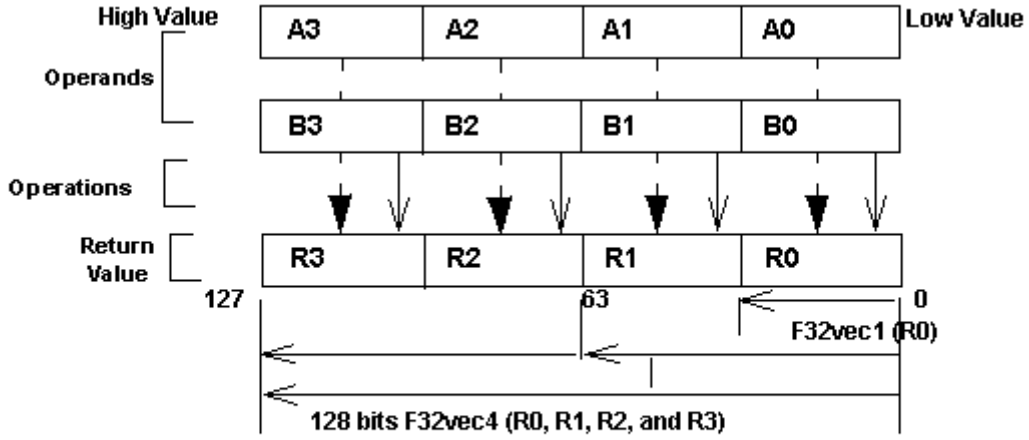
```
F64vec2 A(double x, double y);
```

```
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	F32vec
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)`

Compare Operators

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 !(A1	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B3) B3)	0xffffffff	0x0000000	X	N/A	N/A
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] _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_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);
}
```

Index