

# **Intel® C++ Compiler for Linux\***

## **Reference**

---

Document Number: 307777-002US

## 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, Intel386, Intel486, 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.

Copyright © 1998-2006, Intel Corporation.

## Table Of Contents

Compiler Reference .....	1
Introduction to Intel(R) C++ Compiler Reference .....	1
Compiler Limits .....	1
Key Files Summary .....	2
/bin Files .....	2
/include Files .....	3
/lib Files .....	4
Intel(R) C++ Compiler Pragmas .....	5
Overview: Using Intel(R) C++ Compiler Pragmas .....	5
Using Pragmas .....	5
Example .....	5
Intel®-Specific Pragmas .....	6
Other Pragmas .....	6
Predefined Macros .....	7
ANSI Standard Predefined Macros .....	7
Additional Predefined Macros .....	8
Intel(R) Math Library .....	11
Math Functions .....	11
Function List .....	11
Trigonometric Functions .....	14
Hyperbolic Functions .....	19
Exponential Functions .....	22
Special Functions .....	27

Nearest Integer Functions .....	31
Remainder Functions .....	34
Miscellaneous Functions .....	36
Complex Functions.....	41
C99 Macros .....	47
Intel(R) C++ Class Libraries .....	48
Introduction to the Class Libraries .....	48
Overview: Intel® C++ Class Libraries .....	48
Hardware and Software Requirements .....	48
About the Classes .....	49
Details About the Libraries .....	49
C++ Classes and SIMD Operations .....	50
Capabilities.....	53
Integer Vector Classes.....	55
Overview: Integer Vector Classes .....	55
Terms, Conventions, and Syntax .....	56
Rules for Operators .....	57
Assignment Operator .....	59
Logical Operators .....	59
Addition and Subtraction Operators .....	61
Multiplication Operators.....	63
Shift Operators .....	64
Comparison Operators .....	66
Conditional Select Operators .....	67

Debug.....	69
Unpack Operators .....	72
Pack Operators .....	77
Clear MMX(TM) Instructions State Operator.....	78
Integer Functions for Streaming SIMD Extensions .....	78
Conversions Between Fvec and Ivec.....	79
Floating-point Vector Classes .....	80
Floating-point Vector Classes .....	80
Fvec Notation Conventions .....	81
Data Alignment.....	82
Conversions .....	82
Constructors and Initialization .....	82
Arithmetic Operators .....	83
Minimum and Maximum Operators .....	87
Logical Operators .....	88
Compare Operators.....	88
Conditional Select Operators for Fvec Classes.....	91
Cacheability Support Operations.....	93
Debugging .....	94
Load and Store Operators.....	95
Unpack Operators for Fvec Operators .....	96
Move Mask Operator.....	96
Classes Quick Reference .....	96
Programming Example .....	102

Index ..... 105

# Compiler Reference

## Introduction to Intel(R) C++ 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++ 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
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

The following tables list and briefly describe files that are installed with the compiler. The following designations apply:

Label	Meaning
i32	Included with Intel® IA-32 compilers
i32em	Included with Intel® EM64T compilers
i64	Included with Itanium® compilers

### /bin Files

File	Description	i32	i32em	i64
codecov	Code-coverage tool	X	X	X
ias	Itanium Assembler			X
eccvars.sh eccvars.csh	Script to set environment variables			X
iccvars.sh iccvars.csh	Script to set environment variables	X	X	X
ecc ecpc	Scripts that check for license file and call compiler driver			X
icc icpc	Scripts that check for license file and call compiler driver	X	X	X
eccbin ecpcbin	Compiler drivers			X
iccbn icpcbin	Compiler drivers	X	X	X
iccec	Script to start Eclipse*	X		
ecc.cfg ecpc.cfg	Configuration Files			X
icc.cfg icpc.cfg	Configuration Files	X	X	X
map_opts	Compiler Option Mapping tool	X	X	X
mcpcom	Intel® C++ Compiler	X	X	X
prelink		X	X	X
profdcg				X
profmerge	Utility used for Profile Guided Optimizations	X	X	X
proforder	Utility used for Profile Guided Optimizations	X	X	X
profrun	Utility used for Profile Guided Optimizations	X	X	X



profrun.bin	Utility used for Profile Guided Optimizations	X	X	X
pronto_tool		X	X	X
tselect	Test-prioritization tool	X	X	X
uninstall.sh	Compiler uninstall script	X	X	X
xiar	Tool used for Interprocedural Optimizations	X	X	X
xild	Tool used for Interprocedural Optimizations	X	X	X

## /include Files

File	Description	i32	i32em	i64
complex.h	Library for _Complex math functions	X	X	X
dvec.h	SSE 2 intrinsics for Class Libraries	X	X	
emm_func.h	Header file for SSE2 intrinsics (used by emmintrin.h)	X	X	
emmintrin.h	Principal header file for SSE2 intrinsics	X	X	X
float.h	IEEE 754 version of standard float.h	X	X	X
fenv.h		X	X	X
fvec.h	SSE intrinsics for Class Libraries	X	X	X
ia32intrin.h	Header file for IA-32 intrinsics	X	X	
ia64intrin.h	Header file for intrinsics on Itanium-based systems	X	X	X
ia64regs.h	Header file for intrinsics on Itanium-based systems	X	X	X
iso646.h	Standard header file	X	X	X
ivec.h	MMX(TM) instructions intrinsics for Class Libraries	X	X	X
limits.h	Standard header file	X	X	X
math.h	Header file for math library	X	X	X
mathf.h	Principal header file for legacy Intel Math Library	X	X	
mathimf.h	Principal header file for current Intel Math Library	X	X	X
mmintrin.h	Intrinsics for MMX instructions	X	X	X
omp.h	Principal header file OpenMP*	X	X	X
pgouser.h	For use in the instrumentation compilation phase of profile-guided optimizations	X	X	X
pmmmintrin.h	Principal header file SSE3 intrinsics	X	X	
proto.h		X	X	X
sse2mmx.h	Principal header file for Streaming SIMD Extensions 2 intrinsics	X	X	X
stdarg.h	Replacement header for standard stdarg.h	X		X
stdbool.h	Defines _Bool keyword	X	X	X

stddef.h	Standard header file	X	X	X
syslimits.h		X	X	X
tgmath.h	Generic c99 math header	X	X	X
varargs.h	Replacement header for standard varargs.h	X		X
xarg.h	Header file used by stdargs.h and varargs.h	X	X	X
xmm_func.h.h	Header file for Streaming SIMD Extensions	X	X	
xmm_utils.h	Utilities for Streaming SIMD Extensions	X	X	
xmmintrin.h	Principal header file for Streaming SIMD Extensions intrinsics	X	X	X

## /lib Files

File	Description	i32	i32em	i64
libguide.a libguide.so	For OpenMP* implementation	X	X	X
libguide_stats.a libguide_stats.so	OpenMP static library for the parallelizer tool with performance statistics and profile information	X	X	X
libompstub.a	Library that resolves references to OpenMP subroutines when OpenMP is not in use	X	X	X
libsvml.a libsvml.so	Short vector math library	X	X	
libirc.a libirc_s.a	Intel support library for PGO and CPU dispatch	X	X	X
libirc.so	Intel support library for PGO and CPU dispatch		X	
libimf.a libimf.so	Intel math library	X	X	X
libimf.so.6	Intel math library			X
libcprts.a libcprts.so	Dinkumware* C++ Library	X	X	X
libcprts.so.5	Dinkumware* C++ Library	X	X	
libcprts.so.6	Dinkumware* C++ Library			X
libunwind.a libunwind.so	Unwinder library	X	X	X
libunwind.so.5	Unwinder library	X	X	
libunwind.so.6	Unwinder library			X
libcxa.a libcxa.so	Intel run time support for C++ features	X	X	X
libcxa.so.5	Intel run time support for C++ features	X	X	
libcxa.so.6	Intel run time support for C++ features			X

libcxaguard.a libcxaguard.so	Used for interoperability support with the <code>-cxxlib-gcc</code> option.	X	X	X
libcxaguard.so.5	Used for interoperability support with the <code>-cxxlib-gcc</code> option.	X	X	
libcxaguard.so.6	Used for interoperability support with the <code>-cxxlib-gcc</code> option.			X
libipr.a libipr.so libipr.so.6				X

## Intel(R) C++ Compiler Pragmas

### Overview: Using Intel(R) C++ Compiler Pragmas

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

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

### Using Pragmas

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

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

### Example

The `vector always` directive instructs the compiler to override any efficiency heuristic during the decision to vectorize or not, and will vectorize non-unit strides or very unaligned memory accesses.

Example of the `vector always` directive

```
#pragma vector always

for(i=0; i<=N; i++)
{
    a[32*i]=b[99*i];
}
```

## Intel®-Specific Pragmas

The following pragmas are specific to the Intel® C++ Compiler:

Pragma	Description
<code>force_align</code>	specifies the alignment of a class type
<code>ivdep</code>	instructs the compiler to ignore assumed vector dependencies
<code>nounroll</code>	instructs the compiler not to unroll a loop
<code>novector</code>	specifies that the loop should never be vectorized
<code>optimization_level</code>	enables control of optimization for a specific function
<code>intel_omp_task</code>	specifies a unit of work, potentially executed by a different thread
<code>intel_omp_taskq</code>	specifies an environment for the while loop in which to enqueue the units of work specified by the enclosed task pragma
<code>unroll</code>	tells the compiler how many times to unroll a loop
<code>vector</code>	specifies how to vectorize the loop and indicates that efficiency heuristics should be ignored

## Other Pragmas

Please also see Intel®-Specific Pragmas.

The Intel® C++ Compiler supports the following pragmas:

Pragma	Description
<code>alloc_text</code>	names the code section where the specified function definitions are to reside
<code>auto_inline</code>	excludes any function defined within the range where <code>off</code> is specified from being considered as candidates for automatic inline expansion
<code>check_stack</code>	<code>on</code> argument indicates that stack checking should be enabled for functions that follow and <code>off</code> argument indicates that stack checking should be disabled for functions that follow.
<code>code_seg</code>	specifies a code section where functions are to be allocated
<code>comment</code>	places a comment record into an object file or executable file
<code>conform</code>	specifies the run-time behavior of the <code>/Zc:forScope</code> compiler option
<code>stdc</code> <code>cx_limited_range</code>	informs the implementation that the usual formulas are acceptable

<code>data_seg</code>	specifies the default section for initialized data
<code>stdc_fenv_access</code>	informs an implementation that a program may test status flags or run under a non-default control mode
<code>float_control</code>	specifies floating-point behavior for a function
<code>stdc_fp_contract</code>	allows or disallows the implementation to contract expressions
<code>ident</code>	places the string in the comment section of the executable
<code>init_seg</code>	specifies the section to contain C++ initialization code for the translation unit
<code>message</code>	displays the specified string literal to the standard output device
<code>optimize</code>	specifies optimizations to be performed on a function-by-function basis
<code>pointers_to_members</code>	specifies whether a pointer to a class member can be declared before its associated class definition and is used to control the pointer size and the code required to interpret the pointer
<code>pop_macro</code>	sets the value of the <code>macro_name</code> macro to the value on the top of the stack for this macro
<code>push_macro</code>	saves the value of the <code>macro_name</code> macro on the top of the stack for this macro
<code>section</code>	creates a section in an <code>.obj</code> file. Once a section is defined, it remains valid for the remainder of the compilation
<code>vtordisp</code>	<code>on</code> argument enables the generation of hidden <code>vtordisp</code> members and <code>off</code> disables them
<code>warning</code>	allows selective modification of the behavior of compiler warning messages
<code>weak</code>	declares symbol you enter to be weak

## Predefined Macros

### 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
<code>__DATE__</code>	The date of compilation as a string literal in the form <code>Mmm dd yyyy.</code>

<code>__FILE__</code>	A string literal representing the name of the file being compiled.
<code>__LINE__</code>	The current line number as a decimal constant.
<code>__STDC__</code>	The name <code>__STDC__</code> is defined when compiling a C translation unit.
<code>__STDC_HOSTED__</code>	1
<code>__TIME__</code>	The time of compilation as a string literal in the form <code>hh:mm:ss</code> .

## See Also

- Additional Predefined Macros

## Additional Predefined Macros

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

The following designations apply:

Label	Meaning
i32	Supported on Intel® IA-32 compilers
i32em	Supported on Intel® EM64T compilers
i64	Supported on Itanium® compilers

Macro Name	Value	i32	i32em	i64
<code>__ARRAY_OPERATORS</code>	1	X	X	X
<code>__BASE_FILE__</code>	Name of source file	X	X	X
<code>__BOOL</code>	1	X	X	X
<code>__cplusplus</code>	1	X	X	X
<code>__DEPRECATED</code>	1	X	X	X
<code>__EDG__</code>	1	X	X	X
<code>__EDG_VERSION__</code>	304	X	X	X
<code>__ELF__</code>	1	X	X	X
<code>__extension__</code>		X	X	X
<code>__EXCEPTIONS</code>	Defined as 1 when <code>-fno-exceptions</code> is not used.	X	X	X
<code>__GNUC__</code>	2 - if gcc version is less than 3.2 3 - if gcc version is 3.2, 3.3, or 3.4 4 - if gcc version is 4.x	X	X	X
<code>__gnu_linux__</code>	1	X	X	X

__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	X	X	X
__GNUC_PATCHLEVEL__	3 - if gcc version is 3.x	X	X	X
__GXX_ABI_VERSION	102	X	X	X
__HONOR_STD	1	X	X	
__i386	1	X		
__i386__	1	X		
i386	1	X		
__ia64	1			X
__ia64__	1			X
ia64 (deprecated)	1			X
__ICC	910	X	X	
_INTEGRAL_MAX_BITS	64			X
__INTEL_COMPILER	910	X	X	X
__INTEL_COMPILER_BUILD_DATE	YYYYMMDD	X	X	X
__INTEL_CXXLIB_ICC	Defined as 1 when -cxxlib-icc option is specified during compilation.	X		X
__INTEL_RTTI__	Defined as 1 when -fno-rtti is not specified.	X	X	X
__INTEL_STRICT_ANSI__	Defined as 1 when -strict-ansi is specified.	X	X	X
__itanium__	1			X
__linux	1	X	X	X
__linux__	1	X	X	X
linux	1	X	X	X
__LONG_DOUBLE_SIZE__	80	X	X	X
__LONG_MAX__	9223372036854775807L		X	X
__lp64	1			X
__LP64__	1		X	X
_LP64	1		X	X
_MT	1		X	
__MMX__	1		X	

<code>__NO_INLINE__</code>	1	X	X	X
<code>__NO_MATH_INLINES</code>	1	X	X	X
<code>__NO_STRING_INLINES</code>	1	X	X	X
<code>__OPENMP</code>	Defined as 200505 when <code>-openmp</code> is specified.	X	X	X
<code>__OPTIMIZE__</code>	1	X	X	X
<code>__pentium4</code>	1		X	
<code>__pentium4__</code>	1		X	
<code>__PIC__</code>	Defined as 1 when <code>-fPIC</code> is specified.	X	X	X
<code>__pic__</code>	Defined as 1 when <code>-fPIC</code> is specified.	X	X	X
<code>__PGO_INSTRUMENT</code>	Defined as 1 when <code>-prof-gen[x]</code> is specified.	X	X	X
<code>__PLACEMENT_DELETE</code>	1	X	X	X
<code>__PTRDIFF_TYPE__</code>	int on IA-32 long on EM64T long on Itanium architecture	X	X	X
<code>__REGISTER_PREFIX__</code>		X	X	X
<code>__SIGNED_CHARS__</code>	1	X	X	X
<code>__SIZE_TYPE__</code>	unsigned on IA-32 unsigned long on EM64T unsigned long on Itanium architecture	X	X	X
<code>__SSE__</code>	1		X	
<code>__SSE2__</code>	1		X	
<code>__unix</code>	1	X	X	X
<code>__unix__</code>	1	X	X	X
<code>unix</code>	1	X	X	X
<code>__USER_LABEL_PREFIX__</code>		X	X	X
<code>__VERSION__</code>	Intel(R) C++ gcc 3.0 mode	X	X	X
<code>__WCHAR_T</code>	1	X	X	X
<code>__WCHAR_TYPE__</code>	long int on IA-32 int on EM64T int on Itanium architecture	X	X	X
<code>__WINT_TYPE__</code>	unsigned int	X	X	X
<code>__x86_64</code>	1		X	
<code>__x86_64__</code>	1		X	



## See Also

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

## Intel(R) Math Library

### Math Functions

#### Function List

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

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

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

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

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

## Trigonometric Functions

The Intel Math library supports the following trigonometric functions:

### **acos**

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

**errno:** EDOM, for  $|x| > 1$

**Calling interface:**

```
double acos(double x);
long double acosl(long double x);
float acosf(float x);
```

## acosh

**Description:** The `acosh` 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 acosh(double x);
long double acoshl(long double x);
float acoshf(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);
```

## asinh

**Description:** The `asinh` 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 asinh(double x);
long double asinhl(long double x);
float asinhf(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 asinh1(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 atanh1(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 cosh1(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 sinh1(long double x);  
float sinh1f(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 sinhcosh1(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 cbrt1(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  $x \cdot 2^{\text{exp}}$ , 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  $x \cdot 2^y$ , 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  $x \cdot 2^n$ , 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  $x \cdot 2^n$ , 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 erfc1(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  $x \leq 0$

**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  $x \leq 0$

**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  $x \leq 0$

**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 trunc1(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  $x - n \cdot y$  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  $x \text{ REM } y$  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  $x \text{ REM } y$ . 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  $x \leq y$ ).

**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 finitelf(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 islessequal1(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 isnormal1(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 cabs1(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 cacos1(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 cacosh1(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 carg1(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 casin1(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 casinh1(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);
```

## Printed Documentation

```
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(R) C++ Class Libraries

### Introduction to the Class Libraries

#### Overview: Intel® C++ Class Libraries

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

### Hardware and Software Requirements

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

#### Processor Requirements for Use of Class Libraries

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

## About the Classes

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

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

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

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

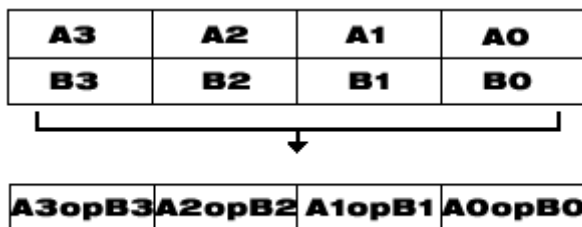
Streaming SIMD Extensions 2 intrinsics cannot be used on Itanium®-based systems. The `mmclass.h` header file includes the classes that are usable on the Itanium architecture.

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

## Details About the Libraries

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

### SIMD Data Flow



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

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

### Comparison Between Inlining, Intrinsics and Class Libraries

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

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

## C++ Classes and SIMD Operations

The use of C++ classes for SIMD operations is based on the concept of operating on arrays, or vectors of data, in parallel. Consider the addition of two vectors, A and B, where each vector contains four elements. Using the integer vector (*Ivec*) class, the elements A[i] and B[i] from each array are summed as shown in the following example.

### Typical Method of Adding Elements Using a Loop

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

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

### SIMD Method of Adding Elements Using Ivec Classes

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

## Available Classes

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

## SIMD Vector Classes



Instruction Set	Class	Signedness	Data Type	Size	Elements	Header File
MMX(TM) technology	I64vec1	unspecified	__m64	64	1	ivec.h
	I32vec2	unspecified	int	32	2	ivec.h
	Is32vec2	signed	int	32	2	ivec.h
	Iu32vec2	unsigned	int	32	2	ivec.h
	I16vec4	unspecified	short	16	4	ivec.h
	Is16vec4	signed	short	16	4	ivec.h
	Iu16vec4	unsigned	short	16	4	ivec.h
	I8vec8	unspecified	char	8	8	ivec.h
	Is8vec8	signed	char	8	8	ivec.h
	Iu8vec8	unsigned	char	8	8	ivec.h
	F32vec4	signed	float	32	4	fvec.h
	F32vec1	signed	float	32	1	fvec.h
Streaming SIMD Extensions 2	F64vec2	signed	double	64	2	dvec.h
	I128vec1	unspecified	__m128i	128	1	dvec.h
	I64vec2	unspecified	long int	64	4	dvec.h
	Is64vec2	signed	long int	64	4	dvec.h
	Iu64vec2	unsigned	long int	32	4	dvec.h
	I32vec4	unspecified	int	32	4	dvec.h
	Is32vec4	signed	int	32	4	dvec.h
	Iu32vec4	unsigned	int	32	4	dvec.h
	I16vec8	unspecified	int	16	8	dvec.h
	Is16vec8	signed	int	16	8	dvec.h
	Iu16vec8	unsigned	int	16	8	dvec.h
	I8vec16	unspecified	char	8	16	dvec.h
	Is8vec16	signed	char	8	16	dvec.h
	Iu8vec16	unsigned	char	8	16	dvec.h

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

**Note**

Intrinsics that take immediate values and cannot be expressed easily in classes are not implemented.

(For example, `_mm_shuffle_ps`, `_mm_shuffle_pi16`, `_mm_extract_pi16`, `_mm_insert_pi16`).

## Access to Classes Using Header Files

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

### Include Directives for Enabling Classes

Instruction Set Extension	Include Directive
MMX Technology	<code>#include &lt;ivec.h&gt;</code>
Streaming SIMD Extensions	<code>#include &lt;fvec.h&gt;</code>
Streaming SIMD Extensions 2	<code>#include &lt;dvec.h&gt;</code>

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.

<code>ivecA = ivecA &amp; ivecB;</code>	ivec logical operation that uses MMX instructions
<code>empty ();</code>	clear state
<code>cout &lt;&lt; f32vec4a;</code>	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

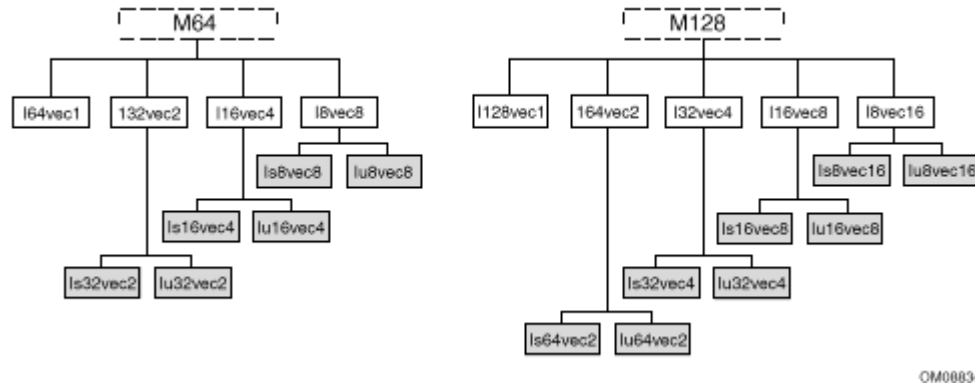
Streaming SIMD Extensions provide prefetching and streaming hints. Prefetching data can minimize the effects of memory latency. Streaming hints allow you to indicate that certain data should not be cached. This results in higher performance for data that should be cached.

## Integer Vector Classes

### Overview: 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



OM00834

The `M64` and `M128` classes define the `__m64` and `__m128i` data types from which the rest of the `Ivec` classes are derived. The first generation of child classes are derived based solely on bit sizes of 128, 64, 32, 16, and 8 respectively for the `I128vec1`, `I64vec1`, `I64vec2`, `I32vec2`, `I32vec4`, `I16vec4`, `I16vec8`, `I8vec16`, and `I8vec8` classes. The latter seven of 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 documentation to address typecasting, operator overloading, and other rules.

### Class Syntax Notation Conventions

Class Name	Description
<code>I [s u] [N] vec [N]</code>	Any value except <code>I128vec1</code> nor <code>I64vec1</code>
<code>I64vec1</code>	<code>_m64</code> data type
<code>I [s u] 64vec2</code>	two 64-bit values of any signedness
<code>I [s u] 32vec4</code>	four 32-bit values of any signedness
<code>I [s u] 8vec16</code>	eight 16-bit values of any signedness
<code>I [s u] 16vec8</code>	sixteen 8-bit values of any signedness
<code>I [s u] 32vec2</code>	two 32-bit values of any signedness
<code>I [s u] 16vec4</code>	four 16-bit values of any signedness
<code>I [s u] 8vec8</code>	eight 8-bit values of any signedness

## Rules for Operators

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

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

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

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

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

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

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

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

`[ Ivec_Class ]` an `Ivec` class

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

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

## Summary of Rules Major Operators

Operators	Sign Typecasting	Size Typecasting	Other Typecasting Requirements
-----------	------------------	------------------	--------------------------------

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

## Data Declaration and Initialization

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

### Declaration and Initialization Data Types for Ivec Classes

Operation	Class	Syntax
Declaration	M128	<code>I128vec1 A; Iu8vec16 A;</code>
Declaration	M64	<code>I64vec1 A; Iu8vec16 A;</code>
<code>__m128</code> Initialization	M128	<code>I128vec1 A(__m128 m); Iu16vec8(__m128 m);</code>
<code>__m64</code> Initialization	M64	<code>I64vec1 A(__m64 m); Iu8vec8 A(__m64 m);</code>
<code>__int64</code> Initialization	M64	<code>I64vec1 A = __int64 m; Iu8vec8 A = __int64 m;</code>
<code>int i</code> Initialization	M64	<code>I64vec1 A = int i; Iu8vec8 A = int i;</code>
<code>int</code> initialization	I32vec2	<code>I32vec2 A(int A1, int A0); Is32vec2 A(signed int A1, signed int A0); Iu32vec2 A(unsigned int A1, unsigned int A0);</code>
<code>int</code> Initialization	I32vec4	<code>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);</code>
<code>short int</code>	I16vec4	<code>I16vec4 A(short A3, short A2, short A1, short</code>



Initialization		<pre>A0); Is16vec4 A(signed short A3, ..., signed short A0); Iu16vec4 A(unsigned short A3, ..., unsigned short A0);</pre>
short int Initialization	I16vec8	<pre>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);</pre>
char Initialization	I8vec8	<pre>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);</pre>
char Initialization	I8vec16	<pre>I8vec16 A(char A15, ..., char A0); Is8vec16 A(signed char A15, ..., signed char A0); Iu8vec16 A(unsigned char A15, ..., unsigned char A0);</pre>

## 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	<code>_mm_and_si64</code> <code>_mm_and_si128</code>
OR		=	R = A   B	R  = A	<code>_mm_and_si64</code> <code>_mm_and_si128</code>

XOR	$\wedge$	$\wedge=$	$R = A \wedge B$	$R \wedge= A$	<code>_mm_and_si64</code> <code>_mm_and_si128</code>
ANDNOT	<code>andnot</code>	N/A	$R = A \text{ andnot } B$	N/A	<code>_mm_and_si64</code> <code>_mm_and_si128</code>

## 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
<code>I64vec1 R</code>	<code>&amp;</code>	<code> </code>	$\wedge$	<code>andnot</code>	<code>I[s u]64vec2 A</code>	<code>I[s u]64vec2 B</code>
<code>I64vec2 R</code>	<code>&amp;</code>	<code> </code>	$\wedge$	<code>andnot</code>	<code>I[s u]64vec2 A</code>	<code>I[s u]64vec2 B</code>
<code>I32vec2 R</code>	<code>&amp;</code>	<code> </code>	$\wedge$	<code>andnot</code>	<code>I[s u]32vec2 A</code>	<code>I[s u]32vec2 B</code>
<code>I32vec4 R</code>	<code>&amp;</code>	<code> </code>	$\wedge$	<code>andnot</code>	<code>I[s u]32vec4 A</code>	<code>I[s u]32vec4 B</code>
<code>I16vec4 R</code>	<code>&amp;</code>	<code> </code>	$\wedge$	<code>andnot</code>	<code>I[s u]16vec4 A</code>	<code>I[s u]16vec4 B</code>
<code>I16vec8 R</code>	<code>&amp;</code>	<code> </code>	$\wedge$	<code>andnot</code>	<code>I[s u]16vec8 A</code>	<code>I[s u]16vec8 B</code>
<code>I8vec8 R</code>	<code>&amp;</code>	<code> </code>	$\wedge$	<code>andnot</code>	<code>I[s u]8vec8 A</code>	<code>I[s u]8vec8 B</code>
<code>I8vec16 R</code>	<code>&amp;</code>	<code> </code>	$\wedge$	<code>andnot</code>	<code>I[s u]8vec16 A</code>	<code>I[s u]8vec16 B</code>

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

## Ivec Logical Operator Overloading with Assignment

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

## Addition and Subtraction Operators

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

### Syntax Usage for Addition and Subtraction Operators

Return nearest common ancestor type, I16vec4.

```
Is16vec4 A;
```

```
Iu16vec4 B;
```

```
I16vec4 C;
```

```
C = A + B;
```

Returns type left-hand operand type.

```
Is16vec4 A;
```

```
Iu16vec4 B;
```

```
A += B;
```

```
B -= A;
```

Explicitly convert B to Is16vec4.

```
Is16vec4 A,C;
```

```
Iu32vec24 B;
```

```
C = A + C;
```

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

## Addition and Subtraction Operators with Corresponding Intrinsics

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

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

## Addition and Subtraction Operator Overloading

Return Value	Available Operators		Right Side Operands	
	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 R	mul_add	Is16vec8 A	Is16vec8 B
Is32vec2 R	mul_high	Is16vec4 A	Is16vec4 B
Is32vec4 R	mul_high	Is16vec8 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 lvec 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	<code>_mm_sll_si64</code> <code>_mm_slli_si64</code> <code>_mm_sll_pi32</code> <code>_mm_slli_pi32</code> <code>_mm_sll_pi16</code> <code>_mm_slli_pi16</code>
Shift Right	>>	R = A >> B R >>= A	<code>_mm_srl_si64</code> <code>_mm_srli_si64</code> <code>_mm_srl_pi32</code> <code>_mm_srli_pi32</code> <code>_mm_srl_pi16</code> <code>_mm_srli_pi16</code> <code>_mm_sra_pi32</code> <code>_mm_srai_pi32</code> <code>_mm_sra_pi16</code> <code>_mm_srai_pi16</code>

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	<code>I64vec1</code>	>> >>=	<< <<=	<code>I64vec1 A;</code>	<code>I64vec1 B;</code>
Logical	<code>I32vec2</code>	>> >>=	<< <<=	<code>I32vec2 A</code>	<code>I32vec2 B;</code>
Arithmetic	<code>Is32vec2</code>	>> >>=	<< <<=	<code>Is32vec2 A</code>	<code>I[s u][N]vec[N] B;</code>
Logical	<code>Iu32vec2</code>	>> >>=	<< <<=	<code>Iu32vec2 A</code>	<code>I[s u][N]vec[N] B;</code>
Logical	<code>I16vec4</code>	>> >>=	<< <<=	<code>I16vec4 A</code>	<code>I16vec4 B</code>
Arithmetic	<code>Is16vec4</code>	>> >>=	<< <<=	<code>Is16vec4 A</code>	<code>I[s u][N]vec[N] B;</code>

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	<code>select_eq</code>	R = <code>select_eq(A, B, C, D)</code>	<code>_mm_cmpeq_pi32</code> <code>_mm_cmpeq_pi16</code> <code>_mm_cmpeq_pi8</code>	<code>_mm_and_si64</code> <code>_mm_or_si64</code> <code>_mm_andnot_si64</code>
Inequality	<code>select_neq</code>	R = <code>select_neq(A, B, C, D)</code>	<code>_mm_cmpeq_pi32</code> <code>_mm_cmpeq_pi16</code> <code>_mm_cmpeq_pi8</code>	
Greater Than	<code>select_gt</code>	R = <code>select_gt(A, B, C, D)</code>	<code>_mm_cmpgt_pi32</code> <code>_mm_cmpgt_pi16</code> <code>_mm_cmpgt_pi8</code>	
Greater Than or Equal To	<code>select_ge</code>	R = <code>select_gt(A, B, C, D)</code>	<code>_mm_cmpge_pi32</code> <code>_mm_cmpge_pi16</code> <code>_mm_cmpge_pi8</code>	
Less Than	<code>select_lt</code>	R = <code>select_lt(A, B, C, D)</code>	<code>_mm_cmplt_pi32</code> <code>_mm_cmplt_pi16</code> <code>_mm_cmplt_pi8</code>	
Less Than or Equal To	<code>select_le</code>	R = <code>select_le(A, B, C, D)</code>	<code>_mm_cmple_pi32</code> <code>_mm_cmple_pi16</code> <code>_mm_cmple_pi8</code>	

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	<code>select_eq</code> <code>select_ne</code>	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	Is32vec2	Is32vec2	Is32vec2
I16vec4 R	select_ge	Is16vec4	Is16vec4	Is16vec4
	select_lt			
I8vec8 R	select_le	Is8vec8	Is8vec8	Is8vec8

The following table shows the mapping of return values from  $R_0$  to  $R_7$  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

## Printed Documentation

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 Intrinsic: 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 Intrinsic: 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 Intrinsic: 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 Intrinsic: 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;
```

## Printed Documentation

```
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**.

## Printed Documentation

```
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;

```

## Printed Documentation

```
R2 = A1;  
R3 = B1;  
R4 = A2;  
R5 = B2;  
R6 = A3;  
R7 = B3;
```

Corresponding intrinsic: `_mm_unpacklo_epi16`

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

```
I16vec4 unpack_low(I16vec4 A, I16vec4 B);  
  
Is16vec4 unpack_low(Is16vec4 A, Is16vec4 B);  
  
Iu16vec4 unpack_low(Iu16vec4 A, Iu16vec4 B);  
  
R0 = A0;  
R1 = B0;  
R2 = A1;  
R3 = B1;
```

Corresponding intrinsic: `_mm_unpacklo_pi16`

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

```
I8vec16 unpack_low(I8vec16 A, I8vec16 B);  
  
Is8vec16 unpack_low(Is8vec16 A, Is8vec16 B);  
  
Iu8vec16 unpack_low(Iu8vec16 A, Iu8vec16 B);  
  
R0 = A0;  
R1 = B0;  
R2 = A1;  
R3 = B1;  
R4 = A2;  
R5 = B2;  
R6 = A3;  
R7 = B3;  
R8 = A4;  
R9 = B4;  
R10 = A5;  
R11 = B5;  
R12 = A6;  
R13 = B6;  
R14 = A7;  
R15 = B7;
```

Corresponding intrinsic: `_mm_unpacklo_epi8`

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

```

I8vec8 unpack_low(I8vec8 A, I8vec8 B);

Is8vec8 unpack_low(Is8vec8 A, Is8vec8 B);

Iu8vec8 unpack_low(Iu8vec8 A, Iu8vec8 B);

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

```

Corresponding intrinsic: `_mm_unpacklo_pi8`

## Pack Operators

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

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

## Clear MMX(TM) Instructions State Operator

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

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

## Integer Functions for Streaming SIMD Extensions



### Note

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
void store_nta(__m64 *p, M64 A);
```

Corresponding intrinsic: `_mm_stream_pi`

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

```
Iu8vec8 simd_avg(Iu8vec8 A, Iu8vec8 B);
```

Corresponding intrinsic: `_mm_avg_pu8`

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

```
Iu16vec4 simd_avg(Iu16vec4 A, Iu16vec4 B);
```

Corresponding intrinsic: `_mm_avg_pu16`

## Conversions Between Fvec and Ivec

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

```
int F64vec2ToInt(F64vec42 A);
```

`r := (int)A0;`

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

```
F64vec2 F32vec4ToF64vec2(F32vec4 A);
```

`r0 := (double)A0;`  
`r1 := (double)A1;`

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

```
F32vec4 F64vec2ToF32vec4(F64vec2 A);
```

`r0 := (float)A0;`  
`r1 := (float)A1;`

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

```
F64vec2 InttoF64vec2(F64vec2 A, int B);
```

`r0 := (double)B;`  
`r1 := A1;`

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

```
int F32vec4ToInt(F32vec4 A);
```

`r := (int)A0;`

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

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

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

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

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

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

## Floating-point Vector Classes

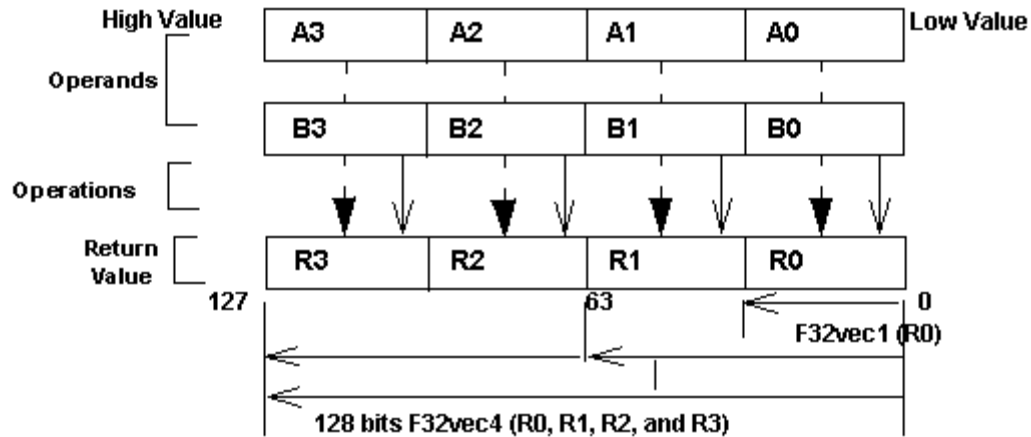
## Floating-point Vector Classes

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:	<code>F32vec4</code>	<code>F64vec2</code>	<code>F32vec1</code>
<code>R0 := A0 &amp; B0;</code>	<code>R0 := A0 andnot B0;</code>	<code>R0 &amp;= A0;</code>	x	x	x
<code>R1 := A1 &amp; B1;</code>	<code>R1 := A1 andnot B1;</code>	<code>R1 &amp;= A1;</code>	x	x	N/A
<code>R2 := A2 &amp; B2;</code>	<code>R2 := A2 andnot B2;</code>	<code>R2 &amp;= A2;</code>	x	N/A	N/A
<code>R3 := A3 &amp; B3;</code>	<code>R3 := A3 andhot B3;</code>	<code>R3 &amp;= A3;</code>	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
<b>__m128 Object Initialization</b>		
F64vec2 A(__m128d mm); F32vec4 B(__m128 mm); F32vec1 C(__m128 mm);	N/A	N/A
<b>Double Initialization</b>		
/* 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;
<b>Float Initialization</b>		
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 * F32vec2 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 value classes of the advanced arithmetic operators, which use the syntax styles described earlier in the Return Value Notation section.

### Advanced Arithmetic Return Value Mapping

R	Operators				A	F32vec4	F64vec2	F32vec1
R0:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A0		
R1:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A1		N/A
R2:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A2	N/A	N/A
R3:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A3	N/A	N/A
f :=	add_horizontal			(A0 + A1 + A2 + A3)			N/A	N/A
d :=	add_horizontal			(A0 + A1)		N/A		N/A

This table shows examples for advanced arithmetic operators.

### Advanced Arithmetic Operations for Fvec Classes

Returns	Example Syntax Usage	Intrinsic
<b>Square Root</b>		
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
<b>Reciprocal</b>		
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
<b>Reciprocal Square Root</b>		
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
<b>Reciprocal Newton Raphson</b>		
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
<b>Reciprocal Square Root Newton Raphson</b>		
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
<b>Horizontal Add</b>		
1 float	float f = add_horizontal(F32vec4 A);	_mm_add_ss _mm_shuffle_ss

1 double	double d = add_horizontal(F64vec2 A);	<code>_mm_add_sd</code> <code>_mm_shuffle_sd</code>
----------	---------------------------------------	--

## 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_max(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	<code>&amp;</code> <code>&amp;=</code>	<code>R = A &amp; B;</code> <code>R &amp;= A;</code>
OR	<code> </code> <code> =</code>	<code>R = A   B;</code> <code>R  = A;</code>
XOR	<code>^</code> <code>^=</code>	<code>R = A ^ B;</code> <code>R ^= A;</code>
andnot	<code>andnot</code>	<code>R = andnot (A) ;</code>

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	<code>F32vec4 &amp; = F32vec4 A &amp; F32vec4 B;</code> <code>F32vec4 &amp; &amp;= F32vec4 A;</code>	<code>_mm_and_ps</code>
	2 doubles	<code>F64vec2 R = F64vec2 A &amp; F32vec2 B;</code> <code>F64vec2 R &amp;= F64vec2 A;</code>	<code>_mm_and_pd</code>
	1 float	<code>F32vec1 R = F32vec1 A &amp; F32vec1 B;</code> <code>F32vec1 R &amp;= F32vec1 A;</code>	<code>_mm_and_ps</code>
OR	4 floats	<code>F32vec4 R = F32vec4 A   F32vec4 B;</code> <code>F32vec4 R  = F32vec4 A;</code>	<code>_mm_or_ps</code>
	2 doubles	<code>F64vec2 R = F64vec2 A   F32vec2 B;</code> <code>F64vec2 R  = F64vec2 A;</code>	<code>_mm_or_pd</code>
	1 float	<code>F32vec1 R = F32vec1 A   F32vec1 B;</code> <code>F32vec1 R  = F32vec1 A;</code>	<code>_mm_or_ps</code>
XOR	4 floats	<code>F32vec4 R = F32vec4 A ^ F32vec4 B;</code> <code>F32vec4 R ^= F32vec4 A;</code>	<code>_mm_xor_ps</code>
	2 doubles	<code>F64vec2 R = F64vec2 A ^ F32vec2 B;</code> <code>F64vec2 R ^= F64vec2 A;</code>	<code>_mm_xor_pd</code>
	1 float	<code>F32vec1 R = F32vec1 A ^ F32vec1 B;</code> <code>F32vec1 R ^= F32vec1 A;</code>	<code>_mm_xor_ps</code>
ANDNOT	2 doubles	<code>F64vec2 R = andnot (F64vec2 A,</code> <code>F64vec2 B) ;</code>	<code>_mm_andnot_pd</code>

## 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	<code>cmpeq</code>	<code>R = cmpeq(A, B)</code>
Inequality	<code>cmpneq</code>	<code>R = cmpneq(A, B)</code>
Greater Than	<code>cmpgt</code>	<code>R = cmpgt(A, B)</code>
Greater Than or Equal To	<code>cmpge</code>	<code>R = cmpge(A, B)</code>
Not Greater Than	<code>cmpngt</code>	<code>R = cmpngt(A, B)</code>
Not Greater Than or Equal To	<code>cmpnge</code>	<code>R = cmpnge(A, B)</code>
Less Than	<code>cmplt</code>	<code>R = cmplt(A, B)</code>
Less Than or Equal To	<code>cmple</code>	<code>R = cmple(A, B)</code>
Not Less Than	<code>cmpnlt</code>	<code>R = cmpnlt(A, B)</code>
Not Less Than or Equal To	<code>cmpnle</code>	<code>R = cmpnle(A, B)</code>

## 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   ng   nge]	B1) B1)	0xffffffff	0x00000000	X	X	X
R1:=	(A1 !(A1	cmp[eq   lt   le   gt   ge] cmp[ne   nlt   nle   ng   nge]	B2) B2)	0xffffffff	0x00000000	X	X	N/A
R2:=	(A1 !(A1	cmp[eq   lt   le   gt   ge] cmp[ne   nlt   nle   ng   nge]	B3) B3)	0xffffffff	0x00000000	X	N/A	N/A
R3:=	A3	cmp[eq   lt   le   gt   ge] cmp[ne   nlt   nle	B3) B3)	0xffffffff	0x00000000	X	N/A	N/A

		ngt   nge]						
--	--	------------	--	--	--	--	--	--

The following table shows examples for arithmetic operators and intrinsics.

## Compare Operations for Fvec Classes

Returns	Example Syntax Usage	Intrinsic
<b>Compare for Equality</b>		
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
<b>Compare for Inequality</b>		
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
<b>Compare for Less Than</b>		
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
<b>Compare for Less Than or Equal</b>		
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_ss
<b>Compare for Greater Than</b>		
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
<b>Compare for Greater Than or Equal To</b>		
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
<b>Compare for Not Less Than</b>		
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
<b>Compare for Not Less Than or Equal</b>		
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
<b>Compare for Not Greater Than</b>		
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
<b>Compare for Not Greater Than or Equal</b>		
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 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	X X	N/A N/A
R2:=	(A2 !(A2	select_[eq   lt   le   gt   ge] select_[ne   nlt   nle   ngt   nge]	B2) B2)	C2 C2	D2 D2	X X	N/A N/A	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 X	N/A N/A	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
<b>Compare for Equality</b>		
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
<b>Compare for Inequality</b>		
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
<b>Compare for Less Than</b>		
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
<b>Compare for Less Than or Equal</b>		
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_ss

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	F32vec4 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	F32vec4 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	F32vec4 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	F32vec4 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	F32vec4 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,	I64vec, I32vec,	F64vec2	F32vec4	F32vec1

		<b>I32vec4, I16vec8, I8vec16</b>	<b>I16vec, I8vec8</b>			
&, &=	_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	<b>I64vec2</b>	<b>I32vec4</b>	<b>I16vec8</b>	<b>I8vec16</b>
+, +=	_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	<b>I32vec2</b>	<b>I16vec4</b>	<b>I8vec8</b>	<b>F64vec2</b>	<b>F32vec4</b>	<b>F32vec1</b>
+, +=	_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	l128vec1	l64vec2	l32vec4	l16vec8	l8vec16
>>, >>=	_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	l64vec1	l32vec2	l16vec4	l8vec8
>>, >>=	_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	l32vec4	l16vec8	l8vec16	l32vec2	l16vec4	l8vec8
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_cmnpnlt_[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_cmnpnlt_[x]	pd	ps	ss
cmpnle	_mm_cmpnle_[x]	pd	ps	ss

## Conditional Select Operators: Corresponding Intrinsics and Classes, Part 1

Operators	Corresponding Intrinsic	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8
select_eq	_mm_cmpeq_[x]	epi32	epi16	epi8	pi32	pi16	pi8
	_mm_and_[y]	si128	si128	si128	si64	si64	si64
	_mm_andnot_[y] *	si128	si128	si128	si64	si64	si64
	_mm_or_[y]	si128	si128	si128	si64	si64	si64
select_neq	_mm_cmpeq_[x]	epi32	epi16	epi8	pi32	pi16	pi8
	_mm_and_[y]	si128	si128	si128	si64	si64	si64
	_mm_andnot_[y] *	si128	si128	si128	si64	si64	si64
	_mm_or_[y]	si128	si128	si128	si64	si64	si64
select_gt	_mm_cmpgt_[x]	epi32	epi16	epi8	pi32	pi16	pi8
	_mm_and_[y]	si128	si128	si128	si64	si64	si64
	_mm_andnot_[y] *	si128	si128	si128	si64	si64	si64
	_mm_or_[y]	si128	si128	si128	si64	si64	si64

select_ge	_mm_cmpge_[x]	epi32	epi16	epi8	pi32	pi16	pi8
	_mm_and_[y]	si128	si128	si128	si64	si64	si64
	_mm_andnot_[y] *	si128	si128	si128	si64	si64	si64
	_mm_or_[y]	si128	si128	si128	si64	si64	si64
select_lt	_mm_cmplt_[x]	epi32	epi16	epi8	pi32	pi16	pi8
	_mm_and_[y]	si128	si128	si128	si64	si64	si64
	_mm_andnot_[y] *	si128	si128	si128	si64	si64	si64
	_mm_or_[y]	si128	si128	si128	si64	si64	si64
select_le	_mm_cmple_[x]	epi32	epi16	epi8	pi32	pi16	pi8
	_mm_and_[y]	si128	si128	si128	si64	si64	si64
	_mm_andnot_[y] *	si128	si128	si128	si64	si64	si64
	_mm_or_[y]	si128	si128	si128	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]	pd	ps	ss
	_mm_and_[y]			
	_mm_andnot_[y] *			
	_mm_or_[y]			
select_neq	_mm_cmpeq_[x]	pd	ps	ss
	_mm_and_[y]			
	_mm_andnot_[y] *			
	_mm_or_[y]			
select_gt	_mm_cmpgt_[x]	pd	ps	ss
	_mm_and_[y]			
	_mm_andnot_[y] *			
	_mm_or_[y]			
select_ge	_mm_cmpge_[x]	pd	ps	ss
	_mm_and_[y]			
	_mm_andnot_[y] *			
	_mm_or_[y]			
select_lt	_mm_cmplt_[x]	pd	ps	ss
	_mm_and_[y]			
	_mm_andnot_[y] *			
	_mm_or_[y]			
select_le	_mm_cmple_[x]	pd	ps	ss
	_mm_and_[y]			
	_mm_andnot_[y] *			
	_mm_or_[y]			
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_cvttssd_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

## A

acos library function .....	14
acosd library function .....	14
acosh library function .....	19
annuity library function .....	26
asin library function .....	14
asind library function .....	14
asinh library function .....	19
atan library function.....	14
atan2 library function.....	14
atand library function.....	14
atand2 library function.....	14
atanh library function.....	19

## C

cabs library function .....	41
cacos library function .....	41
cacosh library function .....	41
carg library function.....	41
casin library function .....	41
casinh library function .....	41
catan library function .....	41
catanh library function .....	41
cbrt library function.....	21

ccos library function .....	41
ccosh library function .....	41
ceil library function .....	30
cexp library function.....	41
cexp10 library function.....	41
cimag library function.....	41
cis library function .....	41
class libraries	
floating-point vector classes.....	79, 80, 81, 82, 85, 86, 87, 90, 92, 94, 95, 100
integer vector classes ..	54, 55, 56, 58, 59, 60, 62, 63, 65, 66, 68, 71, 76, 77, 78
class libraries .....	48, 49, 52
clog library function.....	41
clog2 library function.....	41
compound library function .....	26
conj library function.....	41
copysign library function .....	35
cos library function.....	14
cosd library function.....	14
cosh library function.....	19
cot library function .....	14
cotd library function .....	14
cpow library function .....	41

cproj library function .....	41
creal library function .....	41
csin library function .....	41
csinh library function .....	41
csqrt library function .....	41
ctan library function .....	41
ctanh library function .....	41

## E

erf library function.....	26
erfc library function.....	26
exp library function .....	21
exp10 library function .....	21
exp2 library function .....	21
expm1 library function .....	21

## F

fabs library function .....	35
fdim library function .....	35
finite library function .....	35
floating-point vector classes.....	79
floor library function.....	30
fma library function.....	35
fmax library function .....	35
fmin library function .....	35
fmod library function.....	34

frexp library function .....	21
fvec .....	79

## G

gamma library function .....	26
gamma_r library function .....	26

## H

hypot library function .....	21
------------------------------	----

## I

ilogb library function.....	21
integer vector classes .....	54
Intel math library .. 14, 19, 21, 26, 30, 34, 35, 41	
isnan library function.....	35
ivec	
floating-point .....	79
integer .....	54
ivec .....	54

## J

j0 library function .....	26
j1 library function .....	26
jn library function .....	26

## L

ldexp library function.....	21
lgamma library function .....	26
lgamma_r library function .....	26



llrint library function .....	30	novector .....	5
llround library function .....	30	optimization_level .....	5
log library function .....	21	pointer_size .....	5
log10 library function .....	21	pointers_to_members .....	5
log1p library function .....	21	restrict .....	5
log2 library function .....	21	task .....	5
logb library function .....	21	taskq .....	5
lrint library function .....	30	unroll_count .....	5
lround library function .....	30	using with vectorization .....	5
<b>M</b>		vector .....	5
modf library function .....	30	warning .....	5
<b>N</b>		pragmas .....	5
nearbyint library function .....	30	pragmas non-Intel	
nextafter library function .....	35	alloc_text .....	6
nexttoward library function .....	35	auto_inline .....	6
<b>P</b>		check_stack .....	6
pow library function .....	21	code_and_data_seg .....	6
pragmas		comment .....	6
alloc_section .....	5	ident .....	6
capturedprivate .....	5	init_seg .....	6
conform .....	5	message .....	6
force_align .....	5	optimize .....	6
ivdep .....	5	pointer_size .....	6
no_unroll_count .....	5	pop_macro .....	6

push_macro.....	6	sincosd library function .....	14
section .....	6	sind library function.....	14
vtordisp .....	6	sinh library function.....	19
weak .....	6	sinhcosh library function .....	19
pragmas non-Intel .....	6	sqrt library function .....	21
<b>R</b>		<b>T</b>	
remainder library function.....	34	tan library function .....	14
remquo library function.....	34	tand library function .....	14
rint library function.....	30	tanh library function .....	19
round library function.....	30	tgamma library function .....	26
<b>S</b>		trunc library function .....	30
scalb library function .....	21	<b>Y</b>	
scalbln library function.....	21	y0 library function .....	26
scalbn library function .....	21	y1 library function .....	26
sin library function .....	14	yn library function .....	26
sincos library function.....	14		