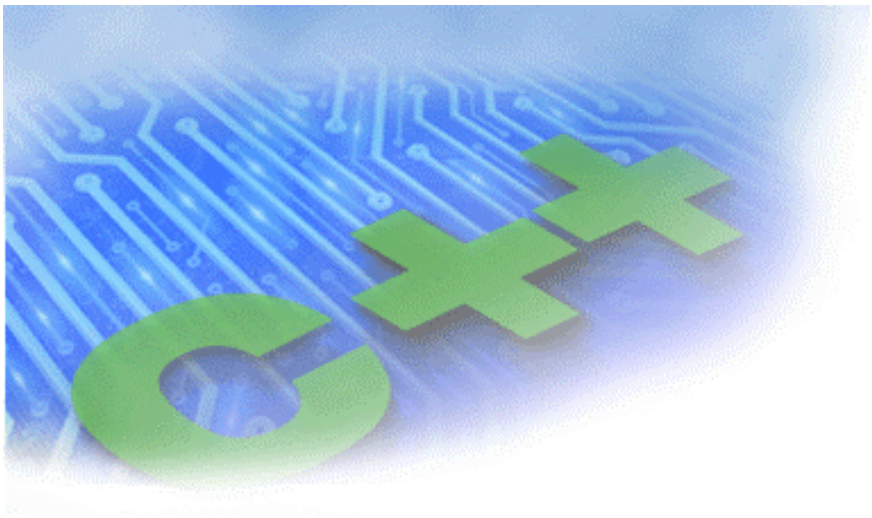


Intel® C++ Compiler
User's Guide

Copyright© 1996-2002 Intel Corporation
All rights reserved
Issued in USA

Document No. CL-600-06

Table of Contents

ABOUT INTEL(R) C++ COMPILER 9

Welcome to the Intel® C++ Compiler 9
What's New in This Release 9
Features and Benefits 10
Product Web Site and Support 10
System Requirements 11
FLEXlm* Electronic Licensing 11
About This Document 12
How to Use This Document 12
Related Publications 14
Disclaimer 16

COMPILER OPTIONS QUICK REFERENCE GUIDES 17

Options Quick Reference Guides Overview 17
New Options in Version 6.0 18
Alphabetical Listing 21
Compiler Options Quick Reference Guide 21
Functional Groups Listing 30
Customizing Compilation Process Options 30
Alternate Tools and Locations 30
Preprocessing Options 30
Controlling Compilation Flow 30
Controlling Compilation Output 31
Debugging Options 31
Diagnostic Messages 32
Language Conformance Options 32
Conformance Options 32
Application Performance Optimization Options 32
Optimization-level Options 32
Floating-point Arithmetic Precision 33
Processor Dispatch Support (IA-32 only) 35
Interprocedural Optimizations 36
Profile-guided Optimizations 36
High-level Language Optimizations 37
Vectorization Options 37
Optimization Reports 38
Compiler Options Cross-Reference for Windows* and Linux* 39
Compiler Options Cross Reference 39

GETTING STARTED WITH THE INTEL(R) C++ COMPILER 43

Invoking the Compiler 43
Invoking the Compiler from the Command Line 43
Invoking the Compiler from the Command Line with make 44
Compiler Input Files 45
Default Behavior of the Compiler 46
Default Compiler Options 46
Default Behavior of the Compiler 48
Compilation Phases 48

CUSTOMIZING COMPILATION ENVIRONMENT 50

Customizing the Compilation Environment 50
Environment Variables 50
Configuration Files 51
Response Files 52
Include Files 53

CUSTOMIZING COMPILATION PROCESS 54

Customizing Compilation Process Overview 54
Specifying Alternate Tools and Paths 54
Preprocessing 55
Preprocessing Overview 55
Preprocessing Only 56
Searching for Include Files 56
Defining Macros 57
Compiling 60
Compilation Overview 60
Compilation Options 60
Controlling Compilation 60
Monitoring Data Settings 61
Assembly File Listing Example 62
Linking 63
Debugging 64
Debugging Options Overview 64
Preparing for Debugging 65
Support for Symbolic Debugging 65
Parsing for Syntax Only 65

LANGUAGE CONFORMANCE 66

Conformance to the C Standard 66

Conformance to the C++ Standard 68

OPTIMIZATIONS 69

Optimization Levels 69
Setting Optimization Levels 69
Restricting Optimizations 70
Floating-point Optimizations 71
Restricting Floating-point Arithmetic Precision 71
Processor Dispatch Extensions Support (IA-32 only) 72
Targeting a Processor and Extensions Support 72
Targeting a Processor (IA-32 only) 72
Exclusive Specialized Code (IA-32 only) 73
Specialized Code with -ax{i|M|K|W} 73
Combining Processor Target and Dispatch Options (IA-32 only) 75
Interprocedural Optimizations 76
Multifile IPO 76
Multifile IPO Overview 76
Compilation with Real Object Files 77
Creating a Multifile IPO Executable 78
Creating a Multifile IPO Executable Using a Project Makefile 79
Creating a Library from IPO Objects 79
Analyzing the Effects of Multifile IPO 80
Using -ip with -Qoption Specifiers 80
Inline Expansion of Funtions 81
Controlling Inline Expansion of User Functions 81
Criteria for Inline Function Expansion 82
Profile-guided Optimizations 83
Profile-guided Optimizations Overview 83
Profile-guided Optimizations Methodology 83
Basic PGO Options 84
Example of Profile-guided Optimization 84
PGO Environment Variables 85
Function Order List 86
Function Order List Usage Guidelines 86
Utilities for Profile-guided Optimization 88
PGO API: Profile Information Generation Support 89
PGO API Support Overview 89
Dumping Profile Information 89
Resetting the Dynamic Profile Counters 90
Dumping and Resetting Profile Information 90
Interval Profile Dumping 90
Environment Variable 91
High-level Language Optimizations (HLO) 92
HLO Overview 92

Loop Transformations 92
Loop Unrolling 93
Absence of Loop-carried Memory Dependency with IVDEP Directive 93
Parallelization 94
Parallelization Options Overview 94
Auto Parallelization 94
Auto-parallelizer's Diagnostic 95
Threshold for Auto-parallelization 95
Parallelization with OpenMP* 95
OpenMP* Standard Options 96
OpenMP* Run Time Library Routines 98
Intel Extensions to OpenMP* 99
Vectorization (IA-32 only) 101
Vectorization Overview 101
Vectorization Key Programming Guidelines 101
Data Dependence 102
Loop Constructs 103
Loop Exit Conditions 104
Types of Loops Vectorized 105
Stripmining and Cleanup 106
Statements in the Loop Body 107
Language Support and Directives 107
Vectorization Examples 112
Loop Interchange and Subscripts: Matrix Multiply 115
LIBRARIES 116
Libraries Overview 116
Default Libraries 116
Intel® Shared Libraries 117
Managing Libraries 118
DIAGNOSTICS AND MESSAGES 119
Diagnostic Overview 119
Language Diagnostics 119
Suppressing Warning Messages with lint Comments 119
Suppressing Warning Messages or Enabling Remarks 120
Limiting the Number of Errors Reported 120
Remark Messages 120
REFERENCE INFORMATION 121
Compiler Limits 121
Compiler Limits 121
Intel C++ Key Files 122

Key Files Summary for IA-32 Compiler 122
Key Files Summary for Itanium(TM) Compiler 123
Intel C++ Intrinsics Reference 124
Overview of the Intrinsics 124
Types of Intrinsics 124
Benefits of Using Intrinsics 125
Naming and Usage Syntax 128
Intrinsics Implementation Across All IA 129
Intrinsics For Implementation for All IA 129
Integer Arithmetic Related 129
Floating-point Related 130
String and Block Copy Related 132
Miscellaneous Intrinsics 132
MMX(TM) Technology Intrinsics 134
Support for MMX(TM) Technology 134
The EMMS Instruction: Why You Need It 134
EMMS Usage Guidelines 135
MMX™ Technology General Support Intrinsics 136
MMX(TM) Technology Packed Arithmetic Intrinsics 138
MMX(TM) Technology Shift Intrinsics 140
MMX(TM) Technology Logical Intrinsics 142
MMX(TM) Technology Compare Intrinsics 143
MMX(TM) Technology Set Intrinsics 144
MMX(TM) Technology Intrinsics on Itanium(TM) Architecture 146
Streaming SIMD Extensions 146
Intrinsics Support for Streaming SIMD Extensions 146
Floating-point Intrinsics for Streaming SIMD Extensions 147
Arithmetic Operations for Streaming SIMD Extensions 147
Logical Operations for Streaming SIMD Extensions 151
Comparisons for Streaming SIMD Extensions 152
Conversion Operations for Streaming SIMD Extensions 158
Load Operations for Streaming SIMD Extensions 160
Set Operations for Streaming SIMD Extensions 161
Store Operations for Streaming SIMD Extensions 162
Cacheability Support Using Streaming SIMD Extensions 163
Integer Intrinsics Using Streaming SIMD Extensions 164
Memory and Initialization Using Streaming SIMD Extensions 167
Miscellaneous Intrinsics Using Streaming SIMD Extensions 171
Using Streaming SIMD Extensions on Itanium(TM) Architecture 173
Macro Functions 174
Macro Function for Shuffle Using Streaming SIMD Extensions 174
Macro Functions to Read and Write the Control Registers 175
Macro Function for Matrix Transposition 176
Streaming SIMD Extensions 2 178
Overview of Streaming SIMD Extensions 2 Intrinsics 178
Floating Point Intrinsics 179
Floating-point Arithmetic Operations for Streaming SIMD Extensions 2 179

Logical Operations for Streaming SIMD Extensions 2 ... 183
Comparison Operations for Streaming SIMD Extensions 2 ... 184
Conversion Operations for Streaming SIMD Extensions 2 ... 191
Floating-point Memory and Initialization Operations ... 194
Streaming SIMD Extensions 2 Floating-point Memory and Initialization Operations ... 194
Load Operations for Streaming SIMD Extensions 2 ... 194
Set Operations for Streaming SIMD Extensions 2 ... 195
Store Operations for Streaming SIMD Extensions 2 ... 196
Miscellaneous Operations for Streaming SIMD Extensions 2 ... 197
Integer Intrinsics ... 198
Integer Arithmetic Operations for Streaming SIMD Extensions 2 ... 198
Integer Logical Operations for Streaming SIMD Extensions 2 ... 205
Integer Shift Operations for Streaming SIMD Extensions 2 ... 205
Integer Comparison Operations for Streaming SIMD Extensions 2 ... 209
Conversion Operations for Streaming SIMD Extensions 2 ... 211
Macro Function for Shuffle ... 212
Cacheability Support Operations for Streaming SIMD Extensions 2 ... 212
Miscellaneous Operations for Streaming SIMD Extensions 2 ... 214
Integer Memory and Initialization Operations ... 218
Streaming SIMD Extensions 2 Integer Memory and Initialization ... 218
Integer Load Operations for Streaming SIMD Extensions 2 ... 218
Integer Set Operations for Streaming SIMD Extensions 2 ... 219
Integer Store Operations for Streaming SIMD Extensions 2 ... 221
Intrinsics for Itanium(TM) Instructions ... 222
Overview of Intrinsics for Itanium(TM) Instructions ... 222
Native Intrinsics for Itanium(TM) Instructions ... 222
Lock and Atomic Operation Related Intrinsics ... 224
Operating System Related Intrinsics ... 226
Itanium(TM) Conversion Intrinsics ... 229
Register Names for getReg() and setReg() ... 229
Itanium(TM) Multimedia Additions ... 232
Data Alignment, Memory Allocation Intrinsics, and Inline Assembly ... 240
Overview of Data Alignment, Memory Allocation Intrinsics, and Inline Assembly ... 240
Alignment Support ... 240
Allocating and Freeing Aligned Memory Blocks ... 241
Inline Assembly ... 242
Intrinsics Cross-processor Implementation ... 244
Intrinsics Cross-processor Implementation ... 244
Intrinsics For Implementation Across All IA ... 244
MMX(TM) Technology Intrinsics Implementation ... 251
Streaming SIMD Extensions Intrinsics Implementation ... 255
Streaming SIMD Extensions 2 Intrinsics Implementation ... 262
Intel® C++ Class Libraries ... 273
Introduction to the Class Libraries ... 273
Welcome to the Class Libraries ... 273
Hardware and Software Requirements ... 273
About the Classes ... 273

Technical Overview 274
Details About the Libraries 274
C++ Classes and SIMD Operations 275
Capabilities 278
Integer Vector Classes 279
Integer Vector Classes 279
Terms, Conventions, and Syntax 280
Rules for Operators 282
Assignment Operator 284
Logical Operators 285
Addition and Subtraction Operators 286
Multiplication Operators 289
Shift Operators 291
Comparison Operators 292
Conditional Select Operators 294
Debug 296
Unpack Operators 299
Pack Operators 305
Clear MMX(TM) Instructions State Operator 305
Integer Intrinsics for Streaming SIMD Extensions 306
Conversions Between Fvec and Ivec 307
Floating-point Vector Classes 309
Floating-point Vector Classes 309
Fvec Notation Conventions 310
Data Alignment 311
Conversions 311
Constructors and Initialization 311
Arithmetic Operators 312
Minimum and Maximum Operators 316
Logical Operators 317
Compare Operators 319
Conditional Select Operators for Fvec Classes 323
Cacheability Support Operations 327
Debugging 327
Load and Store Operators 329
Unpack Operators for Fvec Operators 329
Move Mask Operator 330
Classes Quick Reference 330
Programming Example 338

About Intel(R) C++ Compiler

Welcome to the Intel® C++ Compiler

Welcome to the Intel® C++ Compiler. To use the compiler, you must have Red Hat* Linux* 7.1 or 7.2 operating system software installed on your computer.

The Red Hat 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 to gain an overview of the topics presented. For the latest information, visit the Intel Web site: http://developer.intel.com/design/perftool/cppontheweb.

What's New in This Release

Compiler for Two Architectures

This document combines information about Intel® C++ Compiler for IA-32-based applications and Itanium(TM)-based applications. IA-32-based applications correspond to the applications run on any Intel® Pentium® processor, Intel Celeron® processor, or Intel Xeon(TM).

Itanium-based applications correspond to the applications run on the Intel® Itanium(TM) processor.

The following variations of the compiler are provided for you to use according to your host system's processor architecture:

- **Intel® C++ Compiler for 32-bit Applications** is designed for IA-32 systems, and its command is `icc`. The IA-32 compilations run on any IA-32 Intel processor and produce applications that run only on IA-32 systems. This compiler can be optimized specifically for the Intel Pentium processor or Intel Pentium 4 processor.
- **Intel® C++ Itanium(TM) Compiler for Itanium(TM)-based Applications** is designed for Itanium architecture systems, and its command is `ecc`. This compiler runs on Itanium-based systems and produces Itanium-based applications. Itanium-based compilations can only operate on Itanium-based systems.

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 the 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 new 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 procedures
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 the Intel C++ documentation Web site where you will find links to:

- Intel C++ Compiler home page at http://developer.intel.com/software/products/compilers/c50/linux
- Related topics on the http://developer.intel.com Web site

For specific details on the Intel® Itanium(TM) 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
- Linux* system with glibc 2.2.2 or 2.2.4 and kernel 2.4. The compiler has been validated with Red Hat* Linux 7.1 and 7.2.

Itanium(TM) Processor System Requirements

Note

The native compilers for Itanium-based systems run on an Itanium-based system.

- A computer with an Itanium processor.
- 256 MB of RAM
- 100 MB of disk space
- Linux system with glibc 2.2.2 or 2.2.3 and kernel 2.4. The compiler has been validated with Red Hat Linux 7.1 and 7.2 for Intel® Itanium-based systems.

FLEXlm* Electronic Licensing

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

About This Document

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 of 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{i|M|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.

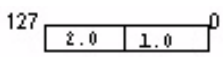
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
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(TM) Assembler User's Guide*.
- *Intel® Itanium(TM)-based Assembly Language Reference Manual*.

- *Itanium(TM) Architecture Software Developer's Manual Vol. 1: Application Architecture*, Intel Corporation, doc. number 245317-001.
- *Itanium(TM) Architecture Software Developer's Manual Vol. 2: System Architecture*, Intel Corporation, doc. number 245318-001.
- *Itanium(TM) Architecture Software Developer's Manual Vol. 3: Instruction Set Reference*, Intel Corporation, doc. number 245319-001.
- *Itanium(TM) Architecture Software Developer's Manual Vol. 4: Itanium(TM) 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.

Disclaimer

This *Intel® C++ Compiler User's Guide* as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Intel® C++ Compiler may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copyright © Intel Corporation 1996-2002.

Intel, Pentium, Itanium, Xeon, Celeron, and MMX are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Compiler Options Quick Reference Guides

Options Quick Reference Guides Overview

Conventions Used in the Options Quick Guide Tables

Convention	Definition
`[-]`	indicates that if option includes the "`-`", the option is disabled.
`[d]`	if present, indicates that the debug mode is implemented.
`[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 in Version 6.0

- Options specific to the Itanium(TM) architecture (Itanium-based systems only)
- Options available for both IA-32 and Itanium architecture

Option	Description	Default
`-falias`	Assume aliasing in program.	ON
`-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
`-fcode-asm`	Produce assembly file with optional code annotations.	OFF
`-fsource-asm`	Produce assembly file with optional source annotations.	OFF
`-fverbose-asm`	Produce assembly file with compiler comments.	ON
`-fnverbose-asm`	Produce assembly file with no compiler comments.	OFF
`-ftz` Itanium-based systems only	Flushes denormal results to zero.	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 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
`-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
`-openmpP`	Same as `openmp` (parallel).	OFF

Option	Description	Default
`-openmpS`	Enables the user to compile OpenMP* programs in sequential mode. The openmp directives are ignored, and a stub OpenMP library is linked (sequential).	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: • `min` • `med` • `max` If a *`level`* is not specified, `min` is used by default.	OFF
`-opt_report_phase`*`phase`*	Specifies the compilation *`phase`* for which reports are generated. The option can be used multiple times in the same compilation to get output from multiple phases. Valid *`phase`* arguments: • `ipo`: Interprocedural Optimizer • `hlo`: High Level Optimizer • `ilo`: Intermediate Language Scalar Optimizer • `ecg`: Electron 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
`-parallel`	Detects parallel loops capable of being executed safely in parallel and automatically generates multithreaded code for these loops.	OFF

Option	Description	Default
`-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
`-x[type]`	All source files found subsequent to `-x[type]` 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`: Assembly file • `assembler-with-cpp`: Assembly file that needs to be preprocessed. • `none`: Disable recognition and revert to file extension.	OFF

Alphabetical Listing

Compiler Options Quick Reference Guide

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

- Options specific to IA-32 architecture
- Options specific to the Itanium(TM) architecture
- Options available for both IA-32 and Itanium(TM) architecture

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[-]`	Enables [disables] assumption of the program's ANSI conformance.	OFF
`-ax{i|M|K|W}` IA-32 only	Generates specialized code for processor-specific codes `i`, `M`, `K`, `W` while also generating generic IA-32 code. • `i` = Pentium® Pro and Pentium II processor instructions • `M` = MMX(TM) instructions • `K` = streaming SIMD extensions • `W` = Pentium 4 processor instructions	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
`-D`*`name`*`[{=|#}`*`value`*`]`	Defines a macro *`name`* and associates it with the specified *`value`*.	OFF
`-dryrun`	Show driver tool commands but do not execute tools.	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

Option	Description	Default
`-falias`	Assume aliasing in program.	ON
`-fcode-asm`	Produce assembly file with optional code annotations.	OFF
`-fno-alias`	Assume no aliasing in program.	OFF
`-ffnalias`	Assume aliasing within functions	ON
`-fno-fnalais`	Assume no aliasing within functions, but assume aliasing across calls.	OFF
`-f[no]verbose-asm`	Produce assembly file with compiler components.	ON
`-fp` IA-32 only	Use EBP stack frame for all functions.	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
`-fsource-asm`	Produce assembly file with optional code annotations.	
`-ftz` Itanium-based systems only	Flushes denormal results to zero.	OFF
`-g`	Generates symbolic debugging information in the object code for use by source-level debuggers.	OFF
`-H`	Print "include" file order; don't compile.	OFF
`-help`	Prints compiler options summary.	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

Option	Description	Default
`-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`	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 assembly 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.	OFF
`-Kc++eh`	Enable C++ exception handling.	ON
`-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
`-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

Option	Description	Default
`-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
`-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.	OFF
`-O2`	Same as `-O1` on IA-32. Same as `-O` on Itanium-based systems.	ON
`-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
`-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.	`-openmp_report1`
`-openmpP`	Same as `openmp` (parallel).	OFF
`-openmpS`	Enables the user to compile OpenMP* programs in sequential mode. The openmp directives are ignored, and a stub OpenMP library is linked (sequential).	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

Option	Description	Default
`-opt_report_level[level]`	Specifies the verbosity `level` of the output. Valid `level` arguments: • `min` • `med` • `max` If a *`level`* is not specified, `min` is used by default.	OFF
`-opt_report_phase`*`name`*	Specifies the compilation *`phase`* for which reports are generated. The option can be used multiple times in the same compilation to get output from multiple phases. Valid *`phase`* arguments: • `ipo`: Interprocedural Optimizer • `hlo`: High Level Optimizer • `ilo`: Intermediate Language Scalar Optimizer • `ecg`: Electron 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

Option	Description	Default
`-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.	ON
`-pc80` IA-32 only	Set internal FPU precision to 64-bit significand.	OFF
`-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
`-Qansi[-]` Itanium-based systems only	Enable [disable] stating ANSI compliance of the compiled program and that optimizations can be based on the ANSI rules.	ON
`-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
`-restrict`	Enables pointer disambiguation with the `restrict` qualifier.	OFF
`-S`	Generates assembly files with `.s` suffix, then stops the compilation.	OFF
`-shared`	Produce a shared object.	OFF

Option	Description	Default
`-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. NOTE: This option is maintained for compatibility only on Itanium(TM)-based systems.	ON
`-static`	Prevents linking with shared libraries.	OFF
`-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
`-tpp5` IA-32 only	Targets the optimizations to the Intel® Pentium® processor.	OFF
`-tpp6` IA-32 only	Targets the optimizations to the Intel Pentium Pro, Pentium II and Pentium III processors.	ON (IA-32) OFF(Itanium-based systems)
`-tpp7` IA-32 only	Tunes code to favor the Intel Pentium 4 processor.	OFF
`-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 defined.	OFF
`-V`	Display compiler version information.	OFF
`-vec[-]` IA-32 only	Enable [disable] the vectorizer.	ON

Option	Description	Default
-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	-vec_report1
-w	Disable 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	OFF
-wd*L1*[,*L2*,...]	Disables diagnostics L1 through LN.	OFF
-we*L1*[,*L2*,...]	Changes severity of diagnostics L1 through LN to error.	OFF
-wn*n*	Limits the number of errors displayed prior to aborting compilation to *n* .	*n*=100
-wp_ipo	Compile all objects over entire program with multifile interprocedural optimizations. This option additionally makes the whole program assumption that all variables and functions seen in compiled sources are referenced only within those sources; the user must guarantee that this assumption is safe.	OFF
-wr*L1*[,*L2*,...]	Changes the severity of diagnostics L1 through LN to remark.	OFF
-ww*L1*[,*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

Option	Description	Default
`-x`*`type`*	All source files found subsequent to `-x`*`type`* 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` - assembly file • `assembler-with-cpp` - Assembly 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.	OFF
`-Xc`, `-ansi`	Select strict ANSI conformance dialect.	OFF
`-x{i|M|K|W}` IA-32 only	Generates specialized code to run exclusively on processors supporting the extensions indicated by processor-specific codes `i`, `M`, `K`, `W`. • `i` = Pentium® Pro and Pentium II processor instructions • `M` = MMX(TM) instructions • `K` = streaming SIMD extensions • `W` = Pentium 4 processor instructions	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.	`-Zp16`

Functional Groups Listing

Customizing Compilation Process Options

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
`-Aname[(values,...)]`	Associates a symbol *name* with the specified sequence of *values*. Equivalent to an `#assert` preprocessing directive.
`-A-`	Causes all predefined macros (other than those beginning with ___ and assertions to be inactive.
`-C`	Preserves comments in preprocessed source output.
`-Dname[(value)]`	Defines the macro *name* and associates it with the specified *value*. The default (`-Dname`) 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`	Same as `-E` but does not include `#line` directives in the output.
`-P`	Directs the preprocessor to expand your source module and store the result in a file in the current directory.
`-Uname`	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.
`-lname`	Link with a library indicated in *name*.

Option	Description
`-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 assembly files with `.s` suffix, then stops the compilation.
`-sox[-]` Itanium(TM)-based systems only	Enables [disables] the saving of compiler options and version information in the executable file.
`-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 assembly files with `.s` suffix, then stops the compilation.

Debugging Options

Option	Description
`-g`	Debugging information produced, `-O0` enabled, `-fp` enabled for IA-32-targeted compilations.
`-g -O2`	Debugging information produced, `-O2` 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.
`-g -ip`	Limited debugging information produced due to function inlining optimization, `-ip` option enabled.

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 Conformance Options

Conformance Options

Option	Description
-ansi[-]	Enables [disables] assumption of the program's ANSI conformance.
-mp	Favors conformance to the ANSI C and IEEE 754 standards for floating-point arithmetic. Behavior for NaN comparisons does not conform.

Application Performance Optimization Options

Optimization-level Options

Option	Description
-O0	Disables optimizations.
-O1	Enables optimizations. Optimizes for speed. -O1 disables inline expansion of library functions. For Itanium(TM) 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.

* -fp is an IA-32 only option and not applicable to compilations targeted for Itanium(TM)-based systems.

Floating-point Arithmetic Precision

Options for IA-32 and Itanium(TM)-based Systems

Option	Description
`-mp`	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 performance versus precision. Specifying this 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, it is spilled as 80 bits (extended precision), not 64 bits (double precision). • Floating-point arithmetic comparisons conform to IEEE 754 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 `-Op`, the extra bits of precision are not always rounded before the variable is reused. • The `-nolib_inline` option, which disables inline functions expansion, is used. **Note:** The `-nolib_inline` and `-mp` options are active by default when you choose the `-Xc` (strict ANSI C conformance) option.
`-long_double`	Use `-long_double` to change the size of the long double type to 80 bits. The Intel compiler's defalt 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.

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.

Option	Description
`-mp1`	Use the `-mp1` option to improve floating-point precision. `-mp1` disables fewer optimizations and has less impact on performance than `-mp`.
`-prec_div`	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*	Use the `-pcn` 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 `-pcn` 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) The default value for *n* is 64, indicating double precision.
`-rcd`	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.

Processor Dispatch Support (IA-32 only)

Option	Description
`-tpp5`	Optimizes for the Intel® Pentium® processor. Enables best performance for Pentium processor
`-tpp6`	Optimizes for the Intel Pentium Pro, Pentium II, and Pentium III processors. Enables best performance for the above processors
`-tpp7`	Optimizes for the Pentium 4 processor. Requires the RedHat* Linux* 7.1 and support of Streaming SIMD Extensions 2. Enables best performance for Pentium 4 processor
`-ax{i|M|K|W}`	Generates, in a single binary, code specialized to the extensions specified by the codes: • `i` Pentium Pro, Pentium II processors • `M` Pentium with MMX(TM) technology processor • `K` Pentium III processor • `W` Pentium 4 processor In addition, `-ax` generates generic IA-32 code. The generic code is usually slower.
`-x{i|M|K|W}`	Generate specialized code to run exclusively on the processors supporting the extensions indicated by the codes: • `i` Pentium Pro, Pentium II processors • `M` Pentium with MMX(TM) technology processor • `K` Pentium III processor • `W` Pentium 4 processor

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 assembly 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.
`-wp_ipo`	Compile all objects over entire program with multifile interprocedural optimizations. This option additionally makes the whole program assumption that all variables and functions seen in compiled sources are referenced only within those sources; the user must guarantee that this assumption is safe.

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(TM)-based applications, `-unroll[0]` used only for compatibility.
IA-32 Applications Only	
`-prefetch[-]`	Enables or disables prefetch insertion (requires `-O3`). Reduces wait time; optimum use is determined empirically.

Vectorization Options

Option	Description
`-ax{i|M|K|W}`	Enables the vectorizer and generates specialized and generic IA-32 code. The generic code is usually slower than the specialized code. `-vec-` disables vectorization, but processor-specific code continues to be generated.
`-x{i|M|K|W}`	Turns on the vectorizer and generates processor-specific specialized code. `-vec-` disables vectorization, but processor-specific code continues to be generated.
`-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.

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 for Windows* and Linux*

Compiler Options Cross Reference

Linux*	Windows*	Description	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[-]`	`-Qansi[-]`	Enable/disable assumption of ANSI conformance.	ON
`-ax{i|K|M|W}`	`-Qax{i|K|M|W}`	Generate code specialized for processor extensions specified by codes (`i,K,M,W`) while also generating generic IA-32 code. • `i` = Pentium® Pro and Pentium II processor instructions • `K` = Steaming SIMD extensions • `M` = MMX(TM) • `W` = Streaming SIMD Extensions 2	OFF
`-C`	`-C`	Don't strip comments.	OFF
`-c`	`-c`	Compile to object (`.o`) only, do not link.	OFF
`-D`*`name`*`[{=|#}{`*`text`*`}]`	`-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`	`-Hn`	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

Linux*	Windows*	Description	Default
`-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`	`NA`	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`*	`-o`*`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.	ON
`-pc80`	`-Qpc 80`	Set internal FPU precision to 64-bit significand.	OFF
`-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

Linux*	Windows*	Description	Default
-prof_file *filename*	NA	Specify filename 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
-w2	-w2	Enable remarks, warnings and errors.	OFF
-rcd	-Qrcd	Enable fast floating-point-to-integer conversions.	OFF
-restrict	-Qrestrict	Enable the restrict keyword for disambiguating pointers.	OFF
-S	-S	Generates assembly files with .s suffix, then stops the compilation.	OFF
-sox[-]	-Qsox	Enable (default)/disable saving of compiler options and version in the executable.	ON
-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	-V text	Display compiler version information.	OFF
-w	-w	Display errors.	OFF
-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

Linux*	Windows*	Description	Default
-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{i|K|M|W}	-Qx[i|M|K|W]	Generate code specialized for processor extensions specified by codes (i,K,M,W) while also generating generic IA-32 code. • i = Pentium® Pro and Pentium II processor instructions • K = Steaming SIMD extensions • M = MMX(TM) • W = Streaming SIMD Extensions 2	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

Getting Started with the Intel(R) C++ Compiler

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 Itaniun(TM)-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/compiler60/ia32/bin/iccvars.sh`
- **Itanium(TM)-based Systems:** `/opt/intel/compiler60/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/compiler60/ia32/bin/iccvars.sh
```

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

```
# set up environment for Intel compiler icc
 source /opt/intel/compiler60/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 for your platform:

- **IA-32 Systems:** `prompt> icc [options] file1 [file2. . .] [linker_options]`
- **Itanium(TM)-based Systems:** `prompt>ecc [options] file1 [file2 . . .] [linker_options]`

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 the C shell, you can edit your `.cshrc` file and add

```
setenv PATH  /usr/bin:<your path>
```

Then you can compile as

```
prompt>make -f your_makefile
```

Compiler Input Files

By default, the compiler recognizes `.cc`, `.cpp`, and `.cxx` files as C++ files. In examples, this documentation uses the `.cpp` extension for C++ files. The compiler recognizes files with the `.i` and `.c` extensions as C files. Also, the Intel® C++ Compiler recognizes the default filename extensions listed in the table below.

Default Filename Extensions

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`	assembly file	Assembled by the assembler

Default Behavior of the Compiler

Default Compiler Options

- Options specific to IA-32 architecture
- Options specific to the Itanium(TM) architecture
- Options available for both IA-32 and Itanium(TM) architecture

Option	Description	Default	Reference
`-ipo_obj`	Forces the compiler to create real object files when used with `-ipo`.	OFF (IA-32) ON (Itanium-based systems)	Interprocedural Optimization (IPO)
`-O1`	Enable optimizations.	ON	Optimization Choices
`-openmp_report{0|1|2}`	Controls the OpenMP* parallelizer's diagnostic levels.	`-openmp_report1`	Parallelization With OpenMP*
`-pc64` IA-32 only	Set internal FPU precision to 53-bit significand.	ON	
`-Qansi[-]` Itanium-based systems only	Enable [disable] stating ANSI compliance of the compiled program and that optimizations can be based on the ANSI rules.	ON	
`-sox[-]` IA-32 only	Enables [disables] the saving of compiler options and version information in the executable file. NOTE: This option is maintained for compatibility only on Itanium(TM)-based systems.	ON	
`-tpp6` IA-32 only	Targets the optimizations to the Intel Pentium Pro, Pentium II and Pentium III processors.	ON	Targeting a Processor and Extensions Support
`-vec[-]`	Enable [disable] the vectorizer.	ON	

Option	Description	Default	Reference
`-vec_report[n]` IA-32 only	Controls the amount of vectorizer diagnostic nformation. • *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	`-vec_report1`	Vectorizer Quick Reference
`-wn`	Control diagnostics. • *n* =0 displays errors (same as -w) • *n* =1 displays warnings and errors (DEFAULT) • *n* =2 displays remarks, warnings, and errors	`-w1`	Supressing Warning Messages
`-wnn`	Limits the number of errors displayed prior to aborting compilation to *n* .	*n*=100	Limiting the Number of Errors Reported
`-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.	`-Zp16`	Specifying Structure Tag Alignments

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.out`.
- Invokes options specified in a configuration file first. See Configuration Files.
- Searches for include files using the INCLUDE variable.
- 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, as described in Optimization Choices.

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.

IA-32-Specific Default

The vectorizor (`-vec`) is on by default.

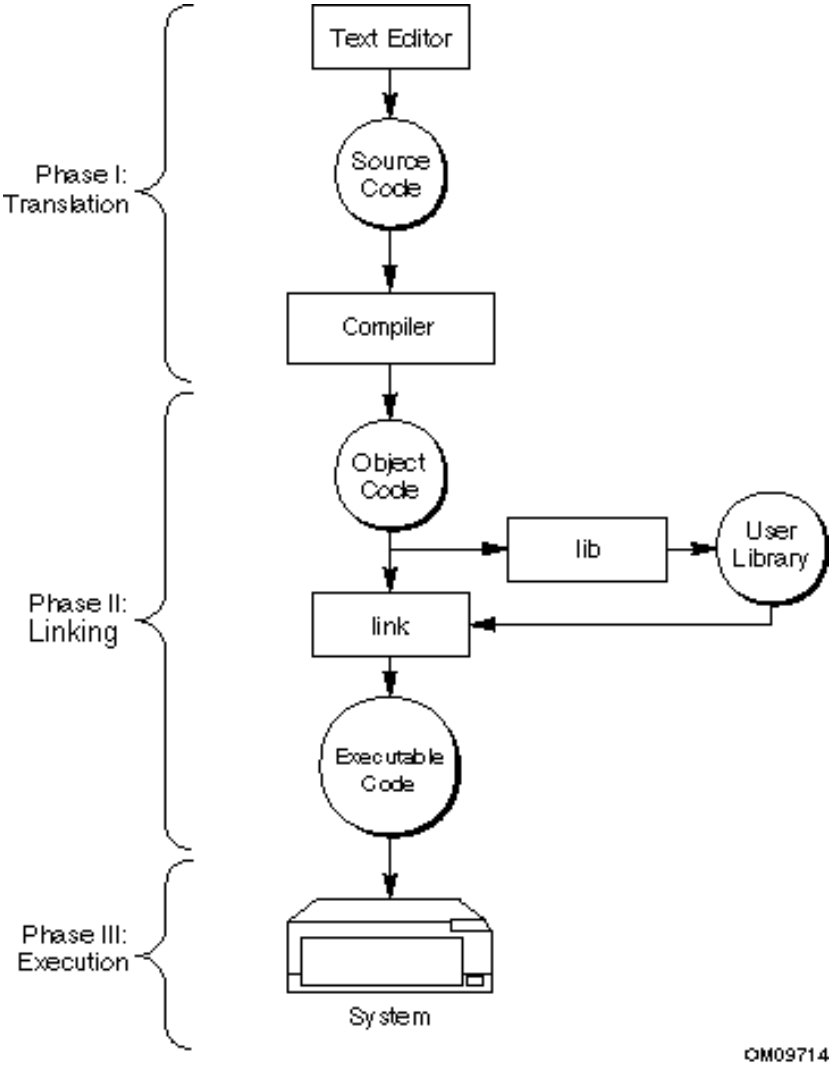
Compilation Phases

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

The compiler passes object files and any unrecognized filename 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 Compilation Environment

Customizing the Compilation Environment

For IA-32 and the Intel® Itanium(TM) 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 can search for special 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 compilation tools

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 directory path for the math libraries. Also, the compiler calls link, the GNU* linker, to produce an executable file from the object files. This linker searches the path specified in the `LD_LIBRARY_PATH` environment variable to find the libraries. Also, the assembler relies on `LD_LIBRARY_PATH` for the location of the associated libraries.
- `PATH` specifies the directory path for the compiler executable files.
- `INCLUDE` specifies the directory path for the "include" files.
- `ICCCFG` specifies a configuration file the compiler should use instead of the default configuration file for the IA-32 compiler.
- `ECCCFG` specifies a configuration file the compiler should use instead of the default configuration file for the Itanium(TM) compiler.
- `TMP` specifies the directory in which 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) – If you choose to install the Intel® C++ Compiler to a location other than the default location, you will need to modify the variable `IA32ROOT` in your environment to point to this location. It should point to the directory containing the `bin`, `lib`, and `include` directories.
- `IA64ROOT` (Itanium(TM)-based systems) -- If you choose to install the Intel C++ Compiler to a location other than the default location, you will need to modify the variable `IA64ROOT` in your environment to point to this location. It should point to the directory containing the `bin`, `lib`, and `include` 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 at the following locations (assuming you installed to the default directories):

- **IA-32 Systems:** `/opt/intel/compiler60/ia32/bin/iccvars.sh`
- **Itanium(TM)-based Systems:** `/opt/intel/compiler60/ia64/bin/eccvars.sh`

Running the Shell Scripts

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

```
prompt: . /opt/intel/compiler60/ia32/bin/iccvars.sh
```

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

```
# set up environment for Intel Compiler icc
 . /opt/intel/compiler60/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 options 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

Be aware that 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 for IA-32 Compilations

The following example illustrates how to write configuration files for IA-32-targeted compilations. 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`.

```
## Sample icc.cfg file.

## Define preprocessor macro MY_PROJECT. -DMY_PROJECT

## Additional directories to be searched for include

## files, before the default. -I /project/include
```

How to Use Configuration Files for Compilations on Itanium(TM)-based Systems

The following example illustrates how to write configuration files targeted for compilations on Itanium(TM)-based systems. 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 pound (#) character defines the text that follows as a comment.) For compilations on Itanium(TM)-based systems, the configuration file is `ecc.cfg`.

```
## Sample ecc.cfg 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; in this way you avoid editing the configuration file when changing projects.

Any number of options or filenames 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(TM)-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

By default, the compiler searches for the standard include files in the directories specified in the `INCLUDE` environment variable. You can indicate the location of include files in the configuration file.

How to Specify an Include Directory (-I)

Use the `-I`*`directory`* option to specify an additional directory in which to search for include files. For multiple search directories, multiple `-I`*`directory`* commands must be used. Included files are brought into the program with a `#include` preprocessor directive. 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
- directories specified in the `INCLUDE` environment variable

How to Remove Include Directories

Use the `-X` option to prevent the compiler from searching the default path specified by the `INCLUDE` environment variable.

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 newmain.cpp`**
- **Itanium(TM)-based Systems:** `prompt>`**`ecc -X -I/alt/include newmain.cpp`**

Customizing Compilation Process

Customizing Compilation Process Overview

This section describes options that customize the compilation process—preprocessing, compiling, linking and various compilation output and debug options.

Specifying Alternate Tools and Paths

You can direct the compiler to go outside default paths and tools 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:

`prompt>-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 (-Qoption, tool, optlist)

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:

`prompt>`**`-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 replace the space or tab with a comma, and 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.cpp`**
- **Itanium(TM)-based Systems:** `prompt>`**`ecc -Qoption,link,-Map,proto.map proto.cpp`**

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.

Preprocessing

Preprocessing Overview

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. The compiler preprocesses files as an optional first phase of the compilation.

Preprocessor Options

Use the options in this section to control preprocessing from the command line. If you specify neither option, the preprocessed source files are not saved but are passed directly to the compiler.

Option	**Description**
`-A`*`name`*`[(`*`value`*`)]`	Associates a symbol *`name`* with the specified sequence of *`values`*. Equivalent to an `#assert` preprocessing directive.
`-A-`	Causes all predefined macros (other than those beginning with ___ 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.

Option	Description
-E	Directs the preprocessor to expand your source module and write the result to standard output.
-EP	Same as -E but does not include #line directives in the output.
-P	Directs the preprocessor to expand your source module and store the result in a file in the current directory.
-U*name*	Suppresses any automatic definition for the specified macro *name* .

Preprocessing Only

Use either the -E or the -P option to preprocess your source files without compiling them.

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

- **IA-32 Systems:** prompt>**icc -E prog1.cpp prog2.cpp**
- **Itanium(TM)-based Systems:** prompt>**ecc -E prog1.cpp prog2.cpp**

When you specify the -P option, the preprocessor expands your source module and stores the result in a file in the current directory. There is no way to change the default name. The preprocessor uses the name of each source file with the .i extension. 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.cpp prog2.cpp**
- **Itanium(TM)-based Systems**: prompt>**ecc -P prog1.cpp prog2.cpp**

The -EP option can be used in combination with -E or -P. It directs the preprocessor to not include #line directives in the output. Specifying -EP alone is the same as specifying -E -EP.

Caution

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

Preserving Comments in Preprocessed Source Output

Use the -C option to preserve comments in your preprocessed source output.

Searching for Include Files

By default, the compiler searches for the standard include files in the directories specified in the INCLUDE environment variable. You can indicate the location of include files in the configuration file.

How to Specify an Include Directory

Use the `-Idirectory` option to specify an additional directory in which to search for include files. For multiple search directories, multiple `-Idirectory` commands must be used. Included files are brought into the program with a `#include` preprocessor directive. 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
- directories specified in the `INCLUDE` environment variable

How to Remove Include Directories

Use the `-X` option to prevent the compiler from searching the default path specified by the `INCLUDE` environment variable.

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 newmain.cpp`**
- **Itanium(TM)-based Systems:** `prompt>`**`ecc -X -I/alt/include newmain.cpp`**

Defining Macros

You can use the `-A` and `-D` options to define the assertion and macro names to be used during preprocessing. The `-U` option directs the preprocessor to suppress an automatic definition of a macro.

Use the `-A` option to make an assertion. This option performs the same function as the #assert preprocessor directive. The form of this option is:

`-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 value orange,banana use the following command:

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

The compiler provides a number of predefined macros. For a list of predefined macros available to the Intel® C++ Compiler, see the Predefined Macros table below.

Enter `-A-` to suppress all predefined macros, except for those beginning with the double underscore.

Use the `-D` option to define a macro. This option performs the same function as the #define preprocessor directive. The form of this option is:

`-Dname[{=|#}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.cpp`
- **Itanium(TM)-based Systems:** `prompt>ecc -DSIZE=100 prog1.cpp`

Use the `-Uname` option to suppress any automatic definition for the specified name. The `-U` option performs the same function as a `#undef` preprocessor directive. It can be used to undefine any macro, in addition to the predefined onces.

For more details about preprocessor directives, see a language reference such as C: A Reference Manual.

Predefined Macros

The predefined macros available for the Intel C++ Compiler compilations targeted for IA-32- and Itanium(TM)-based systems are described in the tables below. The Default column describes whether the macro is enabled (ON) or disabled (OFF) by default. The Disable column lists the option that disables the macro; no indicates that the macro cannot be disabled.

- Predefined macros for compilations targeted for IA-32 systems
- Predefined macros for compilations targeted for Itanium(TM)-based systems

Predefined Macros for Compilations Targeted for IA-32 Systems

Macro Name	Value	Disable	Description / When Used
`__INTEL_COMPILER`	600	no	Defines the compiler version. Defined as 600 for the Intel C++ Compiler V6.0. Always defined.
`__ICC`	600	no	Enables the Intel C++ Compiler. Assigned value refers to version of the compiler (e.g., 600 is 6.00). Supported for legacy reasons. Use `__INTEL_COMPILER` instead.
`__cplusplus`	C++ only	no	Defined when compiling C++ source.
`__EDG__`	1	no	

Predefined Macros for Compilations Targeted for Itanium(TM)-based Systems

Macro Name	Default	Disable	Description / When Used
`__INTEL_COMPILER`	600	no	Defines the compiler version. Defined as 600 for the Intel C++ Compiler V6.0. Always defined.
`__ECC`	600	no	Enables the Intel C++ Compiler. Assigned value refers to version of the compiler (e.g., 600 is 6.00). Supported for legacy reasons. Use `__INTEL_COMPILER` instead.
`__cplusplus`	C++ only	no	Enables compilation of C++ source.
`_INTEGRAL_MAX_BITS`	64	`-U`	Indicates support for the `__int64` type.
`_M_IA64`	64100	`-U`	Indicates the value for the preprocessor identifier to reflect the Itanium(TM) architecture.

Compiling

Compilation Overview

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 and obtain desired output file produced by the compiler.

Having control of the compilation process means, for example, that you can create a file at any of the compilation phases such as assembly, object, or executable with `-P` or `-c` options. Or you can name the output file or designate a set of options that are passed to the linker with the `-S`, `-o` options. 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.

You can use the command-line options discussed as tools to display and check for certain aspects of the compiler's behavior.

The options in this section provide you with the following capabilities:

- monitor the compilation to a phase or to a stage within a phase
- name the output files or directories

Compilation Options

Controlling Compilation

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

Last Phase Completed	Option	Compiler Input	Compiler Output
compile only	`-c`	source	Compile to object only (`.o`), do not link.
	`-S`	source	Generate assembly files with `.s` suffix and stops the compilation process.
syntax checking	`-syntax`	source files preprocessed files	diagnostic list
linking	(default)	source files preprocessed files assembly files object files library	executable file, map file
preprocessing	`-P`, `-E`, or `-Ep`	source files	preprocessed files

Monitoring Data Settings

The options described below provide monitoring the outcome of Intel compiler-generated code without interfering with the way your program runs.

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 `prog1.cpp`, use the following command:

- **IA-32 Systems:** `prompt>`**`icc -Zp2 prog1.cpp`**
- **Itanium(TM)-based Systems:** `prompt>`**`ecc -Zp2 prog1.cpp`**

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. The work-around implemented in the Intel® C++ Compiler avoids generating the susceptible instructions.

Assembly File Listing Example

This topic provides examples of IA-32 and Itanium(TM) architecture assembly file listings and explains how to read them.

IA-32 Assembly Listing Example

```
$B1$6:                          ; Preds $B1$9
        mov       eax, edx                          ;6.26
        shld      eax, esi, 11                      ;6.26
        or        eax, -2147483648                  ;6.26
        neg       ecx                               ;6.26
        add       ecx, 1054                         ;6.26
        shr       eax, cl                           ;6.26
        test      edx, edx                          ;6.26
        jge       $B1$5           ; Prob 50%        ;6.26
                                  ; LOE eax ebx ebp edi
```

The following list describes the annotations:

- The `; Preds` annotation lists all the basic-blocks that are predecessors of this basic-block.
- The `;6.26` annotation occurs next to every instruction and indicates the source line#.column number that this instruction is associated with. When a 0 appears it means that there is no source information associated with that particular instruction.
- The `; Prob` annotation indicates the probability that the conditional jump is taken. This is based either upon a "guess" by the compiler or from profile information from a `-prof_use` compilation.
- The `; LOE` line is the live-on-exit registers. Generally only the integer registers, xmm, and mm registers are printed.

Itanium(TM) Architecture Assembly Listing Example

The following is an example of a portion of an assembly file listing for compilations targeted for Itanium(TM)-based systems:

```
{   .mmi
          alloc   r34=ar.pfs,0,3,1,0                //: 25
          add     sp=-32,sp                         //: 25
          mov     r33=b0                            //: 25
    } { .mib
          add     r35=2,r0                          //: 26
          mov     r9=r0                             //: 26
          br.call.dpnt          b0=bark#;;          //: 26
}
```

The following list describes the annotations:

- `{` identifies the beginning of an bundle.
- `.mmi` and `.mib` identify the instruction template types; `.mmi` indicates two memory and one integer instructions; `.mib` indicates one memory, one integer, and one branch instruction.
- `}` identifies the end of an instruction bundle.
- `br.call.dpnt b0=bark#` identifies a call to the function bark.
- `;;` identifies the end of an instruction group.
- The number following the colon (:) in the comment at the end of each instruction indicates the source line number corresponding to that assembly language instruction.

Linking

This topic describes the options that allow you to control and customize the linking with tools and libraries and define the output of the linking process.

Option	Description
`-L`*`directory`*	Instruct 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`	This linker option instructs the compiler to build a Dynamic Shared Object (DSO) instead of an executable.
`-i_dynamic`	This linker 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.
`-Bstatic`	This linker option is used to statically link libraries at compile time. Compared to dynamic linking, results in larger executables.
`-Bdynamic`	This linker option is used to dynamically link libraries at run time. Compared to static linking, results in smaller executables.

Suppressing Linking

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

- **IA-32 Systems:** `prompt>icc -c file.cpp file2.cpp`
- **Itanium(TM)-based Systems:** `prompt>ecc -c file.cpp file2.cpp`

Note

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

Debugging

Debugging Options Overview

For compilations targeted to IA-32 processor systems, the compiler uses `-O0` as the default when you specify `-g`**.** Specifying the `-g` or `-O0` option automatically enables the `-fp` option for IA-32-targeted compilations. (Option `-fp` is not used for compilations targeted for Itanium(TM)-based systems.)

The `-fp` option (applies to IA-32 compilations only) is disabled by default or when `-O1` or `-O2` is specified 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. The `-fp` option instructs the compiler to generate code for IA-32-targeted compilations without turning off optimization, so that a debugger can still produce a stack backtrace. Using this option disables use of the EBP register in optimizations, and can result in slightly less efficient code.

Options	Descriptions
`-g`	Debugging information produced, `-O0` enabled, `-fp` enabled for IA-32-targeted compilations.
`-g -O2`	Debugging information produced, `-O2` optimizations enabled.
`-g -O3 -fp`	Debugging information produced, `-O3` optimizations enabled, `-fp` enabled for IA-32-targeted compilations.
`-g -ip`	Limited debugging information produced, `-ip` option enabled.

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 prog1.cpp`
- **Itanium(TM)-based Systems:** `prompt>ecc -g prog1.cpp`

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

Support for Symbolic Debugging

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

- If you specify the `-O1`, or `-O2` 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`, or `-O2` options, the `-fp` option will be disabled.

Parsing for Syntax 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 a file named `prog1.cpp`. Any diagnostics appear on the standard error output.

- **IA-32 Systems**: `prompt>icc -syntax prog1.cpp`
- **Itanium(TM)-based Systems**: `prompt>ecc -syntax prog1.cpp`

Language Conformance

Conformance to the C Standard

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

- C code that strictly adheres to the ANSI/ISO standard, or
- C code that contains extensions to this standard.

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.

Understanding the Extensions to ANSI/ISO Standard C Dialect

When you set the compiler to accept extensions to the ANSI/ISO standard, the compiler can process the following extensions:

Extension Type	Description
Files and data storage	Input files with no declarations. Incomplete array types for the last member of a structure, except when this is the only member of the structure. Incomplete struct or union type file-scope arrays. Note: The struct and union types must be completed before the array is subscripted. In addition, if the array is defined in the compilation, these types must be subscripted by the end of the compilation. enum tag names you define. You can declare an enum tag name and then define it later in the source file. Initializer expressions not enclosed in braces though they initialize any of the following: a full static array, structure, or union. (Standard C required the braces.)
Pointers	In initializers, pointer constant values cast to an integral type if the integral type is large enough to contain it. In integral constant expressions, integer constants cast to a pointer type and then cast back to an integral type. Assignments of pointers to integers and to other incompatible pointer types without explicit casts. Fields selected in the form p->m when the p variable is a pointer, including when p does not point to a struct or union that contains m. (All definitions of field must have the same type and offset within their structure or union.) Fields selected in the form x.m, including when x is not a structure or union containing m when (1) variable x is not a structure or union containing m and (2) the x variable is an lvalue. (All definitions of field must have the same type and offset within their structure or union.)
Types and syntax	Bit fields with enum base types or integral types other than int or unsigned int. long float as a synonym for double. Arbitrary text at the end of preprocessing directives. Numbers that do not comply with the pp-number syntax, because numbers are scanned according to the syntax for numbers when extensions are allowed. Example: The compiler would scan 0x123e+1 as three tokens. Under strict ANSI conformance mode, the compiler would use the pp-number syntax and scan this number as one invalid token.
Predicates	#assert and #unassert directives to define and test predicate names.

Extension Type	Description
Syntax with warnings	No warning given for an extra comma at the end of an enum list. Warning given when omitting the final semicolon preceding the closing brace(}) of a structure or union. Warning given for a right brace immediately following a label definition. (Normally, a statement must follow a legal definition.) No warning given for an empty declaration, a semicolon with nothing preceding it.
Semantics with warnings	Differences in assignments and pointers between pointers to types that are interchangeable but not identical, such as unsigned char* and char*. The compiler will not issue a warning in this case. A string constant assigned to a pointer to any kind of character. Comparison using >, >=, <, or <= operators between pointers to void and other kinds of pointers, without using an explicit type cast. (Strict ANSI dialect mode requires such comparisons using == or != and issues no warnings.) Inline assembly code inserted using the asm keyword. (Strict ANSI dialect mode requires the __asm keyword.) Freestanding tag declarations in the parameter declaration list for a function with old-style parameters.

How to Set the Compiler for Extended C Dialect

You set the compiler to accept extensions to the ANSI/ISO standard C code by using the `-Ze` option.

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:

Macro	Description
`__cplusplus`	Defines C++ programs only.
`__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 constant 1 when you set the compiler to accept only standard ANSI conformance. This macro is not defined for use when you set the compiler to accept extensions.
`__TIME__`	The time of compilation. As a string literal in the form `hh:mm:ss`.
`__TIMESTAMP__`	The date and time of the last modification of the current source file in the form:

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

Conformance to the C++ Standard

The Intel® C++ Compiler conforms to the ANSI/ISO standard (ISO/IEC 14882:1998) for the C++ language, with the following exceptions:

- Two-phase name binding in templates, as described in [`temp.res]` and [`temp.dep`] of the standard, is implemented only with `-Za` compiler option.
- Universal character set escapes (for example, `\uabcd`) are not implemented.
- The export keyword for templates is not implemented.

Optimizations

Optimization Levels

Setting Optimization Levels

Each of the command-line options: `-0`,`-01`, `-02` and `-03` turn on several compiler optimizations. `-0` and `-01` are similar and are only mentioned for compatibility with other compilers. The following table summarizes the optimizations that the compiler applies when you invoke `-01`, `-02`, or `-03` optimizations.

Option	Optimization	Affected Aspect of Program
`-01, -02`	global register allocation	register use
`-01, -02`	instruction scheduling	instruction reordering
`-01, -02`	register variable detection	register use
`-01, -02`	common subexpression elimination	constants and expression evaluation
`-01, -02`	dead-code elimination	instruction sequencing
`-01, -02`	variable renaming	register use
`-01, -02`	copy propagation	register use
`-01, -02`	constant propagation	constants and expression evaluation
`-01, -02`	strength reduction-induction variable	simplification instruction, selection-sequencing
`-01, -02`	tail recursion elimination	calls, further optimization
`-01, -02`	software pipelining	calls, further optimization
`-03`	prefetching, scalar replacement, loop transformations	memory access, instruction parallelism, predication, software pipelining

For IA-32 and Itanium(TM) architectures, the options can behave in a different way. To specify the optimizations for your program, use options for depending on the target architecture as follows.

IA-32 and Itanium(TM) compilers

`-O, -O1, -O2`	Confines optimizations to the procedural level. Turns ON intrinsics inlining. All three optimizations are equal.
`-O3`	Enables `-O2` option with more aggressive optimizations, for example: • prefetching • scalar replacement • loop transformations Optimizes for maximum speed, but may not improve performance for some programs.

Restricting Optimizations

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

Option	Description
`-O0`	Disables all optimizations.
`-nolib_inline`	Disable inline expansion of intrinsic functions.

Floating-point Optimizations

Restricting Floating-point Arithmetic Precision

The `-mp` option restricts some optimizations 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 performance versus precision.

Specifying this option has the following effects on program compilation:

- User variables declared as floating-point types are not assigned to registers.
- Floating-point arithmetic comparisons conform to IEEE 754 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 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.
- For IA-32 systems, whenever an expression is spilled, it is spilled as 80 bits (EXTENDED PRECISION), not 64 bits (DOUBLE PRECISION). Floating-point operations conform to IEEE 754. When assignments to type REAL and DOUBLE PRECISION are made, the precision is rounded from 80 bits (EXTENDED) down to 32 bits (REAL) or 64 bits (DOUBLE PRECISION). When you do not specify `-O0`, the extra bits of precision are not always rounded away before the variable is reused.
- Even if vectorization is enabled by the `-xK`, `-xW`, `-axK`, or `-axW` options, the compiler does not vectorize reduction loops (loops computing the dot product) and loops with mixed precision types.

Processor Dispatch Extensions Support (IA-32 only)

Targeting a Processor and Extensions Support

This section describes targeting a processor and processor dispatch options. `-tpp{5|6|7}` optimizes non-specifically for the IA-32 processor, while `-x{i|M|K|W}` and `-ax{i|M|K|W}` provide support to generate processor instruction extensions that are specific to the architecture.

Option	Description
`-tpp{5|6|7}`	Schedules instructions for optimal performance on the architecture specified by `5`, `6`, `7` • `-tpp5` Pentium® processor. • `-tpp6` Pentium Pro, Pentium II, and Pentium III processors. Default. • `-tpp7` Pentium 4 processor.
`-x{i|M|K|W}`	Generates specialized code to run exclusively on the processors supporting the extensions indicated by the `i`, `M`, `K`, `W` codes.
`-ax{i|M|K|W}`	Generates specialized code to run exclusively on the processors supporting the extensions indicated by the `i`, `M`, `K`, `W` codes while also generating generic IA-32 code in the same executable.

For example, on a Pentium III processor, if you have mostly integer code and only a small portion of floating-point code, you may want to compile with `-axM` rather than `-axK` because MMX(TM) technology extensions perform the best with integer data and the optimized code will run on a larger subset of Intel processors.

The `-ax` and `-x` options are backward compatible with the extensions supported. The Intel® Pentium 4 processor can run code targeted to any of the previous processors specified by `K`, `M`, or `i`.

Targeting a Processor (IA-32 only)

The Intel® C++ Compiler lets you choose whether to optimize the performance of your application for specific processors or to ensure your application can execute on a range of processors.

Optimizing for a Specific Processor without Excluding Others

Use the `-tpp{n}` option to optimize your application's performance for specific processors. Regardless of which `-tpp{n}` suboption you choose, your application is optimized to use all the benefits of that processor with the resulting binary file still capable of running on any of the processors listed.

To optimize for...	Use...
Pentium® and Pentium processor with MMX(TM) technology	`-tpp5`
Pentium Pro, Pentium II and Pentium III	`-tpp6` (default)
Pentium 4 Processor	`-tpp7`

For example, the following commands compile and optimize the source program `prog.cpp` for the Pentium Pro processor:

```
prompt> icc prog.cpp

prompt> icc -tpp6 prog.cpp
```

Exclusive Specialized Code (IA-32 only)

The `-x{i|M|K|W}` option specifies the minimum set of processor extensions required to exist on processors on which you execute your program. The resulting code can contain unconditional use of the specified processor extensions. When you use `-x{i|M|K|W}` the code generated by the compiler might not execute correctly on IA-32 processors that lack the specified extensions.

The following example compiles the program `myprog.cpp`, using the `i` extension. This means the program will require Intel® Pentium® Pro, Pentium II, or later, processors to execute.

```
prompt> icc -O2 tpp6 -xi -o myprog myprog.cpp
```

The resulting program, `myprog`, might not execute on a Pentium processor, but will execute on Pentium Pro, Pentium II, Pentium III, and Pentium 4 processors.

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

-x Summary

To Optimize for...	Use this option
Pentium Pro and Pentium II processors, which use the `CMOV`, `FCMOV`, and `FCOMI` instructions	`-xi`
Pentium processors with MMX(TM) technology instructions (does not imply `i` instructions).	`-xM`
Pentium III processor with the Streaming SIMD Extensions, implies `i` and `M` instructions	`-xK`
Pentium 4 processor with the Streaming SIMD Extensions 2, implies `i`, `M`, and `K` instructions	`-xW`

Specialized Code with -ax{i|M|K|W}

When the `-ax{i|M|K|W}` option is used, your compiled application includes processor-specific extensions. When the compiled application is run, it detects the extensions supported by the processor:

- If the processor supports the specialized extensions, the extensions are executed.
- If the processor does not support the specialized extensions, the extensions are not executed, and a more generic version of the code is executed instead.

Applications compiled with `-ax{i|M|K|W}` have increased code size, but increased performance over standard optimized code.

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

-ax Summary

To Optimize for...	Use this option
Intel® Pentium® Pro and Pentium II processors, which use the `CMOV` and `FCMOV`, and `FCOMI` instructions	`-axi`
Pentium processors with MMX(TM) technology instructions	`-axM`
Pentium III processor with the Streaming SIMD Extensions, implies `i` and `M` instructions	`-axK`
Pentium 4 processor with the Streaming SIMD Extensions 2, implies `i`, `M`, and `K` instructions	`-axW`

Checking for Performance Gain

The `-ax{i|M|K|W}` option directs the compiler to find opportunities to generate separate versions of functions that use instructions supported on the specified processors. 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 that function that will run on any IA-32 architecture 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 get large performance gains on more advanced processors, while still working properly on older processors.

The disadvantages of using `-ax{i|M|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.
- The runtime checks to determine which code to run slightly affect performance.

Combining Processor Target and Dispatch Options (IA-32 only)

The following table shows how to combine processor target and dispatch options to compile applications with different optimizations and exclusions.

Optimize exclusively for...	**...without excluding...**					
	Intel® Pentium® Processor	**Pentium Processor with MMX(TM) technology**	**Pentium Pro Processor**	**Pentium II Processor**	**Pentium III Processor**	**Pentium 4 Processor**
Pentium Processor	`-tpp5`	`-tpp5`	`-tpp6`	`-tpp6`	`-tpp6`	`-tpp7`
Pentium Processor with MMX(TM) technology	N-A	`-tpp5`, `-xM`	`-tpp6`, `-xM`	`-tpp6`, `-xM`	`-tpp6`, `-xM`	`-tpp7`, `-xM`
Pentium Pro Processor	N-A	N-A	`-tpp6`,`-xi`	`-tpp6`,`-xi`	`-tpp6`,`-xi`	`-tpp7`,`-xi`
Pentium II Processor	N-A	N-A	N-A	`-tpp6`,`-xiM`	`-tpp6`,`-xiM`	`-tpp7`,`-xiM`
Pentium III Processor	N-A	N-A	N-A	N-A	`-tpp6`,`-xK`	`-tpp7`,`-xK`
Pentium 4 Processor	N-A	N-A	N-A	N-A	N-A	`-tpp7`, `-xW`

Example of -x and -ax Combinations

If you wanted your application to

- always require the MMX(TM) technology extensions
- use Pentium Pro processor extensions when the processor it is run on offers it
- and to not use them when it does not

you could generate such an application with the following command line:

```
prompt>icc -O2 -xM -axi myprog.cpp
```

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

```
prompt>icc -O2 -axiM myprog.cpp
```

Note that this specifically optimized code will run only on processors that support both the `i` and `M` extensions.

Interprocedural Optimizations

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

IA-32 and Itanium(TM)-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.

With `-ip`, the compiler performs inline function expansion for calls to procedures defined within the current source file. However, when you use `-ipo` to specify multifile IPO, the compiler performs inline function expansion for calls to procedures defined in separate files.

The IPO optimizations are disabled by default.

Multifile IPO

Multifile IPO Overview

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.

`-wp_ipo` is a whole program assertion flag that tells the compiler the whole program is present. It enables multifile optimization with the whole program assumption that all user variables and user functions seen in the compiled sources are referenced only within those sources. This is an unsafe option. The user must guarantee that this assumption 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 assembly listing 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 assembly file is produced for each compiled source file.

Creating a Multifile IPO Executable

This topic describes how to enable multifile IPO for compilations targeted for IA-32 and Itanium(TM)-based systems.

Procedure for IA-32 Systems

Compile your modules with `-ipo` as follows:

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

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

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.cpp b.cpp c.cpp
```

Procedure for Itanium(TM)-based Systems

Compile your modules with `-ipo` as follows:

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

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.out b.out c.out
```

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.cpp b.cpp c.cpp
```

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 Using a Project Makefile

Most applications use a makefile or something similar to call a linker such as `link`. This is done automatically when you compile and link with the compiler. Therefore, when `-ipo` must result in a separate linking step, you must use the Intel linker driver `xild` instead, as follows:

```
prompt>xild -ipo link_command_line
```

`-ipo`	optional; enables additional IPO diagnostic output
`link_command_line`	is your linker command line

Use of `-ipo` is optional with `xild` for Multifile IPO in providing additional diagnostic output. You can use the `xild` syntax when you use a makefile instead of step 2 in the example Creating a Multifile IPO Executable. The following example places the multifile IPO executable in file name:

```
prompt>xild -o:filename a.out b.out c.out
```

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 cause your program to have incorrect behavior.

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 user.a a.out b.out
```

A library named `user.a` will be created containing `a.out` 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.

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

will produce `a.out` 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 -o user.a
```

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 assembly file. This option performs optimizations as described for `-ipo`, but stops prior to the final link stage, leaving an optimized assembly file. The default name for this file is `ipo_out.s`.

See Inline Expansion of Functions.

Using -ip 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:

`-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:

`-ip -Qoption,c,ip_specifier`

where `ip_specifier` is one of the specifiers described in the table that follows.

-option Specifiers

`-ip_args_in_regs=FALSE`	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 is set to the maximum number of 200.
`-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(TM) compiler: `ip_ninl_min_stats` = 15
`-ip_ninl_max_total_stats=n`	Sets the maximum increase in the `total_stats`. The number of intermediate language statements for each function that is expanded in line. The number is a positive integer. By default, each function can increase to a maximum of 5000 statements.
`-ip_no_external_ref`	Indicates that the source file contains the main program and does not contain functions that are referenced by external functions. If you do not specify this option, the compiler retains an original copy of each expanded in-line function.

The following command activates procedural and interprocedural optimizations on `source.cpp` 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.cpp
```

Inline Expansion of Funtions

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

For a routine to be considered for inlining, it has to meet certain minimum criteria. There are criteria to be met by the call-site, the caller, and the callee.

- The **call-site** is the site of the call to the function that might be inlined.
- The **caller** is the function that contains the call-site.
- The **callee** is the function being called that might be inlined.

Minimum call-site criteria:

- The number of actual arguments must match the number of formal arguments of the callee.
- The number of return values must be the same as the callees' number.
- The data types of the actual and formal arguments must be compatible.
- No multi-lingual inlining is allowed. Caller and callee must be written in the same source language.

Minimum criteria for the caller:

- At most, 2000 intermediate statements will be inlined into the caller from all the call-sites being inlined to the caller. You can change this value by specifying the option `-Qoption,c,-ip_ninl_max_total_stats=new value`
- The function must be called or have its address used if it is declared as static. Otherwise, it will be deleted.

Minimum criteria for the callee:

- Routines that contain the following substrings in their names are not inlined: abort, alloca, denied, err, exit, fail, fatal, fault, halt, init, interrupt, invalid, quit, rare, stop, timeout, trace, trap, and warn. Once these criteria are met, the compiler picks the routines whose inline expansions provide the greatest benefit to program performance. This is done using the following default heuristics. When you use profile-guided optimizations, a number of other heuristics are used.
- The default heuristic focuses on call-sites in loops or calls to functions containing loops.
- When profile information is available, the focus changes to the most frequently executed call-sites. Also, the default inline heuristic does not allow the inlining of functions with more than 230 intermediate statements, or the number specified by the option `-Qoption,c,-ip_ninl_max_stats`.
- The default inline heuristic stops when it detects direct recursion.
- The default heuristic will always inline very small functions that meet the minimum inline criteria. By default, functions are inlined. This limit can be modified with the option `-Qoption,c,-ip_ninl_min_stats`. Default for Itanium(TM)-based applications: `ip_ninl_min_stats = 15`. Default for IA-32 applications: `ip_ninl_min_stats` = 7.

Profile-guided Optimizations

Profile-guided Optimizations Overview

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 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 compiler 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

In this version 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 allows the compiler to make better decisions about function inlining, thereby increasing the effectiveness of interprocedural optimizations.

PGO Phases

The PGO methodology requires three phases:

1. instrumentation compilation and linking with `-prof_gen[x]`
2. instrumented execution by running the executable
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. 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.

Example of Profile-guided Optimization

The following is an example of the basic PGO phases:

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.cpp a2.cpp a3.cpp
prompt>icc a1.o a2.o a3.o
```

Itanium(TM)-based Systems

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

The resulting dynamic information file has a unique name and `.dyn` suffix every time you run `a.out`. 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.cpp a2.cpp a3.cpp
```

Itanium(TM)-based Systems

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

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 (`-ip` or `-ipo`) for phase 3. This example used `-O2` in phase 1 and `-O2 -ip` in phase 3.

Note

The compiler ignores the `-ip` or the `-ipo` options with `-prof_gen[x]`.

PGO Environment Variables

The "Profile-Guided Optimization Environment Variables" table below describes environment values to determine the directory in which 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.dpifile if .dyn files are newer than an existing pgopti.dpifile. 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.

Function Order List

Function Order List Usage Guidelines

A function order list is a text file that specifies the order in which the linker should link the non-static functions of your program. This improves the performance of your program by reducing paging and improving code locality. Profile-guided optimizations support the generation of a function order list to be used by the linker. The compiler determines the order using profile information.

To enable the Intel® C++ Compiler and `proforder` tool to generate a function order list, you must use the `-prof_gen[x]` and `-prof_dir` options described in the table below.

Option	Description
`-prof_gen[x]`	Generates an instrumented object file and creates a static profile information file (`.spi`), which contains source position information for the calls of each compiled function. This information, combined with the dynamic profile information from the `.dpi` file, enables optimized ordering of functions. When you use `-prof_gen[x]` instead of `-prof_gen[x]`, you can use the `proforder` tool to create a function order list for the linker. However, `-prof_gen[x]` also requires more memory at runtime, produces larger `.dyn` files, and disables execution of parallel make files.
`-prof_dir` *dirname*	Specifies the directory where `.dyn` files are to be created. The default is the directory where the program is compiled. The specified directory must already exist. You should specify the same `-prof_dir` option for both the instrumentation and feedback compilations. If you move the `.dyn` files, you need to specify the new path.

You will need to use the utilities `profmerge` and `proforder` described in Utilities for Profile-Guided Optimization.

Use the following guidelines to create a function order list:

- The order list only affects the order of non-static functions.
- Do not use `-prof_gen[x]` to compile two files from the same program simultaneously. This means that you cannot use the `-prof_gen[x]` option with parallel makefile utilities.
- You must compile to enable function-level linking. This option is active when you specify `-O`, `-O1`, `-O2`, or `-O3`.

Function Order List Example

Assume you have a C program that consists of files `file1.c` and `file2.c` and that you have created a directory for the profile data files in `/home/usr/profdata`. Do the following to generate and use a function order list.

1. Compile your program by specifying `-prof_gen[x]` and `-prof_dir`: **IA-32 Systems** `prompt>`**`icc -oMYPROG -prof_genx -prof_dir /home/usr/profdata file1.c file2.c`** **Itanium(TM)-based Systems** `prompt>`**`ecc -oMYPROG -prof_genx -prof_dir /home/usr/profdata file1.c file2.c`**
2. Run the instrumented program on one or more sets of input data `prompt>./MYPROG`
3. The program produces a `.dyn` file each time it is executed.
4. Merge the data from one or more runs of the instrumented program using the `profmerge` tool to produce the `pgopti.dpi` file. `prompt>`**`profmerge -prof_dir /home/usr/profdata`**

5. Generate the function order list using the `proforder` tool. By default, the function order list is produced in the file `proford.txt`. `prompt>`**`proforder -prof_dir /home/usr/profdata -o MYPROG.txt`**

6. Compile your application with profile feedback by specifying the `-prof_use` and the `/ORDER` option to the linker. Again, use the `-prof_dir` option to specify the location of the profile files. **IA-32 Systems** `prompt>`**`icc -oMYPROG -prof_use -prof_dir /home/usr/profdata file1.c file2.c -link /ORDER:@MYPROG.txt`** **Itanium(TM)-based Systems** `prompt>`**`ecc -oMYPROG -prof_use -prof_dir /home/usr/profdata file1.c file2.c -link /ORDER:@MYPROG.txt`**

Comparison of Function Order Lists and IPO Code Layout

The Intel C++ Compiler provides two methods of optimizing the layout of functions in the executable:

1. use of a function order list
2. use of `-ipo`

Each method has its advantages. A function order list, created with `proforder`, enables you to optimize the layout of non-static functions; that is, external and library functions whose names are exposed to the linker. The linker cannot directly affect the layout order for static functions because the names of these functions are not available in the object files.

On the other hand, using `-ipo` allows you to optimize the layout of all static or extern functions compiled with the Intel C++ Compiler. The compiler cannot affect the layout order for functions it does not compile, such as library functions. The function layout optimization is performed automatically when IPO is active.

Function Order List Effects

Function Type	Code Layout with -ipo	Function Ordering with proforder
Static	X	No effect.
Extern	X	X
Library	No effect.	X

Function Call to Dump Profile Data Explicitly

As part of the instrumented execution phase of profile-guided optimization, the instrumented program writes profile data to the dynamic information file (`.dyn` file). The file is written after the instrumented program returns normally from `main()` or calls the standard C `exit` function. For programs that do not terminate normally, the `_PGOPTI_Prof_Dump` function is provided. During the instrumentation compilation (`-prof_gen`), you can add a call to this function to your program. You should add the following function prototype prior to the call:

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

Note

You must remove the call or comment it out prior to the feedback compilation with `-prof_use`.

Utilities for Profile-guided Optimization

The `profmerge` and `proforder` tools are used when generating a function order list.

The profmerge Tool

Use `profmerge` to merge dynamic profile information (`.dyn`) files. The compiler executes this tool automatically during the feedback compilation phase when you specify `-prof_use`. You can also invoke it as follows:

- **IA-32 Systems:** `prompt>`**`profmerge [-prof_dir`** *`dir_name`*`]`
- **Itanium(TM)-based Systems:** `prompt>`**`profmerge -em -p64 [-prof_dir`** *`dir_name`*`]`

This merges all `.dyn` files in the current directory or the directory specified by `-prof_dir`, and produces the summary file `pgopti.dpi`.

The proforder Tool

Use `proforder` to generate a function order list for use with the `/ORDER` linker option. The syntax for this tool is as follows:

`prompt>`**`proforder [-prof_dir`** *`dir_name`*`] [-o`** *`order_file`*`]`

Argument	Description
`dir_name`	the directory containing the profile files (`.dpi`, `.dyn`, and `.spi`)
`order_file`	the optional name of the function order list file. The default name is `proford.txt`.

The `proforder` utility is used as part of the feedback compilation phase to improve program performance.

PGO API: Profile Information Generation Support

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.

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 allows 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.

High-level Language Optimizations (HLO)

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 Fortran and C++. They include loop interchange, loop fusion, loop unrolling, loop distribution, unroll-and-jam, blocking, data prefetch, scalar replacement, data layout optimizations and some others. The option that turns on the high-level optimizations is `-O3`.

IA-32 and Itanium(TM)-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(TM) architectures above, the Itanium(TM) 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.cpp`

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.cpp`

Absence of Loop-carried Memory Dependency with IVDEP Directive

For Itanium(TM)-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.

```
#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(). /a[]` for `C/`

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

Parallelization

Parallelization Options Overview

For shared memory parallel programming, the Intel® C++ Compiler supports the OpenMP, version 1.0 API. The Parallelization capability of the Intel C++ Compiler uses the following options.

Option	Description
`-parallel`	Enables the auto-parallelizer to generate multi-threaded 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. Default: OFF
`-par_report{0|1|2|3}`	Controls the auto-parallelizer's diagnostic levels. Default: `-par_report1`.
`-openmp`	Enables the parallelizer to generate multi-threaded code based on the OpenMP directives. Default: OFF
`-openmp_report{0|1|2}`	Controls the OpenMP parallelizer's diagnostic levels. Default: `-openmp_report1`

Auto Parallelization

In addition to the low-level "hand-thread" approach based on threading API for Win32* and Itanium™-based Windows* 2000, the Intel® C++ Compiler with the auto-parallelization feature and a high-level symmetric multi-rpocessing (SMP) programming model enable the user with an easy way to exploit the parallelism on SMP systems.

Enabling Auto-parallelizer

To enable 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. Automatic parallelization relieves the user from having to deal with the low-level details of iteration partitioning, data sharing, thread scheduling and synchronizations. It also provides the benefit of the performance available from multiprocessor systems.

Guidelines for Effective Auto-parallelization Usage

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, procedure calls or global references.

Auto-parallelization Environment Variables

Option	Description	Default
OMP_NUM_THREADS	Controls the number of threads used.	Number of processors currently installed in the system.
OMP_SCHEDULE	Specifies the type of run time scheduling.	STATIC

Auto-parallelizer's Diagnostic

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).
- -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.

Threshold for Auto-parallelization

The -Qpar_threshold{n} option 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. Default is n=75. This option is used for loops whose computation work volume cannot be determined at compile-time.

- -Qpar_threshold0 - loops get auto-parallelized regardless of computation work volume.
- -Qpar_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 speedup. For example, n=50 would mean parallelize only if there is a 50% probability of the code speeding up if executed in parallel. 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.

Parallelization with OpenMP*

The OpenMP* C/C++ API has recently emerged as the de facto standard for shared memory, parallel programming. It shelters you from having to deal with the low-level details of iteration partitioning, data sharing, thread scheduling, and synchronization. The Intel® C++ Compiler supports the OpenMP* API version 1.0 and performs code transformation to generate multithreaded code automatically as determined by your OpenMP* directive annotations to the program.

Note

As with many advanced features of compilers, you must be sure to properly understand the functionality of the auto-parallelization switches in order to use them effectively and avoid unwanted program behavior.

OpenMP* Parallelization Reference

Option	Description
`-openmp`	Enables the parallelizer to generate multi-threaded code based on the OpenMP* directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems. The `-openmp` option only works at an optimization level of `-O2` (the default) or higher.
`-openmp_report{0|1|2}`	Controls the output of diagnostic messages. The level of the message output is controlled by `0`, `1`, or `2`. • `0` = no diagnostic information is displayed. • `1` = display diagnostics indicating loops, regions, and sections successfully parallelized (default). • `2` = same as 1 plus diagnostics indicating master construct, single construct, critical sections, order construct, atomic directive, etc. successfully handled.

OpenMP* Standard Options

For complete information on the OpenMP* standard, visit the http://www.openmp.org Web site. The Intel Extensions to OpenMP* topic describes the extensions to the standard that have been added by Intel in the Intel® C++ Compiler.

OpenMP* C/C++ Directives

An OpenMP* directive has the form:

```
#pragma omp directive [directive clause . . . ]
```

The following tables list and describe OpenMP* directives and clauses.

Directive	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 constuct that specifies a set of constucts that are to be divided among threads in a team.
`section`	Indicates that the associated code block should be executed in parallel.
`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.
`parallel sections`	Provides a shortcut form for specifying a parallel region containing a single sections directive.
`master`	Identifies a constuct that specifies a structured block that is executed by the master thread of the team.
`critical`	Identifies a construct that restricts execution of the associated structured block to a single thread at a time.

Directive	Description
`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.
`threadprivate`	Makes the named file-scope or namespace-scope variables specified private to a thread but file-scope visible within the thread.
`ordered`	The structured block following an ordered directive

Clauses	Description
`private`	Declares variables to be private to each thread in a team.
`firstprivate`	A private copy of the private variable is created for each thread. In addition, each new private object is initialized with the value of the original object.
`lastprivate`	A private copy of the private variable is created for each thread. In addition, the last iteration's value of each lastprivate is assigned to the original object.
`shared`	Shares variables among all the threads in a team.
`default`	Allows you to affect the data-scope attributes of variables.
`reduction`	Performs a reduction on scalar variables.
`nowait`	Specifies that threads that finish the loop early may continue executing code after the loop without waiting for the remaining threads to finish.
`if`	If `if(scalar_logical_expression)` clause is present, the enclosed code block is executed in parallel only if the `scalar_logical_expression` is true. Otherwise, the code block is serialized.
`ordered`	Must be present when ordered directives are contained in the dynamic extent of the for construct.
`schedule`	Specifies how iterations of the 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* Environment Variables

Variable	Description	Default
`OMP_SCHEDULE`	Sets the run-time schedule type and chunk size.	STATIC
`OMP_NUM_THREADS`	Sets the number of threads to use during execution.	Number of processors
`OMP_DYNAMIC`	Enables or disables the dynamic adjustment of the number of threads.	FALSE
`OMP_NESTED`	Enables or disables nested parallelism.	FALSE

OpenMP* Run Time Library Routines

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

The following table specifies the interface to these routines. The names for the routines are in user namespace. `omp.h` is provided in the include directory of your compiler installation. There are definitions for two different locks, `omp_lock_t` and `omp_nest_lock_t`, which are used by the functions in the table.

Function	Description
`void omp_set_num_threads(int num_threads)`	Dynamically set the number of threads to use for this region.
`int omp_get_num_threads(void)`	Determine what the current number of threads is that is allowed to execute a region.
`int omp_get_max_threads(void)`	Obtains the maximum number of threads ever allowed with this OpenMP* implementation.
`int omp_get_thread_num(void)`	Determines the unique thread number of the thread currently executing this section of code.
`int omp_get_num_procs(void)`	Determines the number of processors on the current machine.
`int omp_in_parallel(void)`	Returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel, otherwise it returns zero.
`void omp_set_dynamic(int dynamic_threads)`	Enable or disable dynamic adjustment of the number of threads used to execute a parallel region. If `dynamic_threads` is non-zero, dynamic threads are enabled. If `dynamic_threads` is zero, dynamic threads are disabled.
`int omp_get_dynamic(void)`	Returns non-zero if dynamic thread adjustment is enabled and returns zero otherwise.
`void omp_set_nested(int nested)`	Enable or disable nested parallelism. If parameter is non-zero, enable. Default is disabled.
`int omp_get_nested(void)`	Always returns zero in the current version of compiler.
`void omp_init_lock(omp_lock_t *lock)`	Initialize a unique lock and set lock to point to it.
`void omp_destroy_lock(omp_lock_t *lock)`	Disassociate lock from any locks.

Function	Description
`void omp_set_lock(omp_lock_t *lock)`	Force 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.
`void omp_unset_lock(omp_lock_t *lock)`	Release executing thread from ownership of lock associated with lock. lock must be initialized via `omp_init_lock()`, and behavior undefined if executing thread does not own the lock associated with lock.
`int omp_test_lock(omp_lock_t *lock);`	Attempt to set lock associated with lock. If successful, return non-zero. lock must be initialized via `omp_init_lock()`.
`void omp_init_nest_lock(omp_nest_lock_t *lock)`	Initialize a unique nested lock and set lock to point to it.
`void omp_destroy_nest_lock(omp_nest_lock_t *lock)`	Disassociate the nested lock lock from any locks.
`void omp_set_nest_lock(omp_nest_lock_t *lock)`	Force 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
`void omp_unset_nest_lock(omp_nest_lock_t *lock)`	Release executing thread from ownership of lock associated with lock if count is zero. lock must be initialized via `omp_init_nest_lock()`. Behavior is undefined if executing thread does not own the lock associated with lock.
`int omp_test_nest_lock(omp_nest_lock_t *lock)`	Attempt to set lock associated with lock. If successful, return nesting count, otherwise return zero. lock must be initialized via `omp_init_lock()`.

Intel Extensions to OpenMP*

For complete information on the OpenMP* standard, visit the Web site http://www.openmp.org. This topic describes the extensions to the standard that have been added by Intel in the Intel® C++ Compiler.

Environment Variables

Environment Variable	Description
`KMP_STACKSIZE`	Used to set the number of bytes that will be allocated for each parallel thread to use as its private stack.
`KMP_BLOCKTIME`	Used to set the integer value of time, in milliseconds, that the libraries wait after completing the execution of a parallel region before putting threads to sleep.
`KMP_SPIN_COUNT`	Used to help fine-tune the critical section.

Thread-level malloc( )

The Intel C++ Compiler implements an extension to the OpenMP* run-time library to allow threads to allocate memory from a heap local to each thread.

The memory allocated by these routines must also be freed by these routines. While it is legal for the memory to be allocated by one thread and freed by a different thread, this mode of operation has a slight performance penalty.

The interface is identical to the `malloc()` interface except the entry points are prefixed with `kmp_`, as shown below:

```
#include omp.h
        void * kmp_malloc( size_t );
        void * kmp_calloc( size_t, size_t );
        void * kmp_realloc( void *, size_t );
        void   kmp_free( void * );
```

Vectorization (IA-32 only)

Vectorization Overview

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

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

```
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]
     RITE data[2]
```

In the normal sequential version of the loop shown, the value of `data[1]` read from during the second iteration was written to in 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 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:

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

...

while (count != 1b)

{

     /* 1b is not affected within loop */

     a[i] = b[i] * x;
     b[i] = c[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(R) 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

These pragmas control the vectorization of only the subsequent loop in the program, but the compiler does not apply them to any nested loops. Each nested loop needs its own `pragma` preceding it in order for the `pragma` to be applied. You must place a `pragma` only before the loop control statement.

Name: #pragma vector always

Syntax: `#pragma vector always`

Definition: This pragma instructs the compiler to override any efficiency heuristic during the decision to vectorize or not. `#pragma vector always` will vectorize non-unit strides or very unaligned memory accesses.

Example:

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

Name: #pragma ivdep

Syntax: `#pragma ivdep`

Definition: This `pragma` 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 `pragma` overrides that decision. Only use this when you know that the assumed loop dependences are safe to ignore.

The loop in this example will not vectorize with the `ivdep pragma`, since the value of `k` is not known (vectorization would be illegal if `k`<0).

Example:

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

Name: #pragma vector

Syntax: `#pragma vector{aligned | unaligned}`

Definition: The vector loop `pragma` 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 with this `pragma`, the loop should be vectorized using `aligned` (or `unaligned`) operations. Specify one and only one of `aligned` or `unaligned`.

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:

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

The compiler has at its disposal 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.

Alignment Strategies Example

```
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;
}
```

Name: #pragma novector

Syntax: `#pragma novector`

Definition: The `novector` loop `pragma` 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 `#pragma novector` to tell the compiler not to vectorize, even if the loop is considered vectorizable.

Example: #pragma novector

```
void foo (int lb, int ub)

{

     #pragma novector

     for (j = lb; j < ub; j++)

     {

          a[j] = a[j] + b[j];

     }

}
```

Dynamic Dependence Testing Example

```
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

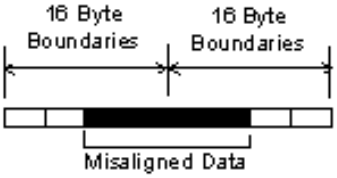
```
void vec_copy(float *restrict dest, float *restrict src, int len)
{
     int i;
     for (i = 0; i < len; i++)
     {
          dest[i] = src[i];
     }
}
```

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



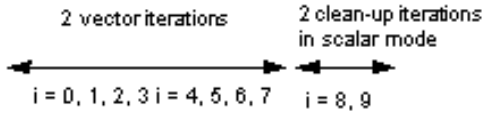
For example, if you know that elements `a[0]` and `b[0]` are aligned on a 16-byte boundary, then the following loop can be vectorized with the alignment option on (`#pragma vector aligned`):

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



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];
     }
}
```

The use of `assert` checks that `lb` is a multiple of 4.

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];
          }
     }
}
```

Libraries

Libraries Overview

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 compiler allows you to use all the standard run-time libraries. By default, the compiler automatically expands a number of standard C, C++, and math library functions. For more information, see Inline Expansion of Library Functions.

The following libraries are supplied.

Library	Description
`libc.a`	GNU* C library (included with Red Hat* Linux*)
`libguide.a`	for OpenMP* implementation
`libsvml.a`	short vector math library
`libirc.a`	Intel support library for PGO and CPU dispatch
`libimf.a`	Intel math library
`libcprts.a`	Dinkumware C++ Library
`libunwind.a`	Unwinder library
`libcxa.a`	Intel support library for EH and RTTI

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 hello.cpp with mylib.a, use the following command:

- **IA-32 Systems:** `prompt>icc -ohello hello.cpp mylib.a`
- **Itanium(TM)-based Systems:** `prompt>ecc -ohello hello.cpp mylib.a`

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

In the compiler package, you received the Intel math library, `libimf.a`, which contains optimized versions of the math functions in the standard C run-time library. The functions in the library are optimized for program execution speed on the Pentium® processor.

To enable the optimized math library, the installation creates a directory for `libimf.a` and adds the new directory path to the `LD_LIBRARY_PATH` variable. Intel recommends you keep `libimf.a` in the first directory specified in the path.

Intel® Shared Libraries

The Intel® C++ Compiler (both IA-32 and Itanium(TM) compilers) links the 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 to conform to C++ ABI.
- 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.

The libraries `libcprts.a` and `libcxa.so` are C++ language support libraries used by Fortran when Fortran includes code written in C++.

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 *myprog.cpp***

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 *myprog.cpp***

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 semicolon-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 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. For more information on adding library names to the response file and the configuration file, see Response Files and Configuration Files.

To specify a library name on the command line, you must first add the library's path to the `LIB` environment variable. Then, to compile `file.cpp` and link it with the library `mylib.a`, enter the following command:

- **IA-32 Systems:** `prompt>`**`icc file.cpp mylib.a`**
- **Itanium(TM)-based Systems:** `prompt>`**`ecc file.cpp 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

Diagnostics and Messages

Diagnostic Overview

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.

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 `-Wn` 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.cpp` and displays compiler errors, but not warnings:

- **IA-32 System:** `prompt>`**`icc -W0 newprog.cpp`**
- **Itanium(TM)-based System:** `prompt>`**`ecc -W0 newprog.cpp`**

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.cpp`, compilation aborts.

- **IA-32 Systems:** `prompt>`**`icc -wn50 -c a.cpp`**
- **Itanium(TM)-based Systems:** `prompt>`**`ecc -wn50 -c a.cpp`**

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`

Reference Information

Compiler Limits

Compiler Limits

The Compiler Limits 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

Intel C++ Key Files

Key Files Summary for IA-32 Compiler

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

/bin Files

File	Description
`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

File	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(TM) Compiler

The following tables list and briefly describe files that are installed for use by the Itanium(TM) compiler version of the compiler.

/bin Files

File	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

File	Description
libcprts.a	C++ standard language library
libcxa.so	C++ language library indicating I/O data location
libirc.a	Intel-specific library (optimizations)
libm.a	Math library
libguide.a	OpenMP library
libguide.so	Shared OpenMP library
libmofl.a	Multiple Object Format Library, used by the Intel assembler
libmofl.so	Shared Multiple Object Format Library, used by the Intel assembler
libunwinder.a	Unwinder library
libintrins.a	Intrinsic functions library

Intel C++ Intrinsics Reference

Overview of the Intrinsics

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(TM) 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(TM)-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

Processors	MMX(TM) Technology Intrinsics	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium™ Processor Instructions
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:

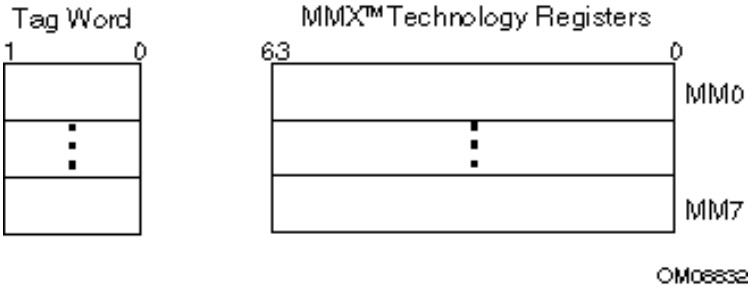
- New Registers--Enable packed data of up to 128 bits in length for optimal SIMD processing.
- New Data Types--Enable packing of up to 16 elements of data in one register.

The Streaming SIMD Extensions 2 intrinsics are defined only for IA-32, not for Itanium(TM)-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

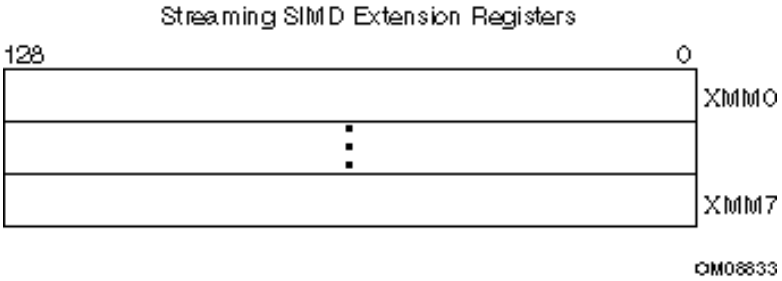
A key feature provided by the architecture of the processors are new register sets. The MMX instructions use eight 64-bit registers (`mm0` to `mm7`) which are aliased on the floating-point stack registers.

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(TM) 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.

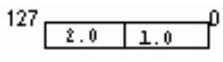
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
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 Implementation Across All IA

Intrinsics For Implementation for All IA

The intrinsics in this section function across all IA-32 and Itanium(TM)-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.

Intrinsic	Description
`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(TM)-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(TM) 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.

MMX(TM) Technology Intrinsics

Support for MMX(TM) Technology

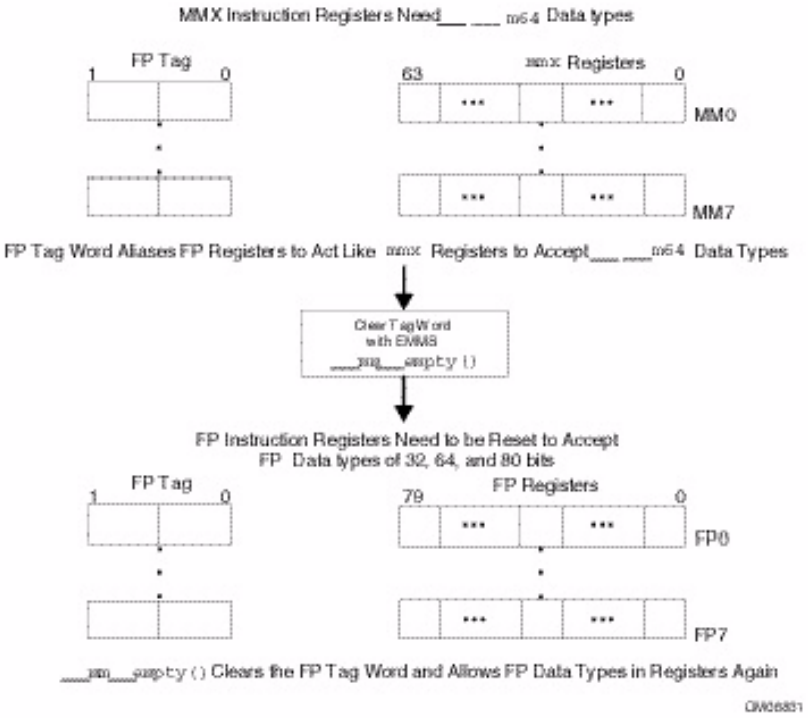
MMX(TM) technology is an extension to the Intel architecture (IA) instruction set. The MMX instruction set adds 57 opcodes and a 64-bit quadword data type, and eight 64-bit registers. Each of the eight registers can be directly addressed using the register names `mm0` to `mm7`.

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

The EMMS Instruction: Why You Need It

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

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



Caution

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

EMMS Usage Guidelines

The guidelines when to use EMMS are:

- Do not use on Itanium(TM)-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™ Technology General Support Intrinsics

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

Intrinsic Name	Alternate Name	Corresponding Instruction	Operation	Signed	Argument Values/Bits	Result Values/Bits
_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 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 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 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 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(TM) Architecture

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

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

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

Data Types

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

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

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

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

Streaming SIMD Extensions

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(TM) 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

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

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

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

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

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

Adds the four SP FP values of a and b.

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

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

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

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

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

Subtracts the four SP FP values of a and b.

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

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

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

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

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

Multiplies the four SP FP values of a and b.

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

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

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

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

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

Divides the four SP FP values of a and b.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Logical Operations for Streaming SIMD Extensions

The prototypes for Streaming SIMD Extensions 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

Intrinsic Name	Comparison	Corresponding Instruction
_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`

Intrinsic Name	Alternate Name	Operation	Corresponding Instruction
_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(TM) Architecture

The Streaming SIMD Extensions intrinsics provide access to Itanium(TM) 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 Functions

Macro Function for Shuffle Using Streaming SIMD Extensions

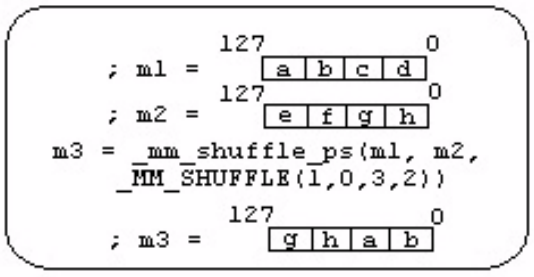
The Streaming SIMD Extensions provide a macro function to help create constants that describe shuffle operations. The macro takes four small integers (in the range of `0` to `3`) and combines them into an 8-bit immediate value used by the `SHUFPS` instruction. See the example below.

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



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(TM)-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

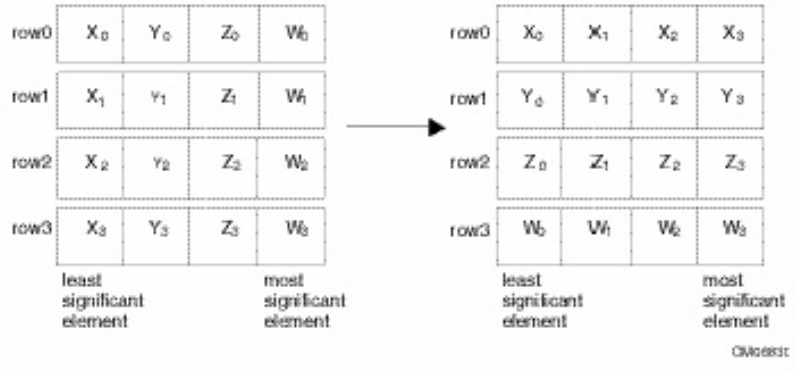
The Streaming SIMD Extensions also provide the following macro function to transpose a 4 by 4 matrix of single precision floating point values.

`_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



Streaming SIMD Extensions 2

Overview of 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(TM)-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 Intrinsics

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

Intrinsic Name	Corresponding Instruction	Compare For:
_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
```

Floating-point Memory and Initialization Operations

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 l`oad` 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
r1 := b0
```

```
int _mm_movemask_pd(__m128d a)
```

(uses MOVMSKPD) Creates a two-bit mask from the sign bits of the two DP FP values of a.

```
r := sign(a1) << 1 | sign(a0)
```

```
__m128d _mm_shuffle_pd(__m128d a, __m128d b, int i)
```

(uses SHUFPD) Selects two specific DP FP values from a and b, based on the mask i. The mask must be an immediate. See Macro Function for Shuffle for a description of the shuffle semantics.

Integer Intrinsics

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

Intrinsic	Instruction	Operation
_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

Intrinsic	Shift Direction	Shift Type	Corresponding Instruction
_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)`

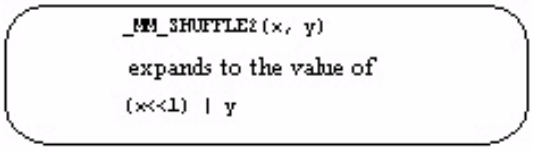
Converts the 4 SP FP values of `a` to signed 32 bit integer values using truncate.

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

Macro Function for Shuffle

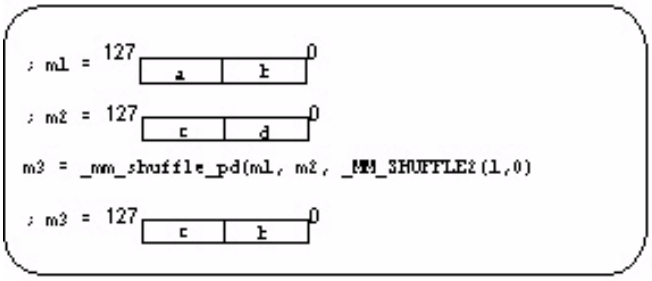
The Streaming SIMD Extensions 2 provide a macro function to help create constants that describe shuffle operations. The macro takes two small integers (in the range of 0 to 1) and combines them into an 2-bit immediate value used by the SHUFPD instruction. See the following example.

Shuffle Function Macro



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

View of Original and Result Words with Shuffle Function Macro



Cacheability Support Operations for Streaming SIMD Extensions 2

The prototypes for Streaming SIMD Extensions 2 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

Intrinsic	Corresponding Instruction	Operation
_m128i_mm_movpi64_epi64	MOVQ2DQ	move
_mm_move_epi64	MOVQ	move

```
__m128i _mm_packs_epi16(__m128i a, __m128i b)
```

Packs the 16 signed 16-bit integers from a and b into 8-bit integers and saturates.

```
r0 := SignedSaturate(a0)
r1 := SignedSaturate(a1)
...
r7 := SignedSaturate(a7)
r8 := SignedSaturate(b0)
r9 := SignedSaturate(b1)
...
r15 := SignedSaturate(b7)
```

```
__m128i _mm_packs_epi32(__m128i a, __m128i b)
```

Packs the 8 signed 32-bit integers from a and b into signed 16-bit integers and saturates.

```
r0 := SignedSaturate(a0)
r1 := SignedSaturate(a1)
r2 := SignedSaturate(a2)
r3 := SignedSaturate(a3)
r4 := SignedSaturate(b0)
r5 := SignedSaturate(b1)
r6 := SignedSaturate(b2)
r7 := SignedSaturate(b3)
```

```
__m128i _mm_packus_epi16(__m128i a, __m128i b)
```

Packs the 16 signed 16-bit integers from a and b into 8-bit unsigned integers and saturates.

```
r0 := UnsignedSaturate(a0)
r1 := UnsignedSaturate(a1)
...
r7 := UnsignedSaturate(a7)
r8 := UnsignedSaturate(b0)
r9 := UnsignedSaturate(b1)
...
r15 := UnsignedSaturate(b7)
```

```
int _mm_extract_epi16(__m128i a, int imm)
```

Extracts the selected signed or unsigned 16-bit integer from a and zero extends. The selector `imm` must be an immediate.

```
r := (imm == 0) ? a0 :
( (imm == 1) ? a1 :
...
(imm == 7) ? a7 )
```

```
__m128i _mm_insert_epi16(__m128i a, int b, int imm)
```

Inserts the least significant 16 bits of b into the selected 16-bit integer of a. The selector imm must be an immediate.

```
r0 := (imm == 0) ? b : a0;
r1 := (imm == 1) ? b : a1;
...
r7 := (imm == 7) ? b : a7;
```

```
int _mm_movemask_epi8(__m128i a)
```

Creates a 16-bit mask from the most significant bits of the 16 signed or unsigned 8-bit integers in a and zero extends the upper bits.

```
r := a15[7] << 15 |
a14[7] << 14 |
...
a1[7] << 1 |
a0[7]
```

```
__m128i _mm_shuffle_epi32(__m128i a, int imm)
```

Shuffles the 4 signed or unsigned 32-bit integers in a as specified by imm. The shuffle value, imm, must be an immediate. See Macro Function for Shuffle for a description of shuffle semantics.

```
__m128i _mm_shufflehi_epi16(__m128i a, int imm)
```

Shuffles the upper 4 signed or unsigned 16-bit integers in a as specified by imm. The shuffle value, imm, must be an immediate. See Macro Function for Shuffle for a description of shuffle semantics.

```
__m128i _mm_shufflelo_epi16(__m128i a, int imm)
```

Shuffles the lower 4 signed or unsigned 16-bit integers in a as specified by imm. The shuffle value, imm, must be an immediate. See Macro Function for Shuffle for a description of shuffle semantics.

```
__m128i _mm_unpackhi_epi8(__m128i a, __m128i b)
```

Interleaves the upper 8 signed or unsigned 8-bit integers in a with the upper 8 signed or unsigned 8-bit integers in b.

```
r0 := a8 ; r1 := b8
r2 := a9 ; r3 := b9
...
r14 := a15 ; r15 := b15
```

```
__m128i _mm_unpackhi_epi16(__m128i a, __m128i b)
```

Interleaves the upper 4 signed or unsigned 16-bit integers in a with the upper 4 signed or unsigned 16-bit integers in b.

```
r0 := a4 ; r1 := b4
r2 := a5 ; r3 := b5
r4 := a6 ; r5 := b6
r6 := a7 ; r7 := b7
```

```
__m128i _mm_unpackhi_epi32(__m128i a, __m128i b)
```

Interleaves the upper 2 signed or unsigned 32-bit integers in a with the upper 2 signed or unsigned 32-bit integers in b.

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

```
__m128i _mm_unpackhi_epi64(__m128i a, __m128i b)
```

Interleaves the upper signed or unsigned 64-bit integer in a with the upper signed or unsigned 64-bit integer in b.

```
r0 := a1 ; r1 := b1
```

```
__m128i _mm_unpacklo_epi8(__m128i a, __m128i b)
```

Interleaves the lower 8 signed or unsigned 8-bit integers in a with the lower 8 signed or unsigned 8-bit integers in b.

```
r0 := a0 ; r1 := b0
r2 := a1 ; r3 := b1
...
r14 := a7 ; r15 := b7
```

```
__m128i _mm_unpacklo_epi16(__m128i a, __m128i b)
```

Interleaves the lower 4 signed or unsigned 16-bit integers in a with the lower 4 signed or unsigned 16-bit integers in b.

```
r0 := a0 ; r1 := b0
r2 := a1 ; r3 := b1
r4 := a2 ; r5 := b2
r6 := a3 ; r7 := b3
```

```
__m128i _mm_unpacklo_epi32(__m128i a, __m128i b)
```

Interleaves the lower 2 signed or unsigned 32-bit integers in a with the lower 2 signed or unsigned 32-bit integers in b.

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

```
__m128i _mm_unpacklo_epi64(__m128i a, __m128i b)
```

Interleaves the lower signed or unsigned 64-bit integer in a with the lower signed or unsigned 64-bit integer in b.

```
r0 := a0 ; r1 := b0
```

```
__m64 _mm_movepi64_pi64(__m128i a)
```

Returns the lower 64 bits of a as an __m64 type.

```
r0 := a0 ;
```

```
__128i _mm_movpi64_pi64(__m64 a)
```

Moves the 64 bits of a to the lower 64 bits of the result, zeroing the upper bits.

```
r0 := a0 ; r1 := 0X0 ;
```

```
__128i _mm_move_epi64(__128i a)
```

Moves the lower 64 bits of the lower 64 bits of the result, zeroing the upper bits.

```
r0 := a0 ; r1 := 0X0 ;
```

Integer Memory and Initialization Operations

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
```

Intrinsics for Itanium(TM) Instructions

Overview of Intrinsics for Itanium(TM) Instructions

This section lists and describes the native intrinsics for Itanium(TM) 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 Itanium intrinsics are in the `ia64intrin.h` header file.

Native Intrinsics for Itanium(TM) Instructions

The prototypes for Itanium 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. Also, it is necessary to link a library to satisfy the function call generated by the compiler.

`__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 Itanium 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.

Intrinsic	Description
`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.

Intrinsic	Description
`__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.

Operating System Related Intrinsics

The prototypes for Itanium 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

Intrinsic	Description
`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.

Intrinsic	Description
`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.

Intrinsic	Description
`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.

Itanium(TM) Conversion Intrinsics

The prototypes for Itanium 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

Name	whichReg
_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

Name	whichReg
_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

Itanium(TM) Multimedia Additions

The prototypes for Itanium 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)

Intrinsic	Corresponding Instruction
`__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)`

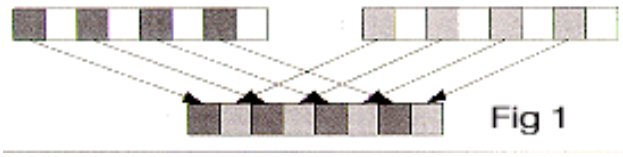
The 64-bit value a is scanned for a zero element from the most significant element to the least significant element, and the index of the first zero element is returned. The element width is 16 bits, so the range of the result is from 0 - 3. If no zero element is found, the default result is 4.

`__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.

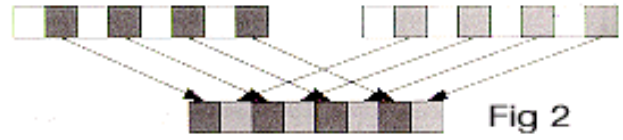
`__m64 _m64_mix1l(__m64 a, __m64 b)`

Interleave 64-bit quantities a and b in 1-byte groups, starting from the left, as shown in Figure 1, and return the result.



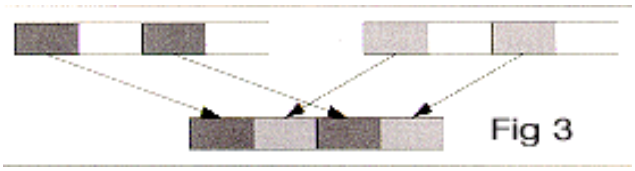
`__m64 _m64_mix1r(__m64 a, __m64 b)`

Interleave 64-bit quantities a and b in 1-byte groups, starting from the right, as shown in Figure 2, and return the result.



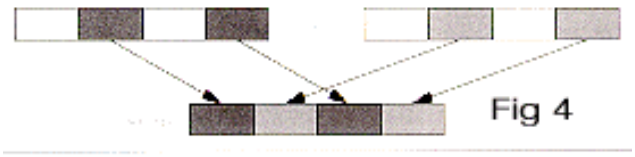
`__m64 _m64_mix2l(__m64 a, __m64 b)`

Interleave 64-bit quantities a and b in 2-byte groups, starting from the left, as shown in Figure 3, and return the result.



`__m64 _m64_mix2r(__m64 a, __m64 b)`

Interleave 64-bit quantities a and b in 2-byte groups, starting from the right, as shown in Figure 4, and return the result.



`__m64 _m64_mix4l(__m64 a, __m64 b)`

Interleave 64-bit quantities `a` and `b` in 4-byte groups, starting from the left, as shown in Figure 5, and return the result.

`__m64 _m64_mix4r(__m64 a, __m64 b)`

Interleave 64-bit quantities `a` and `b` in 4-byte groups, starting from the right, as shown in Figure 6, and return the result.

```
__m64 _m64_mux1(__m64 a, const int n)
```

Based on the value of n, a permutation is performed on a as shown in Figure 7, and the result is returned. Table 1 shows the possible values of n.

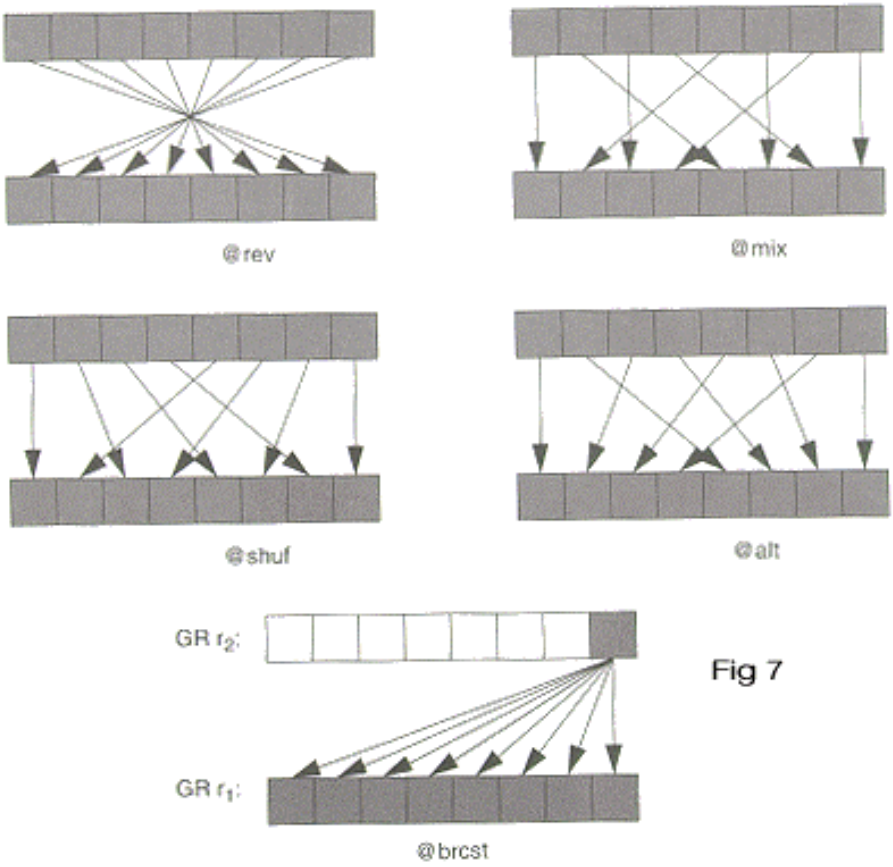


Fig 7

Table 1. Values of n for m64_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.

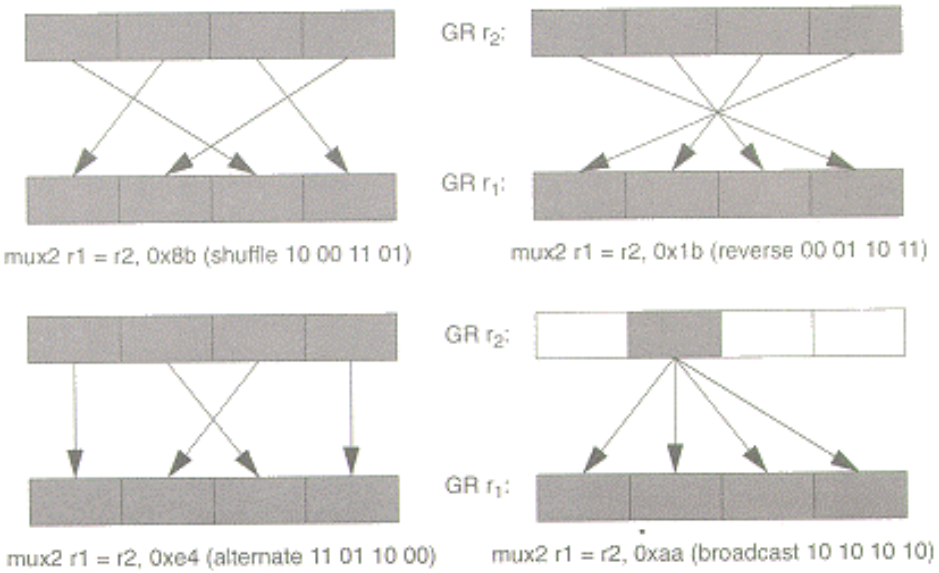


Fig 8

`__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.

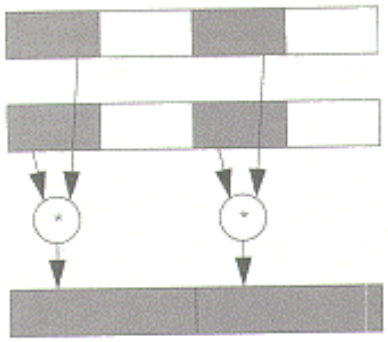


Fig 9

`__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.

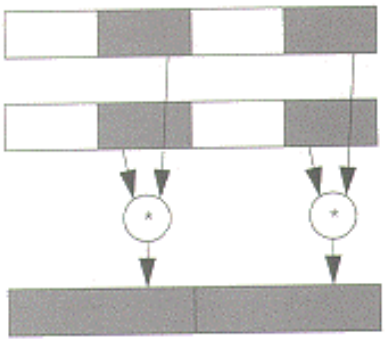


Fig 10

`__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.

Data Alignment, Memory Allocation Intrinsics, and Inline Assembly

Overview of Data Alignment, Memory Allocation Intrinsics, and Inline Assembly

This section describes features that support usage of the intrinsics. The following topics are described:

- Alignment Support
- Allocating and Freeing Aligned Memory Blocks
- 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(TM)-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. 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.cpp without expanding the library functions, but with inline expansion from interprocedural optimizations (IPO):

- **IA-32 Systems:** `prompt>icc -ip -nolib_inline sum.cpp`
- **Itanium(TM)-based Systems:** `prompt>ecc -ip -nolib_inline sum.cpp`

For details on IPO, see Interprocedural Optimizations.

MASM* Style Inline Assembly

The Intel® C++ Compiler supports MASM style inline assembly with the `-use_msasm` option. See your MASM documentation for the proper syntax.

GNU*-like Style Inline Assembly

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.

Syntax Element	Description
asm-interface	The asm-interface consists of three parts: 1. an optional output-list 2. an optional input-list 3. an optional clobber-list These are separated by colon (:) characters. If the output-list is missing, but an input-list is given, the input list may be preceded by two colons (::)to take the place of the missing output-list. If the asm-interface is omitted altogether, the asm statement is considered volatile regardless of whether a volatile-keyword was specified.
output-list	An output-list consists of one or more output-specs separated by commas. For the purposes of substitution in the asm-template, each output-spec is numbered. The first operand in the output-list is numbered 0, the second is 1, and so on. Numbering is continuous through the output-list and into the input-list. The total number of operands is limited to 10 (i.e. 0-9).
input-list	Similar to an output-list, an input-list consists of one or more input-specs separated by commas. For the purposes of substitution in the asm-template, each input-spec is numbered, with the numbers continuing from those in the output-list.
clobber-list	A clobber-list tells the compiler that the asm uses or changes a specific machine register that is either coded directly into the asm or is changed implicitly by the assembly instruction. The clobber-list is a comma-separated list of clobber-specs.
input-spec	The input-specs tell the compiler about expressions whose values may be needed by the inserted assembly instruction. In order to describe fully the input requirements of the asm, you can list input-specs that are not actually referenced in the asm-template.
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

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(TM) 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(TM) 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`	A	A	A	A	A

Intrinsic	Across All IA	MMX(TM) Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM) Architecture
`__rotr(unsigned int value, int shift)`					
`__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

Intrinsic	Across All IA	MMX(TM) Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM) Architecture
`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

Intrinsic	Across All IA	MMX(TM) Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM) Architecture
`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

Intrinsic	Across All IA	MMX(TM) Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM) Architecture
`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

Intrinsic	Across All IA	MMX(TM) Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM) Architecture
`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

Intrinsic	Across All IA	MMX(TM) Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM) Architecture
`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	Itanium(TM Architecture
_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_packssw b	_mm_packs_pi16	N/A	A	A	A	A
_m_packssd w	_mm_packs_pi32	N/A	A	A	A	A
_m_packusw b	_mm_packs_pu16	N/A	A	A	A	A
_m_punpckh bw	_mm_unpackhi_pi8	N/A	A	A	A	A
_m_punpckh wd	_mm_unpackhi_pi16	N/A	A	A	A	A
_m_punpckh dq	_mm_unpackhi_pi32	N/A	A	A	A	A
_m_punpckl bw	_mm_unpacklo_pi8	N/A	A	A	A	A
_m_punpckl wd	_mm_unpacklo_pi16	N/A	A	A	A	A
_m_punpckl dq	_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

Intrinsic Name	Alternate Name	Across All IA	MMX(TM Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic Name	Alternate Name	Across All IA	MMX(TM Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic Name	Alternate Name	Across All IA	MMX(TM Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM Architecture
_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(TM)-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	Itanium(TM Architecture
_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

Intrinsic Name	Alternate Name	Across All IA	MMX(TM Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic Name	Alternate Name	Across All IA	MMX(TM Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM Architecture
_ss						
_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

Intrinsic Name	Alternate Name	Across All IA	MMX(TM Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic Name	Alternate Name	Across All IA	MMX(TM Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic Name	Alternate Name	Across All IA	MMX(TM Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic Name	Alternate Name	Across All IA	MMX(TM Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Itanium(TM Architecture
_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(TM) processor does not support parallel double precision computation, so Streaming SIMD Extensions 2 are not implemented on Itanium-based systems.

Key to the table entries:

- A = Expected to give significant performance gain over non-intrinsic-based code equivalent.
- B = Non-intrinsic-based source code would be better; the intrinsic's implementation may map directly to native instructions, but they offer no significant performance gain.
- C = Requires contorted implementation for particular microarchitecture. Will result in very poor performance if used.

Intrinsic	Across All IA	MMX(TM Technology	Streaming SIMD Extenions	Pentium® 4 Processor Streaming SIMD Extensions 2	Itanium(TM 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

Intrinsic	Across All IA	MMX(TM Technology	Streaming SIMD Extenions	Pentium® 4 Processor Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic	Across All IA	MMX(TM Technology	Streaming SIMD Extenions	Pentium® 4 Processor Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic	Across All IA	MMX(TM Technology	Streaming SIMD Extenions	Pentium® 4 Processor Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic	Across All IA	MMX(TM Technology	Streaming SIMD Extenions	Pentium® 4 Processor Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic	Across All IA	MMX(TM Technology	Streaming SIMD Extenions	Pentium® 4 Processor Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic	Across All IA	MMX(TM Technology	Streaming SIMD Extenions	Pentium® 4 Processor Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic	Across All IA	MMX(TM Technology	Streaming SIMD Extenions	Pentium® 4 Processor Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic	Across All IA	MMX(TM Technology	Streaming SIMD Extenions	Pentium® 4 Processor Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic	Across All IA	MMX(TM Technology	Streaming SIMD Extenions	Pentium® 4 Processor Streaming SIMD Extensions 2	Itanium(TM Architecture
_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

Intrinsic	Across All IA	MMX(TM Technology	Streaming SIMD Extenions	Pentium® 4 Processor Streaming SIMD Extensions 2	Itanium(TM Architecture
_mm_maskmoveu_si128	N/A	N/A	N/A	A	N/A
_mm_stream_pd	N/A	N/A	N/A	A	N/A
_mm_stream_si128	N/A	N/A	N/A	A	N/A
_mm_clflush	N/A	N/A	N/A	A	N/A
_mm_lfence	N/A	N/A	N/A	A	N/A
_mm_mfence	N/A	N/A	N/A	A	N/A
_mm_stream_si32	N/A	N/A	N/A	A	N/A
_mm_pause	N/A	N/A	N/A	A	N/A

Intel® C++ Class Libraries

Introduction to the Class Libraries

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, and Itanium(TM) processors
`fvec.h`	Streaming SIMD Extensions	Pentium III, Pentium 4 and Itanium processors
`dve.ch`	Streaming SIMD Extensions 2	Pentium 4 processor only

About the Classes

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

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

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

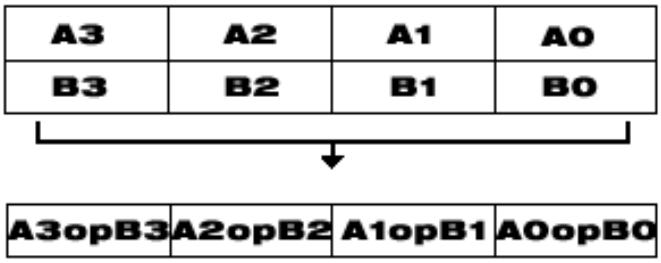
This documentation is intended for programmers writing code for the Intel Architecture, particularly code that would benefit from the use of SIMD instructions. You should be familiar with C++ and the use of C++ classes.

Technical Overview

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 usage 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(TM)-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

Instruction Set	Class	Signedness	Data Type	Size	Elements	Header File
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 Fvec and Ivec classes.

Capabilities

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

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

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

Computation

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

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

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

Horizontal Data Support

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

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

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

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

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

Branch Compression/Elimination

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

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

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

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

Caching Hints

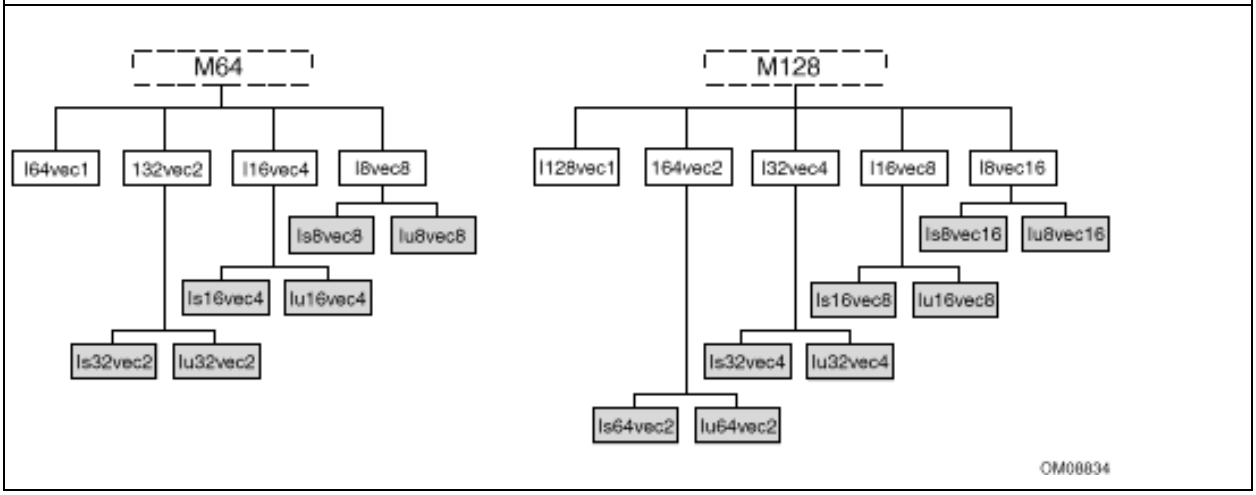
Streaming SIMD Extensions provide prefetching and streaming hints. Prefetching data can minimize the effects of memory latency. Streaming hints allow you to indicate that certain data should not be cached. This results in higher performance for data that should be cached.

Integer Vector Classes

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);

Operation	Class	Syntax
		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		
	Standard	w/ assign	Standard	w/assign	Corresponding Intrinsic
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

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

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

Addition and Subtraction with Assignment

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

Multiplication Operators

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

Syntax Usage for Multiplication Operators

```
/* Explicitly convert B to Is16vec4 */

Is16vec4 A,C;

Iu32vec2 B;

C = A * C;

C = A * (Is16vec4)B;

/* Return nearest common ancestor type, I16vec4 */

Is16vec4 A;

Iu16vec4 B;

I16vec4 C;

C = A + B;

/* The mul_high and mul_add functions take Is16vec4 data only */

Is16vec4 A,B,C,D;

C = mul_high(A,B);

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

Multiplication Operators with Corresponding Intrinsics

Operation	Symbols		Syntax Usage		Intrinsic
Multiplication	*	*=	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`	

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

```
cout << Is32vec4 A;

cout << Iu32vec4 A;

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

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

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

Corresponding Intrinsics: none

```
cout << Is32vec2 A;

cout << Iu32vec2 A;

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

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

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

Corresponding Intrinsics: none

cout << Is16vec8 A;

cout << Iu16vec8 A;

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

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

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

Corresponding Intrinsics: none

cout << Is16vec4 A;

cout << Iu16vec4 A;

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

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

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

Corresponding Intrinsics: none

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

/* print in hex format instead of decimal*/

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

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

/* print in hex format instead of decimal*/

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

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

```
Is64vec2 unpack_high(Is64vec2 A, Is64vec2 B)
```

```
Iu64vec2 unpack_high(Iu64vec2 A, Iu64vec2 B)
```

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

```
R0 = A1;
```

```
R1 = B1;
```

Corresponding intrinsic: `_mm_unpackhi_epi64`

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

```
Is32vec4 unpack_high(Is32vec4 A, Is32vec4 B)
```

```
Iu32vec4 unpack_high(Iu32vec4 A, Iu32vec4 B)
```

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

```
R0 = A1;
```

```
R1 = B1;
```

```
R2 = A2;
```

```
R3 = B2;
```

Corresponding intrinsic: `_mm_unpackhi_epi32`

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

```
Is32vec2 unpack_high(Is32vec2 A, Is32vec2 B)
```

```
Iu32vec2 unpack_high(Iu32vec2 A, Iu32vec2 B)
```

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

```
R0 = A1;
R1 = B1;
```

Corresponding intrinsic: `_mm_unpackhi_pi32`

```
I16vec8 unpack_high(I16vec8 A, I16vec8 B)
Is16vec8 unpack_high(Is16vec8 A, Is16vec8 B)
Iu16vec8 unpack_high(Iu16vec8 A, Iu16vec8 B)
```

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

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

Corresponding intrinsic: `_mm_unpackhi_epi16`

```
I16vec4 unpack_high(I16vec4 A, I16vec4 B)
Is16vec4 unpack_high(Is16vec4 A, Is16vec4 B)
Iu16vec4 unpack_high(Iu16vec4 A, Iu16vec4 B)
```

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

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

Corresponding intrinsic: `_mm_unpackhi_pi16`

```
I8vec8 unpack_high(I8vec8 A, I8vec8 B)
Is8vec8 unpack_high(Is8vec8 A, I8vec8 B)
Iu8vec8 unpack_high(Iu8vec8 A, I8vec8 B)
```

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

```
R0 = A4;
R1 = B4;
R2 = A5;
R3 = B5;
R4 = A6;
R5 = B6;
```

```
R6 = A7;
R7 = B7;
```

Corresponding intrinsic: `_mm_unpackhi_pi8`

```
I8vec16 unpack_high(I8vec16 A, I8vec16 B)
Is8vec16 unpack_high(Is8vec16 A, I8vec16 B)
Iu8vec16 unpack_high(Iu8vec16 A, I8vec16 B)
```

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

```
I64vec2 unpack_low(I64vec2 A, I64vec2 B);
Is64vec2 unpack_low(Is64vec2 A, Is64vec2 B);
```

```
Iu64vec2 unpack_low(Iu64vec2 A, Iu64vec2 B);
```

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

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

Corresponding intrinsic: `_mm_unpacklo_epi32`

```
I32vec4 unpack_low(I32vec4 A, I32vec4 B);
Is32vec4 unpack_low(Is32vec4 A, Is32vec4 B);
Iu32vec4 unpack_low(Iu32vec4 A, Iu32vec4 B);
```

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

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

Corresponding intrinsic: `_mm_unpacklo_epi32`

```
I32vec2 unpack_low(I32vec2 A, I32vec2 B);
Is32vec2 unpack_low(Is32vec2 A, Is32vec2 B);
Iu32vec2 unpack_low(Iu32vec2 A, Iu32vec2 B);
```

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_pi32`

```
I16vec8 unpack_low(I16vec8 A, I16vec8 B);
Is16vec8 unpack_low(Is16vec8 A, Is16vec8 B);
Iu16vec8 unpack_low(Iu16vec8 A, Iu16vec8 B);
```

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

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

```
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
```

Corresponding intrinsic: `_mm_unpacklo_epi16`

```
I16vec4 unpack_low(I16vec4 A, I16vec4 B);
Is16vec4 unpack_low(Is16vec4 A, Is16vec4 B);
Iu16vec4 unpack_low(Iu16vec4 A, Iu16vec4 B);
```

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

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

Corresponding intrinsic: `_mm_unpacklo_pi16`

```
I8vec16 unpack_low(I8vec16 A, I8vec16 B);
Is8vec16 unpack_low(Is8vec16 A, Is8vec16 B);
Iu8vec16 unpack_low(Iu8vec16 A, Iu8vec16 B);
```

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

```
I8vec8 unpack_low(I8vec8 A, I8vec8 B);
Is8vec8 unpack_low(Is8vec8 A, Is8vec8 B);
Iu8vec8 unpack_low(Iu8vec8 A, Iu8vec8 B);
```

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

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

Corresponding intrinsic: `_mm_unpacklo_pi8`

Pack Operators

`Is16vec8 pack_sat(Is32vec2 A,Is32vec2 B);`

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

Corresponding intrinsic: `_mm_packs_epi32`

`Is16vec4 pack_sat(Is32vec2 A,Is32vec2 B);`

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

Corresponding intrinsic: `_mm_packs_pi32`

`Is8vec16 pack_sat(Is16vec4 A,Is16vec4 B);`

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

Corresponding intrinsic: `_mm_packs_epi16`

`Is8vec8 pack_sat(Is16vec4 A,Is16vec4 B);`

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

Corresponding intrinsic: `_mm_packs_pi16`

`Iu8vec16 packu_sat(Is16vec4 A,Is16vec4 B);`

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

Corresponding intrinsic: `_mm_packus_epi16`

`Iu8vec8 packu_sat(Is16vec4 A,Is16vec4 B);`

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

Corresponding intrinsic: `_mm_packs_pu16`

Clear MMX(TM) Instructions State Operator

`void empty(void);`

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

Corresponding intrinsic: `_mm_empty`

Integer Intrinsics for Streaming SIMD Extensions

Note

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

Is16vec4 simd_max(Is16vec4 A, Is16vec4 B);

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

Corresponding intrinsic: `_mm_max_pi16`

Is16vec4 simd_min(Is16vec4 A, Is16vec4 B);

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

Corresponding intrinsic: `_mm_min_pi16`

Iu8vec8 simd_max(Iu8vec8 A, Iu8vec8 B);

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

Corresponding intrinsic: `_mm_max_pu8`

Iu8vec8 simd_min(Iu8vec8 A, Iu8vec8 B);

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

Corresponding intrinsic: `_mm_min_pu8`

int move_mask(I8vec8 A);

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

Corresponding intrinsic: `_mm_movemask_pi8`

void mask_move(I8vec8 A, I8vec8 B, signed char *p);

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.

Corresponding intrinsic: `_mm_maskmove_si64`

void store_nta(__m64 *p, M64 A);

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

Corresponding intrinsic: `_mm_stream_pi`

Iu8vec8 simd_avg(Iu8vec8 A, Iu8vec8 B);

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

Corresponding intrinsic: `_mm_avg_pu8`

Iu16vec4 simd_avg(Iu16vec4 A, Iu16vec4 B);

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

Corresponding intrinsic: `_mm_avg_pu16`

Conversions Between Fvec and Ivec

int F64vec2ToInt(F64vec42 A)

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

r := (int)A0

F64vec2 F32vec4ToF64vec2(F32vec4 A)

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

r0 := (double)A0;

r1 := (double)A1;

F32vec4 F64vec2ToF32vec4(F64vec2 A)

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

r0 := (float)A0;

r1 := (float)A1;

F64vec2 InttoF64vec2(F64vec2 A, int B)

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.

r0 := (double)B;

r1 := A1;

int F32vec4ToInt(F32vec4 A)

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

r := (int)A0

Is32vec2 F32vec4ToIs32vec2 (F32vec4 A)

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

r0 := (int)A0

r1 := (int)A1

F32vec4 IntToF32vec4(F32vec4 A, int B)

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

r0 := (float)B

r1 := A1;

r2 := A2 ;

r3 := A3

F32vec4 Is32vec2ToF32vec4(F32vec4 A, Is32vec2 B)

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.

r0 := (float)B0

r1 := (float)B1

r2 := A2

r3 := A3

Floating-point Vector Classes

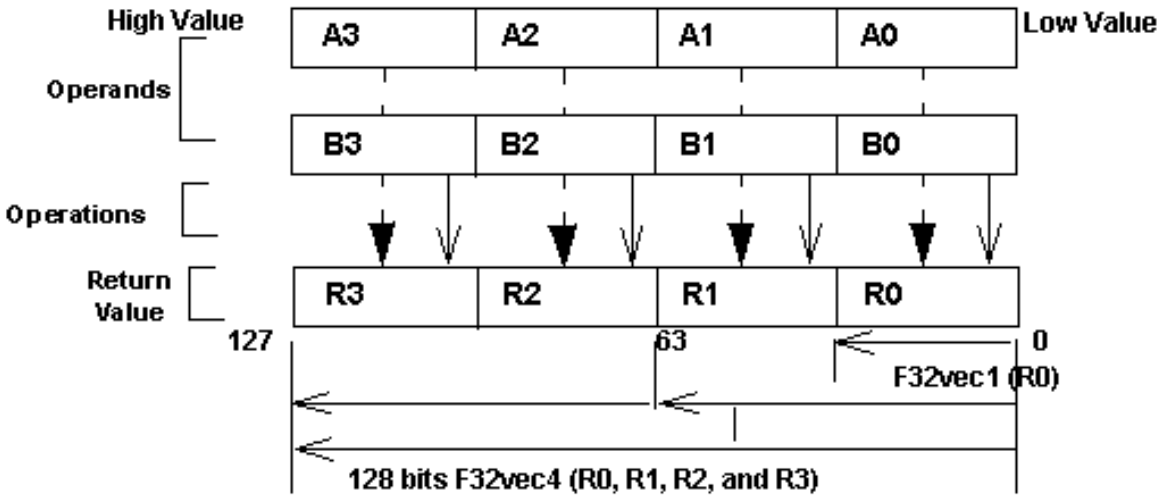
Floating-point Vector Classes

The floating-point vector classes (Fvec), 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		

Example	Intrinsic	Returns
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

Operation	Returns	Example Syntax Usage	Intrinsic
	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`

Returns	Example Syntax Usage	Intrinsic

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

F64vec2 R = simd_min(F64vec2 A, F64vec2 B)

Compute the minimums of the two double precision floating-point values of A and B.

R0 := min(A0,B0);

R1 := min(A1,B1);

Corresponding intrinsic: `_mm_min_pd`

F32vec4 R = simd_min(F32vec4 A, F32vec4 B)

Compute the minimums of the four single precision floating-point values of A and B.

R0 := min(A0,B0);

R1 := min(A1,B1);

R2 := min(A2,B2);

R3 := min(A3,B3);

Corresponding intrinsic: `_mm_min_ps`

F32vec1 R = simd_min(F32vec1 A, F32vec1 B)

Compute the minimum of the lowest single precision floating-point values of A and B.

R0 := min(A0,B0);

Corresponding intrinsic: `_mm_min_ss`

F64vec2 simd_max(F64vec2 A, F64vec2 B)

Compute the maximums of the two double precision floating-point values of A and B.

```
R0 := max(A0,B0);

R1 := max(A1,B1);
```

Corresponding intrinsic: _mm_max_pd

F32vec4 R = simd_man(F32vec4 A, F32vec4 B)

Compute the maximums of the four single precision floating-point values of A and B.

R0 := max(A0,B0);

R1 := max(A1,B1);

R2 := max(A2,B2);

R3 := max(A3,B3);

Corresponding intrinsic: _mm_max_ps

F32vec1 simd_max(F32vec1 A, F32vec1 B)

Compute the maximum of the lowest single precision floating-point values of A and B.

R0 := max(A0,B0);

Corresponding intrinsic: _mm_max_ss

Logical Operators

The "Fvec Logical Operators Return Value Mapping" table lists the logical operators of the Fvec classes and generic syntax. The logical operators for F32vec1 classes use only the lower 32 bits.

Fvec Logical Operators Return Value Mapping

Bitwise Operation	Operators	Generic Syntax
AND	& &=	R = A & B; R &= A;
OR	| |=	R = A | B; R |= A;
XOR	^ ^=	R = A ^ B; R ^= A;
andnot	andnot	R = andnot(A);

The following table lists standard logical operators syntax and corresponding intrinsics. Note that there is no corresponding scalar intrinsic for the `F32vec1` classes, which accesses the lower 32 bits of the packed vector intrinsics.

Logical Operations for Fvec Classes

Operation	Returns	Example Syntax Usage	Intrinsic
AND	4 floats	`F32vec4 & = F32vec4 A & F32vec4 B;` `F32vec4 & &= F32vec4 A;`	`_mm_and_ps`
	2 doubles	`F64vec2 R = F64vec2 A & F32vec2 B;` `F64vec2 R &= F64vec2 A;`	`_mm_and_pd`
	1 float	`F32vec1 R = F32vec1 A & F32vec1 B;` `F32vec1 R &= F32vec1 A;`	`_mm_and_ps`
OR	4 floats	`F32vec4 R = F32vec4 A | F32vec4 B;` `F32vec4 R |= F32vec4 A;`	`_mm_or_ps`
	2 doubles	`F64vec2 R = F64vec2 A | F32vec2 B;` `F64vec2 R |= F64vec2 A;`	`_mm_or_pd`
	1 float	`F32vec1 R = F32vec1 A | F32vec1 B;` `F32vec1 R |= F32vec1 A;`	`_mm_or_ps`
XOR	4 floats	`F32vec4 R = F32vec4 A ^ F32vec4 B;` `F32vec4 R ^= F32vec4 A;`	`_mm_xor_ps`
	2 doubles	`F64vec2 R = F64vec2 A ^ F364vec2 B;` `F64vec2 R ^= F64vec2 A;`	`_mm_xor_pd`
	1 float	`F32vec1 R = F32vec1 A ^ F32vec1 B;` `F32vec1 R ^= F32vec1 A;`	`_mm_xor_ps`
ANDNOT	2 doubles	`F64vec2 R = andnot(F64vec2 A,` `F64vec2 B);`	`_mm_andnot_pd`

Compare Operators

The operators described in this section compare the single precision floating-point values of A and B. Comparison between objects of any Fvec class return the same class being compared.

The following table lists the compare operators for the Fvec classes.

Compare Operators and Corresponding Intrinsics

Compare For:	Operators	Syntax
Equality	cmpeq	R = cmpeq(A, B)
Inequality	cmpneq	R = cmpneq(A, B)
Greater Than	cmpgt	R = cmpgt(A, B)
Greater Than or Equal To	cmpge	R = cmpge(A, B)
Not Greater Than	cmpngt	R = cmpngt(A, B)
Not Greater Than or Equal To	cmpnge	R = cmpnge(A, B)
Less Than	cmplt	R = cmplt(A, B)
Less Than or Equal To	cmple	R = cmple(A, B)
Not Less Than	cmpnlt	R = cmpnlt(A, B)
Not Less Than or Equal To	cmpnle	R = cmpnle(A, B)

Compare Operators

The mask is set to `0xffffffff` for each floating-point value where the comparison is true and `0x00000000` where the comparison is false. The 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	0x00000000	X	X	X

R	A0	For Any Operators	B	If True	If False	F32vec4	F64vec2	F32vec1
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 Compare Operations for Fvec Classes table shows examples for arithmetic operators and intrinsics.

Compare Operations for Fvec Classes

Returns	Example Syntax Usage	Intrinsic
Compare for Equality		
4 floats	F32vec4 R = cmpeq(F32vec4 A);	_mm_cmpeq_ps
2 doubles	F64vec2 R = cmpeq(F64vec2 A);	_mm_cmpeq_pd
1 float	F32vec1 R = cmpeq(F32vec1 A);	_mm_cmpeq_ss
Compare for Inequality		
4 floats	F32vec4 R = cmpneq(F32vec4 A);	_mm_cmpneq_ps
2 doubles	F64vec2 R = cmpneq(F64vec2 A);	_mm_cmpneq_pd
1 float	F32vec1 R = cmpneq(F32vec1 A);	_mm_cmpneq_ss
Compare for Less Than		
4 floats	F32vec4 R =	_mm_cmplt_ps

	cmplt(F32vec4 A);	
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

R	A0	Operators	B	C	D	F32vec4	F64vec2	F32vec1
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

`void store_nta(double *p, F64vec2 A);`

Stores (non-temporal) the two double-precision floating-point values of A. Requires a 16-byte aligned address.

Corresponding intrinsic: `_mm_stream_pd`

`void store_nta(float *p, F32vec4 A);`

Stores (non-temporal) the four single precision floating-point values of A. Requires a 16-byte aligned address.

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

cout << F64vec2 A;

The two single double precision floating-point values of A are placed in the output buffer and printed in decimal format as follows:

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

Corresponding intrinsics: none

cout << F32vec4 A;

The four single precision floating-point values of A are placed in the output buffer and printed in decimal format as follows:

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

Corresponding intrinsics: none

cout << F32vec1 A;

The lowest single precision floating-point value of A is placed in the output buffer and printed.

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:

double d = F64vec2 A[1];

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.

Corresponding intrinsics: none

float f = F32vec4 A[int i]

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[2];

If DEBUG is enabled and i is not one of the permitted values (0-3), a diagnostic message is printed and the program aborts.

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:

F32vec4 A[3] = float f;

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.

Corresponding intrinsics: none.

Load and Store Operators

```
void loadu(F64vec2 A, double *p)
```

Loads two double-precision floating-point values, copying them into the two floating-point values of A. No assumption is made for alignment.

Corresponding intrinsic: `_mm_loadu_pd`

```
void storeu(float *p, F64vec2 A);
```

Stores the two double-precision floating-point values of A. No assumption is made for alignment.

Corresponding intrinsic: `_mm_storeu_pd`

```
void loadu(F32vec4 A, double *p)
```

Loads four single-precision floating-point values, copying them into the four floating-point values of A. No assumption is made for alignment.

Corresponding intrinsic: `_mm_loadu_ps`

```
void storeu(float *p, F32vec4 A);
```

Stores the four single-precision floating-point values of A. No assumption is made for alignment.

Corresponding intrinsic: `_mm_storeu_ps`

Unpack Operators for Fvec Operators

```
F64vec2 R = unpack_low(F64vec2 A, F64vec2 B);
```

Selects and interleaves the lower double precision floating-point values from A and B.

Corresponding intrinsic: `_mm_unpacklo_pd(a, b)`

```
F64vec2 R = unpack_high(F64vec2 A, F64vec2 B);
```

Selects and interleaves the higher double precision floating-point values from A and B.

Corresponding intrinsic: `_mm_unpackhi_pd(a, b)`

```
F32vec4 R = unpack_low(F32vec4 A, F32vec4 B);
```

Selects and interleaves the lower two single precision floating-point values from A and B.

Corresponding intrinsic: `_mm_unpacklo_ps(a, b)`

```
F32vec4 R = unpack_high(F32vec4 A, F32vec4 B);
```

Selects and interleaves the higher two single precision floating-point values from A and B.

Corresponding intrinsic: `_mm_unpackhi_ps(a, b)`

Move Mask Operator

```
int i = move_mask(F64vec2 A)
```

Creates a 2-bit mask from the most significant bits of the two double precision floating-point values of A, as follows:

```
i := sign(a1)<<1 | sign(a0)<<0
```

Corresponding intrinsic: `_mm_movemask_pd`

```
int i = move_mask(F32vec4 A)
```

Creates a 4-bit mask from the most significant bits of the four single precision floating-point values of A, as follows:

```
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 Intrinsics	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	F64vec2	F32vec4	F32vec1
+, +=	_mm_add_[x]	epi64	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	ss
-, -=	_mm_sub_[x]	epi64	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	ss
*, *=	_mm_mullo_[x]	N/A	N/A	epi16	N/A	N/A	pi16	N/A	pd	ps	ss
/, /=	_mm_div_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
mul_high	_mm_mulhi_[x]	N/A	N/A	epi16	N/A	N/A	pi16	N/A	N/A	N/A	N/A
mul_add	_mm_madd_[x]	N/A	N/A	epi16	N/A	N/A	pi16	N/A	N/A	N/A	N/A
sqrt	_mm_sqrt_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
rcp	_mm_rcp_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
rcp_nr	_mm_rcp_[x] _mm_add_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss

Operators	Corresponding Intrinsic	I64vec2	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
rsqrt	_mm_rsqrt_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	*pd*	*ps*	*ss*
rsqrt_nr	_mm_rsqrt_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	*pd*	*ps*	*ss*

Shift Operators: Corresponding Intrinsics and Classes

Operators	Corresponding Intrinsic	I128vec1	I64vec2	I32vec4	I16vec8	I8vec16	I64vec1	I32vec2	I16vec4	I8vec8
>>,>>=	_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	pi16 pi16 pi16 pi16	N/A N/A N/A N/A
<<, <<=	_mm_sll_[x] _mm_slli_[x]	N/A N/A	epi64 epi64	epi32 epi32	*epi16* epi16	N/A N/A	si64 si64	pi32 pi32	pi16 pi16	N/A N/A

Comparison Operators: Corresponding Intrinsics and Classes

Operators	Corresponding Intrinsic	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
cmpeq	_mm_cmpeq_[x]	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	ss
cmpneq	_mm_cmpeq_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64	pd	ps	ss
cmpgt	_mm_cmpgt_[x]	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	ss
cmpge	_mm_cmpge_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64	pd	ps	ss
cmplt	_mm_cmplt_[x]	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	ss
cmple	_mm_cmple_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64	pd	ps	ss
cmpngt	_mm_cmpngt_[x]	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	ss

Operators	Corresponding Intrinsic	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
cmpnge	_mm_cmpnge_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
cmnpnlt	_mm_cmpnlt_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
cmpnle	_mm_cmpnle_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
* 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	F64vec2	F32vec4	F32vec1
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	ps	ss

Operators	Corresponding Intrinsic	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
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	ps	ss
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	ps	ss
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	ps	ss

Operators	Corresponding Intrinsic	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
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	ps	ss
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	ps	ss
select_ngt	_mm_cmpgt_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
select_nge	_mm_cmpge_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
select_nlt	_mm_cmplt_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
select_nle	_mm_cmple_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss

Operators	Corresponding Intrinsic	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
* 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	I8vec8	F64vec2	F32vec4	F32vec1
unpack_high	`_mm_unpackhi_[x]`	epi64	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	N/A
unpack_low	`_mm_unpacklo_[x]`	epi64	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	N/A
pack_sat	`_mm_packs_[x]`	N/A	epi32	epi16	N/A	pi32	pi16	N/A	N/A	N/A	N/A
packu_sat	`_mm_packus_[x]`	N/A	N/A	epi16	N/A	N/A	pu16	N/A	N/A	N/A	N/A
sat_add	`_mm_adds_[x]`	N/A	N/A	epi16	epi8	N/A	pi16	pi8	pd	ps	ss
sat_sub	`_mm_subs_[x]`	N/A	N/A	epi16	epi8	N/A	pi16	pi8	pi16	pi8	pd

Conversions Operators: Corresponding Intrinsics and Classes

Operators	Corresponding Intrinsic
F64vec2ToInt	`_mm_cvttsd_si32`
F32vec4ToF64vec2	`_mm_cvtps_pd`
F64vec2ToF32vec4	`_mm_cvtpd_ps`
IntToF64vec2	`_mm_cvtsi32_sd`

Operators	Corresponding Intrinsic
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);

}
```