Welcome to the Intel® C++ Compiler

Welcome to the Intel® C++ Compiler. Before you use the compiler, see System Requirements.

Most Linux* distributions include the GNU* C library, assembler, linker, and others. The Intel C++ Compiler includes the Dinkumware* C++ library. See Libraries Overview.

Please look at the individual sections within each main section of this User's Guide to gain an overview of the topics presented. For the latest information, visit the Intel Web site: http://developer.intel.com/.

Disclaimer

What's New in This Release

For a complete list of new features, see the Release Notes.

Features and Benefits

The Intel® C++ Compiler allows your software to perform best on computers based on the Intel architecture. Using new compiler optimizations, such as profile-guided optimization, prefetch instruction and support for Streaming SIMD Extensions (SSE) and Streaming SIMD Extensions 2 (SSE2), the Intel C++ Compiler provides high performance.

Feature	**Benefit**
High Performance	Achieve a significant performance gain by using optimizations
Support for Streaming SIMD Extensions	Advantage of Intel microarchitecture
Automatic vectorizer	Advantage of SIMD parallelism in your code achieved automatically
OpenMP* Support	Shared memory parallel programming
Floating-point optimizations	Improved floating-point performance
Data prefetching	Improved performance due to the accelerated data delivery
Interprocedural optimizations	Larger application modules perform better
Profile-guided optimization	Improved performance based on profiling frequently-used functions
Processor dispatch	Taking advantage of the latest Intel architecture features while maintaining object code compatibility with previous generations of Intel® Pentium® processors (for IA-32-based systems only).

Product Web Site and Support

For the latest information about Intel® C++ Compiler, visit http://developer.intel.com/software/products/

For specific details on the Itanium® architecture, visit the web site at http://developer.intel.com/design/itanium/under_lnx.htm.

System Requirements

IA-32 Processor System Requirements

- A computer based on a Pentium® processor or subsequent IA-32 based processor (Pentium 4 processor recommended).
- 128 MB of RAM (256 MB recommended).
- 100 MB of disk space.

Itanium® Processor System Requirements

- A computer with an Itanium processor.
- 256 MB of RAM.
- 100 MB of disk space.

Software Requirements

For a complete list of system requirements, see the Release Notes.

FLEXlm* Electronic Licensing

The Intel® C++ Compiler uses GlobeTrotter FLEXlm* licensing technology. The compiler requires a valid license file in the `/licenses` directory in the installation path. The default directory is `/opt/intel/licenses`. The license files have a `.lic` file extension.

If you require a counted license, see "Using the Intel® License Manager for FLEXlm*" (`flex_ug.pdf`).

How to Use This Document

This User's Guide explains how you can use the Intel® C++ Compiler. It provides information on how to get started with the Intel C++ Compiler, how this compiler operates and what capabilities it offers for high performance. You learn how to use the standard and advanced compiler optimizations to gain maximum performance for your application.

This documentation assumes that you are familiar with the C and C++ programming languages and with the Intel processor architecture. You should also be familiar with the host computer's operating system.

Note

This document explains how information and instructions apply differently to each targeted architecture. If there is no specific indication to either architecture, the description is applicable to both architectures.

Conventions

This documentation uses the following conventions:

`This type style`	Indicates an element of syntax, reserved word, keyword, filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant.
`This type style`	Indicates the exact characters you type as input.
`This type style`	Indicates a placeholder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder.
`[ items ]`	Indicates that the items enclosed in brackets are optional.
`{ item1 | item2 |... }`	Indicates to elect one of the items listed between braces. A vertical bar (|) separates the items. Some options, such as `-ax {M|i|K|M}`, permit the use of more than one `item`.
`... (ellipses)`	Indicate that you can repeat the preceding item.

Naming Syntax for the Intrinsics

Most intrinsic names use a notational convention as follows:

`_mm_<intrin_op>_<suffix>`

<intrin_op>	Indicates the intrinsics basic operation; for example, `add` for addition and `sub` for subtraction.
<suffix>	Denotes the type of data operated on by the instruction. The first one or two letters of each suffix denotes whether the data is packed (`p`), extended packed (`ep`), or scalar (`s`). The remaining letters denote the type: • __s single-precision floating point • __d double-precision floating point • __`i128` signed 128-bit integer • __`i64` signed 64-bit integer • __u64 unsigned 64-bit integer • __`i32` signed 32-bit integer • __u32 unsigned 32-bit integer • __`i16` signed 16-bit integer • __u16 unsigned 16-bit integer • __`i8` signed 8-bit integer • __u8 unsigned 8-bit integer

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

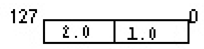
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

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

The result is the same as either of the following:

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

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



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

Naming Syntax for the Class Libraries

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

Related Publications

The following documents provide additional information relevant to the Intel® C++ Compiler:

- ISO/IEC 9989:1990, Programming Languages--C
- ISO/IEC 14882:1998, Programming Languages--C++.
- *The Annotated C++ Reference Manual,* 3rd edition, Ellis, Margaret; Stroustrup, Bjarne, Addison Wesley, 1991. Provides information on the C++ programming language.
- *The C++ Programming Language*, 3rd edition, 1997: Addison-Wesley Publishing Company, One Jacob Way, Reading, MA 01867.
- *The C Programming Language*, 2nd edition, Kernighan, Brian W.; Ritchie, Dennis W., Prentice Hall, 1988. Provides information on the K & R definition of the C language.
- *C: A Reference Manual*, 3rd edition, Harbison, Samual P.; Steele, Guy L., Prentice Hall, 1991. Provides information on the ANSI standard and extensions of the C language.
- *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, Intel Corporation, doc. number 243190.
- *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual*, Intel Corporation, doc. number 243191.
- *Intel Architecture Software Developer's Manual, Volume 3: System Programming*, Intel Corporation, doc. number 243192.
- *Intel® Itanium® Assembler User's Guide*.
- *Intel® Itanium®-based Assembly Language Reference Manual*.
- *Itanium® Architecture Software Developer's Manual Vol. 1: Application Architecture*, Intel Corporation, doc. number 245317-001.
- *Itanium® Architecture Software Developer's Manual Vol. 2: System Architecture*, Intel Corporation, doc. number 245318-001.
- *Itanium® Architecture Software Developer's Manual Vol. 3: Instruction Set Reference*, Intel Corporation, doc. number 245319-001.
- *Itanium® Architecture Software Developer's Manual Vol. 4: Itanium® Processor Programmer's Guide*, Intel Corporation, doc. number 245319-001.
- *Intel Architecture Optimization Manual*, Intel Corporation, doc. number 245127.
- *Intel Processor Identification with the CPUID Instruction*, Intel Corporation, doc. number 241618.
- *Intel Architecture MMX(TM) Technology Programmer's Reference Manual*, Intel Corporation, doc. number 241618.
- *Pentium® Pro Processor Developer's Manual* (3-volume Set), Intel Corporation, doc. number 242693.
- *Pentium®* II *Processor Developer's Manual*, Intel Corporation, doc. number 243502-001.
- *Pentium® Processor Specification Update*, Intel Corporation, doc. number 242480.
- *Pentium® Processor Family Developer's Manual*, Intel Corporation, doc. numbers 241428-005.

Most Intel documents are also available from the Intel Corporation Web site at http://www.intel.com.

Overview: Options Quick Reference Guides

Conventions Used in the Options Quick Guide Tables

Convention	Definition
`[-]`	If an option includes "`[-]`" as part of the definition, then the option can be used to enable or disable the feature. For example, the `-c99[-]` option can be used as `-c99` (enable c99 support) or `-c99-` (disable c99 support).
`[n]`	Indicates that the value n in `[]` can be omitted or have various values.
Values in `{}` with vertical bars	Are used for option's version; for example, option `-i{2|4|8}` has these versions: `-i2`, `-i4`, `-i8`.
{n}	Indicates that option must include one of the fixed values for n.
Words in *`this style`* following an `option`	Indicate option's required argument(s). Arguments are separated by comma if more than one are required.

New Options

- Options specific to IA-32 architecture
- Options specific to the Itanium® architecture (Itanium-based systems only)

All other options are supported on both IA-32 and Itanium-based systems.

Option	Description	Default
`-dM`	Output macro definitions in effect after preprocessing (use with `-E`).	OFF
`-dynamic-linker`*`filename`*	Selects a dynamic linker (*`filename`*) other than the default.	OFF
`-fno-rtti`	Disable RTTI support.	OFF
`-fnsplit[-]` Itanium-based systems only	Enables [disables] function splitting. Default is ON with `-prof_use`. To disable function splitting when you use `-prof_use`, also specify `-fnsplit-`.	OFF
`-fshort-enums`	Allocate as many bytes as needed for enumerated types.	OFF
`-fsyntax-only`	Same as `-syntax`.	OFF
`-funsigned-char`	Change default `char` type to `unsigned`.	OFF
`-funsigned-bitfields`	Change default `bitfield` type to `unsigned`.	OFF
`-idirafter`*`dir`*	Add directory (*`dir`*) to the second include file search path (after `-I`).	OFF
`-march=`*`cpu`* IA-32 only	Generate code excusively for a given *`cpu`*. Values for *`cpu`* are: • `pentiumpro` - Pentium® Pro and Pentium II processor instructions. • `pentiumii` - MMX(TM) instructions. • `pentiumiii` - Streaming SIMD extensions. • `pentium4` - Pentium 4 instructions (default).	OFF

`-mcpu=`*`cpu`*	Optimize for a specific *cpu*. Values for *cpu* are: • `pentium` - Optimize for Pentium processor. • `pentiumpro` - Optimize for Pentium Pro, Pentium II and Pentium III processors. • `pentium4` - Optimize for Pentium 4 processor. For Itanium-based Systems, *cpu* values are: • `itanium` - Optimize for Itanium processor. • `itanium2` - Optimize for Itanium 2 processor (Default).	ON `pentium` on IA-32 `itanium2` on Itanium-based Systems
`-MD`	Preprocess and compile. Generate output file (`.d` extension) containing dependency information.	OFF
`-MF`*`file`*	Generate makefile dependency information in *`file`*. Must specify `-M` or `-MM`.	OFF
`-MG`	Similar to `-M`, but treats missing header files as generated files.	OFF
`-MM`	Similar to `-M`, but does not include system header files.	OFF
`-MMD`	Similar to `-MD`, but does not include system header files.	OFF
`-MX`	Generate dependency file (`.o.dep` extension) containing information used for the Intel wb tool.	OFF
`-mrelax`	Pass `-relax` to the linker.	ON
`-mno-relax`	Do not pass `-relax` to the linker.	OFF
`-mserialize-volatile` Itanium-based systems only	Impose strict memory access ordering for volatile data object references.	OFF
`-mno-serialize-volatile` Itanium-based systems only	The compiler may suppress both run-time and compile-time memory access ordering for volatile data object references. Specifically, the `.rel/.acq` completers will not be issued on referencing loads and stores.	OFF
`-nodefaultlibs`	Do not use standard libraries when linking.	OFF

`-Ob`*`n`*	Controls the compiler's inline expansion. The amount of inline expansion performed varies with the value of n as follows: • `0`: Disables inlining. • `1`: Enables (default) inlining of functions declared with the `__inline` keyword. Also enables inlining according to the C++ language. • `2`: Enables inlining of any function. However, the compiler decides which functions to inline. Enables interprocedural optimizations and has the same effect as `-ip`.	ON
`-openmp_stubs`	Enables OpenMP* programs to compile in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked sequentially.	OFF
`-std=c99` IA-32 only	Enable C99 support for C programs.	ON
`-tpp1` Itanium-based systems only	Target optimization to the Itanium processor.	OFF
`-tpp2` Itanium-based systems only	Target optimization to Itanium® 2 processor. Generated code is compatible with the Itanium processor.	ON
`-v`	Show driver tool commands and execute tools.	OFF
`-Wall`	Enable all warnings.	OFF
`-Werror`	Force warnings to be reported as errors.	OFF

Compiler Options Quick Reference Guide

This topic provides you with a reference to all the compiler options and some linker control options.

- Options specific to IA-32 architecture
- Options specific to the Itanium® architecture

All other options are supported on both IA-32 and Itanium-based systems.

Option	Description	Default
`-0f_check` IA-32 only	Avoids the incorrect decoding of certain `0f` instructions for code targeted at older processors.	OFF
`-A-`	Disables all predefined macros.	OFF
`-[no]align` IA-32 only	Analyze and reorder memory layout for variables and arrays.	OFF
`-A`*`name`*`[(`*`value`*`)]`	Associates a symbol *`name`* with the specified sequence of *`value`*. Equivalent to an `#assert` preprocessing directive.	OFF
`-ansi`	Select strict ANSI C/C++ conformance dialect.	OFF
`-ansi_alias[-]`	`-ansi_alias` directs the compiler to assume the following: • Arrays are not accessed out of bounds. • Pointers are not cast to non-pointer types, and vice-versa. • References to objects of two different scalar types cannot alias. For example, an object of type `int` cannot alias with an object of type `float`, or an object of type `float` cannot alias with an object of type `double`. If your program satisfies the above conditions, setting the `-ansi_alias` flag will help the compiler better optimize the program. However, if your program does not satisfy one of the above conditions, the `-ansi_alias` flag may lead the compiler to generate incorrect code.	OFF

`-ax{M|i|K|W}` IA-32 only	Generates specialized code for processor-specific codes `M`, `i`, `K`, `W` while also generating generic IA-32 code. • `M` = Intel® Pentium® processors with MMX(TM) technology • `i` = Intel Pentium Pro and Intel Pentium II processors • `K` = Intel Pentium III processors • `W` = Intel Pentium 4 processors, Intel® Xeon(TM) processors, and Intel® Pentium® M processors	OFF
`-C`	Places comments in preprocessed source output.	OFF
`-c`	Stops the compilation process after an object file has been generated. The compiler generates an object file for each C or C++ source file or preprocessed source file. Also takes an assembler file and invokes the assembler to generate an object file.	OFF
`-c99[-]`	Enables [disables] C99 support for C programs.	ON
`-complex_limited_range`	This option causes the compiler to use the highest performance formulations of complex arithmetic operations, which may not produce acceptable results for input values near the top or bottom of the legal range. Without this option, the compiler users a better formulation of complex arithmetic operations which produces acceptable results for the full range of input values, at some loss in performance.	OFF
`-dM`	Output macro definitions in effect after preprocessing (use with `-E`).	OFF
`-D`*`name`*`[=`*`value`*`]`	Defines a macro *`name`* and associates it with the specified *`value`*. Equivalent to a `#define` preprocessor directive.	OFF
`-dryrun`	Show driver tool commands but do not execute tools.	OFF
`-dynamic-linker`*`filename`*	Selects a dynamic linker (*`filename`*) other than the default.	OFF
`-E`	Stops the compilation process after the C or C++ source files have been preprocessed, and writes the results to stdout.	OFF

`-EP`	Preprocess to stdout omitting `#line` directives.	OFF
`-falias`	Assume aliasing in program.	ON
`-fcode-asm`	Produce assemblable file with optional code annotations. Requires -S.	OFF
`-fno-alias`	Assume no aliasing in program.	OFF
`-ffnalias`	Assume aliasing within functions	ON
`-fno-fnalias`	Assume no aliasing within functions, but assume aliasing across calls.	OFF
`-fno-rtti`	Disable RTTI support.	OFF
`-f[no]verbose-asm`	Produce assemblable file with compiler comments.	ON
`-fnsplit[-]` Itanium-based systems only	Enables [disables] function splitting. Default is ON with `-prof_use`. To disable function splitting when you use `-prof_use`, also specify `-fnsplit-`.	OFF
`-fp` IA-32 only	Disable using the EBP register as general purpose register.	OFF
`-fpic, -fPIC`	Generate position independent code.	OFF
`-fp_port` IA-32 only	Round fp results at assignments and casts. Some speed impact.	OFF
`-fr32` Itanium-based systems only	Use only lower 32 floating-point registers.	OFF
`-fshort-enums`	Allocate as many bytes as needed for enumerated types.	OFF
`-fsource-asm`	Produce assemblable file with optional code annotations. Requires -S.	OFF
`-fsyntax-only`	Same as `-syntax`.	OFF
`-ftz[-]` Itanium-based systems only	Flushes denormal results to zero. The option is turned ON with `-O3`.	OFF
`-funsigned-char`	Change default `char` type to `unsigned`.	OFF
`-funsigned-bitfields`	Change default bitfield type to `unsigned`.	OFF
`-g`	Generates symbolic debugging information in the object code for use by source-level debuggers.	OFF

`-H`	Print "include" file order and continue compilation.	OFF
`-help`	Prints compiler options summary.	OFF
`-idirafter`*`dir`*	Add directory (*`dir`*) to the second include file search path (after `-I`).	OFF
`-I`*`directory`*	Specifies an additional *`directory`* to search for include files.	OFF
`-i_dynamic`	Link Intel provided libraries dynamically.	OFF
`-inline_debug_info`	Preserve the source position of inlined code instead of assigning the call-site source position to inlined code.	OFF
`-ip`	Enables interprocedural optimizations for single file compilation.	OFF
`-IPF_fma[-]` Itanium-based systems only	Enable [disable] the combining of floating-point multiplies and add/subtract operations.	OFF
`-IPF_fltacc[-]` Itanium-based systems only	Enable [disable] optimizations that affect floating-point accuracy.	OFF
`-IPF_flt_eval_method0` Itanium-based systems only	Floating-point operands evaluated to the precision indicated by the program.	OFF
`-IPF_fp_speculation`*`mode`* Itanium-based systems only	Enable floating-point speculations with the following *`mode`* conditions: • `fast` - speculate floating-point operations • `safe` - speculate only when safe • `strict` - same as off • `off` - disables speculation of floating-point operations	OFF
`-ip_no_inlining`	Disables inlining that would result from the `-ip` interprocedural optimization, but has no effect on other interprocedural optimizations.	OFF
`-ip_no_pinlining` IA-32 only	Disable partial inlining. Requires `-ip` or `-ipo`.	OFF
`-ipo`	Enables interprocedural optimizations across files.	OFF
`-ipo_c`	Generates a multifile object file (`ipo_out.o`) that can be used in further link steps.	OFF

`-ipo_obj`	Forces the compiler to create real object files when used with `-ipo`.	OFF
`-ipo_S`	Generates a multifile assemblable file named `ipo_out.s` that can be used in further link steps.	OFF
`-ivdep_parallel` Itanium-based systems only	This option indicates there is absolutely no loop-carried memory dependency in the loop where IVDEP directive is specified.	OFF
`-Kc++`	Compile all source or unrecognized file types as C++ source files.	ON (for `icpc/ecpc`)
`-Knopic`, `-KNOPIC` Itanium-based systems only	Don't generate position independent code.	OFF
`-KPIC`, `-Kpic`	Generate position independent code.	OFF for IA-32 ON for Itanium-based systems
`-L`*`directory`*	Instruct linker to search *`directory`* for libraries.	OFF
`-long_double` IA-32 only	Changes the default size of the long double data type from 64 to 80 bits.	OFF
`-M`	Generates makefile dependency lines for each source file, based on the `#include` lines found in the source file.	OFF
`-march=`*`cpu`* IA-32 only	Generate code excusively for a given *`cpu`*. Values for *`cpu`* are: • `pentiumpro` - Pentium® Pro and Pentium II processor instructions. • `pentiumii` - MMX(TM) instructions. • `pentiumiii` - Streaming SIMD extensions. • `pentium4` - Pentium 4 instructions.	OFF

`-mcpu=`*`cpu`*	Optimize for a specific *`cpu`*. For IA-32, *`cpu`* values are: • `pentium` - Optimize for Pentium processor. • `pentiumpro` - Optimize for Pentium Pro, Pentium II and Pentium III processors. • `pentium4` - Optimize for Pentium 4 processor (Default). For Itanium-based Systems, *`cpu`* values are: • `itanium` - Optimize for Itanium processor. • `itanium2` - Optimize for Itanium 2 processor (Default).	ON `pentium` on IA-32 `itanium2` on Itanium-based Systems
`-MD`	Preprocess and compile. Generate output file (`.d` extension) containing dependency information.	OFF
`-MF`*`file`*	Generate makefile dependency information in *`file`*. Must specify `-M` or `-MM`.	OFF
`-MG`	Similar to `-M`, but treats missing header files as generated files.	OFF
`-MM`	Similar to `-M`, but does not include system header files.	OFF
`-MMD`	Similar to `-MD`, but does not include system header files.	OFF
`-MX`	Generate dependency file (`.o.dep` extension) containing information used for the Intel wb tool.	OFF
`-mp`	Favors conformance to the ANSI C and IEEE 754 standards for floating-point arithmetic.	OFF
`-mp1`	Improve floating-point precision (speed impact is less than `-mp`).	OFF
`-mrelax`	Pass `-relax` to the linker.	ON
`-mno-relax`	Do not pass `-relax` to the linker.	OFF
`-mserialize-volatile` Itanium-based systems only	Impose strict memory access ordering for volatile data object references.	OFF

`-mno-serialize-volatile` Itanium-based systems only	The compiler may suppress both run-time and compile-time memory access ordering for volatile data object references. Specifically, the `.rel/.acq` completers will not be issued on referencing loads and stores.	OFF
`-nobss_init`	Places variables that are initialized with zeroes in the DATA section. Disables placement of zero-initialized variables in BSS (use DATA).	OFF
`-no_cpprt`	Do not link in C++ run-time libraries.	OFF
`-nodefaultlibs`	Do not use standard libraries when linking.	
`-nolib_inline`	Disables inline expansion of standard library functions.	OFF
`-nostartfiles`	Do not use standard startup files when linking.	OFF
`-nostdlib`	Do not use standard libraries and startup files when linking.	OFF
`-O`	Same as `-O1` on IA-32. Same as `-O2` on Itanium-based systems.	OFF
`-O0`	Disables optimizations.	OFF
`-O1`	Enable optimizations. Optimizes for speed. For Itanium compiler, `-O1` turns off software pipelining to reduce code size.	ON
`-O2`	Same as `-O1` on IA-32. Same as `-O` on Itanium-based systems.	OFF
`-O3`	Enable `-O2` plus more aggressive optimizations that may increase the compilation time. Impact on performance is application dependent, some applications may not see a performance improvement.	OFF

`-Ob`*`n`*	Controls the compiler's inline expansion. The amount of inline expansion performed varies with the value of *n* as follows: <ul><li>`0`: Disables inlining.</li><li>`1`: Enables (default) inlining of functions declared with the `__inline` keyword. Also enables inlining according to the C++ language.</li><li>`2`: Enables inlining of any function. However, the compiler decides which functions to inline. Enables interprocedural optimizations and has the same effect as `-ip`.</li></ul>	ON
`-o`*`file`*	Name output *`file`*.	OFF
`-openmp`	Enables the parallelizer to generate multi-threaded code based on the OpenMP* directives. The `-openmp` option only works at an optimization level of `-O2` (the default) or higher.	OFF
`-openmp_report{0|1|2}`	Controls the OpenMP parallelizer's diagnostic levels.	ON `-openmp_report1`
`-openmp_stubs`	Enables OpenMP programs to compile in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked sequentially.	OFF
`-opt_report`	Generates an optimization report directed to stderr, unless `-opt_report_file` is specified.	OFF
`-opt_report_file`*`filename`*	Specifies the *`filename`* for the optimization report. It is not necessary to invoke `-opt_report` when this option is specified.	OFF
`-opt_report_level`*`level`*	Specifies the verbosity *`level`* of the output. Valid *`level`* arguments: <ul><li>`min`</li><li>`med`</li><li>`max`</li></ul>If a *`level`* is not specified, `min` is used by default.	OFF

`-opt_report_phase`*`name`*	Specifies the compilation *name* for which reports are generated. The option can be used multiple times in the same compilation to get output from multiple phases. Valid *name* arguments: • `ipo`: Interprocedural Optimizer • `hlo`: High Level Optimizer • `ilo`: Intermediate Language Scalar Optimizer • `ecg`: Code Generator • `omp`: OpenMP* • `all`: All phases	OFF
`-opt_report_routine`*`substring`*	Specifies a routine *`substring`*. Reports from all routines with names that include *`substring`* as part of the name are generated. By default, reports for all routines are generated.	OFF
`-opt_report_help`	Displays all possible settings for `-opt_report_phase`. No compilation is performed.	OFF
`-P`, `-F`	Stops the compilation process after C or C++ source files have been preprocessed and writes the results to files named according to the compiler's default file-naming conventions.	OFF
`-parallel`	Detects parallel loops capable of being executed safely in parallel and automatically generates multithreaded code for these loops.	OFF
`-par_report{0|1|2|3}`	Controls the auto-parallelizer's diagnostic levels 0, 1, 2, or 3 as follows: • `-par_report0`: no diagnostic information is displayed. • `-par_report1`: indicates loops successfully auto-parallelized (default). • `-par_report2`: loops successfully and unsccessfully auto-parallelized. • `-par_report3`: same as 2 plus additional information about any proven or assumed dependences inhibiting auto-parallelization.	OFF

`-par_threshold[n]`	Sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel, n=0 to 100. This option is used for loops whose computation work volume cannot be determined at compile time. • `-par_threshold0`: loops get auto-parallelized regardless of computation work volume. • `-par_threshold100`: loops get auto-parallelized only if profitable parallel execution is almost certain.	OFF
`-pc32` IA-32 only	Set internal FPU precision to 24-bit significand.	OFF
`-pc64` IA-32 only	Set internal FPU precision to 53-bit significand.	OFF
`-pc80` IA-32 only	Set internal FPU precision to 64-bit significand.	ON
`-prec_div` IA-32 only	Disables the floating point division-to-multiplication optimization. Improves precision of floating-point divides.	OFF
`-prof_dir` *`dirname`*	Specify the directory (*`dirname`*) to hold profile information (`*.dyn`, `*.dpi`).	OFF
`-prof_file` *`filename`*	Specify the *`filename`* for profiling summary file.	OFF
`-prof_gen[x]`	Instruments the program to prepare for instrumented execution and also creates a new static profile information file (`.spi`). With the `x` qualifier, extra information is gathered.	OFF
`-prof_use`	Uses dynamic feedback information.	OFF
`-Qinstall` *`dir`*	Sets *`dir`* as root of compiler installation.	OFF
`-Qlocation,`*`tool,path`*	Sets *`path`* as the location of the tool specified by tool .	OFF
`-Qoption,`*`tool,list`*	Passes an argument *`list`* to another *`tool`* in the compilation sequence, such as the assembler or linker.	OFF
`-qp`, `-p`	Compile and link for function profiling with UNIX* `prof tool`	OFF

`-rcd` IA-32 only	Disables changing of the FPU rounding control. Enables fast float-to-int conversions.	OFF
`-[no]restrict`	Enables/disables pointer disambiguation with the `restrict` qualifier.	OFF
`-S`	Generates assemblable files with `.s` suffix, then stops the compilation.	OFF
`-shared`	Produce a shared object.	OFF
`-size_lp64` Itanium-based systems only	Assume 64-bit size for long and pointer types.	OFF
`-sox[-]` IA-32 only	Enables [disables] the saving of compiler options and version information in the executable file.	`-sox-`
`-static`	Prevents linking with shared libraries.	OFF
`-std=c99`	Enable C99 support for C programs.	ON
`-syntax`	Checks the syntax of a program and stops the compilation process after the C or C++ source files and preprocessed source files have been parsed. Generates no code and produces no output files. Warnings and messages appear on stderr.	OFF
`-tpp1` Itanium-based systems only	Target optimization to the Itanium processor.	OFF
`-tpp2` Itanium-based systems only	Target optimization to the Itanium® 2 processor. Generated code is compatible with the Itanium processor.	ON
`-tpp5` IA-32 only	Targets the optimizations to the Pentium processor.	OFF
`-tpp6` IA-32 only	Targets the optimizations to the Pentium Pro, Pentium II and Pentium III processors.	OFF
`-tpp7` IA-32 only	Tunes code to favor the Pentium 4 and Intel® Xeon(TM) processor.	ON
`-U`*`name`*	Suppresses any definition of a macro *name*. Equivalent to a `#undef` preprocessing directive.	OFF
`-unroll0` Itanium-based systems only	Disable loop unrolling.	OFF

`-unroll[n]` IA-32 only	Set maximum number of times to unroll loops. Omit n to use default heuristics. Use n =0 to disable loop unroller.	OFF
`-use_asm`	Produce objects through assembler.	OFF
`-use_msasm` IA-32 only	Accept the Microsoft* MASM-style inlined assembly format instead of GNU-style.	OFF
`-u` *`symbol`*	Pretend the *`symbol`* is undefined.	OFF
`-V`	Display compiler version information.	OFF
`-v`	Show driver tool commands and execute tools.	
`-vec_report[n]` IA-32 only	Controls the amount of vectorizer diagnostic information. • n = 0 no diagnostic information • n = 1 indicates vectorized loops (DEFAULT) • n = 2 indicates vectorized/non-vectorized loops • n = 3 indicates vectorized/non-vectorized loops and prohibiting data dependence information • n = 4 indicates non-vectorized loops • n = 5 indicates non-vectorized loops and prohibiting data	ON `-vec_report1`
`-w`	Disable all warnings.	OFF
`-Wall`	Enable all warnings.	OFF
`-w`*`n`*	Control diagnostics. • n = 0 displays errors (same as `-w`) • n = 1 displays warnings and errors (DEFAULT) • n = 2 displays remarks, warnings, and errors	ON `-w1`
`-wd`*`L1`*`[,`*`L2`*`,...]`	Disables diagnostics *`L1`* through *`LN`*.	OFF
`-we`*`L1`*`[,`*`L2`*`,...]`	Changes severity of diagnostics *`L1`* through *`LN`* to error.	OFF
`-Werror`	Force warnings to be reported as errors.	OFF
`-wn`*`n`*	Limits the number of errors displayed prior to aborting compilation to *`n`*.	ON `-wn100`

`-wrL1[,L2,...]`	Changes the severity of diagnostics *L1* through *LN* to remark.	OFF
`-wwL1[,L2,...]`	Changes severity of diagnostics *L1* through *LN* to warning.	OFF
`-Wl,o1[,o2,...]`	Pass options *o1*, *o2*, etc. to the linker for processing.	OFF
`-xtype`	All source files found subsequent to `-xtype` will be recognized as one of the following *types*: • `c` - C source file • `c++` - C++ source file • `c-header` - C header file • `cpp-output` - C preprocessed file • `assembler` - assemblable file • `assembler-with-cpp` - Assemblable file that needs to be preprocessed. • `none` - Disable recognition and revert to file extension.	OFF
`-X`	Removes the standard directories from the list of directories to be searched for include files.	OFF
`-Xa`	Select extended ANSI C dialect.	ON
`-Xc`	Select strict ANSI conformance dialect.	OFF
`-x{M|i|K|W}` IA-32 only	Generates specialized code for processor-specific codes `M`, `i`, `K`, `W`. • `M` = Intel® Pentium® processors with MMX(TM) technology • `i` = Intel Pentium Pro and Intel Pentium II processors • `K` = Intel Pentium III processors • `W` = Intel Pentium 4 processors, Intel Xeon processors, and Intel Pentium M processors	OFF
`-Xlinker val`	Pass *val* directly to the linker for processing.	OFF
`-Zp{1|2|4|8|16}`	Specifies the strictest alignment constraint for structure and union types as one of the following: 1, 2, 4, 8, or 16 bytes.	ON `-Zp16`

Alternate Tools and Locations

Option	Description
`-Qlocation,`*`tool,path`*	Allows you to specify the path for tools such as the assembler, linker, preprocessor, and compiler.
`-Qoption,`*`tool,optlist`*	Passes an option specified by *`optlist`* to a *`tool`*, where *`optlist`* is a comma-separated list of options.

Preprocessing Options

Option	Description
`-A`*`name`*`[(`*`values`*`,...)]`	Associates a symbol *`name`* with the specified sequence of *`values`* . Equivalent to an `#assert` preprocessing directive.
`-A-`	Causes all predefined macros and assertions to be inactive.
`-C`	Preserves comments in preprocessed source output.
`-D`*`name`*`[(`*`value`*`)]`	Defines the macro *`name`* and associates it with the specified *`value`* . The default (`-D`*`name`*) defines a macro with a *`value`* of 1.
`-E`	Directs the preprocessor to expand your source module and write the result to standard output.
`-EP`	Directs the preprocessor to expand your source module and write the result to standard output. Does not include `#line` directives in the output.
`-P`	Directs the preprocessor to expand your source module and store the result in a `.i` file in the current directory.
`-U`*`name`*	Suppresses any automatic definition for the specified macro *`name`* .

Controlling Compilation Flow

Option	Description
`-c`	Stops the compilation process after an object file has been generated. The compiler generates an object file for each C or C++ source file or preprocessed source file. Also takes an assembler file and invokes the assembler to generate an object file.
`-Kpic`, `-KPIC`	Generate position-independent code.
`-l`*`name`*	Link with a library indicated in *`name`*.
`-nobss_init`	Places variables that are initialized with zeroes in the DATA section.
`-P, -F`	Stops the compilation process after C or C++ source files have been preprocessed and writes the results to files named according to the compiler's default file-naming conventions.
`-S`	Generates assemblable file with `.s` suffix, then stops the compilation.
`-sox[-]` (IA-32 only)	Enables [disables] the saving of compiler options and version information in the executable file. Default is `-sox-`.
`-Zp{1|2|4|8|16}`	Specifies the strictest alignment constraint for structure and union types as one of the following: 1, 2, 4, 8, or 16 bytes.
`-0f_check` (IA-32 only)	Avoids the incorrect decoding of certain 0f instructions for code targeted at older processors.

Controlling Compilation Output

Option	Description
`-L`*`directory`*	Instruct linker to search *`directory`* for libraries.
`-o`*`name`*	Produces an executable output file with the specified file *`name`* , or the default file name if file *`name`* is not specified.
`-S`	Generates assemblable file with `.s` suffix, then stops the compilation.

Debugging Options

Option(s)	Result
`-g`	Debugging information produced, `-O0` enabled, `-fp` enabled for IA-32-targeted compilations.
`-g -O1`	Debugging information produced, `-O1` optimizations enabled.
`-g -O2`	Debugging information produced, `-O2` optimizations enabled.
`-g -O3`	Debugging information produced, `-O3` optimizations enabled.
`-g -O3 -fp`	Debugging information produced, `-O3` optimizations enabled, `-fp` enabled for IA-32-targeted compilations.

Conformance Options

Option	Description
`-ansi`	Enables assumption of the program's ANSI conformance.
`-ansi_alias[-]`	`-ansi_alias` directs the compiler to assume the following: • Arrays are not accessed out of bounds. • Pointers are not cast to non-pointer types, and vice-versa. • References to objects of two different scalar types cannot alias. For example, an object of type `int` cannot alias with an object of type `float`, or an object of type `float` cannot alias with an object of type `double`. If your program satisfies the above conditions, setting the `-ansi_alias` flag will help the compiler better optimize the program. However, if your program does not satisfy one of the above conditions, the `-ansi_alias` flag may lead the compiler to generate incorrect code.
`-mp`	Favors conformance to the ANSI C and IEEE 754 standards for floating-point arithmetic. Behavior for `NaN` comparisons does not conform.

Optimization-level Options

Option	Description
`-O0`	Disables optimizations.
`-O1`	Enables optimizations. Optimizes for speed. `-O1` disables inline expansion of library functions. For Itanium® compiler, `-O1` turns off software pipelining to reduce code size.
`-O2`	Equivalent to option `-O1`.
`-O3`	Builds on `-O1` and `-O2` by enabling high-level optimization. This level does not guarantee higher performance unless loop and memory access transformation take place. In conjunction with `-axK`/`-xK`, this switch causes the compiler to perform more aggressive data dependency analysis than for `-O2`. This may result in longer compilation times.

Processor Optimizations

Processor Optimization for IA-32 only

The `-tpp{5|6|7}` options optimize your application's performance for a specific Intel processor. The resulting binary will also run on the other processors listed in the table below. The Intel® C++ Compiler includes gcc*-compatible versions of the `-tpp` options. These options are listed in the gcc* Version column.

Option	gcc* Version	Optimizes for
`-tpp5`	`-mcpu=pentium`	Intel® Pentium® processors
`-tpp6`	`-mcpu=pentiumpro`	Intel Pentium Pro, Intel Pentium II, and Intel Pentium III processors
`-tpp7`	`-mcpu=pentium4`	Intel Pentium 4 processors

Note
The `-tpp7` option is ON by default when you invoke `icc` or `icpc`.

Example

The invocations listed below all result in a compiled binary optimized for Pentium 4 and Intel® Xeon(TM) processors. The same binary will also run on Pentium, Pentium Pro, Pentium II, and Pentium III processors.

```
prompt>icc prog.c

prompt>icc -tpp7 prog.c

prompt>icc -mcpu=pentium4 prog.c
```

Processor Optimization (Itanium®-based Systems only)

The `-tpp{1|2}` options optimize your application's performance for a specific Intel® Itanium® processor. The resulting binary will also run on the processors listed in the table below. The Intel® C++ Compiler includes gcc*-compatible versions of the `-tpp` options. These options are listed in the gcc* Version column.

Option	gcc* Version	Optimizes for
`-tpp1`	`-mcpu=itanium`	Itanium® processors
`-tpp2`	`-mcpu=itanium2`	Itanium® 2 processors

Note

The `-tpp2` option is ON by default when you invoke `ecc` or `ecpc`.

Example

The invocations listed below all result in a compiled binary optimized for the Intel Itanium 2 processor. The same binary will also run on Intel Itanium processors.

```
prompt>ecc prog.c
```

```
prompt>ecc -tpp2 prog.c
```

```
prompt>ecc -mcpu=itanium2 prog.c
```

Interprocedural Optimizations

Option	Description
`-ip`	Enables interprocedural optimizations for single file compilation.
`-ip_no_inlining`	Disables inlining that would result from the `-ip` interprocedural optimization, but has no effect on other interprocedural optimizations.
`-ipo`	Enables interprocedural optimizations across files.
`-ipo_c`	Generates a multifile object file that can be used in further link steps.
`-ipo_obj`	Forces the compiler to create real object files when used with `-ipo`.
`-ipo_S`	Generates a multifile assemblable file named ipo_out.asm that can be used in further link steps.
`-inline_debug_info`	Preserve the source position of inlined code instead of assigning the call-site source position to inlined code.
`-nolib_inline`	Disables inline expansion of standard library functions.

Profile-guided Optimizations

Option	Description
`-prof_gen[x]`	Instructs the compiler to produce instrumented code in your object files in preparation for instrumented execution. NOTE: The dynamic information files are produced in phase 2 when you run the instrumented executable.
`-prof_use`	Instructs the compiler to produce a profile-optimized executable and merges available dynamic information (.dyn) files into a pgopti.dpi file. If you perform multiple executions of the instrumented program, `-prof_use` merges the dynamic information files again and overwrites the previous pgopti.dpi file.
`-prof_dir`*`dir`*	Specifies the directory (*dir*) to hold profile information in the profiling output files, `*.dyn` and `*.dpi`.
`-prof_file`*`file`*	Specifies *`file`* name for profiling summary file.

High-level Language Optimizations

Option	Description
`-openmp`	Enables the parallelizer to generate multi-threaded code based on the OpenMP* directives. Enables parallel execution on both uni- and multiprocessor systems.
`-openmp_report{0|1|2}`	Controls the OpenMP* parallelizer's diagnostic levels 0, 1, or 2: • `0` - no information • `1` - loops, regions, and sections parallelized (default) • `2` - same as 1 plus master construct, single construct, etc.
`-unroll[`*`n`*`]`	Set maximum number (*`n`*) of times to unroll loops. Omit *`n`* to use default heuristics. Use *`n`* =0 to disable loop unrolling. For Itanium®-based applications, `-unroll[0]` used only for compatibility.

Optimization Reports

Option	Description
`-opt_report`	Generates optimizations report and directs to `stderr`.
`-opt_report_file`*`filename`*	Specifies the *`filename`* for the optimizations report.
`-opt_report_level {`*`min`*`|`*`med`*`|`*`max`*`}`	Specifies the detail level of the optimizations report. Default: `-opt_report_level`*`min`*
`-opt_report_phase`*`phase`*	Specifies the optimization to generate the report for. Can be specified multiple times on the command line for multiple optimizations.
`-opt_report_help`	Prints to the screen all available phases for `-opt_report_phase`.
`-opt_report_routine`*`substring`*	Generates reports from all routines with names containing the *`substring`* as part of their name. If not specified, reports from all routines are generated.

Compiler Options Cross Reference

Linux*	Windows*	Description	Linux Default
`-0f`	`-QI0f`	Enable/disable the patch for the Pentium® `0f` erratum.	OFF
`-A-`	`-QA-`	Remove all predefined macros.	OFF
`-A`*`name`*`[(`*`val`*`)]`	`-QA`*`name`*`[(`*`val`*`)]`	Create an assertion name having value *`val`*.	OFF
`-ansi`	`-Za`	Enable/disable assumption of ANSI conformance.	ON
`-ax{M|i|K|W}`	`-Qax{M|i|K|W}`	Generates specialized code for processor-specific codes `M`, `i`, `K`, `W` while also generating generic IA-32 code. • `M` = Intel® Pentium® processors with MMX (TM) technology • `i` = Intel Pentium Pro and Intel Pentium II processors • `K` = Intel Pentium III processors • `W` = Intel Pentium 4 processors, Intel® Xeon (TM) processors, and Intel® Pentium® M processors	OFF
`-C`	`-C`	Don't strip comments.	OFF
`-c`	`-c`	Compile to object (`.o`) only, do not link.	OFF
`-D`*`name`*`[=`*`value`*`]`	`-D`*`name`*`[=`*`value`*`]`	Define macro.	OFF
`-E`	`-E`	Preprocess to stdout.	OFF
`-fp`	`-Oy-`	Use EBP-based stack frame for all functions.	OFF
`-g`	`-Zi`	Produce symbolic debug information in object file.	OFF
`-H`	`-QH`	Print include file order.	OFF
`-help`	`-help`	Print help message listing.	OFF

`-I`*`directory`*	`-I`*`directory`*	Add directory to include file search path.	OFF
`-inline_debug_info`	`-Qinline_debug_info`	Preserve the source position of inlined code instead of assigning the call-site source position to inlined code.	OFF
`-ip`	`-Qip`	Enable single-file IP optimizations (within files).	OFF
`-ip_no_inlining`	`-Qip_no_inlining`	Optimize the behavior of IP: disable full and partial inlining (requires `-ip` or `-ipo`).	OFF
`-ipo`	`-Qipo`	Enable multifile IP optimizations (between files).	OFF
`-ipo_obj`	`-Qipo_obj`	Optimize the behavior of IP: force generation of real object files (requires `-ipo`).	OFF
`-KPIC`	NA	Generate position independent code (same as `-Kpic`).	OFF
`-Kpic`	NA	Generate position independent code (same as `-KPIC`).	OFF
`-long_double`	`-Qlong_double`	Enable 80-bit long double.	OFF
`-m`	NA	Instruct linker to produce map file.	OFF
`-M`	`-QM`	Generate makefile dependency information.	OFF
`-mp`	`-Op[-]`	Maintain floating-point precision (disables some optimizations).	OFF
`-mp1`	`-Qprec`	Improve floating-point precision (speed impact is less than `-mp`).	OFF
`-nobss_init`	`-Qnobss_init`	Disable placement of zero-initialized variables in BSS (use DATA).	OFF
`-nolib_inline`	`-Oi[-]`	Disable inline expansion of intrinsic functions.	OFF
`-O`	`-O2`		OFF
`-o`*`file`*	`-Fe`*`file`* or `-Fo`*`file`*	Name output file.	OFF
`-O0`	`-Od`	Disable optimizations.	OFF
`-O1`	`-O1`	Optimizes for speed.	OFF

`-O2`	`-O2`		ON
`-P`	`-EP`	Preprocess to file.	OFF
`-pc32`	`-Qpc 32`	Set internal FPU precision to 24-bit significand.	OFF
`-pc64`	`-Qpc 64`	Set internal FPU precision to 53-bit significand.	OFF
`-pc80`	`-Qpc 80`	Set internal FPU precision to 64-bit significand.	ON
`-prec_div`	`-Qprec_div`	Improve precision of floating-point divides (some speed impact).	OFF
`-prof_dir` *`directory`*	`-Qprof_dir` *`directory`*	Specify directory for profiling output files (`*.dyn` and `*.dpi`).	OFF
`-prof_file` *`filename`*	`-Qprof_file`*`filename`*	Specify file name for profiling summary file.	OFF
`-prof_gen[x]`	`-Qprof_genx`	Instrument program for profiling; with the x qualifier, extra information is gathered.	OFF
`-prof_use`	`-Qprof_use`	Enable use of profiling information during optimization.	OFF
`-Qinstall` *`dir`*	NA	Set `dir` as root of compiler installation.	OFF
`-Qlocation,`*`str`*`,`*`dir`*	`-Qlocation,` *`tool`*`,` *`path`*	Set `dir` as the location of tool specified by `str`.	OFF
`-Qoption,`*`str`*`,`*`opts`*	`-Qoption,` *`tool`*`,` *`list`*	Pass options opts to tool specified by str.	OFF
`-qp`, `-p`	NA	Compile and link for function profiling with UNIX* gprof tool.	OFF
`-rcd`	`-Qrcd`	Enable fast floating-point-to-integer conversions.	OFF
`-restrict`	`-Qrestrict`	Enable the restrict keyword for disambiguating pointers.	OFF
`-S`	`-S`	Generates assemblable files with `.s` suffix, then stops the compilation.	OFF
`-sox[-]`	`-Qsox`	Enable [disable] saving of compiler options and version in the executable.	`-sox-`
`-syntax`	`-Zs`	Perform syntax check only.	OFF

`-tpp5`	`-G5`	Optimize for Pentium processor.	OFF
`-tpp6`	`-G6`	Optimize for Pentium Pro, Pentium II and Pentium III processors.	OFF
`-tpp7`	`-G7`	Optimize for Pentium 4 processor.	OFF
`-U`*`name`*	`-U`*`name`*	Remove predefined macro.	OFF
`-unroll[`*`n`*`]`	`-Qunroll`*`n`*	Set maximum number of times to unroll loops. Omit n to use default heuristics. Use n=0 to disable loop unroller.	OFF
`-V`	`-QV`	Display compiler version information.	OFF
`-w`	`-w`	Display errors.	OFF
`-w2`	`-W4`	Enable remarks, warnings and errors.	
`-w`*`n`*	`-Wn`	Control diagnostics. Display errors (n=0). Display warnings and errors (n=1). Display remarks, warnings, and errors (n=2).	OFF
`-wd`*`L1`*`[,`*`L2`*`,...]`	`-Qwd[tag]`	Disable diagnostics L1 through LN.	OFF
`-we`*`L1`*`[,`*`L2`*`,...]`	`-Qwe[tag]`	Change severity of diagnostics L1 through LN to error.	OFF
`-wn`*`n`*	`-Qwn[tag]`	Print a maximum of *n* errors.	OFF
`-wr`*`L1`*`[,`*`L2`*`,...]`	`-Qwr[tag]`	Change severity of diagnostics L1 through LN to remark.	OFF
`-ww`*`L1`*`[,`*`L2`*`,...]`	`-Qww[tag]`	Change severity of diagnostics L1 through LN to warning.	OFF
`-X`	`-X`	Remove standard directories from include file search path.	OFF

`-x{M|i|K|W}`	`-Qx{M|i|K|W}`	Generates specialized code for processor-specific codes `M`, `i`, `K`, `W`. • `M` = Intel® Pentium® processors with MMX (TM) technology • `i` = Intel Pentium Pro and Intel Pentium II processors • `K` = Intel Pentium III processors • `W` = Intel Pentium 4 processors, Intel® Xeon (TM) processors, and Intel® Pentium® M processors	OFF
`-Xa`	`-Ze`	Select extended ANSI C dialect.	OFF
`-Xc`	`-Za`	Select strict ANSI conformance dialect.	OFF
`-Zp{1|2|4|8|16}`	`-Zp[`*n*`]`	Specify, in bytes, alignment constraint for structures (*n* =1,2,4,8,16). Default *n* =8. This option overrides the default alignment of code.	OFF

Invoking the Compiler

The ways to invoke Intel® C++ Compiler are as follows:

- Invoke directly: Running Compiler from the Command Line
- Use system make file: Running from the Command Line with make

Invoking the Compiler from the Command Line

There are two necessary steps to invoke the Intel® C++ Compiler from the command line:

1. Set the environment variables.
2. Invoke the compiler with `icc` or `ecc`.

Note

You can also invoke the compiler with `icpc` and `ecpc` for C++ source files on IA-32 and Itanium®-based systems respectively. The `icc` and `ecc` compiler examples in this documentation apply to C and C++ source files.

Set the Environment Variables

Before you can operate the compiler, you must set the environment variables to specify locations for the various components. The Intel C++ Compiler installation includes shell scripts that you can use to set environment variables. From the command line, execute the shell script that corresponds to your installation. With the default compiler installation, these scripts are located at:

- **IA-32 Systems:** `/opt/intel/compiler70/ia32/bin/iccvars.sh`
- **Itanium®-based Systems:** `/opt/intel/compiler70/ia64/bin/eccvars.sh`

Running the Shell Scripts

To run the `iccvars.sh` script on IA-32, enter the following on the command line:

```
prompt>source /opt/intel/compiler70/ia32/bin/iccvars.sh
```

If you want the `iccvars.sh` to run automatically when you start Linux*, edit your startup file and add the same line to the end of your file:

```
# set up environment for Intel compiler icc
source /opt/intel/compiler70/ia32/bin/iccvars.sh
```

The procedure is similar for running the `eccvars.sh` shell script on Itanium-based systems.

Invoke the Compiler

Once the environment variables are set, you can invoke the compiler as follows:

- **IA-32 Systems:** `prompt>`**`icc`**` [options] file1 [file2 . . .]`
- **Itanium®-based Systems:** `prompt>`**`ecc`**` [options] file1 [file2 . . .]`

Syntax	**Description**
`options`	Indicates one or more command-line options. The compiler recognizes one or more letters preceded by a hyphen (`-`).
`file1, file2 . . .`	Indicates one or more files to be processed by the compilation system. You can specify more than one file. Use a space as a delimiter for multiple files.
`linker_options`	Indicates options directed to the linker.

Invoking the Compiler from the Command Line with make

To run from the command line using Intel® C++ Compiler, make sure that `/usr/bin` is your path. If you use a C shell, you can edit your `.cshrc` file and add

```
setenv PATH /usr/bin:<full path to Intel compiler>
```

Note

To use the Intel compiler, your makefile must include the setting `CC=icc`. Use the same setting on the command line to instruct the makefile to use the Intel compiler. If your makefile is written for gcc, the GNU* C compiler, you will need to change those command line options not recognized by the Intel compiler.

Then you can compile:

```
prompt>make -f my_makefile
```

Compiler Input Files

By default, the compiler recognizes `.cpp` and `.cxx` files as C++ files and `.c` files as C language source files. Examples in this documentation use the `.c` extension. The Intel® C++ Compilers, `icpc` and `ecpc`, compile `.c` files as C++ files. Also, the Intel C++ Compiler recognizes the default file name extensions listed in the table below:

Filename	Interpretation	Action
`filename.a`	Object library	Passed to linker
`filename.i`	C or C++ source preprocessed and expanded by the C++ preprocessor	Passed to compiler
`filename.o`	Compiled object module	Passed to linker
`filename.s`	Assemblable file	
`filename.so`	Shared object file	
`filename.S`	Assemblable file that requires preprocessing	

Default Compiler Options

- Options specific to IA-32 architecture
- Options specific to the Itanium® architecture

All other options are supported on both IA-32 and Itanium-based systems.

Option	Description
`-c99`	Enables C99 support for C programs
`-falias`	Assume aliasing in program.
`-ffnalias`	Assume aliasing within functions
`-fverbose-asm`	Produce assemblable file with compiler components.
`-KPIC`, `-Kpic`	Generate position independent code.
`-mcpu=pentium4`	Optimizes for Pentium® 4 processor (IA-32 systems only).
`-mcpu=itanium2`	Optimizes for Itanium 2 processor (Itanium-based systems only)
`-O1`	Enable optimizations. Optimizes for speed. For Itanium compiler, `-O1` turns off software pipelining to reduce code size.
`-O2`	Same as `-O1` on IA-32. Same as `-O` on Itanium-based systems.
`-Ob1`	Enables inlining of functions declared with the `__inline` keyword. Also enables inlining according to the C++ language.
`-openmp report1`	Controls the OpenMP parallelizer's diagnostic levels.
`-pc80` IA-32 only	Set internal FPU precision to 64-bit significand.
`-sox-` IA-32 only	Disables the saving of compiler options and version information in the executable file.
`-std=c99`	Enable C99 support for C programs.
`-tpp2` Itanium-based systems only	Target optimization to the Itanium® 2 processor. Generated code is compatible with the Itanium processor.
`-tpp7` IA-32 only	Tunes code to favor the Pentium 4 and Intel® Xeon(TM) processor.
`-vec_report1` IA-32 only	Controls the amount of vectorizer diagnostic information to indicate vectorized loops.
`-w1`	Control diagnostics. Displays warnings and errors.
`-Zp16`	Specifies 16-byte alignment constraint for struct and union types.

Default Behavior of the Compiler

If you do not specify any options when you invoke the Intel® C++ Compiler, the compiler uses the following default settings:

- Produces executable output with filename `a.o`.
- Invokes options specified in a configuration file first. See Configuration Files.
- Searches for library files in directories specified by the `LD_LIBRARY_PATH` variable, if they are not found in the current directory.
- Sets 8 bytes as the strictest alignment constraint for structures.
- Displays error and warning messages.
- Performs standard optimizations using the default `-O2` option. See Setting Optimization Levels.

If the compiler does not recognize a command-line option, that option is ignored and a warning is displayed. See Diagnostic Messages for detailed descriptions about system messages.

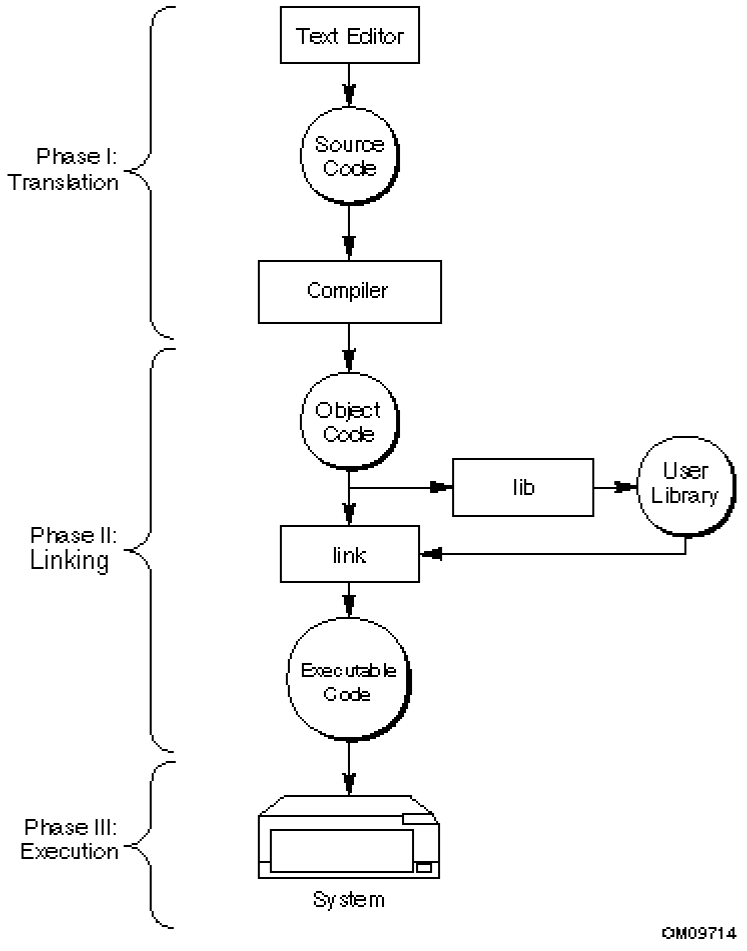
Compilation Phases

To produce an executable file, the compiler performs by default the compile and link phases. When invoked, the compiler driver determines which compilation phases to perform based on the file name extension and the compilation options specified in the command line.

The compiler passes object files and any unrecognized file name to the linker. The linker then determines whether the file is an object file (`.o`) or a library (`.a`). The compiler driver handles all types of input files correctly, thus it can be used to invoke any phase of compilation.

The relationship of the compiler to system-specific programming support tools is presented in the diagram below:

Application Development Cycle



Customizing the Compilation Environment

For IA-32 and the Intel® Itanium® architecture, you will need to set a compilation environment. To customize the environment used during compilation, you can specify:

- Environment Variables -- the paths where the compiler and other tools can search for specific files.
- Configuration Files -- the options to use with each compilation.
- Response Files -- the options and files to use for individual projects.
- Include Files -- the names and locations of source header files.

Environment Variables

You can customize your environment by specifying paths where the compiler can search for special files such as libraries and include files.

- `LD_LIBRARY_PATH` specifies the location for all Intel-provided libraries.
- `PATH` specifies the directories the system searches for binary executable files.
- `ICCCFG` specifies the configuration file for customizing compilations with the `icc` compiler.
- `ICPCCFG` specifies the configuration file for customizing compilations with the `icpc` compiler.
- `ECCCFG` specifies the configuration file for customizing compilations with the `ecc` compiler.
- `ECPCCFG` specifies the configuration file for customizing compilations with the `ecpc` compiler.
- `TMP` specifies the directory to store temporary files. If the directory specified by `TMP` does not exist, the compiler places the temporary files in the current directory.
- `IA32ROOT` (IA32-based systems) points to the directory containing the `bin`, `lib`, `include` and substitute header directories.
- `IA64ROOT` (Itanium®-based systems) points to the directory containing the `bin`, `lib`, `include` and substitute header directories.

Compilation Environment Options

The Intel® C++ Compiler installation includes shell scripts that you can use to set environment variables. From the command line, execute the shell script appropriate to your installation. You can find these scripts in the following locations (assuming you installed to the default directories):

Bash Shell Environment

- **IA-32 Systems: `/opt/intel/compiler70/ia32/bin/iccvars.sh`**
- **Itanium®-based Systems: `/opt/intel/compiler70/ia64/bin/eccvars.sh`**

To run the `iccvars.sh` script, enter the following on the command line:

```
prompt>source /opt/intel/compiler70/ia32/bin/iccvars.sh
```

If you want the `iccvars.sh` to run automatically when you start Linux, edit your startup script (`.bash_profile` for a bash shell) and add the same line to the end of your file:

```
# set up environment for icc
source /opt/intel/compiler70/ia32/bin/iccvars.sh
```

Configuration Files

You can decrease the time you spend entering command-line options and ensure consistency by using the configuration file to automate often-used command-line entries. You can insert any valid command-line option into the configuration file. The compiler processes options in the configuration file in the order they appear followed by the command-line options that you specify when you invoke the compiler.

Note

Options in the configuration file will be executed every time you run the compiler. If you have varying option requirements for different projects, see Response Files.

How to Use Configuration Files

The following example illustrates a basic configuration file. After you have written the `.cfg` file, simply ensure it is in the same directory as the compiler's executable file when you run the compiler. The text following the pound (#) character is recognized as a comment. For IA-32 compilations, the configuration file is `icc.cfg`. For compilations targeted for Itanium®-based systems, the configuration file is `ecc.cfg`.

```
## Sample configuration file.
## Define preprocessor macro MY_PROJECT.
-DMY_PROJECT
## Additional directories to be searched
## for INCLUDE files, before the default.
-I /project/include
```

Response Files

Use response files to specify options used during particular compilations, and to save this information in individual files. Response files are invoked as an option in the command line. Options in a response file are inserted in the command line at the point where the response file is invoked.

Response files are used to decrease the time spent entering command-line options, and to ensure consistency by automating command-line entries. Use individual response files to maintain options for specific projects to avoid editing the configuration file when changing projects.

Any number of options or file names can be placed on a line in the response file. Several response files can be referenced in the same command line. Use the pound character(`#`) to treat the rest of the line as a comment.

The syntax for using response files is as follows:

- **IA-32 Systems:** `prompt>`**`icc @response_file filenames`**
- **Itanium®-based Systems:** `prompt>`**`ecc @response_file filenames`**

Note

An "at" sign (`@`) must precede the name of the response file on the command line.

Include Files

Include directories are searched in the default system areas and whatever is specified by the `-I`*`directory`* option. For multiple search directories, multiple `-I`*`directory`* commands must be used. The compiler searches directories for include files in the following order:

- Directory of the source file that contains the include
- Directories specified by the `-I` option

How to Remove Include Directories

Use the `-X` option to prevent the compiler from searching the default system areas. You can use the `-X` option with the `-I` option to prevent the compiler from searching the default path for include files and direct it to use an alternate path.

For example, to direct the compiler to search the path `/alt/include` instead of the default path, do the following:

- **IA-32 Systems:** `prompt>`**`icc -X -I/alt/include prog.c`**
- **Itanium®-based Systems:** `prompt>`**`ecc -X -I/alt/include prog.c`**

Overview: Customizing Compilation Process

This section describes options that customize the compilation process:

- Preprocessing
- Compiling
- Linking
- Debugging

Specifying Alternate Tools and Paths

You can direct the compiler to specify alternate tools for preprocessing, compilation, assembly, and linking. Further, you can invoke options specific to your alternate tools on the command line. The following sections explain how to use `-Qlocation` and `-Qoption` to do this.

How to Specify an Alternate Component

Use `-Qlocation` to specify an alternate path for a tool. This option accepts two arguments using the following syntax:

`-Qlocation``,`*`tool`*`,`*`path`*

`tool`	**Description**
`cpp`	Specifies the compiler front-end preprocessor.
`c`	Specifies the C++ compiler.
`asm`	Specifies the assembler.
`ld`	Specifies the linker.

`path` is the complete path to the tool.

How to Pass Options to Other Programs

Use `-Qoption` to pass an option specified by *`optlist`* to a *`tool`*, where *`optlist`* is a comma-separated list of options. The syntax for this command is the following:

`-Qoption``,`*`tool`*`,`*`optlist`*

`tool`	**Description**
`cpp`	Specifies the compiler front-end preprocessor.
`c`	Specifies the C++ compiler.
`asm`	Specifies the assembler.
`ld`	Specifies the linker.

`optlist` Indicates one or more valid argument strings for the designated program. If the argument is a command-line option, you must include the hyphen. If the argument contains a space or tab character, you must enclose the entire argument in quotation characters (""). You must separate multiple arguments with commas. The following example directs the linker to create a memory map when the compiler produces the executable file from the source.

- **IA-32 Systems:** `prompt>`**`icc -Qoption,link,-map,proto.map proto.c`**
- **Itanium®-based Systems:** `prompt>`**`ecc -Qoption,link,-map,proto.map proto.c`**

The **`-Qoption,link`** option in the preceding example is passing the **`-map`** option to the linker. This is an explicit way to pass arguments to other tools in the compilation process. Also, you can use the `-Xlinker` *`val`* to pass values (`val`) to the linker.

Overview: Preprocessing

This section describes the options you can use to direct the operations of the preprocessor. Preprocessing performs such tasks as macro substitution, conditional compilation, and file inclusion.

Preprocessor Options

Option	Description
`-A`*`name`*`[(`*`value`*`)]`	Associates a symbol *`name`* with the specified *`value`*. Equivalent to an `#assert` preprocessing directive.
`-A-`	Causes all predefined macros and assertions to be inactive.
`-C`	Preserves comments in preprocessed source output.
`-D`*`name`*`[=`*`text`*`]`	Defines the macro *`name`* and associates it with the specified *`text`*. The default (`-D`*`name`*) defines a macro with where *`text`* = 1.
`-E`	Directs the preprocessor to expand your source module and write the result to `stdout`. Output includes `#line` directives.
`-EP`	Directs the preprocessor to expand your source module and write the result to standard output. The output does not include `#line` directives.
`-P`	Directs the preprocessor to expand your source module and store the result in a `.i` file in the current directory. Output does not include `#line` directives.
`-U`*`name`*	Suppresses any automatic definition for the specified macro *`name`*.
`-X`	Remove standard directories from include file search path.
`-H`	Outputs the full path names of all included files to `stdout` in order. Indentation is used to designate the `#include` dependencies.
`-M`	Generate makefile dependency information.
`-MD`	Preprocess and compile. Generate output file (`.d` extension) containing dependency information.
`-MF`*`file`*	Generate makefile dependency information in *`file`*. Must specify `-M` or `-MM`.
`-MG`	Similar to `-M`, but treats missing header files as generated files.
`-MM`	Similar to `-M`, but does not include system header files.
`-MMD`	Similar to `-MD`, but does not include system header files.
`-MX`	Generate dependency file (`.o.dep` extension) containing information used for the Intel wb tool.
`-dM`	Output macro definitions in effect after preprocessing (use with `-E`).
`-MD`	Preprocess and compile. Generate output file (`.d` extension) containing dependency information.
`-I`*`directory`*	Specifies an additional *`directory`* to search for include files.

Preprocessing Only

Use the `-E`, `-P` or `-EP` option to preprocess your source files without compiling them. When using these options, only the preprocessing phase of compilation is activated.

Using -E

When you specify the `-E` option, the compiler's preprocessor expands your source module and writes the result to `stdout`. The preprocessed source contains `#line` directives, which the compiler uses to determine the source file and line number. For example, to preprocess two source files and write them to `stdout`, enter the following command:

- **IA-32 Systems:** `prompt>`**`icc -E prog1.c prog2.c`**
- **Itanium®-based Systems:** `prompt>`**`ecc -E prog1.c prog2.c`**

Using -P

When you specify the `-P` option, the preprocessor expands your source module and directs the output to a `.i` file instead of `stdout`. Unlike the `-E` option, the output from `-P` does not include `#line` number directives. By default, the preprocessor creates the name of the output file using the prefix of the source file name with a `.i` extension. You can change this by using the `-o`*`file`* option. For example, the following command creates two files named `prog1.i` and `prog2.i`, which you can use as input to another compilation:

- **IA-32 Systems**: `prompt>`**`icc -P prog1.c prog2.c`**
- **Itanium®-based Systems**: `prompt>`**`ecc -P prog1.c prog2.c`**

Caution

When you use the `-P` option, any existing files with the same name and extension are overwritten.

Using -EP

Using the `-EP` option directs the preprocessor to not include `#line` directives in the output. `-EP` is equivalent to `-E -P`.

- **IA-32 Systems**: `prompt>`**`icc -EP prog1.c prog2.c`**
- **Itanium®-based Systems**: `prompt>`**`ecc -EP prog1.c prog2.c`**

Preserving Comments in Preprocessed Source Output

Use the `-C` option to preserve comments in your preprocessed source output. Comments following preprocessing directives, however, are not preserved.

Preprocessing Directive Equivalents

You can use the `-A`, `-D`, and `-U` options as equivalents to preprocessing directives:

- `-A` is equivalent to a `#assert` preprocessing directive
- `-D` is equivalent to a `#define` preprocessing directive
- `-U` is equivalent to a `#undef` preprocessing directive

Using -A

Use the `-A` option to make an assertion. **Syntax:** `-A`*`name`*`[(`*`value`*`)]`.

Argument	Description
name	Indicates an identifier for the assertion
value	Indicates a *value* for the assertion. If a *value* is specified, it should be quoted, along with the parentheses delimiting it.

For example, to make an assertion for the identifier `fruit` with the associated values `orange` and `banana` use the following command:

- **IA-32 Systems**: `prompt>`**`icc -A"fruit(orange,banana)" prog1.c`**
- **Itanium®-based Systems**: `prompt>`**`ecc -A"fruit(orange,banana)" prog1.c`**

Using -D

Use the `-D` option to define a macro. **Syntax:** `-D`*`name`*`[=`*`value`*`]`.

Argument	Description
name	The name of the macro to define.
value	Indicates a value to be substituted for name. If you do not enter a value, name is set to 1. The value should be quoted if it contains non-alphanumerics.

For example, to define a macro called `SIZE` with the value 100 use the following command:

- **IA-32 Systems**: `prompt>`**`icc -DSIZE=100 prog1.c`**
- **Itanium®-based Systems:** `prompt>`**`ecc -DSIZE=100 prog1.c`**

The `-D` option can also be used to define functions. For example, `icc -D"f(x)=x" prog1.c`.

Using -U

Use the `-U` option to remove (undefine) a pre-defined macro. **Syntax:** `-U`*`name`*.

Argument	Description
`name`	The name of the macro to undefine.

Note

If you use `-D` and `-U` in the same compilation, the compiler processes the `-D` option before `-U`, rather than processing them in the order they appear on the command line.

Predefined Macros

Intel-specific predefined macros are described in the table below. The Default column indicates whether the macro is enabled (ON) or disabled (OFF) by default. The Architecture column indicates which Intel architecture supports the predefined macro. Predefined macros specified by the ISO/ANSI standard are not listed in the table. For a list of all macro definitions in effect, use the `-E -dM` options. For example:

- **IA-32 Systems**: `prompt>`**`icc -E -dM prog1.c`**
- **Itanium®-based Systems**: `prompt>`**`ecc -E -dM prog1.c`**

Predefined Macros

Macro Name	Default	Architecture	Description / When Used
`__ECC=n`	n=700	Itanium architecture only	Enables the Intel® C++ Compiler. Assigned value refers to version of the compiler (e.g., 700 is 7.00). Supported for legacy reasons. Use `__INTEL_COMPILER` instead.
`__EDG__`	ON	Both	Defined to have the value 1.
`__ELF__`	ON	Both	
`__GXX_ABI_VERSION=100`		Both	
`__i386`	ON	IA-32	
`__i386__`	ON	IA-32	
`i386`	ON	IA-32	
`__ia64`	ON	Itanium architecture only	
`__ia64__`	ON	Itanium architecture only	
`ia64`	ON	Itanium architecture only	
`__ICC=n`	ON n=700	IA-32 only	Enables the Intel C++ Compiler. Assigned value refers to version of the compiler (e.g., 700 is 7.00). Supported for legacy reasons. Use `__INTEL_COMPILER` instead.

`__INTEL_COMPILER=n`	ON n=700	Both	Defines the compiler version. Defined as 700 for the Intel C++ Compiler 7.1.
`_INTEGRAL_MAX_BITS=n`	n=64	Itanium architecture only	Indicates support for the `__int64` type.
`__linux`	ON	Both	
`__linux__`	ON	Both	
`linux`	ON	Both	
`__LONG_MAX=n`	n=9223372036854775807L	Itanium architecture only	
`__LP64`	ON	Itanium architecture only	
`__lp64`	ON	Itanium architecture only	
`__LP64__`	ON	Itanium architecture only	
`_M_IA64=n`	n=64100	Itanium architecture only	Indicates the value for the preprocessor identifier to reflect the Itanium® architecture.
`__OPTIMIZE__`	ON	Both	Not enabled if all optimizations are turned off.
_PGO_INSTRUMENT	OFF	Both	Defined when compile with either `-prof_gen` or `-prof_genx`.
`__PTRDIFF_TYPE__`	ON	Both	For IA-32, `__PTRDIFF_TYPE__`=int For Itanium architecture, `__PTRDIFF_TYPE__`=long
`__SIZE_TYPE__`	ON	Both	For IA-32, `__SIZE_TYPE__`=unsignec For Itanium architecture, `__SIZE_TYPE__`=unsignec long
`__unix`	ON	Both	
`__unix__`	ON	Both	
`unix`	ON	Both	
`__USER_LABEL_PREFIX__`	ON	Both	

Suppress Macro Definition

Use the `-U`*`name`* option to suppress any macro definition currently in effect for the specified *`name`*. The `-U` option performs the same function as an `#undef` preprocessor directive.

Overview: Compilation

This section describes the Intel® C++ Compiler options that determine the compilation process and output. By default, the compiler converts source code directly to an executable file. Appropriate options allow you to control the process by directing the compiler to produce:

- Preprocessed files (`.i`) with the `-P` option.
- Assemblable files (`.s`) with the `-S` option.
- Object files (`.o`) with the `-c` option.
- Executable files (`.out`) by default.

You can also name the output file or designate a set of options that are passed to the linker. If you specify a phase-limiting option, the compiler produces a separate output file representing the output of the last phase that completes for each primary input file.

Controlling Compilation

If no errors occur during processing, you can use the output files from a particular phase as input to a subsequent compiler invocation. The table below describes the options to control the output:

Last Phase Completed	**Option**	**Compiler Input**	**Compiler Output**
Preprocessing	`-E`, `-P`, or `-EP`	• Source files	Preprocessed files (`.i` files)
Compile only	`-c`	• Source files • Preprocessed files	Compile to object only (`.o`), do not link.
	`-S`	• Source files • Preprocessed files	Generate assemblable files with `.s` suffix and stops the compilation process.
Syntax checking	`-syntax`	• Source files • Preprocessed files	Diagnostic list
Linking	(Default)	• Source files • Preprocessed files • Assemblable files • Object files • Libraries	Executable file (`.out` files)

Monitoring Data Settings

The options described below provide monitoring of Intel compiler-generated code.

Specifying Structure Tag Alignments

You can specify an alignment constraint for structures and unions in two ways:

- Place a pack pragma in your source file, or
- Enter the alignment option on the command line

Both specifications change structure tag alignment constraints.

Use the `-Zp` option to determine the alignment constraint for structure declarations. Generally, smaller constraints result in smaller data sections while larger constraints support faster execution.

The form of the `-Zp` option is `-Zp`*n*.

The alignment constraint is indicated by one of the following values:

n=1	1 byte.
n=2	2 bytes.
n=4	4 bytes.
n=8	8 bytes
n=16	16 bytes.

For example, to specify 2 bytes as the alignment constraint for all structures and unions in the file `prog.c`, use the following command:

- **IA-32 Systems:** `prompt>`**`icc -Zp2 prog.c`**
- **Itanium®-based Systems:** `prompt>`**`ecc -Zp2 prog.c`**

Note
Changing the alignment may cause problems if you are using system libraries compiled with the default alignment.

Flushing Denormal Values to Zero for Itanium-based Systems Only

Option `-ftz` flushes denormal results to zero when the application is in the gradual underflow mode. Use this option if the denormal values are not critical to application behavior. Flushing the denormal values to zero with `-ftz` may improve performance of your application. The default status of `-ftz` is OFF. By default, the compiler lets results gradually underflow.

The `-ftz` switch only needs to be used on the source containing function `main()`. The effect of the `-ftz` switch is to turn on FTZ mode for the process started by `main()`. The initial thread and any threads subsequently created by that process will operate in FTZ mode.

Note

The `-O3` option turns `-ftz` ON. Use `-ftz-` to disable flushing denormal results to zero.

Allocation of Zero-initialized Variables

By default, variables explicitly initialized with zeros are placed in the BSS section. But using the `-nobss_init` option, you can place any variables that are explicitly initialized with zeros in the DATA section if required.

Avoiding Incorrect Decoding of Certain Instructions (IA-32 Only)

Some instructions have 2-byte `opcodes` in which the first byte contains `0f`. In rare cases, the Pentium® processor can decode these instructions incorrectly. Specify the `-0f_check` option to avoid the incorrect decoding of these instructions.

Linking

This topic describes the options that let you control and customize the linking with tools and libraries and define the output of the `ld` linker. See the `ld` man page for more information on the linker.

Option	Description
`-L`*`directory`*	Instruct the linker to search *`directory`* for libraries.
`-Qoption,`*`tool`*`,`*`list`*	Passes an argument list to another program in the compilation sequence, such as the assembler or linker.
`-shared`	Instructs the compiler to build a Dynamic Shared Object (DSO) instead of an executable.
`-shared-libcxa`	`-shared-libcxa` has the opposite effect of `-static-libcxa`. When it is used, the Intel-provided `libcxa` C++ library is linked in dynamically, allowing the user to override the static linking behavior when the `-static` option is used.
`-i_dynamic`	Specifies that all Intel-provided libraries should be linked dynamically.
`-static`	Causes the executable to link all libraries statically, as opposed to dynamically.
`-static-libcxa`	By default, the Intel-provided `libcxa` C++ library is linked in dynamically. Use `-static-libcxa` on the command line to link `libcxa` statically, while still allowing the standard libraries to be linked in by the default behavior.
`-Bstatic`	This option is placed in the linker command line corresponding to its location on the user command line. This option is used to control the linking behavior of any library being passed in via the command line. When `-Bstatic` is not used: • `/lib/ld-linux.so.2` is linked in • `libm`, `libcxa`, and `libc` are linked dynamically • all other libs are linked statically When `-Bstatic` is used: • `/lib/ld-linux.so.2` is not linked in • all other libs are linked statically
`-Bdynamic`	This option is placed in the linker command line corresponding to its location on the user command line. This option is used to control the linking behavior of any library being passed in via the command line.

Suppressing Linking

Use the `-c` option to suppress linking. For example, entering the following command produces the object files `file1.o` and `file2.o`:

```
<compiler> -c file1.c file2.c
```

Note

The preceding command does not link these files to produce an executable file.

Overview: Debugging Options

Use the `-g` option to produce debug information. When you specify `-g`, the compiler disables optimizations by invoking `-O0`. Specifying the `-g` or `-O0` option automatically enables the `-fp` option (IA-32 only). The `-fp` option disables using the EBP register as general purpose register.

If you specify `-g` with `-O1`, `-O2`, or `-O3`, then `-fp` is disabled and allows the compiler to use the EBP register as a general purpose register in optimizations. However, most debuggers expect EBP to be used as a stack frame pointer, and cannot produce a stack backtrace unless this is so. Using the `-fp` option can result in slightly less efficient code.

Option(s)	Result
`-g`	Debugging information produced, `-O0` enabled, `-fp` enabled for IA-32-targeted compilations.
`-g -O1`	Debugging information produced, `-O1` optimizations enabled, `-fp` disabled for IA-32-targeted compilations.
`-g -O2`	Debugging information produced, `-O2` optimizations enabled, `-fp` disabled for IA-32-targeted compilations.
`-g -O3`	Debugging information produced, `-O3` optimizations enabled, `-fp` disabled for IA-32-targeted compilations.
`-g -O3 -fp`	Debugging information produced, `-O3` optimizations enabled, `-fp` enabled for IA-32-targeted compilations.
`-ip`	Symbols and line numbers produced for debugging.
`-ipo`	Symbols and line numbers produced for debugging.

Preparing for Debugging

Use the `-g` option to direct the compiler to generate code to support symbolic debugging. For example:

- **IA-32 Systems:** `prompt>`**`icc -g prog.c`**
- **Itanium®-based Systems:** `prompt>`**`ecc -g prog.c`**

The compiler does not support the generation of debugging information in assemblable files. If you specify the `-g` option, the resulting object file will contain debugging information, but the assemblable file will not.

Support for Symbolic Debugging

The compiler lets you generate code to support symbolic debugging while the `-O1`, `-O2`, or `-O3` optimization options are specified on the command line along with `-g`. However, you can receive these unexpected results:

- If you specify the `-O1`, `-O2`, or `-O3` options with the `-g` option, some of the debugging information returned may be inaccurate as a side-effect of optimization.
- If you specify the `-O1`, `-O2`, or `-O3` options, the `-fp` option (IA-32 only) will be disabled.

Parsing for Syntax and Semantics Only

Use the `-syntax` option to stop processing source files after they have been parsed for C++ language errors. This option provides a method to quickly check whether sources are syntactically and semantically correct. The compiler creates no output file. In the following example, the compiler checks `prog.c`. and displays diagnostic information to the standard error output:

- **IA-32 Systems**: `prompt>`**`icc -syntax prog.c`**
- **Itanium®-based Systems**: `prompt>`**`ecc -syntax prog.c`**

Conformance to the C Standard

You can set the Intel® C++ Compiler to accept either

- strict ANSI conformance dialect using the `-Xc` or `-ansi` option, or
- extended ANSI C dialect using the `-Xa` option

The compiler is set by default to accept extensions and not be limited to the ANSI/ISO standard.

Understanding the ANSI/ISO Standard C Dialect

The Intel C++ Compiler provides conformance to the ANSI/ISO standard for C language compilation (ISO/IEC 9899:1990). This standard requires that conforming C compilers accept minimum translation limits. This compiler exceeds all of the ANSI/ISO requirements for minimum translation limits.

Macros Included with the Compiler

The ANSI/ISO standard for 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 provides predefined macros in addition to the predefined macros required by the standard.

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

C99 Support

The following C99 features are supported in this version of the Intel C++ Compiler when using the `-c99[-]` option:

- Restricted pointers (`restrict` keyword, available with `-restrict`). See Note below.
- Variable-length Arrays
- Flexible array members
- Complex number support (`_Complex` keyword)
- Hexadecimal floating-point constants
- Compound literals
- Designated initializers
- Mixed declarations and code
- Macros with a variable number of arguments
- Inline functions (`inline` keyword)
- Boolean type (`_Bool` keyword)

Note

The `-restrict` option enables the recognition of the `restrict` keyword as defined by the ANSI standard. By qualifying a pointer with the `restrict` keyword, the user asserts that an object accessed via the pointer is only accessed via that pointer in the given scope. It is the user's responsibility to use the `restrict` keyword only when this assertion is true. In these cases, the use of `restrict` will have no effect on program correctness, but may allow better optimization.

These features are not supported:

- `#pragma STDC FP_CONTRACT`
- `#pragma STDC FENV_ACCESS`
- `#pragma STDC CX_LIMITED_RANGE`
- `long double` (128-bit representations)

Conformance to the C++ Standard

The Intel® C++ Compiler conforms to the ANSI/ISO standard (ISO/IEC 14882:1998) for the C++ language, however, the `export` keyword for templates is not implemented.

Overview: Optimization Levels

The table below shows the optimizations that the Intel® C++ Compiler applies when you invoke the `-O1`, `-O2`, or `-O3` options.

Optimization
Constant propagation
Copy propagation
Dead-code elimination
Global register allocation
Instruction scheduling
Loop unrolling (`-O2`, `-O3` only)
Loop-invariant code movement
Partial redundancy elimination
Strength reduction/induction variable simplification
Variable renaming
Exception handling optimizations
Tail recursions
Peephole optimizations
Structure assignment lowering and optimizations
Dead store elimination

Setting Optimization Levels

Depending on the Intel architecture, optimization can have different effects. To specify optimizations for your target architecture, refer to the tables below.

Itanium® Compiler

Option	Effect
`-O1`	Optimizes for code size by turning off software pipelining. Enables the same optimizations as `-O` except for loop unrolling and software pipelining. `-O` and `-O2` turn on software pipelining. Generally, `-O` or `-O2` are recommended over `-O1`.

IA-32 Compiler

Option	Effect
`-O`, `-O1`, `-O2`	Optimize for speed. Disable option `-fp`. The `-O2` option is ON by default. Intrinsic recognition is disabled.
`-O3`	Enables `-O2` option with more aggressive optimization. Optimizes for maximum speed, but does not guarantee higher performance unless loop and memory access transformation take place. In conjunction with `-axK` and `-xK` options (IA-32 only), this option causes the compiler to perform more aggressive data dependency analysis than for `-O2`. This may result in longer compilation times.

IA-32 and Itanium Compilers

Option	Effect
`-O2`	ON by default. `-O2` turns ON intrinsics inlining. Enables the following capabilities for performance gain: • Constant propagation • Copy propagation • Dead-code elimination • Global register allocation • Global instruction scheduling and control speculation • Loop unrolling • Optimized code selection • Partial redundancy elimination • Strength reduction/induction variable simplification • Variable renaming • Exception handling optimizations • Tail recursions • Peephole optimizations • Structure assignment lowering and optimizations • Dead store elimination
`-O3`	Enables `-O2` option with more aggressive optimization, for example, prefetching, scalar replacement, and loop transformations. Optimizes for maximum speed, but does not guarantee higher performance unless loop and memory access transformation take place. To time your application, see Timing Your Application.

Restricting Optimizations

The following options restrict or preclude the compiler's ability to optimize your program:

Option	Description
`-O0`	Disables all optimizations.
`-mp1`	Improve floating-point precision. Speed impact is less than with `-mp`.
`-fp` IA-32 only	Disable using the EBP register as a general purpose register.
`-prec_div` IA-32 only	Disables the floating point division-to-multiplication optimization.
`-fp_port` IA-32 only	Round fp results at assignments and casts (some speed impact).
`-ftz[-]` Itanium-based systems only	Enable [disable] flush denormal results to zero. The `-ftz` option is OFF by default, but turned ON with `-O3`.
`-IPF_fma[-]` Itanium-based systems only	Enable [disable] the combining of floating point multiplies and add/subtract operations.
`-IPF_fltacc[-]` Itanium-based systems only	Enable [disable] optimizations that affect floating point accuracy.
`-IPF_flt_eval_method0` Itanium-based systems only	Floating-point operands evaluated to the precision indicated by program.
`-IPF_fp_speculation<mode>` Itanium-based systems only	Enable floating point speculations with the following `<mode>` conditions: • `fast` - speculate floating point operations • `safe` - speculate only when safe • `strict` - same as off • `off` - disables speculation of floating-point operations

Note

You can turn off all optimizations for specific functions by using `#pragma optimize`. In the following example, all optimization is turned off for function `foo()`:

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

Valid second arguments for `#pragma optimize` are "`on`" or "`off`." With the "`on`" argument, `foo()` is compiled with the same optimization as the rest of the program. The compiler ignores first argument values.

Floating-point Arithmetic Precision

Options for IA-32 and Itanium®-based Systems

-mp Option

The `-mp` option restricts optimization to maintain declared precision and to ensure that floating-point arithmetic conforms more closely to the ANSI and IEEE standards. For most programs, specifying this option adversely affects performance. If you are not sure whether your application needs this option, try compiling and running your program both with and without it to evaluate the effects on both performance and precision. Specifying the `-mp` option has the following effects on program compilation:

- User variables declared as floating-point types are not assigned to registers.
- Whenever an expression is spilled (moved from a register to memory), it is spilled as 80 bits (extended precision), not 64 bits (double precision).
- Floating-point arithmetic comparisons conform to the IEEE 754 specification except for `NaN` behavior.
- The exact operations specified in the code are performed. For example, division is never changed to multiplication by the reciprocal.
- The compiler performs floating-point operations in the order specified without reassociation.
- The compiler does not perform the constant-folding optimization on floating-point values. Constant folding also eliminates any multiplication by 1, division by 1, and addition or subtraction of 0. For example, code that adds 0.0 to a number is executed exactly as written. Compile-time floating-point arithmetic is not performed to ensure that floating-point exceptions are also maintained.
- Floating-point operations conform to ANSI C. When assignments to type `float` and `double` are made, the precision is rounded from 80 bits (extended) down to 32 bits (float) or 64 bits (double). When you do not specify `-mp`, the extra bits of precision are not always rounded before the variable is reused.
- Sets the `-nolib_inline` option, which disables inline functions expansion.

Note: The `-nolib_inline` and `-mp` options are active by default when you choose the `-Xc` (strict ANSI C conformance) option.

-mp1 Option

Use the `-mp1` option to improve floating-point precision. `-mp1` disables fewer optimizations and has less impact on performance than `-mp`.

Options for IA-32 Only

Caution

A change of the default precision control or rounding mode (for example, by using the `-pc32` flag or by user intervention) may affect the results returned by some of the mathematical functions.

-long_double Option

Use `-long_double` to change the size of the long double type to 80 bits. The Intel compiler's default `long double` type is 64 bits in size, the same as the `double` type. This option introduces a number of incompatibilities with other files compiled without this option and with calls to library routines. Therefore, Intel recommends that the use of `long double` variables be local to a single file when you compile with this option.

-prec_div Option

With some optimizations, such as `-xK` and `-xW`, the Intel® C++ Compiler changes floating-point division computations into multiplication by the reciprocal of the denominator. For example, A/B is computed as A x (1/B) to improve the speed of the computation. However, for values of B greater than 2^{126}, the value of 1/B is "flushed" (changed) to 0. When it is important to maintain the value of 1/B, use `-prec_div` to disable the floating-point division-to-multiplication optimization. The result of `-prec_div` is greater accuracy with some loss of performance.

-pc*n* Option

Use the `-pc`*n* option to enable floating-point significand precision control. Some floating-point algorithms are sensitive to the accuracy of the significand or fractional part of the floating-point value. For example, iterative operations like division and finding the square root can run faster if you lower the precision with the `-pc`*n* option. Set *n* to one of the following values to round the significand to the indicated number of bits:

- `-pc32`: 24 bits (single precision) -- See Caution statement above.
- `-pc64`: 53 bits (single precision)
- `-pc80`: 64 bits (single precision) -- Default

The default value for *n* is 80, indicating double precision. This option allows full optimization. Using this option does not have the negative performance impact of using the `-Op` option because only the fractional part of the floating-point value is affected. The range of the exponent is not affected. The `-pc`*n* option causes the compiler to change the floating point precision control when the `main()` function is compiled. The program that uses `-pc`*n* must use `main()` as its entry point, and the file containing `main()` must be compiled with `-pc`*n*.

-rcd Option

The Intel compiler uses the `-rcd` option to improve the performance of code that requires floating-point-to-integer conversions. The optimization is obtained by controlling the change of the rounding mode. The system default floating point rounding mode is round-to-nearest. This means that values are rounded during floating point calculations. However, the C language requires floating point values to be truncated when a conversion to an integer is involved. To do this, the compiler must change the rounding mode to truncation before each floating-point-to-integer conversion and change it back afterwards. The `-rcd` option disables the change to truncation of the rounding mode for all floating point calculations, including floating point-to-integer conversions. Turning on this option can improve performance, but floating point conversions to integer will not conform to C semantics.

-fp_port Option

The `-fp_port` option rounds floating-point results at assignments and casts. An impact on speed may result.

Floating-point Arithmetic Options for Itanium(R)-based Systems

The following options enable you to control the compiler optimizations for floating-point computations on Itanium®-based systems:

- `-ftz[-]`
- `-IPF_fma[-]`
- `-IPF_fp_speculation`*`mode`*
- `-IPF_flt_eval_method0`
- `-IPF_fltacc[-]`(Default:`-IPF_fltacc-`)

Flush Denormal Results to Zero

Use the `-ftz` option to flush denormal results to zero.

Contraction of FP Multiply and Add/Subtract Operations

`-IPF_fma[-]` enables [disables] the contraction of floating-point multiply and add/subtract operations into a single operation. Unless `-mp` is specified, the compiler contracts these operations whenever possible. The `-mp` option disables the contractions. `-IPF_fma` and `-IPF_fma-` can be used to override the default compiler behavior. For example, a combination of `-mp` and `-IPF_fma` enables the compiler to contract operations:

```
prompt>ecc -mp -IPF_fma prog.c
```

FP Speculation

`-IPF_fp_speculation`*`mode`* sets the compiler to speculate on floating-point operations in one of the following *`mode`*s:

- `fast`: sets the compiler to speculate on floating-point operations
- `safe`: enables the compiler to speculate on floating-point operations only when it is safe
- `strict`: disables the speculation of floating-point operations.
- `off`: disables the speculation on floating-point operations.

FP Operations Evaluation

`-IPF_flt_eval_method0` directs the compiler to evaluate the expressions involving floating-point operands in the precision indicated by the variable types declared in the program.

Controlling Accuracy of the FP Results

`-IPF_fltacc[-]` enables [disables] optimizations that affect floating-point accuracy. By default (`-IPF_fltacc-`) the compiler may apply optimizations that reduce floating-point accuracy. You may use `-IPF_fltacc` or `-mp` to improve floating-point accuracy, but at the cost of disabling some optimizations.

Processor Optimization

Processor Optimization for IA-32 only

The `-tpp{5|6|7}` options optimize your application's performance for a specific Intel processor. The resulting binary will also run on the other processors listed in the table below. The Intel® C++ Compiler includes gcc*-compatible versions of the `-tpp` options. These options are listed in the gcc* Version column.

Option	gcc* Version	Optimizes for
`-tpp5`	`-mcpu=pentium`	Intel® Pentium® processors
`-tpp6`	`-mcpu=pentiumpro`	Intel Pentium Pro, Intel Pentium II, and Intel Pentium III processors
`-tpp7`	`-mcpu=pentium4`	Intel Pentium 4 processors

Note
The `-tpp7` option is ON by default when you invoke `icc` or `icpc`.

Example

The invocations listed below all result in a compiled binary optimized for Pentium 4 and Intel® Xeon(TM) processors. The same binary will also run on Pentium, Pentium Pro, Pentium II, and Pentium III processors.

```
prompt>icc prog.c

prompt>icc -tpp7 prog.c

prompt>icc -mcpu=pentium4 prog.c
```

Processor Optimization (Itanium®-based Systems only)

The `-tpp{1|2}` options optimize your application's performance for a specific Intel® Itanium® processor. The resulting binary will also run on the processors listed in the table below. The Intel® C++ Compiler includes gcc*-compatible versions of the `-tpp` options. These options are listed in the gcc* Version column.

Option	gcc* Version	Optimizes for
`-tpp1`	`-mcpu=itanium`	Itanium® processors
`-tpp2`	`-mcpu=itanium2`	Itanium® 2 processors

Note
The `-tpp2` option is ON by default when you invoke `ecc` or `ecpc`.

Example

The invocations listed below all result in a compiled binary optimized for the Intel Itanium 2 processor. The same binary will also run on Intel Itanium processors.

```
prompt>ecc prog.c

prompt>ecc -tpp2 prog.c

prompt>ecc -mcpu=itanium2 prog.c
```

Processor-specific Optimization (IA-32 only)

The `-x{M|i|K|W}` options target your program to run on a specific IA-32 processor by specifying the minimum set of processor instructions required for the processor that executes your program. The resulting code can contain unconditional use of the specified processor instructions. The Intel® C++ Compiler includes gcc*-compatible versions of the `-x{i|M|K|W}` options. These options are listed in the "gcc Version" column.

Option	gcc* Version	Specific Optimization for...
`-xM`	`-march=pentiumii`	Intel® Pentium® processors with MMX(TM) technology
`-xi`	`-march=pentiumpro`	Intel Pentium Pro and Intel Pentium II processors
`-xK`	`-march=pentiumiii`	Intel Pentium III processors
`-xW`	`-march=pentium4`	Intel Pentium 4 processors, Intel® Xeon(TM) processors, and Intel® Pentium® M processors

To execute the program on x86 processors not provided by Intel Corporation, do not specify the `-x{M|i|K|W}` option.

Example

The invocation below compiles `prog.c` for processors that support the `K` set of instructions. The optimized binary will require a Pentium III, Pentium 4, Intel Xeon processor, or Intel Pentium M processor to execute correctly. The resulting binary may not execute correctly on a Pentium, Pentium Pro, Pentium II, Pentium with MMX technology processors, or on x86 processors not provided by Intel Corporation.

```
prompt> icc -xK prog.c
```

Caution
If a program compiled with `-x{M|i|K|W}` is executed on a processor that lacks the specified set of instructions, it can fail with an illegal instruction exception, or display other unexpected behavior.

Auto CPU Dispatch (IA-32 only)

The `-ax{M|i|K|W}` options direct the compiler to find opportunities to generate separate versions of functions that use instructions supported on the specified processors (see table below). If the compiler finds such an opportunity, it first checks whether generating a processor-specific version of a function results in a performance gain. If this is the case, the compiler generates both a processor-specific version of a function and a generic version of the function. The generic version will run on any IA-32 processor.

At run time, one of the two versions is chosen to execute, depending on the processor the program is currently running on. In this way, the program can benefit from performance gains on more advanced processors, while still working properly on older processors.

The disadvantages of using `-ax{M|i|K|W}` are:

- The size of the compiled binary increases because it contains both a processor-specific version and a generic version of the code.
- Performance is affected by the run-time checks to determine which code to run.

Note
Programs that you compile with this option will execute on any IA-32 processor. Such compilations are, however, subject to any exclusive specialized code restrictions you impose during compilation with the `-x` option.

Option	Optimizes for...
`-axM`	Intel® Pentium® processors with MMX(TM) technology
`-axi`	Intel Pentium Pro and Intel Pentium II processors
`-axK`	Intel Pentium III processors. Implies `i` and `M` instructions.
`-axW`	Intel Pentium 4 processors, Intel® Xeon(TM) processors, and Intel® Pentium® M processors. Implies `M`, `i`, and `K` instructions.

Example

The compilation below will generate a single executable that includes:

- A generic version for use on any x86-compatible processor.
- A version optimized for Intel Pentium III, as long as there is a performance benefit.
- A version optimized for Intel Pentium 4 processors, Intel Xeon processors, and Intel Pentium M processors, as long as there is a performance benefit.

```
prompt>icc -axKW prog.c
```

Combining Processor Optimization and Auto CPU Dispatch (IA-32 only)

The following table shows how to combine processor target and dispatch options to compile programs with different optimizations and exclusions. See Processor Legend below table.

Optimize exclusively for...	**...without excluding...**					
	A	**B**	**C**	**D**	**E**	**F**
A	`-tpp5`	`-tpp5`	`-tpp5`	`-tpp5`	`-tpp5`	`-tpp5`
B	N-A	`-tpp5 -xM`	`-tpp5`	`-tpp5 -xM`	`-tpp5 -xM`	`-tpp5 -xM`
C	N-A	N-A	`-tpp6 -xi`	`-tpp6 -xi`	`-tpp6 -xi`	`-tpp6 -xi`
D	N-A	N-A	N-A	`-tpp6 -xiM`	`-tpp6 -xiM`	`-tpp6 -xiM`
E	N-A	N-A	N-A	N-A	`-tpp6 -xK`	`-tpp6 -xK`
F	N-A	N-A	N-A	N-A	N-A	`-tpp7 -xW`

Processor Legend

- **A** - Intel® Pentium® processors
- **B** - Intel Pentium processors with MMX(TM) technology
- **C** - Intel Pentium Pro processors
- **D** - Intel Pentium II processors
- **E** - Intel Pentium III processors
- **F** - Intel Pentium 4 processors, Intel® Xeon(TM) processors, and Intel® Pentium® M processors

Example

If you wanted your program to

- Always require the MMX(TM) technology extensions
- Use Pentium Pro processor extensions when the processor it is run on offers it
- Not use them when it does not

use the following command line options:

```
prompt>icc -xM -axi prog.c
```

In this example, `-xM` restricts the application to running on Pentium processors with MMX(TM) technology or later processors. If you wanted the program to run on earlier generations of IA-32 processors as well, you would use the following command line:

```
prompt>icc -axiM prog.c
```

This compilation generates optimized code for processors that support both the `i` and `M` extensions, but the compiled program will run on any IA-32 processor.

Interprocedural Optimizations

Use `-ip` and `-ipo` to enable interprocedural optimizations (IPO), which allow the compiler to analyze your code to determine where to apply the optimizations listed in tables that follow.

IA-32 and Itanium®-based Applications

Optimization	**Affected Aspect of Program**
Inline function expansion	Calls, jumps, branches, and loops
Interprocedural constant propagation	Arguments, global variables, and return values
Monitoring module-level static variables	Further optimizations, loop invariant code
Dead code elimination	Code size
Propagation of function characteristics	Call deletion and call movement
Multifile optimization	Affects the same aspects as `-ip`, but across multiple files

IA-32 applications only

Optimization	**Affected Aspect of Program**
Passing arguments in registers	Calls, register usage
Loop-invariant code motion	Further optimizations, loop invariant code

Inline function expansion is one of the main optimizations performed by the interprocedural optimizer. For function calls that the compiler believes are frequently executed, the compiler might decide to replace the instructions of the call with code for the function itself (inline the call).

With `-ip`, the compiler performs inline function expansion for calls to functions defined within the current source file. However, when you use `-ipo` to specify multifile IPO, the compiler performs inline function expansion for calls to functions defined in separate files. For this reason, it is important to compile the entire application or multiple, related source files together when you specify `-ipo`.

The IPO optimizations are disabled by default.

Overview: Multifile IPO

Multifile IPO obtains potential optimization information from individual program modules of a multifile program. Using the information, the compiler performs optimizations across modules.

Building a program is divided into two phases -- compilation and linkage. Multifile IPO performs different work depending on whether the compilation, linkage, or both are performed.

Compilation Phase

As each source file is compiled, multifile IPO stores an intermediate representation (IR) of the source code in the object file, which includes summary information used for optimization.

By default, the compiler produces "mock" object files during the compilation phase of multifile IPO. Generating mock files instead of real object files reduces the time spent in the multifile IPO compilation phase. Each mock object file contains the IR for its corresponding source file, but no real code or data. These mock objects must be linked using the `-ipo` option and `icc`, or using the `xild` tool.

Note

Failure to link "mock" objects with `icc`, `-ipo`, or `xild` will result in linkage errors. There are situations where mock object files cannot be used. See Compilation with Real Object Files for more information.

Linkage Phase

When you specify `-ipo`, the compiler is invoked a final time before the linker. The compiler performs multifile IPO across all object files that have an IR.

Note

The compiler does not support multifile IPO for static libraries (`.a` files). See Compilation with Real Object Files for more information.

`-ipo` enables the driver and compiler to attempt detecting a whole program automatically. If a whole program is detected, the interprocedural constant propagation, stack frame alignment, data layout and padding of common blocks optimizations perform more efficiently, while more dead functions get deleted. This option is safe.

Compilation with Real Object Files

In certain situations you might need to generate real object files with `-ipo`. To force the compiler to produce real object files instead of "mock" ones with IPO, you must specify `-ipo_obj` in addition to `-ipo`.

Use of `-ipo_obj` is necessary under the following conditions:

- The objects produced by the compilation phase of `-ipo` will be placed in a static library without the use of `xild` or `xild -lib`. The compiler does not support multifile IPO for static libraries, so all static libraries are passed to the linker. Linking with a static library that contains "mock" object files will result in linkage errors because the objects do not contain real code or data. Specifying `-ipo_obj` causes the compiler to generate object files that can be used in static libraries.
- Alternatively, if you create the static library using `xild` or `xild -lib`, then the resulting static library will work as a normal library.
- The objects produced by the compilation phase of `-ipo` might be linked without the `-ipo` option and without the use of `xild`.
- You want to generate an assemblable file for each source file (using `-S`) while compiling with `-ipo`. If you use `-ipo` with `-S`, but without `-ipo_obj`, the compiler issues a warning and an empty assemblable file is produced for each compiled source file.

Creating a Multifile IPO Executable

This topic describes how to create a multifile IPO executable for compilations targeted for IA-32 and Itanium®-based systems.

Procedure for IA-32 Systems

If you separately compile and link your source modules with `-ipo`:

1. Compile your modules with `-ipo` as follows:
   `prompt>`**`icc -ipo -c a.c b.c c.c`**
2. Use the `-c` option to stop compilation after generating `.o` files. Each object file has the IR for the corresponding source file. With preceding results, you can now optimize interprocedurally:
   `prompt>`**`icc -ipo a.o b.o c.o`**

Multifile IPO is applied only to modules that have an IR, otherwise the object file passes to the link stage. For efficiency, combine steps 1 and 2:

```
prompt>icc -ipo a.c b.c c.c
```

Procedure for Itanium®-based Systems

If you separately compile and link your source modules with `-ipo`:

1. Compile your modules with `-ipo` as follows:
   `prompt>`**`ecc -ipo -c a.c b.c c.c`**
2. Use `-c` to stop compilation after generating `.o` files. Each object file has the IR for the corresponding source file. With preceding results, you can now optimize interprocedurally:
   `prompt>`**`ecc -ipo a.o b.o c.o`**

Multifile IPO is applied only to modules that have an IR, otherwise the object file passes to link stage. For efficiency, combine steps 1 and 2:

```
prompt>ecc -ipo a.c b.c c.c
```

See Using Profile-Guided Optimization: An Example for a description of how to use multifile IPO with profile information for further optimization.

Creating a Multifile IPO Executable with xild

The Intel linker, `xild`, performs the following steps:

- Invokes the Intel compiler to perform multifile IPO if objects containing `IR` are found.
- Invokes the GNU linker, `ld`, to link the application.

The command-line syntax for `xild` is:

```
prompt>xild [<options>] <LINK_commandline>
```

where:

- `[<options>]` (optional) may include any `gcc` linker options or options supported only by `xild`.
- `<LINK_commandline>` is the linker command line containing a set of valid arguments to `ld`.

To place the multifile IPO executable in `ipo_file`, use the option `-o`*`filename`*, for example:

```
prompt>xild oipo_file a.o b.o c.o
```

`xild` calls Intel compiler to perform IPO for objects containing IR and creates a new list of object(s) to be linked. Then `xild` calls `ld` to link the object files that are specified in the new list and produce `ipo_file` executable specified by the `-o`*`filename`* option.

Note

The `-ipo` option can reorder object files and linker arguments on the command line. Therefore, if your program relies on a precise order of arguments on the command line, `-ipo` can affect the behavior of your program.

Usage Rules

You must use the Intel linker `xild` to link your application if:

- Your source files were compiled with multifile IPO enabled. Multifile IPO is enabled by specifying the `-ipo` command-line option
- You normally would invoke `ld` to link your application.

The xild Options

The additional options supported by `xild` may be used to examine the results of multifile IPO. These options are described in the following table.

Option	Description
`-ipo_o[file.s]`	Produces assemblable files for the multifile IPO compilation. You may specify an optional name for the listing file, or a directory (with the backslash) in which to place the file. The default listing name is `ipo_out.s`.
`-ipo_o[file.o]`	Produces object file for the multifile IPO compilation. You may specify an optional name for the object file, or a directory (with the backslash) in which to place the file. The default object file name is `ipo_out.o`.
`-ipo_fcode-asm`	Add code bytes to assemblable files
`-ipo_fsource-asm`	Add high-level source code to assemblable files
`-ipo_fverbose-asm`, `-ipo_fnoverbose-asm`	Enable and disable, respectively, inserting comments containing version and options used in the assemblable file for `xild`

Creating a Library from IPO Objects

Normally, libraries are created using a library manager such as `ar`. Given a list of objects, the library manager will insert the objects into a named library to be used in subsequent link steps.

```
prompt>xiar cru user.a a.o b.o
```

A library named `user.a` will be created containing `a.o` and `b.o`.

If, however, the objects have been created using `-ipo -c`, then the objects will not contain a valid object but only the intermediate representation (IR) for that object file. For example:

```
prompt>icc -ipo -c a.c b.c
```

will produce `a.o` and `b.o` that only contains IR to be used in a link time compilation. The library manager will not allow these to be inserted in a library.

In this case you must use the Intel library driver `xild -ar`. This program will invoke the compiler on the IR saved in the object file and generate a valid object that can be inserted in a library.

```
prompt>xild -lib cru user.a a.o b.o
```

See Creating a Multifile IPO Executable Using `xild`.

Analyzing the Effects of Multifile IPO

The `-ipo_c` and `-ipo_S` options are useful for analyzing the effects of multifile IPO, or when experimenting with multifile IPO between modules that do not make up a complete program.

Use the `-ipo_c` option to optimize across files and produce an object file. This option performs optimizations as described for `-ipo`, but stops prior to the final link stage, leaving an optimized object file. The default name for this file is `ipo_out.o`.

Use the `-ipo_S` option to optimize across files and produce an assemblable file. This option performs optimizations as described for `-ipo`, but stops prior to the final link stage, leaving an optimized assemblable file. The default name for this file is `ipo_out.s`.

See Inline Expansion of Functions.

Using -ip or -ipo with -Qoption Specifiers

Use `-Qoption` with the applicable keywords to select particular inline expansions and loop optimizations. The option must be entered with a `-ip` or `-ipo` specification, as follows:

```
prompt>icc -ip -Qoption,tool,opts
```

where *`tool`* is `C++` (c) and *`opts`* are `-Qoption` specifiers (see below).

-option Specifiers

If you specify `-ip` or `-ipo` without any `-Qoption` qualification, the compiler

- Expands functions in line
- Propagates constant arguments
- Passes arguments in registers
- Monitors module-level static variables.

You can refine interprocedural optimizations by using the following `-Qoption` specifiers. To have an effect, the `-Qoption` option must be entered with either `-ip` or `-ipo` also specified, as in this example:

```
prompt>icc -ip -Qoption,c,ip_specifier
```

where *`ip_specifier`* is one of the specifiers described in the table below:

Specifer	**Description**
`-ip_args_in_regs=0`	Disables the passing of arguments in registers. By default, external functions can pass arguments in registers when called locally. Normally, only static functions can pass arguments in registers, provided the address of the function is not taken and the function does not use a variable number of arguments.
`-ip_ninl_max_stats=n`	Sets the valid max number of intermediate language statements for a function that is expanded in line. The number n is a positive integer. The number of intermediate language statements usually exceeds the actual number of source language statements. The default value for `n` is 230. The compiler uses a larger limit for user inline functions.
`-ip_ninl_min_stats=n`	Sets the valid `min` number of intermediate language statements for a function that is expanded in line. The number n is a positive integer. The default value for `ip_ninl_min_stats` is: • IA-32 compiler: `ip_ninl_min_stats` = 7 • Itanium® compiler: `ip_ninl_min_stats` = 15

`-ip_ninl_max_total_stats=n`	Sets the maximum increase in size of a function, measured in intermediate language statements, due to inlining. n is a positive integer whose default value is 2000.

The following command activates procedural and interprocedural optimizations on `source.c` and sets the maximum increase in the number of intermediate language statements to 5 for each function:

```
prompt>icc -ip -Qoption,c,-ip_ninl_max_stats=5 source.c
```

Controlling Inline Expansion of User Functions

The compiler enables you to control the amount of inline function expansion, with the options shown in the following summary:

`-ip_no_inlining`	This option is only useful if `-ip` is also specified. In this case, `-ip_no_inlining` disables inlining that would result from the `-ip` interprocedural optimizations, but has no effect on other interprocedural optimizations.
`ip_no_pinlining`	Disables partial inlining; can be used if `-ip` or `-ipo` is also specified.

Criteria for Inline Function Expansion

Once the criteria are met, the compiler picks the routines whose inline expansion will provide the greatest benefit to program performance. The inlining heuristics used by the compiler differ, based on whether or not you use profile-guided optimizations (`-prof_use`). When you use profile-guided optimizations with `-ip` or `-ipo`, the compiler uses the following heuristics:

- The default heuristic focuses on the most frequently executed call sites, based on the profile information gathered for the program.
- By default, the compiler will not inline functions with more than 230 intermediate statements. You can change this value by specifying the option `-Qoption,c,-ip_ninl_max_stats=`*`new_value`*. Note: there is a higher limit for functions declared by the user as `inline` or `__inline`.
- The default inline heuristic will stop inlining when direct recursion is detected.
- The default heuristic will always inline very small functions that meet the minimum inline criteria.
  - Default for Itanium®-based applications: `ip_ninl_min_stats=15`.
  - Default for IA-32 applications: `ip_ninl_min_stats=7`. This limit can be modified with the option `-Qoption,c,-ip_ninl_min_stats=`*`new_value`*.

If you do not use profile-guided optimizations with `-ip` or `-ipo`, the compiler uses less aggressive inlining heuristics:

- Inline a function if the inline expansion will not increase the size of the final program.
- Inline a function if it is declared with the `inline` or `__inline` keywords.

Overview: Profile-guided Optimizations

Profile-guided optimizations (PGO) tell the compiler which areas of an application are most frequently executed. By knowing these areas, the compiler is able to use feedback from a previous compilation to be more selective in optimizing the application. For example, the use of PGO often enables the compiler to make better decisions about function inlining, thereby increasing the effectiveness of interprocedural optimizations.

Instrumented Program

Profile-guided optimization creates an instrumented program from your source code and special code from the compiler. Each time this instrumented code is executed, the instrumented program generates a dynamic information file. When you compile a second time, the dynamic information files are merged into a summary file. Using the profile information in this file, the compiler attempts to optimize the execution of the most heavily travelled paths in the program.

Unlike other optimizations, such as those used strictly for size or speed, the results of IPO and PGO vary. This is due to each program having a different profile and different opportunities for optimizations. The guidelines provided here help you determine if you can benefit by using IPO and PGO.

Added Performance with PGO

Beginning with version 6.0 of the Intel® C++ Compiler, PGO is improved in the following ways:

- Register allocation uses the profile information to optimize the location of spill code.
- For direct function calls, branch prediction is improved by identifying the most likely targets. With the Pentium® 4 processor's longer pipeline, improved branch prediction translates to higher performance gains.
- The compiler detects and does not vectorize loops that execute only a small number of iterations, reducing the run time overhead that vectorization might otherwise add.

Profile-guided Optimizations Methodology

PGO works best for code with many frequently executed branches that are difficult to predict at compile time. An example is code that is heavy with error-checking in which the error conditions are false most of the time. The "cold" error-handling code can be placed such that the branch is rarely mispredicted. Eliminating the interleaving of "hot" and "cold" code improves instruction cache behavior. For example, the use of PGO often enables the compiler to make better decisions about function inlining, thereby increasing the effectiveness of interprocedural optimizations.

PGO Phases

The PGO methodology requires three phases:

- Phase 1: Instrumentation compilation and linking with `-prof_gen[x]`
- Phase 2: Instrumented execution by running the executable
- Phase 3: Feedback compilation with `-prof_use`

A key factor in deciding whether you want to use PGO lies in knowing which sections of your code are the most heavily used. If the data set provided to your program is very consistent and it elicits a similar behavior on every execution, then PGO can probably help optimize your program execution. However, different data sets can elicit different algorithms to be called. This can cause the behavior of your program to vary from one execution to the next.

In cases where your code behavior differs greatly between executions, PGO may not provide noticeable benefits. You have to ensure that the benefit of the profile information is worth the effort required to maintain up-to-date profiles.

Basic PGO Options

Option	**Description**
`-prof_gen[x]`	Instructs the compiler to produce instrumented code in your object files in preparation for instrumented execution.
`-prof_use`	Instructs the compiler to produce a profile-optimized executable and merges available dynamic information (`.dyn`) files into a `pgopti.dpi` file.

In cases where your code behavior differs greatly between executions, you have to ensure that the benefit of the profile information is worth the effort required to maintain up-to-date profiles. In the basic profile-guided optimization, the following options are used in the phases of the PGO:

Generating Instrumented Code

The `-prof_gen[x]` option instruments the program for profiling to get the execution count of each basic block. It is used in Phase 1 of the PGO to instruct the compiler to produce instrumented code in your object files in preparation for instrumented execution. Parallel make is automatically supported for `-prof_genx` compilations.

Generating a Profile-optimized Executable

The `-prof_use` option is used in Phase 3 of the PGO to instruct the compiler to produce a profile-optimized executable and merges available dynamic-information (`.dyn`) files into a `pgopti.dpi` file.

Note

The dynamic-information files are produced in Phase 2 when you run the instrumented executable.

If you perform multiple executions of the instrumented program, `-prof_use` merges the dynamic-information files again and overwrites the previous `pgopti.dpi` file.

Disabling Function Splitting (Itanium® Compiler only)

`-fnsplit-` disables function splitting. Function splitting is enabled by `-prof_use` in Phase 3 to improve code locality by splitting routines into different sections: one section to contain the cold or very infrequently executed code and one section to contain the rest of the code (hot code).

You can use `-fnsplit-` to disable function splitting for the following reasons:

- Most importantly, to get improved debugging capability. In the debug symbol table, it is difficult to represent a split routine, that is, a routine with some of its code in the hot code section and some of its code in the cold code section.
- The `-fnsplit-` option disables the splitting within a routine but enables function grouping, an optimization in which entire routines are placed either in the cold code section or the hot code section. Function grouping does not degrade debugging capability.
- Another reason can arise when the profile data does not represent the actual program behavior, that is, when the routine is actually used frequently rather than infrequently.

Note

For Itanium®-based applications, if you intend to use the `-prof_use` option with optimizations at the `-O3` level, the `-O3` option must be on. If you intend to use the `-prof_use` option with optimizations at the `-O2` level or lower, you can generate the profile data with the default options.

Example of Profile-guided Optimization

The three basic phases of PGO are:

- Instrumentation Compilation and Linking
- Instrumented Execution
- Feedback Compilation

Instrumentation Compilation and Linking

Use `-prof_gen` to produce an executable with instrumented information. Use also the `-prof_dir` option as recommended for most programs, especially if the application includes the source files located in multiple directories. `-prof_dir` ensures that the profile information is generated in one consistent place. For example:

IA-32 Systems

```
prompt>icc -prof_gen -prof_dirc:\profdata -c a1.c a2.c a3.c
prompt>icc a1.o a2.o a3.o
```

Itanium®-based Systems

```
prompt>ecc -prof_gen -prof_dirc:\profdata -c a1.c a2.c a3.c
prompt>ecc a1.o a2.o a3.o
```

In place of the second command, you could use the linker directly to produce the instrumented program.

Instrumented Execution

Run your instrumented program with a representative set of data to create a dynamic information file.

```
prompt>./a.o
```

The resulting dynamic information file has a unique name and `.dyn` suffix every time you run `a.o`. The instrumented file helps predict how the program runs with a particular set of data. You can run the program more than once with different input data.

Feedback Compilation

Compile and link the source files with `-prof_use` to use the dynamic information to optimize your program according to its profile:

IA-32 Systems:

```
prompt>icc -prof_use -ipo a1.c a2.c a3.c
```

Itanium®-based Systems:

```
prompt>ecc -prof_use -ipo a1.c a2.c a3.c
```

Besides the optimization, the compiler produces a `pgopti.dpi` file. You typically specify the default optimizations (`-O2`) for phase 1, and specify more advanced optimizations with `-ipo` for phase 3. This example used `-O2` in phase 1 and `-O2 -ipo` in phase 3.

Note

The compiler ignores the `-ipo` options with `-prof_gen[x]`. With the `x` qualifier, extra information is gathered.

PGO Environment Variables

The table below describes environment values to determine the directory to store dynamic information files or whether to overwrite `pgopti.dpi`. Refer to your operating system documentation for instructions on how to specify environment values.

Profile-guided Optimization Environment Variables

Variable	Description
`PROF_DIR`	Specifies the directory in which dynamic information files are created. This variable applies to all three phases of the profiling process.
`PROF_NO_CLOBBER`	Alters the feedback compilation phase slightly. By default, during the feedback compilation phase, the compiler merges the data from all dynamic information files and creates a new `pgopti.dpi` file if `.dyn` files are newer than an existing `pgopti.dpi` file. When this variable is set, the compiler does not overwrite the existing `pgopti.dpi` file. Instead, the compiler issues a warning and you must remove the `pgopti.dpi` file if you want to use additional dynamic information files.

Using profmerge to Relocate the Source Files

The compiler uses the full path to the source file to look up profile summary information. By default, this prevents you from:

- Using the profile summary file (`.dpi`) if you move your application sources.
- Sharing the profile summary file with another user who is building identical application sources that are located in a different directory.

Source Relocation

To enable the movement of application sources, as well as the sharing of profile summary files, use `profmerge` with the `-src_old` and `-src_new` options. For example:

IA-32 Systems: `prompt>`**`profmerge -prof_dir`** `<p1>` **`-src_old`** `<p2>` **`-src_new`** `<p3>`

Itanium®-based Systems: `prompt>`**`profmerge -em -p64 -prof_dir`** `<p1>` **`-src_old`** `<p2>` **`-src_new`** `<p3>`

where:

- <p1> is the full path to dynamic information file (`.dpi`).
- <p2> is the old full path to source files.
- <p3> is the new full path to source files.

The above command will read the `pgopti.dpi` file. For each function represented in the `pgopti.dpi` file, whose source path begins with the <p2> prefix, `profmerge` replaces that prefix with <p3>. The `pgopti.dpi` file is updated with the new source path information.

Notes

- You can execute `profmerge` more than once on a given `pgopti.dpi` file. You may need to do this if the source files are located in multiple directories. For example:

  ```
  profmerge -prof_dir -src_old /src/prog_1 -src_new /src/prog_2

  profmerge -prof_dir -src_old /proj_1 -src_new /proj_2
  ```

- In the values specified for `-src_old` and `-src_new`, uppercase and lowercase characters are treated as identical. Likewise, forward slash (`/`) and backward slash (`\`) characters are treated as identical.
- Because the source relocation feature of `profmerge` modifies the `pgopti.dpi` file, you may wish to make a backup copy of the file prior to performing the source relocation.

PGO API Support Overview

Profile Information Generation Support lets you control of the generation of profile information during the instrumented execution phase of profile-guided optimizations. Normally, profile information is generated by an instrumented application when it terminates by calling the standard `exit()` function. The functions described in this section may be necessary in assuring that profile information is generated in the following situations:

- When the instrumented application exits using a non-standard exit routine
- When instrumented application is a non-terminating application where `exit()` is never called
- When you want control of when the profile information is generated

This section includes descriptions of the functions and environment variable that comprise Profile Information Generation Support. The functions are available by inserting `#include <pgouser.h>` at the top of any source file where the functions may be used.

The compiler sets a `define` for _PGO_INSTRUMENT when you compile with either `-prof_gen` or `-prof_genx`.

Dumping Profile Information

```
void _PGOPTI_Prof_Dump(void);
```

Description

This function dumps the profile information collected by the instrumented application. The profile information is recorded in a `.dyn` file.

Recommended Usage

Insert a single call to this function in the body of the function which terminates your application. Normally, `_PGOPTI_Prof_Dump` should be called just once. It is also possible to use this function in conjunction with `_PGOPTI_Prof_Reset()` to generate multiple `.dyn` files (presumably from multiple sets of input data).

Example

```
// Selectively collect profile information for the portion
// of the application involved in processing input data.

input_data = get_input_data();

while(input_data)
{
   _PGOPTI_Prof_Reset();
   process_data(input_data);
   _PGOPTI_Prof_Dump();
   input_data = get_input_data();
}
```

Resetting the Dynamic Profile Counters

```
void _PGOPTI_Prof_Reset(void);
```

Description

This function resets the dynamic profile counters.

Recommended Usage

Use this function to clear the profile counters prior to collecting profile information on a section of the instrumented application. See the example under `PGOPTI_Prof_Dump()`.

Dumping and Resetting Profile Information

```
void _PGOPTI_Prof_Dump_And_Reset(void);
```

Description

This function may be called more than once. Each call will dump the profile information to a new `.dyn` file. The dynamic profile counters are then reset, and execution of the instrumented application continues.

Recommended Usage

Periodic calls to this function allow a non-terminating application to generate one or more profile information files. These files are merged during the feedback phase of profile-guided optimization. The direct use of this function allows your application to control precisely when the profile information is generated.

Interval Profile Dumping

```
void _PGOPTI_Set_Interval_Prof_Dump(int interval);
```

Description

This function activates Interval Profile Dumping and sets the approximate frequency at which dumps will occur. The `interval` parameter is measured in milliseconds and specifies the time interval at which profile dumping will occur. For example, if `interval` is set to 5000, then a profile dump and reset will occur approximately every 5 seconds. The interval is approximate because the time check controlling the dump and reset is only performed upon entry to any instrumented function in your application.

Note

- Setting `interval` to zero or a negative number will disable interval profile dumping.
- Setting `interval` to a very small value may cause the instrumented application to spend nearly all of its time dumping profile information. Be sure to set `interval` to a large enough value so that the application can perform actual work and collect substantial profile information.

Recommended Usage

Call this function at the start of a non-terminating application to initiate Interval Profile Dumping. Note that an alternative method of initiating Interval Profile Dumping is by setting the environment variable, `PROF_DUMP_INTERVAL`, to the desired `interval` value prior to starting the application. The intention of Interval Profile Dumping is to allow a non-terminating application to be profiled with minimal changes to the application source code.

Environment Variable

```
PROF_DUMP_INTERVAL
```

This environment variable may be used to initiate Interval Profile Dumping in an instrumented application. See the Recommended Usage of `_PGOPTI_Set_Interval_Prof_Dump` for more information.

HLO Overview

High-level optimizations (HLO) exploit the properties of source code constructs, such as loops and arrays, in the applications developed in high-level programming languages, such as C++. They include loop interchange, loop fusion, loop unrolling, loop distribution, unroll-and-jam, blocking, data prefetch, scalar replacement, data layout optimizations, and others. The option that turns on the high-level optimizations is `-O3`.

IA-32 and Itanium®-based applications	
`-O3`	Enable `-O2` option plus more aggressive optimizations, for example, loop transformation and prefetching. `-O3` optimizes for maximum speed, but may not improve performance for some programs.
IA-32 applications	
`-O3`	In addition, in conjunction with the vectorization options, `-ax{M|K|W}` and `-x{M|K|W}`, `-O3` causes the compiler to perform more aggressive data dependency analysis than for `-O2`. This may result in longer compilation times.

Loop Transformations

All these transformations are supported by data dependence. These techniques also include induction variable elimination, constant propagation, copy propagation, forward substitution, and dead code elimination. The loop transformation techniques include:

- Loop normalization
- Loop reversal
- Loop interchange and permutation
- Loop skewing
- Loop distribution
- Loop fusion
- Scalar replacement

In addition to the loop transformations listed for both IA-32 and Itanium® architectures above, the Itanium architecture allows collapsing techniques.

Loop Unrolling

You can unroll loops and specify the maximum number of times you want the compiler to do so.

How to Enable Loop Unrolling

You use the `-unroll[n]` option to unroll loops. *n* determines the maximum number of times for the unrolling operation. This applies only to loops that the compiler determines should be unrolled. Omit *n* to let the compiler decide whether to perform unrolling or not.

The following example unrolls a loop at most four times:

IA-32 Systems: `prompt>`**`icc -unroll4 a.c`**

When specifying high values to unroll loops, be aware that your application may exhaust certain resources, such as registers, that can slow program performance. You should consider timing your application (see Timing Your Application) if you specify high values to unroll loops.

How to Disable Loop Unrolling

Disable loop unrolling by setting *n* to 0. The following example disables loop unrolling:

IA-32 Systems: `prompt>`**`icc -unroll0 a.c`**

Absence of Loop-carried Memory Dependency with IVDEP Directive

For Itanium®-based applications, the `-ivdep_parallel` option indicates there is absolutely no loop-carried memory dependency in the loop where `IVDEP` directive is specified. This technique is useful for some sparse matrix applications. For example, the following loop requires `-ivdep_parallel` in addition to the directive `IVDEP` to indicate there is no loop-carried dependencies.

Example

```
#pragma ivdep

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

The following example shows that using this option and the `IVDEP` directive ensures there is no loop-carried dependency for the store into `a().`

Example

```
#pragma ivdep

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

Overview: Parallelization Options

For parallel programming, the Intel® C++ Compiler supports both the OpenMP* 2.0 API and an automatic parallelization capability. The following table lists the options that perform OpenMP and auto-parallelization support.

Option	Description
`-openmp`	Enables the parallelizer to generate multithreaded code based on the OpenMP directives. Default: OFF.
`-openmp_report{0|1|2}`	Controls the OpenMP parallelizer's diagnostic levels. Default: `-openmp_report1`.
`-openmp_stubs`	Enables compilation of OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked. Default: OFF.
`-parallel`	Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. Default: OFF.
`-par_threshold{n}`	Sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel, n=0 to 100. n=0 implies "always." Default: `-par_threshold75`.
`-par_report{0|1|2|3}`	Controls the auto-parallelizer's diagnostic levels. Default: `-par_report1`

Note

When both `-openmp` and `-parallel` are specified on the command line, the `-parallel` option is honored only in routines that do not contain OpenMP directives. For routines that contain OpenMP directives, only the `-openmp` option is honored.

Overview: Parallelization with OpenMP*

The Intel® C++ Compiler supports the OpenMP* C++ version 2.0 API specification. OpenMP provides symmetric multiprocessing (SMP) with the following major features:

- Relieves the user from having to deal with the low-level details of iteration space partitioning, data sharing, and thread scheduling and synchronization.
- Provides the benefit of the performance available from shared memory, multiprocessor systems.

The Intel C++ Compiler performs transformations to generate multithreaded code based on the user's placement of OpenMP directives in the source program making it easy to add threading to existing software. The Intel compiler supports all of the current industry-standard OpenMP directives, except `WORKSHARE`, and compiles parallel programs annotated with OpenMP directives. In addition, the Intel C++ Compiler provides Intel-specific extensions to the OpenMP C++ version 2.0 specification including run-time library routines and environment variables.

Note

As with many advanced features of compilers, you must properly understand the functionality of the OpenMP directives in order to use them effectively and avoid unwanted program behavior.

See parallelization options summary for all of the options of the OpenMP feature in the Intel C++ Compiler.

For complete information on the OpenMP standard, visit the OpenMP Web site at http://www.openmp.org. For OpenMP* C++ version 2.0 API specifications, see http://www.openmp.org/specs/.

Parallel Processing with OpenMP

To compile with OpenMP, you need to prepare your program by annotating the code with OpenMP directives. The Intel C++ Compiler first processes the application and produces a multithreaded version of the code which is then compiled. The output is a executable program with the parallelism implemented by threads that execute parallel regions or constructs.

Performance Analysis

For performance analysis of your program, you can use the Intel® VTune™ Performance Analyzer to show performance information. You can obtain detailed information about which portions of the code require the largest amount of time to execute and where parallel performance problems are located.

Parallel Processing Thread Model

This topic explains the processing of the parallelized program and adds more definitions of the terms used in parallel programming.

The Execution Flow

As previously mentioned, a program containing OpenMP* C++ API compiler directives begins execution as a single process, called the **master** thread of execution. The master thread executes sequentially until the first **parallel construct** is encountered.

In the OpenMP C++ API, the `#pragma omp parallel` directive defines the parallel construct. When the master thread encounters a parallel construct, it creates a **team** of threads, with the master thread becoming the master of the team. The program statements enclosed by the parallel construct are executed in parallel by each thread in the team. These statements include routines called from within the enclosed statements.

The statements enclosed lexically within a construct define the **static extent** of the construct. The **dynamic extent** includes the static extent as well as the routines called from within the construct. When the `#pragma omp parallel` directive reaches completion, the threads in the team synchronize, the team is dissolved, and only the master thread continues execution. The other threads in the team enter a wait state. You can specify any number of parallel constructs in a single program. As a result, thread teams can be created and dissolved many times during program execution.

Using Orphaned Directives

In routines called from within parallel constructs, you can also use directives. Directives that are not in the lexical extent of the parallel construct, but are in the dynamic extent, are called orphaned directives. Orphaned directives allow you to execute major portions of your program in parallel with only minimal changes to the sequential version of the program. Using this functionality, you can code parallel constructs at the top levels of your program and use directives to control execution in any of the called routines. For example:

```
int main(void)
{
     ...
     #pragma omp parallel
     {
          phase1();
     }
}



void phase1(void)
{
     ...

     #pragma omp for private(i) shared(n)
          for(i=0; i < n; i++)
          {
               some_work(i);
          }
}
```

This is an orphaned directive because the parallel region is not lexically present.

Data Environment Directive

A data environment directive controls the data environment during the execution of parallel constructs. You can control the data environment within parallel and worksharing constructs. Using directives and data environment clauses on directives, you can:

- Privatize scope variables by using the `THREADPRIVATE` directive
- Control data scope attributes by using the `THREADPRIVATE` directive's clauses. The data scope attribute clauses are:
  - COPYIN
  - DEFAULT
  - PRIVATE
  - FIRSTPRIVATE
  - LASTPRIVATE
  - REDUCTION
  - SHARED

You can use several directive clauses to control the data scope attributes of variables for the duration of the construct in which you specify them. If you do not specify a data scope attribute clause on a directive, the default is `SHARED` for those variables affected by the directive.

Pseudo Code of the Parallel Processing Model

A sample pseudo program using some of the more common OpenMP directives is shown in the code example that follows. This example also indicates the difference between serial regions and parallel regions.

```
main() {                  // Begin serial execution
 ...                      // Only the master thread executes
#pragma omp parallel      // Begin a Parallel Construct, form
 {                        // a team. This is Replicated Code
  ...                     // (each team member executes
  ...                     // the same code)
                          //
#pragma omp sections      // Begin a Worksharing Construct
 {                        //
  #pragma omp section     // One unit of work
   {...}                  //
  #pragma omp section     // Another unit of work
   {...}                  //
 }                        // Wait until both units of work complete
 ...                      // More Replicated Code
                          //
 #pragma omp for nowait   // Begin a Worksharing Construct;
  for(...) {              // each iteration is unit of work
                          //
   ...                    // Work is distributed among the team members
                          //
 }                        // End of Worksharing Construct;
                          // nowait was specified, so
                          // threads proceed
                          //
 #pragma omp critical     // Begin a Critical Section
 {                        //
  ...                     // Replicated Code, but only one
                          // thread can execute it at a
 }                        // given time
 ...                      // More Replicated Code
```

```
                          //
   #pragma omp barrier    // Wait for all team members to arrive
   ...                    // More Replicated Code
                          //
  }                       // End of Parallel Construct;
                          // disband team and continue
                          // serial execution
                          //
  ...                     // Possibly more Parallel constructs
                          //
  }                       // End serial execution
```

Compiling with OpenMP, Directive Format, and Diagnostics

To run the Intel® C++ Compiler in OpenMP* mode, invoke the compiler with the `-openmp` option:

IA-32 applications: **`icc -openmp`** `input_file`

Itanium®-based applications: **`ecc -openmp`** `input_file`

Before you run the multithreaded code, you can set the number of desired threads in the OpenMP environment variable, `OMP_NUM_THREADS`. See OpenMP Environment Variables for further information.

-openmp Option

The `-openmp` option enables the parallelizer to generate multithreaded code based on the OpenMP directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems. The `-openmp` option works with both `-O0` (no optimization) and any optimization level of `-O1`, `-O2` (default) and `-O3`. Specifying `-O0` with `-openmp` helps to debug OpenMP applications.

OpenMP Directive Format and Syntax

An OpenMP directive has the form:

```
#pragma omp directive-name [clause, ...] newline
```

where:

- `#pragma omp` -- Required for all OpenMP directives.
- `directive-name` -- A valid OpenMP directive. Must appear after the `pragma` and before any clauses.
- `clause` -- Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.
- `newline` -- Required. Proceeds the structured block which is enclosed by this directive.

OpenMP Diagnostics

The `-openmp_report{0|1|2}` option controls the OpenMP parallelizer's diagnostic levels `0`, `1`, or 2 as follows:

- `-openmp_report0` = no diagnostic information is displayed.
- `-openmp_report1` = display diagnostics indicating loops, regions, and sections successfully parallelized.
- `-openmp_report2` = same as `-openmp_report1` plus diagnostics indicating `MASTER` constructs, `SINGLE` constructs, `CRITICAL` constructs, `ORDERED` constructs, `ATOMIC` directives, etc. are successfully handled.

The default is `-openmp_report1`.

OpenMP* Directives and Clauses

OpenMP Directives

Directive Name	Description
`parallel`	Defines a parallel region.
`for`	Identifies an iterative work-sharing construct that specifies a region in which the iterations of the associated loop should be executed in parallel.
`sections`	Identifies a non-iterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team.
`single`	Identifies a construct that specifies that the associated structured block is executed by only one thread in the team.
`parallel for`	A shortcut for a `parallel` region that contains a single `for` directive. **Note** The `parallel` or `for` OpenMP directive must be immediately followed by a `for` statement. If you place other statement or an OpenMP directive between the `parallel` or `for` directive and the `for` statement, the Intel C++ Compiler issues a syntax error.
`parallel sections`	Provides a shortcut form for specifying a parallel region containing a single `sections` directive.
`master`	Identifies a construct that specifies a structured block that is executed by the master thread of the team.
`critical[lock]`	Identifies a construct that restricts execution of the associated structured block to a single thread at a time.
`barrier`	Synchronizes all the threads in a team.
`atomic`	Ensures that a specific memory location is updated atomically.
`flush`	Specifies a "cross-thread" sequence point at which the implementation is required to ensure that all the threads in a team have a consistent view of certain objects in memory.
`ordered`	The structured block following an `ordered` directive is executed in the order in which iterations would be executed in a sequential loop.
`threadprivate`	Makes the named file-scope or namespace-scope variables specified private to a thread but file-scope visible within the thread.

OpenMP Clauses

Clause	Description
`private`	Declares variables to be `private` to each thread in a team.
`firstprivate`	Provides a superset of the functionality provided by the `private` clause.
`lastprivate`	Provides a superset of the functionality provided by the `private` clause.
`shared`	Shares variables among all the threads in a team.
`default`	Enables you to affect the data-scope attributes of variables.
`reduction`	Performs a reduction on scalar variables.
`ordered`	The structured block following an `ordered` directive is executed in the order in which iterations would be executed in a sequential loop.
`if`	If the `if(scalar_logical_expression)` clause is present, the enclosed code block is executed in parallel only if the `scalar_logical_expression` evaluates to TRUE. Otherwise the code block is serialized.
`schedule`	Specifies how iterations of the `for` loop are divided among the threads of the team.
`copyin`	Provides a mechanism to assign the same name to `threadprivate` variables for each thread in the team executing the parallel region.

OpenMP* Support Libraries

The Intel® C++ Compiler with OpenMP* support provides a production support library, `libguide.a`. This library enables you to run an application under different execution modes. It is used for normal or performance-critical runs on applications that have already been tuned.

Execution modes

The Intel compiler with OpenMP enables you to run an application under different execution modes that can be specified at run time. The libraries support the serial, turnaround, and throughput modes. These modes are selected by using the `KMP_LIBRARY` environment variable at run time.

Serial

The serial mode forces parallel applications to run on a single processor.

Turnaround

In a dedicated (batch or single user) parallel environment where all processors are exclusively allocated to the program for its entire run, it is most important to effectively utilize all of the processors all of the time. The turnaround mode is designed to keep active all of the processors involved in the parallel computation in order to minimize the execution time of a single job. In this mode, the worker threads actively wait for more parallel work, without yielding to other threads.

Note

Avoid over-allocating system resources. This occurs if either too many threads have been specified, or if too few processors are available at run time. If system resources are over-allocated, this mode will cause poor performance. The throughput mode should be used instead if this occurs.

Throughput

In a multi-user environment where the load on the parallel machine is not constant or where the job stream is not predictable, it may be better to design and tune for throughput. This minimizes the total time to run multiple jobs simultaneously. In this mode, the worker threads will yield to other threads while waiting for more parallel work.

The throughput mode is designed to make the program aware of its environment (that is, the system load) and to adjust its resource usage to produce efficient execution in a dynamic environment. Throughput mode is the default.

OpenMP* Environment Variables

This topic describes the OpenMP* environment variables (with the OMP_ prefix) and Intel-specific environment variables (with the KMP_ prefix).

Standard Environment Variables

Variable	Description	Default
OMP_SCHEDULE	Sets the runtime schedule type and chunk size.	STATIC (no chunk size specified)
OMP_NUM_THREADS	Sets the number of threads to use during execution.	Number of processors
OMP_DYNAMIC	Enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.	FALSE
OMP_NESTED	Enables (TRUE) or disables (FALSE) nested parallelism.	FALSE

Intel Extension Environment Variables

Environment Variable	Description	Default
KMP_LIBRARY	Selects the OpenMP run-time library throughput. The options for the variable value are: serial, turnaround, or throughput indicating the execution mode. The default value of throughput is used if this variable is not specified.	throughput (execution mode)
KMP_STACKSIZE	Sets the number of bytes to allocate for each parallel thread to use as its private stack. Use the optional suffix b, k, m, g, or t, to specify bytes, kilobytes, megabytes, gigabytes, or terabytes.	IA-32: 2m Itanium® compiler: 4m

OpenMP* Run-time Library Routines

OpenMP* provides several run-time library functions to assist you in managing your program in parallel mode. Many of these functions have corresponding environment variables that can be set as defaults. The run-time library functions enable you to dynamically change these factors to assist in controlling your program. In all cases, a call to a run-time library function overrides any corresponding environment variable.

The following table specifies the interfaces to these routines. The names for the routines are in user name space. The `omp.h` and `omp_lib.h` header files are provided in the `INCLUDE` directory of your compiler installation.

There are definitions for two different locks, `omp_lock_kind` and `omp_nest_lock_kind`, which are used by the functions in the table that follows:

Function	**Description**
Execution Environment Routines	
`omp_set_num_threads(`*`nthreads`*`)`	Sets the number of threads to use for subsequent parallel regions.
`omp_get_num_threads()`	Returns the number of threads that are being used in the current parallel region.
`omp_get_max_threads()`	Returns the maximum number of threads that are available for parallel execution.
`omp_get_thread_num()`	Returns the unique thread number of the thread currently executing this section of code.
`omp_get_num_procs()`	Returns the number of processors available to the program.
`omp_in_parallel()`	Returns `TRUE` if called within the dynamic extent of a parallel region executing in parallel; otherwise returns `FALSE`.
`omp_set_dynamic(`*`dynamic_threads`*`)`	Enables or disables dynamic adjustment of the number of threads used to execute a parallel region. If *`dynamic_threads`* is `TRUE`, dynamic threads are enabled. If *`dynamic_threads`* is `FALSE`, dynamic threads are disabled. Dynamics threads are disabled by default.
`omp_get_dynamic()`	Returns `TRUE` if dynamic thread adjustment is enabled, otherwise returns `FALSE`.
`omp_set_nested(`*`nested`*`)`	Enables or disables nested parallelism. If *`nested`* is `TRUE`, nested parallelism is enabled. If *`nested`* is `FALSE`, nested parallelism is disabled. Nested parallelism is disabled by default.
`omp_get_nested()`	Returns `TRUE` if nested parallelism is enabled, otherwise returns `FALSE`.

Lock Routines	
`omp_init_lock(`*`lock`*`)`	Initializes the lock associated with *`lock`* for use in subsequent calls.
`omp_destroy_lock(`*`lock`*`)`	Causes the lock associated with *`lock`* to become undefined.
`omp_set_lock(`*`lock`*`)`	Forces the executing thread to wait until the lock associated with *`lock`* is available. The thread is granted ownership of the lock when it becomes available.
`omp_unset_lock(`*`lock`*`)`	Releases the executing thread from ownership of the lock associated with *`lock`*. The behavior is undefined if the executing thread does not own the lock associated with *`lock`*.
`omp_test_lock(`*`lock`*	Attempts to set the lock associated with *`lock`*. If successful, returns `TRUE`, otherwise returns `FALSE`.
`omp_init_nest_lock(`*`lock`*`)`	Initializes the nested lock associated with *`lock`* for use in the subsequent calls.
`omp_destroy_nest_lock(`*`lock`*`)`	Causes the nested lock associated with *`lock`* to become undefined.
`omp_set_nest_lock(`*`lock`*`)`	Forces the executing thread to wait until the nested lock associated with *`lock`* is available. The thread is granted ownership of the nested lock when it becomes available.
`omp_unset_nest_lock(`*`lock`*`)`	Releases the executing thread from ownership of the nested lock associated with *`lock`* if the nesting count is zero. Behavior is undefined if the executing thread does not own the nested lock associated with *`lock`*.
`omp_test_nest_lock(`*`lock`*`)`	Attempts to set the nested lock associated with *`lock`*. If successful, returns the nesting count, otherwise returns zero.
Timing Routines	
`omp_get_wtime()`	Returns a double-precision value equal to the elapsed wallclock time (in seconds) relative to an arbitrary reference time. The reference time does not change during program execution.
`omp_get_wtick()`	Returns a double-precision value equal to the number of seconds between successive clock ticks.

Intel Extensions

The Intel® C++ Compiler implements the following groups of functions as extensions to the OpenMP* run-time library:

- Getting and setting stack size for parallel threads
- Memory allocation

The Intel extensions described in this section can be used for low-level debugging to verify that the library code and application are functioning as intended. It is recommended to use these functions with caution because using them requires the use of the `-Qopenmp_stubs` command-line option to execute the program sequentially. These functions are also generally not recognized by other vendor's OpenMP-compliant compilers, which may cause the link stage to fail for these other compilers.

Note

The functions below require the pre-processor directive `#include <omp.h>`.

Stack Size

In most cases, directives can be used in place of extensions. For example, the stack size of the parallel threads may be set using the `KMP_STACKSIZE` environment variable rather than the `kmp_set_stacksize_s()` function.

Note

A run-time call to an Intel extension takes precedence over the corresponding environment variable setting. See the definitions of stack size functions in the Stack Size table below.

Memory Allocation

The Intel® C++ Compiler implements a group of memory allocation functions as extensions to the OpenMP run-time library to enable threads to allocate memory from a heap local to each thread. These functions are `kmp_malloc()`, `kmp_calloc()`, and `kmp_realloc()`. The memory allocated by these functions must also be freed by the `kmp_free()` function. While it is legal for the memory to be allocated by one thread and `kmp_free()`'d by a different thread, this mode of operation has a slight performance penalty. See the definitions of these functions in the Memory Allocation table below.

Stack Size

Function	Description
`kmp_get_stacksize_s()`	Returns the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can be changed with `kmp_set_stacksize_s()` prior to the first parallel region or with the `KMP_STACKSIZE` environment variable.
`kmp_get_stacksize()`	This function is provided for backwards compatibility only. Use `kmp_get_stacksize_s()` for compatibility across different families of Intel processors.
`kmp_set_stacksize_s(`*`size`*`)`	Sets to *`size`* the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can also be set via the `KMP_STACKSIZE` environment variable. In order for `kmp_set_stacksize_s()` to have an effect, it must be called before the beginning of the first (dynamically executed) parallel region in the program.
`kmp_set_stacksize(`*`size`*`)`	This function is provided for backward compatibility only; use `kmp_set_stacksize_s()` for compatibility across different families of Intel processors.

Memory Allocation

Function	Description
`kmp_malloc(`*`size`*`)`	Allocate memory block of *`size`* bytes from thread-local heap.
`kmp_calloc(`*`nelem, elsize`*`)`	Allocate array of *`nelem`* elements of size *`elsize`* from thread-local heap.
`kmp_realloc(`*`ptr, size`*`)`	Reallocate memory block at address *`ptr`* and *`size`* bytes from thread-local heap.
`kmp_free(`*`ptr`*`)`	Free memory block at address *`ptr`* from thread-local heap. Memory must have been previously allocated with `kmp_malloc()`, `kmp_calloc()`, or `kmp_realloc()`.

Overview: Intel Workqueuing Model

The workqueuing model lets you parallelize control structures that are beyond the scope of those supported by the OpenMP* model, while attempting to fit into the framework defined by OpenMP. In particular, the workqueuing model is a flexible mechanism for specifying units of work that are not pre-computed at the start of the worksharing construct. For `single`, `for`, and `sections` constructs all work units that can be executed are known at the time the construct begins execution. The workqueuing pragmas taskq and task relax this restriction by specifying an environment (the taskq) and the units of work (the tasks) separately.

Workqueuing Constructs

taskq Pragma

The `taskq` pragma specifies the environment within which the enclosed units of work (tasks) are to be executed. From among all the threads that encounter a `taskq` pragma, one is chosen to execute it initially. Conceptually, the `taskq` pragma causes an empty queue to be created by the chosen thread, and then the code inside the `taskq` block is executed single-threaded. All the other threads wait for work to be enqueued on the conceptual queue. The `task` pragma specifies a unit of work, potentially executed by a different thread. When a `task` pragma is encountered lexically within a `taskq` block, the code inside the `task` block is conceptually enqueued on the queue associated with the `taskq`. The conceptual queue is disbanded when all work enqueued on it finishes, and when the end of the `taskq` block is reached.

Control Structures

Many control structures exhibit the pattern of separated work iteration and work creation, and are naturally parallelized with the workqueuing model. Some common cases are:

- `while` loops
- C++ iterators
- Recursive functions.

while Loops

If the computation in each iteration of a `while` loop is independent, the entire loop becomes the environment for the `taskq` pragma, and the statements in the body of the `while` loop become the units of work to be specified with the `task` pragma. The conditional in the `while` loop and any modifications to the control variables are placed outside of the `task` blocks and executed sequentially to enforce the data dependencies on the control variables.

C++ Iterators

C++ Standard Template Library (STL) iterators are very much like the `while` loops just described, whereby the operations on the data stored in the STL are very distinct from the act of iterating over all the data. If the operations are data-independent, they can be done in parallel as long as the iteration over the work is sequential. This type of `while` loop parallelism is a generalization of the standard OpenMP* worksharing for loops. In the worksharing for loops, the loop increment operation is the iterator and the body of the loop is the unit of work. However, because the `for` loop iteration variable frequently has a closed form solution, it can be computed in parallel and the sequential step avoided.

Recursive Functions

Recursive functions also can be used to specify parallel iteration spaces. The mechanism is similar to specifying parallelism using the `sections` pragma, but is much more flexible because it allows arbitrary code to sit between the `taskq` and the `task` pragmas, and because it allows recursive nesting of the function to build a conceptual tree of `taskq` queues. The recursive nesting of the `taskq` pragmas is a conceptual extension of OpenMP worksharing constructs to behave more like nested OpenMP parallel regions. Just like nested parallel regions, each nested workqueuing construct is a new instance and is encountered by exactly one thread. However, the major difference is that nested workqueuing constructs do not cause new threads or teams to be formed, but rather re-use the threads from the team. This permits very easy multi-algorithmic parallelism in dynamic environments, such that the number of threads need not be committed at each level of parallelism, but instead only at the top level. From that point on, if a large amount of work suddenly appears at an inner level, the idle threads from the outer level can assist in getting that work finished. For example, it is very common in server environments to dedicate a thread to handle each incoming request, with a large number of threads awaiting incoming requests. For a particular request, its size may not be obvious at the time the thread begins handling it. If the thread uses nested workqueuing constructs, and the scope of the request becomes large after the inner construct is started, the threads from the outer construct can easily migrate to the inner construct to help finish the request.

Since the workqueuing model is designed to preserve sequential semantics, synchronization is inherent in the semantics of the `taskq` block. There is an implicit team barrier at the completion of the `taskq` block for the threads that encountered the `taskq` construct to ensure that all of the tasks specified inside of the `taskq` block have finished execution. This `taskq` barrier enforces the sequential semantics of the original program. Just like the OpenMP worksharing constructs, it is assumed you are responsible for ensuring that either no dependences exist or that dependencies are appropriately synchronized between the task blocks, or between code in a task block and code in the `taskq` block outside of the task blocks.

The syntax, semantics, and allowed clauses are designed to resemble OpenMP* worksharing constructs. Most of the clauses allowed on OpenMP worksharing constructs have a reasonable meaning when applied to the workqueuing pragmas.

taskq Construct

```
#pragma intel omp taskq [clause[[,]clause]...]
     structured-block
```

where `clause` can be any of the following:

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

private

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

firstprivate

The `firstprivate` clause creates a private, copy-constructed version for each object in `variable-list` for the `taskq`. It also implies `captureprivate` on each enclosed task. The original object referenced by each variable must not be modified within the dynamic extent of the construct and has an indeterminate value upon exit from the construct.

lastprivate

The `lastprivate` clause creates a private, default-constructed version for each object in `variable-list` for the `taskq`. It also implies `captureprivate` on each enclosed task. The original object referenced by each variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and is copy-assigned the value of the object from the last enclosed task after that task completes execution.

reduction

The `reduction` clause performs a reduction operation with the given operator in enclosed task constructs for each object in `variable-list`. `operator` and `variable-list` are defined the same as in the OpenMP Specifications.

ordered

The `ordered` clause performs ordered constructs in enclosed `task` constructs in original sequential execution order. The `taskq` directive, to which the `ordered` is bound, must have an `ordered` clause present.

nowait

The `nowait` clause removes the implied barrier at the end of the `taskq`. Threads may exit the `taskq` construct before completing all the `task` constructs queued within it.

task Construct

```
#pragma intel omp task [clause[[,]clause]...]
     structured-block
```

where `clause` can be any of the following:

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

private

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

captureprivate

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

Combined parallel and taskq Construct

```
#pragma intel omp parallel taskq
     [clause[[,]clause]...]
     structured-block
```

where `clause` can be any of the following:

- `if(scalar-expression)`
- `num_threads(integer-expression)`
- `copyin(variable-list)`
- `default(shared | none)`
- `shared(variable-list)`
- `private(variable-list)`
- `firstprivate(variable-list)`
- `lastprivate(variable-list)`
- `reduction(operator : variable-list)`
- `ordered`

`Clause` descriptions are the same as for the OpenMP `parallel` construct or the `taskq` construct above as appropriate.

Example Function

The `test1` function below is a natural candidate to be parallelized using the workqueuing model. You can express the parallelism by annotating the loop with a parallel `taskq` pragma and the work in the loop body with a `task` pragma. The parallel `taskq` pragma specifies an environment for the `while` loop in which to enqueue the units of work specified by the enclosed `task` pragma. Thus, the loop's control structure and the enqueuing are executed single-threaded, while the other threads in the team participate in dequeuing the work from the `taskq` queue and executing it. The `captureprivate` clause ensures that a private copy of the link pointer p is captured at the time each task is being enqueued, hence preserving the sequential semantics.

```
void test1(LIST p)
{
  #pragma intel omp parallel taskq shared(p)
  {
    while (p != NULL)
    {
      #pragma intel omp task captureprivate(p)
      {
        do_work1(p);
      }
      p = p->next;
    }
  }
}
```

Examples of OpenMP* Usage

The following examples show how to use the OpenMP* feature.

A Simple Difference Operator

This example shows a simple parallel loop where the amount of work in each iteration is different. Dynamic scheduling is used to get good load balancing. The `for` has a `nowait` because there is an implicit `barrier` at the end of the parallel region.

```
void for_1 (float a[], float b[], int n)
{
  int i, j;
  #pragma omp parallel shared(a,b,n) private(i,j)
  {
      #pragma omp for schedule(dynamic,1) nowait
          for(i = 1; i < n; i++)
          {
             for(j = 0; j <= i; j++)
             b[j + n*i] = (a[j + n*i] + a[j + n*(i-1)])/2.0;
          }
   }
}
```

Two Difference Operators

The example below uses two parallel loops fused to reduce fork/join overhead. The first `for` has a `nowait` because all the data used in the second loop is different than all the data used in the first loop.

```
void for_2 (float a[], float b[], float c[], \
float d[], int n, int m)
{
  int i, j;
  #pragma omp parallel shared(a,b,c,d,n,m) private(i,j)
  {
    #pragma omp for schedule(dynamic,1) nowait
    for(i = 1; i < n; i++)
    {
      for(j = 0; j <= i; j++)
      b[j + n*i] = ( a[j + n*i] + a[j + n*(i-1)] )/2.0;
    }

    #pragma omp for schedule(dynamic,1) nowait
    for(i = 1; i < m; i++)
    {
      for(j = 0; j <= i; j++)
      d[j + m*i] = ( c[j + m*i] + c[j + m*(i-1)] )/2.0;
    }
  }
}
```

Overview: Auto-parallelization

The auto-parallelization feature of the Intel® C++ Compiler automatically translates serial portions of the input program into equivalent multithreaded code. The auto-parallelizer analyzes the dataflow of the program’s loops and generates multithreaded code for those loops which can be safely and efficiently executed in parallel. This enables the potential exploitation of the parallel architecture found in symmetric multiprocessor (SMP) systems.

Automatic parallelization relieves the user from:

- Having to deal with the details of finding loops that are good worksharing candidates
- Performing the dataflow analysis to verify correct parallel execution
- Partitioning the data for threaded code generation as is needed in programming with OpenMP directives.

The parallel run-time support provides the same run-time features found in OpenMP*, such as handling the details of loop iteration modification, thread scheduling, and synchronization.

While OpenMP directives enable serial applications to transform into parallel applications quickly, the programmer must explicitly identify specific portions of the application code that contain parallelism and add the appropriate compiler directives. Auto-parallelization triggered by the `-parallel` option automatically identifies those loop structures which contain parallelism. During compilation, the compiler automatically attempts to decompose the code sequences into separate threads for parallel processing. No other effort by the programmer is needed.

The following example illustrates how a loop’s iteration space can be divided so that it can be executed concurrently on two threads:

Original Serial Code

```
for (i=1; i<100; i++)
{
  a[i] = a[i] + b[i] * c[i];
}
```

Transformed Parallel Code

Thread 1

```
for (i=1; i<50; i++)
{
  a[i] = a[i] + b[i] * c[i];
}
```

Thread 2

```
for (i=50; i<100; i++)
{
  a[i] = a[i] + b[i] * c[i];
}
```

Programming with Auto-parallelization

The auto-parallelization feature implements some concepts of OpenMP*, such as worksharing construct (with the `parallel for` directive). This section provides specifics of auto-parallelization.

Guidelines for Effective Auto-parallelization Usage

A loop is parallelizable if:

- The loop is countable at compile time. This means that an expression representing how many times the loop will execute (also called "the loop trip count") can be generated just before entering the loop.
- There are no `FLOW` (`READ` after `WRITE`), `OUTPUT` (`WRITE` after `READ`) or `ANTI` (`WRITE` after `READ`) loop-carried data dependences. A loop-carried data dependence occurs when the same memory location is referenced in different iterations of the loop. At the compiler's discretion, a loop may be parallelized if any assumed inhibiting loop-carried dependencies can be resolved by run-time dependency testing.

The compiler may generate a run-time test for the profitability of executing in `parallel for` loop with loop parameters that are not compile-time constants.

Coding Guidelines

Enhance the power and effectiveness of the auto-parallelizer by following these coding guidelines:

- Expose the trip count of loops whenever possible. Specifically use constants where the trip count is known and save loop parameters in local variables.
- Avoid placing structures inside loop bodies that the compiler may assume to carry dependent data, for example, function calls, ambiguous indirect references, or global references.

Auto-parallelization Data Flow

For auto-parallelization processing, the compiler performs the following steps:

1. Data flow analysis
2. Loop classification
3. Dependence analysis
4. High-level parallelization
5. Data partitioning
6. Multi-threaded code generation

These steps include:

- Data flow analysis: compute the flow of data through the program
- Loop classification: determine loop candidates for parallelization based on correctness and efficiency as shown by threshold analysis
- Dependence analysis: compute the dependence analysis for references in each loop nest
- High-level parallelization:
    - analyze dependence graph to determine loops which can execute in parallel.
    - compute run-time dependency
- Data partitioning: examine data reference and partition based on the following types of access: `shared`, `private`, and `firstprivate.`
- Multi-threaded code generation:
    - modify loop parameters
    - generate entry/exit per threaded task
    - generate calls to parallel runtime routines for thread creation and synchronization

Auto-parallelization: Enabling, Options, and Environment Variables

To enable the auto-parallelizer, use the `-parallel` option. The `-parallel` option detects parallel loops capable of being executed safely in parallel and automatically generates multithreaded code for these loops. An example of the command using auto-parallelization follows:

IA-32 Systems: `prompt>`**`icc -c -parallel prog.c`**
Itanium®-based Systems: `prompt>`**`ecc -c -parallel prog.c`**

Auto-parallelization Options

The `-parallel` option enables the auto-parallelizer if the `-O2` (or `-O3`) optimization option is also on (the default is `-O2`). The `-parallel` option detects parallel loops capable of being executed safely in parallel and automatically generates multithreaded code for these loops.

Option	Description
`-parallel`	Enables the auto-parallelizer
`-parallel_threshold{1-100}`	Controls the work threshold needed for auto-parallelization, see later subsection.
`-par_report{1|2|3}`	Controls the diagnostic messages from the auto-parallelizer, see later subsection.

Auto-parallelization Environment Variables

Variable	Description	Default
`OMP_NUM_THREADS`	Controls the number of threads used.	Number of processors currently installed in the system while generating the executable
`OMP_SCHEDULE`	Specifies the type of runtime scheduling.	`static`

Auto-parallelization Threshold Control and Diagnostics

Threshold Control

The `-par_threshold{n}` option sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel. The value of n can be from 0 to 100. The default value is 75. This option is used for loops whose computation work volume cannot be determined at compile-time. The threshold is usually relevant when the loop trip count is unknown at compile-time.

The `-par_threshold{n}` option has the following versions and functionality:

- Default: `-par_threshold` is not specified in the command line, which is the same as when `-par_threshold0` is specified. The loops get auto-parallelized regardless of computation work volume, that is, parallelize always.
- `-par_threshold100` - loops get auto-parallelized only if profitable parallel execution is almost certain.
- The intermediate 1 to 99 values represent the percentage probability for profitable speed-up. For example, n=50 would mean: parallelize only if there is a 50% probability of the code speeding up if executed in parallel.
- The default value of n is n=75 (or `-par_threshold75`). When `-par_threshold` is used on the command line without a number, the default value passed is 75.

The compiler applies a heuristic that tries to balance the overhead of creating multiple threads versus the amount of work available to be shared amongst the threads.

Diagnostics

The `-par_report{0|1|2|3}` option controls the auto-parallelizer's diagnostic levels 0, 1, 2, or 3 as follows:

- `-par_report0` = no diagnostic information is displayed.
- `-par_report1` = indicates loops successfully auto-parallelized (default). Issues a "`LOOP AUTO-PARALLELIZED`" message for parallel loops.
- `-par_report2` = indicates successfully auto-parallelized loops as well as unsuccessful loops.
- `-par_report3` = same as 2 plus additional information about any proven or assumed dependencies inhibiting auto-parallelization (reasons for not parallelizing).

Example of Parallelization Diagnostics Report

The example below shows output generated by `-par_report3`:

IA-32 Systems: `prompt>`**`icc -c -parallel -par_report3 prog.c`**

Sample Ouput

```
program prog
procedure: prog
serial loop: line 5: not a parallel candidate due to
statement at line 6
serial loop: line 9
flow data dependence from line 10 to line 10, due to "a"
12 Lines Compiled
```

where the program `prog.c` is as follows:

Sample prog.c

```
/* Assumed side effects */

for (i=1; i<10000; i++)
{
  a[i] = foo(i);
}

/* Actual dependence */

for (i=1; i<10000; i++)
{
  a[i] = a[i-1] + i;
}
```

Troubleshooting Tips

- Use `-par_threshold0` to see if the compiler assumed there was not enough computational work
- Use `-par_report3` to view diagnostics
- Use `-ipo` to eliminate assumed side-effects done to function calls

Overview: Vectorization

The vectorizer is a component of the Intel® C++ Compiler that automatically uses SIMD instructions in the MMX(TM), SSE, and SSE2 instruction sets. The vectorizer detects operations in the program that can be done in parallel, and then converts the sequential program to process 2, 4, 8, or 16 elements in one operation, depending on the data type.

This section provides guidelines, option descriptions, and examples for the Intel C++ Compiler vectorization on IA-32 systems only. The following list summarizes this section's contents.

- A quick reference of vectorization functionality and features
- Descriptions of compiler switches to control vectorization
- Descriptions of the C++ language features to control vectorization
- Discussion and general guidelines on vectorization levels:
  - Automatic vectorization
  - Vectorization with user intervention
- Examples demonstrating typical vectorization issues and resolutions

Vectorizer Options

Option	Description
`-ax{M|K|W}`	Enables the vectorizer and generates specialized and generic IA-32 code. The generic code is usually slower than the specialized code.
`-x{M|K|W}`	Turns on the vectorizer and generates processor-specific specialized code.
`-vec_report`*n*	Controls the vectorizer's level of diagnostic messages: • *n* =0 no diagnostic information is displayed. • *n* =1 display diagnostics indicating loops successfully vectorized (default). • *n* =2 same as *n* =1, plus diagnostics indicating loops not successfully vectorized. • *n* =3 same as *n* =2, plus additional information about any proven or assumed dependences.

Usage

If you use `-c`, `-ipo` with `-vec_report{n}` option or `-c`, `-x{M|K|W}` or `-ax{M|K|W}` with `-vec_report{n}`, the compiler issues a warning and no report is generated.

To produce a report when using the afore mentioned options, you need to add the `-ipo_obj` option. The combination of `-c` and `-ipo_obj` produces a single file compilation, and hence does generate object code, and eventually a report is generated.

The following commands generate a vectorization report:

- `prompt>`**`icc -x{M|K|W} -vec_report3 file.c`**
- `prompt>`**`icc -x{M|K|W} -ipo -ipo_obj -vec_report3 file.c`**
- `prompt>`**`icc -c -x{M|K|W} -ipo -ipo_obj -vec_report3 file.c`**

The following commands do not generate a vectorization report:

- `prompt>`**`icc -c -x{M|K|W} -vec_report3 file.c`**
- `prompt>`**`icc -x{M|K|W} -ipo -vec_report3 file.c`**
- `prompt>`**`icc -c -x{M|K|W} -ipo -vec_report3 file.c`**

Loop Parallelization and Vectorization

Combining the `-parallel` and `-x{M|K|W}` options instructs the compiler to attempt both automatic loop parallelization and automatic loop vectorization in the same compilation. In most cases, the compiler will consider outermost loops for parallelization and innermost loops for vectorization. If deemed profitable, however, the compiler may even apply loop parallelization and vectorization to the same loop.

Note that in some cases successful loop parallelization (either automatically or by means of OpenMP* directives) may affect the messages reported by the compiler for loop vectorization; for example, under the `-vec_report2` option indicating loops not successfully vectorized.

Vectorization Key Programming Guidelines

The goal of vectorizing compilers is to exploit single-instruction multiple data (SIMD) processing automatically. Review these guidelines and restrictions, see code examples in further topics, and check them against your code to eliminate ambiguities that prevent the compiler from achieving optimal vectorization.

Guidelines for loop bodies:

- Use straight-line code (a single basic block)
- Use vector data only; that is, arrays and invariant expressions on the right hand side of assignments. Array references can appear on the left hand side of assignments.
- Use only assignment statements

Avoid the following in loop bodies:

- Function calls
- Unvectorizable operations
- Mixing vectorizable types in the same loop
- Data-dependent loop exit conditions

Preparing your code for vectorization

To make your code vectorizable, you will often need to make some changes to your loops. However, you should make only the changes needed to enable vectorization and no others. In particular, you should avoid these common changes:

- Do not unroll your loops, the compiler does this automatically.
- Do not decompose one loop with several statements in the body into several single-statement loops.

Restrictions

Hardware. The compiler is limited by restrictions imposed by the underlying hardware. In the case of Streaming SIMD Extensions, the vector memory operations are limited to `stride-1` accesses with a preference to 16-byte-aligned memory references. This means that if the compiler abstractly recognizes a loop as vectorizable, it still might not vectorize it for a distinct target architecture.

Style. The style in which you write source code can inhibit optimization. For example, a common problem with global pointers is that they often prevent the compiler from being able to prove two memory references at distinct locations. Consequently, this prevents certain reordering transformations.

Many stylistic issues that prevent automatic vectorization by compilers are found in loop structures. The ambiguity arises from the complexity of the keywords, operators, data references, and memory operations within the loop bodies.

However, by understanding these limitations and by knowing how to interpret diagnostic messages, you can modify your program to overcome the known limitations and enable effective vectorizations. The following topics summarize the capabilities and restrictions of the vectorizer with respect to loop structures.

Data Dependence

Data dependence relations represent the required ordering constraints on the operations in serial loops. Because vectorization rearranges the order in which operations are executed, any auto-vectorizer must have at its disposal some form of data dependence analysis. The "Data-dependent Loop" example shows some code that exhibits data dependence. The value of each element of an array is dependent on itself and its two neighbors.

Data-dependent Loop

```
float data[N];
int i;

for (i=1; i<N-1; i++)
{
   data[i]=data[i-1]*0.25+data[i]*0.5+data[i+1]*0.25;
}
```

The loop in the example above is not vectorizable because the write to the current element `data[i]` is dependent on the use of the preceding element `data[i-1]`, which has already been written to and changed in the previous iteration. To see this, look at the access patterns of the array for the first two iterations as shown in the following example:

Data Dependence Vectorization Patterns

```
for(i=0; i<100; i++)
a[i]=b[i];
```

has access pattern

```
read b[0]
write a[0]
read b[1]
write a[1]

i=1: READ data[0]
READ data[1]
READ data[2]
WRITE data[1]

i=2: READ data[1]
READ data[2]
READ data[3]
WRITE data[2]
```

In the normal sequential version of the loop shown, the value of `data[1]` read during the second iteration was written into the first iteration. For vectorization, the iterations must be done in parallel, without changing the semantics of the original loop.

Data Dependence Theory

Data dependence analysis involves finding the conditions under which two memory accesses may overlap. Given two references in a program, the conditions are defined by:

- whether the referenced variables may be aliases for the same (or overlapping) regions in memory,
- for array references, the relationship between the subscripts.

For array references, the Intel® C++ Compiler's data dependence analyzer is organized as a series of tests that progressively increase in power as well as time and space costs. First, a number of simple tests are performed in a dimension-by-dimension manner, since independence in any dimension will exclude any dependence relationship. Multi-dimensional arrays references that may cross their declared dimension boundaries can be converted to their linearized form before the tests are applied. Some of the simple tests used are the fast GCD test, proving independence if the greatest common divisor of the coefficients of loop indices cannot evenly divide the constant term, and the extended bounds test, which tests potential overlap for the extreme values of subscript expressions.

If all simple tests fail to prove independence, the compiler will eventually resort to a powerful hierarchical dependence solver that uses Fourier-Motzkin elimination to solve the data dependence problem in all dimensions.

Loop Constructs

Loops can be formed with the usual `for` and `while-do`, or `repeat-until` constructs or by using a `goto` and a label. However, the loops must have a single entry and a single exit to be vectorized.

Correct Usage

```
while(i<n)
{
   // If branch is inside body of loop

   a[i]=b[i]*c[i];
   if(a[i]<0.0)
   {
      a[i]=0.0;
   }
   i++;
}
```

Incorrect Usage

```
while(i<n)
{
   if (condition) break;
   // 2nd exit.
   ++i;
}
```

Loop Exit Conditions

Loop exit conditions determine the number of iterations that a loop executes. For example, fixed indexes for loops determine the iterations. The loop iterations must be countable; that is, the number of iterations must be expressed as one of the following:

- A constant
- A loop invariant term
- A linear function of outermost loop indices

Loops whose exit depends on computation are not countable. Examples below show countable and non-countable loop constructs.

Correct Usage for Countable Loop

```
// Exit condition specified by "N-1b+1"
count=N;

...

while(count!=1b)
{
   // 1b is not affected within loop
   a[i]=b[i]*x;
   b[i]=[i]+sqrt(d[i]);
   --count;
}
```

Correct Usage for Countable Loop

```
// Exit condition is "(n-m+2)/2"
i=0;
for(l=m; l<n; l+=2)
{
   a[i]=b[i]*x;
   b[i]=c[i]+sqrt(d[i]);
   ++i;
}
```

Incorrect Usage for Non-Countable Loop

```
i=0;

// Iterations dependent on a[i]
while(a[i]>0.0)
{
   a[i]=b[i]*c[i];
   ++i;
}
```

Types of Loops Vectorized

For integer loops, MMX(TM) technology and Streaming SIMD Extensions provide SIMD instructions for most arithmetic and logical operators on 32-bit, 16-bit, and 8-bit integer data types. Vectorization may proceed if the final precision of integer wrap-around arithmetic will be preserved. A 32-bit shift-right operator, for instance, is not vectorized if the final stored value is a 16-bit integer. Also, note that because the MMX(TM) instructions and Streaming SIMD Extensions instruction sets are not fully orthogonal (byte shifts, for instance, are not supported), not all integer operations can actually be vectorized.

For loops that operate on 32-bit single-precision and 64-bit double-precision floating-point numbers, the Streaming SIMD Extensions provide SIMD instructions for the arithmetic operators `+`, `-`, `*`, and `/`. Also, the Streaming SIMD Extensions provide SIMD instructions for the binary `MIN`, `MAX`, and unary `SQRT` operators. SIMD versions of several other mathematical operators (like the trigonometric functions `SIN`, `COS`, `TAN`) are supported in software in a vector mathematical run-time library that is provided with the Intel® C++ Compiler.

Stripmining and Cleanup

The compiler automatically stripmines your loop and generates a cleanup loop. This means you do not need to unroll your loops, and, in most cases, this will also enable more vectorization.

Before Vectorization

```
i=0;
while(i<n)
{
   // Original loop code
   a[i]=b[i]+c[i];
   ++i;
}
```

After Vectorization

```
// The vectorizer generates the following two loops
i=0;

while(i<(n-n%4))
{
   // Vector strip-mined loop
   // Subscript [i:i+3] denotes SIMD execution
   a[i:i+3]=b[i:i+3]+c[i:i+3];
   i=i+4;
}

while(i<n)
{
   // Scalar clean-up loop
   a[i]=b[i]+c[i];
}
```

Statements in the Loop Body

The vectorizable operations are different for floating-point and integer data.

Floating-point Array Operations

The statements within the loop body may contain float operations (typically on arrays). Supported arithmetic operations include addition, subtraction, multiplication, division, negation, square root, max, and min. Operation on double precision types is not permitted unless optimizing for a Pentium® 4 processor system, using the `-xW` or `-axW` compiler option.

Integer Array Operations

The statements within the loop body may contain `char`, `unsigned char`, `short`, `unsigned short`, `int`, and `unsigned int`. Calls to functions such as `sqrt` and `fabs` are also supported. Arithmetic operations are limited to addition, subtraction, bitwise `AND`, `OR`, and `XOR` operators, division (16-bit only), multiplication (16-bit only), min, and max. You can mix data types only if the conversion can be done without a loss of precision. Some example operators where you can mix data types are multiplication, shift, or unary operators.

Other Operations

No statements other than the preceding floating-point and integer operations are allowed. In particular, note that the special `__m64` and `__m128` datatypes are not vectorizable. The loop body cannot contain any function calls. Use of the Streaming SIMD Extensions intrinsics (`_mm_add_ps`) are not allowed.

Language Support and Directives

This topic addresses language features that better help to vectorize code. The `__declspec(align(n))` declaration enables you to overcome hardware alignment constraints. The `restrict` qualifier and the pragmas address the stylistic issues due to lexical scope, data dependence, and ambiguity resolution.

Language Support

Option	Description
`__declspec(align(n))`	Directs the compiler to align the variable to an n-byte boundary. Address of the variable is `address mod n`=0.
`__declspec(align(n,off))`	Directs the compiler to align the variable to an n-byte boundary with offset off within each n-byte boundary. Address of the variable is `address mod n` = off.
`restrict`	Permits the disambiguator flexibility in alias assumptions, which enables more vectorization.
`__assume_aligned(a,n)`	Instructs the compiler to assume that array `a` is aligned on an n-byte boundary; used in cases where the compiler has failed to obtain alignment information.
`#pragma ivdep`	Instructs the compiler to ignore assumed vector dependencies.
`#pragma vector {aligned | unaligned | always}`	Specifies how to vectorize the loop and indicates that efficiency heuristics should be ignored.
`#pragma novector`	Specifies that the loop should never be vectorized

Multi-version Code

Multi-version code is generated by the compiler in cases where data dependence analysis fails to prove independence for a loop due to the occurrence of pointers with unknown values. This functionality is referred to as dynamic dependence testing.

Pragma Scope

See Vectorization Support.

Dynamic Dependence Testing Example

Sample Code

```
float *p, *q;

for(i=L; I<=U; i++)
{
   p[i]=q[i];
}

...

pL=p*4*L;
pH=p+4*U;
qL=q+4*L;
qH=q+4*U;

if(pH<qL || pL>qH)
{

   // Loop without data dependence
   for(i=L; i<=U; i++)
   {
      p[i]=q[i];
   } else {

   for(i=L; i<=U; i++)
   {
      p[i]=q[i];
   }
}
```

Vectorization Examples

This section contains a few simple examples of some common issues in vector programming.

Argument Aliasing: A Vector Copy

The loop in the example below, a vector copy operation, vectorizes because the compiler can prove `dest[i]` and `src[i]` are distinct.

Vectorizable Copy Due To Unproven Distinction

```
void vec_copy(float *dest, float *src, int len)
{
   int i;
   for(i=0; i<len; i++;)
   {dest[i]=src[i];}
}
```

The restrict keyword in the example below indicates that the pointers refer to distinct objects. Therefore, the compiler allows vectorization without generation of multi-version code.

Using restrict to Prove Vectorizable Distinction

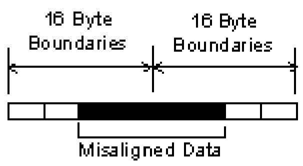
```
void vec_copy(float *restrict dest, float *restrict src, int len)
{
   int i;
   for(i=0; i<len; i++)
   {dest[i]=src[i];}
}
```

Data Alignment

A 16-byte or greater data structure or array should be aligned so that the beginning of each structure or array element is aligned in a way that its base address is a multiple of sixteen.

The "Misaligned Data Crossing 16-Byte Boundary" figure shows the effect of a data cache unit (DCU) split due to misaligned data. The code loads the misaligned data across a 16-byte boundary, which results in an additional memory access causing a six- to twelve-cycle stall. You can avoid the stalls if you know that the data is aligned and you specify to assume alignment.

Misaligned Data Crossing 16-Byte Boundary



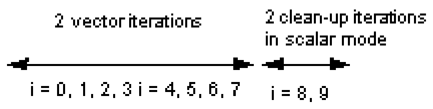
For example, if you know that elements `a[0]` and `b[0]` are aligned on a 16-byte boundary, then the following loop can be vectorized with the alignment option on (`#pragma vector aligned`):

Alignment of Pointers is Known

```
float *a, *b;
int i;

for(int i=0; i<10; i++)
{
   a[i]=b[i];
}
```

After vectorization, the loop is executed as shown here:

Vector and Scalar Clean-up Iterations


2 vector iterations
2 clean-up iterations in scalar mode
i = 0, 1, 2, 3 i = 4, 5, 6, 7
i = 8, 9

Both the vector iterations `a[0:3]` = `b[0:3]`; and `a[4:7]` = `b[4:7]`; can be implemented with aligned moves if both the elements `a[0]` and `b[0]` (or, likewise, `a[4]` and `b[4]`) are 16-byte aligned.

Caution

If you specify the vectorizer with incorrect alignment options, the compiler will generate unexpected behavior. Specifically, using aligned moves on unaligned data, will result in an illegal instruction exception.

Data Alignment Examples

The example below contains a loop that vectorizes but only with unaligned memory instructions. The compiler can align the local arrays, but because `lb` is not known at compile-time. The correct alignment cannot be determined.

Loop Unaligned Due to Unknown Variable Value at Compile Time

```
void f(int lb)
{
   float z2[N], a2[N], y2[N], x2;
   for(i=lb; i<N; i++)
   {a2[i]=a2[i]*x2+y2[i];}
}
```

If you know that `lb` is a multiple of 4, you can align the loop with `#pragma vector aligned` as shown in the example that follows:

Alignment Due to Assertion of Variable as Multiple of 4

```
void f(int lb)
{
   float z2[N], a2[N], y2[N], x2;
   assert(lb%4==0);

   #pragma vector aligned

   for(i=lb; i<N; i++)
   {a2[i]=a2[i]*x2+y2[i];}
}
```

Loop Interchange and Subscripts: Matrix Multiply

Matrix multiplication is commonly written as shown in the example below:

Typical Matrix Multiplication

```
for(i=0; i<N; i++)
{
    for(j=0; j<n; j++)
    {
        for(k=0; k<n; k++)
        {
            c[i][j]=c[i][j]+a[i][k]*b[k][j];
        }
    }
}
```

The use of `b[k][j]`, is not a `stride-1` reference and therefore will not normally be vectorizable. If the loops are interchanged, however, all the references will become `stride-1` as shown in the "Matrix Multiplication With Stride-1" example.

Caution

Interchanging is not always possible because of dependencies, which can lead to different results.

Matrix Multiplication With Stride-1

```
for(i = 0; i<N; i++)
{
    for(k=0; k<n; k++)
    {
        for(j=0; j<n; j++)
        {
            c[i][j]=c[i][j]+a[i][k]*b[k][j];
        }
    }
}
```

Optimization Support Features Overview

This section describes language extensions to the Intel® C++ Compiler that let you optimize your source code directly. Examples are included of optimizations supported by Intel extended directives and library routines that enhance and/or help analyze performance.

Compiler Directives

This section discusses the language extended directives used in:

- Software Pipelining
- Loop Count and Loop Distribution
- Loop Unrolling
- Prefetching
- Vectorization

Pipelining for Itanium®-based Applications

The `swp` and `noswp` directives indicate preference for a loop to get software-pipelined or not. The `swp` directive does not help data dependence, but overrides heuristics based on profile counts or lop-sided control flow. The syntax for this directive is:

```
#pragma swp
#pragma noswp
```

Example of swp Directive

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

The software pipelining optimization triggered by the `swp` directive applies instruction scheduling to certain innermost loops, allowing instructions within a loop to be split into different stages, allowing increased instruction level parallelism. This can reduce the impact of long-latency operations, resulting in faster loop execution. Loops chosen for software pipelining are always innermost loops that do not contain procedure calls that are not inlined. Because the optimizer no longer considers fully unrolled loops as innermost loops, fully unrolling loops can allow an additional loop to become the innermost loop. You can request and view the optimization report to see whether software pipelining was applied (see Optimizer Report Generation).

Loop Count and Loop Distribution

loop count (n) Directive

The `loop count (n)` directive indicates the loop count is likely to be n. The syntax for this directive is:

```
#pragma loop count (n)
```

where n is an integer constant. The value of `loop count` affects heuristics used in software pipelining, vectorization and loop-transformations.

Example of loop count (n) Directive

`#pragma loop count (10000)`
 	
 `for(i=0; i<m; i++)`
 `{`
 `    //swp likely to occur in this loop`
 `    a[i]=b[i]+1.2;`
 `}`

distribute point Directive

The `distribute point` directive indicates to the compiler a preference of performing loop distribution. The syntax for this directive is:

```
#pragma distribute point
```

Loop distribution may cause large loops be distributed into smaller ones. This may enable software pipelining for more loops. If the directive is placed inside a loop, the distribution is performed after the directive and any loop-carried dependency is ignored. If the directive is placed before a loop, the compiler will determine where to distribute and data dependency is observed. Only one distribute directive is supported when placed inside the loop.

Example of distribute point Directive

```
#pragma distribute point

for(i=1; i<m; i++)
{
   b[i]=a[i]+1;

   ...

   //Compiler will automatically
   //decide where to distribute.
   //Data dependency is observed.

   c[i]=a[i]+b[i];

   ...

   d[i]=c[i]+1;
}

for(i=1; i<m; i++)
{
   b[i]=a[i]+1;

   ...

   #pragma distribute point

   //Distribution will start here,
   //ignoring all loop-carried dependency.

   sub(a,n);
   c[i]=a[i]+b[i];

   ...

   d[i]=c[i]+1;
}
```

Loop Unrolling Support

unroll Directive

The `unroll` directive (`unroll(n)|nounroll`) tells the compiler how many times to unroll a counted loop. The syntax for this directive is:

```
#pragma unroll
#pragma unroll(n)
#pragma nounroll
```

where `n` is an integer constant from 0 through 255. The `unroll` directive must precede the `for` statement for each `for` loop it affects. If `n` is specified, the optimizer unrolls the loop `n` times. If `n` is omitted, or if it is outside the allowed range, the optimizer assigns the number of times to unroll the loop. The `unroll` directive overrides any setting of loop unrolling from the command line. The directive can be applied only for the innermost nested loop. If applied to the outer loops, it is ignored. The compiler generates correct code by comparing `n` and the loop count.

Example of unroll Directive

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

Prefetching Support

prefetch Directive

The `prefetch` and `noprefetch` directives assert that the data prefetches are generated or not generated for some memory references. This affects the heuristics used in the compiler. The syntax for this directive is:

```
#pragma noprefetch
#pragma prefetch
#pragma prefetch a,b
```

If the expression `a[j]` is used within a loop, by placing `prefetch a` in front of the loop, the compiler will insert prefetches for `a[j+d]` within the loop, where d is determined by the compiler. This directive is supported when option `-O3` is on.

Example of prefetch Directive

```
#pragma noprefetch b
#pragma prefetch a

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

Vectorization Support (IA-32)

The `vector` directives control the vectorization of the subsequent loop in the program, but the compiler does not apply them to nested loops. Each nested loop needs its own directive preceding it. You must place the vector directive before the loop control statement.

vector always Directive

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 vector always Directive

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

ivdep Directive

The `ivdep` directive instructs the compiler to ignore assumed vector dependences. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization. This directive overrides that decision. Use `ivdep` only when you know that the assumed loop dependences are safe to ignore. The loop in the example below will not vectorize with the `ivdep`, since the value of k is not known (vectorization would be illegal if k<0).

Example of ivdep Directive

`#pragma ivdep`
 	
`for(i=0; i<m; i++)`
`{`
`   a[i]=a[i+k]*c;`
`}`

vector aligned Directive

The `vector aligned` directive means the loop should be vectorized, if it is legal to do so, ignoring normal heuristic decisions about profitability. When the `aligned` or `unaligned` qualifier is used, the loop should be vectorized using `aligned` or `unaligned` operations. Specify either `aligned` or `unaligned`, but not both.

⚠ **Caution**

If you specify `aligned` as an argument, you must be absolutely sure that the loop will be vectorizable using this instruction. Otherwise, the compiler will generate incorrect code. The loop in the example below uses the `aligned` qualifier to request that the loop be vectorized with `aligned` instructions, as the arrays are declared in such a way that the compiler could not normally prove this would be safe to do so.

Example of vector aligned Directive

```
#void foo(float *a)
{
   #pragma vector aligned
   for(i=0; i<m; i++)
   {
      a[i]=a[i]*c;
   }
}
```

The compiler includes several alignment strategies in case the alignment of data structures is not known at compile time. A simple example is shown below, but several other strategies are supported as well. If, in the loop shown below, the alignment of a is unknown, the compiler will generate a prelude loop that iterates until the array reference that occurs the most hits an aligned address. This makes the alignment properties of a known, and the vector loop is optimized accordingly.

Example of Alignment Strategies

```
float *a;

//Alignment unknown
for(i=0; i<100; i++)
{
   a[i]=a[i]+1.0f;
}

//Dynamic loop peeling
p=a & 0x0f;
if(p!=0)
{
   p=(16-p)/4;
   for(i=0; i<p; i++)
   {
      a[i]=a[i]+1.0f;
   }
}

//Loop with a aligned.
//Will be vectorized accordingly.
for(i=p; i<100; i++)
{
   a[i]=a[i]+1.0f;
}
```

novector Directive

The `novector` directive specifies that the loop should never be vectorized, even if it is legal to do so. In this example, suppose you know the trip count `(ub - lb)` is too low to make vectorization worthwhile. You can use `novector` to tell the compiler not to vectorize, even if the loop is considered vectorizable.

Example of novector Directive

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

Timing Your Application

How fast your application executes is one indication of performance. When timing the speed of applications, consider the following circumstances:

- Run program timings when other users are not active. Your timing results can be affected by one or more CPU-intensive processes also running while doing your timings.
- Try to run the program under the same conditions each time to provide the most accurate results, especially when comparing execution times of a previous version of the same program. Use the same system (processor model, amount of memory, version of the operating system, and so on) if possible.
- If you do need to change systems, you should measure the time using the same version of the program on both systems, so you know each system's effect on your timings.
- For programs that run for less than a few seconds, run several timings to ensure that the results are not misleading. Certain overhead functions, like loading external programs, might influence short timings considerably.
- If your program displays a lot of text, consider redirecting the output from the program. Redirecting output from the program will change the times reported because of reduced screen I/O.

The following program illustrates a model for program timing:

Sample Timing

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
   clock_t start, finish;
   long loop;
   double  duration, loop_calc;
   start = clock();
   for(loop=0; loop <= 2000; loop++)
   {
      loop_calc = 123.456 * 789;

      //printf() inculded to facilitate example
      printf("\nThe value of loop is: %d", loop);
   }
   finish = clock();
   duration = (double)(finish - start)/CLOCKS_PER_SEC;
   printf("\n%2.3f seconds\n", duration);
}
```

Optimizer Report Generation

The Intel® C++ Compiler provides options to generate and manage optimization reports:

- `-opt_report` generates an optimization report and directs it to `stderr`. By default, the compiler does not generate optimization reports.
- `-opt_report_file`*`filename`* generates an optimization report and directs it to a file specified in *`filename`*.
- `-opt_report_level{`*`min`*`|`*`med`*`|`*`max`*`}` specifies the detail level of the optimization report. The *`min`* argument provides the minimal summary and *`max`* produces the full report. The default is `-opt_report_level`*`min`*.
- `-opt_report_routine`*`fileroutine_substring`* generates reports from all routines with names containing the *`substring`* as part of their name. If not specified, reports from all routines are generated. By default, the compiler generates reports for all routines.

Specifying Optimizations to Generate Reports

The compiler can generate reports for an optimizer you specify in the *`phase`* argument of the `-opt_report_phase`*`phase`* option. The option can be used multiple times on the same command line to generate reports for multiple optimizers. Currently, the following optimizer reports are supported.

Optimizer Logical Name	Optimizer Full Name
`ipo`	Interprocedural Optimizer
`hlo`	High Level Optimizer
`ilo`	Intermediate Language Scalar Optimizer
`ecg`	Code Generator
`omp`	Open MP
`all`	All phases

When one of the above logical names for optimizers is specified, all reports from that optimizer are generated.

For example, `-opt_report_phase`*`ipo`* `-opt_report_phase`*`ecg`* generates reports from the interprocedural optimizer and the code generator.

Each of the optimizers can potentially have specific optimizations within them. Each of these optimizations are prefixed with one of the optimizer logical names. For example:

Optimizer_optimization	Full Name
`ipo_inline`	Interprocedural Optimizer, inline expansion of functions
`ipo_constant_propagation`	Interprocedural Optimizer, constant propagation
`ipo_function_reorder`	Interprocedural Optimizer, function reorder
`ilo_constant_propagation`	Intermediate Language Scalar Optimizer, constant propagation
`ilo_copy_propagation`	Intermediate Language Scalar Optimizer, copy propagation
`ecg_software_pipelining`	Code Generator, software pipelining

All optimization reports that have a matching prefix with the specified optimizer are generated. For example, if `-opt_report_phase ilo_co` is specified, a report from both the constant propagation and the copy propagation are generated.

The Availability of Report Generation

The `-opt_report_help` option lists the logical names of optimizers available for report generation.

Overview: Libraries

The Intel® C++ Compiler uses the GNU* C Library and Dinkumware* C++ Library. These libraries are documented at the following Internet locations:

GNU C Library

http://www.gnu.org/manual/glibc-2.2.3/html_chapter/libc_toc.html

Dinkumware C++ Library

http://www.dinkumware.com/htm_cpl/lib_cpp.html

Default Libraries

The following libraries are supplied with the Intel® C++ Compiler:

Library	Description
`libguide.a` `libguide.so`	for OpenMP* implementation
`libsvml.a`	short vector math library
`libirc.a`	Intel support library for PGO and CPU dispatch
`libimf.a`	Intel math library
`libimf.so`	Intel math library
`libcprts.a` `libcprts.so`	Dinkumware C++ Library
`libunwind.a` `libunwind.so`	Unwinder library
`libcxa.a` `libcxa.so`	Intel run time support for C++ features.

If you want to link your program with alternate or additional libraries, specify them at the end of the command line. For example, to compile and link `prog.c` with `mylib.a`, use the following command:

- **IA-32 Systems:** `prompt>`**`icc -oprog prog.c mylib.a`**
- **Itanium®-based Systems:** `prompt>`**`ecc -oprog prog.c mylib.a`**

The `mylib.a` library appears prior to the `libimf.a` library in the command line for the `ld` linker.

Caution

The Linux* system libraries and the compiler libraries are not built with the `-align` option. Therefore, if you compile with the `-align` option and make a call to a compiler distributed or system library, and have `long long`, `double`, or `long double` types in your interface, you will get the wrong answer due to the difference in alignment. Any code built with `-align` cannot make calls to libraries that use these types in their interfaces unless they are built with `-align` (in which case they will not work without `-align`).

Math Libraries

The Intel math library, `libimf.a`, is included with the Intel C++ Compiler. This math library contains optimized versions of the math functions in the standard C run-time library. The functions in `libimf.a` are optimized for program execution speed on the Pentium® III and Pentium 4 processors. The Itanium® compiler also includes a `libimf.a` designed to optimize execution on Itanium-based systems.

Note

The `-lm` switch is used for linking, precede it with `-limf` so that `libimf.a` is linked in before the system `libm.a`.

Example: `prompt>`**`icc prog.c -limf`**

See Managing Libraries.

Intel® Shared Libraries

The Intel® C++ Compiler links libraries statically at link time and dynamically at run time, the latter as dynamically-shared objects (DSO).

By default, the libraries are linked as follows:

- C++, math, and `libcprts.a` libraries are linked at link time, that is, statically.
- `libcxa.so` is linked dynamically.
- GNU* and Linux* system libraries are linked dynamically.

Advantages of This Approach

This approach:

- Enables to maintain the same model for both IA-32 and Itanium® compilers.
- Provides a model consistent with the Linux model where system libraries are dynamic and application libraries are static.
- The users have the option of using dynamic versions of our libraries to reduce the size of their binaries if desired.
- The users are licensed to distribute Intel-provided libraries.

Shared Library Options

The main options used with shared libraries are `-i_dynamic` and `-shared`.

The `-i_dynamic` option can be used to specify that all Intel-provided libraries should be linked dynamically. The comparison of the following commands illustrates the effects of this option.

1. `prompt>`**`icc prog.c`**

This command produces the following results (default):

- C++, math, `libirc.a`, and `libcprts.a` libraries are linked statically (at link time).
- Dynamic version of `libcxa.so` is linked at run time.

The statically linked libraries increase the size of the application binary, but do not need to be installed on the systems where the application runs.

2. `prompt>`**`icc -i_dynamic prog.c`**

This command links all of the above libraries dynamically. This has the advantage of reducing the size of the application binary, but it requires all the dynamic versions installed on the systems where the application runs.

The `-shared` option instructs the compiler to build a Dynamic Shared Object (DSO) instead of an executable. For more details, refer to the `ld` man page documentation.

Managing Libraries

The `LD_LIBRARY_PATH` environment variable contains a colon-separated list of directories in which the linker will search for library (`.a`) files. If you want the linker to search additional libraries, you can add their names to `LD_LIBRARY_PATH`, to the command line, to a response file, or to the configuration file. In each case, the names of these libraries are passed to the linker before the names of the Intel libraries that the driver always specifies.

Modifying LD_LIBRARY_PATH

If you want to add a directory, `/libs` for example, to the `LD_LIBRARY_PATH`, you can do either of the following:

- Command line: `prompt>`**`export LD_LIBRARY_PATH=/libs:$LD_LIBRARY_PATH`**
- Startup file **`export LD_LIBRARY_PATH=/libs:$LD_LIBRARY_PATH`**

To compile `file.c` and link it with the library `mylib.a`, enter the following command:

- **IA-32 Systems:** `prompt>`**`icc file.c mylib.a`**
- **Itanium®-based Systems:** `prompt>`**`ecc file.c mylib.a`**

The compiler passes file names to the linker in the following order:

1. The object file
2. Any objects or libraries specified on the command line, in a response file, or in a configuration file
3. The `libimf`.a library

Overview: Intel Math Library

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

Math Libraries for IA-32 and Itanium®-based Systems

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

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

See also Math Libraries.

Using the Intel Math Library

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

Example Using Real Functions

```
// real_math.c

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

int main() {

float  fp32bits;
double fp64bits;
long double fp80bits;
long double pi_by_four = 3.141592653589793238/4.0;

// pi/4 radians is about 45 degrees.

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

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

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

return 0;
}
```

Since the example program above includes the `long double` data type, be sure to include the `-long_double` compiler option:

IA-32 Systems: `icc -long_double real_math.c`

Itanium®-based Systems: `ecc -long_double real_math.c`

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

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

Example Using Complex Functions

```
// complex_math.c

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

int main()
{

float  _Complex c32in,c32out;
double _Complex c64in,c64out;
double pi_by_four= 3.141592653589793238/4.0;

c64in = 1.0 + __I__* pi_by_four;

// Create the double precision complex number 1 + pi/4 I
// where __I__ is the imaginary unit.

c32in = (float _Complex) c64in;

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

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

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

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

return 0;
}
```

IA-32 Systems: `icc complex_math.c`

Itanium®-based Systems: `ecc complex_math.c`

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

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

Note

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

Other Considerations

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

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

Depending on the data types used, some important compiler options include:

- `-long_double:` Use this option when compiling programs that require support for the `long double` data type (80-bit floating-point). Without this option, compilation will be successful, but `long double` data types will be mapped to `double` data types.
- `-c99:` Use this option when compiling programs that require support for `_Complex` data types.

Trigonometric Functions

The Intel Math library supports the following trigonometric functions:

ACOS

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

Calling interface:

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

ACOSD

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

Calling interface:

```
long double acosdl(long double x);
double acosd(double x);
float acosdf(float x);
```

ASIN

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

Calling interface:

```
long double asinl(long double x);
double asin(double x);
float asinf(float x);
```

ASIND

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

Calling interface:

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

ATAN

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

Calling interface:

```
long double atanl(long double x);
double atan(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 [-p, +pi] radians.

Calling interface:

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

ATAND

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

Calling interface:

```
long double atandl(long double x);
double atand(double x);
float atandf(float x);
```

ATAND2

Description: The `atand2` function returns the principal value of the inverse tangent of `y/x` in the range [-180, +180] degrees.

Calling interface:

```
long double atand2l(long double x, long double y); /* IA-32 only */
double atand2(double x, double y);
float atand2f(float x, float y);
```

COS

Description: The `cos` function returns the cosine of `x` measured in radians.

Calling interface:

```
long double cosl(long double x);
double cos(double x);
float cosf(float x);
```

COSD

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

Calling interface:

```
long double cosdl(long double x);
double cosd(double x);
float cosdf(float x);
```

COT

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

Calling interface:

```
long double cotl(long double x);
double cot(double x);
float cotf(float x);
```

COTD

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

Calling interface:

```
long double cotdl(long double x);
double cotd(double x);
float cotdf(float x);
```

SIN

Description: The `sin` function returns the sine of `x` measured in radians.

Calling interface:

```
long double sinl(long double x);
double sin(double x);
float sinf(float x);
```

SINCOS

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

Calling interface:

```
void sincosl(long double x, long double *sinval, long double *cosval);
void sincos(double x, double *sinval, 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 sincosdl(long double x, long double *sinval, long double
*cosval);
void sincosd(double x, double *sinval, 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:

```
long double sindl(long double x);
double sind(double x);
float sindf(float x);
```

TAN

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

Calling interface:

```
long double tanl(long double x);
double tan(double x);
float tanf(float x);
```

TAND

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

Calling interface:

```
long double tandl(long double x);
double tand(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`.

Calling interface:

```
long double acoshl(long double x);
double acosh(double x);
float acoshf(float x);
```

ASINH

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

Calling interface:

```
long double asinhl(long double x);
double asinh(double x);
float asinhf(float x);
```

ATANH

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

Calling interface:

```
long double atanhl(long double x);
double atanh(double x);
float atanhf(float x);
```

COSH

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

Calling interface:

```
long double coshl(long double x);
double cosh(double x);
float coshf(float x);
```

SINH

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

Calling interface:

```
long double sinhl(long double x);
double sinh(double x);
float sinhf(float x);
```

SINHCOSH

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

Calling interface:

```
void sinhcoshl(long double x, long double *sinval, long double
*cosval);

void sinhcosh(double x, float *sinval, float *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:

```
long double tanhl(long double x);
double tanh(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:

```
long double cbrtl(long double x);
double cbrt(double x);
float cbrtf(float x);
```

EXP

Description: The `exp` function returns e raised to the `x` power, e^x.

Calling interface:

```
long double expl(long double x);
double exp(double x);
float expf(float x);
```

EXP10

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

Calling interface:

```
long double exp10l(long double x);
double exp10(double x);
float exp10f(float x);
```

EXP2

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

Calling interface:

```
long double exp2l(long double x);
double exp2(double x);
float exp2f(float x);
```

EXPM1

Description: The `expm1` function returns e raised to the x power minus 1, $e^{x}-1$.

Calling interface:

```
long double expm1l(long double x);
double expm1(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:

```
long double frexp(long double x, int *exp);
double frexp(double x, int *exp);
float frexpf(float x, int *exp);
```

HYPOT

Description: The `hypot` function returns the value of the square root of the sum of the squares.

Calling interface:

```
long double hypotl(long double x, long double y);
double hypot(double x, 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.

Calling interface:

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

LDEXP

Description: The `ldexp` function returns the value of `x` times 2 raised to the power `exp`, $x*2^{exp}$.

Calling interface:

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

LOG

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

Calling interface:

```
long double logl(long double x);
double log(double x);
float logf(float x);
```

LOG10

Description: The `log10` function returns the base-10 log of `x`, $\log_{10}(x)$.

Calling interface:

```
long double log10l(long double x);
double log10(double x);
float log10f(float x);
```

LOG1P

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

Calling interface:

```
long double log1pl(long double x);
double log1p(double x);
float log1pf(float x);
```

LOG2

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

Calling interface:

```
long double log2l(long double x);
double log2(double x);
float log2f(float x;
```

LOGB

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

Calling interface:

```
long double logbl(long double x);
double logb(double x);
float logbf(float x);
```

POW

Description: The `pow` function returns `x` raised to the power of `y`, x^y.

Calling interface:

```
long double powl(double x, double y); /* Itanium®-based systems only
*/
double pow(double x, double y);
float powf(float x, float y);
```

SCALB

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

Calling interface:

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

SCALBN

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

Calling interface:

```
long double scalbnl (long double x, int y);
double scalbn(double x, int y);
float scalbnf(float x, int y);
```

SCALBLN

Description: The `scalbln` function returns $x*2^n$.

Calling interface:

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

SQRT

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

Calling interface:

```
long double sqrtl(long double x);
double sqrt(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.

Calling interface:

```
double annuity(double x, double y);
float annuityf(float x, double y);

/* All annuity functions: IA-32 only */
```

COMPOUND

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

Calling interface:

```
double compound(double x, double y);
float compoundf(float x, double y);

/* All compound functions: IA-32 only */
```

ERF

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

Calling interface:

```
long double erfl(long double x);
double erf(double x);
float erff(float x);
```

ERFC

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

Calling interface:

```
long double erfcl(long double x);
double erfc(double x);
float erfcf(float x);
```

GAMMA

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

Calling interface:

```
double gamma(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 external integer `signgam`.

Calling interface:

```
double gamma_r(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);
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);
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);
float jnf(int n, float x);
```

LGAMMA

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

Calling interface:

```
long double lgammal(long double x); /* Itanium®-based systems only */
double lgamma(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 external integer `signgam`.

Calling interface:

```
double lgamma_r(double x, int *signgam);
float lgammaf_r(float x, int *signgam);
```

TGAMMA

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

Calling interface:

```
long double tgammal(long double x); /* Itanium-based systems only */
double tgamma(double x);
float tgammaf(float x);
```

Y0

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

Calling interface:

```
double y0(double x);
float y0f(float x);
```

Y1

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

Calling interface:

```
double y1(double x);
float y1f(float x);
```

YN

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

Calling interface:

```
double yn(int n, 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.

Calling interface:

```
long double ceill(long double x);
double ceil(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.

Calling interface:

```
long double floorl(long double x);
double floor(double x);
float floorf(float x);
```

LRINT

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

Calling interface:

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

LLRINT

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

Calling interface:

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

LROUND

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

Calling interface:

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

LLROUND

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

Calling interface:

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

MODF

Description: The `modf` function returns the value of the signed fractional part of `x` and stores the integral part in floating-point format in `*iptr`.

Calling interface:

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

NEARBYINT

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

Calling interface:

```
long double nearbyintl(long double x);
double nearbyint(double x);
float nearbyintf(float x);
```

RINT

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

Calling interface:

```
long double rintl(long double x);
double rint(double x);
float rintf(float x);
```

ROUND

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

Calling interface:

```
long double roundl(long double x);
double round(double x);
float roundf(float x);
```

TRUNC

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

Calling interface:

```
long double truncl(long double x);
double trunc(double x);
float truncf(float x);
```

Remainder Functions

The Intel Math library supports the following remainder functions:

FMOD

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

Calling interface:

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

REMAINDER

Description: The `remainder` function returns the value of `x REM y`.

Calling interface:

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

REMQUO

Description: The `remquo` function returns the value of `x REM y`.

Calling interface:

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

/* All remquo functions: Itanium®-based systems only */
```

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:

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

FABS

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

Calling interface:

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

FDIM

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

Calling interface:

```
long double fdiml(long double x, long double y);
double fdim(double x, 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 finitel(long double x);
int finite(double x);
int finitef(float x);

/* All finite functions: Itanium®-based systems only */
```

FMA

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

Calling interface:

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

/* All the fma functions: Itanium-based systems only */
```

FMAX

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

Calling interface:

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

FMIN

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

Calling interface:

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

ISNAN

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

Calling interface:

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

NEXTAFTER

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

Calling interface:

```
long double nextafterl(long double x, long double y);
double nextafter(double x, 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.

Calling interface:

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

/* All nexttoward functions: Itanium-based systems only */
```

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);
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);
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);
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);
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);
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);
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);
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);
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);
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);
float _Complex ccoshf(float _Complex z);
```

CEXP

Description: The `cexp` function computes the complex base-e exponential of `z`.

Calling interface:

```
double _Complex cexp(double _Complex z);
float _Complex cexpf(float _Complex z);
```

CIMAG

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

Calling interface:

```
double cimag(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);
float _Complex cis(float z);
```

CLOG

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

Calling interface:

```
double _Complex clog(double _Complex z);
float _Complex clogf(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);
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);
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);
float _Complex cprojf(float _Complex z);
```

CREAL

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

Calling interface:

```
double creal(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);
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);
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);
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);
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);
float _Complex ctanhf(float _Complex z);
```

Overview: Diagnostics and Messages

This section describes the various messages that the compiler produces. These messages include the sign-on message and diagnostic messages for remarks, warnings, or errors. The compiler always displays any diagnostic message, along with the erroneous source line, on the standard output.

This section also describes how to control the severity of diagnostic messages.

Diagnostic Messages

Option	Description
`-w0,-w`	Displays error messages only. Both `-w0` and `-w` display exactly the same messages.
`-w1,-w2`	Displays warnings and error messages. Both `-w1` and `-w2` display exactly the same messages.The compiler uses this level as the default.

Language Diagnostics

These messages describe diagnostics that are reported during the processing of the source file. These diagnostics have the following format:

```
filename (linenum): type [#nn]: message
```

`filename`	Indicates the name of the source file currently being processed.
`linenum`	Indicates the source line where the compiler detects the condition.
`type`	Indicates the severity of the diagnostic message: warning, remark, error, or catastrophic error.
`[#nn]`	The number assigned to the error (or warning) message. Hard errors or catastrophes are not assigned a number.
`message`	Describes the diagnostic.

The following is an example of a warning message:

```
tantst.cpp(3): warning #328: Local variable "increment" never used.
```

The compiler can also display internal error messages on the standard error. If your compilation produces any internal errors, contact your Intel representative. Internal error messages are in the following form:

`FATAL COMPILER ERROR:` *`message`*

Suppressing Warning Messages with lint Comments

The UNIX `lint` program attempts to detect features of a C or C++ program that are likely to be bugs, non-portable, or wasteful. The compiler recognizes three `lint`-specific comments:

1. `/*ARGSUSED*/`
2. `/*NOTREACHED*/`
3. `/*VARARGS*/`

Like the `lint` program, the compiler suppresses warnings about certain conditions when you place these comments at specific points in the source.

Suppressing Warning Messages or Enabling Remarks

Use the `-w` or `-W`*n* option to suppress warning messages or to enable remarks during the preprocessing and compilation phases. You can enter the option with one of the following arguments:

Option	**Description**
`-w0,-w`	Displays error messages only. Both `-w0` and `-w` display exactly the same messages.
`-w1,-w2`	Displays warnings and error messages. Both `-w1` and `-w2` display exactly the same messages.The compiler uses this level as the default.

For some compilations, you might not want warnings for known and benign characteristics, such as the K&R C constructs in your code. For example, the following command compiles `newprog.c` and displays compiler errors, but not warnings:

- **IA-32 System:** `prompt>`**`icc -W0 newprog.c`**
- **Itanium®-based System:** `prompt>`**`ecc -W0 newprog.c`**

Limiting the Number of Errors Reported

Use the `-wn`*`n`* option to limit the number of error messages displayed before the compiler aborts. By default, if more than 100 errors are displayed, compilation aborts.

Option	Description
`-wn`*`n`*	Limit the number of error diagnostics that will be displayed prior to aborting compilation to *n* . Remarks and warnings do not count towards this limit.

For example, the following command line specifies that if more than 50 error messages are displayed during the compilation of `a.c`, compilation aborts.

- **IA-32 Systems:** `prompt>`**`icc -wn50 -c a.c`**
- **Itanium®-based Systems:** `prompt>`**`ecc -wn50 -c a.c`**

Remark Messages

These messages report common, but sometimes unconventional, use of C or C++. The compiler does not print or display remarks unless you specify level 4 for the `-W` option, as described in Suppressing Warning Messages or Enabling Remarks. Remarks do not stop translation or linking. Remarks do not interfere with any output files. The following are some representative remark messages:

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

gcc Compatibility

C language object files created with the Intel® C++ Compiler are binary compatible with the GNU* gcc compiler and glibc, the GNU C language library. C language object files can be linked with either the Intel compiler or the gcc compiler. However, to correctly pass the Intel libraries to the linker, use the Intel compiler. See Linking and Default Libraries for more information.

GNU C includes several, non-standard features not found in ISO standard C. Many of these extensions to the C language are supported in this version of the Intel C++ Compiler. See the GNU Web site at http://www.gnu.org for more information.

gcc Extension to C Language	Intel Support	GNU Description and Examples
Statements and Declarations in Expressions	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Statement-Exprs.html#Statement%
Locally Declared Labels	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Local-Labels.html#Local%20Labels
Labels as Values	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Labels-as-Values.html#Labels%20a 20Values
Nested Functions	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Nested-Functions.html#Nested%20
Constructing Function Calls	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Constructing-Calls.html#Constructi
Naming an Expression's Type	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Naming-Types.html#Naming%20Ty
Referring to a Type with typeof	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Typeof.html#Typeof
Generalized Lvalues	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Lvalues.html#Lvalues
Conditionals with Omitted Operands	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Conditionals.html#Conditionals
Double-Word Integers	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Long-Long.html#Long%20Long
Complex Numbers	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Complex.html#Complex

Hex Floats	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Hex-Floats.html#Hex%20Floats
Arrays of Length Zero	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Zero-Length.html#Zero%20Length
Arrays of Variable Length	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Variable-Length.html#Variable%20
Macros with a Variable Number of Arguments.	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Variadic-Macros.html#Variadic%20
Slightly Looser Rules for Escaped Newlines	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Escaped-Newlines.html#Escaped% 20Newlines
String Literals with Embedded Newlines	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Multi-line-Strings.html#Multi-line%2
Non-Lvalue Arrays May Have Subscripts	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Subscripting.html#Subscripting
Arithmetic on void-Pointers	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Pointer-Arith.html#Pointer%20Arith
Arithmetic on Function-Pointers	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Pointer-Arith.html#Pointer%20Arith
Non-Constant Initializers	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Initializers.html#Initializers
Compound Literals	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Compound-Literals.html#Compoun 20Literals
Designated Initializers	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Designated-Inits.html#Designated%
Cast to a Union Type	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Cast-to-Union.html#Cast%20to%20
Case Ranges	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Case-Ranges.html#Case%20Rang
Mixed Declarations and Code	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Mixed-Declarations.html#Mixed% 20Declarations
Declaring Attributes of Functions	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Function-Attributes.html#Function% 20Attributes
Attribute Syntax	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Attribute-Syntax.html#Attribute%20

Prototypes and Old-Style Function Definitions	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Function-Prototypes.html#Function 20Prototypes
C++ Style Comments	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/C---Comments.html#C++%20Comr
Dollar Signs in Identifier Names	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Dollar-Signs.html#Dollar%20Signs
The Character ESC in Constants	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Character-Escapes.html#Character 20Escapes
Specifying Attributes of Variables	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Variable-Attributes.html#Variable%
Specifying Attributes of Types	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Type-Attributes.html#Type%20Attri
Inquiring on Alignment of Types or Variables	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Alignment.html#Alignment
An Inline Function is As Fast As a Macro	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Inline.html#Inline
Assembler Instructions with C Expression Operands	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Extended-Asm.html#Extended%20
Controlling Names Used in Assembler Code	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Asm-Labels.html#Asm%20Labels
Variables in Specified Registers	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Explicit-Reg-Vars.html#Explicit%20 20Vars
Alternate Keywords	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Alternate-Keywords.html#Alternate 20Keywords
Incomplete enum Types	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Incomplete-Enums.html#Incomplete 20Enums
Function Names as Strings	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Function-Names.html#Function%2

Getting the Return or Frame Address of a Function	Yes	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Return-Address.html#Return%20A
Using Vector Instructions Through Built-in Functions	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Vector-Extensions.html#Vector%2
Other built-in functions provided by GCC	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Other-Builtins.html#Other%20Builti
Built-in Functions Specific to Particular Target Machines	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Target-Builtins.html#Target%20Bui
Pragmas Accepted by GCC	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Pragmas.html#Pragmas
Unnamed struct/union fields within structs/unions	No	http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Unnamed-Fields.html#Unnamed%

Compiler Limits

The table below 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

Key Files Summary for IA-32

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

/bin Files	Description
`iccvars.sh`	Batch file to set environment variables
`icc.cfg`	Configuration file for use from command line
`icc`	Intel® C++ Compiler
`profmerge`	Utility used for Profile Guided Optimizations
`proforder`	Utility used for Profile Guided Optimizations
`xild`	Tool used for Interprocedural Optimizations

/lib Files	Description
`libcprts.a`	C++ standard language library
`libcxa.so`	C++ language library indicating I/O data location
`libguide.a`	OpenMP* library
`libguide.so`	Shared OpenMP library
`libimf.a`	Special purpose math library functions, including some transcendentals, built only for Linux*.
`libintrins.a`	Intrinsic functions library
`libirc.a`	Intel-specific library (optimizations)
`libunwind.a`	Unwinder library
`libsvml.a`	Short-vector math library (used by vectorizer)

Key Files Summary for Itanium®-based Systems

The following tables list and briefly describe files that are installed for Itanium®-based systems.

/bin Files	Description
`eccvars.sh`	Batch file to set environment variables
`ecc.cfg`	Configuration file for use from command line
`ecc`	Intel® C++ Compiler
`ias`	Assembler
`profmerge`	Utility used for Profile Guided Optimizations
`proforder`	Utility used for Profile Guided Optimizations
`xild`	Tool used for Interprocedural Optimizations

/lib Files	Description
`libcprts.a`	C++ standard language library
`libcxa.so`	C++ language library indicating I/O data location
`libirc.a`	Intel-specific library (optimizations)
`libm.a`	Math library
`libguide.a`	OpenMP library
`libguide.so`	Shared OpenMP library
`libmofl.a`	Multiple Object Format Library, used by the Intel assembler
`libmofl.so`	Shared Multiple Object Format Library, used by the Intel assembler
`libunwinder.a`	Unwinder library
`libintrins.a`	Intrinsic functions library

Types of Intrinsics

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

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

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

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

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

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

Itanium®-based Application Developer's Architecture Guide, Intel Corporation

Intrinsics Availability on Intel Processors

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

Pentium Pro Processor	N/A	N/A	N/A	N/A
Pentium Processor	N/A	N/A	N/A	N/A

Benefits of Using Intrinsics

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

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

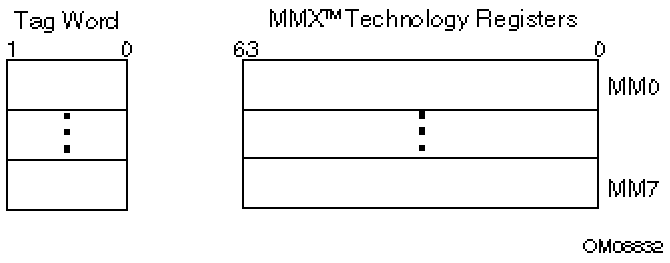
- New Registers--Enable packed data of up to 128 bits in length for optimal SIMD processing.
- New Data Types--Enable packing of up to 16 elements of data in one register.

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

New Registers

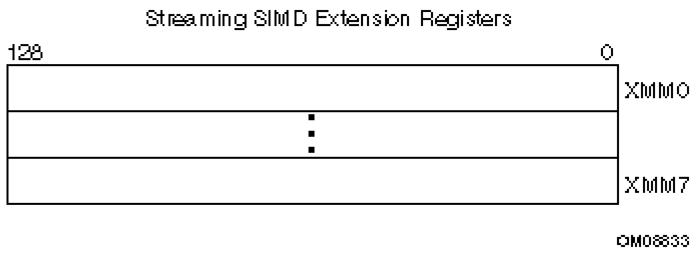
A key feature provided by the architecture of the processors are new register sets. The MMX instructions use eight 64-bit registers (`mm0` to `mm7`) which are aliased on the floating-point stack registers.

MMX(TM) Technology Registers



Streaming SIMD Extensions Registers

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



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

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

Note

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

New Data Types

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

New Data Types Available

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

__m64 Data Type

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

__m128 Data Types

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

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

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

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

New Data Types Usage Guidelines

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

- Use new data types only on either side of an assignment, as a return value, or as a parameter. You cannot use it with other arithmetic expressions ("+", "-", and so on).
- Use new data types as objects in aggregates, such as unions to access the byte elements and structures.
- Use new data types only with the respective intrinsics described in this documentation. The new data types are supported on both sides of an assignment statement: as parameters to a function call, and as a return value from a function call.

Naming and Usage Syntax

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

```
_mm_<intrin_op>_<suffix>
```

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

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

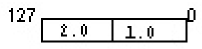
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
double a[2] = {1.0, 2.0};

__m128d t = _mm_load_pd(a);
```

The result is the same as either of the following:

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

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

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



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

Intrinsic Syntax

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

```
data_type intrinsic_name (parameters)
```

Where,

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

Intrinsics for All IA

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

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

Integer Arithmetic Related

Note

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

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

Floating-point Related

Intrinsic	Description
`double fabs(double)`	Returns the absolute value of a floating-point value.
`double log(double)`	Returns the natural logarithm ln(x), x>0, with double precision.
`float logf(float)`	Returns the natural logarithm ln(x), x>0, with single precision.
`double log10(double)`	Returns the base 10 logarithm log10(x), x>0, with double precision.
`float log10f(float)`	Returns the base 10 logarithm log10(x), x>0, with single precision.
`double exp(double)`	Returns the exponential function with double precision.
`float expf(float)`	Returns the exponential function with single precision.
`double pow(double, double)`	Returns the value of x to the power y with double precision.
`float powf(float, float)`	Returns the value of x to the power y with single precision.
`double sin(double)`	Returns the sine of x with double precision.
`float sinf(float)`	Returns the sine of x with single precision.
`double cos(double)`	Returns the cosine of x with double precision.
`float cosf(float)`	Returns the cosine of x with single precision.
`double tan(double)`	Returns the tangent of x with double precision.
`float tanf(float)`	Returns the tangent of x with single precision.
`double acos(double)`	Returns the arccosine of x with double precision
`float acosf(float)`	Returns the arccosine of x with single precision
`double acosh(double)`	Compute the inverse hyperbolic cosine of the argument with double precision.
`float acoshf(float)`	Compute the inverse hyperbolic cosine of the argument with single precision.
`double asin(double)`	Compute arc sine of the argument with double precision.
`float asinf(float)`	Compute arc sine of the argument with single precision.
`double asinh(double)`	Compute inverse hyperbolic sine of the argument with double precision.
`float asinhf(float)`	Compute inverse hyperbolic sine of the argument with single precision.
`double atan(double)`	Compute arc tangent of the argument with double precision.

`float atanf(float)`	Compute arc tangent of the argument with single precision.
`double atanh(double)`	Compute inverse hyperbolic tangent of the argument with double precision.
`float atanhf(float)`	Compute inverse hyperbolic tangent of the argument with single precision.
`float cabs(double)**`	Computes absolute value of complex number.
`double ceil(double)`	Computes smallest integral value of double precision argument not less than the argument.
`float ceilf(float)`	Computes smallest integral value of single precision argument not less than the argument.
`double cosh(double)`	Computes the hyperbolic cosine of double precison argument.
`float coshf(float)`	Computes the hyperbolic cosine of single precison argument.
`float fabsf(float)`	Computes absolute value of single precision argument.
`double floor(double)`	Computes the largest integral value of the double precision argument not greater than the argument.
`float floorf(float)`	Computes the largest integral value of the single precision argument not greater than the argument.
`double fmod(double)`	Computes the floating-point remainder of the division of the first argument by the second argument with double precison.
`float fmodf(float)`	Computes the floating-point remainder of the division of the first argument by the second argument with single precison.
`double hypot(double, double)`	Computes the length of the hypotenuse of a right angled triangle with double precision.
`float hypotf(float)`	Computes the length of the hypotenuse of a right angled triangle with single precision.
`double rint(double)`	Computes the integral value represented as double using the IEEE rounding mode.
`float rintf(float)`	Computes the integral value represented with single precision using the IEEE rounding mode.
`double sinh(double)`	Computes the hyperbolic sine of the double precision argument.
`float sinhf(float)`	Computes the hyperbolic sine of the single precision argument.
`float sqrtf(float)`	Computes the square root of the single precision argument.
`double tanh(double)`	Computes the hyperbolic tangent of the double precision argument.

`float tanhf(float)`	Computes the hyperbolic tangent of the single precision argument.

* Not implemented on Itanium®-based systems.

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

String and Block Copy Related

Note

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

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

Miscellaneous Intrinsics

Note

Except for `_enable()` and `_disable()`, these functions have not been implemented for Itanium® instructions.

Intrinsic	Description
`void *_alloca(int)`	Allocates the buffers.
`int _setjmp(jmp_buf)*`	A fast version of `setjmp()`, which bypasses the termination handling. Saves the callee-save registers, stack pointer and return address.
`_exception_code(void)`	Returns the exception code.
`_exception_info(void)`	Returns the exception information.
`_abnormal_termination(void)`	Can be invoked only by termination handlers. Returns `TRUE` if the termination handler is invoked as a result of a premature exit of the corresponding try-finally region.
`void _enable()`	Enables the interrupt.
`void _disable()`	Disables the interrupt.
`int _bswap(int)`	Intrinsic that maps to the IA-32 instruction `BSWAP` (swap bytes). Convert little/big endian 32-bit argument to big/little endian form
`int _in_byte(int)`	Intrinsic that maps to the IA-32 instruction `IN`. Transfer data byte from port specified by argument.
`int _in_dword(int)`	Intrinsic that maps to the IA-32 instruction `IN`. Transfer double word from port specified by argument.
`int _in_word(int)`	Intrinsic that maps to the IA-32 instruction `IN`. Transfer word from port specified by argument.
`int _inp(int)`	Same as `_in_byte`
`int _inpd(int)`	Same as `_in_dword`
`int _inpw(int)`	Same as `_in_word`
`int _out_byte(int, int)`	Intrinsic that maps to the IA-32 instruction `OUT`. Transfer data byte in second argument to port specified by first argument.
`int _out_dword(int, int)`	Intrinsic that maps to the IA-32 instruction `OUT`. Transfer double word in second argument to port specified by first argument.
`int _out_word(int, int)`	Intrinsic that maps to the IA-32 instruction `OUT`. Transfer word in second argument to port specified by first argument.
`int _outp(int, int)`	Same as `_out_byte`

`int _outpd(int, int)`	Same as `_out_dword`
`int _outpw(int, int)`	Same as `_out_word`

* Implemented as a library function call.

Support for MMX(TM) Technology

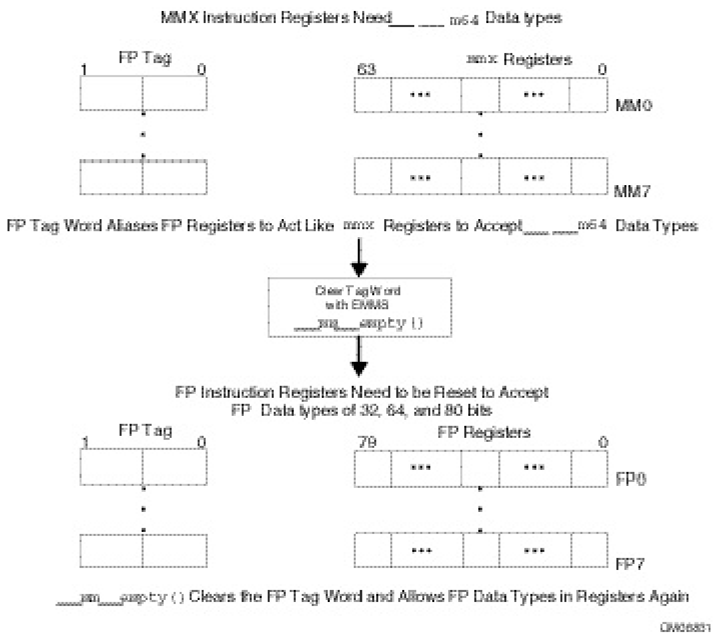
MMX(TM) technology is an extension to the Intel architecture (IA) instruction set. The MMX instruction set adds 57 opcodes and a 64-bit quadword data type, and eight 64-bit registers. Each of the eight registers can be directly addressed using the register names `mm0` to `mm7`.

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

The EMMS Instruction: Why You Need It

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

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


MMX Instruction Registers Need __m64 Data types
FP Tag
1
0
mmx Registers
63
0
MM0
MM7
FP Tag Word Aliases FP Registers to Act Like mmx Registers to Accept __m64 Data Types
Clear Tag Word with EMMS
__mm_empty ()
FP Instruction Registers Need to be Reset to Accept FP Data types of 32, 64, and 80 bits
FP Tag
FP Registers
79
FP0
FP7
__mm_empty () Clears the FP Tag Word and Allows FP Data Types in Registers Again
OM06831

Caution

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

EMMS Usage Guidelines

The guidelines when to use `EMMS` are:

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

Incorrect Usage	**Correct Usage**

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

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

MMX(TM) Technology General Support Intrinsics

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

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

```
void  _m_empty(void)
```

Empty the multimedia state.
See The EMMS Instruction: Why You Need It for details.

```
__m64 _m_from_int(int i)
```

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

```
int   _m_to_int(__m64 m)
```

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

```
__m64 _m_packsswb(__m64 m1, __m64 m2)
```

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

```
__m64 _m_packssdw(__m64 m1, __m64 m2)
```

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

`__m64 _m_packuswb(__m64 m1, __m64 m2)`

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

`__m64 _m_punpckhbw(__m64 m1, __m64 m2)`

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

`__m64 _m_punpckhwd(__m64 m1, __m64 m2)`

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

`__m64 _m_punpckhdq(__m64 m1, __m64 m2)`

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

`__m64 _m_punpcklbw(__m64 m1, __m64 m2)`

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

`__m64 _m_punpcklwd(__m64 m1, __m64 m2)`

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

`__m64 _m_punpckldq(__m64 m1, __m64 m2)`

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

MMX(TM) Technology Packed Arithmetic Intrinsics

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

Intrinsic Name	Alternate Name	Corresponding Instruction	Operation	Signed	Argument Values/Bits	Result Values/B
`_m_paddb`	`_mm_add_pi8`	`PADDB`	Addition	--	8/8	8/8
`_m_paddw`	`_mm_add_pi16`	`PADDW`	Addition	--	4/16	4/16
`_m_paddd`	`_mm_add_pi32`	`PADDD`	Addition	--	2/32	2/32
`_m_paddsb`	`_mm_adds_pi8`	`PADDSB`	Addition	Yes	8/8	8/8
`_m_paddsw`	`_mm_adds_pi16`	`PADDSW`	Addition	Yes	4/16	4/16
`_m_paddusb`	`_mm_adds_pu8`	`PADDUSB`	Addition	No	8/8	8/8
`_m_paddusw`	`_mm_adds_pu16`	`PADDUSW`	Addition	No	4/16	4/16
`_m_psubb`	`_mm_sub_pi8`	`PSUBB`	Subtraction	--	8/8	8/8
`_m_psubw`	`_mm_sub_pi16`	`PSUBW`	Subtraction	--	4/16	4/16
`_m_psubd`	`_mm_sub_pi32`	`PSUBD`	Subtraction	--	2/32	2/32
`_m_psubsb`	`_mm_subs_pi8`	`PSUBSB`	Subtraction	Yes	8/8	8/8
`_m_psubsw`	`_mm_subs_pi16`	`PSUBSW`	Subtraction	Yes	4/16	4/16
`_m_psubusb`	`_mm_subs_pu8`	`PSUBUSB`	Subtraction	No	8/8	8/8
`_m_psubusw`	`_mm_subs_pu16`	`PSUBUSW`	Subtraction	No	4/16	4/16
`_m_pmaddwd`	`_mm_madd_pi16`	`PMADDWD`	Multiplication	--	4/16	2/32
`_m_pmulhw`	`_mm_mulhi_pi16`	`PMULHW`	Multiplication	Yes	4/16	4/16 (high
`_m_pmullw`	`_mm_mullo_pi16`	`PMULLW`	Multiplication	--	4/16	4/16 (low)

`__m64 _m_paddb(__m64 m1, __m64 m2)`

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

`__m64 _m_paddw(__m64 m1, __m64 m2)`

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

`__m64 _m_paddd(__m64 m1, __m64 m2)`

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

`__m64 _m_paddsb(__m64 m1, __m64 m2)`

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

`__m64 _m_paddsw(__m64 m1, __m64 m2)`

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

`__m64 _m_paddusb(__m64 m1, __m64 m2)`

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

`__m64 _m_paddusw(__m64 m1, __m64 m2)`

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

`__m64 _m_psubb(__m64 m1, __m64 m2)`

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

`__m64 _m_psubw(__m64 m1, __m64 m2)`

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

`__m64 _m_psubd(__m64 m1, __m64 m2)`

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

`__m64 _m_psubsb(__m64 m1, __m64 m2)`

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

`__m64 _m_psubsw(__m64 m1, __m64 m2)`

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

`__m64 _m_psubusb(__m64 m1, __m64 m2)`

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

`__m64 _m_psubusw(__m64 m1, __m64 m2)`

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

`__m64 _m_pmaddwd(__m64 m1, __m64 m2)`

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

`__m64 _m_pmulhw(__m64 m1, __m64 m2)`

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

`__m64 _m_pmullw(__m64 m1, __m64 m2)`

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

MMX(TM) Technology Shift Intrinsics

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

Intrinsic Name	Alternate Name	Shift Direction	Shift Type	Corresponding Instruction
`_m_psllw`	`_mm_sll_pi16`	left	Logical	`PSLLW`
`_m_psllwi`	`_mm_slli_pi16`	left	Logical	`PSLLWI`
`_m_pslld`	`_mm_sll_pi32`	left	Logical	`PSLLD`
`_m_pslldi`	`_mm_slli_pi32`	left	Logical	`PSLLDI`
`_m_psllq`	`_mm_sll_si64`	left	Logical	`PSLLQ`
`_m_psllqi`	`_mm_slli_si64`	left	Logical	`PSLLQI`
`_m_psraw`	`_mm_sra_pi16`	right	Arithmetic	`PSRAW`
`_m_psrawi`	`_mm_srai_pi16`	right	Arithmetic	`PSRAWI`
`_m_psrad`	`_mm_sra_pi32`	right	Arithmetic	`PSRAD`
`_m_psradi`	`_mm_srai_pi32`	right	Arithmetic	`PSRADI`
`_m_psrlw`	`_mm_srl_pi16`	right	Logical	`PSRLW`
`_m_psrlwi`	`_mm_srli_pi16`	right	Logical	`PSRLWI`
`_m_psrld`	`_mm_srl_pi32`	right	Logical	`PSRLD`
`_m_psrldi`	`_mm_srli_pi32`	right	Logical	`PSRLDI`
`_m_psrlq`	`_mm_srl_si64`	right	Logical	`PSRLQ`
`_m_psrlqi`	`_mm_srli_si64`	right	Logical	`PSRLQI`

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

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

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

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

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

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

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

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

`__m64 _m_psllq(__m64 m, __m64 count)`

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

`__m64 _m_psllqi(__m64 m, int count)`

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

`__m64 _m_psraw(__m64 m, __m64 count)`

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

`__m64 _m_psrawi(__m64 m, int count)`

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

`__m64 _m_psrad(__m64 m, __m64 count)`

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

`__m64 _m_psradi(__m64 m, int count)`

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

`__m64 _m_psrlw(__m64 m, __m64 count)`

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

`__m64 _m_psrlwi(__m64 m, int count)`

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

`__m64 _m_psrld(__m64 m, __m64 count)`

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

`__m64 _m_psrldi(__m64 m, int count)`

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

`__m64 _m_psrlq(__m64 m, __m64 count)`

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

`__m64 _m_psrlqi(__m64 m, int count)`

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

MMX(TM) Technology Logical Intrinsics

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

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

`__m64 _m_pand(__m64 m1, __m64 m2)`

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

`__m64 _m_pandn(__m64 m1, __m64 m2)`

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

`__m64 _m_por(__m64 m1, __m64 m2)`

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

`__m64 _m_pxor(__m64 m1, __m64 m2)`

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

MMX(TM) Technology Compare Intrinsics

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

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

`__m64 _m_pcmpeqb(__m64 m1, __m64 m2)`

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

`__m64 _m_pcmpeqw(__m64 m1, __m64 m2)`

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

`__m64 _m_pcmpeqd(__m64 m1, __m64 m2)`

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

`__m64 _m_pcmpgtb(__m64 m1, __m64 m2)`

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

`__m64 _m_pcmpgtw(__m64 m1, __m64 m2)`

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

`__m64 _m_pcmpgtd(__m64 m1, __m64 m2)`

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

MMX(TM) Technology Set Intrinsics

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

Intrinsic Name	Operation	Number of Elements	Element Bit Size	Signed	Reverse Order
`_mm_setzero_si64`	set to zero	1	64	No	No
`_mm_set_pi32`	set integer values	2	32	No	No
`_mm_set_pi16`	set integer values	4	16	No	No
`_mm_set_pi8`	set integer values	8	8	No	No
`_mm_set1_pi32`	set integer values	2	32	Yes	No
`_mm_set1_pi16`	set integer values	4	16	Yes	No
`_mm_set1_pi8`	set integer values	8	8	Yes	No
`_mm_setr_pi32`	set integer values	2	32	No	Yes
`_mm_setr_pi16`	set integer values	4	16	No	Yes
`_mm_setr_pi8`	set integer values	8	8	No	Yes

Note

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

`__m64 _mm_setzero_si64()`

PXOR
Sets the 64-bit value to zero.
```
r := 0x0
```

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

Sets the 2 signed 32-bit integer values.
```
r0 := i0
r1 := i1
```

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

(composite) Sets the 4 signed 16-bit integer values.
```
r0 := w0
r1 := w1
r2 := w2
r3 := w3
```

`__m64 _mm_set_pi8(char b7, char b6, char b5, char b4, char b3, char b2, char b1, char b0)` (composite) Sets the 8 signed 8-bit integer values.

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

`__m64 _mm_set1_pi32(int i)`

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

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

`__m64 _mm_set1_pi16(short s)`

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

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

`__m64 _mm_set1_pi8(char b)`

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

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

`__m64 _mm_setr_pi32(int i1, int i0)`

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

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

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

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

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

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

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

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

MMX(TM) Technology Intrinsics on Itanium® Architecture

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

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

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

Data Types

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

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

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

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

Intrinsics Support for Streaming SIMD Extensions

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

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

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

Floating-point Intrinsics for Streaming SIMD Extensions

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

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

Arithmetic Operations for Streaming SIMD Extensions

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

Intrinsic	Instruction	Operation	R0	R1	R2	R3
`_mm_add_ss`	`ADDSS`	Addition	a0 [op] b0	a1	a2	a3
`_mm_add_ps`	`ADDPS`	Addition	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
`_mm_sub_ss`	`SUBSS`	Subtraction	a0 [op] b0	a1	a2	a3
`_mm_sub_ps`	`SUBPS`	Subtraction	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
`_mm_mul_ss`	`MULSS`	Multiplication	a0 [op] b0	a1	a2	a3
`_mm_mul_ps`	`MULPS`	Multiplication	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
`_mm_div_ss`	`DIVSS`	Division	a0 [op] b0	a1	a2	a3
`_mm_div_ps`	`DIVPS`	Division	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
`_mm_sqrt_ss`	`SQRTSS`	Squared Root	[op] a0	a1	a2	a3
`_mm_sqrt_ps`	`SQRTPS`	Squared Root	[op] a0	[op] b1	[op] b2	[op] b3
`_mm_rcp_ss`	`RCPSS`	Reciprocal	[op] a0	a1	a2	a3
`_mm_rcp_ps`	`RCPPS`	Reciprocal	[op] a0	[op] b1	[op] b2	[op] b3
`_mm_rsqrt_ss`	`RSQRTSS`	Reciprocal Square Root	[op] a0	a1	a2	a3
`_mm_rsqrt_ps`	`RSQRTPS`	Reciprocal Squared Root	[op] a0	[op] b1	[op] b2	[op] b3
`_mm_min_ss`	`MINSS`	Computes Minimum	[op] (a0,b0)	a1	a2	a3
`_mm_min_ps`	`MINPS`	Computes Minimum	[op] (a0,b0)	[op] (a1, b1)	[op] (a2, b2)	[op] (a3, b3)
`_mm_max_ss`	`MAXSS`	Computes Maximum	[op] (a0,b0)	a1	a2	a3
`_mm_max_ps`	`MAXPS`	Computes Maximum	[op] (a0,b0)	[op] (a1, b1)	[op] (a2, b2)	[op] (a3, b3)

`__m128 _mm_add_ss(__m128 a, __m128 b)`

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

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

`__m128 _mm_add_ps(__m128 a, __m128 b)`

Adds the four SP FP values of a and b.

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

`__m128 _mm_sub_ss(__m128 a, __m128 b)`

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

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

`__m128 _mm_sub_ps(__m128 a, __m128 b)`

Subtracts the four SP FP values of a and b.

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

`__m128 _mm_mul_ss(__m128 a, __m128 b)`

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

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

`__m128 _mm_mul_ps(__m128 a, __m128 b)`

Multiplies the four SP FP values of a and b.

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

`__m128 _mm_div_ss(__m128 a, __m128 b )`

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

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

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

Divides the four SP FP values of a and b.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Logical Operations for Streaming SIMD Extensions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Comparisons for Streaming SIMD Extensions

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

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

Intrinsic Name	Comparison	Corresponding Instruction
`_mm_cmpeq_ss`	Equal	`CMPEQSS`
`_mm_cmpeq_ps`	Equal	`CMPEQPS`
`_mm_cmplt_ss`	Less Than	`CMPLTSS`
`_mm_cmplt_ps`	Less Than	`CMPLTPS`
`_mm_cmple_ss`	Less Than or Equal	`CMPLESS`
`_mm_cmple_ps`	Less Than or Equal	`CMPLEPS`
`_mm_cmpgt_ss`	Greater Than	`CMPLTSS`
`_mm_cmpgt_ps`	Greater Than	`CMPLTPS`
`_mm_cmpge_ss`	Greater Than or Equal	`CMPLESS`
`_mm_cmpge_ps`	Greater Than or Equal	`CMPLEPS`
`_mm_cmpneq_ss`	Not Equal	`CMPNEQSS`
`_mm_cmpneq_ps`	Not Equal	`CMPNEQPS`
`_mm_cmpnlt_ss`	Not Less Than	`CMPNLTSS`
`_mm_cmpnlt_ps`	Not Less Than	`CMPNLTPS`
`_mm_cmpnle_ss`	Not Less Than or Equal	`CMPNLESS`
`_mm_cmpnle_ps`	Not Less Than or Equal	`CMPNLEPS`
`_mm_cmpngt_ss`	Not Greater Than	`CMPNLTSS`
`_mm_cmpngt_ps`	Not Greater Than	`CMPNLTPS`
`_mm_cmpnge_ss`	Not Greater Than or Equal	`CMPNLESS`
`_mm_cmpnge_ps`	Not Greater Than or Equal	`CMPNLEPS`
`_mm_cmpord_ss`	Ordered	`CMPORDSS`
`_mm_cmpord_ps`	Ordered	`CMPORDPS`
`_mm_cmpunord_ss`	Unordered	`CMPUNORDSS`

_mm_cmpunord_ps	Unordered	CMPUNORDPS
_mm_comieq_ss	Equal	COMISS
_mm_comilt_ps	Less Than	COMISS
_mm_comile_ss	Less Than or Equal	COMISS
_mm_comigt_ss	Greater Than	COMISS
_mm_comige_ss	Greater Than or Equal	COMISS
_mm_comineq_ss	Not Equal	COMISS
_mm_ucomieq_ss	Equal	UCOMISS
_mm_ucomilt_ss	Less Than	UCOMISS
_mm_ucomile_ss	Less Than or Equal	UCOMISS
_mm_ucomigt_ss	Greater Than	UCOMISS
_mm_ucomige_ss	Greater Than or Equal	UCOMISS
_mm_ucomineq_ss	Not Equal	UCOMISS

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

Compare for equality.

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

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

Compare for equality.

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

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

Compare for less-than.

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

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

Compare for less-than.

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

```
r2 := (a2 < b2) ? 0xffffffff : 0x0
r3 := (a3 < b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmple_ss(__m128 a, __m128 b)`

Compare for less-than-or-equal.

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

`__m128 _mm_cmple_ps(__m128 a, __m128 b)`

Compare for less-than-or-equal.

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

`__m128 _mm_cmpgt_ss(__m128 a, __m128 b)`

Compare for greater-than.

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

`__m128 _mm_cmpgt_ps(__m128 a, __m128 b)`

Compare for greater-than.

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

`__m128 _mm_cmpge_ss(__m128 a, __m128 b)`

Compare for greater-than-or-equal.

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

`__m128 _mm_cmpge_ps(__m128 a, __m128 b)`

Compare for greater-than-or-equal.

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

`__m128 _mm_cmpneq_ss(__m128 a, __m128 b)`

Compare for inequality.

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

`__m128 _mm_cmpneq_ps(__m128 a, __m128 b)`

Compare for inequality.

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

`__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)`

Compare for not-less-than.

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

`__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)`

Compare for not-less-than.

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

`__m128 _mm_cmpnle_ss(__m128 a, __m128 b)`

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

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

`__m128 _mm_cmpnle_ps(__m128 a, __m128 b)`

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

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

`__m128 _mm_cmpngt_ss(__m128 a, __m128 b)`

Compare for not-greater-than.

```
r0 := !(a0 > b0) ? 0xffffffff : 0x0
```

```
      r1 := a1 ; r2 := a2 ; r3 := a3
```

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

Compare for not-greater-than.

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

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

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

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

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

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

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

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

Compare for ordered.

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

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

Compare for ordered.

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

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

Compare for unordered.

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

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

Compare for unordered.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Conversion Operations for Streaming SIMD Extensions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Load Operations for Streaming SIMD Extensions

See summary table in Summary of Memory and Initialization topic.

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

`__m128 _mm_load_ss(float * p )`

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

```
r0 := *p

r1 := 0.0 ; r2 := 0.0 ; r3 := 0.0
```

`__m128 _mm_load_ps1(float * p )`

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

```
r0 := *p

r1 := *p

r2 := *p

r3 := *p
```

`__m128 _mm_load_ps(float * p )`

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

```
r0 := p[0]

r1 := p[1]

r2 := p[2]

r3 := p[3]
```

`__m128 _mm_loadu_ps(float * p)`

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

```
r0 := p[0]

r1 := p[1]

r2 := p[2]

r3 := p[3]
```

`__m128 _mm_loadr_ps(float * p)`

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

```
r0 := p[3]

r1 := p[2]

r2 := p[1]

r3 := p[0]
```

Set Operations for Streaming SIMD Extensions

See summary table in Summary of Memory and Initialization topic.

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

`__m128 _mm_set_ss(float w )`

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

```
r0 := w

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

`__m128 _mm_set_ps1(float w )`

Sets the four SP FP values to `w`.

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

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

Sets the four SP FP values to the four inputs.

```
r0 := w

r1 := x

r2 := y

r3 := z
```

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

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

```
r0 := z

r1 := y

r2 := x

r3 := w
```

`__m128 _mm_setzero_ps(void)`

Clears the four SP FP values.

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

Store Operations for Streaming SIMD Extensions

See summary table in Summary of Memory and Initialization topic.

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

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

Stores the lower SP FP value.

```
*p := a0
```

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

Stores the lower SP FP value across four words.

```
p[0] := a0

p[1] := a0

p[2] := a0

p[3] := a0
```

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

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

```
p[0] := a0

p[1] := a1

p[2] := a2

p[3] := a3
```

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

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

```
p[0] := a0

p[1] := a1

p[2] := a2

p[3] := a3
```

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

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

```
p[0] := a3

p[1] := a2

p[2] := a1

p[3] := a0
```

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

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

```
r0 := b0

r1 := a1

r2 := a2

r3 := a3
```

Cacheability Support Using Streaming SIMD Extensions

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

```
void _mm_pause(void)
```

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

PAUSE Intrinsic

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

Example of loop with the PAUSE instruction:

```
spin_loop:pause

cmp eax, A

jne spin_loop
```

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

```
get_lock: mov eax, 1

xchg eax, A ; Try to get lock

cmp eax, 0 ; Test if successful

jne spin_loop
```

Critical Section:

<critical_section code>

```
mov A, 0 ; Release lock

jmp continue

spin_loop: pause; Spin-loop hint

cmp 0, A ; Check lock availability
```

```
jne spin_loop

jmp get_lock

continue: <other code>
```

Note that the first branch is predicted to fall-through to the critical section in anticipation of successfully gaining access to the lock. It is highly recommended that all spin-wait loops include the `PAUSE` instruction. Since `PAUSE` is backwards compatible to all existing IA-32 processor generations, a test for processor type (a CPUID test) is not needed. All legacy processors will execute `PAUSE` as a `NOP`, but in processors which use the `PAUSE` as a hint there can be significant performance benefit.

Integer Intrinsics Using Streaming SIMD Extensions

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

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

Intrinsic Name	Alternate Name	Operation	Corresponding Instruction
`_m_pextrw`	`_mm_extract_pi16`	Extract on of four words	`PEXTRW`
`_m_pinsrw`	`_mm_insert_pi16`	Insert a word	`PINSRW`
`_m_pmaxsw`	`_mm_max_pi16`	Compute the maximum	`PMAXSW`
`_m_pmaxub`	`_mm_max_pu8`	Compute the maximum, unsigned	`PMAXUB`
`_m_pminsw`	`_mm_min_pi16`	Compute the minimum	`PMINSW`
`_m_pminub`	`_mm_min_pu8`	Compute the minimum, unsigned	`PMINUB`
`_m_pmovmskb`	`_mm_movemask_pi8`	Create an eight-bit mask	`PMOVMSKB`
`_m_pmulhuw`	`_mm_mulhi_pu16`	Multiply, return high bits	`PMULHUW`
`_m_pshufw`	`_mm_shuffle_pi16`	Return a combination of four words	`PSHUFW`
`_m_maskmovq`	`_mm_maskmove_si64`	Conditional Store	`MASKMOVQ`
`_m_pavgb`	`_mm_avg_pu8`	Compute rounded average	`PAVGB`
`_m_pavgw`	`_mm_avg_pu16`	Compute rounded average	`PAVGW`
`_m_psadbw`	`_mm_sad_pu8`	Compute sum of absolute differences	`PSADBW`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Memory and Initialization Using Streaming SIMD Extensions

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

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

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

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

`_mm_getcsr`		Return register contents	`STMXCSR`
`_mm_setcsr`		Control Register	`LDMXCSR`
`_mm_prefetch`			
`_mm_stream_pi`			
`_mm_stream_ps`			
`_mm_sfence`			

`__m128 _mm_load_ss(float const*a)`

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

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

`__m128 _mm_load_ps1(float const*a)`

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

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

`__m128 _mm_load_ps(float const*a)`

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

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

`__m128 _mm_loadu_ps(float const*a)`

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

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

`__m128 _mm_loadr_ps(float const*a)`

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

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

`__m128 _mm_set_ss(float a)`

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

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

`__m128 _mm_set_ps1(float a)`

Sets the four SP FP values to a.

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

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

Sets the four SP FP values to the four inputs.

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

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

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

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

`__m128 _mm_setzero_ps(void)`

Clears the four SP FP values.

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

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

Stores the lower SP FP value.

```
*v := a0
```

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

Stores the lower SP FP value across four words.

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

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

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

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

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

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

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

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

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

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

`__m128 _mm_move_ss(__m128 a, __m128 b)`

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

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

`unsigned int _mm_getcsr(void)`

Returns the contents of the control register.

`void _mm_setcsr(unsigned int i)`

Sets the control register to the value specified.

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

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

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

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

Why You Need It and When to Use It topic.

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

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

`void _mm_sfence(void)`

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

Miscellaneous Intrinsics Using Streaming SIMD Extensions

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

Intrinsic Name	**Operation**	**Corresponding Instruction**
`_mm_shuffle_ps`	Shuffle	`SHUFPS`
`_mm_unpackhi_ps`	Unpack High	`UNPCKHPS`
`_mm_unpacklo_ps`	Unpack Low	`UNPCKLPS`
`_mm_loadh_pi`	Load High	`MOVHPS reg, mem`
`_mm_storeh_pi`	Store High	`MOVHPS mem, reg`
`_mm_movehl_ps`	Move High to Low	`MOVHLPS`
`_mm_movelh_ps`	Move Low to High	`MOVLHPS`
`_mm_loadl_pi`	Load Low	`MOVLPS reg, mem`
`_mm_storel_pi`	Store Low	`MOVLPS mem, reg`
`_mm_movemask_ps`	Create four-bit mask	`MOVMSKPS`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Using Streaming SIMD Extensions on Itanium® Architecture

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

To write programs with the intrinsics, you should be familiar with the hardware features provided by the Streaming SIMD Extensions. Keep the following four important issues in mind:

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

Data Types

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

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

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

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

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

Implementation on Itanium-based systems

Streaming SIMD Extensions intrinsics are defined for the `__m128` data type, a 128-bit quantity consisting of four single-precision FP values. SIMD instructions for Itanium-based systems operate on 64-bit FP register quantities containing two single-precision floating-point values. Thus, each `__m128` operand is actually a pair of FP registers and therefore each intrinsic corresponds to at least one pair of Itanium instructions operating on the pair of FP register operands.

Compatibility versus Performance

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

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

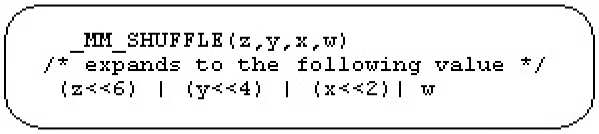
- Any Streaming SIMD Extensions scalar intrinsic (`_ss variety`) - use packed (`_ps`) version if possible
- `comi` and `ucomi` Streaming SIMD Extensions comparisons - these correspond to IA-32 `COMISS` and `UCOMISS` instructions only. A sequence of Itanium instructions are required to implement these.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

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

Macro Function for Shuffle Using Streaming SIMD Extensions

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

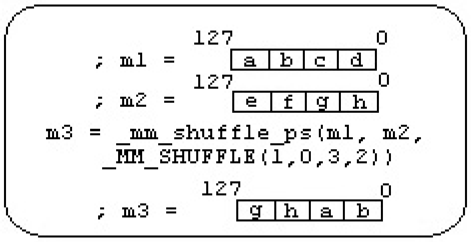
Shuffle Function Macro



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

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

View of Original and Result Words with Shuffle Function Macro



```
m3 = _mm_shuffle_ps(m1, m2,
     _MM_SHUFFLE(1,0,3,2))
```

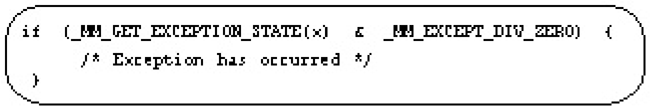
Macro Functions to Read and Write the Control Registers

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

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

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

Exception State Macros with _MM_EXCEPT_DIV_ZERO



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

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

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

Exception Mask with _MM_MASK_OVERFLOW and _MM_MASK_UNDERFLOW
`_MM_SET_EXCEPTION_MASK(MM_MASK_OVERFLOW | _MM_MASK_UNDERFLOW)`

Rounding Mode	Macro Arguments
`_MM_SET_ROUNDING_MODE(x)`	`_MM_ROUND_NEAREST`
`_MM_GET_ROUNDING_MODE()`	`_MM_ROUND_DOWN`
Macro Definition Write to and read from bits thirteen and fourteen of the control register.	`_MM_ROUND_UP`
	`_MM_ROUND_TOWARD_ZERO`

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

Rounding Mode with _MM_ROUND_TOWARD_ZERO

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

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

The following example disables flush-to-zero mode.

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

Macro Function for Matrix Transposition

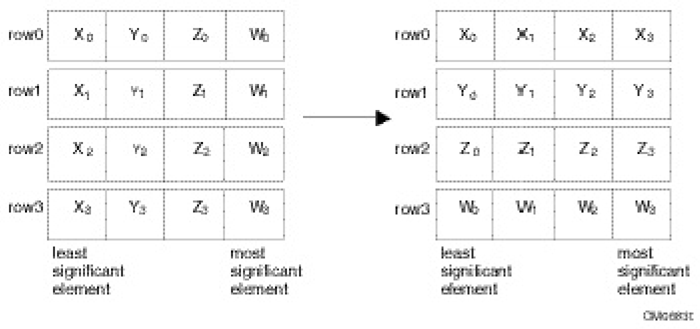
The Streaming SIMD Extensions also provide the following macro function to transpose a 4 by 4 matrix of single precision floating point values.

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

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

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

Matrix Transposition Using _MM_TRANSPOSE4_PS Macro



Overview: Streaming SIMD Extensions 2 Intrinsics

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

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

Note

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

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

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

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

Floating-point Arithmetic Operations for Streaming SIMD Extensions 2

The arithmetic operations for the Streaming SIMD Extensions 2 are listed in the following table and are followed by descriptions of each intrinsic.

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

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

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

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

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

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

Adds the two DP FP values of a and b.

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

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

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

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

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

Subtracts the two DP FP values of b from a.

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

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

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

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

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

Multiplies the two DP FP values of a and b.

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

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

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

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

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

Divides the two DP FP values of a and b.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Logical Operations for Streaming SIMD Extensions 2

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

`__m128d _mm_and_pd(__m128d a, __m128d b)`

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

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

`__m128d _mm_andnot_pd(__m128d a, __m128d b)`

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

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

`__m128d _mm_or_pd(__m128d a, __m128d b)`

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

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

`__m128d _mm_xor_pd(__m128d a, __m128d b)`

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

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

Comparison Operations for Streaming SIMD Extensions 2

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

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

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

`_mm_cmpnle_sd`	`CMPNLESD`	Not Less Than or Equal
`_mm_cmpngt_sd`	`CMPNLTSDr`	Not Greater Than
`_mm_cmpnge_sd`	`CMPNLESDR`	Not Greater Than or Equal
`_mm_comieq_sd`	`COMISD`	Equality
`_mm_comilt_sd`	`COMISD`	Less Than
`_mm_comile_sd`	`COMISD`	Less Than or Equal
`_mm_comigt_sd`	`COMISD`	Greater Than
`_mm_comige_sd`	`COMISD`	Greater Than or Equal
`_mm_comineq_sd`	`COMISD`	Not Equal
`_mm_ucomieq_sd`	`UCOMISD`	Equality
`_mm_ucomilt_sd`	`UCOMISD`	Less Than
`_mm_ucomile_sd`	`UCOMISD`	Less Than or Equal
`_mm_ucomigt_sd`	`UCOMISD`	Greater Than
`_mm_ucomige_sd`	`UCOMISD`	Greater Than or Equal
`_mm_ucomineq_sd`	`UCOMISD`	Not Equal

`__m128d _mm_cmpeq_pd(__m128d a, __m128d b)`

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

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

`__m128d _mm_cmplt_pd(__m128d a, __m128d b)`

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

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

`___m128d _mm_cmple_pd(__m128d a, __m128d b)`

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

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

`__m128d _mm_cmpgt_pd(__m128d a, __m128d b)`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`__m128d _mm_cmpeq_sd(__m128d a, __m128d b)`

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

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

`__m128d _mm_cmplt_sd(__m128d a, __m128d b)`

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

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

`__m128d _mm_cmple_sd(__m128d a, __m128d b)`

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

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

`__m128d _mm_cmpgt_sd(__m128d a, __m128d b)`

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

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

`__m128d _mm_cmpge_sd(__m128d a, __m128d b)`

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

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

`__m128d _mm_cmpord_sd(__m128d a, __m128d b)`

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

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

`__m128d _mm_cmpunord_sd(__m128d a, __m128d b)`

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

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

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

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

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

```
__m128d _mm_cmpnlt_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a not less than b. The upper DP FP value is passed through from a.

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

```
__m128d _mm_cmpnle_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a not less than or equal to b. The upper DP FP value is passed through from a.

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

```
__m128d _mm_cmpngt_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a not greater than b. The upper DP FP value is passed through from a.

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

```
__m128d _mm_cmpnge_sd(__m128d a, __m128d b)
```

Compares the lower DP FP value of a and b for a not greater than or equal to b. The upper DP FP value is passed through from a.

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

```
int _mm_comieq_sd(__m128d a, __m128d b)
```

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

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

`int _mm_comilt_sd(__m128d a, __m128d b)`

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

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

`int _mm_comile_sd(__m128d a, __m128d b)`

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

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

`int _mm_comigt_sd(__m128d a, __m128d b)`

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

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

`int _mm_comige_sd(__m128d a, __m128d b)`

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

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

`int _mm_comineq_sd(__m128d a, __m128d b)`

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

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

`int _mm_ucomieq_sd(__m128d a, __m128d b)`

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

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

`int _mm_ucomilt_sd(__m128d a, __m128d b)`

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

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

`int _mm_ucomile_sd(__m128d a, __m128d b)`

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

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

```
int _mm_ucomigt_sd(__m128d a, __m128d b)
```

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

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

```
int _mm_ucomige_sd(__m128d a, __m128d b)
```

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

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

```
int _mm_ucomineq_sd(__m128d a, __m128d b)
```

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

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

Conversion Operations for Streaming SIMD Extensions 2

Each conversion intrinsic takes one data type and performs a conversion to a different type. Some conversions such as `_mm_cvtpd_ps` result in a loss of precision. The rounding mode used in such cases is determined by the value in the MXCSR register. The default rounding mode is round-to-nearest. Note that the rounding mode used by the C and C++ languages when performing a type conversion is to truncate. The `_mm_cvttpd_epi32` and `_mm_cvttsd_si32` intrinsics use the truncate rounding mode regardless of the mode specified by the `MXCSR` register.

The conversion-operation intrinsics for Streaming SIMD Extensions 2 are listed in the following table followed by detailed descriptions.

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

Intrinsic Name	**Corresponding Instruction**	**Return Type**	**Parameters**
`_mm_cvtpd_ps`	`CVTPD2PS`	`__m128`	`(__m128d a)`
`_mm_cvtps_pd`	`CVTPS2PD`	`__m128d`	`(__m128 a)`
`_mm_cvtepi32_pd`	`CVTDQ2PD`	`__m128d`	`(__m128i a)`
`_mm_cvtpd_epi32`	`CVTPD2DQ`	`__m128i`	`(__m128d a)`
`_mm_cvtsd_si32`	`CVTSD2SI`	`int`	`(__m128d a)`
`_mm_cvtsd_ss`	`CVTSD2SS`	`__m128`	`(__m128 a, __m128d b)`
`_mm_cvtsi32_sd`	`CVTSI2SD`	`__m128d`	`(__m128d a, int b)`
`_mm_cvtss_sd`	`CVTSS2SD`	`__m128d`	`(__m128d a, __m128 b)`
`_mm_cvttpd_epi32`	`CVTTPD2DQ`	`__m128i`	`(__m128d a)`
`_mm_cvttsd_si32`	`CVTTSD2SI`	`int`	`(__m128d a)`
`_mm_cvtpd_pi32`	`CVTPD2PI`	`__m64`	`(__m128d a)`
`_mm_cvttpd_pi32`	`CVTTPD2PI`	`__m64`	`(__m128d a)`
`_mm_cvtpi32_pd`	`CVTPI2PD`	`__m128d`	`(__m64 a)`

`__m128 _mm_cvtpd_ps(__m128d a)`

Converts the two DP FP values of a to SP FP values.

```
r0 := (float) a0
r1 := (float) a1
r2 := 0.0 ; r3 := 0.0
```

`__m128d _mm_cvtps_pd(__m128 a)`

Converts the lower two SP FP values of a to DP FP values.

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

`__m128d _mm_cvtepi32_pd(__m128i a)`

Converts the lower two signed 32-bit integer values of a to DP FP values.

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

`__m128i _mm_cvtpd_epi32(__m128d a)`

Converts the two DP FP values of a to 32-bit signed integer values.

```
r0 := (int) a0
r1 := (int) a1
r2 := 0x0 ; r3 := 0x0
```

`int _mm_cvtsd_si32(__m128d a)`

Converts the lower DP FP value of a to a 32-bit signed integer value.

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

`__m128 _mm_cvtsd_ss(__m128 a, __m128d b)`

Converts the lower DP FP value of b to an SP FP value. The upper SP FP values in a are passed through.

```
r0 := (float) b0
r1 := a1; r2 := a2 ; r3 := a3
```

`__m128d _mm_cvtsi32_sd(__m128d a, int b)`

Converts the signed integer value in b to a DP FP value. The upper DP FP value in a is passed through.

```
r0 := (double) b
r1 := a1
```

`__m128d _mm_cvtss_sd(__m128d a, __m128 b)`

Converts the lower SP FP value of b to a DP FP value. The upper value DP FP value in a is passed through.

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

`__m128i _mm_cvttpd_epi32(__m128d a)`

Converts the two DP FP values of a to 32-bit signed integers using truncate.

```
r0 := (int) a0
r1 := (int) a1
r2 := 0x0 ; r3 := 0x0
```

`int _mm_cvttsd_si32(__m128d a)`

Converts the lower DP FP value of a to a 32-bit signed integer using truncate.

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

`__m64 _mm_cvtpd_pi32(__m128d a)`

Converts the two DP FP values of a to 32-bit signed integer values.

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

`__m64 _mm_cvttpd_pi32(__m128d a)`

Converts the two DP FP values of a to 32-bit signed integer values using truncate.

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

`__m128d _mm_cvtpi32_pd(__m64 a)`

Converts the two 32-bit signed integer values of a to DP FP values.

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

Streaming SIMD Extensions 2 Floating-point Memory and Initialization Operations

This section describes the `load`, `set`, and `store` operations, which let you load and store data into memory. The `load` and `set` operations are similar in that both initialize `__m128d` data. However, the `set` operations take a double argument and are intended for initialization with constants, while the `load` operations take a double pointer argument and are intended to mimic the instructions for loading data from memory. The `store` operation assigns the initialized data to the address.

Note

There is no intrinsic for move operations. To move data from one register to another, a simple assignment, `A = B`, suffices, where `A` and `B` are the source and target registers for the move operation.

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

Load Operations for Streaming SIMD Extensions 2

The following load operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2.

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

`__m128d _mm_load_pd(double const*dp)`

(uses `MOVAPD`) Loads two DP FP values. The address p must be 16-byte aligned.

```
r0 := p[0]
r1 := p[1]
```

`__m128d _mm_load1_pd(double const*dp)`

(uses `MOVSD` + shuffling) Loads a single DP FP value, copying to both elements. The address p need not be 16-byte aligned.

```
r0 := *p
r1 := *p
```

`__m128d _mm_loadr_pd(double const*dp)`

(uses `MOVAPD` + shuffling) Loads two DP FP values in reverse order. The address p must be 16-byte aligned.

```
r0 := p[1]
r1 := p[0]
```

`__m128d _mm_loadu_pd(double const*dp)`

(uses `MOVUPD`) Loads two DP FP values. The address p need not be 16-byte aligned.

```
r0 := p[0]
r1 := p[1]
```

`__m128d _mm_load_sd(double const*dp)`

(uses `MOVSD`) Loads a DP FP value. The upper DP FP is set to zero. The address p need not be 16-byte aligned.

```
r0 := *p
r1 := 0.0
```

`__m128d _mm_loadh_pd(__m128d a, double const*dp)`

(uses `MOVHPD`) Loads a DP FP value as the upper DP FP value of the result. The lower DP FP value is passed through from a. The address p need not be 16-byte aligned.

```
r0 := a0
r1 := *p
```

```
__m128d _mm_loadl_pd(__m128d a, double const*dp)
```

(uses `MOVLPD`) Loads a DP FP value as the lower DP FP value of the result. The upper DP FP value is passed through from `a`. The address `p` need not be 16-byte aligned.

```
r0 := *p
r1 := a1
```

Set Operations for Streaming SIMD Extensions 2

The following `set` operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2.

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

`__m128d _mm_set_sd(double w)`

(composite) Sets the lower DP FP value to `w` and sets the upper DP FP value to zero.

```
r0 := w
r1 := 0.0
```

`__m128d _mm_set1_pd(double w)`

(composite) Sets the 2 DP FP values to `w`.

```
r0 := w
r1 := w
```

`__m128d _mm_set_pd(double w, double x)`

(composite) Sets the lower DP FP value to x and sets the upper DP FP value to `w`.

```
r0 := x
r1 := w
```

`__m128d _mm_setr_pd(double w, double x)`

(composite) Sets the lower DP FP value to `w` and sets the upper DP FP value to `x`.

```
r0 := w
r1 := x
```

`__m128d _mm_setzero_pd(void)`

(uses `XORPD`) Sets the 2 DP FP values to zero.

```
r0 := 0.0
r1 := 0.0
```

`__m128d _mm_move_sd( __m128d a, __m128d b)`

(uses `MOVSD`) Sets the lower DP FP value to the lower DP FP value of b. The upper DP FP value is passed through from a.

```
r0 := b0
r1 := a1
```

Store Operations for Streaming SIMD Extensions 2

The following `store` operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2.

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

```
void _mm_store_sd(double *dp, __m128d a)
```

(uses `MOVSD`) Stores the lower DP FP value of a. The address `dp` need not be 16-byte aligned.

```
*dp := a0
```

```
void _mm_store1_pd(double *dp, __m128d a)
```

(uses `MOVAPD` + shuffling) Stores the lower DP FP value of a twice. The address `dp` must be 16-byte aligned.

```
dp[0] := a0
dp[1] := a0
```

```
void _mm_store_pd(double *dp, __m128d a)
```

(uses `MOVAPD`) Stores two DP FP values. The address `dp` must be 16-byte aligned.

```
dp[0] := a0
dp[1] := a1
```

```
void _mm_storeu_pd(double *dp, __m128d a)
```

(uses `MOVUPD`) Stores two DP FP values. The address `dp` need not be 16-byte aligned.

```
dp[0] := a0
dp[1] := a1
```

```
void _mm_storer_pd(double *dp, __m128d a)
```

(uses `MOVAPD` + shuffling) Stores two DP FP values in reverse order. The address `dp` must be 16-byte aligned.

```
dp[0] := a1
dp[1] := a0
```

```
void _mm_storeh_pd(double *dp, __m128d a)
```

(uses `MOVHPD`) Stores the upper DP FP value of a.

```
*dp := a1
```

```
void _mm_storel_pd(double *dp, __m128d a)
```

(uses `MOVLPD`) Stores the lower DP FP value of a.

```
*dp := a0
```

Miscellaneous Operations for Streaming SIMD Extensions 2

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

```
__m128d _mm_unpackhi_pd(__m128d a, __m128d b)
```

(uses `UNPCKHPD`) Interleaves the upper DP FP values of a and b.

```
r0 := a1
r1 := b1
```

```
__m128d _mm_unpacklo_pd(__m128d a, __m128d b)
```

(uses `UNPCKLPD`) Interleaves the lower DP FP values of a and b.

```
r0 := a0
1 := b0
```

```
int _mm_movemask_pd(__m128d a)
```

(uses `MOVMSKPD`) Creates a two-bit mask from the sign bits of the two DP FP values of a.

```
r := sign(a1) << 1 | sign(a0)
```

```
__m128d _mm_shuffle_pd(__m128d a, __m128d b, int i)
```

(uses `SHUFPD`) Selects two specific DP FP values from a and b, based on the mask `i`. The mask must be an immediate. See Macro Function for Shuffle for a description of the shuffle semantics.

Integer Arithmetic Operations for Streaming SIMD Extensions 2

The integer arithmetic operations for Streaming SIMD Extensions 2 are listed in the following table followed by their descriptions. The packed arithmetic intrinsics for Streaming SIMD Extensions 2 are listed in the Floating-point Arithmetic Operations topic.

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

Intrinsic	Instruction	Operation
`_mm_add_epi8`	`PADDB`	Addition
`_mm_add_epi16`	`PADDW`	Addition
`_mm_add_epi32`	`PADDD`	Addition
`_mm_add_si64`	`PADDQ`	Addition
`_mm_add_epi64`	`PADDQ`	Addition
`_mm_adds_epi8`	`PADDSB`	Addition
`_mm_adds_epi16`	`PADDSW`	Addition
`_mm_adds_epu8`	`PADDUSB`	Addition
`_mm_adds_epu16`	`PADDUSW`	Addition
`_mm_avg_epu8`	`PAVGB`	Computes Average
`_mm_avg_epu16`	`PAVGW`	Computes Average
`_mm_madd_epi16`	`PMADDWD`	Multiplication/Addition
`_mm_max_epi16`	`PMAXSW`	Computes Maxima
`_mm_max_epu8`	`PMAXUB`	Computes Maxima
`_mm_min_epi16`	`PMINSW`	Computes Minima
`_mm_min_epu8`	`PMINUB`	Computes Minima
`_mm_mulhi_epi16`	`PMULHW`	Multiplication
`_mm_mulhi_epu16`	`PMULHUW`	Multiplication
`_mm_mullo_epi16`	`PMULLW`	Multiplication
`_mm_mul_su32`	`PMULUDQ`	Multiplication
`_mm_mul_epu32`	`PMULUDQ`	Multiplication
`_mm_sad_epu8`	`PSADBW`	Computes Difference/Adds
`_mm_sub_epi8`	`PSUBB`	Subtraction

_mm_sub_epi16	PSUBW	Subtraction
_mm_sub_epi32	PSUBD	Subtraction
_mm_sub_si64	PSUBQ	Subtraction
_mm_sub_epi64	PSUBQ	Subtraction
_mm_subs_epi8	PSUBSB	Subtraction
_mm_subs_epi16	PSUBSW	Subtraction
_mm_subs_epu8	PSUBUSB	Subtraction
_mm_subs_epu16	PSUBUSW	Subtraction

`__mm128i _mm_add_epi8(__m128i a, __m128i b)`

Adds the 16 signed or unsigned 8-bit integers in a to the 16 signed or unsigned 8-bit integers in b.

```
r0 := a0 + b0
r1 := a1 + b1
...
r15 := a15 + b15
```

`__mm128i _mm_add_epi16(__m128i a, __m128i b)`

Adds the 8 signed or unsigned 16-bit integers in a to the 8 signed or unsigned 16-bit integers in b.

```
r0 := a0 + b0
r1 := a1 + b1
...
r7 := a7 + b7
```

`__m128i _mm_add_epi32(__m128i a, __m128i b)`

Adds the 4 signed or unsigned 32-bit integers in a to the 4 signed or unsigned 32-bit integers in b.

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

`__m64 _mm_add_si64(__m64 a, __m64 b)`

Adds the signed or unsigned 64-bit integer a to the signed or unsigned 64-bit integer b.

```
r := a + b
```

`__m128i _mm_add_epi64(__m128i a, __m128i b)`

Adds the 2 signed or unsigned 64-bit integers in a to the 2 signed or unsigned 64-bit integers in b.

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

`__m128i _mm_adds_epi8(__m128i a, __m128i b)`

Adds the 16 signed 8-bit integers in a to the 16 signed 8-bit integers in b using saturating arithmetic.

```
r0 := SignedSaturate(a0 + b0)
r1 := SignedSaturate(a1 + b1)
...
r15 := SignedSaturate(a15 + b15)
```

`__m128i _mm_adds_epi16(__m128i a, __m128i b)`

Adds the 8 signed 16-bit integers in a to the 8 signed 16-bit integers in b using saturating arithmetic.

```
r0 := SignedSaturate(a0 + b0)
r1 := SignedSaturate(a1 + b1)
...
r7 := SignedSaturate(a7 + b7)
```

`__m128i _mm_adds_epu8(__m128i a, __m128i b)`

Adds the 16 unsigned 8-bit integers in a to the 16 unsigned 8-bit integers in b using saturating arithmetic.

```
r0 := UnsignedSaturate(a0 + b0)
r1 := UnsignedSaturate(a1 + b1)
...
r15 := UnsignedSaturate(a15 + b15)
```

`__m128i _mm_adds_epu16(__m128i a, __m128i b)`

Adds the 8 unsigned 16-bit integers in a to the 8 unsigned 16-bit integers in b using saturating arithmetic.

```
r0 := UnsignedSaturate(a0 + b0)
r1 := UnsignedSaturate(a1 + b1)
...
r15 := UnsignedSaturate(a7 + b7)
```

`__m128i _mm_avg_epu8(__m128i a, __m128i b)`

Computes the average of the 16 unsigned 8-bit integers in a and the 16 unsigned 8-bit integers in b and rounds.

```
r0 := (a0 + b0) / 2
r1 := (a1 + b1) / 2
...
r15 := (a15 + b15) / 2
```

`__m128i _mm_avg_epu16(__m128i a, __m128i b)`

Computes the average of the 8 unsigned 16-bit integers in a and the 8 unsigned 16-bit integers in b and rounds.

```
r0 := (a0 + b0) / 2
r1 := (a1 + b1) / 2
...
r7 := (a7 + b7) / 2
```

```
__m128i _mm_madd_epi16(__m128i a, __m128i b)
```

Multiplies the 8 signed 16-bit integers from a by the 8 signed 16-bit integers from b. Adds the signed 32-bit integer results pairwise and packs the 4 signed 32-bit integer results.

```
r0 := (a0 * b0) + (a1 * b1)
r1 := (a2 * b2) + (a3 * b3)
r2 := (a4 * b4) + (a5 * b5)
r3 := (a6 * b6) + (a7 * b7)
```

```
__m128i _mm_max_epi16(__m128i a, __m128i b)
```

Computes the pairwise maxima of the 8 signed 16-bit integers from a and the 8 signed 16-bit integers from b.

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

```
__m128i _mm_max_epu8(__m128i a, __m128i b)
```

Computes the pairwise maxima of the 16 unsigned 8-bit integers from a and the 16 unsigned 8-bit integers from b.

```
r0 := max(a0, b0)
r1 := max(a1, b1)
...
r15 := max(a15, b15)
```

```
__m128i _mm_min_epi16(__m128i a, __m128i b)
```

Computes the pairwise minima of the 8 signed 16-bit integers from a and the 8 signed 16-bit integers from b.

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

```
__m128i _mm_min_epu8(__m128i a, __m128i b)
```

Computes the pairwise minima of the 16 unsigned 8-bit integers from a and the 16 unsigned 8-bit integers from b.

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

```
...
r15 := min(a15, b15)
```

`__m128i _mm_mulhi_epi16(__m128i a, __m128i b)`

Multiplies the 8 signed 16-bit integers from a by the 8 signed 16-bit integers from b. Packs the upper 16-bits of the 8 signed 32-bit results.

```
r0 := (a0 * b0)[31:16]
r1 := (a1 * b1)[31:16]
...
r7 := (a7 * b7)[31:16]
```

`__m128i _mm_mulhi_epu16(__m128i a, __m128i b)`

Multiplies the 8 unsigned 16-bit integers from a by the 8 unsigned 16-bit integers from b. Packs the upper 16-bits of the 8 unsigned 32-bit results.

```
r0 := (a0 * b0)[31:16]
r1 := (a1 * b1)[31:16]
...
r7 := (a7 * b7)[31:16]
```

`__m128i_mm_mullo_epi16(__m128i a, __m128i b)`

Multiplies the 8 signed or unsigned 16-bit integers from a by the 8 signed or unsigned 16-bit integers from b. Packs the lower 16-bits of the 8 signed or unsigned 32-bit results.

```
r0 := (a0 * b0)[15:0]
r1 := (a1 * b1)[15:0]
...
r7 := (a7 * b7)[15:0]
```

`__m64 _mm_mul_su32(__m64 a, __m64 b)`

Multiplies the lower 32-bit integer from a by the lower 32-bit integer from b, and returns the 64-bit integer result.

```
r := a0 * b0
```

`__m128i _mm_mul_epu32(__m128i a, __m128i b)`

Multiplies 2 unsigned 32-bit integers from a by 2 unsigned 32-bit integers from b. Packs the 2 unsigned 64-bit integer results.

```
r0 := a0 * b0
r1 := a2 * b2
```

`__m128i _mm_sad_epu8(__m128i a, __m128i b)`

Computes the absolute difference of the 16 unsigned 8-bit integers from a and the 16 unsigned 8-bit integers from b. Sums the upper 8 differences and lower 8 differences, and packs the resulting 2 unsigned 16-bit integers into the upper and lower 64-bit elements.

```
r0 := abs(a0 - b0) + abs(a1 - b1) +...+ abs(a7 - b7)
```

```
r1 := 0x0 ; r2 := 0x0 ; r3 := 0x0
r4 := abs(a8 - b8) + abs(a9 - b9) +...+ abs(a15 - b15)
r5 := 0x0 ; r6 := 0x0 ; r7 := 0x0
```

`__m128i _mm_sub_epi8(__m128i a, __m128i b)`

Subtracts the 16 signed or unsigned 8-bit integers of b from the 16 signed or unsigned 8-bit integers of a.

```
r0 := a0 - b0
r1 := a1 - b1
...
r15 := a15 - b15
```

`__m128i_mm_sub_epi16(__m128i a, __m128i b)`

Subtracts the 8 signed or unsigned 16-bit integers of b from the 8 signed or unsigned 16-bit integers of a.

```
r0 := a0 - b0
r1 := a1 - b1
...
r7 := a7 - b7
```

`__m128i _mm_sub_epi32(__m128i a, __m128i b)`

Subtracts the 4 signed or unsigned 32-bit integers of b from the 4 signed or unsigned 32-bit integers of a.

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

`__m64 _mm_sub_si64 (__m64 a, __m64 b)`

Subtracts the signed or unsigned 64-bit integer b from the signed or unsigned 64-bit integer a.

```
r := a - b
```

`__m128i _mm_sub_epi64(__m128i a, __m128i b)`

Subtracts the 2 signed or unsigned 64-bit integers in b from the 2 signed or unsigned 64-bit integers in a.

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

`__m128i _mm_subs_epi8(__m128i a, __m128i b)`

Subtracts the 16 signed 8-bit integers of b from the 16 signed 8-bit integers of a using saturating arithmetic.

```
r0 := SignedSaturate(a0 - b0)
r1 := SignedSaturate(a1 - b1)
...
```

```
r15 := SignedSaturate(a15 - b15)
```

```
__m128i _mm_subs_epi16(__m128i a, __m128i b)
```

Subtracts the 8 signed 16-bit integers of b from the 8 signed 16-bit integers of a using saturating arithmetic.

```
r0 := SignedSaturate(a0 - b0)
r1 := SignedSaturate(a1 - b1)
...
r7 := SignedSaturate(a7 - b7)
```

```
__m128i _mm_subs_epu8(__m128i a, __m128i b)
```

Subtracts the 16 unsigned 8-bit integers of b from the 16 unsigned 8-bit integers of a using saturating arithmetic.

```
r0 := UnsignedSaturate(a0 - b0)
r1 := UnsignedSaturate(a1 - b1)
...
r15 := UnsignedSaturate(a15 - b15)
```

```
__m128i _mm_subs_epu16(__m128i a, __m128i b)
```

Subtracts the 8 unsigned 16-bit integers of b from the 8 unsigned 16-bit integers of a using saturating arithmetic.

```
r0 := UnsignedSaturate(a0 - b0)
r1 := UnsignedSaturate(a1 - b1)
...
r7 := UnsignedSaturate(a7 - b7)
```

Integer Logical Operations for Streaming SIMD Extensions 2

The following four logical-operation intrinsics and their respective instructions are functional as part of Streaming SIMD Extensions 2.

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

```
__m128i _mm_and_si128(__m128i a, __m128i b)
```

(uses `PAND`) Computes the bitwise AND of the 128-bit value in a and the 128-bit value in b.

```
r := a & b
```

```
__m128i _mm_andnot_si128(__m128i a, __m128i b)
```

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

```
r := (~a) & b
```

```
__m128i _mm_or_si128(__m128i a, __m128i b)
```

(uses `POR`) Computes the bitwise OR of the 128-bit value in a and the 128-bit value in b.

```
r := a | b
```

```
__m128i _mm_xor_si128(__m128i a, __m128i b)
```

(uses `PXOR`) Computes the bitwise XOR of the 128-bit value in a and the 128-bit value in b.

```
r := a ^ b
```

Integer Shift Operations for Streaming SIMD Extensions 2

The shift-operation intrinsics for Streaming SIMD Extensions 2 and the description for each are listed in the following table.

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

Intrinsic	Shift Direction	Shift Type	Corresponding Instruction
`_mm_slli_si128`	Left	Logical	`PSLLDQ`
`_mm_slli_epi16`	Left	Logical	`PSLLW`
`_mm_sll_epi16`	Left	Logical	`PSLLW`
`_mm_slli_epi32`	Left	Logical	`PSLLD`
`_mm_sll_epi32`	Left	Logical	`PSLLD`
`_mm_slli_epi64`	Left	Logical	`PSLLQ`
`_mm_sll_epi64`	Left	Logical	`PSLLQ`
`_mm_srai_epi16`	Right	Arithmetic	`PSRAW`
`_mm_sra_epi16`	Right	Arithmetic	`PSRAW`
`_mm_srai_epi32`	Right	Arithmetic	`PSRAD`
`_mm_sra_epi32`	Right	Arithmetic	`PSRAD`
`_mm_srli_si128`	Right	Logical	`PSRLDQ`
`_mm_srli_epi16`	Right	Logical	`PSRLW`
`_mm_srl_epi16`	Right	Logical	`PSRLW`
`_mm_srli_epi32`	Right	Logical	`PSRLD`
`_mm_srl_epi32`	Right	Logical	`PSRLD`
`_mm_srli_epi64`	Right	Logical	`PSRLQ`
`_mm_srl_epi64`	Right	Logical	`PSRLQ`

```
__m128i _mm_slli_si128(__m128i a, int imm)
```

Shifts the 128-bit value in a left by `imm` bytes while shifting in zeros. `imm` must be an immediate.

```
r := a << (imm * 8)
```

```
__m128i _mm_slli_epi16(__m128i a, int count)
```

Shifts the 8 signed or unsigned 16-bit integers in a left by count bits while shifting in zeros.

```
r0 := a0 << count
r1 := a1 << count
...
r7 := a7 << count
```

`__m128i _mm_sll_epi16(__m128i a, __m128i count)`

Shifts the 8 signed or unsigned 16-bit integers in a left by `count` bits while shifting in zeros.

```
r0 := a0 << count
r1 := a1 << count
...
r7 := a7 << count
```

`__m128i _mm_slli_epi32(__m128i a, int count)`

Shifts the 4 signed or unsigned 32-bit integers in a left by `count` bits while shifting in zeros.

```
r0 := a0 << count
r1 := a1 << count
r2 := a2 << count
r3 := a3 << count
```

`__m128i _mm_sll_epi32(__m128i a, __m128i count)`

Shifts the 4 signed or unsigned 32-bit integers in a left by `count` bits while shifting in zeros.

```
r0 := a0 << count
r1 := a1 << count
r2 := a2 << count
r3 := a3 << count
```

`__m128i _mm_slli_epi64(__m128i a, int count)`

Shifts the 2 signed or unsigned 64-bit integers in a left by `count` bits while shifting in zeros.

```
r0 := a0 << count
r1 := a1 << count
```

`__m128i _mm_sll_epi64(__m128i a, __m128i count)`

Shifts the 2 signed or unsigned 64-bit integers in a left by `count` bits while shifting in zeros.

```
r0 := a0 << count
r1 := a1 << count
```

`__m128i _mm_srai_epi16(__m128i a, int count)`

Shifts the 8 signed 16-bit integers in a right by `count` bits while shifting in the sign bit.

```
r0 := a0 >> count
r1 := a1 >> count
...
r7 := a7 >> count
```

`__m128i _mm_sra_epi16(__m128i a, __m128i count)`

Shifts the 8 signed 16-bit integers in a right by `count` bits while shifting in the sign bit.

```
r0 := a0 >> count
r1 := a1 >> count
...
r7 := a7 >> count
```

`__m128i _mm_srai_epi32(__m128i a, int count)`

Shifts the 4 signed 32-bit integers in a right by `count` bits while shifting in the sign bit.

```
r0 := a0 >> count
r1 := a1 >> count
r2 := a2 >> count
r3 := a3 >> count
```

`__m128i _mm_sra_epi32(__m128i a, __m128i count)`

Shifts the 4 signed 32-bit integers in a right by `count` bits while shifting in the sign bit.

```
r0 := a0 >> count
r1 := a1 >> count
r2 := a2 >> count
r3 := i3 >> count
```

`__m128i _mm_srli_si128(__m128i a, int imm)`

Shifts the 128-bit value in a right by `imm` bytes while shifting in zeros. `imm` must be an immediate.

```
r := srl(a, imm*8)
```

`__m128i _mm_srli_epi16(__m128i a, int count)`

Shifts the 8 signed or unsigned 16-bit integers in a right by `count` bits while shifting in zeros.

```
r0 := srl(a0, count)
r1 := srl(a1, count)
...
r7 := srl(a7, count)
```

`__m128i _mm_srl_epi16(__m128i a, __m128i count)`

Shifts the 8 signed or unsigned 16-bit integers in a right by `count` bits while shifting in zeros.

```
r0 := srl(a0, count)
r1 := srl(a1, count)
...
r7 := srl(a7, count)
```

`__m128i _mm_srli_epi32(__m128i a, int count)`

Shifts the 4 signed or unsigned 32-bit integers in a right by `count` bits while shifting in zeros.

```
r0 := srl(a0, count)
r1 := srl(a1, count)
r2 := srl(a2, count)
r3 := srl(a3, count)
```

```
__m128i _mm_srl_epi32(__m128i a, __m128i count)
```

Shifts the 4 signed or unsigned 32-bit integers in a right by `count` bits while shifting in zeros.

```
r0 := srl(a0, count)
r1 := srl(a1, count)
r2 := srl(a2, count)
r3 := srl(a3, count)
```

```
__m128i _mm_srli_epi64(__m128i a, int count)
```

Shifts the 2 signed or unsigned 64-bit integers in a right by `count` bits while shifting in zeros.

```
r0 := srl(a0, count)
r1 := srl(a1, count)
```

```
__m128i _mm_srl_epi64(__m128i a, __m128i count)
```

Shifts the 2 signed or unsigned 64-bit integers in a right by `count` bits while shifting in zeros.

```
r0 := srl(a0, count)
r1 := srl(a1, count)
```

Integer Comparison Operations for Streaming SIMD Extensions 2

The comparison intrinsics for Streaming SIMD Extensions 2 and descriptions for each are listed in the following table.

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

Intrinsic Name	Instruction	Comparison	Elements	Size of Elements
`_mm_cmpeq_epi8`	`PCMPEQB`	Equality	16	8
`_mm_cmpeq_epi16`	`PCMPEQW`	Equality	8	16
`_mm_cmpeq_epi32`	`PCMPEQD`	Equality	4	32
`_mm_cmpgt_epi8`	`PCMPGTB`	Greater Than	16	8
`_mm_cmpgt_epi16`	`PCMPGTW`	Greater Than	8	16
`_mm_cmpgt_epi32`	`PCMPGTD`	Greater Than	4	32
`_mm_cmplt_epi8`	`PCMPGTBr`	Less Than	16	8
`_mm_cmplt_epi16`	`PCMPGTWr`	Less Than	8	16
`_mm_cmplt_epi32`	`PCMPGTDr`	Less Than	4	32

```
__m128i _mm_cmpeq_epi8(__m128i a, __m128i b)
```

Compares the 16 signed or unsigned 8-bit integers in a and the 16 signed or unsigned 8-bit integers in b for equality.

```
r0 := (a0 == b0) ? 0xff : 0x0
r1 := (a1 == b1) ? 0xff : 0x0
...
r15 := (a15 == b15) ? 0xff : 0x0
```

```
__m128i _mm_cmpeq_epi16(__m128i a, __m128i b)
```

Compares the 8 signed or unsigned 16-bit integers in a and the 8 signed or unsigned 16-bit integers in b for equality.

```
r0 := (a0 == b0) ? 0xffff : 0x0
r1 := (a1 == b1) ? 0xffff : 0x0
...
r7 := (a7 == b7) ? 0xffff : 0x0
```

```
__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)
```

Compares the 4 signed or unsigned 32-bit integers in a and the 4 signed or unsigned 32-bit integers in b for equality.

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

`__m128i _mm_cmpgt_epi8(__m128i a, __m128i b)`

Compares the 16 signed 8-bit integers in a and the 16 signed 8-bit integers in b for greater than.

```
r0 := (a0 > b0) ? 0xff : 0x0
r1 := (a1 > b1) ? 0xff : 0x0
...
r15 := (a15 > b15) ? 0xff : 0x0
```

`__m128i _mm_cmpgt_epi16(__m128i a, __m128i b)`

Compares the 8 signed 16-bit integers in a and the 8 signed 16-bit integers in b for greater than.

```
r0 := (a0 > b0) ? 0xffff : 0x0
r1 := (a1 > b1) ? 0xffff : 0x0
...
r7 := (a7 > b7) ? 0xffff : 0x0
```

`__m128i _mm_cmpgt_epi32(__m128i a, __m128i b)`

Compares the 4 signed 32-bit integers in a and the 4 signed 32-bit integers in b for greater than.

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

`__m128i _mm_cmplt_epi8( __m128i a, __m128i b)`

Compares the 16 signed 8-bit integers in a and the 16 signed 8-bit integers in b for less than.

```
r0 := (a0 < b0) ? 0xff : 0x0
r1 := (a1 < b1) ? 0xff : 0x0
...
r15 := (a15 < b15) ? 0xff : 0x0
```

`__m128i _mm_cmplt_epi16( __m128i a, __m128i b)`

Compares the 8 signed 16-bit integers in a and the 8 signed 16-bit integers in b for less than.

```
r0 := (a0 < b0) ? 0xffff : 0x0
r1 := (a1 < b1) ? 0xffff : 0x0
...
r7 := (a7 < b7) ? 0xffff : 0x0
```

`__m128i _mm_cmplt_epi32( __m128i a, __m128i b)`

Compares the 4 signed 32-bit integers in a and the 4 signed 32-bit integers in b for less than.

```
r0 := (a0 < b0) ? 0xffff : 0x0
```

```
r1 := (a1 < b1) ? 0xffff : 0x0
r2 := (a2 < b2) ? 0xffff : 0x0
r3 := (a3 < b3) ? 0xffff : 0x0
```

Conversion Operations for Streaming SIMD Extensions 2

The following two conversion intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2.

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

`__m128i _mm_cvtsi32_si128(int a)`

(uses `MOVD`) Moves 32-bit integer a to the least significant 32 bits of an `__m128i` object. Copies the sign bit of a into the upper 96 bits of the `__m128i` object.

```
r0 := a
r1 := 0x0 ; r2 := 0x0 ; r3 := 0x0
```

`int _mm_cvtsi128_si32(__m128i a)`

(uses `MOVD`) Moves the least significant 32 bits of a to a 32 bit integer.

```
r := a0
```

`__m128 _mm_cvtepi32_ps(__m128i a)`

Converts the 4 signed 32-bit integer values of a to SP FP values.

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

`__m128i _mm_cvtps_epi32(__m128 a)`

Converts the 4 SP FP values of a to signed 32-bit integer values.

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

`__m128i _mm_cvttps_epi32(__m128 a)`

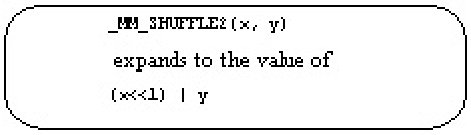
Converts the 4 SP FP values of a to signed 32 bit integer values using truncate.

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

Macro Function for Shuffle

The Streaming SIMD Extensions 2 provide a macro function to help create constants that describe shuffle operations. The macro takes two small integers (in the range of 0 to 1) and combines them into an 2-bit immediate value used by the `SHUFPD` instruction. See the following example.

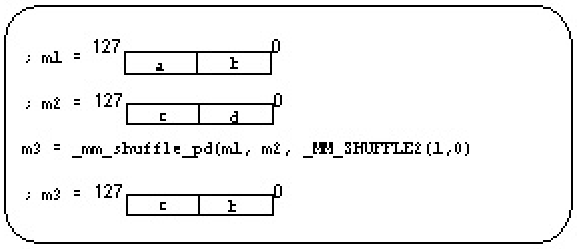
Shuffle Function Macro



```
_MM_SHUFFLE2(x, y)
expands to the value of
(x<<1) | y
```

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

View of Original and Result Words with Shuffle Function Macro



```
m3 = _mm_shuffle_pd(m1, m2, _MM_SHUFFLE2(1,0))
```

Cacheability Support Operations for Streaming SIMD Extensions 2

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

`void _mm_stream_pd(double *p, __m128d a)`

> (uses `MOVNTPD`) Stores the data in a to the address p without polluting caches. The address p must be 16-byte aligned. If the cache line containing address p is already in the cache, the cache will be updated.
>
> ```
> p[0] := a0
> p[1] := a1
> ```

`void _mm_stream_si128(__m128i *p, __m128i a)`

> Stores the data in a to the address p without polluting the caches. If the cache line containing address p is already in the cache, the cache will be updated. Address p must be 16-byte aligned.
>
> ```
> *p := a
> ```

`void _mm_stream_si32(int *p, int a)`

> Stores the data in a to the address p without polluting the caches. If the cache line containing address p is already in the cache, the cache will be updated.
>
> ```
> *p := a
> ```

`void _mm_clflush(void const*p)`

> Cache line containing p is flushed and invalidated from all caches in the coherency domain.

`void _mm_lfence(void)`

> Guarantees that every load instruction that precedes, in program order, the load fence instruction is globally visible before any load instruction which follows the fence in program order.

`void _mm_mfence(void)`

> Guarantees that every memory access that precedes, in program order, the memory fence instruction is globally visible before any memory instruction which follows the fence in program order.

`void _mm_pause(void)`

> The execution of the next instruction is delayed an implementation specific amount of time. The instruction does not modify the architectural state. This intrinsic provides especially significant performance gain and described in more detail below.

PAUSE Intrinsic

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

Example of loop with the `PAUSE` instruction:

```
spin_loop:pause

cmp eax, A

jne spin_loop
```

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

```
get_lock: mov eax, 1

xchg eax, A ; Try to get lock

cmp eax, 0 ; Test if successful

jne spin_loop
```

<critical_section code>

```
mov A, 0 ; Release lock

jmp continue

spin_loop: pause ; Spin-loop hint

cmp 0, A ; Check lock availability

jne spin_loop

jmp get_lock
```

`continue:` <other code>

Note that the first branch is predicted to fall-through to the critical section in anticipation of successfully gaining access to the lock. It is highly recommended that all spin-wait loops include the `PAUSE` instruction. Since `PAUSE` is backwards compatible to all existing IA-32 processor generations, a test for processor type (a `CPUID` test) is not needed. All legacy processors will execute `PAUSE` as a `NOP`, but in processors which use the `PAUSE` as a hint there can be significant performance benefit.

Miscellaneous Operations for Streaming SIMD Extensions 2

The miscellaneous intrinsics for Streaming SIMD Extensions 2 are listed in the following table followed by their descriptions.

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

Intrinsic	**Corresponding Instruction**	**Operation**
`_mm_packs_epi16`	`PACKSSWB`	Packed Saturation
`_mm_packs_epi32`	`PACKSSDW`	Packed Saturation
`_mm_packus_epi16`	`PACKUSWB`	Packed Saturation
`_mm_extract_epi16`	`PEXTRW`	Extraction
`_mm_insert_epi16`	`PINSRW`	Insertion
`_mm_movemask_epi8`	`PMOVMSKB`	Mask Creation
`_mm_shuffle_epi32`	`PSHUFD`	Shuffle
`_mm_shufflehi_epi16`	`PSHUFHW`	Shuffle
`_mm_shufflelo_epi16`	`PSHUFLW`	Shuffle
`_mm_unpackhi_epi8`	`PUNPCKHBW`	Interleave
`_mm_unpackhi_epi16`	`PUNPCKHWD`	Interleave
`_mm_unpackhi_epi32`	`PUNPCKHDQ`	Interleave
`_mm_unpackhi_epi64`	`PUNPCKHQDQ`	Interleave
`_mm_unpacklo_epi8`	`PUNPCKLBW`	Interleave
`_mm_unpacklo_epi16`	`PUNPCKLWD`	Interleave
`_mm_unpacklo_epi32`	`PUNPCKLDQ`	Interleave
`_mm_unpacklo_epi64`	`PUNPCKLQDQ`	Interleave
`_mm_movepi64_pi64`	`MOVDQ2Q`	move
`_m128i_mm_movpi64_epi64`	`MOVQ2DQ`	move
`_mm_move_epi64`	`MOVQ`	move

```
__m128i _mm_packs_epi16(__m128i a, __m128i b)
```

Packs the 16 signed 16-bit integers from `a` and `b` into 8-bit integers and saturates.

```
r0 := SignedSaturate(a0)
r1 := SignedSaturate(a1)
...
r7 := SignedSaturate(a7)
r8 := SignedSaturate(b0)
r9 := SignedSaturate(b1)
...
r15 := SignedSaturate(b7)
```

`__m128i _mm_packs_epi32(__m128i a, __m128i b)`

Packs the 8 signed 32-bit integers from a and b into signed 16-bit integers and saturates.

```
r0 := SignedSaturate(a0)
r1 := SignedSaturate(a1)
r2 := SignedSaturate(a2)
r3 := SignedSaturate(a3)
r4 := SignedSaturate(b0)
r5 := SignedSaturate(b1)
r6 := SignedSaturate(b2)
r7 := SignedSaturate(b3)
```

`__m128i _mm_packus_epi16(__m128i a, __m128i b)`

Packs the 16 signed 16-bit integers from a and b into 8-bit unsigned integers and saturates.

```
r0 := UnsignedSaturate(a0)
r1 := UnsignedSaturate(a1)
...
r7 := UnsignedSaturate(a7)
r8 := UnsignedSaturate(b0)
r9 := UnsignedSaturate(b1)
...
r15 := UnsignedSaturate(b7)
```

`int _mm_extract_epi16(__m128i a, int imm)`

Extracts the selected signed or unsigned 16-bit integer from a and zero extends. The selector `imm` must be an immediate.

```
r := (imm == 0) ? a0 :
( (imm == 1) ? a1 :
...
(imm == 7) ? a7 )
```

`__m128i _mm_insert_epi16(__m128i a, int b, int imm)`

Inserts the least significant 16 bits of b into the selected 16-bit integer of a. The selector `imm` must be an immediate.

```
r0 := (imm == 0) ? b : a0;
r1 := (imm == 1) ? b : a1;
...
r7 := (imm == 7) ? b : a7;
```

`int _mm_movemask_epi8(__m128i a)`

Creates a 16-bit mask from the most significant bits of the 16 signed or unsigned 8-bit integers in a and zero extends the upper bits.

```
r := a15[7] << 15 |
a14[7] << 14 |
...
a1[7] << 1 |
a0[7]
```

`__m128i _mm_shuffle_epi32(__m128i a, int imm)`

Shuffles the 4 signed or unsigned 32-bit integers in a as specified by `imm`. The shuffle value, `imm`, must be an immediate. See Macro Function for Shuffle for a description of shuffle semantics.

`__m128i _mm_shufflehi_epi16(__m128i a, int imm)`

Shuffles the upper 4 signed or unsigned 16-bit integers in a as specified by `imm`. The shuffle value, `imm`, must be an immediate. See Macro Function for Shuffle for a description of shuffle semantics.

`__m128i _mm_shufflelo_epi16(__m128i a, int imm)`

Shuffles the lower 4 signed or unsigned 16-bit integers in a as specified by `imm`. The shuffle value, `imm`, must be an immediate. See Macro Function for Shuffle for a description of shuffle semantics.

`__m128i _mm_unpackhi_epi8(__m128i a, __m128i b)`

Interleaves the upper 8 signed or unsigned 8-bit integers in a with the upper 8 signed or unsigned 8-bit integers in b.

```
r0 := a8 ; r1 := b8
r2 := a9 ; r3 := b9
...
r14 := a15 ; r15 := b15
```

`__m128i _mm_unpackhi_epi16(__m128i a, __m128i b)`

Interleaves the upper 4 signed or unsigned 16-bit integers in a with the upper 4 signed or unsigned 16-bit integers in b.

```
r0 := a4 ; r1 := b4
r2 := a5 ; r3 := b5
r4 := a6 ; r5 := b6
r6 := a7 ; r7 := b7
```

`__m128i _mm_unpackhi_epi32(__m128i a, __m128i b)`

Interleaves the upper 2 signed or unsigned 32-bit integers in a with the upper 2 signed or unsigned 32-bit integers in b.

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

```
__m128i _mm_unpackhi_epi64(__m128i a, __m128i b)
```

Interleaves the upper signed or unsigned 64-bit integer in a with the upper signed or unsigned 64-bit integer in b.

```
r0 := a1 ; r1 := b1
```

```
__m128i _mm_unpacklo_epi8(__m128i a, __m128i b)
```

Interleaves the lower 8 signed or unsigned 8-bit integers in a with the lower 8 signed or unsigned 8-bit integers in b.

```
r0 := a0 ; r1 := b0
r2 := a1 ; r3 := b1
...
r14 := a7 ; r15 := b7
```

```
__m128i _mm_unpacklo_epi16(__m128i a, __m128i b)
```

Interleaves the lower 4 signed or unsigned 16-bit integers in a with the lower 4 signed or unsigned 16-bit integers in b.

```
r0 := a0 ; r1 := b0
r2 := a1 ; r3 := b1
r4 := a2 ; r5 := b2
r6 := a3 ; r7 := b3
```

```
__m128i _mm_unpacklo_epi32(__m128i a, __m128i b)
```

Interleaves the lower 2 signed or unsigned 32-bit integers in a with the lower 2 signed or unsigned 32-bit integers in b.

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

```
__m128i _mm_unpacklo_epi64(__m128i a, __m128i b)
```

Interleaves the lower signed or unsigned 64-bit integer in a with the lower signed or unsigned 64-bit integer in b.

```
r0 := a0 ; r1 := b0
```

```
__m64 _mm_movepi64_pi64(__m128i a)
```

Returns the lower 64 bits of a as an `__m64` type.

```
r0 := a0 ;
```

```
__128i _mm_movpi64_pi64(__m64 a)
```

Moves the 64 bits of a to the lower 64 bits of the result, zeroing the upper bits.

```
r0 := a0 ; r1 := 0X0 ;
```

```
__128i _mm_move_epi64(__128i a)
```

Moves the lower 64 bits of the lower 64 bits of the result, zeroing the upper bits.

```
r0 := a0 ; r1 := 0X0 ;
```

Streaming SIMD Extensions 2 Integer Memory and Initialization

The integer `load`, `set`, and `store` intrinsics and their respective instructions provide memory and initialization operations for the Streaming SIMD Extensions 2.

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

- Load Operations
- Set Operations
- Store Operations

Integer Load Operations for Streaming SIMD Extensions 2

The following `load` operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2.

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

`__m128i _mm_load_si128(__m128i const*p)`

(uses `MOVDQA`) Loads 128-bit value. Address p must be 16-byte aligned.

```
r := *p
```

`__m128i _mm_loadu_si128(__m128i const*p)`

(uses `MOVDQU`) Loads 128-bit value. Address p not need be 16-byte aligned.

```
r := *p
```

`__m128i _mm_loadl_epi64(__m128i const*p)`

(uses `MOVQ`) Load the lower 64 bits of the value pointed to by p into the lower 64 bits of the result, zeroing the upper 64 bits of the result.

```
r0:= *p[63:0]
r1:=0x0
```

Integer Set Operations for Streaming SIMD Extensions 2

The following `set` operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2.

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

```
__m128i _mm_set_epi64(__m64 q1, __m64 q0)
```

Sets the 2 64-bit integer values.

```
r0 := q0
r1 := q1
```

```
__m128i _mm_set_epi32(int i3, int i2, int i1, int i0)
```

Sets the 4 signed 32-bit integer values.

```
r0 := i0
r1 := i1
r2 := i2
r3 := i3
```

```
__m128i _mm_set_epi16(short w7, short w6, short w5, short w4, short w3,
short w2, short w1, short w0)
```

Sets the 8 signed 16-bit integer values.

```
r0 := w0
r1 := w1
...
r7 := w7
```

```
__m128i _mm_set_epi8(char b15, char b14, char b13, char b12, char b11, char
b10, char b9, char b8, char b7, char b6, char b5, char b4, char b3, char
b2, char b1, char b0)
```

Sets the 16 signed 8-bit integer values.

```
r0 := b0
r1 := b1
...
r15 := b15
```

```
__m128i _mm_set1_epi64(__m64 q)
```

Sets the 2 64-bit integer values to q.

```
r0 := q
r1 := q
```

```
__m128i _mm_set1_epi32(int i)
```

Sets the 4 signed 32-bit integer values to `i`.

```
r0 := i
r1 := i
r2 := i
r3 := i
```

```
__m128i _mm_set1_epi16(short w)
```

Sets the 8 signed 16-bit integer values to `w`.

```
r0 := w
r1 := w
...
r7 := w
```

```
__m128i _mm_set1_epi8(char b)
```

Sets the 16 signed 8-bit integer values to `b`.

```
r0 := b
r1 := b
...
r15 := b
```

```
__m128i _mm_setr_epi64(__m64 q0, __m64 q1)
```

Sets the 2 64-bit integer values in reverse order.

```
r0 := q0
r1 := q1
```

```
__m128i _mm_setr_epi32(int i0, int i1, int i2, int i3)
```

Sets the 4 signed 32-bit integer values in reverse order.

```
r0 := i0
r1 := i1
r2 := i2
r3 := i3
```

```
__m128i _mm_setr_epi16(short w0, short w1, short w2, short w3,  short w4,
short w5, short w6, short w7)
```

Sets the 8 signed 16-bit integer values in reverse order.

```
r0 := w0
r1 := w1
...
r7 := w7
```

```
__m128i _mm_setr_epi8(char b15, char b14, char b13, char b12,  char b11,
char b10, char b9, char b8, char b7, char b6, char b5, char b4, char b3,
char b2, char b1, char b0)
```

Sets the 16 signed 8-bit integer values in reverse order.

```
r0 := b0
r1 := b1
...
r15 := b15
```

```
__m128i _mm_setzero_si128()
```

Sets the 128-bit value to zero.

```
r := 0x0
```

Integer Store Operations for Streaming SIMD Extensions 2

The following `store` operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2.

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

```
void _mm_store_si128(__m128i *p, __m128i b)
```

(uses `MOVDQA`) Stores 128-bit value. Address p must be 16 byte aligned.

```
*p := a
```

```
void _mm_storeu_si128(__m128i *p, __m128i b)
```

(uses `MOVDQU`) Stores 128-bit value. Address p need not be 16-byte aligned.

```
*p := a
```

```
void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p)
```

(uses `MASKMOVDQU`) Conditionally store byte elements of d to address p. The high bit of each byte in the selector n determines whether the corresponding byte in d will be stored. Address p need not be 16-byte aligned.

```
if (n0[7]) p[0] := d0
if (n1[7]) p[1] := d1
...
if (n15[7]) p[15] := d15
```

```
void _mm_storel_epi64(__m128i *p, __m128i q)
```

(uses `MOVQ`) Stores the lower 64 bits of the value pointed to by p.

```
*p[63:0]:=a0
```

Overview: Intrinsics for Itanium® Instructions

This section lists and describes the native intrinsics for Itanium® instructions. These intrinsics cannot be used on the IA-32 architecture. The intrinsics for Itanium instructions give programmers access to Itanium instructions that cannot be generated using the standard constructs of the C and C++ languages.

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Note

The Intel® C++ Compiler for Itanium-base applications provides intrinsic functions that provide equivalent functionality as inline assembly without inhibiting compiler optimizations and affecting instruction scheduling.

Native Intrinsics for Itanium® Instructions

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Integer Operations

Intrinsic	Corresponding Instruction
`__int64 _m64_dep_mr(__int64 r, __int64 s, const int pos, const int len)`	`dep` (Deposit)
`__int64 _m64_dep_mi(const int v, __int64 s, const int p, const int len)`	`dep` (Deposit)
`__int64 _m64_dep_zr(__int64 s, const int pos, const int len)`	`dep.z` (Deposit)
`__int64 _m64_dep_zi(const int v, const int pos, const int len)`	`dep.z` (Deposit)
`__int64 _m64_extr(__int64 r, const int pos, const int len)`	`extr` (Extract)
`__int64 _m64_extru(__int64 r, const int pos, const int len)`	`extr.u` (Extract)
`__int64 _m64_xmal(__int64 a, __int64 b, __int64 c)`	`xma.l` (Fixed-point multiply add using the low 64 bits of the 128-bit result. The result is signed.)
`__int64 _m64_xmalu(__int64 a, __int64 b, __int64 c)`	`xma.lu` (Fixed-point multiply add using the low 64 bits of the 128-bit result. The result is unsigned.)
`__int64 _m64_xmah(__int64 a, __int64 b, __int64 c)`	`xma.h` (Fixed-point multiply add using the high 64 bits of the 128-bit result. The result is signed.)
`__int64 _m64_xmahu(__int64 a, __int64 b, __int64 c)`	`xma.hu` (Fixed-point multiply add using the high 64 bits of the 128-bit result. The result is unsigned.)
`__int64 _m64_popcnt(__int64 a)`	`popcnt` (Population count)
`__int64 _m64_shladd(__int64 a, const int count, __int64 b)`	`shladd` (Shift left and add)
`__int64 _m64_shrp(__int64 a, __int64 b, const int count)`	`shrp` (Shift right pair)

FSR Operations

Intrinsic	Description
`void _fsetc(int amask, int omask)`	Sets the control bits of `FPSR.sf0`. Maps to the `fsetc.sf0 r, r` instruction. There is no corresponding instruction to read the control bits. Use `_mm_getfpsr()`.
`void _fclrf(void)`	Clears the floating point status flags (the 6-bit flags of `FPSR.sf0`). Maps to the `fclrf.sf0` instruction.

`__int64 _m64_dep_mr(__int64 r, __int64 s, const int pos, const int len)`

The right-justified 64-bit value `r` is deposited into the value in `s` at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `pos` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

`__int64 _m64_dep_mi(const int v, __int64 s, const int p, const int len)`

The sign-extended value `v` (either all 1s or all 0s) is deposited into the value in `s` at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `p` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

`__int64 _m64_dep_zr(__int64 s, const int pos, const int len)`

The right-justified 64-bit value `s` is deposited into a 64-bit field of all zeros at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `pos` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

`__int64 _m64_dep_zi(const int v, const int pos, const int len)`

The sign-extended value `v` (either all 1s or all 0s) is deposited into a 64-bit field of all zeros at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `pos` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

`__int64 _m64_extr(__int64 r, const int pos, const int len)`

A field is extracted from the 64-bit value `r` and is returned right-justified and sign extended. The extracted field begins at position `pos` and extends `len` bits to the left. The sign is taken from the most significant bit of the extracted field.

`__int64 _m64_extru(__int64 r, const int pos, const int len)`

A field is extracted from the 64-bit value `r` and is returned right-justified and zero extended. The extracted field begins at position `pos` and extends `len` bits to the left.

`__int64 _m64_xmal(__int64 a, __int64 b, __int64 c)`

The 64-bit values `a` and `b` are treated as signed integers and multiplied to produce a full 128-bit signed result. The 64-bit value `c` is zero-extended and added to the product. The least significant 64 bits of the sum are then returned.

`__int64 _m64_xmalu(__int64 a, __int64 b, __int64 c)`

The 64-bit values a and b are treated as signed integers and multiplied to produce a full 128-bit unsigned result. The 64-bit value c is zero-extended and added to the product. The least significant 64 bits of the sum are then returned.

```
__int64 _m64_xmah(__int64 a, __int64 b, __int64 c)
```

The 64-bit values a and b are treated as signed integers and multiplied to produce a full 128-bit signed result. The 64-bit value c is zero-extended and added to the product. The most significant 64 bits of the sum are then returned.

```
__int64 _m64_xmahu(__int64 a, __int64 b, __int64 c)
```

The 64-bit values a and b are treated as unsigned integers and multiplied to produce a full 128-bit unsigned result. The 64-bit value c is zero-extended and added to the product. The most significant 64 bits of the sum are then returned.

```
__int64 _m64_popcnt(__int64 a)
```

The number of bits in the 64-bit integer a that have the value 1 are counted, and the resulting sum is returned.

```
__int64 _m64_shladd(__int64 a, const int count, __int64 b)
```

a is shifted to the left by `count` bits and then added to b. The result is returned.

```
__int64 _m64_shrp(__int64 a, __int64 b, const int count)
```

a and b are concatenated to form a 128-bit value and shifted to the right `count` bits. The least significant 64 bits of the result are returned.

Lock and Atomic Operation Related Intrinsics

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Intrinsic	Description
`unsigned __int64 _InterlockedExchange8 (volatile unsigned char *Target, unsigned __int64 value)`	Map to the `xchg1` instruction. Atomically write the least significant byte of its 2nd argument to address specified by its 1st argument.
`unsigned __int64 _InterlockedCompareExchange8_rel(volatile unsigned char *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Compare and exchange atomically the least significant byte at the address specified by its 1st argument. Maps to the `cmpxchg1.rel` instruction with appropriate setup.
`unsigned __int64 _InterlockedCompareExchange8_acq(volatile unsigned char *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Same as above, but using `acquire` semantic.
`unsigned __int64 _InterlockedExchange16 (volatile unsigned short *Target, unsigned __int64 value)`	Map to the `xchg2` instruction. Atomically write the least significant word of its 2nd argument to address specified by its 1st argument.
`unsigned __int64 _InterlockedCompareExchange16_rel(volatile unsigned short *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Compare and exchange atomically the least significant word at the address specified by its 1st argument. Maps to the `cmpxchg2.rel` instruction with appropriate setup.
`unsigned __int64 _InterlockedCompareExchange16_acq(volatile unsigned short *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Same as above, but using `acquire` semantic.
`int _InterlockedIncrement(volatile int *addend)`	Atomically increment by one the value specified by its argument. Maps to the `fetchadd4` instruction.
`int _InterlockedDecrement(volatile int *addend)`	Atomically decrement by one the value specified by its argument. Maps to the `fetchadd4` instruction.
`int _InterlockedExchange(volatile int *Target, int value)`	Do an exchange operation atomically. Maps to the `xchg4` instruction.
`int _InterlockedCompareExchange(volatile int *Destination, int Exchange, int Comparand)`	Maps to the `cmpxchg4` instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 32-bit pointer).

`int _InterlockedExchangeAdd(volatile int *addend, int increment)`	Use compare and exchange to do an atomic add of the increment value to the addend. Maps to a loop with the `cmpxchg4` instruction to guarantee atomicity.
`int _InterlockedAdd(volatile int *addend, int increment)`	Same as above; but returns new value, not the original one.
`void * _InterlockedCompareExchangePointer (void * volatile *Destination, void *Exchange, void *Comparand)`	Map the `exch8` instruction; Atomically compare and exchange the pointer value specified by its first argument (all arguments are pointers)
`unsigned __int64 _InterlockedExchangeU (volatile unsigned int *Target, unsigned __int64 value)`	Atomically exchange the 32-bit quantity specified by the 1st argument. Maps to the `xchg4` instruction.
`unsigned __int64 _InterlockedCompareExchange_rel(volatile unsigned int *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Maps to the `cmpxchg4.rel` instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 64-bit pointer).
`unsigned __int64 _InterlockedCompareExchange_acq(volatile unsigned int *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Same as above; but map the `cmpxchg4.acq` instruction.
`void _ReleaseSpinLock(volatile int *x)`	Release spin lock.
`__int64 _InterlockedIncrement64(volatile __int64 *addend)`	Increment by one the value specified by its argument. Maps to the `fetchadd` instruction.
`__int64 _InterlockedDecrement64(volatile __int64 *addend)`	Decrement by one the value specified by its argument. Maps to the `fetchadd` instruction.
`__int64 _InterlockedExchange64(volatile __int64 *Target, __int64 value)`	Do an exchange operation atomically. Maps to the `xchg` instruction.
`unsigned __int64 _InterlockedExchangeU64 (volatile unsigned __int64 *Target, unsigned __int64 value)`	Same as `InterlockedExchange64` (for unsigned quantities).
`unsigned __int64 _InterlockedCompareExchange64_rel(volatile unsigned __int64 *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Maps to the `cmpxchg.rel` instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 64-bit pointer).
`unsigned __int64 _InterlockedCompareExchange64_acq(volatile unsigned __int64 *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Maps to the `cmpxchg.acq` instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 64-bit pointer).

`__int64 _InterlockedCompareExchange64 (volatile __int64 *Destination, __int64 Exchange, __int64 Comparand)`	Same as above for signed quantities.
`__int64 _InterlockedExchangeAdd64(volatile __int64 *addend, __int64 increment)`	Use compare and exchange to do an atomic add of the increment value to the addend. Maps to a loop with the `cmpxchg` instruction to guarantee atomicity
`__int64 _InterlockedAdd64(volatile __int64 *addend, __int64 increment);`	Same as above. Returns the new value, not the original value. See **Note** below.

Note

`_InterlockedSub64` is provided as a macro definition based on `_InterlockedAdd64`.
`#define _InterlockedSub64(target, incr) _InterlockedAdd64((target),( -(incr)))`.

Uses `cmpxchg` to do an atomic sub of the `incr` value to the `target`. Maps to a loop with the `cmpxchg` instruction to guarantee atomicity.

Load and Store

You can use the `load` and `store` intrinsic to force the strict memory access ordering of specific data objects. This intended use is for the case when the user suppresses the strict memory access ordering by using the `–mno-serialize-volatile` option.

Intrinsic	Prototype	Description
`__st1_rel`	`void __st1_rel(void *dst, const char value);`	Generates an `st1.rel` instruction.
`__st2_rel`	`void __st2_rel(void *dst, const short value);`	Generates an `st2.rel` instruction.
`__st4_rel`	`void __st4_rel(void *dst, const int value);`	Generates an `st4.rel` instruction.
`__st8_rel`	`void   st8 rel(void *dst, const   int64 value);`	Generates an `st8.rel` instruction.
`__ld1_acq`	`unsigned char __ld1_acq(void *src);`	Generates an `ld1.acq` instruction.
`__ld2_acq`	`unsigned short __ld2_acq(void *src);`	Generates an `ld2.acq` instruction.
`__ld4_acq`	`unsigned int __ld4_acq(void *src);`	Generates an `ld4.acq` instruction.
`__ld8_acq`	`unsigned __int64 __ld8_acq(void *src);`	Generates an `ld8.acq` instruction.

Operating System Related Intrinsics for Itanium®-based Systems

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Intrinsic	Description
`unsigned __int64 __getReg (const int whichReg)`	Gets the value from a hardware register based on the index passed in. Produces a corresponding `mov = r` instruction. Provides access to the following registers: See Register Names for getReg() and setReg().
`void __setReg(const int whichReg, unsigned __int64 value)`	Sets the value for a hardware register based on the index passed in. Produces a corresponding `mov = r` instruction. See Register Names for getReg() and setReg().
`unsigned __int64 __getIndReg(const int whichIndReg, __int64 index)`	Return the value of an indexed register. The index is the 2nd argument; the register file is the first argument.
`void __setIndReg(const int whichIndReg, __int64 index, unsigned __int64 value)`	Copy a value in an indexed register. The index is the 2nd argument; the register file is the first argument.
`void *_rdteb(void)`	Gets TEB address. The TEB address is kept in `r13` and maps to the move `r=tp` instruction
`void __isrlz(void)`	Executes the serialize instruction. Maps to the `srlz.i` instruction.
`void __dsrlz(void)`	Serializes the data. Maps to the `srlz.d` instruction.
`unsigned __int64 __fetchadd4_acq(unsigned int *addend, const int increment)`	Map the `fetchadd4.acq` instruction.
`unsigned __int64 __fetchadd4_rel(unsigned int *addend, const int increment)`	Map the `fetchadd4.rel` instruction.
`unsigned __int64 __fetchadd8_acq(unsigned __int64 *addend, const int increment)`	Map the `fetchadd8.acq` instruction.
`unsigned __int64 __fetchadd8_rel(unsigned __int64 *addend, const int increment)`	Map the `fetchadd8.rel` instruction.
`void __fwb(void)`	Flushes the write buffers. Maps to the `fwb` instruction.
`void __ldfs(const int whichFloatReg, void *src)`	Map the `ldfs` instruction. Load a single precision value to the specified register.

`void __ldfd(const int whichFloatReg, void *src)`	Map the `ldfd` instruction. Load a double precision value to the specified register.
`void __ldfe(const int whichFloatReg, void *src)`	Map the `ldfe` instruction. Load an extended precision value to the specified register.
`void __ldf8(const int whichFloatReg, void *src)`	Map the `ldf8` instruction.
`void __ldf_fill(const int whichFloatReg, void *src)`	Map the `ldf.fill` instruction.
`void __stfs(void *dst, const int whichFloatReg)`	Map the `sfts` instruction.
`void __stfd(void *dst, const int whichFloatReg)`	Map the `stfd` instruction.
`void __stfe(void *dst, const int whichFloatReg)`	Map the `stfe` instruction.
`void __stf8(void *dst, const int whichFloatReg)`	Map the `stf8` instruction.
`void __stf_spill(void *dst, const int whichFloatReg)`	Map the `stf.spill` instruction.
`void __mf(void)`	Executes a memory fence instruction. Maps to the `mf` instruction.
`void __mfa(void)`	Executes a memory fence, acceptance form instruction. Maps to the `mf.a` instruction.
`void __synci(void)`	Enables memory synchronization. Maps to the `sync.i` instruction.
`void __thash(__int64)`	Generates a translation hash entry address. Maps to the `thash r = r` instruction.
`void __ttag(__int64)`	Generates a translation hash entry tag. Maps to the `ttag r=r` instruction.
`void __itcd(__int64 pa)`	Insert an entry into the data translation cache (Map `itc.d` instruction).
`void __itci(__int64 pa)`	Insert an entry into the instruction translation cache (Map `itc.i`).
`void __itrd(__int64 whichTransReg, __int64 pa)`	Map the `itr.d` instruction.
`void __itri(__int64 whichTransReg, __int64 pa)`	Map the `itr.i` instruction.
`void __ptce(__int64 va)`	Map the `ptc.e` instruction.
`void __ptcl(__int64 va, __int64 pagesz)`	Purges the local translation cache. Maps to the `ptc.l r, r` instruction.

`void __ptcg(__int64 va, __int64 pagesz)`	Purges the global translation cache. Maps to the `ptc.g r, r` instruction.
`void __ptcga(__int64 va, __int64 pagesz)`	Purges the global translation cache and ALAT. Maps to the `ptc.ga r, r` instruction.
`void __ptri(__int64 va, __int64 pagesz)`	Purges the translation register. Maps to the `ptr.i r, r` instruction.
`void __ptrd(__int64 va, __int64 pagesz)`	Purges the translation register. Maps to the `ptr.d r, r` instruction.
`__int64 __tpa(__int64 va)`	Map the `tpa` instruction.
`void __invalat(void)`	Invalidates ALAT. Maps to the `invala` instruction.
`void __invala (void)`	Same as `void __invalat(void)`
`void __invala_gr(const int whichGeneralReg)`	`whichGeneralReg` = 0-127
`void __invala_fr(const int whichFloatReg)`	`whichFloatReg` = 0-127
`void __break(const int)`	Generates a break instruction with an immediate.
`void __nop(const int)`	Generate a `nop` instruction.
`void __debugbreak(void)`	Generates a Debug Break Instruction fault.
`void __fc(__int64)`	Flushes a cache line associated with the address given by the argument. Maps to the `fcr` instruction.
`void __sum(int mask)`	Sets the user mask bits of PSR. Maps to the `sum imm24` instruction.
`void __rum(int mask)`	Resets the user mask.
`void __ssm(int mask)`	Sets the system mask.
`void __rsm(int mask)`	Resets the system mask bits of PSR. Maps to the `rsm imm24` instruction.
`__int64 _ReturnAddress (void)`	Get the caller's address.
`void __lfetch(int lfhint, void *y)`	Generate the `lfetch.lfhint` instruction. The value of the first argument specifies the hint type.
`void __lfetch_fault(int lfhint, void *y)`	Generate the `lfetch.fault.lfhint` instruction. The value of the first argument specifies the hint type.
`unsigned int __cacheSize (unsigned int cacheLevel)`	`__cacheSize(n)` returns the size in bytes of the cache at level n. 1 represents the first-level cache. 0 is returned for a non-existent cache level. For example, an application may query the cache size and use it to select block sizes in algorithms that operate on matrices.
`void __memory_barrier (void)`	Creates a barrier across which the compiler will not schedule any data access instruction. The compiler may allocate local data in registers across a memory barrier, but not global data.

Conversion Intrinsics Itanium®-based Systems

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Intrinsic	Description
`__int64 _m_to_int64(__m64 a)`	Convert a of type `__m64` to type `__int64`. Translates to nop since both types reside in the same register on Itanium-based systems.
`__m64 _m_from_int64(__int64 a)`	Convert a of type `__int64` to type `__m64`. Translates to nop since both types reside in the same register on Itanium-based systems.
`__int64 __round_double_to_int64 (double d)`	Convert its double precision argument to a signed integer.
`unsigned __int64 __getf_exp (double d)`	Map the `getf.exp` instruction and return the 16-bit exponent and the sign of its operand.

Register Names for getReg() and setReg()

The prototypes for getReg() and setReg() intrinsics are in the `ia64regs.h` header file.

Name	whichReg
`_IA64_REG_IP`	1016
`_IA64_REG_PSR`	1019
`_IA64_REG_PSR_L`	1019

General Integer Registers

Name	whichReg
`_IA64_REG_GP`	1025
`_IA64_REG_SP`	1036
`_IA64_REG_TP`	1037

Application Registers

Name	whichReg
`_IA64_REG_AR_KR0`	3072
`_IA64_REG_AR_KR1`	3073
`_IA64_REG_AR_KR2`	3074
`_IA64_REG_AR_KR3`	3075
`_IA64_REG_AR_KR4`	3076
`_IA64_REG_AR_KR5`	3077
`_IA64_REG_AR_KR6`	3078
`_IA64_REG_AR_KR7`	3079
`_IA64_REG_AR_RSC`	3088
`_IA64_REG_AR_BSP`	3089
`_IA64_REG_AR_BSPSTORE`	3090
`_IA64_REG_AR_RNAT`	3091
`_IA64_REG_AR_FCR`	3093
`_IA64_REG_AR_EFLAG`	3096
`_IA64_REG_AR_CSD`	3097

`_IA64_REG_AR_SSD`	3098
`_IA64_REG_AR_CFLAG`	3099
`_IA64_REG_AR_FSR`	3100
`_IA64_REG_AR_FIR`	3101
`_IA64_REG_AR_FDR`	3102
`_IA64_REG_AR_CCV`	3104
`_IA64_REG_AR_UNAT`	3108
`_IA64_REG_AR_FPSR`	3112
`_IA64_REG_AR_ITC`	3116
`_IA64_REG_AR_PFS`	3136
`_IA64_REG_AR_LC`	3137
`_IA64_REG_AR_EC`	3138

Control Registers

Name	whichReg
`_IA64_REG_CR_DCR`	4096
`_IA64_REG_CR_ITM`	4097
`_IA64_REG_CR_IVA`	4098
`_IA64_REG_CR_PTA`	4104
`_IA64_REG_CR_IPSR`	4112
`_IA64_REG_CR_ISR`	4113
`_IA64_REG_CR_IIP`	4115
`_IA64_REG_CR_IFA`	4116
`_IA64_REG_CR_ITIR`	4117
`_IA64_REG_CR_IIPA`	4118
`_IA64_REG_CR_IFS`	4119
`_IA64_REG_CR_IIM`	4120
`_IA64_REG_CR_IHA`	4121
`_IA64_REG_CR_LID`	4160
`_IA64_REG_CR_IVR`	4161 *
`_IA64_REG_CR_TPR`	4162
`_IA64_REG_CR_EOI`	4163

`_IA64_REG_CR_IRR0`	4164 *
`_IA64_REG_CR_IRR1`	4165 *
`_IA64_REG_CR_IRR2`	4166 *
`_IA64_REG_CR_IRR3`	4167 *
`_IA64_REG_CR_ITV`	4168
`_IA64_REG_CR_PMV`	4169
`_IA64_REG_CR_CMCV`	4170
`_IA64_REG_CR_LRR0`	4176
`_IA64_REG_CR_LRR1`	4177

* `getReg` only

Indirect Registers for getIndReg() and setIndReg()

Name	**whichReg**
`_IA64_REG_INDR_CPUID`	9000 *
`_IA64_REG_INDR_DBR`	9001
`_IA64_REG_INDR_IBR`	9002
`_IA64_REG_INDR_PKR`	9003
`_IA64_REG_INDR_PMC`	9004
`_IA64_REG_INDR_PMD`	9005
`_IA64_REG_INDR_RR`	9006
`_IA64_REG_INDR_RESERVED`	9007

* `getIndReg` only

Multimedia Additions for Itanium®-based Systems

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Intrinsic	Corresponding Instruction
`__int64 _m64_czx1l(__m64 a)`	`czx1.l` (Compute Zero Index)
`__int64 _m64_czx1r(__m64 a)`	`czx1.r` (Compute Zero Index)
`__int64 _m64_czx2l(__m64 a)`	`czx2.l` (Compute Zero Index)
`__int64 _m64_czx2r(__m64 a)`	`czx2.r` (Compute Zero Index)
`__m64 _m64_mix1l(__m64 a, __m64 b)`	`mix1.l` (Mix)
`__m64 _m64_mix1r(__m64 a, __m64 b)`	`mix1.r` (Mix)
`__m64 _m64_mix2l(__m64 a, __m64 b)`	`mix2.l` (Mix)
`__m64 _m64_mix2r(__m64 a, __m64 b)`	`mix2.r` (Mix)
`__m64 _m64_mix4l(__m64 a, __m64 b)`	`mix4.l` (Mix)
`__m64 _m64_mix4r(__m64 a, __m64 b)`	`mix4.r` (Mix)
`__m64 _m64_mux1(__m64 a, const int n)`	`mux1` (Mux)
`__m64 _m64_mux2(__m64 a, const int n)`	`mux2` (Mux)
`__m64 _m64_padd1uus(__m64 a, __m64 b)`	`padd1.uus` (Parallel add)
`__m64 _m64_padd2uus(__m64 a, __m64 b)`	`padd2.uus` (Parallel add)
`__m64 _m64_pavg1_nraz(__m64 a, __m64 b)`	`pavg1` (Parallel average)
`__m64 _m64_pavg2_nraz(__m64 a, __m64 b)`	`pavg2` (Parallel average)
`__m64 _m64_pavgsub1(__m64 a, __m64 b)`	`pavgsub1` (Parallel average subtract)
`__m64 _m64_pavgsub2(__m64 a, __m64 b)`	`pavgsub2` (Parallel average subtract)
`__m64 _m64_pmpy2r(__m64 a, __m64 b)`	`pmpy2.r` (Parallel multiply)
`__m64 _m64_pmpy2l(__m64 a, __m64 b)`	`pmpy2.l` (Parallel multiply)
`__m64 _m64_pmpyshr2(__m64 a, __m64 b, const int count)`	`pmpyshr2` (Parallel multiply and shift right)
`__m64 _m64_pmpyshr2u(__m64 a, __m64 b, const int count)`	`pmpyshr2.u` (Parallel multiply and shift right)
`__m64 _m64_pshladd2(__m64 a, const int count, __m64 b)`	`pshladd2` (Parallel shift left and add)

`__m64 _m64_pshradd2(__m64 a, const int count, __m64 b)`	`pshradd2` (Parallel shift right and add)
`__m64 _m64_psub1uus(__m64 a, __m64 b)`	`psub1.uus` (Parallel subtract)
`__m64 _m64_psub2uus(__m64 a, __m64 b)`	`psub2.uus` (Parallel subtract)

`__int64 _m64_czx1l(__m64 a)`

The 64-bit value a is scanned for a zero element from the most significant element to the least significant element, and the index of the first zero element is returned. The element width is 8 bits, so the range of the result is from 0 - 7. If no zero element is found, the default result is 8.

`__int64 _m64_czx1r(__m64 a)`

The 64-bit value a is scanned for a zero element from the least significant element to the most significant element, and the index of the first zero element is returned. The element width is 8 bits, so the range of the result is from 0 - 7. If no zero element is found, the default result is 8.

`__int64 _m64_czx2l(__m64 a)`

The 64-bit value a is scanned for a zero element from the most significant element to the least significant element, and the index of the first zero element is returned. The element width is 16 bits, so the range of the result is from 0 - 3. If no zero element is found, the default result is 4.

`__int64 _m64_czx2r(__m64 a)`

The 64-bit value a is scanned for a zero element from the least significant element to the most significant element, and the index of the first zero element is returned. The element width is 16 bits, so the range of the result is from 0 - 3. If no zero element is found, the default result is 4.

`__m64 _m64_mix1l(__m64 a, __m64 b)`

Interleave 64-bit quantities a and b in 1-byte groups, starting from the left, as shown in Figure 1, and return the result.

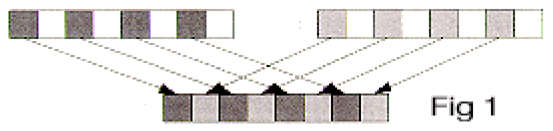


Fig 1

`__m64 _m64_mix1r(__m64 a, __m64 b)`

Interleave 64-bit quantities a and b in 1-byte groups, starting from the right, as shown in Figure 2, and return the result.

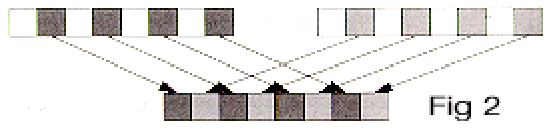
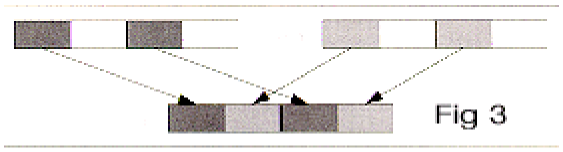


Fig 2

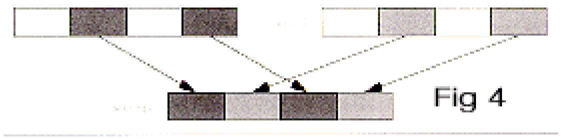
`__m64 _m64_mix2l(__m64 a, __m64 b)`

Interleave 64-bit quantities a and b in 2-byte groups, starting from the left, as shown in Figure 3, and return the result.



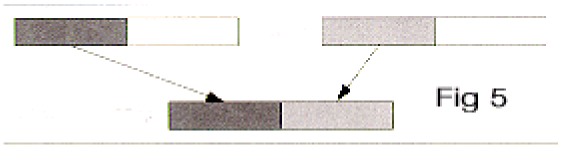
```
__m64 _m64_mix2r(__m64 a, __m64 b)
```

Interleave 64-bit quantities a and b in 2-byte groups, starting from the right, as shown in Figure 4, and return the result.



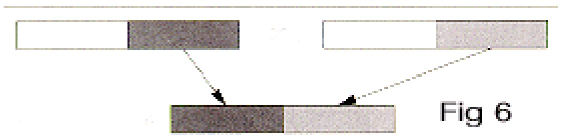
```
__m64 _m64_mix4l(__m64 a, __m64 b)
```

Interleave 64-bit quantities a and b in 4-byte groups, starting from the left, as shown in Figure 5, and return the result.



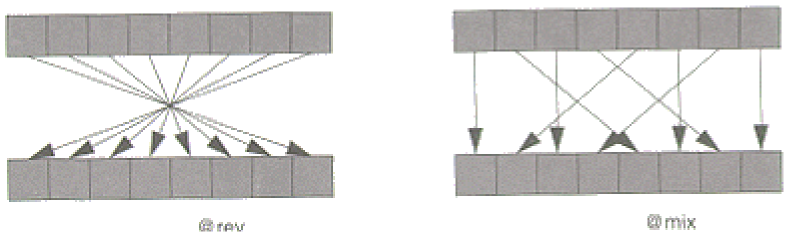
```
__m64 _m64_mix4r(__m64 a, __m64 b)
```

Interleave 64-bit quantities a and b in 4-byte groups, starting from the right, as shown in Figure 6, and return the result.



```
__m64 _m64_mux1(__m64 a, const int n)
```

Based on the value of n, a permutation is performed on a as shown in Figure 7, and the result is returned. Table 1 shows the possible values of n.



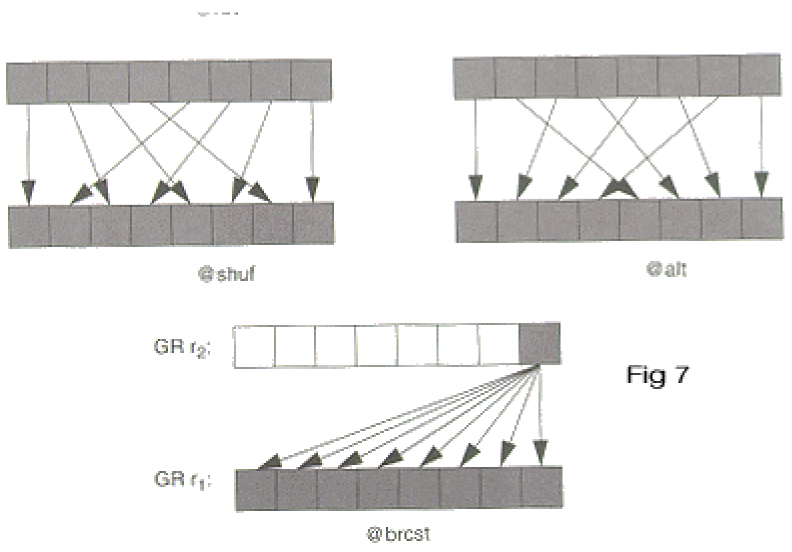


Fig 7

Table 1. Values of n for m64_mux1Operation

	n
`@brcst`	`0`
`@mix`	`8`
`@shuf`	`9`
`@alt`	`0xA`
`@rev`	`0xB`

```
__m64 _m64_mux2(__m64 a, const int n)
```

Based on the value of n, a permutation is performed on a as shown in Figure 8, and the result is returned.

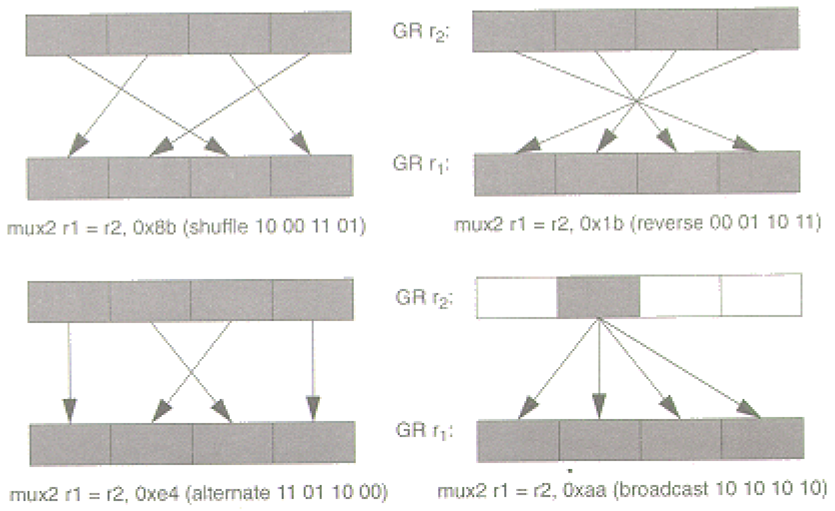

GR r_2:
GR r_1:
mux2 r1 = r2, 0x8b (shuffle 10 00 11 01)
mux2 r1 = r2, 0x1b (reverse 00 01 10 11)
GR r_2:
GR r_1:
mux2 r1 = r2, 0xe4 (alternate 11 01 10 00)
mux2 r1 = r2, 0xaa (broadcast 10 10 10 10)

Fig 8

```
__m64 _m64_pavgsub1(__m64 a, __m64 b)
```

The unsigned data elements (bytes) of b are subtracted from the unsigned data elements (bytes) of a and the results of the subtraction are then each independently shifted to the right by one position. The high-order bits of each element are filled with the borrow bits of the subtraction.

```
__m64 _m64_pavgsub2(__m64 a, __m64 b)
```

The unsigned data elements (double bytes) of b are subtracted from the unsigned data elements (double bytes) of a and the results of the subtraction are then each independently shifted to the right by one position. The high-order bits of each element are filled with the borrow bits of the subtraction.

```
__m64 _m64_pmpy2l(__m64 a, __m64 b)
```

Two signed 16-bit data elements of a, starting with the most significant data element, are multiplied by the corresponding two signed 16-bit data elements of b, and the two 32-bit results are returned as shown in Figure 9.

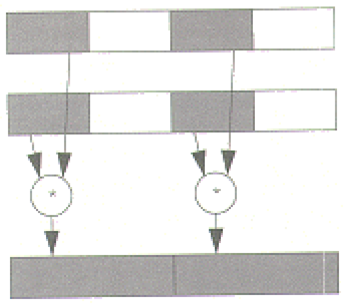


Fig 9

```
__m64 _m64_pmpy2r(__m64 a, __m64 b)
```

Two signed 16-bit data elements of a, starting with the least significant data element, are multiplied by the corresponding two signed 16-bit data elements of b, and the two 32-bit results are returned as shown in Figure 10.

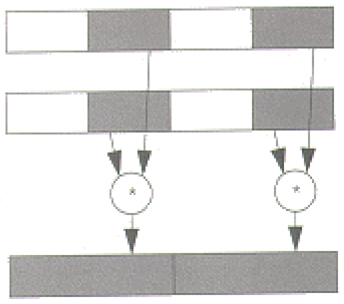


Fig 10

```
__m64 _m64_pmpyshr2(__m64 a, __m64 b, const int count)
```

The four signed 16-bit data elements of a are multiplied by the corresponding signed 16-bit data elements of b, yielding four 32-bit products. Each product is then shifted to the right count bits and the least significant 16 bits of each shifted product form 4 16-bit results, which are returned as one 64-bit word.

```
__m64 _m64_pmpyshr2u(__m64 a, __m64 b, const int count)
```

The four unsigned 16-bit data elements of a are multiplied by the corresponding unsigned 16-bit data elements of b, yielding four 32-bit products. Each product is then shifted to the right count bits and the least significant 16 bits of each shifted product form 4 16-bit results, which are returned as one 64-bit word.

```
__m64 _m64_pshladd2(__m64 a, const int count, __m64 b)
```

a is shifted to the left by count bits and then is added to b. The upper 32 bits of the result are forced to 0, and then bits [31:30] of b are copied to bits [62:61] of the result. The result is returned.

```
__m64 _m64_pshradd2(__m64 a, const int count, __m64 b)
```

The four signed 16-bit data elements of a are each independently shifted to the right by `count` bits (the high order bits of each element are filled with the initial value of the sign bits of the data elements in a); they are then added to the four signed 16-bit data elements of b. The result is returned.

```
__m64 _m64_padd1uus(__m64 a, __m64 b)
```

a is added to b as eight separate byte-wide elements. The elements of a are treated as unsigned, while the elements of b are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_padd2uus(__m64 a, __m64 b)
```

a is added to b as four separate 16-bit wide elements. The elements of a are treated as unsigned, while the elements of b are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_psub1uus(__m64 a, __m64 b)
```

a is subtracted from b as eight separate byte-wide elements. The elements of a are treated as unsigned, while the elements of b are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_psub2uus(__m64 a, __m64 b)
```

a is subtracted from b as four separate 16-bit wide elements. The elements of a are treated as unsigned, while the elements of b are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_pavg1_nraz(__m64 a, __m64 b)
```

The unsigned byte-wide data elements of a are added to the unsigned byte-wide data elements of b and the results of each add are then independently shifted to the right by one position. The high-order bits of each element are filled with the carry bits of the sums.

```
__m64 _m64_pavg2_nraz(__m64 a, __m64 b)
```

The unsigned 16-bit wide data elements of a are added to the unsigned 16-bit wide data elements of b and the results of each add are then independently shifted to the right by one position. The high-order bits of each element are filled with the carry bits of the sums.

Overview: Data Alignment, Memory Allocation Intrinsics, and Inline Assembly

This book describes features that support usage of the intrinsics. The following topics are described:

- Alignment Support
- Allocating and Freeing Aligned Memory Blocks
- Inline Assembly

Alignment Support

To improve intrinsics performance, you need to align data. For example, when you are using the Streaming SIMD Extensions, you should align data to 16 bytes in memory operations to improve performance. Specifically, you must align `__m128` objects as addresses passed to the `_mm_load` and `_mm_store` intrinsics. If you want to declare arrays of floats and treat them as `__m128` objects by casting, you need to ensure that the float arrays are properly aligned.

Use `__declspec(align)` to direct the compiler to align data more strictly than it otherwise does on both IA-32 and Itanium®-based systems. For example, a data object of type int is allocated at a byte address which is a multiple of 4 by default (the size of an int). However, by using `__declspec (align)`, you can direct the compiler to instead use an address which is a multiple of 8, 16, or 32 with the following restrictions on IA-32:

- 32-byte addresses must be statically allocated
- 16-byte addresses can be locally or statically allocated

You can use this data alignment support as an advantage in optimizing cache line usage. By clustering small objects that are commonly used together into a `struct`, and forcing the `struct` to be allocated at the beginning of a cache line, you can effectively guarantee that each object is loaded into the cache as soon as any one is accessed, resulting in a significant performance benefit.

The syntax of this extended-attribute is as follows:

```
align(n)
```

where n is an integral power of 2, less than or equal to 32. The value specified is the requested alignment.

Caution

In this release, `__declspec(align(8))` does not function correctly. Use `__declspec(align (16))` instead.

Note

If a value is specified that is less than the alignment of the affected data type, it has no effect. In other words, data is aligned to the maximum of its own alignment or the alignment specified with `__declspec(align)`.

You can request alignments for individual variables, whether of static or automatic storage duration. (Global and static variables have static storage duration; local variables have automatic storage duration by default.) You cannot adjust the alignment of a parameter, nor a field of a `struct` or `class`. You can, however, increase the alignment of a `struct` (or `union` or `class`), in which case every object of that type is affected.

As an example, suppose that a function uses local variables `i` and `j` as subscripts into a 2-dimensional array. They might be declared as follows:

```
int i, j;
```

These variables are commonly used together. But they can fall in different cache lines, which could be detrimental to performance. You can instead declare them as follows:

```
__declspec(align(8)) struct { int i, j; } sub;
```

The compiler now ensures that they are allocated in the same cache line. In C++, you can omit the `struct` variable name (written as sub in the above example). In C, however, it is required, and you must write references to `i` and `j` as `sub.i` and `sub.j`.

If you use many functions with such subscript pairs, it is more convenient to declare and use a `struct` type for them, as in the following example:

```
typedef struct __declspec(align(8)) { int i, j; } Sub;
```

By placing the `__declspec(align)` after the keyword `struct`, you are requesting the appropriate alignment for all objects of that type. However, that allocation of parameters is unaffected by `__declspec(align)`. (If necessary, you can assign the value of a parameter to a local variable with the appropriate alignment.)

You can also force alignment of global variables, such as arrays:

```
__declspec(align(16)) float array[1000];
```

Allocating and Freeing Aligned Memory Blocks

Use the `_mm_malloc` and `_mm_free` intrinsics to allocate and free aligned blocks of memory. These intrinsics are based on `malloc` and `free`, which are in the `libirc.a` library. You need to include `malloc.h`. The syntax for these intrinsics is as follows:

```
void* _mm_malloc (int size, int align)
```

```
void _mm_free (void *p)
```

The `_mm_malloc` routine takes an extra parameter, which is the alignment constraint. This constraint must be a power of two. The pointer that is returned from `_mm_malloc` is guaranteed to be aligned on the specified boundary.

Note

Memory that is allocated using `_mm_malloc` must be freed using `_mm_free` . Calling `free` on memory allocated with `_mm_malloc` or calling `_mm_free` on memory allocated with `malloc` will cause unpredictable behavior.

Inline Assembly

By default, the compiler inlines a number of standard C, C++, and math library functions. This usually results in faster execution of your program.

Sometimes inline expansion of library functions can cause unexpected results. The inlined library functions do not set the `errno` variable. So, in code that relies upon the setting of the `errno` variable, you should use the `-nolib_inline` option, which turns off inline expansion of library functions. Also, if one of your functions has the same name as one of the compiler's supplied library functions, the compiler assumes that it is one of the latter and replaces the call with the inlined version. Consequently, if the program defines a function with the same name as one of the known library routines, you must use the `-nolib_inline` option to ensure that the program's function is the one used.

Note

Automatic inline expansion of library functions is not related to the inline expansion that the compiler does during interprocedural optimizations. For example, the following command compiles the program sum.c without expanding the library functions, but with inline expansion from interprocedural optimizations (IPO):

- **IA-32 Systems:** `prompt>`**`icc -ip -nolib_inline sum.c`**
- **Itanium®-based Systems:** `prompt>`**`ecc -ip -nolib_inline sum.c`**

For details on IPO, see Interprocedural Optimizations.

MASM* Style Inline Assembly

The Intel® C++ Compiler supports MASM style inline assembly with the `-use_msasm` option. See your MASM documentation for the proper syntax.

GNU*-like Style Inline Assembly

The Intel® C++ Compiler supports GNU-like style inline assembly. The syntax is as follows:

```
asm-keyword [ volatile-keyword ] ( asm-template [ asm-interface ] ) ;
```

Syntax Element	Description
`asm-keyword`	`asm` statements begin with the keyword `asm`. Alternatively, either `__asm` or `__asm__` may be used for compatibility.
`volatile-keyword`	If the optional keyword `volatile` is given, the `asm` is volatile. Two `volatile asm` statements will never be moved past each other, and a reference to a `volatile` variable will not be moved relative to a volatile `asm`. Alternate keywords `__volatile` and `__volatile__` may be used for compatibility.

`asm-template`	The `asm-template` is a C language ASCII string which specifies how to output the assembly code for an instruction. Most of the template is a fixed string; everything but the substitution-directives, if any, is passed through to the assembler. The syntax for a substitution directive is a `%` followed by one or two characters. The supported substitution directives are specified in a subsequent section.
`asm-interface`	The `asm-interface` consists of three parts: 1. an optional `output-list` 2. an optional `input-list` 3. an optional `clobber-list` These are separated by colon (:) characters. If the `output-list` is missing, but an `input-list` is given, the input list may be preceded by two colons (::)to take the place of the missing `output-list`. If the `asm-interface` is omitted altogether, the `asm` statement is considered `volatile` regardless of whether a `volatile-keyword` was specified.
`output-list`	An `output-list` consists of one or more `output-specs` separated by commas. For the purposes of substitution in the `asm-template`, each `output-spec` is numbered. The first operand in the `output-list` is numbered 0, the second is 1, and so on. Numbering is continuous through the `output-list` and into the `input-list`. The total number of operands is limited to 10 (i.e. 0-9).
`input-list`	Similar to an `output-list`, an `input-list` consists of one or more `input-specs` separated by commas. For the purposes of substitution in the `asm-template`, each `input-spec` is numbered, with the numbers continuing from those in the `output-list`.
`clobber-list`	A `clobber-list` tells the compiler that the `asm` uses or changes a specific machine register that is either coded directly into the `asm` or is changed implicitly by the assembly instruction. The `clobber-list` is a comma-separated list of `clobber-specs`.
`input-spec`	The `input-specs` tell the compiler about expressions whose values may be needed by the inserted assembly instruction. In order to describe fully the input requirements of the `asm`, you can list `input-specs` that are not actually referenced in the `asm-template`.
`clobber-spec`	Each `clobber-spec` specifies the name of a single machine register that is clobbered. The register name may optionally be preceded by a `%`. The following are the valid register names: eax, ebx, ecx, edx, esi, edi, ebp, esp, ax, bx, cx, dx, si, di, bp, sp, al, bl, cl, dl, ah, bh, ch, dh, st, st(1) – st(7), mm0 – mm7, xmm0 – xmm7, and cc. It is also legal to specify "memory" in a `clobber-spec`. This prevents the compiler from keeping data cached in registers across the `asm` statement.

Intrinsics Cross-processor Implementation

This section provides a series of tables that compare intrinsics performance across architectures. Before implementing intrinsics across architectures, please note the following.

- Instrinsics may generate code that does not run on all IA processors. Therefore the programmer is responsible for using `CPUID` to detect the processor and generating the appropriate code.
- Implement intrinsics by processor family, not by specific processor. The guiding principle for which family–IA-32 or Itanium® processors–the intrinsic is implemented on is performance, not compatibility. Where there is added performance on both families, the intrinsic will be identical.

Intrinsics for Implementation Across All IA

Key to the table entries

- A = Expected to give significant performance gain over non-intrinsic-based code equivalent.
- B = Non-intrinsic-based source code would be better; the intrinsic's implementation may map directly to native instructions, but they offer no significant performance gain.
- C = Requires contorted implementation for particular microarchitecture. Will result in very poor performance if used.

Intrinsic	Across All IA	MMX(TM) Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium® Architecture
`int abs(int)`	A	A	A	A	A
`long labs(long)`	A	A	A	A	A
`unsigned long __lrotl (unsigned long value, int shift)`	A	A	A	A	A
`unsigned long __lrotr (unsigned long value, int shift)`	A	A	A	A	A
`unsigned int __rotl (unsigned int value, int shift)`	A	A	A	A	A
`unsigned int __rotr (unsigned int value, int shift)`	A	A	A	A	A
`__int64 __i64_rotl (__int64 value, int shift)`	A	A	A	A	A
`__int64 __i64_rotr (__int64 value, int shift)`	A	A	A	A	A
`double fabs(double)`	A	A	A	A	A
`double log(double)`	A	A	A	A	A
`float logf(float)`	A	A	A	A	A
`double log10(double)`	A	A	A	A	A
`float log10f(float)`	A	A	A	A	A
`double exp(double)`	A	A	A	A	A
`float expf(float)`	A	A	A	A	A
`double pow(double, double)`	A	A	A	A	A

`float powf(float, float)`	A	A	A	A	A
`double sin(double)`	A	A	A	A	A
`float sinf(float)`	A	A	A	A	A
`double cos(double)`	A	A	A	A	A
`float cosf(float)`	A	A	A	A	A
`double tan(double)`	A	A	A	A	A
`float tanf(float)`	A	A	A	A	A
`double acos(double)`	A	A	A	A	A
`float acosf(float)`	A	A	A	A	A
`double acosh(double)`	A	A	A	A	A
`float acoshf(float)`	A	A	A	A	A
`double asin(double)`	A	A	A	A	A
`float asinf(float)`	A	A	A	A	A
`double asinh(double)`	A	A	A	A	A
`float asinhf(float)`	A	A	A	A	A
`double atan(double)`	A	A	A	A	A
`float atanf(float)`	A	A	A	A	A
`double atanh(double)`	A	A	A	A	A
`float atanhf(float)`	A	A	A	A	A
`float cabs(double)*`	A	A	A	A	A
`double ceil(double)`	A	A	A	A	A
`float ceilf(float)`	A	A	A	A	A
`double cosh(double)`	A	A	A	A	A
`float coshf(float)`	A	A	A	A	A
`float fabsf(float)`	A	A	A	A	A
`double floor(double)`	A	A	A	A	A
`float floorf(float)`	A	A	A	A	A
`double fmod(double)`	A	A	A	A	A
`float fmodf(float)`	A	A	A	A	A
`double hypot(double, double)`	A	A	A	A	A
`float hypotf(float)`	A	A	A	A	A

double rint(double)	A	A	A	A	A
float rintf(float)	A	A	A	A	A
double sinh(double)	A	A	A	A	A
float sinhf(float)	A	A	A	A	A
float sqrtf(float)	A	A	A	A	A
double tanh(double)	A	A	A	A	A
float tanhf(float)	A	A	A	A	A
char *_strset(char *, _int32)	A	A	A	A	A
void *memcmp(const void *cs, const void *ct, size_t n)	A	A	A	A	A
void *memcpy(void *s, const void *ct, size t n)	A	A	A	A	A
void *memset(void * s, int c, size_t n)	A	A	A	A	A
char *Strcat(char * s, const char * ct)	A	A	A	A	A
int *strcmp(const char *, const char *)	A	A	A	A	A
char *strcpy(char * s, const char * ct)	A	A	A	A	A
size_t strlen(const char * cs)	A	A	A	A	A
int strncmp(char *, char *, int)	A	A	A	A	A
int strncpy(char *, char *, int)	A	A	A	A	A
void *__alloca(int)	A	A	A	A	A
int _setjmp(jmp_buf)	A	A	A	A	A
_exception_code(void)	A	A	A	A	A
_exception_info(void)	A	A	A	A	A
_abnormal_termination (void)	A	A	A	A	A
void _enable()	A	A	A	A	A
void _disable()	A	A	A	A	A
int _bswap(int)	A	A	A	A	A
int _in_byte(int)	A	A	A	A	A

`int _in_dword(int)`	A	A	A	A	A
`int _in_word(int)`	A	A	A	A	A
`int _inp(int)`	A	A	A	A	A
`int _inpd(int)`	A	A	A	A	A
`int _inpw(int)`	A	A	A	A	A
`int _out_byte(int, int)`	A	A	A	A	A
`int _out_dword(int, int)`	A	A	A	A	A
`int _out_word(int, int)`	A	A	A	A	A
`int _outp(int, int)`	A	A	A	A	A
`int _outpd(int, int)`	A	A	A	A	A
`int _outpw(int, int)`	A	A	A	A	A

MMX(TM) Technology Intrinsics Implementation

Key to the table entries

- A = Expected to give significant performance gain over non-intrinsic-based code equivalent.
- B = Non-intrinsic-based source code would be better; the intrinsic's implementation may map directly to native instructions, but they offer no significant performance gain.
- C = Requires contorted implementation for particular microarchitecture. Will result in very poor performance if used.

Intrinsic Name	Alternate Name	Across All IA	MMX(TM Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Ita Ar
`_m_empty`	`_mm_empty`	N/A	A	A	A	B
`_m_from_int`	`_mm_cvtsi32_si64`	N/A	A	A	A	A
`_m_to_int`	`_mm_cvtsi64_si32`	N/A	A	A	A	A
`_m_packsswb`	`_mm_packs_pi16`	N/A	A	A	A	A
`_m_packssdw`	`_mm_packs_pi32`	N/A	A	A	A	A
`_m_packuswb`	`_mm_packs_pu16`	N/A	A	A	A	A
`_m_punpckhbw`	`_mm_unpackhi_pi8`	N/A	A	A	A	A
`_m_punpckhwd`	`_mm_unpackhi_pi16`	N/A	A	A	A	A
`_m_punpckhdq`	`_mm_unpackhi_pi32`	N/A	A	A	A	A
`_m_punpcklbw`	`_mm_unpacklo_pi8`	N/A	A	A	A	A
`_m_punpcklwd`	`_mm_unpacklo_pi16`	N/A	A	A	A	A
`_m_punpckldq`	`_mm_unpacklo_pi32`	N/A	A	A	A	A
`_m_paddb`	`_mm_add_pi8`	N/A	A	A	A	A
`_m_paddw`	`_mm_add_pi16`	N/A	A	A	A	A
`_m_paddd`	`_mm_add_pi32`	N/A	A	A	A	A
`_m_paddsb`	`_mm_adds_pi8`	N/A	A	A	A	A
`_m_paddsw`	`_mm_adds_pi16`	N/A	A	A	A	A
`_m_paddusb`	`_mm_adds_pu8`	N/A	A	A	A	A
`_m_paddusw`	`_mm_adds_pu16`	N/A	A	A	A	A
`_m_psubb`	`_mm_sub_pi8`	N/A	A	A	A	A
`_m_psubw`	`_mm_sub_pi16`	N/A	A	A	A	A
`_m_psubd`	`_mm_sub_pi32`	N/A	A	A	A	A

_m_psubsb	_mm_subs_pi8	N/A	A	A	A	A
_m_psubsw	_mm_subs_pi16	N/A	A	A	A	A
_m_psubusb	_mm_subs_pu8	N/A	A	A	A	A
_m_psubusw	_mm_subs_pu16	N/A	A	A	A	A
_m_pmaddwd	_mm_madd_pi16	N/A	A	A	A	C
_m_pmulhw	_mm_mulhi_pi16	N/A	A	A	A	A
_m_pmullw	_mm_mullo_pi16	N/A	A	A	A	A
_m_psllw	_mm_sll_pi16	N/A	A	A	A	A
_m_psllwi	_mm_slli_pi16	N/A	A	A	A	A
_m_pslld	_mm_sll_pi32	N/A	A	A	A	A
_m_pslldi	_mm_slli_pi32	N/A	A	A	A	A
_m_psllq	_mm_sll_si64	N/A	A	A	A	A
_m_psllqi	_mm_slli_si64	N/A	A	A	A	A
_m_psraw	_mm_sra_pi16	N/A	A	A	A	A
_m_psrawi	_mm_srai_pi16	N/A	A	A	A	A
_m_psrad	_mm_sra_pi32	N/A	A	A	A	A
_m_psradi	_mm_srai_pi32	N/A	A	A	A	A
_m_psrlw	_mm_srl_pi16	N/A	A	A	A	A
_m_psrlwi	_mm_srli_pi16	N/A	A	A	A	A
_m_psrld	_mm_srl_pi32	N/A	A	A	A	A
_m_psrldi	_mm_srli_pi32	N/A	A	A	A	A
_m_psrlq	_mm_srl_si64	N/A	A	A	A	A
_m_psrlqi	_mm_srli_si64	N/A	A	A	A	A
_m_pand	_mm_and_si64	N/A	A	A	A	A
_m_pandn	_mm_andnot_si64	N/A	A	A	A	A
_m_por	_mm_or_si64	N/A	A	A	A	A
_m_pxor	_mm_xor_si64	N/A	A	A	A	A
_m_pcmpeqb	_mm_cmpeq_pi8	N/A	A	A	A	A
_m_pcmpeqw	_mm_cmpeq_pi16	N/A	A	A	A	A
_m_pcmpeqd	_mm_cmpeq_pi32	N/A	A	A	A	A
_m_pcmpgtb	_mm_cmpgt_pi8	N/A	A	A	A	A
_m_pcmpgtw	_mm_cmpgt_pi16	N/A	A	A	A	A

_m_pcmpgtd	_mm_cmpgt_pi32	N/A	A	A	A	A
mm setzero si64		N/A	A	A	A	A
_mm_set_pi32		N/A	A	A	A	A
_mm_set_pi16		N/A	A	A	A	C
_mm_set_pi8		N/A	A	A	A	C
_mm_set1_pi32		N/A	A	A	A	A
_mm_set1_pi16		N/A	A	A	A	A
_mm_set1_pi8		N/A	A	A	A	A
_mm_setr_pi32		N/A	A	A	A	A
_mm_setr_pi16		N/A	A	A	A	C
_mm_setr_pi8		N/A	A	A	A	C

`_mm_empty` is implemented in Itanium® instructions as a NOP for source compatibility only.

Streaming SIMD Extensions Intrinsics Implementation

Regular Streaming SIMD Extensions intrinsics work on 4 32-bit single precision values. On Itanium®-based systemsbasic operations like add or compare will require two SIMD instructions. Both can be executed in the same cycle so the throughput is one basic Streaming SIMD Extensions operation per cycle or 4 32-bit single precision operations per cycle.

Key to the table entries

- A = Expected to give significant performance gain over non-intrinsic-based code equivalent.
- B = Non-intrinsic-based source code would be better; the intrinsic's implementation may map directly to native instructions but they offer no significant performance gain.
- C = Requires contorted implementation for particular microarchitecture. Will result in very poor performance if used.

Intrinsic Name	Alternate Name	Across All IA	MMX(TM Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Ita Ar
`_mm_add_ss`		N/A	N/A	B	B	B
`_mm_add_ps`		N/A	N/A	A	A	A
`_mm_sub_ss`		N/A	N/A	B	B	B
`_mm_sub_ps`		N/A	N/A	A	A	A
`_mm_mul_ss`		N/A	N/A	B	B	B
`_mm_mul_ps`		N/A	N/A	A	A	A
`_mm_div_ss`		N/A	N/A	B	B	B
`_mm_div_ps`		N/A	N/A	A	A	A
`_mm_sqrt_ss`		N/A	N/A	B	B	B
`_mm_sqrt_ps`		N/A	N/A	A	A	A
`_mm_rcp_ss`		N/A	N/A	B	B	B
`_mm_rcp_ps`		N/A	N/A	A	A	A
`_mm_rsqrt_ss`		N/A	N/A	B	B	B
`_mm_rsqrt_ps`		N/A	N/A	A	A	A
`_mm_min_ss`		N/A	N/A	B	B	B
`_mm_min_ps`		N/A	N/A	A	A	A
`_mm_max_ss`		N/A	N/A	B	B	B
`_mm_max_ps`		N/A	N/A	A	A	A

_mm_and_ps		N/A	N/A	A	A	A
_mm_andnot_ps		N/A	N/A	A	A	A
_mm_or_ps		N/A	N/A	A	A	A
_mm_xor_ps		N/A	N/A	A	A	A
_mm_cmpeq_ss		N/A	N/A	B	B	B
_mm_cmpeq_ps		N/A	N/A	A	A	A
_mm_cmplt_ss		N/A	N/A	B	B	B
_mm_cmplt_ps		N/A	N/A	A	A	A
_mm_cmple_ss		N/A	N/A	B	B	B
_mm_cmple_ps		N/A	N/A	A	A	A
_mm_cmpgt_ss		N/A	N/A	B	B	B
_mm_cmpgt_ps		N/A	N/A	A	A	A
_mm_cmpge_ss		N/A	N/A	B	B	B
_mm_cmpge_ps		N/A	N/A	A	A	A
_mm_cmpneq_ss		N/A	N/A	B	B	B
_mm_cmpneq_ps		N/A	N/A	A	A	A
_mm_cmpnlt_ss		N/A	N/A	B	B	B
_mm_cmpnlt_ps		N/A	N/A	A	A	A
_mm_cmpnle_ss		N/A	N/A	B	B	B
_mm_cmpnle_ps		N/A	N/A	A	A	A
_mm_cmpngt_ss		N/A	N/A	B	B	B
_mm_cmpngt_ps		N/A	N/A	A	A	A
_mm_cmpnge_ss		N/A	N/A	B	B	B
_mm_cmpnge_ps		N/A	N/A	A	A	A
_mm_cmpord_ss		N/A	N/A	B	B	B
_mm_cmpord_ps		N/A	N/A	A	A	A
_mm_cmpunord_ss		N/A	N/A	B	B	B
_mm_cmpunord_ps		N/A	N/A	A	A	A
_mm_comieq_ss		N/A	N/A	B	B	B
_mm_comilt_ss		N/A	N/A	B	B	B
_mm_comile_ss		N/A	N/A	B	B	B
_mm_comigt_ss		N/A	N/A	B	B	B

`_mm_comige_ss`		N/A	N/A	B	B	B
`_mm_comineq_ss`		N/A	N/A	B	B	B
`_mm_ucomieq_ss`		N/A	N/A	B	B	B
`_mm_ucomilt_ss`		N/A	N/A	B	B	B
`_mm_ucomile_ss`		N/A	N/A	B	B	B
`_mm_ucomigt_ss`		N/A	N/A	B	B	B
`_mm_ucomige_ss`		N/A	N/A	B	B	B
`_mm_ucomineq_ss`		N/A	N/A	B	B	B
`_mm_cvt_ss2si`	`_mm_cvtss_si32`	N/A	N/A	A	A	B
`_mm_cvt_ps2pi`	`_mm_cvtps_pi32`	N/A	N/A	A	A	A
`_mm_cvtt_ss2si`	`_mm_cvttss_si32`	N/A	N/A	A	A	B
`_mm_cvtt_ps2pi`	`_mm_cvttps_pi32`	N/A	N/A	A	A	A
`_mm_cvt_si2ss`	`_mm_cvtsi32_ss`	N/A	N/A	A	A	B
`_mm_cvt_pi2ps`	`_mm_cvtpi32_ps`	N/A	N/A	A	A	C
`_mm_cvtpi16_ps`		N/A	N/A	A	A	C
`_mm_cvtpu16_ps`		N/A	N/A	A	A	C
`_mm_cvtpi8_ps`		N/A	N/A	A	A	C
`_mm_cvtpu8_ps`		N/A	N/A	A	A	C
`_mm_cvtpi32x2_ps`		N/A	N/A	A	A	C
`_mm_cvtps_pi16`		N/A	N/A	A	A	C
`_mm_cvtps_pi8`		N/A	N/A	A	A	C
`_mm_move_ss`		N/A	N/A	A	A	A
`_mm_shuffle_ps`		N/A	N/A	A	A	A
`_mm_unpackhi_ps`		N/A	N/A	A	A	A
`_mm_unpacklo_ps`		N/A	N/A	A	A	A
`_mm_movehl_ps`		N/A	N/A	A	A	A
`_mm_movelh_ps`		N/A	N/A	A	A	A
`_mm_movemask_ps`		N/A	N/A	A	A	C
`_mm_getcsr`		N/A	N/A	A	A	A
`_mm_setcsr`		N/A	N/A	A	A	A
`_mm_loadh_pi`		N/A	N/A	A	A	A
`_mm_loadl_pi`		N/A	N/A	A	A	A

_mm_load_ss		N/A	N/A	A	A	B
_mm_load_ps1	_mm_load1_ps	N/A	N/A	A	A	A
_mm_load_ps		N/A	N/A	A	A	A
_mm_loadu_ps		N/A	N/A	A	A	A
_mm_loadr_ps		N/A	N/A	A	A	A
_mm_storeh_pi		N/A	N/A	A	A	A
_mm_storel_pi		N/A	N/A	A	A	A
_mm_store_ss		N/A	N/A	A	A	A
_mm_store_ps		N/A	N/A	A	A	A
_mm_store_ps1	_mm_store1_ps	N/A	N/A	A	A	A
_mm_storeu_ps		N/A	N/A	A	A	A
_mm_storer_ps		N/A	N/A	A	A	A
_mm_set_ss		N/A	N/A	A	A	A
_mm_set_ps1	_mm_set1_ps	N/A	N/A	A	A	A
_mm_set_ps		N/A	N/A	A	A	A
_mm_setr_ps		N/A	N/A	A	A	A
_mm_setzero_ps		N/A	N/A	A	A	A
_mm_prefetch		N/A	N/A	A	A	A
_mm_stream_pi		N/A	N/A	A	A	A
_mm_stream_ps		N/A	N/A	A	A	A
_mm_sfence		N/A	N/A	A	A	A
_m_pextrw	_mm_extract_pi16	N/A	N/A	A	A	A
_m_pinsrw	_mm_insert_pi16	N/A	N/A	A	A	A
_m_pmaxsw	_mm_max_pi16	N/A	N/A	A	A	A
_m_pmaxub	_mm_max_pu8	N/A	N/A	A	A	A
_m_pminsw	_mm_min_pi16	N/A	N/A	A	A	A
_m_pminub	_mm_min_pu8	N/A	N/A	A	A	A
_m_pmovmskb	_mm_movemask_pi8	N/A	N/A	A	A	C
_m_pmulhuw	_mm_mulhi_pu16	N/A	N/A	A	A	A
_m_pshufw	_mm_shuffle_pi16	N/A	N/A	A	A	A
_m_maskmovq	_mm_maskmove_si64	N/A	N/A	A	A	C
_m_pavgb	_mm_avg_pu8	N/A	N/A	A	A	A

_m_pavgw	_mm_avg_pu16	N/A	N/A	A	A	A
_m_psadbw	_mm_sad_pu8	N/A	N/A	A	A	A

Streaming SIMD Extensions 2 Intrinsics Implementation

Streaming SIMD Extensions 2 operate on 128-bit quantities with 64-bit double precision floating-point values. The Intel® Itanium® processor does not support parallel double precision computation, so Streaming SIMD Extensions 2 are not implemented on Itanium-based systems.

Key to the table entries:

- A = Expected to give significant performance gain over non-intrinsic-based code equivalent.
- B = Non-intrinsic-based source code would be better; the intrinsic's implementation may map directly to native instructions, but they offer no significant performance gain.
- C = Requires contorted implementation for particular microarchitecture. Will result in very poor performance if used.

Intrinsic	Across All IA	MMX(TM Technology	Streaming SIMD Extentions	Pentium® 4 Processor Streaming SIMD Extensions 2	Itanium® Architecture
_mm_add_sd	N/A	N/A	N/A	A	N/A
_mm_add_pd	N/A	N/A	N/A	A	N/A
_mm_sub_sd	N/A	N/A	N/A	A	N/A
_mm_sub_pd	N/A	N/A	N/A	A	N/A
_mm_mul_sd	N/A	N/A	N/A	A	N/A
_mm_mul_pd	N/A	N/A	N/A	A	N/A
_mm_sqrt_sd	N/A	N/A	N/A	A	N/A
_mm_sqrt_pd	N/A	N/A	N/A	A	N/A
_mm_div_sd	N/A	N/A	N/A	A	N/A
_mm_div_pd	N/A	N/A	N/A	A	N/A
_mm_min_sd	N/A	N/A	N/A	A	N/A
_mm_min_pd	N/A	N/A	N/A	A	N/A
_mm_max_sd	N/A	N/A	N/A	A	N/A
_mm_max_pd	N/A	N/A	N/A	A	N/A
_mm_and_pd	N/A	N/A	N/A	A	N/A
_mm_andnot_pd	N/A	N/A	N/A	A	N/A
_mm_or_pd	N/A	N/A	N/A	A	N/A
_mm_xor_pd	N/A	N/A	N/A	A	N/A

_mm_cmpeq_sd	N/A	N/A	N/A	A	N/A
_mm_cmpeq_pd	N/A	N/A	N/A	A	N/A
_mm_cmplt_sd	N/A	N/A	N/A	A	N/A
_mm_cmplt_pd	N/A	N/A	N/A	A	N/A
_mm_cmple_sd	N/A	N/A	N/A	A	N/A
_mm_cmple_pd	N/A	N/A	N/A	A	N/A
_mm_cmpgt_sd	N/A	N/A	N/A	A	N/A
_mm_cmpgt_pd	N/A	N/A	N/A	A	N/A
_mm_cmpge_sd	N/A	N/A	N/A	A	N/A
_mm_cmpge_pd	N/A	N/A	N/A	A	N/A
_mm_cmpneq_sd	N/A	N/A	N/A	A	N/A
_mm_cmpneq_pd	N/A	N/A	N/A	A	N/A
_mm_cmpnlt_sd	N/A	N/A	N/A	A	N/A
_mm_cmpnlt_pd	N/A	N/A	N/A	A	N/A
_mm_cmpnle_sd	N/A	N/A	N/A	A	N/A
_mm_cmpnle_pd	N/A	N/A	N/A	A	N/A
_mm_cmpngt_sd	N/A	N/A	N/A	A	N/A
_mm_cmpngt_pd	N/A	N/A	N/A	A	N/A
_mm_cmpnge_sd	N/A	N/A	N/A	A	N/A
_mm_cmpnge_pd	N/A	N/A	N/A	A	N/A
_mm_cmpord_pd	N/A	N/A	N/A	A	N/A
_mm_cmpord_sd	N/A	N/A	N/A	A	N/A
_mm_cmpunord_pd	N/A	N/A	N/A	A	N/A
_mm_cmpunord_sd	N/A	N/A	N/A	A	N/A
_mm_comieq_sd	N/A	N/A	N/A	A	N/A
_mm_comilt_sd	N/A	N/A	N/A	A	N/A
_mm_comile_sd	N/A	N/A	N/A	A	N/A
_mm_comigt_sd	N/A	N/A	N/A	A	N/A
_mm_comige_sd	N/A	N/A	N/A	A	N/A
_mm_comineq_sd	N/A	N/A	N/A	A	N/A
_mm_ucomieq_sd	N/A	N/A	N/A	A	N/A
_mm_ucomilt_sd	N/A	N/A	N/A	A	N/A

_mm_ucomile_sd	N/A	N/A	N/A	A	N/A
_mm_ucomigt_sd	N/A	N/A	N/A	A	N/A
_mm_ucomige_sd	N/A	N/A	N/A	A	N/A
_mm_ucomineq_sd	N/A	N/A	N/A	A	N/A
_mm_cvtepi32_pd	N/A	N/A	N/A	A	N/A
_mm_cvtpd_epi32	N/A	N/A	N/A	A	N/A
_mm_cvttpd_epi32	N/A	N/A	N/A	A	N/A
_mm_cvtepi32_ps	N/A	N/A	N/A	A	N/A
_mm_cvtps_epi32	N/A	N/A	N/A	A	N/A
_mm_cvttps_epi32	N/A	N/A	N/A	A	N/A
_mm_cvtpd_ps	N/A	N/A	N/A	A	N/A
_mm_cvtps_pd	N/A	N/A	N/A	A	N/A
_mm_cvtsd_ss	N/A	N/A	N/A	A	N/A
_mm_cvtss_sd	N/A	N/A	N/A	A	N/A
_mm_cvtsd_si32	N/A	N/A	N/A	A	N/A
_mm_cvttsd_si32	N/A	N/A	N/A	A	N/A
_mm_cvtsi32_sd	N/A	N/A	N/A	A	N/A
_mm_cvtpd_pi32	N/A	N/A	N/A	A	N/A
_mm_cvttpd_pi32	N/A	N/A	N/A	A	N/A
_mm_cvtpi32_pd	N/A	N/A	N/A	A	N/A
_mm_unpackhi_pd	N/A	N/A	N/A	A	N/A
_mm_unpacklo_pd	N/A	N/A	N/A	A	N/A
_mm_unpacklo_pd	N/A	N/A	N/A	A	N/A
_mm_shuffle_pd	N/A	N/A	N/A	A	N/A
_mm_load_pd	N/A	N/A	N/A	A	N/A
_mm_load1_pd	N/A	N/A	N/A	A	N/A
_mm_loadr_pd	N/A	N/A	N/A	A	N/A
_mm_loadu_pd	N/A	N/A	N/A	A	N/A
_mm_load_sd	N/A	N/A	N/A	A	N/A
_mm_loadh_pd	N/A	N/A	N/A	A	N/A
_mm_loadl_pd	N/A	N/A	N/A	A	N/A
_mm_set_sd	N/A	N/A	N/A	A	N/A

_mm_set1_pd	N/A	N/A	N/A	A	N/A
_mm_set_pd	N/A	N/A	N/A	A	N/A
_mm_setr_pd	N/A	N/A	N/A	A	N/A
_mm_setzero_pd	N/A	N/A	N/A	A	N/A
_mm_move_sd	N/A	N/A	N/A	A	N/A
_mm_store_sd	N/A	N/A	N/A	A	N/A
_mm_store1_pd	N/A	N/A	N/A	A	N/A
_mm_store_pd	N/A	N/A	N/A	A	N/A
_mm_storeu_pd	N/A	N/A	N/A	A	N/A
_mm_storer_pd	N/A	N/A	N/A	A	N/A
_mm_storeh_pd	N/A	N/A	N/A	A	N/A
_mm_storel_pd	N/A	N/A	N/A	A	N/A
_mm_add_epi8	N/A	N/A	N/A	A	N/A
_mm_add_epi16	N/A	N/A	N/A	A	N/A
_mm_add_epi32	N/A	N/A	N/A	A	N/A
_mm_add_si64	N/A	N/A	N/A	A	N/A
_mm_add_epi64	N/A	N/A	N/A	A	N/A
_mm_adds_epi8	N/A	N/A	N/A	A	N/A
_mm_adds_epi16	N/A	N/A	N/A	A	N/A
_mm_adds_epu8	N/A	N/A	N/A	A	N/A
_mm_adds_epu16	N/A	N/A	N/A	A	N/A
_mm_avg_epu8	N/A	N/A	N/A	A	N/A
_mm_avg_epu16	N/A	N/A	N/A	A	N/A
_mm_madd_epi16	N/A	N/A	N/A	A	N/A
_mm_max_epi16	N/A	N/A	N/A	A	N/A
_mm_max_epu8	N/A	N/A	N/A	A	N/A
_mm_min_epi16	N/A	N/A	N/A	A	N/A
_mm_min_epu8	N/A	N/A	N/A	A	N/A
_mm_mulhi_epi16	N/A	N/A	N/A	A	N/A
_mm_mulhi_epu16	N/A	N/A	N/A	A	N/A
_mm_mullo_epi16	N/A	N/A	N/A	A	N/A
_mm_mul_su32	N/A	N/A	N/A	A	N/A

_mm_mul_epu32	N/A	N/A	N/A	A	N/A
_mm_sad_epu8	N/A	N/A	N/A	A	N/A
_mm_sub_epi8	N/A	N/A	N/A	A	N/A
_mm_sub_epi16	N/A	N/A	N/A	A	N/A
_mm_sub_epi32	N/A	N/A	N/A	A	N/A
_mm_sub_si64	N/A	N/A	N/A	A	N/A
_mm_sub_epi64	N/A	N/A	N/A	A	N/A
_mm_subs_epi8	N/A	N/A	N/A	A	N/A
_mm_subs_epi16	N/A	N/A	N/A	A	N/A
_mm_subs_epu8	N/A	N/A	N/A	A	N/A
_mm_subs_epu16	N/A	N/A	N/A	A	N/A
_mm_and_si128	N/A	N/A	N/A	A	N/A
_mm_andnot_si128	N/A	N/A	N/A	A	N/A
_mm_or_si128	N/A	N/A	N/A	A	N/A
_mm_xor_si128	N/A	N/A	N/A	A	N/A
_mm_slli_si128	N/A	N/A	N/A	A	N/A
_mm_slli_epi16	N/A	N/A	N/A	A	N/A
_mm_sll_epi16	N/A	N/A	N/A	A	N/A
_mm_slli_epi32	N/A	N/A	N/A	A	N/A
_mm_sll_epi32	N/A	N/A	N/A	A	N/A
_mm_slli_epi64	N/A	N/A	N/A	A	N/A
_mm_sll_epi64	N/A	N/A	N/A	A	N/A
_mm_srai_epi16	N/A	N/A	N/A	A	N/A
_mm_sra_epi16	N/A	N/A	N/A	A	N/A
_mm_srai_epi32	N/A	N/A	N/A	A	N/A
_mm_sra_epi32	N/A	N/A	N/A	A	N/A
_mm_srli_si128	N/A	N/A	N/A	A	N/A
_mm_srli_epi16	N/A	N/A	N/A	A	N/A
_mm_srl_epi16	N/A	N/A	N/A	A	N/A
_mm_srli_epi32	N/A	N/A	N/A	A	N/A
_mm_srl_epi32	N/A	N/A	N/A	A	N/A
_mm_srli_epi64	N/A	N/A	N/A	A	N/A

`_mm_srl_epi64`	N/A	N/A	N/A	A	N/A
`_mm_cmpeq_epi8`	N/A	N/A	N/A	A	N/A
`_mm_cmpeq_epi16`	N/A	N/A	N/A	A	N/A
`_mm_cmpeq_epi32`	N/A	N/A	N/A	A	N/A
`_mm_cmpgt_epi8`	N/A	N/A	N/A	A	N/A
`_mm_cmpgt_epi16`	N/A	N/A	N/A	A	N/A
`_mm_cmpgt_epi32`	N/A	N/A	N/A	A	N/A
`_mm_cmplt_epi8`	N/A	N/A	N/A	A	N/A
`_mm_cmplt_epi16`	N/A	N/A	N/A	A	N/A
`_mm_cmplt_epi32`	N/A	N/A	N/A	A	N/A
`_mm_cvtsi32_si128`	N/A	N/A	N/A	A	N/A
`_mm_cvtsi128_si32`	N/A	N/A	N/A	A	N/A
`_mm_packs_epi16`	N/A	N/A	N/A	A	N/A
`_mm_packs_epi32`	N/A	N/A	N/A	A	N/A
`_mm_packus_epi16`	N/A	N/A	N/A	A	N/A
`_mm_extract_epi16`	N/A	N/A	N/A	A	N/A
`_mm_insert_epi16`	N/A	N/A	N/A	A	N/A
`_mm_movemask_epi8`	N/A	N/A	N/A	A	N/A
`_mm_shuffle_epi32`	N/A	N/A	N/A	A	N/A
`_mm_shufflehi_epi16`	N/A	N/A	N/A	A	N/A
`_mm_shufflelo_epi16`	N/A	N/A	N/A	A	N/A
`_mm_unpackhi_epi8`	N/A	N/A	N/A	A	N/A
`_mm_unpackhi_epi16`	N/A	N/A	N/A	A	N/A
`_mm_unpackhi_epi32`	N/A	N/A	N/A	A	N/A
`_mm_unpackhi_epi64`	N/A	N/A	N/A	A	N/A
`_mm_unpacklo_epi8`	N/A	N/A	N/A	A	N/A
`_mm_unpacklo_epi16`	N/A	N/A	N/A	A	N/A
`_mm_unpacklo_epi32`	N/A	N/A	N/A	A	N/A
`_mm_unpacklo_epi64`	N/A	N/A	N/A	A	N/A
`_mm_move_epi64`	N/A	N/A	N/A	A	N/A
`_mm_movpi64_epi64`	N/A	N/A	N/A	A	N/A
`_mm_movepi64_pi64`	N/A	N/A	N/A	A	N/A

`_mm_load_si128`	N/A	N/A	N/A	A	N/A
`_mm_loadu_si128`	N/A	N/A	N/A	A	N/A
`_mm_loadl_epi64`	N/A	N/A	N/A	A	N/A
`_mm_set_epi64`	N/A	N/A	N/A	A	N/A
`_mm_set_epi32`	N/A	N/A	N/A	A	N/A
`_mm_set_epi16`	N/A	N/A	N/A	A	N/A
`_mm_set_epi8`	N/A	N/A	N/A	A	N/A
`_mm_set1_epi64`	N/A	N/A	N/A	A	N/A
`_mm_set1_epi32`	N/A	N/A	N/A	A	N/A
`_mm_set1_epi16`	N/A	N/A	N/A	A	N/A
`_mm_set1_epi8`	N/A	N/A	N/A	A	N/A
`_mm_setr_epi64`	N/A	N/A	N/A	A	N/A
`_mm_setr_epi32`	N/A	N/A	N/A	A	N/A
`_mm_setr_epi16`	N/A	N/A	N/A	A	N/A
`_mm_setr_epi8`	N/A	N/A	N/A	A	N/A
`_mm_setzero_si128`	N/A	N/A	N/A	A	N/A
`_mm_store_si128`	N/A	N/A	N/A	A	N/A
`_mm_storeu_si128`	N/A	N/A	N/A	A	N/A
`_mm_storel_epi64`	N/A	N/A	N/A	A	N/A
`_mm_maskmoveu_si128`	N/A	N/A	N/A	A	N/A
`_mm_stream_pd`	N/A	N/A	N/A	A	N/A
`_mm_stream_si128`	N/A	N/A	N/A	A	N/A
`_mm_clflush`	N/A	N/A	N/A	A	N/A
`_mm_lfence`	N/A	N/A	N/A	A	N/A
`_mm_mfence`	N/A	N/A	N/A	A	N/A
`_mm_stream_si32`	N/A	N/A	N/A	A	N/A
`_mm_pause`	N/A	N/A	N/A	A	N/A

Welcome to the Class Libraries

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

Hardware and Software Requirements

You must have the Intel® C++ Compiler version 4.0 or higher installed on your system to use the class libraries. The Intel® C++ Class Libraries are functions abstracted from the instruction extensions available on Intel processors as specified in the table that follows.

Processor Requirements for Use of Class Libraries

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

About the Classes

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

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

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

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

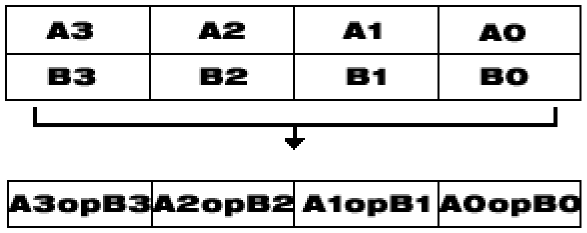
Streaming SIMD Extensions 2 intrinsics cannot be used on Itanium®-based systems. The `mmclass.h` header file includes the classes that are usable on the Itanium architecuture.

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

Details About the Libraries

The Intel® C++ Class Libraries for SIMD Operations provide a convenient interface to access the underlying instructions for processors as specified in Processor Requirements for Use of Class Libraries. These processor-instruction extensions enable parallel processing using the single instruction-multiple data (SIMD) technique as illustrated in the following figure.

SIMD Data Flow



Performing four operations with a single instruction improves efficiency by a factor of four for that particular instruction.

These new processor instructions can be implemented using assembly inlining, intrinsics, or the C++ SIMD classes. Compare the coding required to add four 32-bit floating-point values, using each of the available interfaces:

Comparison Between Inlining, Intrinsics and Class Libraries

Assembly Inlining	Intrinsics	SIMD Class Libraries
`... __m128 a,b,c; __asm { movaps xmm0,b movaps xmm1,c addps xmm0,xmm1 movaps a, xmm0 } ...`	`#include <mmintrin.h> ... __m128 a,b,c; a = _mm_add_ps(b,c); ...`	`#include <fvec.h> ... F32vec4 a,b,c; a = b +c; ...`

The table above shows an addition of two single-precision floating-point values using assembly inlining, intrinsics, and the libraries. You can see how much easier it is to code with the Intel C++ SIMD Class Libraries. Besides using fewer keystrokes and fewer lines of code, the notation is like the standard notation in C++, making it much easier to implement over other methods.

C++ Classes and SIMD Operations

The use of C++ classes for SIMD operations is based on the concept of operating on arrays, or vectors of data, in parallel. Consider the addition of two vectors, A and B, where each vector contains four elements. Using the integer vector (`Ivec`) class, the elements `A[i]` and `B[i]` from each array are summed as shown in the following example.

Typical Method of Adding Elements Using a Loop

```
short a[4], b[4], c[4];
for (i=0; i<4; i++) /* needs four iterations */
c[i] = a[i] + b[i]; /* returns c[0], c[1], c[2], c[3] *
```

The following example shows the same results using one operation with `Ivec` Classes.

SIMD Method of Adding Elements Using Ivec Classes

```
sIs16vec4 ivecA, ivecB, ivec C; /*needs one iteration */
ivecC = ivecA + ivecB; /*returns ivecC0, ivecC1, ivecC2, ivecC3 */
```

Available Classes

The Intel® C++ SIMD classes provide parallelism, which is not easily implemented using typical mechanisms of C++. The following table shows how the Intel C++ SIMD classes use the classes and libraries.

SIMD Vector Classes

Instruction Set	Class	Signedness	Data Type	Size	Elements	Header File
MMX(TM) technology (available for IA-32- and Itanium®-based systems)	`I64vec1`	unspecified	`__m64`	64	1	`ivec.h`
	`I32vec2`	unspecified	`int`	32	2	`ivec.h`
	`Is32vec2`	signed	`int`	32	2	`ivec.h`
	`Iu32vec2`	unsigned	`int`	32	2	`ivec.h`
	`I16vec4`	unspecified	`short`	16	4	`ivec.h`
	`Is16vec4`	signed	`short`	16	4	`ivec.h`
	`Iu16vec4`	unsigned	`short`	16	4	`ivec.h`
	`I8vec8`	unspecified	`char`	8	8	`ivec.h`
	`Is8vec8`	signed	`char`	8	8	`ivec.h`
	`Iu8vec8`	unsigned	`char`	8	8	`ivec.h`

Streaming SIMD Extensions (available for IA-32 and Itanium-based systems)	`F32vec4`	signed	`float`	32	4	`fvec.h`
	`F32vec1`	signed	`float`	32	1	`fvec.h`
Streaming SIMD Extensions 2 (available for IA-32-based systems only)	`F64vec2`	signed	`double`	64	2	`dvec.h`
	`I128vec1`	unspecified	`__m128i`	128	1	`dvec.h`
	`I64vec2`	unspecified	`long int`	64	4	`dvec.h`
	`Is64vec2`	signed	`long int`	64	4	`dvec.h`
	`Iu64vec2`	unsigned	`long int`	32	4	`dvec.h`
	`I32vec4`	unspecified	`int`	32	4	`dvec.h`
	`Is32vec4`	signed	`int`	32	4	`dvec.h`
	`Iu32vec4`	unsigned	`int`	32	4	`dvec.h`
	`I16vec8`	unspecified	`int`	16	8	`dvec.h`
	`Is16vec8`	signed	`int`	16	8	`dvec.h`
	`Iu16vec8`	unsigned	`int`	16	8	`dvec.h`
	`I8vec16`	unspecified	`char`	8	16	`dvec.h`
	`Is8vec16`	signed	`char`	8	16	`dvec.h`
	`Iu8vec16`	unsigned	`char`	8	16	`dvec.h`

Most classes contain similar functionality for all data types and are represented by all available intrinsics. However, some capabilities do not translate from one data type to another without suffering from poor performance, and are therefore excluded from individual classes.

Note

Intrinsics that take immediate values and cannot be expressed easily in classes are not implemented. (For example, `_mm_shuffle_ps`, `_mm_shuffle_pi16`, `_mm_extract_pi16`, `_mm_insert_pi16`).

Access to Classes Using Header Files

The required class header files are installed in the include directory with the Intel® C++ Compiler. To enable the classes, use the `#include` directive in your program file as shown in the table that follows.

Include Directives for Enabling Classes

Instruction Set Extension	Include Directive
MMX Technology	`#include <ivec.h>`
Streaming SIMD Extensions	`#include <fvec.h>`
Streaming SIMD Extensions 2	`#include <dvec.h>`

Each succeeding file from the top down includes the preceding class. You only need to include `fvec.h` if you want to use both the `Ivec` and `Fvec` classes. Similarly, to use all the classes including those for the Streaming SIMD Extensions 2, you need only to include the `dvec.h` file.

Usage Precautions

When using the C++ classes, you should follow some general guidelines. More detailed usage rules for each class are listed in Integer Vector Classes, and Floating-point Vector Classes.

Clear MMX Registers

If you use both the `Ivec` and `Fvec` classes at the same time, your program could mix MMX instructions, called by `Ivec` classes, with Intel x87 architecture floating-point instructions, called by `Fvec` classes. Floating-point instructions exist in the following `Fvec` functions:

- `fvec` constructors
- debug functions (`cout` and element access)
- `rsqrt_nr`

Note

MMX registers are aliased on the floating-point registers, so you should clear the MMX state with the EMMS instruction intrinsic before issuing an x87 floating-point instruction, as in the following example.

`ivecA = ivecA & ivecB;`	`/* Ivec logical operation that uses MMX instructions */`
`empty ();`	`/* clear state */`
`cout << f32vec4a;`	`/* F32vec4 operation that uses x87 floating-point instructions */`

Caution

Failure to clear the MMX registers can result in incorrect execution or poor performance due to an incorrect register state.

Follow EMMS Instruction Guidelines

Intel strongly recommends that you follow the guidelines for using the EMMS instruction. Refer to this topic before coding with the `Ivec` classes.

Capabilities

The fundamental capabilities of each C++ SIMD class include:

- Computation
- Horizontal data motion
- Branch compression/elimination
- Caching hints

Understanding each of these capabilities and how they interact is crucial to achieving desired results.

Computation

The SIMD C++ classes contain vertical operator support for most arithmetic operations, including shifting and saturation.

Computation operations include: `+`, `-`, `*`, `/`, reciprocal (`rcp` and `rcp_nr`), square root (`sqrt`), reciprocal square root (`rsqrt` and `rsqrt_nr`).

Operations `rcp` and `rsqrt` are new approximating instructions with very short latencies that produce results with at least 12 bits of accuracy. Operations `rcp_nr` and `rsqrt_nr` use software refining techniques to enhance the accuracy of the approximations, with a minimal impact on performance. (The "`nr`" stands for Newton-Raphson, a mathematical technique for improving performance using an approximate result.)

Horizontal Data Support

The C++ SIMD classes provide horizontal support for some arithmetic operations. The term "horizontal" indicates computation across the elements of one vector, as opposed to the vertical, element-by-element operations on two different vectors.

The `add_horizontal`, `unpack_low` and `pack_sat` functions are examples of horizontal data support. This support enables certain algorithms that cannot exploit the full potential of SIMD instructions.

Shuffle intrinsics are another example of horizontal data flow. Shuffle intrinsics are not expressed in the C++ classes due to their immediate arguments. However, the C++ class implementation enables you to mix shuffle intrinsics with the other C++ functions. For example:

```
F32vec4 fveca, fvecb, fvecd;
fveca += fvecb;
fvecd = _mm_shuffle_ps(fveca,fvecb,0);
```

Typically every instruction with horizontal data flow contains some inefficiency in the implementation. If possible, implement your algorithms without using the horizontal capabilities.

Branch Compression/Elimination

Branching in SIMD architectures can be complicated and expensive, possibly resulting in poor predictability and code expansion. The SIMD C++ classes provide functions to eliminate branches, using logical operations, max and min functions, conditional selects, and compares. Consider the following example:

```
short a[4], b[4], c[4];
for (i=0; i<4; i++)
c[i] = a[i] > b[i] ? a[i] : b[i];
```

This operation is independent of the value of `i`. For each `i`, the result could be either `A` or `B` depending on the actual values. A simple way of removing the branch altogether is to use the `select_gt` function, as follows:

```
Is16vec4 a, b, c
c = select_gt(a, b, a, b)
```

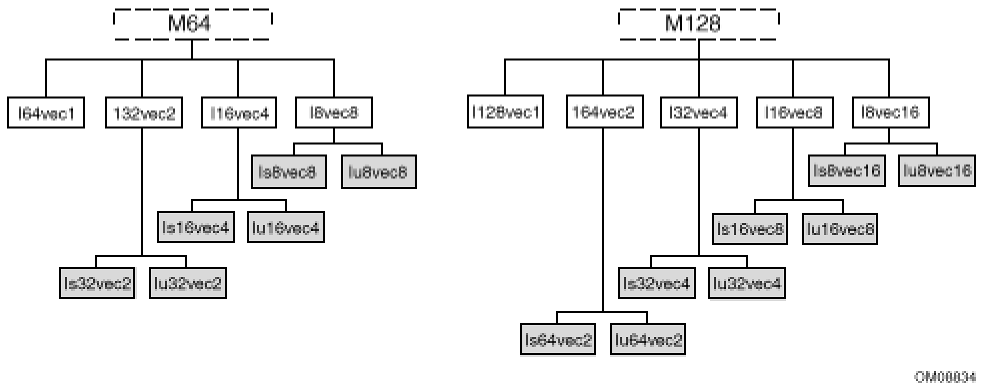
Caching Hints

Streaming SIMD Extensions provide prefetching and streaming hints. Prefetching data can minimize the effects of memory latency. Streaming hints allow you to indicate that certain data should not be cached. This results in higher performance for data that should be cached.

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



The `M64` and `M128` classes define the `__m64` and `__m128i` data types from which the rest of the `Ivec` classes are derived. The first generation of child classes are derived based solely on bit sizes of 128, 64, 32, 16, and 8 respectively for the `I128vec1`, `I64vec1`, `164vec2`, `I32vec2`, `I32vec4`, `I16vec4`, `I16vec8`, `I8vec16`, and `I8vec8` classes. The latter seven of the these classes require specification of signedness and saturation.

Caution

Do not intermix the `M64` and `M128` data types. You will get unexpected behavior if you do.

The signedness is indicated by the `s` and `u` in the class names:

```
Is64vec2
Iu64vec2
Is32vec4
Iu32vec4
Is16vec8
Iu16vec8
Is8vec16
Iu8vec16
Is32vec2
Iu32vec2
Is16vec4
Iu16vec4
Is8vec8
Iu8vec8
```

Terms, Conventions, and Syntax

The following are special terms and syntax used in this chapter to describe functionality of the classes with respect to their associated operations.

Ivec Class Syntax Conventions

The name of each class denotes the data type, signedness, bit size, number of elements using the following generic format:

```
<type><signedness><bits>vec<elements>
```

```
{ F | I } { s | u } { 64 | 32 | 16 | 8 } vec { 8 | 4 | 2 | 1 }
```

where

`type`	indicates floating point (`F`) or integer (`I`)
`signedness`	indicates signed (`s`) or unsigned (`u`). For the Ivec class, leaving this field blank indicates an intermediate class. There are no unsigned Fvec classes, therefore for the Fvec classes, this field is blank.
`bits`	specifies the number of bits per element
`elements`	specifies the number of elements

Special Terms and Conventions

The following terms are used to define the functionality and characteristics of the classes and operations defined in this manual.

- **Nearest Common Ancestor** -- This is the intermediate or parent class of two classes of the same size. For example, the nearest common ancestor of `Iu8vec8` and `Is8vec8` is `I8vec8`. Also, the nearest common ancestor between `Iu8vec8` and `I16vec4` is `M64`.
- **Casting** -- Changes the data type from one class to another. When an operation uses different data types as operands, the return value of the operation must be assigned to a single data type. Therefore, one or more of the data types must be converted to a required data type. This conversion is known as a typecast. Sometimes, typecasting is automatic, other times you must use special syntax to explicitly typecast it yourself.
- **Operator Overloading** -- This is the ability to use various operators on the same user-defined data type of a given class. Once you declare a variable, you can add, subtract, multiply, and perform a range of operations. Each family of classes accepts a specified range of operators, and must comply by rules and restrictions regarding typecasting and operator overloading as defined in the header files. The following table shows the notation used in this documention to address typecasting, operator overloading, and other rules.

Class Syntax Notation Conventions

Class Name	Description
`I[s|u][N]vec[N]`	Any value except `I128vec1` nor `I64vec1`
`I64vec1`	`__m64` data type
`I[s|u]64vec2`	two 64-bit values of any signedness
`I[s|u]32vec4`	four 32-bit values of any signedness
`I[s|u]8vec16`	eight 16-bit values of any signedness
`I[s|u]16vec8`	sixteen 8-bit values of any signedness
`I[s|u]32vec2`	two 32-bit values of any signedness
`I[s|u]16vec4`	four 16-bit values of any signedness
`I[s|u]8vec8`	eight 8-bit values of any signedness

Rules for Operators

To use operators with the `Ivec` classes you must use one of the following three syntax conventions:

`[ Ivec_Class ] R = [ Ivec_Class ] A [ operator ][ Ivec_Class ] B`

Example 1: `I64vec1 R = I64vec1 A & I64vec1 B;`

`[ Ivec_Class ] R =[ operator ] ([ Ivec_Class ] A,[ Ivec_Class ] B)`

Example 2: `I64vec1 R = andnot(I64vec1 A, I64vec1 B);`

`[ Ivec_Class ] R [ operator ]= [ Ivec_Class ] A`

Example 3: `I64vec1 R &= I64vec1 A;`

[`operator`]an operator (for example, &, |, or ^)

[`Ivec_Class`] an `Ivec` class

R, A, B variables declared using the pertinent `Ivec` classes

The table that follows shows automatic and explicit sign and size typecasting. "Explicit" means that it is illegal to mix different types without an explicit typecasting. "Automatic" means that you can mix types freely and the compiler will do the typecasting for you.

Summary of Rules Major Operators

Operators	Sign Typecasting	Size Typecasting	Other Typecasting Requirements
Assignment	N/A	N/A	N/A
Logical	Automatic	Automatic (to left)	Explicit typecasting is required for different types used in non-logical expressions on the right side of the assignment. See Syntax Usage for Logical Operators example.
Addition and Subtraction	Automatic	Explicit	N/A
Multiplication	Automatic	Explicit	N/A
Shift	Automatic	Explicit	Casting Required to ensure arithmetic shift.
Compare	Automatic	Explicit	Explicit casting is required for signed classes for the less-than or greater-than operations.
Conditional Select	Automatic	Explicit	Explicit casting is required for signed classes for less-than or greater-than operations.

Data Declaration and Initialization

The following table shows literal examples of constructor declarations and data type initialization for all class sizes. All values are initialized with the most significant element on the left and the least significant to the right.

Declaration and Initialization Data Types for Ivec Classes

Operation	Class	Syntax
Declaration	`M128`	`I128vec1 A; Iu8vec16 A;`
Declaration	`M64`	`I64vec1 A; Iu8vec16 A;`
`__m128` Initialization	`M128`	`I128vec1 A(__m128 m); Iu16vec8(__m128 m);`
`__m64` Initialization	`M64`	`I64vec1 A(__m64 m);Iu8vec8 A(__m64 m);`
`__int64` Initialization	`M64`	`I64vec1 A = __int64 m; Iu8vec8 A =__int64 m;`
`int i` Initialization	`M64`	`I64vec1 A = int i; Iu8vec8 A = int i;`

`int` initialization	`I32vec2`	`I32vec2 A(int A1, int A0);`
		`Is32vec2 A(signed int A1, signed int A0);`
		`Iu32vec2 A(unsigned int A1, unsigned int A0);`

`int` Initialization	`I32vec4`	`I32vec4 A(short A3, short A2, short A1, short A0);`
		`Is32vec4 A(signed short A3, ..., signed short A0);`
		`Iu32vec4 A(unsigned short A3, ..., unsigned short A0);`

`short int` Initialization	`I16vec4`	`I16vec4 A(short A3, short A2, short A1, short A0);`
		`Is16vec4 A(signed short A3, ..., signed short A0);`
		`Iu16vec4 A(unsigned short A3, ..., unsigned short A0);`

`short int` Initialization	`I16vec8`	`I16vec8 A(short A7, short A6, ..., short A1, short A0);`
		`Is16vec8 A(signed A7, ..., signed short A0);`
		`Iu16vec8 A(unsigned short A7, ..., unsigned short A0);`

`char` Initialization	`I8vec8`	`I8vec8 A(char A7, char A6, ..., char A1, char A0);`
		`Is8vec8 A(signed char A7, ..., signed char A0);`
		`Iu8vec8 A(unsigned char A7, ..., unsigned char A0);`

`char` Initialization	`I8vec16`	`I8vec16 A(char A15, ..., char A0);`
		`Is8vec16 A(signed char A15, ..., signed char A0);`
		`Iu8vec16 A(unsigned char A15, ..., unsigned char A0);`

Assignment Operator

Any `Ivec` object can be assigned to any other `Ivec` object; conversion on assignment from one `Ivec` object to another is automatic.

Assignment Operator Examples

```
Is16vec4 A;

Is8vec8 B;

I64vec1 C;

A = B; /* assign Is8vec8 to Is16vec4 */

B = C; /* assign I64vec1 to Is8vec8 */

B = A & C; /* assign M64 result of '&' to Is8vec8 */
```

Logical Operators

The logical operators use the symbols and intrinsics listed in the following table.

Bitwise Operation	Operator Symbols		Syntax Usage		Corresponding Intrinsic
	Standard	**w/assign**	**Standard**	**w/assign**	
AND	&	&=	R = A & B	R &= A	`_mm_and_si64` `_mm_and_si128`
OR	|	|=	R = A | B	R |= A	`_mm_and_si64` `_mm_and_si128`
XOR	^	^=	R = A^B	R ^= A	`_mm_and_si64` `_mm_and_si128`
ANDNOT	andnot	N/A	R = A andnot B	N/A	`_mm_and_si64` `_mm_and_si128`

Logical Operators and Miscellaneous Exceptions.

```
/* A and B converted to M64. Result assigned to Iu8vec8.*/

I64vec1 A;
Is8vec8 B;
Iu8vec8 C;
C = A & B;

/* Same size and signedness operators return the nearest common ancestor.*/

I32vec2 R = Is32vec2 A ^ Iu32vec2 B;

/* A&B returns M64, which is cast to Iu8vec8.*/

C = Iu8vec8(A&B)+ C;
```

When A and B are of the same class, they return the same type. When A and B are of different classes, the return value is the return type of the nearest common ancestor.

The logical operator returns values for combinations of classes, listed in the following tables, apply when A and B are of different classes.

Ivec Logical Operator Overloading

Return (R)	AND	OR	XOR	NAND	A Operand	B Operand
`I64vec1 R`	&	|	^	andnot	`I[s|u]64vec2 A`	`I[s|u]64vec2 B`
`I64vec2 R`	&	|	^	andnot	`I[s|u]64vec2 A`	`I[s|u]64vec2 B`
`I32vec2 R`	&	|	^	andnot	`I[s|u]32vec2 A`	`I[s|u]32vec2 B`
`I32vec4 R`	&	|	^	andnot	`I[s|u]32vec4 A`	`I[s|u]32vec4 B`
`I16vec4 R`	&	|	^	andnot	`I[s|u]16vec4 A`	`I[s|u]16vec4 B`
`I16vec8 R`	&	|	^	andnot	`I[s|u]16vec8 A`	`I[s|u]16vec8 B`
`I8vec8 R`	&	|	^	andnot	`I[s|u]8vec8 A`	`I[s|u]8vec8 B`
`I8vec16 R`	&	|	^	andnot	`I[s|u]8vec16 A`	`I[s|u]8vec16 B`

For logical operators with assignment, the return value of `R` is always the same data type as the pre-declared value of R as listed in the table that follows.

Ivec Logical Operator Overloading with Assignment

Return Type	Left Side (R)	AND	OR	XOR	Right Side (Any Ivec Type)
`I128vec1`	`I128vec1 R`	&=	|=	^=	`I[s|u][N]vec[N] A;`
`I64vec1`	`I64vec1 R`	&=	|=	^=	`I[s|u][N]vec[N] A;`
`I64vec2`	`I64vec2 R`	&=	|=	^=	`I[s|u][N]vec[N] A;`
`I[x]32vec4`	`I[x]32vec4 R`	&=	|=	^=	`I[s|u][N]vec[N] A;`
`I[x]32vec2`	`I[x]32vec2 R`	&=	|=	^=	`I[s|u][N]vec[N] A;`
`I[x]16vec8`	`I[x]16vec8 R`	&=	|=	^=	`I[s|u][N]vec[N] A;`
`I[x]16vec4`	`I[x]16vec4 R`	&=	|=	^=	`I[s|u][N]vec[N] A;`
`I[x]8vec16`	`I[x]8vec16 R`	&=	|=	^=	`I[s|u][N]vec[N] A;`
`I[x]8vec8`	`I[x]8vec8 R`	&=	|=	^=	`I[s|u][N]vec[N] A;`

Addition and Subtraction Operators

The addition and subtraction operators return the class of the nearest common ancestor when the right-side operands are of different signs. The following code provides examples of usage and miscellaneous exceptions.

Syntax Usage for Addition and Subtraction Operators

```
/* Return nearest common ancestor type, I16vec4 */

Is16vec4 A;

Iu16vec4 B;

I16vec4 C;

C = A + B;

/* Returns type left-hand operand type */

Is16vec4 A;

Iu16vec4 B;

A += B;

B -= A;

/* Explicitly convert B to Is16vec4 */

Is16vec4 A,C;

Iu32vec24 B;

C = A + C;

C = A + (Is16vec4)B;
```

Addition and Subtraction Operators with Corresponding Intrinsics

Operation	Symbols	Syntax	Corresponding Intrinsics

Addition	+
	+=
		R = A + B
		R += A
			`_mm_add_epi64`
			`_mm_add_epi32`
			`_mm_add_epi16`
			`_mm_add_epi8`
			`_mm_add_pi32`
			`_mm_add_pi16`
			`_mm_add_pi8`

Subtraction	-
	-=
		R = A - B
		R -= A
			`_mm_sub_epi64`
			`_mm_sub_epi32`
			`_mm_sub_epi16`
			`_mm_sub_epi8`
			`_mm_sub_pi32`
			`_mm_sub_pi16`
			`_mm_sub_pi8`

The following table lists addition and subtraction return values for combinations of classes when the right side operands are of different signedness. The two operands must be the same size, otherwise you must explicitly indicate the typecasting.

Addition and Subtraction Operator Overloading

Return Value	Available Operators		Right Side Operands	
R	**Add**	**Sub**	**A**	**B**
`I64vec2 R`	+	-	`I[s|u]64vec2 A`	`I[s|u]64vec2 B`
`I32vec4 R`	+	-	`I[s|u]32vec4 A`	`I[s|u]32vec4 B`
`I32vec2 R`	+	-	`I[s|u]32vec2 A`	`I[s|u]32vec2 B`
`I16vec8 R`	+	-	`I[s|u]16vec8 A`	`I[s|u]16vec8 B`
`I16vec4 R`	+	-	`I[s|u]16vec4 A`	`I[s|u]16vec4 B`
`I8vec8 R`	+	-	`I[s|u]8vec8 A`	`I[s|u]8vec8 B`
`I8vec16 R`	+	-	`I[s|u]8vec2 A`	`I[s|u]8vec16 B`

The following table shows the return data type values for operands of the addition and subtraction operators with assignment. The left side operand determines the size and signedness of the return value. The right side operand must be the same size as the left operand; otherwise, you must use an explicit typecast.

Addition and Subtraction with Assignment

Return Value (R)	Left Side (R)	Add	Sub	Right Side (A)
`I[x]32vec4`	`I[x]32vec2 R`	+=	-=	`I[s|u]32vec4 A;`
`I[x]32vec2 R`	`I[x]32vec2 R`	+=	-=	`I[s|u]32vec2 A;`
`I[x]16vec8`	`I[x]16vec8`	+=	-=	`I[s|u]16vec8 A;`
`I[x]16vec4`	`I[x]16vec4`	+=	-=	`I[s|u]16vec4 A;`
`I[x]8vec16`	`I[x]8vec16`	+=	-=	`I[s|u]8vec16 A;`
`I[x]8vec8`	`I[x]8vec8`	+=	-=	`I[s|u]8vec8 A;`

Multiplication Operators

The multiplication operators can only accept and return data types from the `I[s|u]16vec4` or `I[s|u]16vec8` classes, as shown in the following example.

Syntax Usage for Multiplication Operators

```
/* Explicitly convert B to Is16vec4 */

Is16vec4 A,C;

Iu32vec2 B;

C = A * C;

C = A * (Is16vec4)B;

/* Return nearest common ancestor type, I16vec4 */

Is16vec4 A;

Iu16vec4 B;

I16vec4 C;

C = A + B;

/* The mul_high and mul_add functions take Is16vec4 data only */

Is16vec4 A,B,C,D;

C = mul_high(A,B);

D = mul_add(A,B);
```

Multiplication Operators with Corresponding Intrinsics

Symbols		Syntax Usage	Intrinsic
*	*=	R = A * B R *= A	`_mm_mullo_pi16` `_mm_mullo_epi16`
`mul_high`	N/A	`R = mul_high(A, B)`	`_mm_mulhi_pi16` `_mm_mulhi_epi16`
`mul_add`	N/A	`R = mul_high(A, B)`	`_mm_madd_pi16` `_mm_madd_epi16`

The multiplication return operators always return the nearest common ancestor as listed in the table that follows. The two operands must be 16 bits in size, otherwise you must explicitly indicate typecasting.

Multiplication Operator Overloading

R	Mul	A	B
`I16vec4 R`	`*`	`I[s|u]16vec4 A`	`I[s|u]16vec4 B`
`I16vec8 R`	`*`	`I[s|u]16vec8 A`	`I[s|u]16vec8 B`
`Is16vec4 R`	`mul_add`	`Is16vec4 A`	`Is16vec4 B`
`Is16vec8`	`mul_add`	`Is16vec8 A`	`Is16vec8 B`
`Is32vec2 R`	`mul_high`	`Is16vec4 A`	`Is16vec4 B`
`Is32vec4 R`	`mul_high`	`s16vec8 A`	`Is16vec8 B`

The following table shows the return values and data type assignments for operands of the multiplication operators with assignment. All operands must be 16 bytes in size. If the operands are not the right size, you must use an explicit typecast.

Multiplication with Assignment

Return Value (R)	Left Side (R)	Mul	Right Side (A)
`I[x]16vec8`	`I[x]16vec8`	`*=`	`I[s|u]16vec8 A;`
`I[x]16vec4`	`I[x]16vec4`	`*=`	`I[s|u]16vec4 A;`

Shift Operators

The right shift argument can be any integer or Ivec value, and is implicitly converted to a M64 data type. The first or left operand of a << can be of any type except `I[s|u]8vec[8|16]` .

Example Syntax Usage for Shift Operators

```
/* Automatic size and sign conversion */

Is16vec4 A,C;

Iu32vec2 B;

C = A;

/* A&B returns I16vec4, which must be cast to Iu16vec4

to ensure logical shift, not arithmetic shift */

Is16vec4 A, C;

Iu16vec4 B, R;

R = (Iu16vec4)(A & B) C;

/* A&B returns I16vec4, which must be cast to Is16vec4

to ensure arithmetic shift, not logical shift */

R = (Is16vec4)(A & B) C;
```

Shift Operators with Corresponding Intrinsics

Operation	Symbols	Syntax Usage	Intrinsic

Shift Left	<<
	&=
		R = A << B
		R &= A
			`_mm_sll_si64`
			`_mm_slli_si64`
			`_mm_sll_pi32`
			`_mm_slli_pi32`
			`_mm_sll_pi16`
			`_mm_slli_pi16`

Shift Right	>>	R = A >> B
		R >>= A
			`_mm_srl_si64`
			`_mm_srli_si64`
			`_mm_srl_pi32`
			`_mm_srli_pi32`
			`_mm_srl_pi16`
			`_mm_srli_pi16`
			`_mm_sra_pi32`
			`_mm_srai_pi32`
			`_mm_sra_pi16`
			`_mm_srai_pi16`

Right shift operations with signed data types use arithmetic shifts. All unsigned and intermediate classes correspond to logical shifts. The table below shows how the return type is determined by the first argument type.

Shift Operator Overloading

Operation	R	Right Shift		Left Shift		A	B
Logical	`I64vec1`	>>	>>=	<<	<<=	`I64vec1 A;`	`I64vec1 B;`
Logical	`I32vec2`	>>	>>=	<<	<<=	`I32vec2 A`	`I32vec2 B;`
Arithmetic	`Is32vec2`	>>	>>=	<<	<<=	`Is32vec2 A`	`I[s|u][N]vec[N] B;`
Logical	`Iu32vec2`	>>	>>=	<<	<<=	`Iu32vec2 A`	`I[s|u][N]vec[N] B;`
Logical	`I16vec4`	>>	>>=	<<	<<=	`I16vec4 A`	`I16vec4 B`
Arithmetic	`Is16vec4`	>>	>>=	<<	<<=	`Is16vec4 A`	`I[s|u][N]vec[N] B;`
Logical	`Iu16vec4`	>>	>>=	<<	<<=	`Iu16vec4 A`	`I[s|u][N]vec[N] B;`

Comparison Operators

The equality and inequality comparison operands can have mixed signedness, but they must be of the same size. The comparison operators for less-than and greater-than must be of the same sign and size.

Example of Syntax Usage for Comparison Operator

```
/* The nearest common ancestor is returned for compare

for equal/not-equal operations */

Iu8vec8 A;

Is8vec8 B;

I8vec8 C;

C = cmpneq(A,B);

/* Type cast needed for different-sized elements for

equal/not-equal comparisons */

Iu8vec8 A, C;

Is16vec4 B;

C = cmpeq(A,(Iu8vec8)B);

/* Type cast needed for sign or size differences for

less-than and greater-than comparisons */

Iu16vec4 A;

Is16vec4 B, C;

C = cmpge((Is16vec4)A,B);

C = cmpgt(B,C);
```

Inequality Comparison Symbols and Corresponding Intrinsics

Compare For:	Operators	Syntax	Intrinsic	

Equality	`cmpeq`	`R = cmpeq(A, B)`	`_mm_cmpeq_pi32`
			`_mm_cmpeq_pi16`
			`_mm_cmpeq_pi8`	

Inequality	`cmpneq`	`R = cmpneq(A, B)`	`_mm_cmpeq_pi32`
			`_mm_cmpeq_pi16`
			`_mm_cmpeq_pi8`
				`_mm_andnot_si64`

Greater Than	`cmpgt`	`R = cmpgt(A, B)`	`_mm_cmpgt_pi32`
			`_mm_cmpgt_pi16`
			`_mm_cmpgt_pi8`	

Greater Than or Equal To	`cmpge`	`R = cmpge(A, B)`	`_mm_cmpgt_pi32`
			`_mm_cmpgt_pi16`
			`_mm_cmpgt_pi8`
				`_mm_andnot_si64`

Less Than	`cmplt`	`R = cmplt(A, B)`	`_mm_cmpgt_pi32`
			`_mm_cmpgt_pi16`
			`_mm_cmpgt_pi8`	

Less Than or Equal To	`cmple`	`R = cmple(A, B)`	`_mm_cmpgt_pi32`
			`_mm_cmpgt_pi16`
			`_mm_cmpgt_pi8`
				`_mm_andnot_si64`

Comparison operators have the restriction that the operands must be the size and sign as listed in the Compare Operator Overloading table.

Compare Operator Overloading

R	Comparison	A	B
`I32vec2 R`	`cmpeq` `cmpne`	`I[s|u]32vec2 B`	`I[s|u]32vec2 B`
`I16vec4 R`		`I[s|u]16vec4 B`	`I[s|u]16vec4 B`
`I8vec8 R`		`I[s|u]8vec8 B`	`I[s|u]8vec8 B`

`I32vec2 R`	`cmpgt`
	`cmpge`
	`cmplt`
	`cmple`
		`Is32vec2 B`	`Is32vec2 B`

`I16vec4 R`		`Is16vec4 B`	`Is16vec4 B`
`I8vec8 R`		`Is8vec8 B`	`Is8vec8 B`

Conditional Select Operators

For conditional select operands, the third and fourth operands determine the type returned. Third and fourth operands with same size, but different signedness, return the nearest common ancestor data type.

Conditional Select Syntax Usage

```
/* Return the nearest common ancestor data type if third and fourth
operands are of the same size, but different signs */

I16vec4 R = select_neq(Is16vec4, Is16vec4, Is16vec4, Iu16vec4);

/* Conditional Select for Equality */

R0 := (A0 == B0) ? C0 : D0;

R1 := (A1 == B1) ? C1 : D1;

R2 := (A2 == B2) ? C2 : D2;

R3 := (A3 == B3) ? C3 : D3;

/* Conditional Select for Inequality */

R0 := (A0 != B0) ? C0 : D0;

R1 := (A1 != B1) ? C1 : D1;

R2 := (A2 != B2) ? C2 : D2;

R3 := (A3 != B3) ? C3 : D3;
```

Conditional Select Symbols and Corresponding Intrinsics

Conditional Select For:	Operators	Syntax	Corresponding Intrinsic	Additional Intrinsic (Applies to All)
Equality	`select_eq`	R = `select_eq` (A, B, C, D)	`_mm_cmpeq_pi32` `_mm_cmpeq_pi16` `_mm_cmpeq_pi8`	`_mm_and_si64` `_mm_or_si64` `_mm_andnot_si64`
Inequality	`select_neq`	R = `select_neq`(A, B, C, D)	`_mm_cmpeq_pi32` `_mm_cmpeq_pi16` `_mm_cmpeq_pi8`	
Greater Than	`select_gt`	R = `select_gt` (A, B, C, D)	`_mm_cmpgt_pi32` `_mm_cmpgt_pi16` `_mm_cmpgt_pi8`	
Greater Than or Equal To	`select_ge`	R = `select_gt` (A, B, C, D)	`_mm_cmpge_pi32` `_mm_cmpge_pi16` `_mm_cmpge_pi8`	

Less Than	`select_lt`	R = `select_lt` (A, B, C, D)	`_mm_cmplt_pi32` `_mm_cmplt_pi16` `_mm_cmplt_pi8`	
Less Than or Equal To	`select_le`	R = `select_le` (A, B, C, D)	`_mm_cmple_pi32` `_mm_cmple_pi16` `_mm_cmple_pi8`	

All conditional select operands must be of the same size. The return data type is the nearest common ancestor of operands `C` and `D`. For conditional select operations using greater-than or less-than operations, the first and second operands must be signed as listed in the table that follows.

Conditional Select Operator Overloading

R	Comparison	A and B	C	D
`I32vec2 R`	`select_eq` `select_ne`	`I[s|u]32vec2`	`I[s|u]32vec2`	`I[s|u]32vec2`
`I16vec4 R`		`I[s|u]16vec4`	`I[s|u]16vec4`	`I[s|u]16vec4`
`I8vec8 R`		`I[s|u]8vec8`	`I[s|u]8vec8`	`I[s|u]8vec8`
`I32vec2 R`	`select_gt` `select_ge` `select_lt` `select_le`	`Is32vec2`	`Is32vec2`	`Is32vec2`
`I16vec4 R`		`Is16vec4`	`Is16vec4`	`Is16vec4`
`I8vec8 R`		`Is8vec8`	`Is8vec8`	`Is8vec8`

The table below shows the mapping of return values from `R0` to `R7` for any number of elements. The same return value mappings also apply when there are fewer than four return values.

Conditional Select Operator Return Value Mapping

Return Value	A and B Operands								C and D operands
	A0	Available Operators						B0	
R0:=	A0	==	!=	>	>=	<	<=	B0	? C0 : D0;
R1:=	A0	==	!=	>	>=	<	<=	B0	? C1 : D1;
R2:=	A0	==	!=	>	>=	<	<=	B0	? C2 : D2;
R3:=	A0	==	!=	>	>=	<	<=	B0	? C3 : D3;
R4:=	A0	==	!=	>	>=	<	<=	B0	? C4 : D4;
R5:=	A0	==	!=	>	>=	<	<=	B0	? C5 : D5;
R6:=	A0	==	!=	>	>=	<	<=	B0	? C6 : D6;
R7:=	A0	==	!=	>	>=	<	<=	B0	? C7 : D7;

Debug

The debug operations do not map to any compiler intrinsics for MMX(TM) instructions. They are provided for debugging programs only. Use of these operations may result in loss of performance, so you should not use them outside of debugging.

Output

The four 32-bit values of A are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is32vec4 A;

cout << Iu32vec4 A;

cout << hex << Iu32vec4 A; /* print in hex format */

"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The two 32-bit values of A are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is32vec2 A;

cout << Iu32vec2 A;

cout << hex << Iu32vec2 A; /* print in hex format */

"[1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The eight 16-bit values of A are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is16vec8 A;

cout << Iu16vec8 A;

cout << hex << Iu16vec8 A; /* print in hex format */

"[7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The four 16-bit values of A are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is16vec4 A;

cout << Iu16vec4 A;
cout << hex << Iu16vec4 A; /* print in hex format */
```

```
"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```
Corresponding Intrinsics: none

The sixteen 8-bit values of A are placed in the output buffer and printed in the following format (default is decimal):

```
cout << Is8vec16 A; cout << Iu8vec16 A; cout << hex << Iu8vec8 A;
```

```
/* print in hex format instead of decimal*/
```

```
"[15]:A15 [14]:A14 [13]:A13 [12]:A12 [11]:A11 [10]:A10 [9]:A9 [8]:A8
[7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1 [0]:A0"
```
Corresponding Intrinsics: none

The eight 8-bit values of A are placed in the output buffer and printed in the following format (default is decimal):

```
cout << Is8vec8 A; cout << Iu8vec8 A;cout << hex << Iu8vec8 A;
```

```
/* print in hex format instead of decimal*/
```

```
"[7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1 [0]:A0"
```
Corresponding Intrinsics: none

Element Access Operators

```
int R = Is64vec2 A[i];

unsigned int R = Iu64vec2 A[i];

int R = Is32vec4 A[i];

unsigned int R = Iu32vec4 A[i];

int R = Is32vec2 A[i];

unsigned int R = Iu32vec2 A[i];

short R = Is16vec8 A[i];

unsigned short R = Iu16vec8 A[i];

short R = Is16vec4 A[i];

unsigned short R = Iu16vec4 A[i];

signed char R = Is8vec16 A[i];
```

```
unsigned char R = Iu8vec16 A[i];
```

```
signed char R = Is8vec8 A[i];
```

```
unsigned char R = Iu8vec8 A[i];
```

Access and read element `i` of A. If DEBUG is enabled and the user tries to access an element outside of A, a diagnostic message is printed and the program aborts.

Corresponding Intrinsics: none

Element Assignment Operators

```
Is64vec2 A[i] = int R;
```

```
Is32vec4 A[i] = int R;
```

```
Iu32vec4 A[i] = unsigned int R;
```

```
Is32vec2 A[i] = int R;
```

```
Iu32vec2 A[i] = unsigned int R;
```

```
Is16vec8 A[i] = short R;
```

```
Iu16vec8 A[i] = unsigned short R;
```

```
Is16vec4 A[i] = short R;
```

```
Iu16vec4 A[i] = unsigned short R;
```

```
Is8vec16 A[i] = signed char R;
```

```
Iu8vec16 A[i] = unsigned char R;
```

```
Is8vec8 A[i] = signed char R;
```

```
Iu8vec8 A[i] = unsigned char R;
```

Assign R to element i of A. If `DEBUG` is enabled and the user tries to assign a value to an element outside of A, a diagnostic message is printed and the program aborts.

Corresponding Intrinsics: none

Unpack Operators

Interleave the 64-bit value from the high half of A with the 64-bit value from the high half of B.

```
I364vec2 unpack_high(I64vec2 A, I64vec2 B);

Is64vec2 unpack_high(Is64vec2 A, Is64vec2 B);

Iu64vec2 unpack_high(Iu64vec2 A, Iu64vec2 B);

R0 = A1;
R1 = B1;
```

Corresponding intrinsic: `_mm_unpackhi_epi64`

Interleave the two 32-bit values from the high half of A with the two 32-bit values from the high half of B .

```
I32vec4 unpack_high(I32vec4 A, I32vec4 B);

Is32vec4 unpack_high(Is32vec4 A, Is32vec4 B);

Iu32vec4 unpack_high(Iu32vec4 A, Iu32vec4 B);

R0 = A1;
R1 = B1;
R2 = A2;
R3 = B2;
```

Corresponding intrinsic: `_mm_unpackhi_epi32`

Interleave the 32-bit value from the high half of A with the 32-bit value from the high half of B.

```
I32vec2 unpack_high(I32vec2 A, I32vec2 B);

Is32vec2 unpack_high(Is32vec2 A, Is32vec2 B);

Iu32vec2 unpack_high(Iu32vec2 A, Iu32vec2 B);

R0 = A1;
R1 = B1;
```

Corresponding intrinsic: `_mm_unpackhi_pi32`

Interleave the four 16-bit values from the high half of A with the two 16-bit values from the high half of B.

```
I16vec8 unpack_high(I16vec8 A, I16vec8 B);

Is16vec8 unpack_high(Is16vec8 A, Is16vec8 B);
```

```
Iu16vec8 unpack_high(Iu16vec8 A, Iu16vec8 B);
```

```
R0 = A2;
R1 = B2;
R2 = A3;
R3 = B3;
```

Corresponding intrinsic: `_mm_unpackhi_epi16`

Interleave the two 16-bit values from the high half of A with the two 16-bit values from the high half of B.

```
I16vec4 unpack_high(I16vec4 A, I16vec4 B);
```

```
Is16vec4 unpack_high(Is16vec4 A, Is16vec4 B);
```

```
Iu16vec4 unpack_high(Iu16vec4 A, Iu16vec4 B);
```

```
R0 = A2;R1 = B2;
R2 = A3;R3 = B3;
```

Corresponding intrinsic: `_mm_unpackhi_pi16`

Interleave the four 8-bit values from the high half of A with the four 8-bit values from the high half of B.

```
I8vec8 unpack_high(I8vec8 A, I8vec8 B);
```

```
Is8vec8 unpack_high(Is8vec8 A, I8vec8 B);
```

```
Iu8vec8 unpack_high(Iu8vec8 A, I8vec8 B);
```

```
R0 = A4;
R1 = B4;
R2 = A5;
R3 = B5;
R4 = A6;
R5 = B6;
R6 = A7;
R7 = B7;
```

Corresponding intrinsic: `_mm_unpackhi_pi8`

Interleave the sixteen 8-bit values from the high half of A with the four 8-bit values from the high half of B.

```
I8vec16 unpack_high(I8vec16 A, I8vec16 B);
```

```
Is8vec16 unpack_high(Is8vec16 A, I8vec16 B);
```

```
Iu8vec16 unpack_high(Iu8vec16 A, I8vec16 B);
```

```
R0 = A8;
R1 = B8;
R2 = A9;
R3 = B9;
R4 = A10;
R5 = B10;
```

```
R6 = A11;
R7 = B11;
R8 = A12;
R8 = B12;
R2 = A13;
R3 = B13;
R4 = A14;
R5 = B14;
R6 = A15;
R7 = B15;
```

Corresponding intrinsic: `_mm_unpackhi_epi16`

Interleave the 32-bit value from the low half of `A` with the 32-bit value from the low half of `B`.

```
R0 = A0;
R1 = B0;
```

Corresponding intrinsic: `_mm_unpacklo_epi32`

Interleave the 64-bit value from the low half of `A` with the 64-bit values from the low half of `B`

```
I64vec2 unpack_low(I64vec2 A, I64vec2 B);

Is64vec2 unpack_low(Is64vec2 A, Is64vec2 B);

Iu64vec2 unpack_low(Iu64vec2 A, Iu64vec2 B);

R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```

Corresponding intrinsic: `_mm_unpacklo_epi32`

Interleave the two 32-bit values from the low half of `A` with the two 32-bit values from the low half of `B`

```
I32vec4 unpack_low(I32vec4 A, I32vec4 B);

Is32vec4 unpack_low(Is32vec4 A, Is32vec4 B);

Iu32vec4 unpack_low(Iu32vec4 A, Iu32vec4 B);

R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```

Corresponding intrinsic: `_mm_unpacklo_epi32`

Interleave the 32-bit value from the low half of `A` with the 32-bit value from the low half of `B`.

```
I32vec2 unpack_low(I32vec2 A, I32vec2 B);

Is32vec2 unpack_low(Is32vec2 A, Is32vec2 B);

Iu32vec2 unpack_low(Iu32vec2 A, Iu32vec2 B);
```

```
R0 = A0;
R1 = B0;
```
Corresponding intrinsic: `_mm_unpacklo_pi32`

Interleave the two 16-bit values from the low half of A with the two 16-bit values from the low half of B.

```
I16vec8 unpack_low(I16vec8 A, I16vec8 B);

Is16vec8 unpack_low(Is16vec8 A, Is16vec8 B);

Iu16vec8 unpack_low(Iu16vec8 A, Iu16vec8 B);
```

```
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
```
Corresponding intrinsic: `_mm_unpacklo_epi16`

Interleave the two 16-bit values from the low half of A with the two 16-bit values from the low half of B.

```
I16vec4 unpack_low(I16vec4 A, I16vec4 B);

Is16vec4 unpack_low(Is16vec4 A, Is16vec4 B);

Iu16vec4 unpack_low(Iu16vec4 A, Iu16vec4 B);
```

```
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```
Corresponding intrinsic: `_mm_unpacklo_pi16`

Interleave the four 8-bit values from the high low of A with the four 8-bit values from the low half of B.

```
I8vec16 unpack_low(I8vec16 A, I8vec16 B);

Is8vec16 unpack_low(Is8vec16 A, Is8vec16 B);

Iu8vec16 unpack_low(Iu8vec16 A, Iu8vec16 B);
```

```
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
R8 = A4;
```

```
R9 = B4;
R10 = A5;
R11 = B5;
R12 = A6;
R13 = B6;
R14 = A7;
R15 = B7;
```

Corresponding intrinsic: `_mm_unpacklo_epi8`

Interleave the four 8-bit values from the high low of `A` with the four 8-bit values from the low half of `B`.

```
I8vec8 unpack_low(I8vec8 A, I8vec8 B);

Is8vec8 unpack_low(Is8vec8 A, Is8vec8 B);

Iu8vec8 unpack_low(Iu8vec8 A, Iu8vec8 B);

R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
```

Corresponding intrinsic: `_mm_unpacklo_pi8`

Pack Operators

Pack the eight 32-bit values found in A and B into eight 16-bit values with signed saturation.

```
Is16vec8 pack_sat(Is32vec2 A,Is32vec2 B);
```
Corresponding intrinsic: `_mm_packs_epi32`

Pack the four 32-bit values found in A and B into eight 16-bit values with signed saturation.

```
Is16vec4 pack_sat(Is32vec2 A,Is32vec2 B);
```
Corresponding intrinsic: `_mm_packs_pi32`

Pack the sixteen 16-bit values found in A and B into sixteen 8-bit values with signed saturation.

```
Is8vec16 pack_sat(Is16vec4 A,Is16vec4 B);
```
Corresponding intrinsic: `_mm_packs_epi16`

Pack the eight 16-bit values found in A and B into eight 8-bit values with signed saturation.

```
Is8vec8 pack_sat(Is16vec4 A,Is16vec4 B);
```
Corresponding intrinsic: `_mm_packs_pi16`

Pack the sixteen 16-bit values found in A and B into sixteen 8-bit values with unsigned saturation .

```
Iu8vec16 packu_sat(Is16vec4 A,Is16vec4 B);
```
Corresponding intrinsic: `_mm_packus_epi16`

Pack the eight 16-bit values found in A and B into eight 8-bit values with unsigned saturation.

```
Iu8vec8 packu_sat(Is16vec4 A,Is16vec4 B);
```
Corresponding intrinsic: `_mm_packs_pu16`

Clear MMX(TM) Instructions State Operator

Empty the MMX(TM) registers and clear the MMX state. Read the guidelines for using the EMMS instruction intrinsic.

```
void empty(void);
```
Corresponding intrinsic: `_mm_empty`

Integer Intrinsics for Streaming SIMD Extensions

Note

You must include `fvec.h` header file for the following functionality.

Compute the element-wise maximum of the respective signed integer words in A and B.

```
Is16vec4 simd_max(Is16vec4 A, Is16vec4 B);
```
Corresponding intrinsic: _mm_max_pi16

Compute the element-wise minimum of the respective signed integer words in A and B.

```
Is16vec4 simd_min(Is16vec4 A, Is16vec4 B);
```
Corresponding intrinsic: _mm_min_pi16

Compute the element-wise maximum of the respective unsigned bytes in A and B.

```
Iu8vec8 simd_max(Iu8vec8 A, Iu8vec8 B);
```
Corresponding intrinsic: _mm_max_pu8

Compute the element-wise minimum of the respective unsigned bytes in A and B.

```
Iu8vec8 simd_min(Iu8vec8 A, Iu8vec8 B);
```
Corresponding intrinsic: _mm_min_pu8

Create an 8-bit mask from the most significant bits of the bytes in A.

```
int move_mask(I8vec8 A);
```
Corresponding intrinsic: _mm_movemask_pi8

Conditionally store byte elements of A to address p. The high bit of each byte in the selector B determines whether the corresponding byte in A will be stored.

```
void mask_move(I8vec8 A, I8vec8 B, signed char *p);
```
Corresponding intrinsic: _mm_maskmove_si64

Store the data in A to the address p without polluting the caches. A can be any `Ivec` type.

```
void store_nta(__m64 *p, M64 A);
```
Corresponding intrinsic: _mm_stream_pi

Compute the element-wise average of the respective unsigned 8-bit integers in A and B.

```
Iu8vec8 simd_avg(Iu8vec8 A, Iu8vec8 B);
```
Corresponding intrinsic: _mm_avg_pu8

Compute the element-wise average of the respective unsigned 16-bit integers in A and B.

```
Iu16vec4 simd_avg(Iu16vec4 A, Iu16vec4 B);
```

Corresponding intrinsic: `_mm_avg_pu16`

Conversions Between Fvec and Ivec

Convert the lower double-precision floating-point value of A to a 32-bit integer with truncation.

```
int F64vec2ToInt(F64vec42 A);

r := (int)A0;
```

Convert the four floating-point values of A to two the two least significant double-precision floating-point values.

```
F64vec2 F32vec4ToF64vec2(F32vec4 A);

r0 := (double)A0;
r1 := (double)A1;
```

Convert the two double-precision floating-point values of A to two single-precision floating-point values.

```
F32vec4 F64vec2ToF32vec4(F64vec2 A);

r0 := (float)A0;
r1 := (float)A1;
```

Convert the signed `int` in B to a double-precision floating-point value and pass the upper double-precision value from A through to the result.

```
F64vec2 InttoF64vec2(F64vec2 A, int B);

r0 := (double)B;
r1 := A1;
```

Convert the lower floating-point value of A to a 32-bit integer with truncation.

```
int F32vec4ToInt(F32vec4 A);

r := (int)A0;
```

Convert the two lower floating-point values of A to two 32-bit integer with truncation, returning the integers in packed form.

```
Is32vec2 F32vec4ToIs32vec2 (F32vec4 A);

r0 := (int)A0;
r1 := (int)A1;
```

Convert the 32-bit integer value B to a floating-point value; the upper three floating-point values are passed through from A.

```
F32vec4 IntToF32vec4(F32vec4 A, int B);

r0 := (float)B;
```

```
r1 := A1;
r2 := A2;
r3 := A3;
```

Convert the two 32-bit integer values in packed form in `B` to two floating-point values; the upper two floating-point values are passed through from `A`.

```
F32vec4 Is32vec2ToF32vec4(F32vec4 A, Is32vec2 B);
```

```
r0 := (float)B0;
r1 := (float)B1;
r2 := A2;
r3 := A3;
```

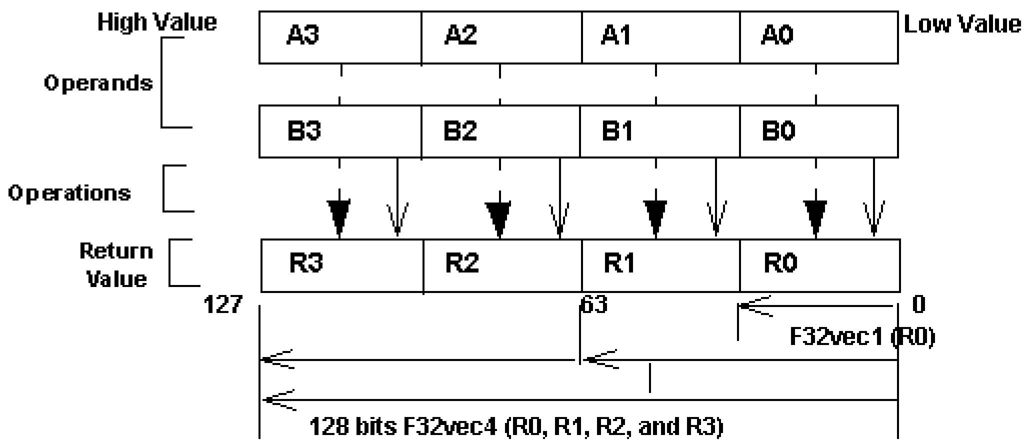
Overview: Floating-point Vector Classes

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 below. `F32vec4` returns four single-precision, floating-point values (R0, R1, R2, and R3); `F64vec2` returns two double-precision, floating-point values, and `F32vec1` returns the lowest single-precision floating-point value (R0).

Return Value Convention Notation Mappings

Example 1:	Example 2:	Example 3:	F32vec4	F64vec2	F32vec1
R0 := A0 & B0;	R0 := A0 andnot B0;	R0 &= A0;	x	x	x
R1 := A1 & B1;	R1 := A1 andnot B1;	R1 &= A1;	x	x	N/A
R2 := A2 & B2;	R2 := A2 andnot B2;	R2 &= A2;	x	N/A	N/A
R3 := A3 & B3	R3 := A3 andhot B3;	R3 &= A3;	x	N/A	N/A

Data Alignment

Memory operations using the Streaming SIMD Extensions should be performed on 16-byte-aligned data whenever possible.

`F32vec4` and `F64vec2` object variables are properly aligned by default. Note that floating point arrays are not automatically aligned. To get 16-byte alignment, you can use the alignment `__declspec`:

```
__declspec( align(16) ) float A[4];
```

Conversions

```
__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 */
```

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.

Constructors and Initialization

The following table shows how to create and initialize `F32vec` objects with the Fvec classes.

Constructors and Initialization for Fvec Classes

Example	Intrinsic	Returns
Constructor Declaration		

`F64vec2 A;`
`F32vec4 B;`
`F32vec1 C;`
	N/A	N/A

__m128 Object Initialization		

`F64vec2 A(__m128d mm);`
`F32vec4 B(__m128 mm);`
`F32vec1 C(__m128 mm);`
	N/A	N/A

Double Initialization		

`/* Initializes two doubles. */`
`F64vec2 A(double d0, double d1);`
`F64vec2 A = F64vec2(double d0, double d1);`
	`_mm_set_pd`	A0 := d0;
		A1 := d1;

`F64vec2 A(double d0);`
`/* Initializes both return values`
`with the same double precision value */.`
	`_mm_set1_pd`	A0 := d0;
		A1 := d0;

Float Initialization		

`F32vec4 A(float f3, float f2,`
`float f1, float f0);`
`F32vec4 A = F32vec4(float f3, float f2,`
`float f1, float f0);`
	`_mm_set_ps`	A0 := f0;
		A1 := f1;
		A2 := f2;
		A3 := f3;

`F32vec4 A(float f0);`
`/* Initializes all return values`
`with the same floating point value. */`
	`_mm_set1_ps`	A0 := f0;
		A1 := f0;
		A2 := f0;
		A3 := f0;

`F32vec4 A(double d0);`
`/* Initialize all return values with`
`the same double-precision value. */`
	`_mm_set1_ps(d)`	A0 := d0;
		A1 := d0;
		A2 := d0;
		A3 := d0;

`F32vec1 A(double d0);`
`/* Initializes the lowest value of A`
`with d0 and the other values with 0.*/`
	`_mm_set_ss(d)`	A0 := d0;
		A1 := 0;
		A2 := 0;
		A3 := 0;

`F32vec1 B(float f0);`
`/* Initializes the lowest value of B`
`with f0 and the other values with 0.*/`
	`_mm_set_ss`	B0 := f0;
		B1 := 0;
		B2 := 0;
		B3 := 0;

`F32vec1 B(int I);`
`/* Initializes the lowest value of B with f0, other values are undefined.*/`
	`_mm_cvtsi32_ss`	B0 := f0;
		B1 := {}
		B2 := {}
		B3 := {}

Arithmetic Operators

The following table lists the arithmetic operators of the `Fvec` classes and generic syntax. The operators have been divided into standard and advanced operations, which are described in more detail later in this section.

Fvec Arithmetic Operators

Category	Operation	Operators	Generic Syntax

Standard	Addition	+
		+=
			R = A + B;
			R += A;

	Subtraction	-
		-=
			R = A - B;
			R -= A;

	Multiplication	*
		*=
			R = A * B;
			R *= A;

	Division	/
		/=
			R = A / B;
			R /= A;

Advanced	Square Root	`sqrt`	R = `sqrt`(A);

	Reciprocal (Newton-Raphson)	`rcp`
		`rcp_nr`
			R = `rcp`(A);
			R = `rcp_nr`(A);

	Reciprocal Square Root (Newton-Raphson)	`rsqrt`
		`rsqrt_nr`
			R = `rsqrt`(A);
			R = `rsqrt_nr`(A);

Standard Arithmetic Operator Usage

The following two tables show the return values for each class of the standard arithmetic operators, which use the syntax styles described earlier in the Return Value Notation section.

Standard Arithmetic Return Value Mapping

R	A	Operators				B	F32vec4	F64vec2	F32vec1
R0:=	A0	+	-	*	/	B0			
R1:=	A1	+	-	*	/	B1			N/A
R2:=	A2	+	-	*	/	B2		N/A	N/A
R3:=	A3	+	-	*	/	B3		N/A	N/A

Arithmetic with Assignment Return Value Mapping

R	Operators				A	F32vec4	F64vec2	F32vec1
R0:=	+=	-=	*=	/=	A0			
R1:=	+=	-=	*=	/=	A1			N/A
R2:=	+=	-=	*=	/=	A2		N/A	N/A
R3:=	+=	-=	*=	/=	A3		N/A	N/A

The table below lists standard arithmetic operator syntax and intrinsics.

Standard Arithmetic Operations for Fvec Classes

Operation	Returns	Example Syntax Usage	Intrinsic

Addition	4 floats	`F32vec4 R = F32vec4 A + F32vec4 B;`
		`F32vec4 R += F32vec4 A;`
			`_mm_add_ps`

	2 doubles	`F64vec2 R = F64vec2 A + F32vec2 B;`
		`F64vec2 R += F64vec2 A;`
			`_mm_add_pd`

	1 float	`F32vec1 R = F32vec1 A + F32vec1 B;`
		`F32vec1 R += F32vec1 A;`
			`_mm_add_ss`

Subtraction	4 floats	`F32vec4 R = F32vec4 A - F32vec4 B;`
		`F32vec4 R -= F32vec4 A;`
			`_mm_sub_ps`

	2 doubles	`F64vec2 R - F64vec2 A + F32vec2 B;`
		`F64vec2 R -= F64vec2 A;`
			`_mm_sub_pd`

	1 float	`F32vec1 R = F32vec1 A - F32vec1 B;`
		`F32vec1 R -= F32vec1 A;`
			`_mm_sub_ss`

Multiplication	4 floats	`F32vec4 R = F32vec4 A * F32vec4 B;`
		`F32vec4 R *= F32vec4 A;`
			`_mm_mul_ps`

	2 doubles	`F64vec2 R = F64vec2 A * F364vec2 B;`
		`F64vec2 R *= F64vec2 A;`
			`_mm_mul_pd`

	1 float	`F32vec1 R = F32vec1 A * F32vec1 B;`
		`F32vec1 R *= F32vec1 A;`
			`_mm_mul_ss`

Division	4 floats	`F32vec4 R = F32vec4 A / F32vec4 B;`
		`F32vec4 R /= F32vec4 A;`
			`_mm_div_ps`

	2 doubles	`F64vec2 R = F64vec2 A / F64vec2 B;`
		`F64vec2 R /= F64vec2 A;`
			`_mm_div_pd`

	1 float	`F32vec1 R = F32vec1 A / F32vec1 B;`
		`F32vec1 R /= F32vec1 A;`
			`_mm_div_ss`

Advanced Arithmetic Operator Usage

The following table shows the return values classes of the advanced arithmetic operators, which use the syntax styles described earlier in the Return Value Notation section.

Advanced Arithmetic Return Value Mapping

R	Operators					A	F32vec4	F64vec2	F32vec1
R0:=	`sqrt`	`rcp`	`rsqrt`	`rcp_nr`	`rsqrt_nr`	A0			
R1:=	`sqrt`	`rcp`	`rsqrt`	`rcp_nr`	`rsqrt_nr`	A1			N/A
R2:=	`sqrt`	`rcp`	`rsqrt`	`rcp_nr`	`rsqrt_nr`	A2		N/A	N/A
R3:=	`sqrt`	`rcp`	`rsqrt`	`rcp_nr`	`rsqrt_nr`	A3		N/A	N/A
f :=	add_horizontal			(A0 + A1 + A2 + A3)				N/A	N/A
d :=	add_horizontal			(A0 + A1)			N/A		N/A

The table below shows examples for advanced arithmetic operators.

Advanced Arithmetic Operations for Fvec Classes

Returns	Example Syntax Usage	Intrinsic
Square Root		
4 floats	`F32vec4 R = sqrt(F32vec4 A);`	`_mm_sqrt_ps`
2 doubles	`F64vec2 R = sqrt(F64vec2 A);`	`_mm_sqrt_pd`
1 float	`F32vec1 R = sqrt(F32vec1 A);`	`_mm_sqrt_ss`
Reciprocal		
4 floats	`F32vec4 R = rcp(F32vec4 A);`	`_mm_rcp_ps`
2 doubles	`F64vec2 R = rcp(F64vec2 A);`	`_mm_rcp_pd`
1 float	`F32vec1 R = rcp(F32vec1 A);`	`_mm_rcp_ss`
Reciprocal Square Root		
4 floats	`F32vec4 R = rsqrt(F32vec4 A);`	`_mm_rsqrt_ps`
2 doubles	`F64vec2 R = rsqrt(F64vec2 A);`	`_mm_rsqrt_pd`
1 float	`F32vec1 R = rsqrt(F32vec1 A);`	`_mm_rsqrt_ss`
Reciprocal Newton Raphson		
4 floats	`F32vec4 R = rcp_nr(F32vec4 A);`	`_mm_sub_ps` `_mm_add_ps` `_mm_mul_ps` `_mm_rcp_ps`
2 doubles	`F64vec2 R = rcp_nr(F64vec2 A);`	`_mm_sub_pd` `_mm_add_pd` `_mm_mul_pd` `_mm_rcp_pd`
1 float	`F32vec1 R = rcp_nr(F32vec1 A);`	`_mm_sub_ss` `_mm_add_ss` `_mm_mul_ss` `_mm_rcp_ss`

Reciprocal Square Root Newton Raphson		
4 float	`F32vec4 R = rsqrt_nr(F32vec4 A);`	`_mm_sub_pd` `_mm_mul_pd` `_mm_rsqrt_ps`
2 doubles	`F64vec2 R = rsqrt_nr(F64vec2 A);`	`_mm_sub_pd` `_mm_mul_pd` `_mm_rsqrt_pd`
1 float	`F32vec1 R = rsqrt_nr(F32vec1 A);`	`_mm_sub_ss` `_mm_mul_ss` `_mm_rsqrt_ss`
Horizontal Add		
1 float	`float f = add_horizontal(F32vec4 A);`	`_mm_add_ss` `_mm_shuffle_ss`
1 double	`double d = add_horizontal(F64vec2 A);`	`_mm_add_sd` `_mm_shuffle_sd`

Minimum and Maximum Operators

Compute the minimums of the two double precision floating-point values of A and B.

```
F64vec2 R = simd_min(F64vec2 A, F64vec2 B)

R0 := min(A0,B0);
R1 := min(A1,B1);
```
Corresponding intrinsic: _mm_min_pd

Compute the minimums of the four single precision floating-point values of A and B.

```
F32vec4 R = simd_min(F32vec4 A, F32vec4 B)

R0 := min(A0,B0);
R1 := min(A1,B1);
R2 := min(A2,B2);
R3 := min(A3,B3);
```
Corresponding intrinsic: _mm_min_ps

Compute the minimum of the lowest single precision floating-point values of A and B.

```
F32vec1 R = simd_min(F32vec1 A, F32vec1 B)

R0 := min(A0,B0);
```
Corresponding intrinsic: _mm_min_ss

Compute the maximums of the two double precision floating-point values of A and B.

```
F64vec2 simd_max(F64vec2 A, F64vec2 B)

R0 := max(A0,B0);
R1 := max(A1,B1);
```
Corresponding intrinsic: _mm_max_pd

Compute the maximums of the four single precision floating-point values of A and B.

```
F32vec4 R = simd_man(F32vec4 A, F32vec4 B)

R0 := max(A0,B0);
R1 := max(A1,B1);
R2 := max(A2,B2);
R3 := max(A3,B3);
```
Corresponding intrinsic: _mm_max_ps

Compute the maximum of the lowest single precision floating-point values of A and B.

```
F32vec1 simd_max(F32vec1 A, F32vec1 B)

R0 := max(A0,B0);
```
Corresponding intrinsic: _mm_max_ss

Logical Operators

The table below lists the logical operators of the Fvec classes and generic syntax. The logical operators for `F32vec1` classes use only the lower 32 bits.

Fvec Logical Operators Return Value Mapping

Bitwise Operation	Operators	Generic Syntax

AND	&
	&=
		R = A & B;
		R &= A;

OR	|
	|=
		R = A | B;
		R |= A;

XOR	^
	^=
		R = A ^ B;
		R ^= A;

andnot	andnot	R = andnot(A);

The following table lists standard logical operators syntax and corresponding intrinsics. Note that there is no corresponding scalar intrinsic for the `F32vec1` classes, which accesses the lower 32 bits of the packed vector intrinsics.

Logical Operations for Fvec Classes

Operation	Returns	Example Syntax Usage	Intrinsic

AND	4 floats	`F32vec4 & = F32vec4 A & F32vec4 B;`
		`F32vec4 & &= F32vec4 A;`
			`_mm_and_ps`

	2 doubles	`F64vec2 R = F64vec2 A & F32vec2 B;`
		`F64vec2 R &= F64vec2 A;`
			`_mm_and_pd`

	1 float	`F32vec1 R = F32vec1 A & F32vec1 B;`
		`F32vec1 R &= F32vec1 A;`
			`_mm_and_ps`

OR	4 floats	`F32vec4 R = F32vec4 A | F32vec4 B;`
		`F32vec4 R |= F32vec4 A;`
			`_mm_or_ps`

	2 doubles	`F64vec2 R = F64vec2 A | F32vec2 B;`
		`F64vec2 R |= F64vec2 A;`
			`_mm_or_pd`

	1 float	`F32vec1 R = F32vec1 A | F32vec1 B;`
		`F32vec1 R |= F32vec1 A;`
			`_mm_or_ps`

XOR	4 floats	`F32vec4 R = F32vec4 A ^ F32vec4 B;`
		`F32vec4 R ^= F32vec4 A;`
			`_mm_xor_ps`

	2 doubles	`F64vec2 R = F64vec2 A ^ F364vec2 B;`
		`F64vec2 R ^= F64vec2 A;`
			`_mm_xor_pd`

	1 float	`F32vec1 R = F32vec1 A ^ F32vec1 B;`
		`F32vec1 R ^= F32vec1 A;`
			`_mm_xor_ps`

ANDNOT	2 doubles	`F64vec2 R = andnot(F64vec2 A,`
		`F64vec2 B);`
			`_mm_andnot_pd`

Compare Operators

The operators described in this section compare the single precision floating-point values of `A` and `B`. Comparison between objects of any `Fvec` class return the same class being compared.

The following table lists the compare operators for the `Fvec` classes.

Compare Operators and Corresponding Intrinsics

Compare For:	Operators	Syntax
Equality	`cmpeq`	`R = cmpeq(A, B)`
Inequality	`cmpneq`	`R = cmpneq(A, B)`
Greater Than	`cmpgt`	`R = cmpgt(A, B)`
Greater Than or Equal To	`cmpge`	`R = cmpge(A, B)`
Not Greater Than	`cmpngt`	`R = cmpngt(A, B)`
Not Greater Than or Equal To	`cmpnge`	`R = cmpnge(A, B)`
Less Than	`cmplt`	`R = cmplt(A, B)`
Less Than or Equal To	`cmple`	`R = cmple(A, B)`
Not Less Than	`cmpnlt`	`R = cmpnlt(A, B)`
Not Less Than or Equal To	`cmpnle`	`R = cmpnle(A, B)`

Compare Operators

The mask is set to `0xffffffff` for each floating-point value where the comparison is true and `0x00000000` where the comparison is false. The table below shows the return values for each class of the compare operators, which use the syntax described earlier in the Return Value Notation section.

Compare Operator Return Value Mapping

R	A0	For Any Operators	B	If True	If False	F32vec4	F64vec2	F32vec1

R0:=	(A1
	
	!
	(A1
		`cmp[eq | lt | le | gt | ge]`
		`cmp[ne | nlt | nle | ngt | nge]`
			B1)
			B1)
				`0xffffffff`	`0x0000000`	X	X	X

R1:=	(A1
	
	!
	(A1
		`cmp[eq | lt | le | gt | ge]`
		`cmp[ne | nlt | nle | ngt | nge]`
			B2)
			B2)
				`0xffffffff`	`0x0000000`	X	X	N/A

R2:=	(A1 ! (A1	`cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]`	B3) B3)	`0xffffffff`	`0x0000000`	X	N/A	N/A
R3:=	A3	`cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]`	B3) B3)	`0xffffffff`	`0x0000000`	X	N/A	N/A

The table below shows examples for arithmetic operators and intrinsics.

Compare Operations for Fvec Classes

Returns	Example Syntax Usage	Intrinsic
Compare for Equality		
4 floats	`F32vec4 R = cmpeq(F32vec4 A);`	`_mm_cmpeq_ps`
2 doubles	`F64vec2 R = cmpeq(F64vec2 A);`	`_mm_cmpeq_pd`
1 float	`F32vec1 R = cmpeq(F32vec1 A);`	`_mm_cmpeq_ss`
Compare for Inequality		
4 floats	`F32vec4 R = cmpneq(F32vec4 A);`	`_mm_cmpneq_ps`
2 doubles	`F64vec2 R = cmpneq(F64vec2 A);`	`_mm_cmpneq_pd`
1 float	`F32vec1 R = cmpneq(F32vec1 A);`	`_mm_cmpneq_ss`
Compare for Less Than		
4 floats	`F32vec4 R = cmplt(F32vec4 A);`	`_mm_cmplt_ps`
2 doubles	`F64vec2 R = cmplt(F64vec2 A);`	`_mm_cmplt_pd`
1 float	`F32vec1 R = cmplt(F32vec1 A);`	`_mm_cmplt_ss`
Compare for Less Than or Equal		
4 floats	`F32vec4 R = cmple(F32vec4 A);`	`_mm_cmple_ps`
2 doubles	`F64vec2 R = cmple(F64vec2 A);`	`_mm_cmple_pd`
1 float	`F32vec1 R = cmple(F32vec1 A);`	`_mm_cmple_pd`
Compare for Greater Than		
4 floats	`F32vec4 R = cmpgt(F32vec4 A);`	`_mm_cmpgt_ps`
2 doubles	`F64vec2 R = cmpgt(F32vec42 A);`	`_mm_cmpgt_pd`
1 float	`F32vec1 R = cmpgt(F32vec1 A);`	`_mm_cmpgt_ss`
Compare for Greater Than or Equal To		
4 floats	`F32vec4 R = cmpge(F32vec4 A);`	`_mm_cmpge_ps`

2 doubles	`F64vec2 R = cmpge(F64vec2 A);`	`_mm_cmpge_pd`
1 float	`F32vec1 R = cmpge(F32vec1 A);`	`_mm_cmpge_ss`
Compare for Not Less Than		
4 floats	`F32vec4 R = cmpnlt(F32vec4 A);`	`_mm_cmpnlt_ps`
2 doubles	`F64vec2 R = cmpnlt(F64vec2 A);`	`_mm_cmpnlt_pd`
1 float	`F32vec1 R = cmpnlt(F32vec1 A);`	`_mm_cmpnlt_ss`
Compare for Not Less Than or Equal		
4 floats	`F32vec4 R = cmpnle(F32vec4 A);`	`_mm_cmpnle_ps`
2 doubles	`F64vec2 R = cmpnle(F64vec2 A);`	`_mm_cmpnle_pd`
1 float	`F32vec1 R = cmpnle(F32vec1 A);`	`_mm_cmpnle_ss`
Compare for Not Greater Than		
4 floats	`F32vec4 R = cmpngt(F32vec4 A);`	`_mm_cmpngt_ps`
2 doubles	`F64vec2 R = cmpngt(F64vec2 A);`	`_mm_cmpngt_pd`
1 float	`F32vec1 R = cmpngt(F32vec1 A);`	`_mm_cmpngt_ss`
Compare for Not Greater Than or Equal		
4 floats	`F32vec4 R = cmpnge(F32vec4 A);`	`_mm_cmpnge_ps`
2 doubles	`F64vec2 R = cmpnge(F64vec2 A);`	`_mm_cmpnge_pd`
1 float	`F32vec1 R = cmpnge(F32vec1 A);`	`_mm_cmpnge_ss`

Conditional Select Operators for Fvec Classes

Each conditional function compares single-precision floating-point values of A and B. The C and D parameters are used for return value. Comparison between objects of any Fvec class returns the same class.

Conditional Select Operators for Fvec Classes

Conditional Select for:	Operators	Syntax
Equality	`select_eq`	`R = select_eq(A, B)`
Inequality	`select_neq`	`R = select_neq(A, B)`
Greater Than	`select_gt`	`R = select_gt(A, B)`
Greater Than or Equal To	`select_ge`	`R = select_ge(A, B)`
Not Greater Than	`select_gt`	`R = select_gt(A, B)`
Not Greater Than or Equal To	`select_ge`	`R = select_ge(A, B)`
Less Than	`select_lt`	`R = select_lt(A, B)`
Less Than or Equal To	`select_le`	`R = select_le(A, B)`
Not Less Than	`select_nlt`	`R = select_nlt(A, B)`
Not Less Than or Equal To	`select_nle`	`R = select_nle(A, B)`

Conditional Select Operator Usage

For conditional select operators, the return value is stored in C if the comparison is true or in D if false. The following table shows the return values for each class of the conditional select operators, using the Return Value Notation described earlier.

Compare Operator Return Value Mapping

R	A0	Operators	B	C	D	F32vec4	F64vec2	F32vec1
R0:=	(A1 ! (A1 | `select_[eq | lt | le | gt | ge]` `select_[ne | nlt | nle | ngt | nge]` | B0) B0)	C0 C0	D0 D0	X	X	X
R1:=	(A2 ! (A2 | `select_[eq | lt | le | gt | ge]` `select_[ne | nlt | nle | ngt | nge]` | B1) B1)	C1 C1	D1 D1	X	X	N/A
R2:=	(A2 ! (A2 | `select_[eq | lt | le | gt | ge]` `select_[ne | nlt | nle | ngt | nge]` | B2) B2)	C2 C2	D2 D2	X	N/A	N/A

R3:=	(A3 ! (A3	select_[eq | lt | le | gt | ge] select_[ne | nlt | nle | ngt | nge]	B3) B3)	C3 C3	D3 D3	X	N/A	N/A

The following table shows examples for conditional select operations and corresponding intrinsics.

Conditional Select Operations for Fvec Classes

Returns	Example Syntax Usage	Intrinsic
Compare for Equality		
4 floats	`F32vec4 R = select_eq(F32vec4 A);`	`_mm_cmpeq_ps`
2 doubles	`F64vec2 R = select_eq(F64vec2 A);`	`_mm_cmpeq_pd`
1 float	`F32vec1 R = select_eq(F32vec1 A);`	`_mm_cmpeq_ss`
Compare for Inequality		
4 floats	`F32vec4 R = select_neq(F32vec4 A);`	`_mm_cmpneq_ps`
2 doubles	`F64vec2 R = select_neq(F64vec2 A);`	`_mm_cmpneq_pd`
1 float	`F32vec1 R = select_neq(F32vec1 A);`	`_mm_cmpneq_ss`
Compare for Less Than		
4 floats	`F32vec4 R = select_lt(F32vec4 A);`	`_mm_cmplt_ps`
2 doubles	`F64vec2 R = select_lt(F64vec2 A);`	`_mm_cmplt_pd`
1 float	`F32vec1 R = select_lt(F32vec1 A);`	`_mm_cmplt_ss`
Compare for Less Than or Equal		
4 floats	`F32vec4 R = select_le(F32vec4 A);`	`_mm_cmple_ps`
2 doubles	`F64vec2 R = select_le(F64vec2 A);`	`_mm_cmple_pd`
1 float	`F32vec1 R = select_le(F32vec1 A);`	`_mm_cmple_ps`
Compare for Greater Than		
4 floats	`F32vec4 R = select_gt(F32vec4 A);`	`_mm_cmpgt_ps`
2 doubles	`F64vec2 R = select_gt(F64vec2 A);`	`_mm_cmpgt_pd`
1 float	`F32vec1 R = select_gt(F32vec1 A);`	`_mm_cmpgt_ss`
Compare for Greater Than or Equal To		
4 floats	`F32vec1 R = select_ge(F32vec4 A);`	`_mm_cmpge_ps`
2 doubles	`F64vec2 R = select_ge(F64vec2 A);`	`_mm_cmpge_pd`
1 float	`F32vec1 R = select_ge(F32vec1 A);`	`_mm_cmpge_ss`
Compare for Not Less Than		

4 floats	`F32vec1 R = select_nlt(F32vec4 A);`	`_mm_cmpnlt_ps`
2 doubles	`F64vec2 R = select_nlt(F64vec2 A);`	`_mm_cmpnlt_pd`
1 float	`F32vec1 R = select_nlt(F32vec1 A);`	`_mm_cmpnlt_ss`
Compare for Not Less Than or Equal		
4 floats	`F32vec1 R = select_nle(F32vec4 A);`	`_mm_cmpnle_ps`
2 doubles	`F64vec2 R = select_nle(F64vec2 A);`	`_mm_cmpnle_pd`
1 float	`F32vec1 R = select_nle(F32vec1 A);`	`_mm_cmpnle_ss`
Compare for Not Greater Than		
4 floats	`F32vec1 R = select_ngt(F32vec4 A);`	`_mm_cmpngt_ps`
2 doubles	`F64vec2 R = select_ngt(F64vec2 A);`	`_mm_cmpngt_pd`
1 float	`F32vec1 R = select_ngt(F32vec1 A);`	`_mm_cmpngt_ss`
Compare for Not Greater Than or Equal		
4 floats	`F32vec1 R = select_nge(F32vec4 A);`	`_mm_cmpnge_ps`
2 doubles	`F64vec2 R = select_nge(F64vec2 A);`	`_mm_cmpnge_pd`
1 float	`F32vec1 R = select_nge(F32vec1 A);`	`_mm_cmpnge_ss`

Cacheability Support Operations

Stores (non-temporal) the two double-precision, floating-point values of A. Requires a 16-byte aligned address.

```
void store_nta(double *p, F64vec2 A);
```
Corresponding intrinsic: `_mm_stream_pd`

Stores (non-temporal) the four single-precision, floating-point values of A. Requires a 16-byte aligned address.

```
void store_nta(float *p, F32vec4 A);
```
Corresponding intrinsic: `_mm_stream_ps`

Debugging

The debug operations do not map to any compiler intrinsics for MMX(TM) technology or Streaming SIMD Extensions. They are provided for debugging programs only. Use of these operations may result in loss of performance, so you should not use them outside of debugging.

Output Operations

The two single, double-precision floating-point values of A are placed in the output buffer and printed in decimal format as follows:

```
cout << F64vec2 A;
"[1]:A1 [0]:A0"
```
Corresponding intrinsics: none

The four, single-precision floating-point values of A are placed in the output buffer and printed in decimal format as follows:

```
cout << F32vec4 A;
"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```
Corresponding intrinsics: none

The lowest, single-precision floating-point value of A is placed in the output buffer and printed.

```
cout << F32vec1 A;
```
Corresponding intrinsics: none

Element Access Operations

```
double d = F64vec2 A[int i]
```

Read one of the two, double-precision floating-point values of A without modifying the corresponding floating-point value. Permitted values of `i` are 0 and 1. For example:

If DEBUG is enabled and `i` is not one of the permitted values (0 or 1), a diagnostic message is printed and the program aborts.

```
double d = F64vec2 A[1];
```
Corresponding intrinsics: none

Read one of the four, single-precision floating-point values of A without modifying the corresponding floating point value. Permitted values of `i` are 0, 1, 2, and 3. For example:

```
float f = F32vec4 A[int i]
```

If DEBUG is enabled and `i` is not one of the permitted values (0-3), a diagnostic message is printed and the program aborts.

```
float f = F32vec4 A[2];
```
Corresponding intrinsics: none

Element Assignment Operations

```
F64vec4 A[int i] = double d;
```

Modify one of the two, double-precision floating-point values of A. Permitted values of `int i` are 0 and 1. For example:

```
F32vec4 A[1] = double d;
F32vec4 A[int i] = float f;
```

Modify one of the four, single-precision floating-point values of A. Permitted values of `int i` are 0, 1, 2, and 3. For example:

If DEBUG is enabled and `int i` is not one of the permitted values (0-3), a diagnostic message is printed and the program aborts.

```
F32vec4 A[3] = float f;
```

Corresponding intrinsics: none.

Load and Store Operators

Loads two, double-precision floating-point values, copying them into the two, floating-point values of A. No assumption is made for alignment.

```
void loadu(F64vec2 A, double *p)
```
Corresponding intrinsic: `_mm_loadu_pd`

Stores the two, double-precision floating-point values of A. No assumption is made for alignment.

```
void storeu(float *p, F64vec2 A);
```
Corresponding intrinsic: `_mm_storeu_pd`

Loads four, single-precision floating-point values, copying them into the four floating-point values of A. No assumption is made for alignment.

```
void loadu(F32vec4 A, double *p)
```
Corresponding intrinsic: `_mm_loadu_ps`

Stores the four, single-precision floating-point values of A. No assumption is made for alignment.

```
void storeu(float *p, F32vec4 A);
```
Corresponding intrinsic: `_mm_storeu_ps`

Unpack Operators for Fvec Operators

Selects and interleaves the lower, double-precision floating-point values from A and B.

```
F64vec2 R = unpack_low(F64vec2 A, F64vec2 B);
```
Corresponding intrinsic: `_mm_unpacklo_pd(a, b)`

Selects and interleaves the higher, double-precision floating-point values from A and B.

```
F64vec2 R = unpack_high(F64vec2 A, F64vec2 B);
```
Corresponding intrinsic: `_mm_unpackhi_pd(a, b)`

Selects and interleaves the lower two, single-precision floating-point values from A and B.

```
F32vec4 R = unpack_low(F32vec4 A, F32vec4 B);
```
Corresponding intrinsic: `_mm_unpacklo_ps(a, b)`

Selects and interleaves the higher two, single-precision floating-point values from A and B.

```
F32vec4 R = unpack_high(F32vec4 A, F32vec4 B);
```
Corresponding intrinsic: `_mm_unpackhi_ps(a, b)`

Move Mask Operator

Creates a 2-bit mask from the most significant bits of the two, double-precision floating-point values of A, as follows:

```
int i = move_mask(F64vec2 A)
i := sign(a1)<<1 | sign(a0)<<0
```
Corresponding intrinsic: `_mm_movemask_pd`

Creates a 4-bit mask from the most significant bits of the four, single-precision floating-point values of A, as follows:

```
int i = move_mask(F32vec4 A)
i := sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 | sign(a0)<<0
```
Corresponding intrinsic: `_mm_movemask_ps`

Classes Quick Reference

This appendix contains tables listing the class, functionality, and corresponding intrinsics for each class in the Intel® C++ Class Libraries for SIMD Operations. The following table lists all Intel C++ Compiler intrinsics that are not implemented in the C++ SIMD classes.

Logical Operators: Corresponding Intrinsics and Classes

Operators	Corresponding Intrinsic	I128vec1, I64vec2, I32vec4, I16vec8, I8vec16	I64vec, I32vec, I16vec, I8vec8	F64vec2	F32vec4	F32vec1
&, &=	_mm_and_[x]	si128	si64	pd	ps	ps
|, |=	_mm_or_[x]	si128	si64	pd	ps	ps
^, ^=	_mm_xor_[x]	si128	si64	pd	ps	ps
Andnot	_mm_andnot_[x]	si128	si64	pd	N/A	N/A

Arithmetic: Corresponding Intrinsics and Classes

Operators	Corresponding Intrinsic	I64vec2	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F6
+, +=	_mm_add_[x]	epi64	epi32	epi16	epi8	pi32	pi16	pi8	pc
-, -=	_mm_sub_[x]	epi64	epi32	epi16	epi8	pi32	pi16	pi8	pc
*, *=	_mm_mullo_[x]	N/A	N/A	epi16	N/A	N/A	pi16	N/A	pc
/, /=	_mm_div_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	pc

mul_high	_mm_mulhi_ [x]	N/A	N/A	epi16	N/A	N/A	pi16	N/A	N/
mul_add	_mm_madd_ [x]	N/A	N/A	epi16	N/A	N/A	pi16	N/A	N/
sqrt	_mm_sqrt_ [x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	p
rcp	_mm_rcp_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	p
rcp_nr	_mm_rcp_[x] _mm_add_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	p
rsqrt	_mm_rsqrt_ [x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	p
rsqrt_nr	_mm_rsqrt_ [x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	p

Shift Operators: Corresponding Intrinsics and Classes

Operators	Corresponding Intrinsic	I128vec1	I64vec2	I32vec4	I16vec8	I8vec16	I64vec1	I32vec2
>>,>>=	_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	si64 si64 N/A N/A	pi32 pi32 pi32 pi32
<<, <<=	_mm_sll_[x] _mm_slli_ [x]	N/A N/A	epi64 epi64	epi32 epi32	epi16 epi16	N/A N/A	si64 si64	pi32 pi32

Comparison Operators: Corresponding Intrinsics and Classes

Operators	Corresponding Intrinsic	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F64vec
cmpeq	_mm_cmpeq_ [x]	epi32	epi16	epi8	pi32	pi16	pi8	pd
cmpneq	_mm_cmpeq_ [x] _mm_andnot_ [y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64	pd
cmpgt	_mm_cmpgt_ [x]	epi32	epi16	epi8	pi32	pi16	pi8	pd

cmpge	_mm_cmpge_ [x] _mm_andnot_ [y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64	pd
cmplt	_mm_cmplt_ [x]	epi32	epi16	epi8	pi32	pi16	pi8	pd
cmple	_mm_cmple_ [x] _mm_andnot_ [y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64	pd
cmpngt	_mm_cmpngt_ [x]	epi32	epi16	epi8	pi32	pi16	pi8	pd
cmpnge	_mm_cmpnge_ [x]	N/A	N/A	N/A	N/A	N/A	N/A	pd
cmnpnlt	_mm_cmpnlt_ [x]	N/A	N/A	N/A	N/A	N/A	N/A	pd
cmpnle	_mm_cmpnle_ [x]	N/A	N/A	N/A	N/A	N/A	N/A	pd
* Note that _mm_andnot_ [y] intrinsics do not apply to the fvec classes.								

Conditional Select Operators: Corresponding Intrinsics and Classes

Operators	Corresponding Intrinsic	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F6
select_eq	_mm_cmpeq_ [x] _mm_and_[y] _mm_andnot_ [y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64	pd
select_neq	_mm_cmpeq_ [x] _mm_and_[y] _mm_andnot_ [y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64	pd
select_gt	_mm_cmpgt_ [x] _mm_and_[y] _mm_andnot_ [y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64	pd

select_ge	_mm_cmpge_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64	pd
select_lt	_mm_cmplt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64	pd
select_le	_mm_cmple_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64	pd
select_ngt	_mm_cmpgt_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd
select_nge	_mm_cmpge_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd
select_nlt	_mm_cmplt_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd
select_nle	_mm_cmple_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd
* Note that _mm_andnot_[y] intrinsics do not apply to the fvec classes.								

Packing and Unpacking Operators: Corresponding Intrinsics and Classes

Operators	Corresponding Intrinsic	I64vec2	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec
unpack_high	_mm_unpackhi_[x]	epi64	epi32	epi16	epi8	pi32	pi16	pi8
unpack_low	_mm_unpacklo_[x]	epi64	epi32	epi16	epi8	pi32	pi16	pi8
pack_sat	_mm_packs_[x]	N/A	epi32	epi16	N/A	pi32	pi16	N/A
packu_sat	_mm_packus_[x]	N/A	N/A	epi16	N/A	N/A	pu16	N/A
sat_add	_mm_adds_[x]	N/A	N/A	epi16	epi8	N/A	pi16	pi8
sat_sub	_mm_subs_[x]	N/A	N/A	epi16	epi8	N/A	pi16	pi8

Conversions Operators: Corresponding Intrinsics and Classes

Operators	**Corresponding Intrinsic**
`F64vec2ToInt`	`_mm_cvttsd_si32`
`F32vec4ToF64vec2`	`_mm_cvtps_pd`
`F64vec2ToF32vec4`	`_mm_cvtpd_ps`
`IntToF64vec2`	`_mm_cvtsi32_sd`
`F32vec4ToInt`	`_mm_cvtt_ss2si`
`F32vec4ToIs32vec2`	`_mm_cvttps_pi32`
`IntToF32vec4`	`_mm_cvtsi32_ss`
`Is32vec2ToF32vec4`	`_mm_cvtpi32_ps`

Programming Example

This sample program uses the `F32vec4` class to average the elements of a 20 element floating point array. This code is also provided as a sample in the file, `AvgClass.cpp`.

```
// 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_ALIGN 16 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);
}
```