



Intel[®] C/C++ Compiler Intrinsic

Reference Manual

Intel Confidential

Copyright © 1999 Intel Corporation

All Rights Reserved

Issued in U.S.A.

Order Number: 748639-001

World Wide Web: <http://developer.intel.com>

This **Intel C/C++ Compiler Intrinsic**s 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/C++ Compiler Intrinsic**s 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 1999.

*Third-party brands and names are the property of their respective owners.

Intel Corporation Confidential Information

This software is supplied under the terms of the Intel Pre-Release Program and License Agreement previously executed by your company and Intel, and may only be used, copied, or disclosed in accordance with the terms of that agreement. Notwithstanding the above, Intel may also provide third party software hereunder which may be subject to and used in accordance with any additional terms and conditions included with such software.

Contents

Chapter 1	Overview of New Instructions and New Extensions	
	Benefits of Using Intrinsics	1-2
	New Registers	1-2
	New Data Types	1-3
	The __m64 Data Type	1-4
	The __m128 Data Types	1-4
	New Data Types Usage Guidelines	1-4
	Intrinsic Conventions	1-5
	Intrinsic Syntax	1-6
	Intrinsic Categories and Supporting Extensions.....	1-7
	About This Manual	1-8
	Manual Organization	1-8
	Audience for This Manual	1-8
	Sources of Related Information	1-8
	Notational Conventions	1-9
Chapter 2	Support for MMX™ Technology	
	MMX Technology Intrinsic Groups.....	2-1
	The EMMS Instruction: Why You Need it and When to Use it.....	2-2
	Guidelines for When to Use EMMS.....	2-3
	MMX Technology General Support Intrinsics	2-4
	MMX Technology Packed Arithmetic Intrinsics.....	2-6
	MMX Technology Shift Intrinsics	2-9
	MMX Technology Logical Intrinsics	2-11
	MMX Technology Compare Intrinsics.....	2-12

	MMX Technology Set Intrinsics	2-13
Chapter 3	Streaming SIMD Extensions	
	Floating Point Intrinsics Using Streaming SIMD Extensions	3-1
	Arithmetic Operations	3-2
	Logical Operations	3-7
	Comparisons.....	3-8
	Conversion Operations	3-16
	Miscellaneous Intrinsics Using Streaming SIMD Extensions	3-19
	Macro Function for Shuffle Using Streaming SIMD Extensions	3-22
	Macro Functions to Read and Write the Control Registers.....	3-23
	Macro Function for Matrix Transposition.....	3-25
	Memory and Initialization Using Streaming SIMD Extensions.....	3-26
	Load Operations	3-27
	Set Operations	3-29
	Store Operations.....	3-30
	Integer Intrinsics Using Streaming SIMD Extensions	3-32
	Cacheability Support Using Streaming SIMD Extensions	3-35
Chapter 4	Willamette New Instructions	
	Willamette Floating Point Intrinsics.....	4-1
	Arithmetic Operations	4-2
	Logical Operations	4-4
	Comparisons.....	4-5
	Conversion Operations	4-11
	Miscellaneous Operations	4-14
	Willamette Floating-Point Memory and Initialization Operations	4-15
	Load Operations	4-15
	Set Operations	4-16
	Store Operations.....	4-17
	Willamette Integer Intrinsics	4-19
	Arithmetic Operations	4-19
	Logical Operations	4-26

Shift Operations	4-27
Comparisons	4-31
Conversions	4-33
Miscellaneous Operations	4-34
Macro Function for Shuffle	4-38
Willamette Integer Memory and Initialization	4-39
Load Operations	4-39
Set Operations	4-39
Store Operations	4-43
 Chapter 5 Data Alignment, Assembly and Processor Dispatch Support	
Alignment Support	5-1
Allocating and Freeing Aligned Memory Blocks.....	5-3
Processor Dispatch Support	5-3
Dynamic Stack Frame Alignment	5-6
Assembly Language Support	5-7
Inline Assembly.....	5-7
Generation of Assembly files	5-8

Overview of New Instructions and New Extensions

1

The Pentium® III Processor and other processors such as the Pentium with MMX™ technology and Pentium II processors have instructions to enable development of optimized multimedia applications. The instructions are implemented through extensions to the standard instructions available on other processors. 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 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. Instead, Intel provides easy implementation through the use of API extension sets referred to as intrinsics.

Table 1-1 Intrinsic Availability on Intel® Processors

Processors:	MMX Technology Intrinsics	Streaming SIMD Extensions	Willamette New Instructions
Willamette Processor	√	√	√
Pentium III Processor	√	√	N/A
Pentium II Processor	√	N/A	N/A
Pentium with MMX Technology	√	N/A	N/A
Pentium Pro Processor	N/A	N/A	N/A
Pentium Processor	N/A	N/A	N/A

Benefits of Using Intrinsics

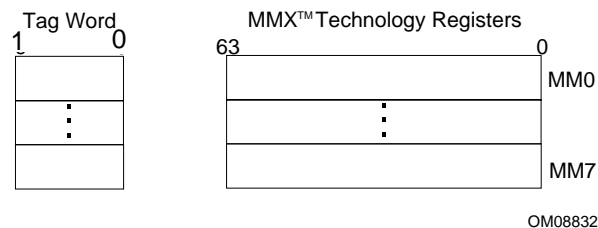
The major benefit of using intrinsics is that you now have access to key features that are not available using standard coding practices:

- *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.

New Registers

A key feature provided by the architecture of the processors are new registers sets. The MMX technology intrinsics provide 8 new registers (mm0 to mm7) that are 64 bits long (0 to 63).

Figure 1-1 MMX™ Technology Register



The Streaming SIMD Extensions and the Willamette New Instructions make use of yet another eight registers (xmm0 to xmm7) that are 128 bits in length.

Figure 1-2 Streaming SIMD and Willamette New Instruction Registers

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). To enable SIMD processing with the C/C++ Compiler, new data types are defined to exploit the expanded size of the new registers.

Using intrinsics enables you to code with the syntax of C function calls and variables instead of assembly language. 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.

New Data Types

New C data types, representing the new registers are used as the operands to these intrinsic functions. These data types are listed in [Table 1-2](#).

Table 1-2 New Data Types Available for Intrinsic Extensions

New Data Type	MMX™ Technology	Streaming SIMD Extensions	Willamette New Instructions
<code>__m64</code>	√	√	√
<code>__m128</code>	N/A	√	√
<code>__m128d</code>	N/A	N/A	√
<code>__m128i</code>	N/A	N/A	√

The __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.

The __m128 Data Types

The `__m128`, `__m128d`, and `__m128i` data types represent the contents of XMM registers.

<code>__m128</code>	used for single-precision floating point data
<code>__m128d</code>	used for double-precision floating point data
<code>__m128i</code>	used for integer data of 8, 16, 32, or 64 bits

The compiler aligns `__m128`, `__m128d`, and `__m128i` local data to 16B 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 alignment `declspec`.

Because the new instruction set treats 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 as you might expect. For scalar operations, you should use the `__m128` objects and the “scalar” forms of the intrinsics; the compiler and the processor implement these operations with 32-bit memory references.

New Data Types Usage Guidelines

The new data types listed earlier in [Table 1-2](#) are not basic ANSI C data types, and therefore you must observe the following usage restrictions:

- Use new data types 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 new data types as objects in aggregates, such as unions to access the byte elements and structures; the address of an `__m64` or `__m128` object may be taken.
- Use new data types only with the respective intrinsics described in this guide.

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: Instruction Set Reference Manual*.

Intrinsic Conventions

The intrinsics use a syntax notation 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:

<code>s</code>	single-precision floating point
<code>d</code>	double-precision floating point
<code>i128</code>	signed 128-bit integer
<code>i64</code>	signed 64-bit integer
<code>u64</code>	unsigned 64-bit integer
<code>i32</code>	signed 32-bit integer
<code>u32</code>	unsigned 32-bit integer
<code>i16</code>	signed 16-bit integer
<code>u16</code>	unsigned 16-bit integer
<code>i8</code>	signed 8-bit integer
<code>u8</code>	unsigned 8-bit integer

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:

```
127 ┌───┬───┐ 0
    │ 2.0 │ 1.0 │
```

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

The syntax is defined for each intrinsic before the definition as follows:

```
data_type intrinsic_name (parameters)
```

Where,

<code>data_type</code>	Is the return data type, which can be either <code>void</code> , <code>int</code> , <code>__m64</code> , <code>__m128</code> , <code>__m128d</code> , <code>__m128i</code> . Only the <code>_mm_empty</code> intrinsic returns <code>void</code> .
<code>intrinsic_name</code>	Is the name of the intrinsic, which behaves like a function that you can use in your C/C++ code instead of inlining the actual instruction.

A table in each section provides the intrinsic names, alternate names, and the corresponding instruction.

Intrinsic Categories and Supporting Extensions

The intrinsics are grouped into the following categories and can be applied if you have the processors specified in the extension sets in [Table 1-3](#).

Table 1-3 Intrinsic Categories and Supporting Extensions

Intrinsic Category	MMX™ Technology	Streaming SIMD Extensions	Willamette New Instructions
General Support	√	√	√
Integer Arithmetic	√		√
Integer Shift	√		√
Integer Logical	√		√
Integer Compare	√		√
Integer Conversions	√		√
Integer Load			√
Integer Set	√		√
Integer Store			√
Integer Miscellaneous	√	√	√
FP Arithmetic		√	√
FP Shift		√	√
FP Logical		√	√
FP Compare		√	√
FP Conversion		√	√
FP Load		√	√
FP Set		√	√
FP Store		√	√
FP Miscellaneous		√	√

About This Manual

This manual describes the intrinsics for the Intel C/C++ compiler.

Manual Organization

This manual contains these chapters and appendixes:

- | | |
|-----------|--|
| Chapter 1 | “Overview of New Instructions and New Extensions.” Introduces the intrinsics, provides information on manual organization and explains notational conventions. |
| Chapter 2 | “Support for MMX™ Technology” Describes the MMX technology intrinsics. |
| Chapter 3 | “Streaming SIMD Extensions.” Describes the Streaming SIMD Extensions. |
| Chapter 4 | “Willamette New Instructions.” Describes the Willamette New Instructions. |
| Chapter 5 | “Data Alignment, Assembly and Processor Dispatch Support.” Describes the available support for data alignment, processor dispatch, and assembly language. |

Audience for This Manual

This manual 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 assembly language programming.

Sources of Related Information

For the latest information about the Intel C/C++ Compiler and other performance tuning tools from Intel, including product updates and technical support, visit this web site at <http://developer.intel.com>.

The following sources of additional information may be of assistance:

- *Intel C/C++ Compiler User's Guide: With Support for the Willamette New Instructions*, order number 749099-B002.
- *The Annotated C++ Reference Manual*, 1st edition, Ellis, Margaret; Stroustrup, Bjarne, Addison Wesley, 1991. Provides information on the C++ programming language.

The following documents provide information on Intel architecture. These documents are available in the Intel Architecture Performance Training Center, or on the corporate web site at <http://developer.intel.com>.

- *Intel Architecture Software Developer's Manual*, Volume 1: Basic Architecture. Intel Corporation, order number 243190.
- *Intel Architecture Software Developer's Manual*, Volume 2: Instruction Set Reference Manual. Intel Corporation, order number 243191.
- *Intel Architecture Optimization Manual*. Intel Corporation, order number 730795.

Notational Conventions

This manual uses the following conventions:

<code>This type style</code>	Indicates an element of syntax, a reserved word, a keyword, a filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant.
<code>1</code>	Indicates lowercase letter L in examples. <code>1</code> is the number 1 in examples. <code>o</code> is the uppercase O in examples. <code>0</code> is the number 0 in examples.
This type style	Indicates the exact characters you type as input.
<i>This type style</i>	Indicates a place-holder 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 options.
{ item item }	Indicates to elect only one of the items listed between braces. A vertical bar () separates the items.
... (ellipses)	Indicates that you can repeat the preceding item.

Support for MMX™ Technology

2

Intel's MMX technology is an extension to the Intel architecture (IA) instruction set. The technology uses a single instruction, multiple data (SIMD) technique to speed up multimedia and communications software by processing data elements in parallel.

The MMX instruction set adds 57 opcodes and a 64-bit quadword data type. In addition, there are eight 64-bit MMX technology registers, each of which can be directly addressed using the register names MM0 to MM7. [Figure 2-1](#) shows the layout of the eight MMX technology registers.

The MMX technology is operating-system-transparent and 100% compatible with all existing Intel architecture software. Therefore all applications will continue to run on processors with MMX technology. Additional information and details about the MMX instructions, data types, and registers can be found in the *Intel Architecture MMX Technology Programmer's Reference Manual*, order number 243007.

MMX Technology Intrinsic Groups

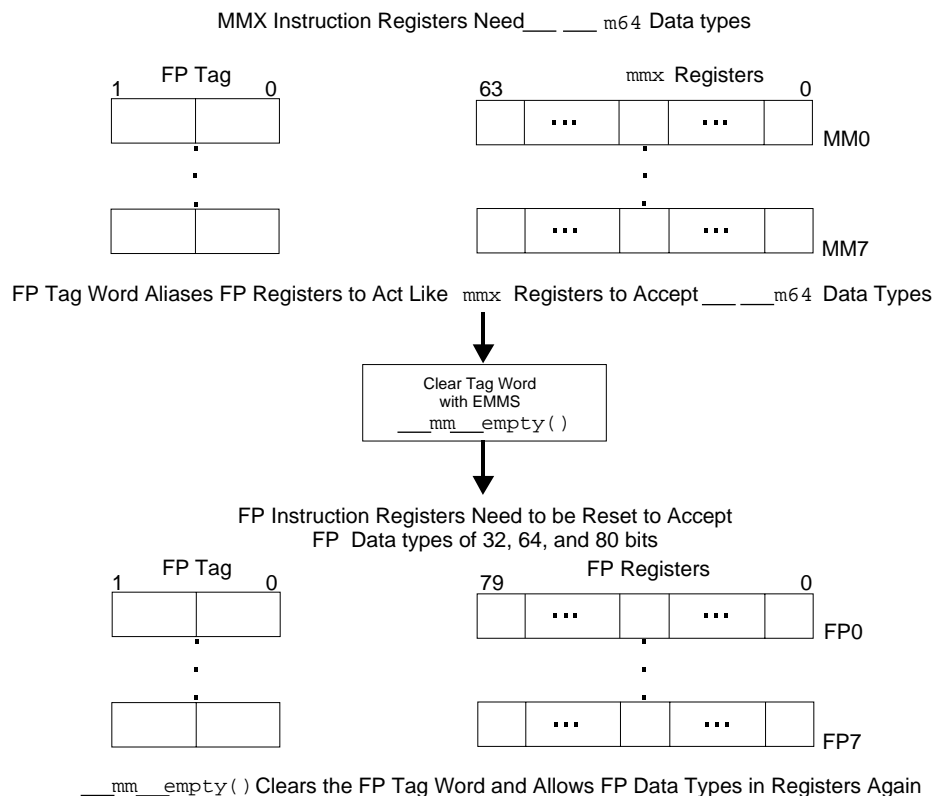
The MMX technology intrinsics are grouped into the following sets:

- General Support Intrinsics
- Packed Arithmetic Intrinsics
- Shift Intrinsics
- Logical Intrinsics
- Compare Intrinsics

The EMMS Instruction: Why You Need it and When to Use it

Using `EMMS` is like emptying a container to accommodate new content. For instance, MMX 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.

Figure 2-1 Why You Need EMMS to Reset After an MMX Instruction



OM08831



CAUTION. *Failure to reset the tag word for FP instructions after using an MMX instruction can result in unexpected execution or poor performance.*

Guidelines for When to Use EMMS

These guidelines help you determine when to use EMMS:

- *If next instruction is FP*—Use `_mm_empty()` after an MMX instruction if the next instruction is an FP instruction; for example, before doing calculations on floats, doubles or long doubles.
- *Don't empty when already empty*—If the next instruction uses an MMX register, `_mm_empty()` incurs an operation with no benefit (no-op).
- *Group Instructions*—Use different functions for regions that use FP instructions and those that use MMX instructions. This eliminates needing an EMMS instruction within the body of a critical loop.
- *Runtime initialization*—Use `_mm_empty()` during runtime initialization of `__m64` and FP data types. This ensures resetting the register between data type transitions. See usage coding [Example 2-1](#)

Example 2-1 Correct EMMS Usage In Initialization Code

Incorrect Usage

```
__m64 x = _m_padd(y, z);
float f = init();
```

Correct Usage

```
__m64 x = _m_padd(y, z);
float f = (_mm_empty(), init());
```

Further, you must be aware of all the situations when your code generates an MMX instruction with the Intel C/C++ compiler:

- when using an MMX intrinsic
- when using the Streaming SIMD Extensions (for those intrinsics that use MMX data)

- when using an MMX instruction through inline assembly
- when referencing an `__m64` data type variable

For more documentation on EMMS, visit the following web site:

<http://developer.intel.com>

MMX Technology General Support Intrinsics

The general support intrinsics are listed in [Table 2-1](#), and are followed by a description of each intrinsic.

Table 2-1 General Support Intrinsics

Intrinsic Name	Assembly Instruction	Operation	Signed	Saturation
<code>_mm_empty</code>	EMMS	Empty MM State	—	—
<code>_mm_cvtsi32_si64</code>	MOVD	Convert from int	—	—
<code>_mm_cvtsi64_si32</code>	MOVD	Convert from int	—	—
<code>_mm_packs_pi16</code>	PACKSSWB	Pack	Yes	Yes
<code>_mm_packs_pi32</code>	PACKSSDW	Pack	Yes	Yes
<code>_mm_packs_pu16</code>	PACKUSWB	Pack	No	Yes
<code>_mm_unpackhi_pi8</code>	PUNPCKHBW	Interleave	—	—
<code>_mm_unpackhi_pi16</code>	PUNPCKHWD	Interleave	—	—
<code>_mm_unpackhi_pi32</code>	PUNPCKHDQ	Interleave	—	—
<code>_mm_unpacklo_pi8</code>	PUNPCKLBW	Interleave	—	—
<code>_mm_unpacklo_pi16</code>	PUNPCKLWD	Interleave	—	—
<code>_mm_unpacklo_pi32</code>	PUNPCKLDQ	Interleave	—	—

```
void _mm_empty (void) EMMS
```

Empty the multimedia state.

See [“The EMMS Instruction: Why You Need it and When to Use it”](#).

```
__m64 _mm_cvtsi32_si64 (int i) MOVD
```

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

```
int _mm_cvtsi64_si32 (__m64 m) MOVD
```

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

```
__m64 _mm_packs_pi16 (__m64 m1, __m64 m2) PACKSSWB
```

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 _mm_packs_pi32 (__m64 m1, __m64 m2) PACKSSDW
```

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 _mm_packs_pu16 (__m64 m1, __m64 m2) PACKUSWB
```

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 _mm_unpackhi_pi8 (__m64 m1, __m64 m2) PUNPCKHBW
```

Interleave the four 8-bit values from the high half of *m1* with the four values from the high half of *m2* and take the least significant element from *m1*.

```
__m64 _mm_unpackhi_pi16 (__m64 m1, __m64 m2) PUNPCKHWD
```

Interleave the two 16-bit values from the high half of *m1* with the two values from the high half of *m2* and take the least significant element from *m1*.

```
__m64 _mm_unpackhi_pi32 (__m64 m1, __m64 m2) PUNPCKHDQ
```

Interleave the 32-bit value from the high half of *m1* with the 32-bit value from the high half of *m2* and take the least significant element from *m1*.

```
__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2) PUNPCKLBW
```

Interleave the four 8-bit values from the low half of *m1* with the four values from the low half of *m2* and take the least significant element from *m1*.

`__m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)` PUNPCKLWD

Interleave the two 16-bit values from the low half of `m1` with the two values from the low half of `m2` and take the least significant element from `m1`.

`__m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)` PUNPCKLDQ

Interleave the 32-bit value from the low half of `m1` with the 32-bit value from the low half of `m2` and take the least significant element from `m1`.

MMX Technology Packed Arithmetic Intrinsics

The packed arithmetic intrinsics are listed in [Table 2-2](#), and are followed by a description of each intrinsic.

Table 2-2 Packed Arithmetic Intrinsics

Intrinsic Name	Corresponding Instruction	Operation	Signed	Argument Values/Bits	Result Values/Bits
<code>_mm_add_pi8</code>	PADDB	Addition	—	8/8	8/8
<code>_mm_add_pi16</code>	PADDW	Addition	—	4/16	4/16
<code>_mm_add_pi32</code>	PADDQ	Addition	—	2/32	2/32
<code>_mm_adds_pi8</code>	PADDSB	Addition	Yes	8/8	8/8
<code>_mm_adds_pi16</code>	PADDSD	Addition	Yes	4/16	4/16
<code>_mm_adds_pu8</code>	PADDUSB	Addition	No	8/8	8/8
<code>_mm_adds_pu16</code>	PADDUSW	Addition	No	4/16	4/16
<code>_mm_sub_pi8</code>	PSUBB	Subtraction	—	8/8	8/8
<code>_mm_sub_pi16</code>	PSUBW	Subtraction	—	4/16	4/16
<code>_mm_sub_pi32</code>	PSUBD	Subtraction	—	2/32	2/32
<code>_mm_subs_pi8</code>	PSUBSB	Subtraction	Yes	8/8	8/8
<code>_mm_subs_pi16</code>	PSUBSD	Subtraction	Yes	4/16	4/16
<code>_mm_subs_pu8</code>	PSUBUSB	Subtraction	No	8/8	8/8
<code>_mm_subs_pu16</code>	PSUBUSW	Subtraction	No	4/16	4/16
<code>_mm_madd_pi16</code>	PMADDWD	Multiplication	—	4/16	2/32
<code>_mm_mulhi_pi16</code>	PMULHW	Multiplication	Yes	4/16	4/16 (high)
<code>_mm_mullo_pi16</code>	PMULLW	Multiplication	—	4/16	4/16 (low)

```
__m64 _mm_add_pi8 (__m64 m1, __m64 m2)          PADDB
```

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

```
__m64 _mm_add_pi16 (__m64 m1, __m64 m2)         PADDW
```

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

```
__m64 _mm_add_pi32 (__m64 m1, __m64 m2)         PADDD
```

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

```
__m64 _mm_adds_pi8 (__m64 m1, __m64 m2)         PADDSB
```

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

```
__m64 _mm_adds_pi16 (__m64 m1, __m64 m2)        PADDSW
```

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

```
__m64 _mm_adds_pu8 (__m64 m1, __m64 m2)         PADDUSB
```

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

```
__m64 _mm_adds_pu16 (__m64 m1, __m64 m2)        PADDUSW
```

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

```
__m64 _mm_sub_pi8 (__m64 m1, __m64 m2)          PSUBB
```

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

```
__m64 _mm_sub_pi16 (__m64 m1, __m64 m2)         PSUBW
```

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

```
__m64 _mm_sub_pi32 (__m64 m1, __m64 m2)         PSUBD
```

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

`__m64 __mm_subs_pi8 (__m64 m1, __m64 m2)` PSUBSB

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

`__m64 __mm_subs_pi16 (__m64 m1, __m64 m2)` PSUBSW

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

`__m64 __mm_subs_pu8 (__m64 m1, __m64 m2)` PSUBUSB

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

`__m64 __mm_subs_pu16 (__m64 m1, __m64 m2)` PSUBUSW

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

`__m64 __mm_madd_pi16 (__m64 m1, __m64 m2)` PMADDWD

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 __mm_mulhi_pi16 (__m64 m1, __m64 m2)` PMULHW

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 __mm_mullo_pi16 (__m64 m1, __m64 m2)` PMULLW

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 Technology Shift Intrinsics

The shift intrinsics are listed in [Table 2-3](#), and are followed by a description of each intrinsic.

Table 2-3 Shift Intrinsics

Intrinsic Name	Shift Direction	Shift Type	Corresponding Instruction
<code>_mm_sll_pi16</code>	left	Logical	PSLLW
<code>_mm_slli_pi16</code>	left	Logical	PSLLWI
<code>_mm_sll_pi32</code>	left	Logical	PSLLD
<code>_mm_slli_pi32</code>	left	Logical	PSLLDI
<code>_mm_sll_si64</code>	left	Logical	PSLLQ
<code>_mm_slli_si64</code>	left	Logical	PSLLQI
<code>_mm_sra_pi16</code>	right	Arithmetic	PSRAW
<code>_mm_srai_pi16</code>	right	Arithmetic	PSRAWI
<code>_mm_sra_pi32</code>	right	Arithmetic	PSRAD
<code>_mm_srai_pi32</code>	right	Arithmetic	PSRADI
<code>_mm_srl_pi16</code>	right	Logical	PSRLW
<code>_mm_srli_pi16</code>	right	Logical	PSRLWI
<code>_mm_srl_pi32</code>	right	Logical	PSRLD
<code>_mm_srli_pi32</code>	right	Logical	PSRLDI
<code>_mm_srl_si64</code>	right	Logical	PSRLQ
<code>_mm_srli_si64</code>	right	Logical	PSRLQI

`__m64 _mm_sll_pi16 (__m64 m, __m64 count)` PSLLW

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

`__m64 _mm_slli_pi16 (__m64 m, int count)` PSLLWI

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 _mm_sll_pi32 (__m64 m, __m64 count)` PSLLD

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

```
__m64 _mm_slli_pi32 (__m64 m, int count)          PSLLDI
```

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 _mm_sll_si64 (__m64 m, __m64 count)         PSLLQ
```

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

```
__m64 _mm_slli_si64 (__m64 m, int count)          PSLLQI
```

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 _mm_sra_pi16 (__m64 m, __m64 count)         PSRAW
```

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

```
__m64 _mm_srai_pi16 (__m64 m, int count)          PSRAWI
```

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 _mm_sra_pi32 (__m64 m, __m64 count)         PSRAD
```

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

```
__m64 _mm_srai_pi32 (__m64 m, int count)          PSRAI
```

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 _mm_srl_pi16 (__m64 m, __m64 count)         PSRLW
```

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


```
__m64 _mm_srli_pi16 (__m64 m, int count)          PSRLWI
```

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 _mm_srl_pi32 (__m64 m, __m64 count)         PSRLD
```

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

```
__m64 _mm_srli_pi32 (__m64 m, int count)          PSRLDI
```

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 _mm_srl_si64 (__m64 m, __m64 count)         PSRLQ
```

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

```
__m64 _mm_srli_si64 (__m64 m, int count)          PSRLQI
```

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 Technology Logical Intrinsics

The logical intrinsics are listed in [Table 2-4](#), and are followed by a description of each intrinsic.

Table 2-4 Logical Intrinsics

Intrinsic Name	Operation	Corresponding Instruction
<code>_mm_and_si64</code>	Bitwise AND	PAND
<code>_mm_andnot_si64</code>	Logical NOT	PANDN
<code>_mm_or_si64</code>	Bitwise OR	POR
<code>_mm_xor_si64</code>	Bitwise Exclusive OR	PXOR

```
__m64 _mm_and_si64 (__m64 m1, __m64 m2)          PAND
```

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

`__m64 _mm_andnot_si64 (__m64 m1, __m64 m2)` PANDN

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 _mm_or_si64 (__m64 m1, __m64 m2)` POR

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

`__m64 _mm_xor_si64 (__m64 m1, __m64 m2)` PXOR

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

MMX Technology Compare Intrinsics

The compare intrinsics are listed in [Table 2-4](#), and are followed by a description of each intrinsic.

Table 2-5 Compare Intrinsics

Intrinsic Name	Comparison	Number of Elements	Element Bit Size	Corresponding Instruction
<code>_mm_cmpeq_pi8</code>	Equal	8	8	PCMPEQB
<code>_mm_cmpeq_pi16</code>	Equal	4	16	PCMPEQW
<code>_mm_cmpeq_pi32</code>	Equal	2	32	PCMPEQD
<code>_mm_cmpgt_pi8</code>	Greater Than	8	8	PCMPGTB
<code>_mm_cmpgt_pi16</code>	Greater Than	4	16	PCMPGTW
<code>_mm_cmpgt_pi32</code>	Greater Than	2	32	PCMPGTD

`__m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)` PCMPEQB

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 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)` PCMPEQW

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 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)          PCMPSEQD
```

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 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)           PCMPGTB
```

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 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)          PCMPGTW
```

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 _mm_cmpgt_pi32 (__m64 m1, __m64__m64 m2)     PCMPGTD
```

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 Technology Set Intrinsics

The set intrinsics are listed in [Table 2-6](#), followed by a description of each intrinsic. All set intrinsics are comprised of composite instructions.

Table 2-6 Set Intrinsics

Intrinsic Name	Operation	Number of Elements	Element Bit Size	Signed	Reverse Order
<code>_mm_setzero_si64</code>	set to zero	1	64	No	No
<code>_mm_set_pi32</code>	set integer values	2	32	No	No
<code>_mm_set_pi16</code>	set integer values	4	16	No	No
<code>_mm_set_pi8</code>	set integer values	8	8	No	No
<code>_mm_setl_pi32</code>	set integer values	2	32	Yes	No
<code>_mm_setl_pi16</code>	set integer values	4	16	Yes	No
<code>_mm_setl_pi8</code>	set integer values	8	8	Yes	No
<code>_mm_setr_pi32</code>	set integer values	2	32	No	Yes
<code>_mm_setr_pi16</code>	set integer values	4	16	No	Yes
<code>_mm_setr_pi8</code>	set integer values	8	8	No	Yes

```
__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 w3, short w2,  
                   short w1, short w0)
```

(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_setl_pi16 (short w)` (composite)

Sets the 4 signed 16-bit integer values to *w*.

`r0 := w`

`r1 := w`

`r2 := w`

`r3 := w`

`__m64 _mm_setl_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 i0, int i1)` (composite)

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

`r0 := i0`

`r1 := i1`

`__m64 _mm_setr_pi16 (short w0, short w1,
short w2, short w3)` (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 b0, char b1,          (composite)
                    char b2, char b3,
                    char b4, char b5,
                    char b6, char b7)
```

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

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

Streaming SIMD Extensions

3

This chapter describes the C/C++ language-level features supporting the Streaming SIMD Extensions in the Intel C/C++ Compiler. The section explains the following features of the intrinsics:

- Floating Point Intrinsics
- Memory and Initialization Intrinsics
- Integer Intrinsics
- Cacheability Support Intrinsics

Floating Point Intrinsics Using Streaming SIMD Extensions

Each intrinsic entry has an informal pseudo-code and it is followed with a corresponding instruction name in upper case letters; for example, `ADDSS` is the name of the first instruction listed in this section, which corresponds to the intrinsic for the following:

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

The variable `r` is generally used for the intrinsic's return value. 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.

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 might 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

The arithmetic intrinsics are listed in [Table 3-1](#), and are followed by a description of each intrinsic.

Table 3-1 Packed Arithmetic Intrinsics

Intrinsic	Instruction	Operation	R0	R1	R2	R3
<code>_mm_add_ss</code>	ADDSS	Addition	a0 [op] b0	a1	a2	a3
<code>_mm_add_ps</code>	ADDPS	Addition	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
<code>_mm_sub_ss</code>	SUBSS	Subtraction	a0 [op] b0	a1	a2	a3
<code>_mm_sub_ps</code>	SUBPS	Subtraction	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
<code>_mm_mul_ss</code>	MULSS	Multiplication	a0 [op] b0	a1	a2	a3
<code>_mm_mul_ps</code>	MULPS	Multiplication	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
<code>_mm_div_ss</code>	DIVSS	Division	a0 [op] b0	a1	a2	a3
Continued						

Table 3-1 Packed Arithmetic Intrinsics (continued)

Intrinsic	Instruction	Operation	R0	R1	R2	R3
<code>_mm_div_ps</code>	DIVPS	Division	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
<code>_mm_sqrt_ss</code>	SQRTSS	Finds Square Root	[op] a0	a1	a2	a3
<code>_mm_sqrt_ps</code>	SQRTPS	Finds Square Root	[op] a0	[op] b1	[op] b2	[op] b3
<code>_mm_rcp_ss</code>	RCPSS	Reciprocal	[op] a0	a1	a2	a3
<code>_mm_rcp_ps</code>	RCPPS	Reciprocal	[op] a0	[op] b1	[op] b2	[op] b3
<code>_mm_rsqrt_ss</code>	RSQRTSS	Finds Reciprocal Square Root	[op] a0	a1	a2	a3
<code>_mm_rsqrt_ps</code>	RSQRTPS	Finds Reciprocal Square Root	[op] a0	[op] b1	[op] b2	[op] b3
<code>_mm_min_ss</code>	MINSS	Computes Minimum	[op] (a0, b0)	a1	a2	a3
<code>_mm_min_ps</code>	MINPS	Computes Minimum	[op] (a0, b0)	[op] (a1, b1)	[op] (a2, b2)	[op] (a3, b3)
<code>_mm_max_ss</code>	MAXSS	Computes Maximum	[op] (a0, b0)	a1	a2	a3
<code>_mm_max_ps</code>	MAXPS	Computes Maximum	[op] (a0, b0)	[op] (a1, b1)	[op] (a2, b2)	[op] (a3, b3)

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

ADDSS

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)
```

ADDPS

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) SUBSS
```

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) SUBPS
```

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) MULSS
```

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) MULPS
```

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) DIVSS
```

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) DIVPS
```

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) SQRTSS
```

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) SQRTPS
```

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) RCPSS
```

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) RCPPS
```

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) RSQRTSS
```

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) RSQRTPS
```

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) MINSS
```

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) MINPS
```

Computes the minimums 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) MAXSS
```

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) MAXPS
```

Computes the maximums 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

The logical intrinsics are listed in [Table 3-2](#), and are followed by a description of each intrinsic.

Table 3-2 Logical Intrinsics

Intrinsic Name	Operation	Corresponding Instruction
<code>_mm_and_ps</code>	Bitwise AND	ANDPS
<code>_mm_andnot_ps</code>	Logical NOT	ANDNPS
<code>_mm_or_ps</code>	Bitwise OR	ORPS
<code>_mm_xor_ps</code>	Bitwise Exclusive OR	XORPS

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

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) ANDNPS
```

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) ORPS
```

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) XORPS
```

Computes bitwise EXOR (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

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 superscript 'r' on the instruction indicates operands are reversed in the instruction implementation. The compare intrinsics are listed in [Table 3-3](#), and are followed by a description of each intrinsic.

Table 3-3 Compare Intrinsics

Intrinsic Name	Comparison	Corresponding Instruction
<code>_mm_cmpeq_ss</code>	Equal	CMPEQSS
<code>_mm_cmpeq_ps</code>	Equal	CMPEQPS
<code>_mm_cmplt_ss</code>	Less Than	CMPLTSS
Continued		

Table 3-3 Compare Intrinsics (continued)

Intrinsic Name	Comparison	Corresponding Instruction
<code>_mm_cmplt_ps</code>	Less Than	CMPLETPS
<code>_mm_cmple_ss</code>	Less Than or Equal	CMPLESS
<code>_mm_cmple_ps</code>	Less Than or Equal	CMPLEPS
<code>_mm_cmpgt_ss</code>	Greater Than	CMPGTSS
<code>_mm_cmpgt_ps</code>	Greater Than	CMPGTTPS
<code>_mm_cmpge_ss</code>	Greater Than or Equal	CMPGESS
<code>_mm_cmpge_ps</code>	Greater Than or Equal	CMPGEPS
<code>_mm_cmpneq_ss</code>	Not Equal	CMPNEQSS
<code>_mm_cmpneq_ps</code>	Not Equal	CMPNEQPS
<code>_mm_cmpnlt_ss</code>	Not Less Than	CMPNLTSS
<code>_mm_cmpnlt_ps</code>	Not Less Than	CMPNLTTPS
<code>_mm_cmpnle_ss</code>	Not Less Than or Equal	CMPNLESS
<code>_mm_cmple_ps</code>	Not Less Than or Equal	CMPNLEPS
<code>_mm_cmpngt_ss</code>	Not Greater Than	CMPNLTSS
<code>_mm_cmpngt_ps</code>	Not Greater Than	CMPNLTTPS
<code>_mm_cmpnge_ss</code>	Not Greater Than or Equal	CMPNLESS
<code>_mm_cmpnge_ps</code>	Not Greater Than or Equal	CMPNLEPS
<code>_mm_cmpord_ss</code>	Ordered	CMPORDSS
<code>_mm_cmpord_ps</code>	Ordered	CMPORDPS
<code>_mm_cmpunord_ss</code>	Unordered	CMPUNORDSS
<code>_mm_cmpunord_ps</code>	Unordered	CMPUNORDPS
<code>_mm_comieq_ss</code>	Equal	COMISS
<code>_mm_comilt_ps</code>	Less Than	COMISS
<code>_mm_comile_ss</code>	Less Than or Equal	COMISS

Continued

Table 3-3 Compare Intrinsics (continued)

Intrinsic Name	Comparison	Corresponding Instruction
<code>_mm_comigt_ss</code>	Greater Than	COMISS
<code>_mm_comige_ss</code>	Greater Than or Equal	COMISS
<code>_mm_comineq_ss</code>	Not Equal	COMISS
<code>_mm_ucomieq_ss</code>	Equal	UCOMISS
<code>_mm_ucomilt_ss</code>	Less Than	UCOMISS
<code>_mm_ucomile_ss</code>	Less Than or Equal	UCOMISS
<code>_mm_ucomigt_ss</code>	Greater Than	UCOMISS
<code>_mm_ucomige_ss</code>	Greater Than or Equal	UCOMISS
<code>_mm_ucomineq_ss</code>	Not Equal	UCOMISS

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

Compare for equality.

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

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

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)` CMPLTSS

Compare for less-than.

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



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

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) CMPLSS
```

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) CMPLPS
```

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) CMPLTSSr
```

Compare for greater-than.

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

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

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) CMPLSSr
```

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) CMPLEPSr
```

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) CMPNEQSS
```

Compare for inequality.

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

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

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) CMPNLTSS
```

Compare for not-less-than.

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

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

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) CMPNLESS
```

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) CMPNLEPS
```

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) CMPNLTSSr
```

Compare for not-greater-than.

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

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

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) CMPNLESSr
```

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) CMPNLEPSr
```

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) CMPORDSS
```

Compare for ordered.

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

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

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)        CMPUNORDSS
```

Compare for unordered.

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

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

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)            COMISS
```

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)            COMISS
```

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)            COMISS
```

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)          COMISS
```

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)          COMISS
```

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)         COMISS
```

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)         UCOMISS
```

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)         UCOMISS
```

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)         UCOMISS
```

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)         UCOMISS
```

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_ucomige_ss (__m128 a, __m128 b)          UCOMISS
```

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)         UCOMISS
```

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

The conversions operations are listed in [Table 3-4](#), 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.

Table 3-4 Conversion Operations

Intrinsic Name	Corresponding Instruction
<code>_mm_cvtss_si32</code>	CVTSS2SI
<code>_mm_cvtps_pi32</code>	CVTPS2PI
<code>_mm_cvtss_si32</code>	CVTSS2SI
<code>_mm_cvttps_pi32</code>	CVTTPS2PI
<code>_mm_cvtsi32_ss</code>	CVTSI2SS
<code>_mm_cvtpi32_ps</code>	CVTTPS2PI
<code>_mm_cvtpi16_ps</code>	composite
<code>_mm_cvtpu16_ps</code>	composite
<code>_mm_cvtpi8_ps</code>	composite
<code>_mm_cvtpu8_ps</code>	composite
<code>_mm_cvtpi32x2_ps</code>	composite
<code>_mm_cvtps_pi16</code>	composite
<code>_mm_cvtps_pi8</code>	composite

```
int _mm_cvtss_si32(__m128 a) CVTSS2SI
```

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

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

```
__m64 _mm_cvtps_pi32(__m128 a) CVTPS2PI
```

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_cvttsi_si32(__m128 a) CVTTSS2SI
```

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

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

```
__m64 _mm_cvttps_pi32(__m128 a) CVTTPS2PI
```

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_cvtsi32_ss(__m128 a, int b) CVTSI2SS
```

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_cvtpi32_ps(__m128 a, __m64 b) CVTPI2PS
```

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
```

`__m128 _mm_cvtpi16_ps(__m64 a)` (composite)

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
```

`__m128 _mm_cvtpu16_ps(__m64 a)` (composite)

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
```

`__m128 _mm_cvtpi8_ps(__m64 a)` (composite)

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
```

`__m128 _mm_cvtpu8_ps(__m64 a)` (composite)

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
```

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

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
```

```
__m64 _mm_cvtps_pi16(__m128 a) (composite)
```

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
```

```
__m64 _mm_cvtps_pi8(__m128 a) (composite)
```

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
```

Miscellaneous Intrinsics Using Streaming SIMD Extensions

The miscellaneous intrinsics are listed in [Table 3-5](#), and are followed by a description of each intrinsic.

Table 3-5 Conversion Operations

Intrinsic Name	Operation	Corresponding Instruction
<code>_mm_shuffle_ps</code>	Shuffle	SHUFPS
<code>_mm_unpackhi_ps</code>	Unpack High	UNPCKHPS
<code>_mm_unpacklo_ps</code>	Unpack Low	UNPCKLPS
<code>_mm_loadh_pi</code>	Load High	MOVHPS reg, mem
<code>_mm_storeh_pi</code>	Store High	MOVHPS mem, reg
Continued		
<code>_mm_movehl_ps</code>	Move High to Low	MOVHLPS
<code>_mm_movelh_ps</code>	Move Low to High	MOVLHPS
<code>_mm_loadl_pi</code>	Load Low	MOVLPS reg, mem
<code>_mm_storel_pi</code>	Store Low	MOVLPS mem, reg

Table 3-5 Conversion Operations (continued)

Intrinsic Name	Operation	Corresponding Instruction
<code>_mm_movemask_ps</code>	Create four-bit mask	MOVMSKPS
<code>_mm_getcsr</code>	Return Register Contents	STMXCSR
<code>_mm_setcsr</code>	Set Control Register	LDMXCSR

```
__m128 _mm_shuffle_ps(__m128 a, __m128 b, int i) SHUFPS
```

Selects four specific SP FP values from *a* and *b*, based on the mask *i*. The mask must be an immediate. See [“Macro Function for Shuffle Using Streaming SIMD Extensions”](#) in the end of this section for a description of the shuffle semantics.

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

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) UNPCKLPS
```

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 a, __m64 *p) MOVHPS reg, mem
```

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

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

```
void _mm_storeh_pi(__m64 *p, __m128 a) MOVHPS mem, reg
```

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

```
*p0 := a2
```

```
*p1 := a3
```

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

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) MOVLHPS
```

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 *p) MOVLPS reg, mem
```

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) MOVLPS mem, reg
```

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

```
*p0 := b0
```

```
*p1 := b1
```

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

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)
```

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

Returns the contents of the control register.

```
void _mm_setcsr(unsigned int i) LDMXCSR
```

Sets the control register to the value specified.

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 [Example 3-1](#).

Example 3-1 Shuffle Function Macro

```
_MM_SHUFFLE(z,y,x,w)
```

expands to the value of

```
(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.

Example 3-2 View of Original and Result Words With Shuffle Function Macro

```

; m1 = 127
; m2 = 127
m3 = _mm_shuffle_ps(m1, m2,
_MM_SHUFFLE(1, 0, 3, 2))
; m3 = 127

```

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”](#) later in this chapter.

Exception State Macros

`_MM_SET_EXCEPTION_STATE(x)`

`_MM_GET_EXCEPTION_STATE()`

Macro Definitions

Write to and read from the sixth-least significant control register bit, respectively.

Macro Arguments

`_MM_EXCEPT_INVALID`

`_MM_EXCEPT_DIV_ZERO`

`_MM_EXCEPT_DENORM`

`_MM_EXCEPT_OVERFLOW`

`_MM_EXCEPT_UNDERFLOW`

`_MM_EXCEPT_INEXACT`

[Example 3-3](#) tests for a divide-by-zero exception.

Example 3-3 Exception State Macros with `_MM_EXCEPT_DIV_ZERO`

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

Exception Mask Macros

`_MM_SET_EXCEPTION_MASK(x)`
`_MM_GET_EXCEPTION_MASK()`

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.

Macro Arguments

`_MM_MASK_INVALID`
`_MM_MASK_DIV_ZERO`
`_MM_MASK_DENORM`
`_MM_MASK_OVERFLOW`
`_MM_MASK_UNDERFLOW`
`_MM_MASK_INEXACT`

[Example 3-4](#) masks the overflow and underflow exceptions and unmask all other exceptions.

Example 3-4 Exception Mask with `_MM_MASK_OVERFLOW` and `_MM_MASK_UNDERFLOW`

```
_MM_SET_EXCEPTION_MASK(_MM_MASK_OVERFLOW | _MM_MASK_UNDERFLOW)
```

Rounding Mode

`_MM_SET_ROUNDING_MODE(x)`
`_MM_GET_ROUNDING_MODE()`

Macro Definition

Macro Arguments

`_MM_ROUND_NEAREST`
`_MM_ROUND_DOWN`
`_MM_ROUND_UP`
`_MM_ROUND_TOWARD_ZERO`

Write to and read from bits thirteen and fourteen of the control register.

[Example 3-5](#) tests the rounding mode for round toward zero.

Example 3-5 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

`_MM_SET_FLUSH_ZERO_MODE(x)`

`_MM_GET_FLUSH_ZERO_MODE()`

Macro Arguments

`_MM_FLUSH_ZERO_ON`

`_MM_FLUSH_ZERO_OFF`

Macro Definition

Write to and read from bit fifteen of the control register.

[Example 3-6](#) disables flush-to-zero mode.

Example 3-6 Flush-to-Zero Mode with `_MM_FLUSH_ZERO_OFF`

```
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_OFF)
```

Macro Function for Matrix Transposition

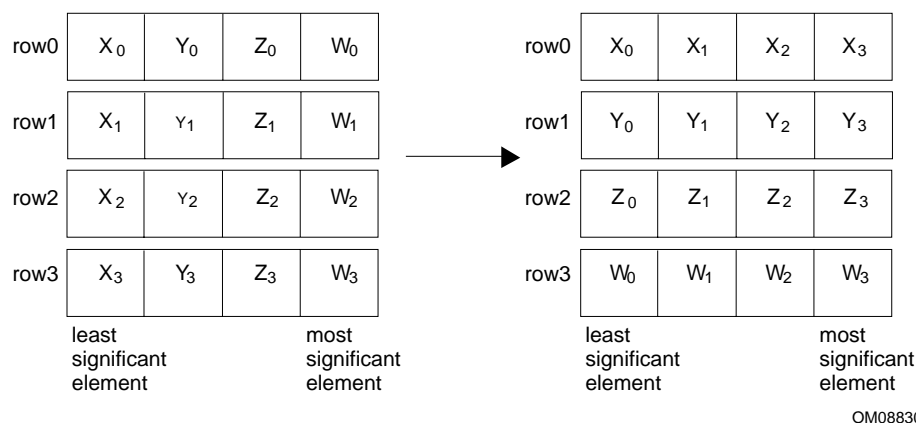
The Streaming SIMD Extensions also provide the following macro function to transpose a 4 by 4 matrix of single precision floating point values.

`_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 [Figure 3-1](#).

Figure 3-1 Matrix Transposition Using the `_MM_TRANSPOSE4_PS` Macro



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 miscellaneous intrinsics are listed in [Table 3-5](#), and are followed by a description of each intrinsic.

Table 3-6 Conversion Operations

Intrinsic Name	Operation	Corresponding Instruction
<code>_mm_load_ss</code>	Load the low value and clear the three high values	MOVSS
<code>_mm_loadl_ps</code>	Load one value into all four words	MOVSS + Shuffling

Table 3-6 Conversion Operations (continued)

Intrinsic Name	Operation	Corresponding Instruction
<code>_mm_load_ps</code>	Load four values, address aligned	MOVAPS
<code>_mm_loadu_ps</code>	Load four values, address unaligned	MOVUPS
<code>_mm_loadr_ps</code>	Load four values, in reverse order	MOVAPS + Shuffling
<code>_mm_set_ss</code>	Set the low value and clear the three high values	Composite
<code>_mm_set1_ps</code>	Set all four words with the same value	Composite
<code>_mm_set_ps</code>	Set four values, address aligned	Composite
<code>_mm_setr_ps</code>	Set four values, in reverse order	Composite
<code>_mm_setzero_ps</code>	Clear all four values	Composite
<code>_mm_store_ss</code>	Store the low value	MOVSS
<code>_mm_store1_ps</code>	Store the low value across all four words	MOVSS + Shuffling
<code>_mm_store_ps</code>	Store four values, address aligned	MOVAPS
<code>_mm_storeu_ps</code>	Store four values, address unaligned	MOVUPS
<code>_mm_storer_ps</code>	Store four values, in reverse order	MOVAPS + Shuffling
<code>_mm_move_ss</code>	Set the low word, and pass in three high values	MOVSS

Load Operations

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

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_load1_ps(float *p) MOVSS + shuffling
```

or

```
__m128 _mm_load_ps1(float *p) MOVSS + shuffling
```

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) MOVAPS
```

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) MOVUPS
```

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)` MOVAPS + shuffling

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

`__m128 _mm_set_ss(float w)` (composite)

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_set1_ps(float w)` (composite)

or

`__m128 _mm_set_ps1(float w)` (composite)

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)` (composite)

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)(composite)
```

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) (composite)
```

Clears the four SP FP values.

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

Store Operations

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

Stores the lower SP FP value.

```
*p := a0
```

```
void _mm_storel_ps(float *p, __m128 a) MOVSS + shuffling
```

or

```
void _mm_store_ps1(float *p, __m128 a) MOVSS + shuffling
```

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) MOVAPS
```

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)           MOVUPS
```

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)           MOVAPS + shuffling
```

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)           MOVSS
```

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
```

Integer Intrinsics Using Streaming SIMD Extensions

The integer intrinsics are listed in [Table 3-7](#), 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.

Table 3-7 Integer Intrinsics

Intrinsic Name	Operation	Corresponding Instruction
<code>_mm_extract_pi16</code>	Extract one of four words	PEXTRW
<code>_mm_insert_pi16</code>	Insert a word	PINSRW
<code>_mm_max_pi16</code>	Compute the maximum	PMAXSW
<code>_mm_max_pu8</code>	Compute the maximum, unsigned	PMAXUB
<code>_mm_min_pi16</code>	Compute the minimum	PMINSW
<code>_mm_min_pu8</code>	Compute the minimum, unsigned	PMINUB
<code>_mm_movemask_pi8</code>	Create an eight-bit mask	PMOVMASKB
<code>_mm_mulhi_pu16</code>	Multiply, return high bits	PMULHUW
<code>_mm_shuffle_pi16</code>	Return a combination of four words	PSHUFW
<code>_mm_maskmove_si64</code>	Conditional Store	MASKMOVQ
<code>_mm_avg_pu8</code>	Compute rounded average	PAVGB
<code>_mm_avg_pu16</code>	Compute rounded average	PAVGW
<code>_mm_sad_pu8</code>	Compute sum of absolute differences	PSADBW

For this section you need to ensure to empty the multimedia state for the mmx register. See [“The EMMS Instruction: Why You Need it and When to Use it”](#).

```
int _mm_extract_pi16(__m64 a, int n)                PEXTRW
```

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 _mm_insert_pi16(__m64 a, int d, int n)        PINSRW
```

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 _mm_max_pi16(__m64 a, __m64 b) PMAWSW
```

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 _mm_max_pu8(__m64 a, __m64 b) PMAWSUB
```

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 _mm_min_pi16(__m64 a, __m64 b) PMINSW
```

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 _mm_min_pu8(__m64 a, __m64 b) PMINSUB
```

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 _mm_movemask_pi8(__m64 a) PMOVMSKB
```

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 _mm_mulhi_pu16(__m64 a, __m64 b) PMULHUW
```

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 _mm_shuffle_pi16(__m64 a, int n) PSHUFW
```

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 _mm_maskmove_si64(__m64 d, __m64 n, char *p) MASKMOVQ
```

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 _mm_avg_pu8(__m64 a, __m64 b) PAVGB
```

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 _mm_avg_pu16(__m64 a, __m64 b) PAVGW
```

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 _mm_sad_pu8(__m64 a, __m64 b) PSADBW
```

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

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

Cacheability Support Using Streaming SIMD Extensions

The following intrinsics provide ways to make efficient use of the cache.

```
void _mm_prefetch(char *p, int i) PREFETCH
```

Loads one cache line of data from address *p* to a location “closer” to the processor. The value *i* 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) 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”](#).

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

Guarantees that every preceding store is globally visible before any subsequent store.

Willamette New Instructions

4

This section describes the C/C++ language-level features supporting the Willamette Extensions in the Intel C/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`).

Willamette Floating Point Intrinsics

The following sections list the floating-point and integer intrinsics broken into groups by the nature of the operation. Each intrinsic entry has an informal pseudo-code and it is followed with a corresponding instruction name in upper case letters; for example, `ADDSD` is the name of the first instruction listed in this section. The variable `r` is generally used for the intrinsic's return value. A number appended to a variable name indicates the element of a packed object. For example, `r0` is the lowest double of `r`. Some intrinsics are “composites” because they require more than one instruction to implement them. For more details, refer to the Willamette New Instructions External Architecture Specification (EAS). You should be familiar with the hardware features provided by the Willamette New Instructions 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.

Arithmetic Operations

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

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)` ADDPD

Adds the two DP FP values of *a* and *b*.

`r0 := a0 + b0`

`r1 := a1 + b1`

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

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)` DIVPD

Divides the two DP FP values of *a* and *b*.

`r0 := a0 / b0`

`r1 := a1 / b1`

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

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)          MAXPD
```

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

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

```
r1 := max(a1, b1)
```

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

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)          MINPD
```

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

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

```
r1 := min(a1, b1)
```

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

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)          MULPD
```

Multiplies the two DP FP values of *a* and *b*.

```
r0 := a0 * b0
```

```
r1 := a1 * b1
```

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

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) SQRTPD
```

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

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

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

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) SUBPD
```

Subtracts the two DP FP values of *b* from *a*.

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

Logical Operations

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

Computes the bitwise AND of the two DP FP values of *a* and *b*.

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

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

Computes the bitwise XOR of the two DP FP values of *a* and *b*.

```
r0 := a0 ^ b0
```

```
r1 := a1 ^ b1
```

Comparisons

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 name are reversed in the actual implementation.

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

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)        CMPLTPD
```

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)        CMPLDPD
```

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)        CMPLTPDr
```

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)          CMPLEPDr
```

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)          CMPORDPD
```

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)        CMPUNORDPD
```

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)          CMPNEQPD
```

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)          CMPNLTPD
```

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)          CMPNLEPD
```

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)    CMPNLTPDr
```

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)    CMPNLEPDr
```

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)    CMPEQSD
```

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)    CMPLTSD
```

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 := a1
```

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

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)    CMPLTSDr
```

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)          CMPLESDr
```

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)          CMPORDSD
```

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)        CMPUNORDSD
```

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)          CMPNEQSD
```

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)          CMPNLTSD
```

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)    CMPNLESD
```

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)    CMPNLTSDr
```

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)    CMPNLESDr
```

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)        COMISD
```

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)        COMISD
```

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)        COMISD
```

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)        COMISD
```

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)          COMISD
```

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)         COMISD
```

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)         UCOMISD
```

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)         UCOMISD
```

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)         UCOMISD
```

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)         UCOMISD
```

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)          UCOMISD
```

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)         UCOMISD
```

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

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`, `_mm_cvttss_sd_si32`, and `_mm_cvttss_epi32` intrinsics use the truncate rounding mode regardless of the mode specified by the `MXCSR` register.

```
__m128 _mm_cvtpd_ps (__m128d a)                   CVTPD2PS
```

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)                   CVTPS2PD
```

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)               CVTDQ2PD
```

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) CVTPD2DQ
```

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) CVTSD2SI
```

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) CVTSD2SS
```

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) CVTSI2SD
```

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) CVTSS2SD
```

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) CVTTPD2DQ
```

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_cvttssd_si32 (__m128d a) CVTTSSD2SI

```

Converts the lower DP FP value of *a* to a 32 bit signed integer using truncate.

```

r := (int) a0

```

```

__m128 _mm_cvtepi32_ps (__m128i a) CVTDQ2PS

```

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) CVTPS2DQ

```



CAUTION. *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) CVTTPS2DQ

```

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

```

```
__m64 _mm_cvtpd_pi32 (__m128d a) CVTPD2PI
```

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) CVTTPD2PI
```

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) CVTPI2PD
```

Converts the two 32-bit signed integer values of *a* to DP FP values.

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

Miscellaneous Operations

```
__m128d _mm_unpackhi_pd (__m128d a, __m128d b) UNPCKHPD
```

Interleaves the upper DP FP values of *a* and *b*.

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

```
__m128d _mm_unpacklo_pd (__m128d a, __m128d b) UNPCKLPD
```

Interleaves the lower DP FP values of *a* and *b*.

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

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


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”](#) in the end of this section for a description of the shuffle semantics.

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

Willamette 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.

Load Operations

```
__m128d _mm_load_pd (double *p) 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 *p) (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 *p) (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 *p) 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 *p) 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 *p) 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 *p) 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

```
__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 () XORPD
```

Sets the 2 DP FP values to zero.

```
r0 := 0.0
r1 := 0.0
```

```
__m128d _mm_move_sd (__m128d a, __m128d b) 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

```
void _mm_store_sd (double *p, __m128d a) MOVSD
```

Stores the lower DP FP value of *a*. The address *p* need not be 16-byte aligned.

```
*p := a0
```

```
void _mm_storel_pd (double *p, __m128d a) (MOVAPD + shuffling)
```

Stores the lower DP FP value of *a* twice. The address *p* must be 16-byte aligned.

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

```
void _mm_store_pd (double *p, __m128d a)          MOVAPD
```

Stores two DP FP values. The address *p* must be 16-byte aligned.

```
p[0] := a0
```

```
p[1] := a1
```

```
void _mm_storeu_pd (double *p, __m128d a)          MOVUPD
```

Stores two DP FP values. The address *p* need not be 16 byte aligned.

```
p[0] := a0
```

```
p[1] := a1
```

```
void _mm_storer_pd (double *p, __m128d a)          (MOVAPD + shuffling)
```

Stores two DP FP values in reverse order. The address *p* must be 16 byte aligned.

```
p[0] := a1
```

```
p[1] := a0
```

```
void _mm_storeh_pd (double *p, __m128d a)          MOVHPD
```

Stores the upper DP FP value of *a*.

```
*p := a1
```

```
void _mm_storel_pd (double *p, __m128d a)          MOVLPD
```

Stores the lower DP FP value of *a*.

```
*p := a0
```

Willamette Integer Intrinsics

Arithmetic Operations

The packed arithmetic intrinsics are listed in [Table 3-1](#), and are followed by a description of each intrinsic.

```
__m128i _mm_add_epi8 (__m128i a, __m128i b)    PADDB
```

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
```

```
__m128i _mm_add_epi16 (__m128i a, __m128i b)    PADDW
```

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)    PADDD
```

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)          PADDQ
```

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)    PADDQ
```

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)    PADDSB
```

Adds the 16 signed 8-bit integers in *a* to the 16 signed 8-bit integers in *b* and saturates.

```
r0 := SignedSaturate(a0 + b0)
```

```
r1 := SignedSaturate(a1 + b1)
```

```
...
```

```
r15 := SignedSaturate(a15 + b15)
```

```
__m128i _mm_adds_epi16 (__m128i a, __m128i b)    PADDSW
```

Adds the 8 signed 16-bit integers in *a* to the 8 signed 16-bit integers in *b* and saturates.

```
r0 := SignedSaturate(a0 + b0)
```

```
r1 := SignedSaturate(a1 + b1)
```

```
...
```

```
r7 := SignedSaturate(a7 + b7)
```

```
__m128i _mm_adds_epu8 (__m128i a, __m128i b)    PADDUSB
```

Adds the 16 unsigned 8-bit integers in *a* to the 16 unsigned 8-bit integers in *b* and saturates.

```
r0 := UnsignedSaturate(a0 + b0)
```

```
r1 := UnsignedSaturate(a1 + b1)
```

```
...
```

```
r15 := UnsignedSaturate(a15 + b15)
```

```
__m128i _mm_adds_epu16 (__m128i a, __m128i b)    PADDUSW
```

Adds the 8 unsigned 16-bit integers in *a* to the 8 unsigned 16-bit integers in *b* and saturates.

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

```
__m128i _mm_avg_epu8 (__m128i a, __m128i b)      PAVGB
```

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)     PAVGW
```

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)    PMADDWD
```

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)    PMAXSW
```

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)    PMAXUB
```

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)    PMINSW
```

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)    PMINUB
```

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)    PMULHW
```

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)    PMULHUW
```

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)    PMULLW
```

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)             PMULUDQ
```

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)    PMULUDQ
```

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)    PSADBW
```

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)    PSUBB
```

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)    PSUBW
```

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)    PSUBD
```

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)          PSUBQ
```

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)    PSUBQ
```

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)    PSUBSB
```

Subtracts the 16 signed 8-bit integers of *b* from the 16 signed 8-bit integers of *a* and saturates.

```
r0 := SignedSaturate(a0 - b0)
```

```
r1 := SignedSaturate(a1 - b1)
```

```
...
```

```
r15 := SignedSaturate(a15 - b15)
```

```
__m128i _mm_subs_epi16 (__m128i a, __m128i b)    PSUBSW
```

Subtracts the 8 signed 16-bit integers of *b* from the 8 signed 16-bit integers of *a* and saturates.

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

```
__m128i _mm_subs_epu8 (__m128i a, __m128i b)    PSUBUSB
```

Subtracts the 16 unsigned 8-bit integers of *b* from the 16 unsigned 8-bit integers of *a* and saturates.

```
r0 := UnsignedSaturate(a0 - b0)
r1 := UnsignedSaturate(a1 - b1)
...
r15 := UnsignedSaturate(a15 - b15)
```

```
__m128i _mm_subs_epu16 (__m128i a, __m128i b)   PSUBUSW
```

Subtracts the 8 unsigned 16-bit integers of *b* from the 8 unsigned 16-bit integers of *a* and saturates.

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

Logical Operations

```
__m128i _mm_and_si128 (__m128i a, __m128i b)    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) 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)    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)    PXOR
```

Computes the bitwise XOR of the 128-bit value in *a* and the 128-bit value in *b*.

```
r := a ^ b
```

Shift Operations

```
__m128i _mm_slli_si128 (__m128i a, int imm)    PSLLDQ
```

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)    PSLLW
```

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) PSLLW
```

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)    PSLLD
```

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) PSLLD
```

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)    PSLLQ
```

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) PSLLQ
```

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)    PSRAW
```

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) PSRAW
```

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)    PSRAD
```

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) PSRAD
```

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_srli_si128 (__m128i a, int imm)     PSRLDQ
```

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)    PSRLW
```

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) PSRLW
```

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)    PSRLD
```

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) PSRLD
```

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)    PSRLQ
```

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) PSRLQ
```

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)
```

Comparisons

```
__m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)    PCMPEQB
```

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)    PCMPEQW
```

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)   PCMP EQD
```

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)   PCMP GTB
```

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)  PCMP GTW
```

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)  PCMP GTD
```

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)    PCMPGTBr
```

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)    PCMPGTWr
```

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)    PCMPGTDr
```

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
```

Conversions

```
__m128i _mm_cvtsi32_si128 (int a)                MOVD
```

Moves 32-bit integer *a* to the least significant 32 bits of an `__m128` object one extending the upper bits.

```
r0 := a
r1 := 0x0 ; r2 := 0x0 ; r3 := 0x0
```

```
int __mm_cvtsi128_si32 (__m128i a)          MOVD
```

Moves the least significant 32 bits of *a* to a 32 bit integer.

```
r := a0
```

Miscellaneous Operations

```
__m128i __mm_packs_epi16 (__m128i a, __m128i b)  PACKSSWB
```

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)  PACKSSDW
```

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) PACKUSWB
```

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) PEXTRW
```

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) PINSRW
```

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) PMOVMASKB
```

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) PSHUFD
```

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” at the end of this section for a description of shuffle semantics.

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

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” at the end of this section for a description of shuffle semantics.

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

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” at the end of this section for a description of shuffle semantics.

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

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) PUNPCKHWD
```

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) PUNPCKHDQ
```

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) PUNPCKHQDQ
```

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) PUNPCKLBW
```

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) PUNPCKLWD
```

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) PUNPCKLDQ
```

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) PUNPCKLQDQ
```

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
```

Macro Function for Shuffle

The Willamette New Instructions 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 [Example 4-1](#).

Example 4-1 Shuffle Function Macro

```
_MM_SHUFFLE2(x, y)
expands to the value of
(x<<1) | y
```

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

Example 4-2 View of Original and Result Words With Shuffle Function Macro

```

; m1 = 127  [ a | b ] 0
; m2 = 127  [ c | d ] 0
m3 = _mm_shuffle_pd(m1, m2, _MM_SHUFFLE2(1,0))
; m3 = 127  [ c | b ] 0

```

Willamette Integer Memory and Initialization**Load Operations**

```
__m128i _mm_load_si128 (__m128i *p)           MOVDQA
```

Loads 128-bit value. Address *p* must be 16-byte aligned.

```
r := *p
```

```
__m128i _mm_loadu_si128 (__m128i *p)         MOVDQU
```

Loads 128-bit value. Address *p* need not be 16-byte aligned.

```
r := *p
```

Set Operations

```
__m128i _mm_set_epi64 (__m64 q1, __m64 q0)    (composite)
```

Sets the 2 64-bit integer values.

```
r0 := q0
```

```
r1 := q1
```

```
__m128i _mm_set_epi32 (int i3, int i2,          (composite)
                      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,      (composite)
                      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,      (composite)
                     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)` (composite)

Sets the 2 64-bit integer values to *q*.

`r0 := q`

`r1 := q`

`__m128i _mm_set1_epi32 (int i)` (composite)

Sets the 4 signed 32-bit integer values to *i*.

`r0 := i`

`r1 := i`

`r2 := i`

`r3 := i`

`__m128i _mm_set1_epi16 (short w)` (composite)

Sets the 8 signed 16-bit integer values to *w*.

`r0 := w`

`r1 := w`

`...`

`r7 := w`

`__m128i _mm_set1_epi8 (char b)` (composite)

Sets the 16 signed 8-bit integer values to *b*.

`r0 := b`

`r1 := b`

`...`

`r15 := b`

`__m128i _mm_setr_epi64 (__m64 q0, __m64 q1)` (composite)

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)` (composite)

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,      (composite)
                        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 b0, char b1,      (composite)
                       char b2, char b3,
                       char b4, char b5,
                       char b6, char b7,
                       char b8, char b9,
                       char b10, char b11,
                       char b12, char b13,
                       char b14, char b15)
```

Sets the 16 signed 8-bit integer values in reverse order.

```
r0 := b0
r1 := b1
...
r15 := b15
```

```
__m128i _mm_setzero_si128 ()                PXOR
```

Sets the 128-bit value to zero.

```
r := 0x0
```

Store Operations

```
void _mm_store_si128 (__m128i *p, __m128i a)    MOVDQA
```

Stores 128-bit value. Address *p* must be 16 byte aligned.

```
*p := a
```

```
void _mm_storeu_si128 (__m128i *p, __m128i a)    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) 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
```

Data Alignment, Assembly and Processor Dispatch Support

5

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

- Alignment Support
- Processor Dispatch Support
- Assembly Language Support

Alignment Support

To improve performance, you should align data to 16 bytes in memory operations when you are using the Streaming SIMD Extensions. 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. 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:

- 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.



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.) However, you cannot adjust the alignment of a parameter, nor a field of a `struct` or `class`. However, you can 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.lib` 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.

Processor Dispatch Support

With `__declspec(cpu_specific)` and `__declspec(cpu_dispatch)`, you can direct the compiler to generate code that automatically determines the processor on which it is running, and selects the appropriate implementation of a function. This enables you, for example, to write code that takes advantage of MMX instructions when executing on a processor that has them, but that also executes correctly on older processors that do not have the MMX instructions. The syntax of these extended-attributes is as follows:

```
cpu_specific(cpuid)
```



```
cpu_dispatch(cpuid-list)
```

where *cpuid* is one of:

generic

pentium

pentium_pro

pentium_mmx

pentium_ii

pentium_iii

pentium_iii_no_xmm_regs

and *cpuid-list* is one of:

cpuid

cpuid-list , *cpuid*

The names of the *cpuid* are not case sensitive. You can specify these attributes only for function definitions. The body of a function declared with `__declspec(cpu_dispatch)` must be empty, and is referred to as a stub.

If a function *f* is defined as `__declspec(cpu_specific(p))`, then a `cpu_dispatch` stub must appear for *f* somewhere in the program, and *p* must be in the *cpuid-list* of that stub; otherwise, that `cpu_specific` definition can never be called and no error will be reported for this condition.

If the `cpu_dispatch` stub for a function *f* contains the *cpuid* *p*, then a `cpu_specific` definition of *f* with *cpuid* *p* must appear somewhere in the program; otherwise an unresolved external error is reported. A `cpu_specific` function definition need not appear in the same translation unit as the corresponding `cpu_dispatch` stub, unless the `cpu_specific` function is declared `static`. The `inline` attribute is disabled for all `cpu_specific` and `cpu_dispatch` functions.

When a `cpu_dispatch` stub is compiled, its body is filled in with code that determines the processor on which the program is running, then dispatches to the “best” `cpu_specific` implementation available (as defined by the *cpuid-list*) that will run on that processor. When a `cpu_specific` function is compiled, optimizations and code generation options appropriate to the specified processor are used regardless of command-line option settings.

Here is an example of how these features can be used:

```
#include <mmintrin.h>

/* array_sum(r, a, b, l) adds two arrays of unsigned short (a and b), of
length l, and stores the result in r. */

__declspec(cpu_specific(Pentium))
void array_sum(unsigned short *result,
unsigned short const *a,
unsigned short const *b,
size_t length)
{
/* The implementation specific to the Pentium processor uses no
special architectural features. */

for (; length > 0; length--)
*result++ = *a++ + *b++;
}

__declspec(cpu_specific(Pentium_MMX))
void array_sum(unsigned short *result,
unsigned short const *a,
unsigned short const *b,
size_t length)
{
/* The implementation for a Pentium processor with MMX technology uses
uses an MMX instruction intrinsic to add four elements at a time, to
reduce the number of loop iterations by 3/4. */

__m64 *mmx_result = (__m64 *)result;
__m64 const *mmx_a = (__m64 const *)a;
__m64 const *mmx_b = (__m64 const *)b;
for (; length > 3; length -= 4)
*mmx_result++ = _m_paddw(*mmx_a++, *mmx_b++);
```

```
/* If the size of all arrays passed to this routine is known to be a
multiple of four, the following code (which takes care of excess
elements) is not necessary. */

result = (unsigned short *)mmx_result;
a = (unsigned short const *)mmx_a;
b = (unsigned short const *)mmx_b;
for (; length > 0; length--)
    *result++ = *a++ + *b++;
}

__declspec(cpu_dispatch(Pentium, Pentium_MMX))
void array_sum(unsigned short *result,
unsigned short const *a,
unsigned short const *b,
size_t length)
{
    /* An empty function body, which informs the compiler that it should
    generate a dispatch function for the CPU-specific implementations
    listed in the cpu_dispatch clause. */
}
```

Dynamic Stack Frame Alignment

By default, the compiler targets functions that use 8- or 16-byte objects for dynamic alignment. It is up to the discretion of the compiler to decide whether dynamic stack alignment is needed on a function-per-function level (for stability of performance). This ensures that references to `double` and `__m64` to `double`, `__m64`, and `__m128` local variables align to addresses evenly divisible by 16. Parameters alignment is not affected.

[Table 5-1](#) shows how `-Qsalign` controls the stack frame alignment.

Table 5-1 Stack Frame Alignment Options

Option	Meaning
<code>-Qsalign8</code>	Align stack for frames or functions with variables of 8 or 16 bytes where it is deemed appropriate by the compiler.
<code>-Qsalign16</code>	Align stack for frames or functions with 16-byte variables where it is deemed appropriate by the compiler.
<code>-Qsalign-</code>	Disable stack alignment for all functions.



NOTE. *If you are using aligned frames, you should not modify the `EBX` register in inlined assembly blocks because `EBX` is used to keep track of the argument block. You can modify `EBX` only if you save and restore `EBX` each time you use it. The Intel C/C++ Compiler uses the `EBX` register to control alignment of variables of these data types so using `EBX`, without preserving it, will cause unexpected program execution.*

For additional information on the use of `EBX` in inline assembly code and other related issues, see the Application Note AP-833 *Data Alignment and Programming Issues with the Intel C/C++ Compiler* (Order #243872-001), and AP-589 *Software Conventions for the Streaming SIMD Extensions* (Order # 243873-001).

Assembly Language Support

The Intel C/C++ Compiler supports inlining of assembly blocks and generation of assembly files.

Inline Assembly

The compiler supports use of all the MMX and Streaming SIMD Extensions in inline assembly (`__asm`) blocks. The compiler also accepts the new syntax `MMWORD PTR` and `XMMWORD PTR` to refer to 64- and 128-bit data.

Generation of Assembly files

Use the `-s` compiler switch to produce an assembly listing. This is useful for hand-tuning compiler-generated code. The assembly files can be directly compiled using MASM version 6.14 with `ia_emm.inc` MASM include files. Generated `.asm` files contain an include directive for them when the MMX technology intrinsics or Streaming SIMD Extensions are used. For details on using inline assembly, see “Producing an Assembly Code Listing” in the *Intel C/C++ Compiler User’s Guide*.