Intel® C++ Intrinsic Reference

Document Number: 312482-003US

Disclaimer and Legal Information

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.
UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

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 information here is subject to change without notice. Do not finalize a design with this information.
The products described in this document 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.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.
Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

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

Table Of Contents

Overview: Intrinsics Reference

Intrinsics are assembly-coded functions that allow you to use C++ function calls and variables in place of assembly instructions.

Intrinsics are expanded inline eliminating function call overhead. Providing the same benefit as using inline assembly, intrinsics improve code readability, assist instruction scheduling, and help reduce debugging.

Intrinsics provide access to instructions that cannot be generated using the standard constructs of the C and C++ languages.

Intrinsics for Intel® C++ Compilers

The Intel® C++ Compiler enables easy implementation of assembly instructions through the use of intrinsics. Intrinsics are provided for Intel® Streaming SIMD Extensions 4 (SSE4), Supplemental Streaming SIMD Extensions 3 (SSSE3), Streaming SIMD Extensions 3 (SSE3), Streaming SIMD Extensions 2 (SSE2), and Streaming SIMD Extensions (SSE) instructions. The Intel C++ Compiler for IA-64 architecture also provides architecture-specific intrinsics.

The Intel C++ Compiler provides intrinsics that work on specific architectures and intrinsics that work across IA-32, Intel® 64, and IA-64 architectures. Most intrinsics map directly to a corresponding assembly instruction, some map to several assembly instructions.

The Intel C++ Compiler also supports Microsoft* Visual Studio 2005 intrinsics (for x86 and x64 architectures) to generate instructions on Intel processors based on IA-32 and Intel® 64 architectures. For more information on these Microsoft* intrinsics, visit http://msdn2.microsoft.com/en-us/library/26td21ds.aspx.

Availability of Intrinsics on Intel Processors

Not all Intel processors support all intrinsics. For information on which intrinsics are supported on Intel processors, visit http://processorfinder.intel.com. The Processor Spec Finder tool links directly to all processor documentation and the data sheets list the features, including intrinsics, supported by each processor.

Details about Intrinsics

The MMX(TM) technology and Streaming SIMD Extension (SSE) instructions use the following features:

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

Registers

Intel processors provide special register sets.

The MMX instructions use eight 64-bit registers (`mm0` to `mm7`) which are aliased on the floating-point stack registers.

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

Because each of these registers can hold more than one data element, the processor can process more than one data element simultaneously. This processing capability is also known as single-instruction multiple data processing (SIMD).

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


Note

The `MM` and `XMM` registers are the SIMD registers used by the IA-32 platforms to implement MMX technology and SSE or SSE2 intrinsics. On the IA-64 architecture, the MMX and SSE intrinsics use the 64-bit general registers and the 64-bit significand of the 80-bit floating-point register.

Data Types

Intrinsic functions use four new C data types as operands, representing the new registers that are used as the operands to these intrinsic functions.

New Data Types Available

The following table details for which instructions each of the new data types are available.

New Data Type	MMX(TM) Technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Streaming SIMD Extensions 3
`__m64`	Available	Available	Available	Available
`__m128`	Not available	Available	Available	Available

__m128d	Not available	Not available	Available	Available
__m128i	Not available	Not available	Available	Available

__m64 Data Type

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

__m128 Data Types

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

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

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

The compiler aligns __m128d and _m128i local and global data to 16-byte boundaries on the stack. To align integer, float, or double arrays, you can use the declspec align statement.

Data Types Usage Guidelines

These data types are not basic ANSI C data types. You must observe the following usage restrictions:

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

Accessing __m128i Data

To access 8-bit data:

```
#define _mm_extract_epi8(x, imm) \

    ((((imm) & 0x1) == 0) ?      \

         _mm_extract_epi16((x), (imm) >> 1) & 0xff : \

         _mm_extract_epi16(_mm_srli_epi16((x), 8), (imm) >> 1))
```

For 16-bit data, use the following intrinsic:

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

To access 32-bit data:

```
#define _mm_extract_epi32(x, imm) \
    _mm_cvtsi128_si32(_mm_srli_si128((x), 4 * (imm)))
```

To access 64-bit data (Intel® 64 architecture only):

```
#define _mm_extract_epi64(x, imm) \
    _mm_cvtsi128_si64(_mm_srli_si128((x), 8 * (imm)))
```

Naming and Usage Syntax

Most intrinsic names use the following notational convention:

```
_mm_<intrin_op>_<suffix>
```

The following table explains each item in the syntax.

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

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

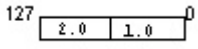
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
double a[2] = {1.0, 2.0};

__m128d t = _mm_load_pd(a);
```

The result is the same as either of the following:

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

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

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



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

References

See the following publications and internet locations for more information about intrinsics and the Intel architectures that support them. You can find all publications on the Intel website.

Internet Location or Publication	Description
developer.intel.com	Technical resource center for hardware designers and developers; contains links to product pages and documentation.
Intel® Itanium® Architecture Software Developer's Manuals, Volume 3: Instruction Set Reference	Contains information and details about Itanium instructions.
IA-32 Intel® Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M	Describes the format of the instruction set of IA-32 Intel Architecture and covers the reference pages of instructions from A to M
IA-32 Intel® Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z	Describes the format of the instruction set of IA-32 Intel Architecture and covers the reference pages of instructions from N to Z
Intel® Itanium® 2 processor website	Intel website for the Itanium 2 processor; select the "Documentation" tab for documentation.

Intrinsics for Use across All IA

Overview: Intrinsics for All IA

The intrinsics in this section function across all IA-32 and IA-64-based platforms. They are offered as a convenience to the programmer. They are grouped as follows:

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

Integer Arithmetic Intrinsics

The following table lists and describes integer arithmetic intrinsics that you can use across all Intel architectures.

Intrinsic	Description
`int abs(int)`	Returns the absolute value of an integer.
`long labs(long)`	Returns the absolute value of a long integer.
`unsigned long _lrotl(unsigned long value, int shift)`	Implements 64-bit left rotate of value by shift positions.
`unsigned long _lrotr(unsigned long value, int shift)`	Implements 64-bit right rotate of value by shift positions.
`unsigned int _rotl(unsigned int value, int shift)`	Implements 32-bit left rotate of value by shift positions.
`unsigned int _rotr(unsigned int value, int shift)`	Implements 32-bit right rotate of value by shift positions.
`unsigned short _rotwl(unsigned short val, int shift)`	Implements 16-bit left rotate of value by shift positions. These intrinsics are not supported on IA-64 platforms.
`unsigned short _rotwr(unsigned short val, int shift)`	Implements 16-bit right rotate of value by shift positions. These intrinsics are not supported on IA-64 platforms.

Note

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

Floating-point Intrinsics

The following table lists and describes floating point intrinsics that you can use across all Intel architectures.

Intrinsic	Description
`double fabs(double)`	Returns the absolute value of a floating-point value.
`double log(double)`	Returns the natural logarithm ln(x), x>0, with double precision.
`float logf(float)`	Returns the natural logarithm ln(x), x>0, with single precision.
`double log10(double)`	Returns the base 10 logarithm log10(x), x>0, with double precision.
`float log10f(float)`	Returns the base 10 logarithm log10(x), x>0, with single precision.
`double exp(double)`	Returns the exponential function with double precision.
`float expf(float)`	Returns the exponential function with single precision.
`double pow(double, double)`	Returns the value of x to the power y with double precision.
`float powf(float, float)`	Returns the value of x to the power y with single precision.
`double sin(double)`	Returns the sine of x with double precision.
`float sinf(float)`	Returns the sine of x with single precision.
`double cos(double)`	Returns the cosine of x with double precision.
`float cosf(float)`	Returns the cosine of x with single precision.
`double tan(double)`	Returns the tangent of x with double precision.
`float tanf(float)`	Returns the tangent of x with single precision.
`double acos(double)`	Returns the inverse cosine of x with double precision
`float acosf(float)`	Returns the inverse cosine of x with single precision

`double acosh(double)`	Compute the inverse hyperbolic cosine of the argument with double precision.
`float acoshf(float)`	Compute the inverse hyperbolic cosine of the argument with single precision.
`double asin(double)`	Compute inverse sine of the argument with double precision.
`float asinf(float)`	Compute inverse sine of the argument with single precision.
`double asinh(double)`	Compute inverse hyperbolic sine of the argument with double precision.
`float asinhf(float)`	Compute inverse hyperbolic sine of the argument with single precision.
`double atan(double)`	Compute inverse tangent of the argument with double precision.
`float atanf(float)`	Compute inverse tangent of the argument with single precision.
`double atanh(double)`	Compute inverse hyperbolic tangent of the argument with double precision.
`float atanhf(float)`	Compute inverse hyperbolic tangent of the argument with single precision.
`double cabs(double complex z)`	Computes absolute value of complex number. The intrinsic argument `complex` is a complex number made up of two double precision elements, one real and one imaginary. The input parameter `z` is made up of two values of double precision type passed together as a single argument.
`float cabsf(float complex z)`	Computes absolute value of complex number. The intrinsic argument `complex` is a complex number made up of two single precision elements, one real and one imaginary. The input parameter `z` is made up of two values of single precision type passed together as a single argument.
`double ceil(double)`	Computes smallest integral value of double precision argument not less than the argument.
`float ceilf(float)`	Computes smallest integral value of single precision argument not less than the argument.
`double cosh(double)`	Computes the hyperbolic cosine of double precision argument.

`float coshf(float)`	Computes the hyperbolic cosine of single precision argument.
`float fabsf(float)`	Computes absolute value of single precision argument.
`double floor(double)`	Computes the largest integral value of the double precision argument not greater than the argument.
`float floorf(float)`	Computes the largest integral value of the single precision argument not greater than the argument.
`double fmod(double)`	Computes the floating-point remainder of the division of the first argument by the second argument with double precision.
`float fmodf(float)`	Computes the floating-point remainder of the division of the first argument by the second argument with single precision.
`double hypot(double, double)`	Computes the length of the hypotenuse of a right angled triangle with double precision.
`float hypotf(float, float)`	Computes the length of the hypotenuse of a right angled triangle with single precision.
`double rint(double)`	Computes the integral value represented as double using the IEEE rounding mode.
`float rintf(float)`	Computes the integral value represented with single precision using the IEEE rounding mode.
`double sinh(double)`	Computes the hyperbolic sine of the double precision argument.
`float sinhf(float)`	Computes the hyperbolic sine of the single precision argument.
`float sqrtf(float)`	Computes the square root of the single precision argument.
`double tanh(double)`	Computes the hyperbolic tangent of the double precision argument.
`float tanhf(float)`	Computes the hyperbolic tangent of the single precision argument.

String and Block Copy Intrinsics

The following table lists and describes string and block copy intrinsics that you can use across all Intel architectures.

The string and block copy intrinsics are not implemented as intrinsics on IA-64 architecture.

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

Miscellaneous Intrinsics

The following table lists and describes intrinsics that you can use across all Intel architectures, except where noted.

Intrinsic	Description
`_abnormal_termination(void)`	Can be invoked only by termination handlers. Returns `TRUE` if the termination handler is

	invoked as a result of a premature exit of the corresponding try-finally region.
__cpuid	Queries the processor for information about processor type and supported features. The Intel® C++ Compiler supports the Microsoft* implementation of this intrinsic. See the Microsoft documentation for details.
void *_alloca(int)	Allocates memory in the local stack frame. The memory is automatically freed upon return from the function.
int _bit_scan_forward(int x)	Returns the bit index of the least significant set bit of x. If x is 0, the result is undefined.
int _bit_scan_reverse(int)	Returns the bit index of the most significant set bit of x. If x is 0, the result is undefined.
int _bswap(int)	Reverses the byte order of x. Bits 0-7 are swapped with bits 24-31, and bits 8-15 are swapped with bits 16-23.
_exception_code(void)	Returns the exception code.
_exception_info(void)	Returns the exception information.
void _enable(void)	Enables the interrupt.
void _disable(void)	Disables the interrupt.
int _in_byte(int)	Intrinsic that maps to the IA-32 instruction IN. Transfer data byte from port specified by argument.
int _in_dword(int)	Intrinsic that maps to the IA-32 instruction IN. Transfer double word from port specified by argument.
int _in_word(int)	Intrinsic that maps to the IA-32 instruction IN. Transfer word from port specified by argument.
int _inp(int)	Same as _in_byte
int _inpd(int)	Same as _in_dword
int _inpw(int)	Same as _in_word
int _out_byte(int, int)	Intrinsic that maps to the IA-32 instruction OUT. Transfer data byte in second argument to port specified by first argument.
int _out_dword(int, int)	Intrinsic that maps to the IA-32 instruction OUT. Transfer double word in second argument to port specified by first argument.
int _out_word(int, int)	Intrinsic that maps to the IA-32 instruction OUT. Transfer word in second argument to port

	specified by first argument.
`int _outp(int, int)`	Same as `_out_byte`
`int _outpd(int, int)`	Same as `_out_dword`
`int _outpw(int, int)`	Same as `_out_word`
`int _popcnt32(int x)`	Returns the number of set bits in x.
`__int64 _rdtsc(void)`	Returns the current value of the processor's 64-bit time stamp counter. This intrinsic is not implemented on systems based on IA-64 architecture. See Time Stamp for an example of using this intrinsic.
`__int64 _rdpmc(int p)`	Returns the current value of the 40-bit performance monitoring counter specified by p.
`int _setjmp(jmp_buf)`	A fast version of `setjmp()`, which bypasses the termination handling. Saves the callee-save registers, stack pointer and return address. This intrinsic is not implemented on systems based on IA-64 architecture.

MMX(TM) Technology Intrinsics

Overview: MMX(TM) Technology Intrinsics

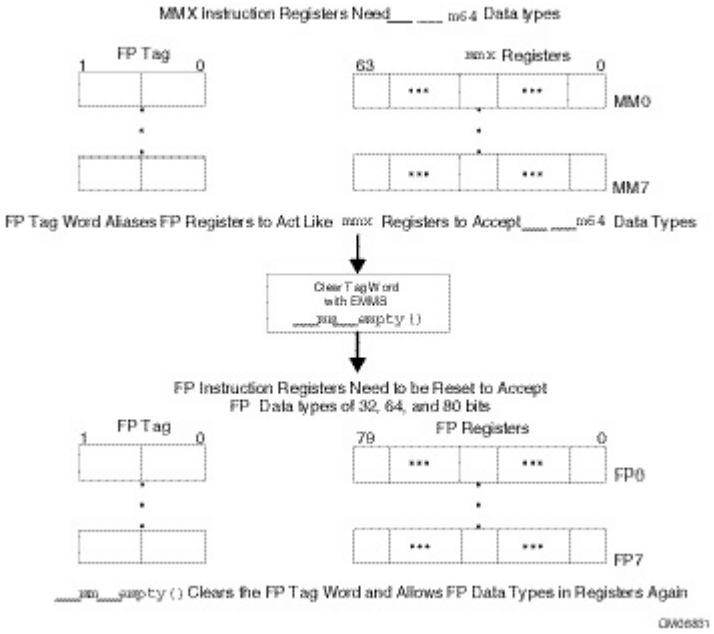
MMX™ technology is an extension to the Intel architecture (IA) instruction set. The MMX instruction set adds 57 opcodes and a 64-bit quadword data type, and eight 64-bit registers. Each of the eight registers can be directly addressed using the register names `mm0` to `mm7`.

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

The EMMS Instruction: Why You Need It

Using EMMS is like emptying a container to accommodate new content. The EMMS instruction clears the MMX™ registers and sets the value of the floating-point tag word to empty. Because floating-point convention specifies that the floating-point stack be cleared after use, you should clear the MMX registers before issuing a floating-point instruction. You should insert the EMMS instruction at the end of all MMX code segments to avoid a floating-point overflow exception.

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



Caution

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

EMMS Usage Guidelines

Here are guidelines for when to use the EMMS instruction:

- Use _mm_empty() after an MMX™ instruction if the next instruction is a floating-point (FP) instruction. For example, you should use the EMMS instruction before performing calculations on float, double or long double. You must be aware of all situations in which your code generates an MMX instruction:
  - when using an MMX technology intrinsic
  - when using Streaming SIMD Extension integer intrinsics that use the __m64 data type
  - when referencing an __m64 data type variable
  - when using an MMX instruction through inline assembly
- Use different functions for operations that use floating point instructions and those that use MMX instructions. This action eliminates the need to empty the multimedia state within the body of a critical loop.
- Use _mm_empty() during runtime initialization of __m64 and FP data types. This ensures resetting the register between data type transitions.
- Do not use _mm_empty() before an MMX instruction, since using _mm_empty() before an MMX instruction incurs an operation with no benefit (no-op).
- Do not use on systems based on IA-64 architecture. There are no special registers (or overlay) for the MMX(TM) instructions or Streaming SIMD Extensions on systems based on IA-64 architecture even though the intrinsics are supported.
- See the Correct Usage and Incorrect Usage coding examples in the following table.

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

MMX(TM) Technology General Support Intrinsics

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

Details about each intrinsic follows the table below.

Intrinsic Name	Operation	Corresponding MMX Instruction
_mm_empty	Empty MM state	EMMS
_mm_cvtsi32_si64	Convert from int	MOVD
_mm_cvtsi64_si32	Convert to int	MOVD
_mm_cvtsi64_m64	Convert from __int64	MOVQ
_mm_cvtm64_si64	Convert to __int64	MOVQ
_mm_packs_pi16	Pack	PACKSSWB
_mm_packs_pi32	Pack	PACKSSDW

_mm_packs_pu16	Pack	PACKUSWB
_mm_unpackhi_pi8	Interleave	PUNPCKHBW
_mm_unpackhi_pi16	Interleave	PUNPCKHWD
_mm_unpackhi_pi32	Interleave	PUNPCKHDQ
_mm_unpacklo_pi8	Interleave	PUNPCKLBW
_mm_unpacklo_pi16	Interleave	PUNPCKLWD
_mm_unpacklo_pi32	Interleave	PUNPCKLDQ

```
void _mm_empty(void)
```

Empty the multimedia state.

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

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)
```

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

```
__m64 _mm_cvtsi64_m64(__int64 i)
```

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

```
__int64 _mm_cvtm64_si64(__m64 m)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

MMX(TM) Technology Packed Arithmetic Intrinsics

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

Details about each intrinsic follows the table below.

Intrinsic Name	Operation	Corresponding MMX Instruction
_mm_add_pi8	Addition	PADDB
_mm_add_pi16	Addition	PADDW
_mm_add_pi32	Addition	PADDD
_mm_adds_pi8	Addition	PADDSB
_mm_adds_pi16	Addition	PADDSW

_mm_adds_pu8	Addition	PADDUSB
_mm_adds_pu16	Addition	PADDUSW
_mm_sub_pi8	Subtraction	PSUBB
_mm_sub_pi16	Subtraction	PSUBW
_mm_sub_pi32	Subtraction	PSUBD
_mm_subs_pi8	Subtraction	PSUBSB
_mm_subs_pi16	Subtraction	PSUBSW
_mm_subs_pu8	Subtraction	PSUBUSB
_mm_subs_pu16	Subtraction	PSUBUSW
_mm_madd_pi16	Multiply and add	PMADDWD
_mm_mulhi_pi16	Multiplication	PMULHW
_mm_mullo_pi16	Multiplication	PMULLW

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
__m64 _mm_subs_pi8(__m64 m1, __m64 m2)
```

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

```
__m64 _mm_subs_pi16(__m64 m1, __m64 m2)
```

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

```
__m64 _mm_subs_pu8(__m64 m1, __m64 m2)
```

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

```
__m64 _mm_subs_pu16(__m64 m1, __m64 m2)
```

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

```
__m64 _mm_madd_pi16(__m64 m1, __m64 m2)
```

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

```
__m64 _mm_mulhi_pi16(__m64 m1, __m64 m2)
```

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

```
__m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
```

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

MMX(TM) Technology Shift Intrinsics

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

Details about each intrinsic follows the table below.

Intrinsic Name	Operation	Corresponding MMX Instruction
_mm_sll_pi16	Logical shift left	PSLLW
_mm_slli_pi16	Logical shift left	PSLLWI
_mm_sll_pi32	Logical shift left	PSLLD
_mm_slli_pi32	Logical shift left	PSLLDI
_mm_sll_pi64	Logical shift left	PSLLQ
_mm_slli_pi64	Logical shift left	PSLLQI
_mm_sra_pi16	Arithmetic shift right	PSRAW
_mm_srai_pi16	Arithmetic shift right	PSRAWI
_mm_sra_pi32	Arithmetic shift right	PSRAD
_mm_srai_pi32	Arithmetic shift right	PSRADI
_mm_srl_pi16	Logical shift right	PSRLW
_mm_srli_pi16	Logical shift right	PSRLWI
_mm_srl_pi32	Logical shift right	PSRLD
_mm_srli_pi32	Logical shift right	PSRLDI
_mm_srl_pi64	Logical shift right	PSRLQ
_mm_srli_pi64	Logical shift right	PSRLQI

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

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)
```

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)
```

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)
```

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_pi64(__m64 m, __m64 count)
```

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

```
__m64 _mm_slli_pi64(__m64 m, int count)
```

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

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

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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

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_pi64(__m64 m, __m64 count)
```

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

```
__m64 _mm_srli_pi64(__m64 m, int count)
```

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

MMX(TM) Technology Logical Intrinsics

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

Details about each intrinsic follows the table below.

Intrinsic Name	Operation	Corresponding MMX Instruction
_mm_and_si64	Bitwise AND	PAND
_mm_andnot_si64	Bitwise ANDNOT	PANDN
_mm_or_si64	Bitwise OR	POR
_mm_xor_si64	Bitwise Exclusive OR	PXOR

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

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)
```

Perform a bitwise 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)
```

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)
```

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

MMX(TM) Technology Compare Intrinsics

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

The intrinsics in the following table perform compare operations. Details about each intrinsic follows the table below.

Intrinsic Name	Operation	Corresponding MMX Instruction
_mm_cmpeq_pi8	Equal	PCMPEQB

_mm_cmpeq_pi16	Equal	PCMPEQW
_mm_cmpeq_pi32	Equal	PCMPEQD
_mm_cmpgt_pi8	Greater Than	PCMPGTB
_mm_cmpgt_pi16	Greater Than	PCMPGTW
_mm_cmpgt_pi32	Greater Than	PCMPGTD

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

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

```
__m64 _mm_cmpeq_pi16(__m64 m1, __m64 m2)
```

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

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

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

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

If the respective 8-bit signed values in `m1` are greater than the respective 8-bit signed 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)
```

If the respective 16-bit signed values in `m1` are greater than the respective 16-bit signed 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 m2)
```

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

MMX(TM) Technology Set Intrinsics

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

Details about each intrinsic follows the table below.

Note

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

Intrinsic Name	Operation	Corresponding MMX Instruction
_mm_setzero_si64	set to zero	PXOR
_mm_set_pi32	set integer values	Composite
_mm_set_pi16	set integer values	Composite
_mm_set_pi8	set integer values	Composite
_mm_set1_pi32	set integer values	
_mm_set1_pi16	set integer values	Composite
_mm_set1_pi8	set integer values	Composite
_mm_setr_pi32	set integer values	Composite
_mm_setr_pi16	set integer values	Composite
_mm_setr_pi8	set integer values	Composite

```
__m64 _mm_setzero_si64()
```
Sets the 64-bit value to zero.

R
0x0

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

Sets the 2 signed 32-bit integer values.

R0	R1
i0	i1

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

Sets the 4 signed 16-bit integer values.

R0	R1	R2	R3
w0	w1	w2	w3

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

Sets the 8 signed 8-bit integer values.

R0	R1	...	R7
b0	b1	...	b7

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

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

R0	R1
i	i

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

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

R0	R1	R2	R3
w	w	w	w

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

Sets the 8 signed 8-bit integer values to `b`

R0	R1	...	R7
b	b	...	b

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

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

R0	R1
i1	i0

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

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

R0	R1	R2	R3
w3	w2	w1	w0

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

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

R0	R1	...	R7

b7	b6	...	b0

MMX(TM) Technology Intrinsics on IA-64 Architecture

MMX™ technology intrinsics provide access to the MMX technology instruction set on systems based on IA-64 architecture. To provide source compatibility with the IA-32 architecture, these intrinsics are equivalent both in name and functionality to the set of IA-32-based MMX intrinsics.

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

Data Types

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

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

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

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

Streaming SIMD Extensions

Overview: Streaming SIMD Extensions

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

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

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

Note

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

Floating-point Intrinsics for Streaming SIMD Extensions

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

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

Arithmetic Operations for Streaming SIMD Extensions

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

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R0-R3. R0, R1, R2 and R3 each represent one of the 4 32-bit pieces of the result register.

Details about each intrinsic follows the table below.

Intrinsic	Operation	Corresponding SSE Instruction
_mm_add_ss	Addition	ADDSS
_mm_add_ps	Addition	ADDPS
_mm_sub_ss	Subtraction	SUBSS
_mm_sub_ps	Subtraction	SUBPS
_mm_mul_ss	Multiplication	MULSS
_mm_mul_ps	Multiplication	MULPS
_mm_div_ss	Division	DIVSS
_mm_div_ps	Division	DIVPS
_mm_sqrt_ss	Squared Root	SQRTSS
_mm_sqrt_ps	Squared Root	SQRTPS
_mm_rcp_ss	Reciprocal	RCPSS
_mm_rcp_ps	Reciprocal	RCPPS
_mm_rsqrt_ss	Reciprocal Squared Root	RSQRTSS
_mm_rsqrt_ps	Reciprocal Squared Root	RSQRTPS
_mm_min_ss	Computes Minimum	MINSS
_mm_min_ps	Computes Minimum	MINPS
_mm_max_ss	Computes Maximum	MAXSS
_mm_max_ps	Computes Maximum	MAXPS

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

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

R0	R1	R2	R3
a0 + b0	a1	a2	a3

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

Adds the four SP FP values of `a` and `b`.

R0	R1	R2	R3
a0 +b0	a1 + b1	a2 + b2	a3 + b3

`__m128 _mm_sub_ss(__m128 a, __m128 b)`

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

R0	R1	R2	R3
a0 - b0	a1	a2	a3

`__m128 _mm_sub_ps(__m128 a, __m128 b)`

Subtracts the four SP FP values of `a` and `b`.

R0	R1	R2	R3
a0 - b0	a1 - b1	a2 - b2	a3 - b3

`__m128 _mm_mul_ss(__m128 a, __m128 b)`

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

R0	R1	R2	R3
a0 * b0	a1	a2	a3

`__m128 _mm_mul_ps(__m128 a, __m128 b)`

Multiplies the four SP FP values of `a` and `b`.

R0	R1	R2	R3
a0 * b0	a1 * b1	a2 * b2	a3 * b3

`__m128 _mm_div_ss(__m128 a, __m128 b )`

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

R0	R1	R2	R3
a0 / b0	a1	a2	a3

`__m128 _mm_div_ps(__m128 a, __m128 b)`

Divides the four SP FP values of `a` and `b`.

R0	R1	R2	R3
a0 / b0	a1 / b1	a2 / b2	a3 / b3

`__m128 _mm_sqrt_ss(__m128 a)`

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

R0	R1	R2	R3
sqrt(a0)	a1	a2	a3

`__m128 _mm_sqrt_ps(__m128 a)`

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

R0	R1	R2	R3
sqrt(a0)	sqrt(a1)	sqrt(a2)	sqrt(a3)

`__m128 _mm_rcp_ss(__m128 a)`

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

R0	R1	R2	R3
recip(a0)	a1	a2	a3

`__m128 _mm_rcp_ps(__m128 a)`

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

R0	R1	R2	R3
recip(a0)	recip(a1)	recip(a2)	recip(a3)

`__m128 _mm_rsqrt_ss(__m128 a)`

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

R0	R1	R2	R3
recip(sqrt(a0))	a1	a2	a3

`__m128 _mm_rsqrt_ps(__m128 a)`

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

R0	R1	R2	R3
recip(sqrt(a0))	recip(sqrt(a1))	recip(sqrt(a2))	recip(sqrt(a3))

__m128 _mm_min_ss(__m128 a, __m128 b)

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

R0	R1	R2	R3
min(a0, b0)	a1	a2	a3

__m128 _mm_min_ps(__m128 a, __m128 b)

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

R0	R1	R2	R3
min(a0, b0)	min(a1, b1)	min(a2, b2)	min(a3, b3)

__m128 _mm_max_ss(__m128 a, __m128 b)

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

R0	R1	R2	R3
max(a0, b0)	a1	a2	a3

__m128 _mm_max_ps(__m128 a, __m128 b)

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

R0	R1	R2	R3
max(a0, b0)	max(a1, b1)	max(a2, b2)	max(a3, b3)

Logical Operations for Streaming SIMD Extensions

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

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R0-R3. R0, R1, R2 and R3 each represent one of the 4 32-bit pieces of the result register.

Details about each intrinsic follows the table below.

Intrinsic Name	Operation	Corresponding SSE Instruction

_mm_and_ps	Bitwise AND	ANDPS
_mm_andnot_ps	Bitwise ANDNOT	ANDNPS
_mm_or_ps	Bitwise OR	ORPS
_mm_xor_ps	Bitwise Exclusive OR	XORPS

__m128 _mm_and_ps(__m128 a, __m128 b)

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

R0	R1	R2	R3
a0 & b0	a1 & b1	a2 & b2	a3 & b3

__m128 _mm_andnot_ps(__m128 a, __m128 b)

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

R0	R1	R2	R3
~a0 & b0	~a1 & b1	~a2 & b2	~a3 & b3

__m128 _mm_or_ps(__m128 a, __m128 b)

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

R0	R1	R2	R3
a0 | b0	a1 | b1	a2 | b2	a3 | b3

__m128 _mm_xor_ps(__m128 a, __m128 b)

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

R0	R1	R2	R3
a0 ^ b0	a1 ^ b1	a2 ^ b2	a3 ^ b3

Comparisons for Streaming SIMD Extensions

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

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R or R0-R3. R0, R1, R2 and R3 each represent one of the 4 32-bit pieces of the result register.

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

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

_mm_cmpnlt_ps	Not Less Than	CMPNLTPS
_mm_cmpnle_ss	Not Less Than or Equal	CMPNLESS
_mm_cmpnle_ps	Not Less Than or Equal	CMPNLEPS
_mm_cmpngt_ss	Not Greater Than	CMPNLTSS
_mm_cmpngt_ps	Not Greater Than	CMPNLTPS
_mm_cmpnge_ss	Not Greater Than or Equal	CMPNLESS
_mm_cmpnge_ps	Not Greater Than or Equal	CMPNLEPS
_mm_cmpord_ss	Ordered	CMPORDSS
_mm_cmpord_ps	Ordered	CMPORDPS
_mm_cmpunord_ss	Unordered	CMPUNORDSS
_mm_cmpunord_ps	Unordered	CMPUNORDPS
_mm_comieq_ss	Equal	COMISS
_mm_comilt_ss	Less Than	COMISS
_mm_comile_ss	Less Than or Equal	COMISS
_mm_comigt_ss	Greater Than	COMISS
_mm_comige_ss	Greater Than or Equal	COMISS
_mm_comineq_ss	Not Equal	COMISS

_mm_ucomieq_ss	Equal	UCOMISS
_mm_ucomilt_ss	Less Than	UCOMISS
_mm_ucomile_ss	Less Than or Equal	UCOMISS
_mm_ucomigt_ss	Greater Than	UCOMISS
_mm_ucomige_ss	Greater Than or Equal	UCOMISS
_mm_ucomineq_ss	Not Equal	UCOMISS

__m128 _mm_cmpeq_ss(__m128 a, __m128 b)

Compare for equality.

R0	R1	R2	R3
(a0 == b0) ? 0xffffffff : 0x0	a1	a2	a3

__m128 _mm_cmpeq_ps(__m128 a, __m128 b)

Compare for equality.

R0	R1	R2	R3
(a0 == b0) ? 0xffffffff : 0x0	(a1 == b1) ? 0xffffffff : 0x0	(a2 == b2) ? 0xffffffff : 0x0	(a3 == b3) ? 0xffffffff : 0x0

__m128 _mm_cmplt_ss(__m128 a, __m128 b)

Compare for less-than.

R0	R1	R2	R3
(a0 < b0) ? 0xffffffff : 0x0	a1	a2	a3

__m128 _mm_cmplt_ps(__m128 a, __m128 b)

Compare for less-than

R0	R1	R2	R3
(a0 < b0) ? 0xffffffff : 0x0	(a1 < b1) ? 0xffffffff : 0x0	(a2 < b2) ? 0xffffffff : 0x0	(a3 < b3) ? 0xffffffff : 0x0

__m128 _mm_cmple_ss(__m128 a, __m128 b)

Compare for less-than-or-equal.

R0	R1	R2	R3
(a0 <= b0) ? 0xffffffff : 0x0	a1	a2	a3

__m128 _mm_cmple_ps(__m128 a, __m128 b)

Compare for less-than-or-equal.

R0	R1	R2	R3
(a0 <= b0) ? 0xffffffff : 0x0	(a1 <= b1) ? 0xffffffff : 0x0	(a2 <= b2) ? 0xffffffff : 0x0	(a3 <= b3) ? 0xffffffff : 0x0

__m128 _mm_cmpgt_ss(__m128 a, __m128 b)

Compare for greater-than.

R0	R1	R2	R3
(a0 > b0) ? 0xffffffff : 0x0	a1	a2	a3

__m128 _mm_cmpgt_ps(__m128 a, __m128 b)

Compare for greater-than.

R0	R1	R2	R3
(a0 > b0) ? 0xffffffff : 0x0	(a1 > b1) ? 0xffffffff : 0x0	(a2 > b2) ? 0xffffffff : 0x0	(a3 > b3) ? 0xffffffff : 0x0

__m128 _mm_cmpge_ss(__m128 a, __m128 b)

Compare for greater-than-or-equal.

R0	R1	R2	R3
(a0 >= b0) ? 0xffffffff : 0x0	a1	a2	a3

__m128 _mm_cmpge_ps(__m128 a, __m128 b)

Compare for greater-than-or-equal.

R0	R1	R2	R3
(a0 >= b0) ? 0xffffffff : 0x0	(a1 >= b1) ? 0xffffffff : 0x0	(a2 >= b2) ? 0xffffffff : 0x0	(a3 >= b3) ? 0xffffffff : 0x0

__m128 _mm_cmpneq_ss(__m128 a, __m128 b)

Compare for inequality.

R0	R1	R2	R3
(a0 != b0) ? 0xffffffff : 0x0	a1	a2	a3

__m128 _mm_cmpneq_ps(__m128 a, __m128 b)

Compare for inequality.

R0	R1	R2	R3
(a0 != b0) ? 0xffffffff : 0x0	(a1 != b1) ? 0xffffffff : 0x0	(a2 != b2) ? 0xffffffff : 0x0	(a3 != b3) ? 0xffffffff : 0x0

__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)

Compare for not-less-than.

R0	R1	R2	R3
!(a0 < b0) ? 0xffffffff : 0x0	a1	a2	a3

__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)

Compare for not-less-than.

R0	R1	R2	R3
!(a0 < b0) ? 0xffffffff : 0x0	!(a1 < b1) ? 0xffffffff : 0x0	!(a2 < b2) ? 0xffffffff : 0x0	!(a3 < b3) ? 0xffffffff : 0x0

__m128 _mm_cmpnle_ss(__m128 a, __m128 b)

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

R0	R1	R2	R3
!(a0 <= b0) ? 0xffffffff : 0x0	a1	a2	a3

__m128 _mm_cmpnle_ps(__m128 a, __m128 b)

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

R0	R1	R2	R3
!(a0 <= b0) ? 0xffffffff : 0x0	!(a1 <= b1) ? 0xffffffff : 0x0	!(a2 <= b2) ? 0xffffffff : 0x0	!(a3 <= b3) ? 0xffffffff : 0x0

__m128 _mm_cmpngt_ss(__m128 a, __m128 b)

Compare for not-greater-than.

R0	R1	R2	R3
!(a0 > b0) ? 0xffffffff : 0x0	a1	a2	a3

__m128 _mm_cmpngt_ps(__m128 a, __m128 b)

Compare for not-greater-than.

R0	R1	R2	R3
!(a0 > b0) ? 0xffffffff : 0x0	!(a1 > b1) ? 0xffffffff : 0x0	!(a2 > b2) ? 0xffffffff : 0x0	!(a3 > b3) ? 0xffffffff : 0x0

__m128 _mm_cmpnge_ss(__m128 a, __m128 b)

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

R0	R1	R2	R3
!(a0 >= b0) ? 0xffffffff : 0x0	a1	a2	a3

__m128 _mm_cmpnge_ps(__m128 a, __m128 b)

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

R0	R1	R2	R3
!(a0 >= b0) ? 0xffffffff : 0x0	!(a1 >= b1) ? 0xffffffff : 0x0	!(a2 >= b2) ? 0xffffffff : 0x0	!(a3 >= b3) ? 0xffffffff : 0x0

__m128 _mm_cmpord_ss(__m128 a, __m128 b)

Compare for ordered.

R0	R1	R2	R3
(a0 ord? b0) ? 0xffffffff : 0x0	a1	a2	a3

__m128 _mm_cmpord_ps(__m128 a, __m128 b)

Compare for ordered.

R0	R1	R2	R3
(a0 ord? b0) ? 0xffffffff : 0x0	(a1 ord? b1) ? 0xffffffff : 0x0	(a2 ord? b2) ? 0xffffffff : 0x0	(a3 ord? b3) ? 0xffffffff : 0x0

__m128 _mm_cmpunord_ss(__m128 a, __m128 b)

Compare for unordered.

R0	R1	R2	R3
(a0 unord? b0) ? 0xffffffff : 0x0	a1	a2	a3

__m128 _mm_cmpunord_ps(__m128 a, __m128 b)

Compare for unordered.

R0	R1	R2	R3
(a0 unord? b0) ? 0xffffffff : 0x0	(a1 unord? b1) ? 0xffffffff : 0x0	(a2 unord? b2) ? 0xffffffff : 0x0	(a3 unord? b3) ? 0xffffffff : 0x0

int _mm_comieq_ss(__m128 a, __m128 b)

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

R
(a0 == b0) ? 0x1 : 0x0

int _mm_comilt_ss(__m128 a, __m128 b)

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

R
(a0 < b0) ? 0x1 : 0x0

int _mm_comile_ss(__m128 a, __m128 b)

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

R
(a0 <= b0) ? 0x1 : 0x0

int _mm_comigt_ss(__m128 a, __m128 b)

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

R
(a0 > b0) ? 0x1 : 0x0

int _mm_comige_ss(__m128 a, __m128 b)

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

R
(a0 >= b0) ? 0x1 : 0x0

int _mm_comineq_ss(__m128 a, __m128 b)

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

R
(a0 != b0) ? 0x1 : 0x0

int _mm_ucomieq_ss(__m128 a, __m128 b)

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

R
(a0 == b0) ? 0x1 : 0x0

int _mm_ucomilt_ss(__m128 a, __m128 b)

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

R
(a0 < b0) ? 0x1 : 0x0

int _mm_ucomile_ss(__m128 a, __m128 b)

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

R
(a0 <= b0) ? 0x1 : 0x0

int _mm_ucomigt_ss(__m128 a, __m128 b)

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

R
(a0 > b0) ? 0x1 : 0x0

int _mm_ucomige_ss(__m128 a, __m128 b)

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

R
(a0 >= b0) ? 0x1 : 0x0

int _mm_ucomineq_ss(__m128 a, __m128 b)

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

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

Conversion Operations for Streaming SIMD Extensions

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R or R0-R3. R0, R1, R2 and R3 each represent one of the 4 32-bit pieces of the result register.

Details about each intrinsic follows the table below.

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

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_cvtss_si32	Convert to 32-bit integer	CVTSS2SI
_mm_cvtss_si64	Convert to 64-bit integer	CVTSS2SI
_mm_cvtps_pi32	Convert to two 32-bit integers	CVTPS2PI
_mm_cvttss_si32	Convert to 32-bit integer	CVTTSS2SI
_mm_cvttss_si64	Convert to 64-bit integer	CVTTSS2SI
_mm_cvttps_pi32	Convert to two 32-bit integers	CVTTPS2PI
_mm_cvtsi32_ss	Convert from 32-bit integer	CVTSI2SS
_mm_cvtsi64_ss	Convert from 64-bit integer	CVTSI2SS
_mm_cvtpi32_ps	Convert from two 32-bit integers	CVTTPI2PS

_mm_cvtpi16_ps	Convert from four 16-bit integers	composite
_mm_cvtpu16_ps	Convert from four 16-bit integers	composite
_mm_cvtpi8_ps	Convert from four 8-bit integers	composite
_mm_cvtpu8_ps	Convert from four 8-bit integers	composite
_mm_cvtpi32x2_ps	Convert from four 32-bit integers	composite
_mm_cvtps_pi16	Convert to four 16-bit integers	composite
_mm_cvtps_pi8	Convert to four 8-bit integers	composite
_mm_cvtss_f32	Extract	composite

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

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

R
(int)a0

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

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

R
(__int64)a0

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

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

R0	R1
(int)a0	(int)a1

```
int _mm_cvttss_si32(__m128 a)
```

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

R
(int)a0

__int64 _mm_cvttss_si64(__m128 a)
Convert the lower SP FP value of a to a 64-bit signed integer with truncation.

R
(__int64)a0

__m64 _mm_cvttps_pi32(__m128 a)

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

R0	R1
(int)a0	(int)a1

__m128 _mm_cvtsi32_ss(__m128 a, int b)

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

R0	R1	R2	R3
(float)b	a1	a2	a3

__m128 _mm_cvtsi64_ss(__m128 a, __int64 b)

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

R0	R1	R2	R3
(float)b	a1	a2	a3

__m128 _mm_cvtpi32_ps(__m128 a, __m64 b)

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	R1	R2	R3
(float)b0	(float)b1	a2	a3

__m128 _mm_cvtpi16_ps(__m64 a)

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

R0	R1	R2	R3
(float)a0	(float)a1	(float)a2	(float)a3

__m128 _mm_cvtpu16_ps(__m64 a)

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

R0	R1	R2	R3
(float)a0	(float)a1	(float)a2	(float)a3

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

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

R0	R1	R2	R3
(float)a0	(float)a1	(float)a2	(float)a3

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

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

R0	R1	R2	R3
(float)a0	(float)a1	(float)a2	(float)a3

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

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

R0	R1	R2	R3
(float)a0	(float)a1	(float)b0	(float)b1

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

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

R0	R1	R2	R3
(short)a0	(short)a1	(short)a2	(short)a3

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

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

R0	R1	R2	R3
(char)a0	(char)a1	(char)a2	(char)a3

```
float _mm_cvtss_f32(__m128 a)
```

This intrinsic extracts a single precision floating point value from the first vector element of an __m128. It does so in the most efficient manner possible in the context used.

Load Operations for Streaming SIMD Extensions

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

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R0-R3. R0, R1, R2 and R3 each represent one of the 4 32-bit pieces of the result register.

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_loadh_pi	Load high	MOVHPS reg, mem
_mm_loadl_pi	Load low	MOVLPS reg, mem
_mm_load_ss	Load the low value and clear the three high values	MOVSS
_mm_load1_ps	Load one value into all four words	MOVSS + Shuffling
_mm_load_ps	Load four values, address aligned	MOVAPS
_mm_loadu_ps	Load four values, address unaligned	MOVUPS
_mm_loadr_ps	Load four values in reverse	MOVAPS + Shuffling

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

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

R0	R1	R2	R3
a0	a1	*p0	*p1

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

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

R0	R1	R2	R3
*p0	*p1	a2	a3

`__m128 _mm_load_ss(float * p )`

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

R0	R1	R2	R3
*p	0.0	0.0	0.0

`__m128 _mm_load1_ps(float * p )`

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

R0	R1	R2	R3
*p	*p	*p	*p

`__m128 _mm_load_ps(float * p )`

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

R0	R1	R2	R3
p[0]	p[1]	p[2]	p[3]

`__m128 _mm_loadu_ps(float * p)`

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

R0	R1	R2	R3
p[0]	p[1]	p[2]	p[3]

`__m128 _mm_loadr_ps(float * p)`

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

R0	R1	R2	R3
p[3]	p[2]	p[1]	p[0]

Set Operations for Streaming SIMD Extensions

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

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R0, R1, R2 and R3 represent the registers in which results are placed.

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_set_ss	Set the low value and clear the three high values	Composite
_mm_set1_ps	Set all four words with the same value	Composite
_mm_set_ps	Set four values, address aligned	Composite
_mm_setr_ps	Set four values, in reverse order	Composite
_mm_setzero_ps	Clear all four values	Composite

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

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

R0	R1	R2	R3
w	0.0	0.0	0.0

```
__m128 _mm_set1_ps(float w )
```

Sets the four SP FP values to `w`.

R0	R1	R2	R3
w	w	w	w

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

Sets the four SP FP values to the four inputs.

R0	R1	R2	R3
w	x	y	z

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

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

R0	R1	R2	R3
z	y	x	w

```
__m128 _mm_setzero_ps (void)
```

Clears the four SP FP values.

R0	R1	R2	R3
0.0	0.0	0.0	0.0

Store Operations for Streaming SIMD Extensions

Details about each intrinsic follows the table below.

The detailed description of each intrinsic contains a table detailing the returns. In these tables, p[n] is an access to the n element of the result.

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

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_storeh_pi	Store high	MOVHPS mem, reg
_mm_storel_pi	Store low	MOVLPS mem, reg
_mm_store_ss	Store the low value	MOVSS
_mm_store1_ps	Store the low value across all four words, address aligned	Shuffling + MOVSS
_mm_store_ps	Store four values, address aligned	MOVAPS
_mm_storeu_ps	Store four values, address unaligned	MOVUPS
_mm_storer_ps	Store four values, in reverse order	MOVAPS + Shuffling

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

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

*p0	*p1
a2	a3

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

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

*p0	*p1
a0	a1

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

Stores the lower SP FP value.

*p
a0

void _mm_store1_ps(float * p, __m128 a)

Stores the lower SP FP value across four words.

p[0]	p[1]	p[2]	p[3]
a0	a0	a0	a0

void _mm_store_ps(float *p, __m128 a)

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

p[0]	p[1]	p[2]	p[3]
a0	a1	a2	a3

void _mm_storeu_ps(float *p, __m128 a)

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

p[0]	p[1]	p[2]	p[3]
a0	a1	a2	a3

void _mm_storer_ps(float * p, __m128 a)

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

p[0]	p[1]	p[2]	p[3]
a3	a2	a1	a0

Cacheability Support Using Streaming SIMD Extensions

Details about each intrinsic follows the table below.

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

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_prefetch	Load	PREFETCH
_mm_stream_pi	Store	MOVNTQ
_mm_stream_ps	Store	MOVNTPS
_mm_sfence	Store fence	SFENCE

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

Loads one cache line of data from address `a` to a location "closer" to the processor. The value `sel` specifies the type of prefetch operation: the constants `_MM_HINT_T0`, `_MM_HINT_T1`, `_MM_HINT_T2`, and `_MM_HINT_NTA` should be used for IA-32, corresponding to the type of `prefetch` instruction. The constants `_MM_HINT_T1`, `_MM_HINT_NT1`, `_MM_HINT_NT2`, and `_MM_HINT_NTA` should be used for systems based on IA-64 architecture.

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

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.

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

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

```
void _mm_sfence(void)
```

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

Integer Intrinsics Using Streaming SIMD Extensions

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R, R0, R1...R7 represent the registers in which results are placed.

Details about each intrinsic follows the table below.

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

Before using these intrinsics, you must empty the multimedia state for the MMX(TM) technology register. See The EMMS Instruction: Why You Need It for more details.

Intrinsic Name	Operation	Corresponding SSE Instruction
`_mm_extract_pi16`	Extract one of four words	`PEXTRW`
`_mm_insert_pi16`	Insert word	`PINSRW`
`_mm_max_pi16`	Compute maximum	`PMAXSW`
`_mm_max_pu8`	Compute maximum, unsigned	`PMAXUB`
`_mm_min_pi16`	Compute minimum	`PMINSW`

_mm_min_pu8	Compute minimum, unsigned	PMINUB
_mm_movemask_pi8	Create eight-bit mask	PMOVMSKB
_mm_mulhi_pu16	Multiply, return high bits	PMULHUW
_mm_shuffle_pi16	Return a combination of four words	PSHUFW
_mm_maskmove_si64	Conditional Store	MASKMOVQ
_mm_avg_pu8	Compute rounded average	PAVGB
_mm_avg_pu16	Compute rounded average	PAVGW
_mm_sad_pu8	Compute sum of absolute differences	PSADBW

`int _mm_extract_pi16(__m64 a, int n)`

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

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

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

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

R0	R1	R2	R3
(n==0) ? d : a0;	(n==1) ? d : a1;	(n==2) ? d : a2;	(n==3) ? d : a3;

`__m64 _mm_max_pi16(__m64 a, __m64 b)`

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

R0	R1	R2	R3
min(a0, b0)	min(a1, b1)	min(a2, b2)	min(a3, b3)

`__m64 _mm_max_pu8(__m64 a, __m64 b)`

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

R0	R1	...	R7
min(a0, b0)	min(a1, b1)	...	min(a7, b7)

`__m64 _mm_min_pi16(__m64 a, __m64 b)`

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

R0	R1	R2	R3
min(a0, b0)	min(a1, b1)	min(a2, b2)	min(a3, b3)

__m64 _mm_min_pu8(__m64 a, __m64 b)

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

R0	R1	...	R7
min(a0, b0)	min(a1, b1)	...	min(a7, b7)

__m64 _mm_movemask_pi8(__m64 b)

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)

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

R0	R1	R2	R3
hiword(a0 * b0)	hiword(a1 * b1)	hiword(a2 * b2)	hiword(a3 * b3)

__m64 _mm_shuffle_pi16(__m64 a, int n)

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

R0	R1	R2	R3
word (n&0x3) of a	word ((n>>2)&0x3) of a	word ((n>>4)&0x3) of a	word ((n>>6)&0x3) of a

void _mm_maskmove_si64(__m64 d, __m64 n, char *p)

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

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

__m64 _mm_avg_pu8(__m64 a, __m64 b)

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

R0	R1	...	R7
(t >> 1) | (t &	(t >> 1) | (t &	...	((t >> 1) | (t &

0x01), where t = (unsigned char)a0 + (unsigned char)b0	0x01), where t = (unsigned char)a1 + (unsigned char)b1		0x01)), where t = (unsigned char)a7 + (unsigned char)b7

`__m64 _mm_avg_pu16(__m64 a, __m64 b)`

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

R0	R1	...	R7
(t >> 1) | (t & 0x01), where t = (unsigned int)a0 + (unsigned int)b0	(t >> 1) | (t & 0x01), where t = (unsigned int)a1 + (unsigned int)b1	...	(t >> 1) | (t & 0x01), where t = (unsigned int)a7 + (unsigned int)b7

`__m64 _mm_sad_pu8(__m64 a, __m64 b)`

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	R1	R2	R3
abs(a0-b0) +... + abs(a7-b7)	0	0	0

Intrinsics to Read and Write Registers for Streaming SIMD Extensions

Details about each intrinsic follows the table below.

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

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_getcsr	Return control register	STMXCSR
_mm_setcsr	Set control register	LDMXCSR

`unsigned int _mm_getcsr(void)`

Returns the contents of the control register.

`void _mm_setcsr(unsigned int i)`

Sets the control register to the value specified.

Miscellaneous Intrinsics Using Streaming SIMD Extensions

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

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R, R0, R1, R2 and R3 represent the registers in which results are placed.

Details about each intrinsic follows the table below.

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_shuffle_ps	Shuffle	SHUFPS
_mm_unpackhi_ps	Unpack High	UNPCKHPS
_mm_unpacklo_ps	Unpack Low	UNPCKLPS
_mm_move_ss	Set low word, pass in three high values	MOVSS
_mm_movehl_ps	Move High to Low	MOVHLPS
_mm_movelh_ps	Move Low to High	MOVLHPS
_mm_movemask_ps	Create four-bit mask	MOVMSKPS

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

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

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

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

R0	R1	R2	R3
a2	b2	a3	b3

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

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

R0	R1	R2	R3
a0	b0	a1	b1

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

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

R0	R1	R2	R3
b0	a1	a2	a3

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

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

R0	R1	R2	R3
b2	b3	a2	a3

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

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

R0	R1	R2	R3
a0	a1	b0	b1

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

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

R
sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 | sign(a0)

Using Streaming SIMD Extensions on IA-64 Architecture

The Streaming SIMD Extensions (SSE) intrinsics provide access to IA-64 instructions for Streaming SIMD Extensions. To provide source compatibility with the IA-32 architecture, these intrinsics are equivalent both in name and functionality to the set of IA-32-based SSE intrinsics.

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

- Certain intrinsics are provided only for compatibility with previously-defined IA-32 intrinsics. Using them on systems based on IA-64 architecture probably leads to performance degradation.
- Floating-point (FP) data loaded stored as `__m128` objects must be 16-byte-aligned.
- Some intrinsics require that their arguments be immediates -- that is, constant integers (literals), due to the nature of the instruction.

Data Types

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

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

Because IA-64 instructions treat the SSE registers in the same way whether you are using packed or scalar data, there is no __m32 data type to represent scalar data. For scalar operations, use the __m128 objects and the "scalar" forms of the intrinsics; the compiler and the processor implement these operations with 32-bit memory references. But, for better performance the packed form should be substituting for the scalar form whenever possible.

The address of a __m128 object may be taken.

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

Implementation on systems based on IA-64 architecture

SSE intrinsics are defined for the __m128 data type, a 128-bit quantity consisting of four single-precision FP values. SIMD instructions for systems based on IA-64 architecture operate on 64-bit FP register quantities containing two single-precision floating-point values. Thus, each __m128 operand is actually a pair of FP registers and therefore each intrinsic corresponds to at least one pair of IA-64 instructions operating on the pair of FP register operands.

Compatibility versus Performance

Many of the SSE intrinsics for systems based on IA-64 architecture were created for compatibility with existing IA-32 intrinsics and not for performance. In some situations, intrinsic usage that improved performance on IA-32 architecture will not do so on systems based on IA-64 architecture. One reason for this is that some intrinsics map nicely into the IA-32 instruction set but not into the IA-64 instruction set. Thus, it is important to differentiate between intrinsics which were implemented for a performance advantage on systems based on IA-64 architecture, and those implemented simply to provide compatibility with existing IA-32 code.

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

- Any SSE scalar intrinsic (_ss variety) - use packed (_ps) version if possible
- comi and ucomi SSE comparisons - these correspond to IA-32 COMISS and UCOMISS instructions only. A sequence of IA-64 instructions are required to implement these.

- Conversions in general are multi-instruction operations. These are particularly expensive: `_mm_cvtpi16_ps`, `_mm_cvtpu16_ps`, `_mm_cvtpi8_ps`, `_mm_cvtpu8_ps`, `_mm_cvtpi32x2_ps`, `_mm_cvtps_pi16`, `_mm_cvtps_pi8`
- SSE utility intrinsic `_mm_movemask_ps`

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

Macro Functions

Macro Function for Shuffle Using Streaming SIMD Extensions

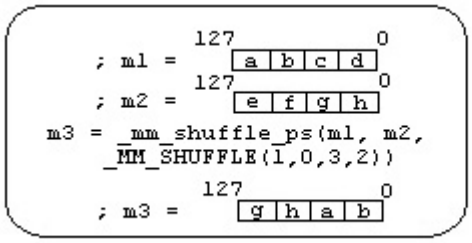
The Streaming SIMD Extensions (SSE) provide a macro function to help create constants that describe shuffle operations. The macro takes four small integers (in the range of 0 to 3) and combines them into an 8-bit immediate value used by the `SHUFPS` instruction.

Shuffle Function Macro

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

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

View of Original and Result Words with Shuffle Function Macro



Macro Functions to Read and Write the Control Registers

The following macro functions enable you to read and write bits to and from the control register. For details, see Intrinsics to Read and Write Registers. For Itanium®-based systems, these macros do not allow you to access all of the bits of the FPSR. See the descriptions for the `getfpsr()` and `setfpsr()` intrinsics in the Native Intrinsics for IA-64 Instructions topic.

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

	_MM_EXCEPT_UNDERFLOW
	_MM_EXCEPT_INEXACT

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

Exception State Macros with _MM_EXCEPT_DIV_ZERO

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

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

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

Exception Mask with _MM_MASK_OVERFLOW and _MM_MASK_UNDERFLOW
_MM_SET_EXCEPTION_MASK(MM_MASK_OVERFLOW | _MM_MASK_UNDERFLOW)

Rounding Mode	Macro Arguments
_MM_SET_ROUNDING_MODE(x)	_MM_ROUND_NEAREST
_MM_GET_ROUNDING_MODE()	_MM_ROUND_DOWN
Macro Definition Write to and read from bits thirteen and fourteen of the control register.	_MM_ROUND_UP
	_MM_ROUND_TOWARD_ZERO

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

Rounding Mode with _MM_ROUND_TOWARD_ZERO

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

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

The following example disables flush-to-zero mode.

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

Macro Function for Matrix Transposition

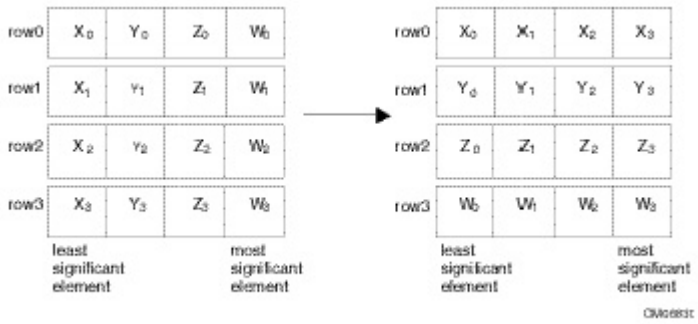
The Streaming SIMD Extensions (SSE) provide the following macro function to transpose a 4 by 4 matrix of single precision floating point values.

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

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

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

Matrix Transposition Using _MM_TRANSPOSE4_PS Macro



Streaming SIMD Extensions 2

Overview: Streaming SIMD Extensions 2

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

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

 Note

There are no intrinsics for floating-point move operations. To move data from one register to another, a simple assignment, `A = B`, suffices, where `A` and `B` are the source and target registers for the move operation.

 Note

On processors that do not support SSE2 instructions but do support MMX Technology, you can use the sse2mmx.h emulation pack to enable support for SSE2 instructions. You can use the sse2mmx.h header file for the following processors:

- Itanium® Processor
- Pentium® III Processor
- Pentium® II Processor
- Pentium® with MMX™ Technology

You should be familiar with the hardware features provided by the SSE2 when writing programs with the intrinsics. The following are three important issues to keep in mind:

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

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

 Note

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

Floating-point Intrinsics

Floating-point Arithmetic Operations for Streaming SIMD Extensions 2

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

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R0 and R1. R0 and R1 each represent one piece of the result register.

The Double Complex code sample contains examples of how to use several of these intrinsics.

Intrinsic Name	Operation	Corresponding SSE2 Instruction
_mm_add_sd	Addition	ADDSD
_mm_add_pd	Addition	ADDPD
_mm_sub_sd	Subtraction	SUBSD
_mm_sub_pd	Subtraction	SUBPD
_mm_mul_sd	Multiplication	MULSD
_mm_mul_pd	Multiplication	MULPD
_mm_div_sd	Division	DIVSD
_mm_div_pd	Division	DIVPD
_mm_sqrt_sd	Computes Square Root	SQRTSD
_mm_sqrt_pd	Computes Square Root	SQRTPD
_mm_min_sd	Computes Minimum	MINSD
_mm_min_pd	Computes Minimum	MINPD
_mm_max_sd	Computes Maximum	MAXSD
_mm_max_pd	Computes Maximum	MAXPD

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

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

R0	R1
a0 + b0	a1

__m128d _mm_add_pd(__m128d a, __m128d b)

Adds the two DP FP values of a and b.

R0	R1
a0 + b0	a1 + b1

__m128d _mm_sub_sd(__m128d a, __m128d b)

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

R0	R1
a0 - b0	a1

__m128d _mm_sub_pd(__m128d a, __m128d b)

Subtracts the two DP FP values of b from a.

R0	R1
a0 - b0	a1 - b1

__m128d _mm_mul_sd(__m128d a, __m128d b)

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

R0	R1
a0 * b0	a1

__m128d _mm_mul_pd(__m128d a, __m128d b)

Multiplies the two DP FP values of a and b.

R0	R1
a0 * b0	a1 * b1

__m128d _mm_div_sd(__m128d a, __m128d b)

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

R0	R1
a0 / b0	a1

__m128d _mm_div_pd(__m128d a, __m128d b)

Divides the two DP FP values of a and b.

R0	R1
a0 / b0	a1 / b1

__m128d _mm_sqrt_sd(__m128d a, __m128d b)

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

R0	R1
sqrt(b0)	A1

__m128d _mm_sqrt_pd(__m128d a)

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

R0	R1
sqrt(a0)	sqrt(a1)

__m128d _mm_min_sd(__m128d a, __m128d b)

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

R0	R1
min (a0, b0)	a1

__m128d _mm_min_pd(__m128d a, __m128d b)

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

R0	R1
min (a0, b0)	min(a1, b1)

__m128d _mm_max_sd(__m128d a, __m128d b)

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

R0	R1
max (a0, b0)	a1

__m128d _mm_max_pd(__m128d a, __m128d b)

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

R0	R1
max (a0, b0)	max (a1, b1)

Floating-point Logical Operations for Streaming SIMD Extensions 2

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

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R0 and R1 represent the registers in which results are placed.

Intrinsic Name	Operation	Corresponding SSE2 Instruction
_mm_and_pd	Computes AND	ANDPD
_mm_andnot_pd	Computes AND and NOT	ANDNPD
_mm_or_pd	Computes OR	ORPD
_mm_xor_pd	Computes XOR	XORPD

__m128d _mm_and_pd(__m128d a, __m128d b)

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

R0	R1
a0 & b0	a1 & b1

__m128d _mm_andnot_pd(__m128d a, __m128d b)

Computes the bitwise AND of the 128-bit value in b and the bitwise NOT of the 128-bit value in a.

R0	R1
(~a0) & b0	(~a1) & b1

__m128d _mm_or_pd(__m128d a, __m128d b)

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

R0	R1
a0 | b0	a1 | b1

__m128d _mm_xor_pd(__m128d a, __m128d b)

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

R0	R1
a0 ^ b0	a1 ^ b1

Floating-point Comparison Operations for Streaming SIMD Extensions 2

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

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R, R0 and R1. R, R0 and R1 each represent one piece of the result register.

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

Intrinsic Name	Operation	Corresponding SSE2 Instruction
_mm_cmpeq_pd	Equality	CMPEQPD
_mm_cmplt_pd	Less Than	CMPLTPD
_mm_cmple_pd	Less Than or Equal	CMPLEPD
_mm_cmpgt_pd	Greater Than	CMPLTPDr
_mm_cmpge_pd	Greater Than or Equal	CMPLEPDr
_mm_cmpord_pd	Ordered	CMPORDPD
_mm_cmpunord_pd	Unordered	CMPUNORDPD
_mm_cmpneq_pd	Inequality	CMPNEQPD

_mm_cmpnlt_pd	Not Less Than	CMPNLTPD
_mm_cmpnle_pd	Not Less Than or Equal	CMPNLEPD
_mm_cmpngt_pd	Not Greater Than	CMPNLTPDr
_mm_cmpnge_pd	Not Greater Than or Equal	CMPNLEPDr
_mm_cmpeq_sd	Equality	CMPEQSD
_mm_cmplt_sd	Less Than	CMPLTSD
_mm_cmple_sd	Less Than or Equal	CMPLESD
_mm_cmpgt_sd	Greater Than	CMPLTSDr
_mm_cmpge_sd	Greater Than or Equal	CMPLESDr
_mm_cmpord_sd	Ordered	CMPORDSD
_mm_cmpunord_sd	Unordered	CMPUNORDSD
_mm_cmpneq_sd	Inequality	CMPNEQSD
_mm_cmpnlt_sd	Not Less Than	CMPNLTSD
_mm_cmpnle_sd	Not Less Than or Equal	CMPNLESD
_mm_cmpngt_sd	Not Greater Than	CMPNLTSDr
_mm_cmpnge_sd	Not Greater Than or Equal	CMPNLESDr
_mm_comieq_sd	Equality	COMISD
_mm_comilt_sd	Less Than	COMISD
_mm_comile_sd	Less Than or Equal	COMISD
_mm_comigt_sd	Greater Than	COMISD
_mm_comige_sd	Greater Than or Equal	COMISD
_mm_comineq_sd	Not Equal	COMISD
_mm_ucomieq_sd	Equality	UCOMISD
_mm_ucomilt_sd	Less Than	UCOMISD
_mm_ucomile_sd	Less Than or Equal	UCOMISD
_mm_ucomigt_sd	Greater Than	UCOMISD
_mm_ucomige_sd	Greater Than or Equal	UCOMISD
_mm_ucomineq_sd	Not Equal	UCOMISD

`__m128d _mm_cmpeq_pd(__m128d a, __m128d b)`

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

R0	R1
(a0 == b0) ? 0xffffffffffffffff : 0x0	(a1 == b1) ? 0xffffffffffffffff : 0x0

__m128d _mm_cmplt_pd(__m128d a, __m128d b)

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

R0	R1
(a0 < b0) ? 0xffffffffffffffff : 0x0	(a1 < b1) ? 0xffffffffffffffff : 0x0

__m128d _mm_cmple_pd(__m128d a, __m128d b)

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

R0	R1
(a0 <= b0) ? 0xffffffffffffffff : 0x0	(a1 <= b1) ? 0xffffffffffffffff : 0x0

__m128d _mm_cmpgt_pd(__m128d a, __m128d b)

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

R0	R1
(a0 > b0) ? 0xffffffffffffffff : 0x0	(a1 > b1) ? 0xffffffffffffffff : 0x0

__m128d _mm_cmpge_pd(__m128d a, __m128d b)

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

R0	R1
(a0 >= b0) ? 0xffffffffffffffff : 0x0	(a1 >= b1) ? 0xffffffffffffffff : 0x0

__m128d _mm_cmpord_pd(__m128d a, __m128d b)

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

R0	R1
(a0 ord b0) ? 0xffffffffffffffff : 0x0	(a1 ord b1) ? 0xffffffffffffffff : 0x0

__m128d _mm_cmpunord_pd(__m128d a, __m128d b)

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

R0	R1
(a0 unord b0) ? 0xffffffffffffffff : 0x0	(a1 unord b1) ? 0xffffffffffffffff : 0x0

__m128d _mm_cmpneq_pd (__m128d a, __m128d b)

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

R0	R1
(a0 != b0) ? 0xffffffffffffffff : 0x0	(a1 != b1) ? 0xffffffffffffffff : 0x0

__m128d _mm_cmpnlt_pd(__m128d a, __m128d b)

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

R0	R1
!(a0 < b0) ? 0xffffffffffffffff : 0x0	!(a1 < b1) ? 0xffffffffffffffff : 0x0

__m128d _mm_cmpnle_pd(__m128d a, __m128d b)

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

R0	R1
!(a0 <= b0) ? 0xffffffffffffffff : 0x0	!(a1 <= b1) ? 0xffffffffffffffff : 0x0

__m128d _mm_cmpngt_pd(__m128d a, __m128d b)

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

R0	R1
!(a0 > b0) ? 0xffffffffffffffff : 0x0	!(a1 > b1) ? 0xffffffffffffffff : 0x0

__m128d _mm_cmpnge_pd(__m128d a, __m128d b)

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

R0	R1
!(a0 >= b0) ? 0xffffffffffffffff : 0x0	!(a1 >= b1) ? 0xffffffffffffffff : 0x0

__m128d _mm_cmpeq_sd(__m128d a, __m128d b)

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

R0	R1
(a0 == b0) ? 0xffffffffffffffff : 0x0	a1

__m128d _mm_cmplt_sd(__m128d a, __m128d b)

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

R0	R1
(a0 < b0) ? 0xffffffffffffffff : 0x0	a1

__m128d _mm_cmple_sd(__m128d a, __m128d b)

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

R0	R1
(a0 <= b0) ? 0xffffffffffffffff : 0x0	a1

__m128d _mm_cmpgt_sd(__m128d a, __m128d b)

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

R0	R1
(a0 > b0) ? 0xffffffffffffffff : 0x0	a1

__m128d _mm_cmpge_sd(__m128d a, __m128d b)

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

R0	R1
(a0 >= b0) ? 0xffffffffffffffff : 0x0	a1

__m128d _mm_cmpord_sd(__m128d a, __m128d b)

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

R0	R1
(a0 ord b0) ? 0xffffffffffffffff : 0x0	a1

__m128d _mm_cmpunord_sd(__m128d a, __m128d b)

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

R0	R1
(a0 unord b0) ? 0xffffffffffffffff : 0x0	a1

__m128d _mm_cmpneq_sd(__m128d a, __m128d b)

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

R0	R1
(a0 != b0) ? 0xffffffffffffffff : 0x0	a1

__m128d _mm_cmpnlt_sd(__m128d a, __m128d b)

Compares the lower DP FP value of a and b for a not less than b. The upper DP FP value is passed through from a.

R0	R1
!(a0 < b0) ? 0xffffffffffffffff : 0x0	a1

__m128d _mm_cmpnle_sd(__m128d a, __m128d b)

Compares the lower DP FP value of a and b for a not less than or equal to b. The upper DP FP value is passed through from a.

R0	R1
!(a0 <= b0) ? 0xffffffffffffffff : 0x0	a1

__m128d _mm_cmpngt_sd(__m128d a, __m128d b)

Compares the lower DP FP value of a and b for a not greater than b. The upper DP FP value is passed through from a.

R0	R1
!(a0 > b0) ? 0xffffffffffffffff : 0x0	a1

__m128d _mm_cmpnge_sd(__m128d a, __m128d b)

Compares the lower DP FP value of a and b for a not greater than or equal to b. The upper DP FP value is passed through from a.

R0	R1
!(a0 >= b0) ? 0xffffffffffffffff : 0x0	a1

int _mm_comieq_sd(__m128d a, __m128d b)

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

R
(a0 == b0) ? 0x1 : 0x0

int _mm_comilt_sd(__m128d a, __m128d b)

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

R
(a0 < b0) ? 0x1 : 0x0

int _mm_comile_sd(__m128d a, __m128d b)

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

R
(a0 <= b0) ? 0x1 : 0x0

int _mm_comigt_sd(__m128d a, __m128d b)

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

R
(a0 > b0) ? 0x1 : 0x0

int _mm_comige_sd(__m128d a, __m128d b)

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

R
(a0 >= b0) ? 0x1 : 0x0

int _mm_comineq_sd(__m128d a, __m128d b)

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

R
(a0 != b0) ? 0x1 : 0x0

int _mm_ucomieq_sd(__m128d a, __m128d b)

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

R
(a0 == b0) ? 0x1 : 0x0

int _mm_ucomilt_sd(__m128d a, __m128d b)

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

R
(a0 < b0) ? 0x1 : 0x0

int _mm_ucomile_sd(__m128d a, __m128d b)

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

R
(a0 <= b0) ? 0x1 : 0x0

int _mm_ucomigt_sd(__m128d a, __m128d b)

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

R
(a0 > b0) ? 0x1 : 0x0

int _mm_ucomige_sd(__m128d a, __m128d b)

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

R
(a0 >= b0) ? 0x1 : 0x0

int _mm_ucomineq_sd(__m128d a, __m128d b)

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

R
(a0 != b0) ? 0x1 : 0x0

Floating-point Conversion Operations for Streaming SIMD Extensions 2

Each conversion intrinsic takes one data type and performs a conversion to a different type. Some conversions such as _mm_cvtpd_ps result in a loss of precision. The rounding mode used in such cases is determined by the value in the MXCSR

register. The default rounding mode is round-to-nearest. Note that the rounding mode used by the C and C++ languages when performing a type conversion is to truncate. The _mm_cvttpd_epi32 and _mm_cvttsd_si32 intrinsics use the truncate rounding mode regardless of the mode specified by the MXCSR register.

The conversion-operation intrinsics for Streaming SIMD Extensions 2 (SSE2) are listed in the following table followed by detailed descriptions.

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R, R0, R1, R2 and R3 represent the registers in which results are placed.

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

Intrinsic Name	Operation	Corresponding SSE2 Instruction
_mm_cvtpd_ps	Convert DP FP to SP FP	CVTPD2PS
_mm_cvtps_pd	Convert from SP FP to DP FP	CVTPS2PD
_mm_cvtepi32_pd	Convert lower integer values to DP FP	CVTDQ2PD
_mm_cvtpd_epi32	Convert DP FP values to integer values	CVTPD2DQ
_mm_cvtsd_si32	Convert lower DP FP value to integer value	CVTSD2SI
_mm_cvtsd_ss	Convert lower DP FP value to SP FP	CVTSD2SS
_mm_cvtsi32_sd	Convert signed integer value to DP FP	CVTSI2SD
_mm_cvtss_sd	Convert lower SP FP value to DP FP	CVTSS2SD
_mm_cvttpd_epi32	Convert DP FP values to signed integers	CVTTPD2DQ
_mm_cvttsd_si32	Convert lower DP FP to signed integer	CVTTSD2SI
_mm_cvtpd_pi32	Convert two DP FP values to signed integer values	CVTPD2PI
_mm_cvttpd_pi32	Convert two DP FP values to signed integer values using truncate	CVTTPD2PI
_mm_cvtpi32_pd	Convert two signed integer values to DP FP	CVTPI2PD
_mm_cvtsd_f64	Extract DP FP value from first vector element	None

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

Converts the two DP FP values of a to SP FP values.

R0	R1	R2	R3
(float) a0	(float) a1	0.0	0.0

```
__m128d _mm_cvtps_pd(__m128 a)
```

Converts the lower two SP FP values of a to DP FP values.

R0	R1
(double) a0	(double) a1

```
__m128d _mm_cvtepi32_pd(__m128i a)
```

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

R0	R1
(double) a0	(double) a1

```
__m128i _mm_cvtpd_epi32(__m128d a)
```

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

R0	R1	R2	R3
(int) a0	(int) a1	0x0	0x0

```
int _mm_cvtsd_si32(__m128d a)
```

Converts the lower DP FP value of a to a 32-bit signed integer value.

R
(int) a0

```
__m128 _mm_cvtsd_ss(__m128 a, __m128d b)
```

Converts the lower DP FP value of b to an SP FP value. The upper SP FP values in a are passed through.

R0	R1	R2	R3
(float) b0	a1	a2	a3

```
__m128d _mm_cvtsi32_sd(__m128d a, int b)
```

Converts the signed integer value in b to a DP FP value. The upper DP FP value in a is passed through.

R0	R1
(double) b	a1

__m128d _mm_cvtss_sd(__m128d a, __m128 b)

Converts the lower SP FP value of b to a DP FP value. The upper value DP FP value in a is passed through.

R0	R1
(double) b0	a1

__m128i _mm_cvttpd_epi32(__m128d a)

Converts the two DP FP values of a to 32-bit signed integers using truncate.

R0	R1	R2	R3
(int) a0	(int) a1	0x0	0x0

int _mm_cvttsd_si32(__m128d a)

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

R
(int) a0

__m64 _mm_cvtpd_pi32(__m128d a)

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

R0	R1
(int)a0	(int) a1

__m64 _mm_cvttpd_pi32(__m128d a)

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

R0	R1
(int)a0	(int) a1

__m128d _mm_cvtpi32_pd(__m64 a)

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

R0	R1
(double)a0	(double)a1

_mm_cvtsd_f64(__m128d a)

This intrinsic extracts a double precision floating point value from the first vector element of an __m128d. It does so in the most efficient manner possible in the context used. This intrinsic does not map to any specific SSE2 instruction.

Floating-point Load Operations for Streaming SIMD Extensions 2

The following load operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

The 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.

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R0 and R1 represent the registers in which results are placed.

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

The Double Complex code sample contains examples of how to use several of these intrinsics.

Intrinsic Name	Operation	Corresponding SSE2 Instruction
_mm_load_pd	Loads two DP FP values	MOVAPD
_mm_load1_pd	Loads a single DP FP value, copying to both elements	MOVSD + shuffling
_mm_loadr_pd	Loads two DP FP values in reverse order	MOVAPD + shuffling
_mm_loadu_pd	Loads two DP FP values	MOVUPD
_mm_load_sd	Loads a DP FP value, sets upper DP FP to zero	MOVSD
_mm_loadh_pd	Loads a DP FP value as the upper DP FP value of the result	MOVHPD
_mm_loadl_pd	Loads a DP FP value as the lower DP FP value of the result	MOVLPD

__m128d _mm_load_pd(double const*dp)

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

R0	R1
p[0]	p[1]

__m128d _mm_load1_pd(double const*dp)

Loads a single DP FP value, copying to both elements. The address p need not be 16-byte aligned.

R0	R1
*p	*p

__m128d _mm_loadr_pd(double const*dp)

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

R0	R1
p[1]	p[0]

__m128d _mm_loadu_pd(double const*dp)

Loads two DP FP values. The address p need not be 16-byte aligned.

R0	R1
p[0]	p[1]

__m128d _mm_load_sd(double const*dp)

Loads a DP FP value. The upper DP FP is set to zero. The address p need not be 16-byte aligned.

R0	R1
*p	0.0

__m128d _mm_loadh_pd(__m128d a, double const*dp)

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	R1
a0	*p

__m128d _mm_loadl_pd(__m128d a, double const*dp)

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	R1
*p	a1

Floating-point Set Operations for Streaming SIMD Extensions 2

The following `set` operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

The `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.

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R0 and R1 represent the registers in which results are placed.

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

Intrinsic Name	Operation	Corresponding SSE2 Instruction
_mm_set_sd	Sets lower DP FP value to w and upper to zero	Composite
_mm_set1_pd	Sets two DP FP valus to w	Composite
_mm_set_pd	Sets lower DP FP to x and upper to w	Composite
_mm_setr_pd	Sets lower DP FP to w and upper to x	Composite
_mm_setzero_pd	Sets two DP FP values to zero	XORPD
_mm_move_sd	Sets lower DP FP value to the lower DP FP value of `b`	MOVSD

```
__m128d _mm_set_sd(double w)
```

Sets the lower DP FP value to `w` and sets the upper DP FP value to zero.

R0	R1
w	0.0

```
__m128d _mm_set1_pd(double w)
```

Sets the 2 DP FP values to `w`.

R0	R1
w	W

__m128d _mm_set_pd(double w, double x)

Sets the lower DP FP value to x and sets the upper DP FP value to w.

R0	R1
x	W

__m128d _mm_setr_pd(double w, double x)

Sets the lower DP FP value to w and sets the upper DP FP value to x.

```
r0 := w
r1 := x
```

R0	R1
w	X

__m128d _mm_setzero_pd(void)

Sets the 2 DP FP values to zero.

R0	R1
0.0	0.0

__m128d _mm_move_sd(__m128d a, __m128d b)

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	R1
b0	A1

Floating-point Store Operations for Streaming SIMD Extensions 2

The following store operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

The store operations assign the initialized data to the address.

Details about each intrinsic follows the table below.

The detailed description of each intrinsic contains a table detailing the returns. In these tables, dp[n] is an access to the n element of the result.

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

The Double Complex code sample contains an example of how to use the _mm_store_pd intrinsic.

Intrinsic Name	Operation	Corresponding SSE2 Instruction
_mm_stream_pd	Store	MOVNTPD
_mm_store_sd	Stores lower DP FP value of a	MOVSD
_mm_store1_pd	Stores lower DP FP value of a twice	MOVAPD + shuffling
_mm_store_pd	Stores two DP FP values	MOVAPD
_mm_storeu_pd	Stores two DP FP values	MOVUPD
_mm_storer_pd	Stores two DP FP values in reverse order	MOVAPD + shuffling
_mm_storeh_pd	Stores upper DP FP value of a	MOVHPD
_mm_storel_pd	Stores lower DP FP value of a	MOVLPD

```
void _mm_store_sd(double *dp, __m128d a)
```

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

*dp
a0

```
void _mm_store1_pd(double *dp, __m128d a)
```

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

dp[0]	dp[1]
a0	a0

```
void _mm_store_pd(double *dp, __m128d a)
```

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

dp[0]	dp[1]
a0	a1

```
void _mm_storeu_pd(double *dp, __m128d a)
```

Stores two DP FP values. The address dp need not be 16-byte aligned.

dp[0]	dp[1]
a0	a1

```
void _mm_storer_pd(double *dp, __m128d a)
```

Stores two DP FP values in reverse order. The address `dp` must be 16-byte aligned.

dp[0]	dp[1]
a1	a0

```
void _mm_storeh_pd(double *dp, __m128d a)
```

Stores the upper DP FP value of `a`.

*dp
a1

```
void _mm_storel_pd(double *dp, __m128d a)
```

Stores the lower DP FP value of `a`.

*dp
a0

Integer Intrinsics

Integer Arithmetic Operations for Streaming SIMD Extensions 2

The integer arithmetic operations for Streaming SIMD Extensions 2 (SSE2) are listed in the following table followed by their descriptions. The floating point packed arithmetic intrinsics for SSE2 are listed in the Floating-point Arithmetic Operations topic.

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R, R0, R1...R15 represent the registers in which results are placed.

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

Intrinsic	Operation	Instruction
_mm_add_epi8	Addition	PADDB
_mm_add_epi16	Addition	PADDW
_mm_add_epi32	Addition	PADDD
_mm_add_si64	Addition	PADDQ
_mm_add_epi64	Addition	PADDQ

_mm_adds_epi8	Addition	PADDSB
_mm_adds_epi16	Addition	PADDSW
_mm_adds_epu8	Addition	PADDUSB
_mm_adds_epu16	Addition	PADDUSW
_mm_avg_epu8	Computes Average	PAVGB
_mm_avg_epu16	Computes Average	PAVGW
_mm_madd_epi16	Multiplication and Addition	PMADDWD
_mm_max_epi16	Computes Maxima	PMAXSW
_mm_max_epu8	Computes Maxima	PMAXUB
_mm_min_epi16	Computes Minima	PMINSW
_mm_min_epu8	Computes Minima	PMINUB
_mm_mulhi_epi16	Multiplication	PMULHW
_mm_mulhi_epu16	Multiplication	PMULHUW
_mm_mullo_epi16	Multiplication	PMULLW
_mm_mul_su32	Multiplication	PMULUDQ
_mm_mul_epu32	Multiplication	PMULUDQ
_mm_sad_epu8	Computes Difference/Adds	PSADBW
_mm_sub_epi8	Subtraction	PSUBB
_mm_sub_epi16	Subtraction	PSUBW
_mm_sub_epi32	Subtraction	PSUBD
_mm_sub_si64	Subtraction	PSUBQ
_mm_sub_epi64	Subtraction	PSUBQ
_mm_subs_epi8	Subtraction	PSUBSB
_mm_subs_epi16	Subtraction	PSUBSW
_mm_subs_epu8	Subtraction	PSUBUSB
_mm_subs_epu16	Subtraction	PSUBUSW

`__mm128i _mm_add_epi8(__m128i a, __m128i b)`

Adds the 16 signed or unsigned 8-bit integers in `a` to the 16 signed or unsigned 8-bit integers in `b`.

R0	R1	...	R15
a0 + b0	a1 + b1;	...	a15 + b15

```
__mm128i _mm_add_epi16(__m128i a, __m128i b)
```

Adds the 8 signed or unsigned 16-bit integers in `a` to the 8 signed or unsigned 16-bit integers in `b`.

R0	R1	...	R7
a0 + b0	a1 + b1	...	a7 + b7

```
__m128i _mm_add_epi32(__m128i a, __m128i b)
```

Adds the 4 signed or unsigned 32-bit integers in `a` to the 4 signed or unsigned 32-bit integers in `b`.

R0	R1	R2	R3
a0 + b0	a1 + b1	a2 + b2	a3 + b3

```
__m64 _mm_add_si64(__m64 a, __m64 b)
```

Adds the signed or unsigned 64-bit integer `a` to the signed or unsigned 64-bit integer `b`.

R0
a + b

```
__m128i _mm_add_epi64(__m128i a, __m128i b)
```

Adds the 2 signed or unsigned 64-bit integers in `a` to the 2 signed or unsigned 64-bit integers in `b`.

R0	R1
a0 + b0	a1 + b1

```
__m128i _mm_adds_epi8(__m128i a, __m128i b)
```

Adds the 16 signed 8-bit integers in `a` to the 16 signed 8-bit integers in `b` using saturating arithmetic.

R0	R1	...	R15
SignedSaturate (a0 + b0)	SignedSaturate (a1 + b1)	...	SignedSaturate (a15 + b15)

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

Adds the 8 signed 16-bit integers in a to the 8 signed 16-bit integers in b using saturating arithmetic.

R0	R1	...	R7
SignedSaturate (a0 + b0)	SignedSaturate (a1 + b1)	...	SignedSaturate (a7 + b7)

__m128i _mm_adds_epu8(__m128i a, __m128i b)

Adds the 16 unsigned 8-bit integers in a to the 16 unsigned 8-bit integers in b using saturating arithmetic.

R0	R1	...	R15
UnsignedSaturate (a0 + b0)	UnsignedSaturate (a1 + b1)	...	UnsignedSaturate (a15 + b15)

__m128i _mm_adds_epu16(__m128i a, __m128i b)

Adds the 8 unsigned 16-bit integers in a to the 8 unsigned 16-bit integers in b using saturating arithmetic.

R0	R1	...	R7
UnsignedSaturate (a0 + b0)	UnsignedSaturate (a1 + b1)	...	UnsignedSaturate (a7 + b7)

__m128i _mm_avg_epu8(__m128i a, __m128i b)

Computes the average of the 16 unsigned 8-bit integers in a and the 16 unsigned 8-bit integers in b and rounds.

R0	R1	...	R15
(a0 + b0) / 2	(a1 + b1) / 2	...	(a15 + b15) / 2

__m128i _mm_avg_epu16(__m128i a, __m128i b)

Computes the average of the 8 unsigned 16-bit integers in a and the 8 unsigned 16-bit integers in b and rounds.

R0	R1	...	R7
(a0 + b0) / 2	(a1 + b1) / 2	...	(a7 + b7) / 2

__m128i _mm_madd_epi16(__m128i a, __m128i b)

Multiplies the 8 signed 16-bit integers from a by the 8 signed 16-bit integers from b. Adds the signed 32-bit integer results pairwise and packs the 4 signed 32-bit integer results.

R0	R1	R2	R3
(a0 * b0) + (a1 * b1)	(a2 * b2) + (a3 * b3)	(a4 * b4) + (a5 * b5)	(a6 * b6) + (a7 * b7)

__m128i _mm_max_epi16(__m128i a, __m128i b)

Computes the pairwise maxima of the 8 signed 16-bit integers from a and the 8 signed 16-bit integers from b.

R0	R1	...	R7
max(a0, b0)	max(a1, b1)	...	max(a7, b7)

__m128i _mm_max_epu8(__m128i a, __m128i b)

Computes the pairwise maxima of the 16 unsigned 8-bit integers from a and the 16 unsigned 8-bit integers from b.

R0	R1	...	R15
max(a0, b0)	max(a1, b1)	...	max(a15, b15)

__m128i _mm_min_epi16(__m128i a, __m128i b)

Computes the pairwise minima of the 8 signed 16-bit integers from a and the 8 signed 16-bit integers from b.

R0	R1	...	R7
min(a0, b0)	min(a1, b1)	...	min(a7, b7)

__m128i _mm_min_epu8(__m128i a, __m128i b)

Computes the pairwise minima of the 16 unsigned 8-bit integers from a and the 16 unsigned 8-bit integers from b.

R0	R1	...	R15
min(a0, b0)	min(a1, b1)	...	min(a15, b15)

__m128i _mm_mulhi_epi16(__m128i a, __m128i b)

Multiplies the 8 signed 16-bit integers from a by the 8 signed 16-bit integers from b. Packs the upper 16-bits of the 8 signed 32-bit results.

R0	R1	...	R7
(a0 * b0)[31:16]	(a1 * b1)[31:16]	...	(a7 * b7)[31:16]

__m128i _mm_mulhi_epu16(__m128i a, __m128i b)

Multiplies the 8 unsigned 16-bit integers from a by the 8 unsigned 16-bit integers from b. Packs the upper 16-bits of the 8 unsigned 32-bit results.

R0	R1	...	R7
(a0 * b0)[31:16]	(a1 * b1)[31:16]	...	(a7 * b7)[31:16]

__m128i_mm_mullo_epi16(__m128i a, __m128i b)

Multiplies the 8 signed or unsigned 16-bit integers from a by the 8 signed or unsigned 16-bit integers from b. Packs the lower 16-bits of the 8 signed or unsigned 32-bit results.

R0	R1	...	R7
(a0 * b0)[15:0]	(a1 * b1)[15:0]	...	(a7 * b7)[15:0]

__m64 _mm_mul_su32(__m64 a, __m64 b)

Multiplies the lower 32-bit integer from a by the lower 32-bit integer from b, and returns the 64-bit integer result.

R0
a0 * b0

__m128i _mm_mul_epu32(__m128i a, __m128i b)

Multiplies 2 unsigned 32-bit integers from a by 2 unsigned 32-bit integers from b. Packs the 2 unsigned 64-bit integer results.

R0	R1
a0 * b0	a2 * b2

__m128i _mm_sad_epu8(__m128i a, __m128i b)

Computes the absolute difference of the 16 unsigned 8-bit integers from a and the 16 unsigned 8-bit integers from b. Sums the upper 8 differences and lower 8 differences, and packs the resulting 2 unsigned 16-bit integers into the upper and lower 64-bit elements.

R0	R1	R2	R3	R4	R5	R6	R7
abs(a0 - b0) + abs(a1 - b1) +...+ abs(a7 - b7)	0x0	0x0	0x0	abs(a8 - b8) + abs(a9 - b9) +...+ abs(a15 - b15)	0x0	0x0	0x0

__m128i _mm_sub_epi8(__m128i a, __m128i b)

Subtracts the 16 signed or unsigned 8-bit integers of b from the 16 signed or unsigned 8-bit integers of a.

R0	R1	...	R15
a0 - b0	a1 – b1	...	a15 - b15

__m128i_mm_sub_epi16(__m128i a, __m128i b)

Subtracts the 8 signed or unsigned 16-bit integers of b from the 8 signed or unsigned 16-bit integers of a.

R0	R1	...	R7
a0 - b0	a1 – b1	...	a7 - b7

__m128i _mm_sub_epi32(__m128i a, __m128i b)

Subtracts the 4 signed or unsigned 32-bit integers of b from the 4 signed or unsigned 32-bit integers of a.

R0	R1	R2	R3
a0 - b0	a1 – b1	a2 - b2	a3 - b3

__m64 _mm_sub_si64 (__m64 a, __m64 b)

Subtracts the signed or unsigned 64-bit integer b from the signed or unsigned 64-bit integer a.

R
a - b

__m128i _mm_sub_epi64(__m128i a, __m128i b)

Subtracts the 2 signed or unsigned 64-bit integers in b from the 2 signed or unsigned 64-bit integers in a.

R0	R1
a0 - b0	a1 – b1

__m128i _mm_subs_epi8(__m128i a, __m128i b)

Subtracts the 16 signed 8-bit integers of b from the 16 signed 8-bit integers of a using saturating arithmetic.

R0	R1	...	R15
SignedSaturate (a0 - b0)	SignedSaturate (a1 - b1)	...	SignedSaturate (a15 - b15)

__m128i _mm_subs_epi16(__m128i a, __m128i b)

Subtracts the 8 signed 16-bit integers of b from the 8 signed 16-bit integers of a using saturating arithmetic.

R0	R1	...	R15
SignedSaturate (a0 - b0)	SignedSaturate (a1 - b1)	...	SignedSaturate (a7 - b7)

__m128i _mm_subs_epu8 (__m128i a, __m128i b)

Subtracts the 16 unsigned 8-bit integers of b from the 16 unsigned 8-bit integers of a using saturating arithmetic.

R0	R1	...	R15
UnsignedSaturate (a0 - b0)	UnsignedSaturate (a1 - b1)	...	UnsignedSaturate (a15 - b15)

__m128i _mm_subs_epu16 (__m128i a, __m128i b)

Subtracts the 8 unsigned 16-bit integers of b from the 8 unsigned 16-bit integers of a using saturating arithmetic.

R0	R1	...	R7
UnsignedSaturate (a0 - b0)	UnsignedSaturate (a1 - b1)	...	UnsignedSaturate (a7 - b7)

Integer Logical Operations for Streaming SIMD Extensions 2

The following four logical-operation intrinsics and their respective instructions are functional as part of Streaming SIMD Extensions 2 (SSE2).

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in register R. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic.

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

Intrinsic Name	Operation	Corresponding SSE2 Instruction
_mm_and_si128	Computes AND	PAND
_mm_andnot_si128	Computes AND and NOT	PANDN
_mm_or_si128	Computes OR	POR
_mm_xor_si128	Computes XOR	PXOR

`__m128i _mm_and_si128(__m128i a, __m128i b)`

Computes the bitwise AND of the 128-bit value in `a` and the 128-bit value in `b`.

R0
a & b

`__m128i _mm_andnot_si128(__m128i a, __m128i b)`

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

R0
(~a) & b

`__m128i _mm_or_si128(__m128i a, __m128i b)`

Computes the bitwise OR of the 128-bit value in `a` and the 128-bit value in `b`.

R0
a | b

`__m128i _mm_xor_si128(__m128i a, __m128i b)`

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

R0
a ^ b

Integer Shift Operations for Streaming SIMD Extensions 2

The shift-operation intrinsics for Streaming SIMD Extensions 2 (SSE2) and the description for each are listed in the following table.

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R and R0-R7. R and R0 R7 each represent one of the pieces of the result register.

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

Note

The `count` argument is one shift count that applies to all elements of the operand being shifted. It is not a vector shift count that shifts each element by a different amount.

Intrinsic	Operation	Shift Type	Corresponding Instruction
_mm_slli_si128	Shift left	Logical	PSLLDQ
_mm_slli_epi16	Shift left	Logical	PSLLW
_mm_sll_epi16	Shift left	Logical	PSLLW
_mm_slli_epi32	Shift left	Logical	PSLLD
_mm_sll_epi32	Shift left	Logical	PSLLD
_mm_slli_epi64	Shift left	Logical	PSLLQ
_mm_sll_epi64	Shift left	Logical	PSLLQ
_mm_srai_epi16	Shift right	Arithmetic	PSRAW
_mm_sra_epi16	Shift right	Arithmetic	PSRAW
_mm_srai_epi32	Shift right	Arithmetic	PSRAD
_mm_sra_epi32	Shift right	Arithmetic	PSRAD
_mm_srli_si128	Shift right	Logical	PSRLDQ
_mm_srli_epi16	Shift right	Logical	PSRLW
_mm_srl_epi16	Shift right	Logical	PSRLW
_mm_srli_epi32	Shift right	Logical	PSRLD
_mm_srl_epi32	Shift right	Logical	PSRLD
_mm_srli_epi64	Shift right	Logical	PSRLQ
_mm_srl_epi64	Shift right	Logical	PSRLQ

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

Shifts the 128-bit value in `a` left by `imm` bytes while shifting in zeros. `imm` must be an immediate.

R
a << (imm * 8)

```
__m128i _mm_slli_epi16(__m128i a, int count)
```

```
Shifts the 8 signed or unsigned 16-bit integers in a left by count bits
while shifting in zeros.
```

R0	R1	...	R7
a0 << count	a1 << count	...	a7 << count

__m128i _mm_sll_epi16(__m128i a, __m128i count)

Shifts the 8 signed or unsigned 16-bit integers in a left by count bits while shifting in zeros.

R0	R1	...	R7
a0 << count	a1 << count	...	a7 << count

__m128i _mm_slli_epi32(__m128i a, int count)

Shifts the 4 signed or unsigned 32-bit integers in a left by count bits while shifting in zeros.

R0	R1	R2	R3
a0 << count	a1 << count	a2 << count	a3 << count

__m128i _mm_sll_epi32(__m128i a, __m128i count)

Shifts the 4 signed or unsigned 32-bit integers in a left by count bits while shifting in zeros.

R0	R1	R2	R3
a0 << count	a1 << count	a2 << count	a3 << count

__m128i _mm_slli_epi64(__m128i a, int count)

Shifts the 2 signed or unsigned 64-bit integers in a left by count bits while shifting in zeros.

R0	R1
a0 << count	a1 << count

__m128i _mm_sll_epi64(__m128i a, __m128i count)

Shifts the 2 signed or unsigned 64-bit integers in a left by count bits while shifting in zeros.

R0	R1
a0 << count	a1 << count

__m128i _mm_srai_epi16(__m128i a, int count)

Shifts the 8 signed 16-bit integers in a right by count bits while shifting in the sign bit.

R0	R1	...	R7
a0 >> count	a1 >> count	...	a7 >> count

__m128i _mm_sra_epi16(__m128i a, __m128i count)

Shifts the 8 signed 16-bit integers in a right by count bits while shifting in the sign bit.

R0	R1	...	R7
a0 >> count	a1 >> count	...	a7 >> count

__m128i _mm_srai_epi32(__m128i a, int count)

Shifts the 4 signed 32-bit integers in a right by count bits while shifting in the sign bit.

R0	R1	R2	R3
a0 >> count	a1 >> count	a2 >> count	a3 >> count

__m128i _mm_sra_epi32(__m128i a, __m128i count)

Shifts the 4 signed 32-bit integers in a right by count bits while shifting in the sign bit.

R0	R1	R2	R3
a0 >> count	a1 >> count	a2 >> count	a3 >> count

__m128i _mm_srli_si128(__m128i a, int imm)

Shifts the 128-bit value in a right by imm bytes while shifting in zeros. imm must be an immediate.

R
srl(a, imm*8)

__m128i _mm_srli_epi16(__m128i a, int count)

Shifts the 8 signed or unsigned 16-bit integers in a right by count bits while shifting in zeros.

R0	R1	...	R7
srl(a0, count)	srl(a1, count)	...	srl(a7, count)

__m128i _mm_srl_epi16(__m128i a, __m128i count)

Shifts the 8 signed or unsigned 16-bit integers in a right by count bits while shifting in zeros.

R0	R1	...	R7
srl(a0, count)	srl(a1, count)	...	srl(a7, count)

__m128i _mm_srli_epi32(__m128i a, int count)

Shifts the 4 signed or unsigned 32-bit integers in a right by count bits while shifting in zeros.

R0	R1	R2	R3
srl(a0, count)	srl(a1, count)	srl(a2, count)	srl(a3, count)

__m128i _mm_srl_epi32(__m128i a, __m128i count)

Shifts the 4 signed or unsigned 32-bit integers in a right by count bits while shifting in zeros.

R0	R1	R2	R3
srl(a0, count)	srl(a1, count)	srl(a2, count)	srl(a3, count)

__m128i _mm_srli_epi64(__m128i a, int count)

Shifts the 2 signed or unsigned 64-bit integers in a right by count bits while shifting in zeros.

R0	R1
srl(a0, count)	srl(a1, count)

__m128i _mm_srl_epi64(__m128i a, __m128i count)

Shifts the 2 signed or unsigned 64-bit integers in a right by count bits while shifting in zeros.

R0	R1
srl(a0, count)	srl(a1, count)

Integer Comparison Operations for Streaming SIMD Extensions 2

The comparison intrinsics for Streaming SIMD Extensions 2 (SSE2) and descriptions for each are listed in the following table.

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R, R0, R1...R15 represent the registers in which results are placed.

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

Intrinsic Name	Operation	Instruction
_mm_cmpeq_epi8	Equality	PCMPEQB
_mm_cmpeq_epi16	Equality	PCMPEQW

_mm_cmpeq_epi32	Equality	PCMPEQD
_mm_cmpgt_epi8	Greater Than	PCMPGTB
_mm_cmpgt_epi16	Greater Than	PCMPGTW
_mm_cmpgt_epi32	Greater Than	PCMPGTD
_mm_cmplt_epi8	Less Than	PCMPGTBr
_mm_cmplt_epi16	Less Than	PCMPGTWr
_mm_cmplt_epi32	Less Than	PCMPGTDr

__m128i _mm_cmpeq_epi8(__m128i a, __m128i b)

Compares the 16 signed or unsigned 8-bit integers in a and the 16 signed or unsigned 8-bit integers in b for equality.

R0	R1	...	R15
(a0 == b0) ? 0xff : 0x0	(a1 == b1) ? 0xff : 0x0	...	(a15 == b15) ? 0xff : 0x0

__m128i _mm_cmpeq_epi16(__m128i a, __m128i b)

Compares the 8 signed or unsigned 16-bit integers in a and the 8 signed or unsigned 16-bit integers in b for equality.

R0	R1	...	R7
(a0 == b0) ? 0xffff : 0x0	(a1 == b1) ? 0xffff : 0x0	...	(a7 == b7) ? 0xffff : 0x0

__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)

Compares the 4 signed or unsigned 32-bit integers in a and the 4 signed or unsigned 32-bit integers in b for equality.

R0	R1	R2	R3
(a0 == b0) ? 0xffffffff : 0x0	(a1 == b1) ? 0xffffffff : 0x0	(a2 == b2) ? 0xffffffff : 0x0	(a3 == b3) ? 0xffffffff : 0x0

__m128i _mm_cmpgt_epi8(__m128i a, __m128i b)

Compares the 16 signed 8-bit integers in a and the 16 signed 8-bit integers in b for greater than.

R0	R1	...	R15
(a0 > b0) ? 0xff : 0x0	(a1 > b1) ? 0xff : 0x0	...	(a15 > b15) ? 0xff : 0x0

__m128i _mm_cmpgt_epi16(__m128i a, __m128i b)

Compares the 8 signed 16-bit integers in a and the 8 signed 16-bit integers in b for greater than.

R0	R1	...	R7
(a0 > b0) ? 0xffff : 0x0	(a1 > b1) ? 0xffff : 0x0	...	(a7 > b7) ? 0xffff : 0x0

__m128i _mm_cmpgt_epi32(__m128i a, __m128i b)

Compares the 4 signed 32-bit integers in a and the 4 signed 32-bit integers in b for greater than.

R0	R1	R2	R3
(a0 > b0) ? 0xffffffff : 0x0	(a1 > b1) ? 0xffffffff : 0x0	(a2 > b2) ? 0xffffffff : 0x0	(a3 > b3) ? 0xffffffff : 0x0

__m128i _mm_cmplt_epi8(__m128i a, __m128i b)

Compares the 16 signed 8-bit integers in a and the 16 signed 8-bit integers in b for less than.

R0	R1	...	R15
(a0 < b0) ? 0xff : 0x0	(a1 < b1) ? 0xff : 0x0	...	(a15 < b15) ? 0xff : 0x0

__m128i _mm_cmplt_epi16(__m128i a, __m128i b)

Compares the 8 signed 16-bit integers in a and the 8 signed 16-bit integers in b for less than.

R0	R1	...	R7
(a0 < b0) ? 0xffff : 0x0	(a1 < b1) ? 0xffff : 0x0	...	(a7 < b7) ? 0xffff : 0x0

__m128i _mm_cmplt_epi32(__m128i a, __m128i b)

Compares the 4 signed 32-bit integers in a and the 4 signed 32-bit integers in b for less than.

R0	R1	R2	R3
(a0 < b0) ? 0xffffffff : 0x0	(a1 < b1) ? 0xffffffff : 0x0	(a2 < b2) ? 0xffffffff : 0x0	(a3 < b3) ? 0xffffffff : 0x0

Integer Conversion Operations for Streaming SIMD Extensions 2

The following conversion intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R, R0, R1, R2 and R3 represent the registers in which results are placed.

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

Intrinsic Name	Operation	Instruction
_mm_cvtsi64_sd	Convert and pass through	CVTSI2SD
_mm_cvtsd_si64	Convert according to rounding	CVTSD2SI
_mm_cvttsd_si64	Convert using truncation	CVTTSD2SI
_mm_cvtepi32_ps	Convert to SP FP	None
_mm_cvtps_epi32	Convert from SP FP	None
_mm_cvttps_epi32	Convert from SP FP using truncate	None

```
__m128d _mm_cvtsi64_sd(__m128d a, __int64 b)
```

Converts the signed 64-bit integer value in `b` to a DP FP value. The upper DP FP value in `a` is passed through.

R0	R1
(double)b	a1

```
__int64 _mm_cvtsd_si64(__m128d a)
```

Converts the lower DP FP value of `a` to a 64-bit signed integer value according to the current rounding mode.

R
(__int64) a0

```
__int64 _mm_cvttsd_si64(__m128d a)
```

Converts the lower DP FP value of `a` to a 64-bit signed integer value using truncation.

R
(__int64) a0

__m128 _mm_cvtepi32_ps(__m128i a)

Converts the 4 signed 32-bit integer values of a to SP FP values.

R0	R1	R2	R3
(float) a0	(float) a1	(float) a2	(float) a3

__m128i _mm_cvtps_epi32(__m128 a)

Converts the 4 SP FP values of a to signed 32-bit integer values.

R0	R1	R2	R3
(int) a0	(int) a1	(int) a2	(int) a3

__m128i _mm_cvttps_epi32(__m128 a)

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

R0	R1	R2	R3
(int) a0	(int) a1	(int) a2	(int) a3

Integer Move Operations for Streaming SIMD Extensions 2

The following conversion intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R, R0, R1, R2 and R3 represent the registers in which results are placed.

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

Intrinsic Name	Operation	Instruction
_mm_cvtsi32_si128	Move and zero	MOVD
_mm_cvtsi64_si128	Move and zero	MOVQ
_mm_cvtsi128_si32	Move lowest 32 bits	MOVD
_mm_cvtsi128_si64	Move lowest 64 bits	MOVQ

__m128i _mm_cvtsi32_si128(int a)

Moves 32-bit integer a to the least significant 32 bits of an __m128i object. Zeroes the upper 96 bits of the __m128i object.

R0	R1	R2	R3
a	0x0	0x0	0x0

__m128i _mm_cvtsi64_si128(__int64 a)

Moves 64-bit integer a to the lower 64 bits of an __m128i object, zeroing the upper bits.

R0	R1
a	0x0

int _mm_cvtsi128_si32(__m128i a)

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

R
a0

__int64 _mm_cvtsi128_si64(__m128i a)

Moves the lower 64 bits of a to a 64-bit integer.

R
a0

Integer Load Operations for Streaming SIMD Extensions 2

The following load operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R, R0 and R1 represent the registers in which results are placed.

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

Intrinsic Name	Operation	Instruction
mm_load_si128	Load	MOVDQA
_mm_loadu_si128	Load	MOVDQU

_mm_loadl_epi64	Load and zero	MOVQ

__m128i _mm_load_si128(__m128i const*p)

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

R
*p

__m128i _mm_loadu_si128(__m128i const*p)

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

R
*p

__m128i _mm_loadl_epi64(__m128i const*p)

Load the lower 64 bits of the value pointed to by p into the lower 64 bits of the result, zeroing the upper 64 bits of the result.

R0	R1
*p[63:0]	0x0

Integer Set Operations for SSE2

The following set operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

Details about each intrinsic follows the table below.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R, R0, R1...R15 represent the registers in which results are placed.

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

Intrinsic Name	Operation	Corresponding SSE2 Instruction
_mm_set_epi64	Set two integer values	
_mm_set_epi32	Set four integer values	
_mm_set_epi16	Set eight integer values	

_mm_set_epi8	Set sixteen integer values	
_mm_set1_epi64	Set two integer values	
_mm_set1_epi32	Set four integer values	
_mm_set1_epi16	Set eight integer values	
_mm_set1_epi8	Set sixteen integer values	
_mm_setr_epi64	Set two integer values in reverse order	
_mm_setr_epi32	Set four integer values in reverse order	
_mm_setr_epi16	Set eight integer values in reverse order	
_mm_setr_epi8	Set sixteen integer values in reverse order	
_mm_setzero_si128	Set to zero	

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

Sets the 2 64-bit integer values.

R0	R1
q0	q1

```
__m128i _mm_set_epi32(int i3, int i2, int i1, int i0)
```

Sets the 4 signed 32-bit integer values.

R0	R1	R2	R3
i0	i1	i2	i3

```
__m128i _mm_set_epi16(short w7, short w6, short w5, short w4, short w3,
short w2, short w1, short w0)
```

Sets the 8 signed 16-bit integer values.

R0	R1	...	R7
w0	w1	...	w7

```
__m128i _mm_set_epi8(char b15, char b14, char b13, char b12, char b11,
char b10, char b9, char b8, char b7, char b6, char b5, char b4, char b3,
char b2, char b1, char b0)
```

Sets the 16 signed 8-bit integer values.

R0	R1	...	R15
b0	b1	...	b15

__m128i _mm_set1_epi64(__m64 q)

Sets the 2 64-bit integer values to q.

R0	R1
q	q

__m128i _mm_set1_epi32(int i)

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

R0	R1	R2	R3
i	i	i	i

__m128i _mm_set1_epi16(short w)

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

R0	R1	...	R7
w	w	w	w

__m128i _mm_set1_epi8(char b)

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

R0	R1	...	R15
b	b	b	b

__m128i _mm_setr_epi64(__m64 q0, __m64 q1)

Sets the 2 64-bit integer values in reverse order.

R0	R1
q0	q1

__m128i _mm_setr_epi32(int i0, int i1, int i2, int i3)

Sets the 4 signed 32-bit integer values in reverse order.

R0	R1	R2	R3
i0	i1	i2	i3

__m128i _mm_setr_epi16(short w0, short w1, short w2, short w3, short w4, short w5, short w6, short w7)

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

R0	R1	...	R7
w0	w1	...	w7

```
__m128i _mm_setr_epi8(char b15, char b14, char b13, char b12, char b11,
char b10, char b9, char b8, char b7, char b6, char b5, char b4, char b3,
char b2, char b1, char b0)
```

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

R0	R1	...	R15
b0	b1	...	b15

```
__m128i _mm_setzero_si128()
```

Sets the 128-bit value to zero.

R
0x0

Integer Store Operations for Streaming SIMD Extensions 2

The following `store` operation intrinsics and their respective instructions are functional in the Streaming SIMD Extensions 2 (SSE2).

The detailed description of each intrinsic contains a table detailing the returns. In these tables, p is an access to the result.

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

Intrinsic Name	Operation	Corresponding SSE2 Instruction
_mm_stream_si128	Store	MOVNTDQ
_mm_stream_si32	Store	MOVNTI
_mm_store_si128	Store	MOVDQA
_mm_storeu_si128	Store	MOVDQU
_mm_maskmoveu_si128	Conditional store	MASKMOVDQU
_mm_storel_epi64	Store lowest	MOVQ

```
void _mm_stream_si128(__m128i *p, __m128i a)
```

Stores the data in `a` to the address `p` without polluting the caches. If the cache line containing address `p` is already in the cache, the cache will be updated. Address `p` must be 16 byte aligned.

*p
a

```
void _mm_stream_si32(int *p, int a)
```

Stores the data in `a` to the address `p` without polluting the caches. If the cache line containing address `p` is already in the cache, the cache will be updated.

*p
a

```
void _mm_store_si128(__m128i *p, __m128i b)
```

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

*p
a

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

Stores 128-bit value. Address `p` need not be 16-byte aligned.

*p
a

```
void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p)
```

Conditionally store byte elements of `d` to address `p`. The high bit of each byte in the selector `n` determines whether the corresponding byte in `d` will be stored. Address `p` need not be 16-byte aligned.

if (n0[7])	if (n1[7]	...	if (n15[7])
p[0] := d0	p[1] := d1	...	p[15] := d15

```
void _mm_storel_epi64(__m128i *p, __m128i a)
```

Stores the lower 64 bits of the value pointed to by `p`.

*p[63:0]
a0

Miscellaneous Functions and Intrinsics

Cacheability Support Operations for Streaming SIMD Extensions 2

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

Intrinsic Name	Operation	Corresponding SSE2 Instruction
_mm_stream_pd	Store	MOVNTPD
_mm_stream_si128	Store	MOVNTDQ
_mm_stream_si32	Store	MOVNTI
_mm_clflush	Flush	CLFLUSH
_mm_lfence	Guarantee visibility	LFENCE
_mm_mfence	Guarantee visibility	MFENCE

```
void _mm_stream_pd(double *p, __m128d a)
```

Stores the data in `a` to the address `p` without polluting caches. The address `p` must be 16-byte aligned. If the cache line containing address `p` is already in the cache, the cache will be updated.

```
p[0] := a0
p[1] := a1
```

p[0]	p[1]
a0	a1

```
void _mm_stream_si128(__m128i *p, __m128i a)
```

Stores the data in `a` to the address `p` without polluting the caches. If the cache line containing address `p` is already in the cache, the cache will be updated. Address `p` must be 16-byte aligned.

*p
a

```
void _mm_stream_si32(int *p, int a)
```

Stores the data in `a` to the address `p` without polluting the caches. If the cache line containing address `p` is already in the cache, the cache will be updated.

*p

a

`void _mm_clflush(void const*p)`

Cache line containing `p` is flushed and invalidated from all caches in the coherency domain.

`void _mm_lfence(void)`

Guarantees that every load instruction that precedes, in program order, the load fence instruction is globally visible before any load instruction which follows the fence in program order.

`void _mm_mfence(void)`

Guarantees that every memory access that precedes, in program order, the memory fence instruction is globally visible before any memory instruction which follows the fence in program order.

Miscellaneous Operations for Streaming SIMD Extensions 2

The miscellaneous intrinsics for Streaming SIMD Extensions 2 (SSE2) are listed in the following table followed by their descriptions.

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

Intrinsic	Operation	Corresponding Instruction
_mm_packs_epi16	Packed Saturation	PACKSSWB
_mm_packs_epi32	Packed Saturation	PACKSSDW
_mm_packus_epi16	Packed Saturation	PACKUSWB
_mm_extract_epi16	Extraction	PEXTRW
_mm_insert_epi16	Insertion	PINSRW
_mm_movemask_epi8	Mask Creation	PMOVMSKB
_mm_shuffle_epi32	Shuffle	PSHUFD
_mm_shufflehi_epi16	Shuffle	PSHUFHW
_mm_shufflelo_epi16	Shuffle	PSHUFLW
_mm_unpackhi_epi8	Interleave	PUNPCKHBW
_mm_unpackhi_epi16	Interleave	PUNPCKHWD
_mm_unpackhi_epi32	Interleave	PUNPCKHDQ
_mm_unpackhi_epi64	Interleave	PUNPCKHQDQ
_mm_unpacklo_epi8	Interleave	PUNPCKLBW

_mm_unpacklo_epi16	Interleave	PUNPCKLWD
_mm_unpacklo_epi32	Interleave	PUNPCKLDQ
_mm_unpacklo_epi64	Interleave	PUNPCKLQDQ
_mm_movepi64_pi64	Move	MOVDQ2Q
_mm_movpi64_epi64	Move	MOVDQ2Q
_mm_move_epi64	Move	MOVQ
_mm_unpackhi_pd	Interleave	UNPCKHPD
_mm_unpacklo_pd	Interleave	UNPCKLPD
_mm_movemask_pd	Create mask	MOVMSKPD
_mm_shuffle_pd	Select values	SHUFPD

__m128i _mm_packs_epi16(__m128i a, __m128i b)

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

R0	...	R7	R8	...	R15
Signed Saturate(a0)	...	Signed Saturate(a7)	Signed Saturate(b0)	...	Signed Saturate(b7)

__m128i _mm_packs_epi32(__m128i a, __m128i b)

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

R0	...	R3	R4	...	R7
Signed Saturate(a0)	...	Signed Saturate(a3)	Signed Saturate(b0)	...	Signed Saturate(b3)

__m128i _mm_packus_epi16(__m128i a, __m128i b)

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

R0	...	R7	R8	...	R15
Unsigned Saturate(a0)	...	Unsigned Saturate(a7)	Unsigned Saturate(b0)	...	Unsigned Saturate(b15)

int _mm_extract_epi16(__m128i a, int imm)

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

R0
(imm == 0) ? a0: ((imm == 1) ? a1: ... (imm==7) ? a7)

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

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

R0	R1	...	R7
(imm == 0) ? b : a0;	(imm == 1) ? b : a1;	...	(imm == 7) ? b : a7;

int _mm_movemask_epi8(__m128i a)

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

R0
a15[7] << 15 | a14[7] << 14 | ... a1[7] << 1 | a0[7]

__m128i _mm_shuffle_epi32(__m128i a, int imm)

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

__m128i _mm_shufflehi_epi16(__m128i a, int imm)

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

__m128i _mm_shufflelo_epi16(__m128i a, int imm)

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

__m128i _mm_unpackhi_epi8(__m128i a, __m128i b)

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

R0	R1	R2	R3	...	R14	R15
a8	b8	a9	b9	...	a15	b15

__m128i _mm_unpackhi_epi16(__m128i a, __m128i b)

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

R0	R1	R2	R3	R4	R5	R6	R7
a4	b4	a5	b5	a6	b6	a7	b7

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

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

R0	R1	R2	R3
a2	b2	a3	b3

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

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

R0	R1
a1	b1

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

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

R0	R1	R2	R3	...	R14	R15
a0	b0	a1	b1	...	a7	b7

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

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

R0	R1	R2	R3	R4	R5	R6	R7
a0	b0	a1	b1	a2	b2	a3	b3

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

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

R0	R1	R2	R3
a0	b0	a1	b1

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

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

R0	R1
a0	b0

`__m64 _mm_movepi64_pi64(__m128i a)`

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

R0
a0

`__128i _mm_movpi64_pi64(__m64 a)`

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

R0	R1
a0	0X0

`__128i _mm_move_epi64(__m128i a)`

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

R0	R1
a0	0X0

`__m128d _mm_unpackhi_pd(__m128d a, __m128d b)`

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

R0	R1
a1	b1

`__m128d _mm_unpacklo_pd(__m128d a, __m128d b)`

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

R0	R1
a0	b0

`int _mm_movemask_pd(__m128d a)`

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)`

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

Intrinsics for Casting Support

This version of the Intel® C++ Compiler supports casting between various SP, DP, and INT vector types. These intrinsics do not convert values; they change one data type to another without changing the value.

The intrinsics for casting support do not correspond to any Streaming SIMD Extensions 2 (SSE2) instructions.

```
 __m128  _mm_castpd_ps(__m128d in);

__m128i _mm_castpd_si128(__m128d in);

__m128d _mm_castps_pd(__m128 in);

__m128i _mm_castps_si128(__m128 in);

__m128  _mm_castsi128_ps(__m128i in);

__m128d _mm_castsi128_pd(__m128i in);
```

Pause Intrinsic for Streaming SIMD Extensions 2

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

```
void _mm_pause(void)
```

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

`PAUSE` Intrinsic

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

Example of loop with the PAUSE instruction:

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

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

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

Critical Section

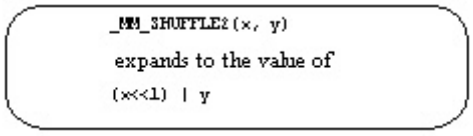
```
// critical section code
mov A, 0 ; Release lock
jmp continue
spin loop: pause;
// spin-loop hint
cmp 0, A ;
// check lock availability
jne spin loop
jmp get lock
// continue: other code
```

Note that the first branch is predicted to fall-through to the critical section in anticipation of successfully gaining access to the lock. It is highly recommended that all spin-wait loops include the PAUSE instruction. Since PAUSE is backwards compatible to all existing IA-32 processor generations, a test for processor type (a CPUID test) is not needed. All legacy processors will execute PAUSE as a NOP, but in processors which use the PAUSE as a hint there can be significant performance benefit.

Macro Function for Shuffle

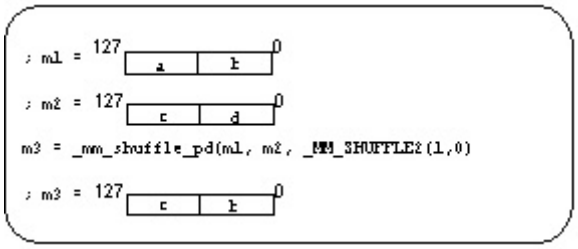
The Streaming SIMD Extensions 2 (SSE2) provide a macro function to help create constants that describe shuffle operations. The macro takes two small integers (in the range of 0 to 1) and combines them into an 2-bit immediate value used by the SHUFPD instruction. See the following example.

Shuffle Function Macro



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

View of Original and Result Words with Shuffle Function Macro


; 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

Streaming SIMD Extensions 3

Overview: Streaming SIMD Extensions 3

The Intel C++ intrinsics listed in this section are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3). They will not function correctly on other IA-32 processors. New SSE3 intrinsics include:

- Floating-point Vector Intrinsics
- Integer Vector Intrinsics
- Miscellaneous Intrinsics
- Macro Functions

The prototypes for these intrinsics are in the `pmmintrin.h` header file.


Note

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

Integer Vector Intrinsics for Streaming SIMD Extensions 3

The integer vector intrinsic listed here is designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3).

The prototype for this intrinsic is in the `pmmintrin.h` header file.

R represents the register into which the returns are placed.

```
__m128i _mm_lddqu_si128(__m128i const *p);
```

Loads an unaligned 128-bit value. This differs from `movdqu` in that it can provide higher performance in some cases. However, it also may provide lower performance than `movdqu` if the memory value being read was just previously written.

R
*p;

Single-precision Floating-point Vector Intrinsics for Streaming SIMD Extensions 3

The single-precision floating-point vector intrinsics listed here are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3).

The results of each intrinsic operation are placed in the registers R0, R1, R2, and R3.

Details about each intrinsic follows the table below.

The prototypes for these intrinsics are in the `pmmintrin.h` header file.

Intrinsic Name	Operation	Corresponding SSE3 Instruction
_mm_addsub_ps	Subtract and add	ADDSUBPS
_mm_hadd_ps	Add	HADDPS
_mm_hsub_ps	Subtracts	HSUBPS
_mm_movehdup_ps	Duplicates	MOVSHDUP
_mm_moveldup_ps	Duplicates	MOVSLDUP

```
extern __m128 _mm_addsub_ps(__m128 a, __m128 b);
```

Subtracts even vector elements while adding odd vector elements.

R0	R1	R2	R3
a0 - b0;	a1 + b1;	a2 - b2;	a3 + b3;

```
extern __m128 _mm_hadd_ps(__m128 a, __m128 b);
```

Adds adjacent vector elements.

R0	R1	R2	R3
a0 + a1;	a2 + a3;	b0 + b1;	b2 + b3;

```
extern __m128 _mm_hsub_ps(__m128 a, __m128 b);
```

Subtracts adjacent vector elements.

R0	R1	R2	R3
a0 - a1;	a2 - a3;	b0 - b1;	b2 - b3;

```
extern __m128 _mm_movehdup_ps(__m128 a);
```

Duplicates odd vector elements into even vector elements.

R0	R1	R2	R3
a1;	a1;	a3;	a3;

```
extern __m128 _mm_moveldup_ps(__m128 a);
```

Duplicates even vector elements into odd vector elements.

R0	R1	R2	R3
a0;	a0;	a2;	a2;

Double-precision Floating-point Vector Intrinsics for Streaming SIMD Extensions 3

The floating-point intrinsics listed here are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3).

The results of each intrinsic operation are placed in the registers R0 and R1.

Details about each intrinsic follows the table below.

The prototypes for these intrinsics are in the pmmintrin.h header file.

Intrinsic Name	Operation	Corresponding SSE3 Instruction
_mm_addsub_pd	Subtract and add	ADDSUBPD
_mm_hadd_pd	Add	HADDPD
_mm_hsub_pd	Subtract	HSUBPD
_mm_loaddup_pd	Duplicate	MOVDDUP
_mm_movedup_pd	Duplicate	MOVDDUP

```
extern __m128d _mm_addsub_pd(__m128d a, __m128d b);
```

Adds upper vector element while subtracting lower vector element.

R0	R1
a0 - b0;	a1 + b1;

```
extern __m128d _mm_hadd_pd(__m128d a, __m128d b);
```

Adds adjacent vector elements.

R0	R1
a0 + a1;	b0 + b1;

```
extern __m128d _mm_hsub_pd(__m128d a, __m128d b);
```

Subtracts adjacent vector elements.

R0	R1
a0 - a1;	b0 - b1;

```
extern __m128d _mm_loaddup_pd(double const * dp);
```

Duplicates a double value into upper and lower vector elements.

R0	R1
*dp;	*dp;

```
extern __m128d _mm_movedup_pd(__m128d a);
```

Duplicates lower vector element into upper vector element.

R0	R1
a0;	a0;

Macro Functions for Streaming SIMD Extensions 3

The macro function intrinsics listed here are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3).

The prototypes for these intrinsics are in the `pmmintrin.h` header file.

```
_MM_SET_DENORMALS_ZERO_MODE(x)
```

Macro arguments: one of `__MM_DENORMALS_ZERO_ON`, `_MM_DENORMALS_ZERO_OFF`
This causes "`denormals are zero`" mode to be turned on or off by setting the appropriate bit of the control register.

```
_MM_GET_DENORMALS_ZERO_MODE()
```

No arguments. This returns the current value of the denormals are zero mode bit of the control register.

Miscellaneous Intrinsics for Streaming SIMD Extensions 3

The miscellaneous intrinsics listed here are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3).

The prototypes for these intrinsics are in the `pmmintrin.h` header file.

```
extern void _mm_monitor(void const *p, unsigned extensions, unsigned
hints);
```

Generates the `MONITOR` instruction. This sets up an address range for the monitor hardware using `p` to provide the logical address, and will be passed to the monitor instruction in register `eax`. The extensions parameter contains optional extensions to the monitor hardware which will be passed in `ecx`. The hints parameter will contain hints to the monitor hardware, which will be passed in `edx`. A non-zero value for extensions will cause a general protection fault.

```
extern void _mm_mwait(unsigned extensions, unsigned hints);
```

Generates the MWAIT instruction. This instruction is a hint that allows the processor to stop execution and enter an implementation-dependent optimized state until occurrence of a class of events. In future processor designs extensions and hints parameters may be used to convey additional information to the processor. All non-zero values of extensions and hints are reserved. A non-zero value for extensions will cause a general protection fault.

Supplemental Streaming SIMD Extensions 3

Overview: Supplemental Streaming SIMD Extensions 3

The Intel C++ intrinsics listed in this section are supported in the Supplemental Streaming SIMD Extensions 3. The prototypes for these intrinsics are in `tmmintrin.h`. You can also use the `ia32intrin.h` header file for these intrinsics.

- Addition Intrinsics
- Subtraction Intrinsics
- Multiplication Intrinsics
- Absolute Value Intrinsics
- Shuffle Intrinsics
- Concatenate Intrinsics
- Negation Intrinsics

Addition Intrinsics

Use the following SSSE3 intrinsics for horizontal addition.

```
extern __m128i _mm_hadd_epi16 (__m128i a, __m128i b);
```

Add horizontally packed signed words.

Interpreting `a`, `b`, and `r` as arrays of 16-bit signed integers:

```
for (i = 0; i < 4; i++) {

  r[i] = a[2*i] + a[2i+1];

  r[i+4] = b[2*i] + b[2*i+1];

}
```

```
extern __m128i _mm_hadd_epi32 (__m128i a, __m128i b);
```

Add horizontally packed signed dwords.

Interpreting `a`, `b`, and `r` as arrays of 32-bit signed integers:

```
for (i = 0; i < 2; i++) {

  r[i] = a[2*i] + a[2i+1];

  r[i+2] = b[2*i] + b[2*i+1];

}
```

```
extern __m128i _mm_hadds_epi16 (__m128i a, __m128i b);
```

Add horizontally packed signed words with signed saturation.

Interpreting a, b, and r as arrays of 16-bit signed integers:

```
for (i = 0; i < 4; i++) {
  r[i] = signed_saturate_to_word(a[2*i] + a[2i+1]);
  r[i+4] = signed_saturate_to_word(b[2*i] + b[2*i+1]);
}
```

```
extern __m64 _mm_hadd_pi16 (__m64 a, __m64 b);
```

Add horizontally packed signed words.

Interpreting a, b, and r as arrays of 16-bit signed integers:

```
for (i = 0; i < 2; i++) {
  r[i] = a[2*i] + a[2i+1];
  r[i+2] = b[2*i] + b[2*i+1];
}
```

```
extern __m64 _mm_hadd_pi32 (__m64 a, __m64 b);
```

Add horizontally packed signed dwords.

Interpreting a, b, and r as arrays of 32-bit signed integers:

```
r[0] = a[1] + a[0];
r[1] = b[1] + b[0];
```

```
extern __m64 _mm_hadds_pi16 (__m64 a, __m64 b);
```

Add horizontally packed signed words with signed saturation.

Interpreting a, b, and r as arrays of 16-bit signed integers:

```
for (i = 0; i < 2; i++) {
  r[i] = signed_saturate_to_word(a[2*i] + a[2i+1]);
  r[i+2] = signed_saturate_to_word(b[2*i] + b[2*i+1]);
}
```

Subtraction Intrinsics

Use the following SSSE3 intrinsics for horizontal subtraction.

```
extern __m128i _mm_hsub_epi16 (__m128i a, __m128i b);
```

Subtract horizontally packed signed words.

Interpreting `a`, `b`, and `r` as arrays of 16-bit signed integers:

```
for (i = 0; i < 4; i++) {
  r[i] = a[2*i] - a[2i+1];
  r[i+4] = b[2*i] - b[2*i+1];
}
```

```
extern __m128i _mm_hsub_epi32 (__m128i a, __m128i b);
```

Subtract horiztonally packed signed dwords.

Interpreting `a`, `b`, and `r` as arrays of 32-bit signed integers:

```
for (i = 0; i < 2; i++) {
  r[i] = a[2*i] - a[2i+1];
  r[i+2] = b[2*i] - b[2*i+1];
}
```

```
extern __m128i _mm_hsubs_epi16 (__m128i a, __m128i b);
```

Subract horizontally packed signed words with signed saturation.

Interpreting `a`, `b`, and `r` as arrays of 16-bit signed integers:

```
for (i = 0; i < 4; i++) {
  r[i] = signed_saturate_to_word(a[2*i] - a[2i+1]);
  r[i+4] = signed_saturate_to_word(b[2*i] - b[2*i+1]);
}
```

```
extern __m64 _mm_hsub_pi16 (__m64 a, __m64 b);
```

Subtract horizontally packed signed words.

Interpreting `a`, `b`, and `r` as arrays of 16-bit signed integers:

```
for (i = 0; i < 2; i++) {
  r[i] = a[2*i] - a[2i+1];
  r[i+2] = b[2*i] - b[2*i+1];
}
extern __m64 _mm_hsub_pi32 (__m64 a, __m64 b);
```

Subtract horizontally packed signed dwords.

Interpreting `a`, `b`, and `r` as arrays of 32-bit signed integers:

```
r[0] = a[0] - a[1];
r[1] = b[0] - b[1];
extern __m64 _mm_hsubs_pi16 (__m64 a, __m64 b);
```

Subtract horizontally packed signed words with signed saturation.

Interpreting `a`, `b`, and `r` as arrays of 16-bit signed integers:

```
for (i = 0; i < 2; i++) {
  r[i] = signed_saturate_to_word(a[2*i] - a[2i+1]);
  r[i+2] = signed_saturate_to_word(b[2*i] - b[2*i+1]);
}
```

Multiplication Intrinsics

Use the following SSSE3 intrinsics for multiplication.

```
extern __m128i _mm_maddubs_epi16 (__m128i a, __m128i b);
```

Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed words.

Interpreting `a` as array of unsigned 8-bit integers, `b` as arrays of signed 8-bit integers, and `r` as arrays of 16-bit signed integers:

```
for (i = 0; i < 8; i++) {
  r[i] = signed_saturate_to_word(a[2*i+1] * b[2*i+1] + a[2*i]*b[2*i]);
}
extern __m64 _mm_maddubs_pi16 (__m64 a, __m64 b);
```

Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed words.

Interpreting `a` as array of unsigned 8-bit integers, `b` as arrays of signed 8-bit integers, and `r` as arrays of 16-bit signed integers:

```
for (i = 0; i < 4; i++) {

  r[i] = signed_saturate_to_word(a[2*i+1] * b[2*i+1] + a[2*i]*b[2*i]);

}
```

```
extern __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b);
```

Multiply signed words, scale and round signed dwords, pack high 16-bits.

Interpreting `a`, `b`, and `r` as arrays of signed 16-bit integers:

```
for (i = 0; i < 8; i++) {

  r[i] =  (( (int32)((a[i] * b[i]) >> 14) + 1) >> 1) & 0xFFFF;

}
```

```
extern __m64 _mm_mulhrs_pi16 (__m64 a, __m64 b);
```

Multiply signed words, scale and round signed dwords, pack high 16-bits.

Interpreting `a`, `b`, and `r` as arrays of signed 16-bit integers:

```
for (i = 0; i < 4; i++) {

  r[i] =  (( (int32)((a[i] * b[i]) >> 14) + 1) >> 1) & 0xFFFF;

}
```

Absolute Value Intrinsics

Use the following SSSE3 intrinsics to compute absolute values.

```
extern __m128i _mm_abs_epi8 (__m128i a);
```

Compute absolute value of signed bytes.

Interpreting `a` and `r` as arrays of signed 8-bit integers:

```
for (i = 0; i < 16; i++) {

   r[i] = abs(a[i]);

}
```

```
extern __m128i _mm_abs_epi16 (__m128i a);
```

Compute absolute value of signed words.

Interpreting `a` and `r` as arrays of signed 16-bit integers:

```
for (i = 0; i < 8; i++) {
   r[i] = abs(a[i]);
}
```

```
extern __m128i _mm_abs_epi32 (__m128i a);
```

Compute absolute value of signed dwords.

Interpreting `a` and `r` as arrays of signed 32-bit integers:

```
for (i = 0; i < 4; i++) {
   r[i] = abs(a[i]);
}
```

```
extern __m64 _mm_abs_pi8 (__m64 a);
```

Compute absolute value of signed bytes.

Interpreting `a` and `r` as arrays of signed 8-bit integers:

```
for (i = 0; i < 8; i++) {
   r[i] = abs(a[i]);
}
```

```
extern __m64 _mm_abs_pi16 (__m64 a);
```

Compute absolute value of signed words.

Interpreting `a` and `r` as arrays of signed 16-bit integers:

```
for (i = 0; i < 4; i++) {
   r[i] = abs(a[i]);
}
```

```
extern __m64 _mm_abs_pi32 (__m64 a);
```

Compute absolute value of signed dwords.

Interpreting a and r as arrays of signed 32-bit integers:

```
for (i = 0; i < 2; i++) {
   r[i] = abs(a[i]);
}
```

Shuffle Intrinsics for Streaming SIMD Extensions 3

Use the following SSSE3 intrinsics for shuffle.

```
extern __m128i _mm_shuffle_epi8 (__m128i a, __m128i b);
```

Shuffle bytes from a according to contents of b.

Interpreting a, b, and r as arrays of unsigned 8-bit integers:

```
for (i = 0; i < 16; i++) {
  if (b[i] & 0x80) {
      r[i] =  0;
  }
  else {
      r[i] = a[b[i] & 0x0F];
  }
}
```

```
extern __m64 _mm_shuffle_pi8 (__m64 a, __m64 b);
```

Shuffle bytes from a according to contents of b.

Interpreting a, b, and r as arrays of unsigned 8-bit integers:

```
for (i = 0; i < 8; i++) {
  if (b[i] & 0x80) {
      r[i] =  0;
  }
  else {
      r[i] = a[b[i] & 0x07];
```

```
  }
}
```

Concatenate Intrinsics

Use the following SSSE3 intrinsics for concatenation.

```
extern __m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n);
```

Concatenate `a` and `b`, extract byte-aligned result shifted to the right by `n`.

Interpreting `t1` as 256-bit unsigned integer, `a`, `b`, and `r` as 128-bit unsigned integers:

```
t1[255:128] = a;

t1[127:0] = b;

t1[255:0] = t1[255:0] >> (8 * n); // unsigned shift

r[127:0] = t1[127:0];
```

```
extern __m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n);
```

Concatenate `a` and `b`, extract byte-aligned result shifted to the right by `n`.

Interpreting `t1` as 127-bit unsigned integer, `a`, `b` and `r` as 64-bit unsigned integers:

```
t1[127:64] = a;

t1[63:0] = b;

t1[127:0] = t1[127:0] >> (8 * n); // unsigned shift

r[63:0] = t1[63:0];
```

Negation Intrinsics

Use the following SSSE3 intrinsics for negation.

```
extern __m128i _mm_sign_epi8 (__m128i a, __m128i b);
```

Negate packed bytes in `a` if corresponding sign in `b` is less than zero.

Interpreting `a`, `b`, and `r` as arrays of signed 8-bit integers:

```
for (i = 0; i < 16; i++) {

  if (b[i] < 0) {
```

```
      r[i] =  -a[i];
  }
  else if (b[i] == 0) {
      r[i] = 0;
  }
  else {
      r[i] = a[i];
  }
}
extern __m128i _mm_sign_epi16 (__m128i a, __m128i b);
```

Negate packed words in `a` if corresponding sign in `b` is less than zero.

Interpreting `a`, `b`, and `r` as arrays of signed 16-bit integers:

```
for (i = 0; i < 8; i++) {
  if (b[i] < 0) {
      r[i] =  -a[i];
  }
  else if (b[i] == 0) {
      r[i] = 0;
  }
  else {
      r[i] = a[i];
  }
}
extern __m128i _mm_sign_epi32 (__m128i a, __m128i b);
```

Negate packed dwords in `a` if corresponding sign in `b` is less than zero.

Interpreting `a`, `b`, and `r` as arrays of signed 32-bit integers:

```
for (i = 0; i < 4; i++) {
```

```
  if (b[i] < 0) {
      r[i] =  -a[i];
  }
  else if (b[i] == 0) {
      r[i] = 0;
  }
  else {
      r[i] = a[i];
  }
}
extern __m64 _mm_sign_pi8 (__m64 a, __m64 b);
```

Negate packed bytes in `a` if corresponding sign in `b` is less than zero.

Interpreting `a`, `b`, and `r` as arrays of signed 8-bit integers:

```
for (i = 0; i < 16; i++) {
  if (b[i] < 0) {
      r[i] =  -a[i];
  }
  else if (b[i] == 0) {
      r[i] = 0;
  }
  else {
      r[i] = a[i];
  }
}
extern __m64 _mm_sign_pi16 (__m64 a, __m64 b);
```

Negate packed words in `a` if corresponding sign in `b` is less than zero.

Interpreting `a`, `b`, and `r` as arrays of signed 16-bit integers:

```
for (i = 0; i < 8; i++) {
  if (b[i] < 0) {
      r[i] =  -a[i];
  }
  else if (b[i] == 0) {
      r[i] = 0;
  }
  else {
      r[i] = a[i];
  }
}
extern __m64 _mm_sign_pi32 (__m64 a, __m64 b);
```

Negate packed dwords in `a` if corresponding sign in `b` is less than zero.

Interpreting `a`, `b`, and `r` as arrays of signed 32-bit integers:

```
for (i = 0; i < 2; i++) {
  if (b[i] < 0) {
      r[i] =  -a[i];
  }
  else if (b[i] == 0) {
      r[i] = 0;
  }
  else {
      r[i] = a[i];
  }
}
```

Streaming SIMD Extensions 4

Overview: Streaming SIMD Extensions 4

The intrinsics in this section correspond to Intel® Streaming SIMD Extensions 4 (SSE4) instructions. SSE4 includes the following categories:

- Streaming SIMD Extensions 4 (SSE4) Vectorizing Compiler and Media Accelerators
  The prototypes for these intrinsics are in the smmitnrin.h file.
- Streaming SIMD Extensions 4 (SSE4) Efficient Accelerated String and Text Processing
  The prototypes for these intrinsics are in the nmmintrin.h file.

Streaming SIMD Extensions 4 Vectorizing Compiler and Media Accelerators

Overview: Streaming SIMD Extensions 4 Vectorizing Compiler and Media Accelerators

The intrinsics in this section correspond to Streaming SIMD Extensions 4 (SSE4) Vectorizing Compiler and Media Accelerators instructions.

- Packed Blending Intrinsics for Streaming SIMD Extensions 4
- Floating Point Dot Product Intrinsics for Streaming SIMD Extensions 4
- Packed Format Conversion Intrinsics for Streaming SIMD Extensions 4
- Packed Integer Min/Max Intrinsics for Streaming SIMD Extensions 4
- Floating Point Rounding Intrinsics for Streaming SIMD Extensions 4
- DWORD Multiply Intrinsics for Streaming SIMD Extensions 4
- Register Insertion/Extraction Intrinsics for Streaming SIMD Extensions 4
- Test Intrinsics for Streaming SIMD Extensions 4
- Packed DWORD to Unsigned WORD Intrinsic for Streaming SIMD Extensions 4
- Packed Compare for Equal Intrinsics for Streaming SIMD Extensions 4
- Cacheability Support Intrinsic for Streaming SIMD Extension 4

The prototypes for these intrinsics are in the `smmintrin.h` file.

Packed Blending Intrinsics for Streaming SIMD Extensions 4

These intrinsics pack multiple operations in a single instruction. Blending conditionally copies one field in the source onto the corresponding field in the destination.

Intrinsic Name	Operation	Corresponding SSE4 Instruction
`_mm_blend_ps`	Select float single precision data from 2 sources using constant mask	`BLENDPS`

_mm_blend_pd	Select float double precision data from 2 sources using constant mask	BLENDPD
_mm_blendv_ps	Select float single precision data from 2 sources using variable mask	BLENDVPS
_mm_blendv_pd	Select float double precision data from 2 sources using variable mask	BLENDVPD
_mm_blendv_epi8	Select integer bytes from 2 sources using variable mask	PBLENDVB
_mm_blend_epi16	Select integer words from 2 sources using constant mask	PBLENDW

```
__m128d _mm_blend_pd (__m128d v1, __m128d v2, const int mask)

__m128  _mm_blend_ps (__m128  v1, __m128  v2, const int mask)

__m128d _mm_blendv_pd (__m128d v1, __m128d v2, __m128d v3)

__m128  _mm_blendv_ps (__m128  v1, __m128  v2, __m128 v3)

__m128i _mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask)

__m128i _mm_blend_epi16 (__m128i v1, __m128i v2, const int mask)
```

Floating Point Dot Product Intrinsics for Streaming SIMD Extensions 4

These intrinsics enable floating point single precision and double precision dot products.

Intrinsic Name	Operation	Corresponding SSE4 Instruction
_mm_dp_pd	Double precision dot product	DPPD
_mm_dp_ps	Single precision dot product	DPPS

```
__m128d _mm_dp_pd ( __m128d a, __m128d b, const int mask)
```

This intrinsic calculates the dot product of double precision packed values with mask-defined summing and zeroing of the parts of the result.

```
__m128 _mm_dp_ps ( __m128 a, __m128 b, const int mask)
```

This intrinsic calculates the dot product of single precision packed values with mask-defined summing and zeroing of the parts of the result.

Packed Format Conversion Intrinsics for Streaming SIMD Extensions 4

These intrinsics convert from a packed integer to a zero-extended or sign-extended integer with wider type.

Intrinsic Name	Operation	Corresponding SSE4 Instruction
_mm_ cvtepi8_epi32	Sign extend 4 bytes into 4 double words	PMOVSXBD
_mm_ cvtepi8_epi64	Sign extend 2 bytes into 2 quad words	PMOVSXBQ
_mm_ cvtepi8_epi16	Sign extend 8 bytes into 8 words	PMOVSXBW
_mm_ cvtepi32_epi64	Sign extend 2 double words into 2 quad words	PMOVSXDQ
_mm_ cvtepi16_epi32	Sign extend 4 words into 4 double words	PMOVSXWD
_mm_ cvtepi16_epi64	Sign extend 2 words into 2 quad words	PMOVSXWQ
_mm_ cvtepu8_epi32	Zero extend 4 bytes into 4 double words	PMOVZXBD
_mm_ cvtepu8_epi64	Zero extend 2 bytes into 2 quad words	PMOVZXBQ
_mm_ cvtepu8_epi16	Zero extend 8 bytes into 8 word	PMOVZXBW
_mm_ cvtepu32_epi64	Zero extend 2 double words into 2 quad words	PMOVZXDQ
_mm_ cvtepu16_epi32	Zero extend 4 words into 4 double words	PMOVZXWD
_mm_ cvtepu16_epi64	Zero extend 2 words into 2 quad words	PMOVZXWQ

```
__m128i _mm_ cvtepi8_epi32 ( __m128i a)

__m128i _mm_ cvtepi8_epi64 ( __m128i a)

__m128i _mm_ cvtepi8_epi16 ( __m128i a)

__m128i _mm_ cvtepi32_epi64 ( __m128i a)

__m128i _mm_ cvtepi16_epi32 ( __m128i a)

__m128i _mm_ cvtepi16_epi64 ( __m128i a)

__m128i _mm_ cvtepu8_epi32 ( __m128i a)

__m128i _mm_ cvtepu8_epi64 ( __m128i a)
```

```
__m128i _mm_ cvtepu8_epi16 ( __m128i a)

__m128i _mm_ cvtepu32_epi64 ( __m128i a)

__m128i _mm_ cvtepu16_epi32 ( __m128i a)

__m128i _mm_ cvtepu16_epi64 ( __m128i a)
```

Packed Integer Min/Max Intrinsics for Streaming SIMD Extensions 4

These intrinsics compare packed integers in the destination operand and the source operand, and return the minimum or maximum for each packed operand in the destination operand.

Intrinsic Name	Operation	Corresponding SSE4 Instruction
_mm_max_epi8	Calculate maximum of signed packed integer bytes	PMAXSB
_mm_max_epi32	Calculate maximum of signed packed integer double words	PMAXSD
_mm_max_epu32	Calculate maximum of unsigned packed integer double words	PMAXUD
_mm_max_epu16	Calculate maximum of unsigned packed integer words	PMAXUW
_mm_min_epi8	Calculate minimum of signed packed integer bytes	PMINSB
_mm_min_epi32	Calculate minimum of signed packed integer double words	PMINSD
_mm_min_epu32	Calculate minimum of unsigned packed integer double words	PMINUD
_mm_min_epu16	Calculate minimum of unsigned packed integer words	PMINUW

```
__m128i _mm_max_epi8 ( __m128i a, __m128i b)

__m128i _mm_max_epi32 ( __m128i a, __m128i b)

__m128i _mm_max_epu32 ( __m128i a, __m128i b)

__m128i _mm_max_epu16 ( __m128i a, __m128i b)

__m128i _mm_min_epi8 ( __m128i a, __m128i b)

__m128i _mm_min_epi32 ( __m128i a, __m128i b)
```

```
__m128i _mm_min_epu32 ( __m128i a, __m128i b)

__m128i _mm_min_epu16 ( __m128i a, __m128i b)
```

Floating Point Rounding Intrinsics for Streaming SIMD Extensions 4

These rounding intrinsics cover scalar and packed single-precision and double precision floating-point operands.

The `floor` and `ceil` intrinsics correspond to the definitions of `floor` and `ceil` in the *ISO 9899:1999* standard for the C programming language.

Intrinsic Name	Operation	Corresponding SSE4 Instruction
_mm_round_pd mm_floor_pd mm_ceil_pd	Packed float double precision rounding	ROUNDPD
_mm_round_ps mm_floor_ps mm_ceil_ps	Packed float single precision rounding	ROUNDPS
_mm_round_sd mm_floor_sd mm_ceil_sd	Single float double precision rounding	ROUNDSD
_mm_round_ss mm_floor_ss mm_ceil_ss	Single float single precision rounding	ROUNDSS

```
__m128d _mm_round_pd(__m128d s1, int iRoundMode)

__m128d mm_floor_pd(__m128d s1)

__m128d mm_ceil_pd(__m128d s1)

__m128 _mm_round_ps(__m128 s1, int iRoundMode)

__m128 mm_floor_ps(__m128 s1)

__m128 mm_ceil_ps(__m128 s1)

__m128d _mm_round_sd(__m128d dst, __m128d s1, int iRoundMode)
```

```
__m128d mm_floor_sd(__m128d dst, __m128d s1)

__m128d mm_ceil_sd(__m128d dst, __m128d s1)

__m128 _mm_round_ss(__m128 dst, __m128 s1, int iRoundMode)

__m128 mm_floor_ss(__m128 dst, __m128 s1)

__m128 mm_ceil_ss(__m128 dst, __m128 s1)
```

DWORD Multiply Intrinsics for Streaming SIMD Extensions 4

These DWORD multiply intrinsics are designed to aid vectorization. They enable four simultaneous 32 bit by 32 bit multiplies.

Intrinsic Name	Operation	Corresponding SSE4 Instruction
_mm_mul_epi32	Packed integer 32-bit multiplication of 2 low pairs of operands producing two 64-bit results	PMULDQ
_mm_mullo_epi32	Packed integer 32-bit multiplication with truncation of upper halves of results	PMULLD

```
__m128i _mm_mul_epi32( __m128i a, __m128i b)

__m128i _mm_mullo_epi32(__m128i a, __m128i b)
```

Register Insertion/Extraction Intrinsics for Streaming SIMD Extensions 4

These intrinsics enable data insertion and extraction between general purpose registers and XMM registers.

Intrinsic Name	Operation	Corresponding SSE4 Instruction
_mm_insert_ps	Insert single precision float into packed single precision array element selected by index	INSERTPS
_mm_extract_ps	Extract single precision float from packed single precision array element selected by index	EXTRACTPS
_mm_extract_epi8	Extract integer byte from packed integer array element selected by index	PEXTRB
_mm_extract_epi32	Extract integer double word from packed integer array element selected by index	PEXTRD

_mm_extract_epi64	Extract integer quad word from packed integer array element selected by index	PEXTRQ
_mm_extract_epi16	Extract integer word from packed integer array element selected by index	PEXTRW
_mm_insert_epi8	Insert integer byte into packed integer array element selected by index	PINSRB
_mm_insert_epi32	Insert integer double word into packed integer array element selected by index	PINSRD
_mm_insert_epi64	Insert integer quad word into packed integer array element selected by index	PINSRQ

```
__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx);

int _mm_extract_ps(__m128 src, const int ndx);

int  _mm_extract_epi8 (__m128i src, const int ndx);

int  _mm_extract_epi32 (__m128i src, const int ndx);

__int64  _mm_extract_epi64 (__m128i src, const int ndx);

int _mm_extract_epi16  (__m128i src, int ndx);

__m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx)

__m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx)

__m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx)
```

Test Intrinsics for Streaming SIMD Extensions 4

These intrinsics perform packed integer 128-bit comparisons.

Intrinsic Name	Operation	Corresponding SSE4 Instruction
_mm_testc_si128	Check for all ones in specified bits of a 128-bit value	PTEST
_mm_testz_si128	Check for all zeros in specified bits of a 128-bit value	PTEST
_mm_testnzc_si128	Check for at least one zero and at least one one in specified bits of a 128-bit value	PTEST

```
int _mm_testz_si128 (__m128i s1, __m128i s2)
```

Returns 1 if the bitwise AND of s1 and s2 is all zero, else returns 0

```
int _mm_testc_si128 (__m128i s1, __m128i s2)
```

Returns 1 if the bitwise AND of s2 ANDNOT of s1 is all ones, else returns 0.

```
int _mm_testnzc_si128 (__m128i s1, __m128i s2)
```

Same as (!_mm)testz) && (!_mm_testc)

Packed DWORD to Unsigned WORD Intrinsic for Streaming SIMD Extensions 4

```
__m128i _mm_packus_epi32(__m128i m1, __m128i m2);
```

Corresponding SSE4 instruction: PACKUSDW

Converts 8 packed signed DWORDs into 8 packed unsigned WORDs, using unsigned saturation to handle overflow condition.

Packed Compare for Equal for Streaming SIMD Extensions 4

```
__m128i _mm_cmpeq_epi64(__m128i a, __m128i b)
```

Corresponding SSE4 instruction: PCMPEQQ

Performs a packed integer 64-bit comparison for equality. The intrinsic zeroes or fills with ones the corresponding parts of the result.

Cacheability Support Intrinsic for Streaming SIMD Extensions 4

```
extern __m128i _mm_stream_load_si128(__m128i* v1);
```

Corresponding SSE4 instruction: MOVNTDQA

Loads `_m128` data from a 16-byte aligned address (v1) to the destination operand (m128i) without polluting the caches.

Streaming SIMD Extensions 4 Efficient Accelerated String and Text Processing

Overview: Streaming SIMD Extensions 4 Efficient Accelerated String and Text Processing

The intrinsics in this section correspond to Streaming SIMD Extensions 4 (SSE4) Efficient Accelerated String and Text Processing instructions. These instructions include:

- Packed Comparison Intrinsics for Streaming SIMD Extensions 4
- Application Targeted Accelerators Intrinsics

The prototypes for these intrinsics are in the `nmmintrin.h` file.

Packed Comparison Intrinsics for Streaming SIMD Extensions 4

These intrinsics perform packed comparisons. They correspond to SSE4 instructions. For intrinsics that could map to more than one instruction, the Intel(R) C++ Compiler selects the instruction to generate.

Intrinsic Name	Operation	Corresponding SSE4 Instruction
_mm_cmpestri	Packed comparison, generates index	PCMPESTRI
_mm_cmpestrm	Packed comparison, generates mask	PCMPESTRM
_mm_cmpistri	Packed comparison, generates index	PCMPISTRI
_mm_cmpistrm	Packed comparison, generates mask	PCMPISTRM
_mm_cmpestrz	Packed comparison	PCMPESTRM or PCMPESTRI
_mm_cmpestrc	Packed comparison	PCMPESTRM or PCMPESTRI
_mm_cmpestrs	Packed comparison	PCMPESTRM or PCMPESTRI
_mm_cmpestro	Packed comparison	PCMPESTRM or PCMPESTRI
_mm_cmpestra	Packed comparison	PCMPESTRM or PCMPESTRI
_mm_cmpistrz	Packed comparison	PCMPISTRM or PCMPISTRI
_mm_cmpistrc	Packed comparison	PCMPISTRM or PCMPISTRI
_mm_cmpistrs	Packed comparison	PCMPISTRM or PCMPISTRI
_mm_cmpistro	Packed comparison	PCMPISTRM or PCMPISTRI
_mm_cmpistra	Packed comparison	PCMPISTRM or PCMPISTRI

```
int _mm_cmpestri(__m128i src1, int len1, __m128i src2, int len2,   const
int mode)
```

This intrinsic performs a packed comparison of string data with explicit lengths, generating an index and storing the result in ECX.

```
__m128i _mm_cmpestrm(__m128i src1, int len1, __m128i src2, int
len2,   const int mode)
```

This intrinsic performs a packed comparison of string data with explicit lengths, generating a mask and storing the result in XMM0.

```
int _mm_cmpistri(__m128i src1, __m128i src2, const int mode)
```

This intrinsic performs a packed comparison of string data with implicit lengths, generating an index and storing the result in ECX.

```
__m128i _mm_cmpistrm(__m128i src1, __m128i src2, const int mode)
```

This intrinsic performs a packed comparison of string data with implicit lengths, generating a mask and storing the result in XMM0.

```
int  _mm_cmpestrz(__m128i src1, int len1, __m128i src2, int len2, const
int mode);
```

This intrinsic performs a packed comparison of string data with explicit lengths. Returns 1 if ZFlag == 1, otherwise 0.

```
int  _mm_cmpestrc(__m128i src1, int len1, __m128i src2, int len2, const
int mode);
```

This intrinsic performs a packed comparison of string data with explicit lengths. Returns 1 if CFlag == 1, otherwise 0.

```
int  _mm_cmpestrs(__m128i src1, int len1, __m128i src2, int len2, const
int mode);
```

This intrinsic performs a packed comparison of string data with explicit lengths. Returns 1 if SFlag == 1, otherwise 0.

```
int  _mm_cmpestro(__m128i src1, int len1, __m128i src2, int len2, const
int mode);
```

This intrinsic performs a packed comparison of string data with explicit lengths. Returns 1 if OFlag == 1, otherwise 0.

```
int  _mm_cmpestra(__m128i src1, int len1, __m128i src2, int len2, const
int mode);
```

This intrinsic performs a packed comparison of string data with explicit lengths. Returns 1 if CFlag == 0 and ZFlag == 0, otherwise 0.

```
int _mm_cmpistrz(__m128i src1, __m128i src2, const int mode);
```

This intrinsic performs a packed comparison of string data with implicit lengths. Returns 1 if (ZFlag == 1), otherwise 0.

```
int _mm_cmpistrc(__m128i src1, __m128i src2, const int mode);
```

This intrinsic performs a packed comparison of string data with implicit lengths. Returns 1 if (CFlag == 1), otherwise 0.

```
int _mm_cmpistrs(__m128i src1, __m128i src2, const int mode);
```

This intrinsic performs a packed comparison of string data with implicit lengths. Returns 1 if (SFlag == 1), otherwise 0.

```
int _mm_cmpistro(__m128i src1, __m128i src2, const int mode);
```

This intrinsic performs a packed comparison of string data with implicit lengths. Returns 1 if (OFlag == 1), otherwise 0.

```
int _mm_cmpistra(__m128i src1, __m128i src2, const int mode);
```

This intrinsic performs a packed comparison of string data with implicit lengths. Returns 1 if (ZFlag == 0 and CFlag == 0), otherwise 0.

Application Targeted Accelerators Intrinsics

Application Targeted Accelerators extend the capabilities of Intel architecture by adding performance-optimized, low-latency, lower power fixed-function accelerators on the processor die to benefit specific applications.

The primitives for these intrinsics are in the file `nmmintrin.h`.

Intrinsic Name	Operation	Corresponding SSE4 Instruction
_mm_popcnt_u32	Counts number of set bits in a data operation	POPCNT
_mm_popcnt_u64	Counts number of set bits in a data operation	POPCNT
_mm_crc32_u8	Accumulate cyclic redundancy check	CRC32
_mm_crc32_u16	Cyclic redundancy check	CRC32
_mm_crc32_u32	Cyclic redundancy check	CRC32
_mm_crc32_u64	Cyclic redundancy check	CRC32

```
int _mm_popcnt_u32(unsigned int v);

int _mm_popcnt_u64(unsigned __int64 v);

unsigned int _mm_crc32_u8 (unsigned int crc, unsigned char v);
```

Starting with an initial value in the first operand, accumulates a CRC32 value for the second operand and stores the result in the destination operand. Accumulates CRC32 on r/m8.

```
unsigned int _mm_crc32_u16(unsigned int crc, unsigned short v);
```

Starting with an initial value in the first operand, accumulates a CRC32 value for the second operand and stores the result in the destination operand. Accumulates CRC32 on r/m16.

```
unsigned int _mm_crc32_u32(unsigned int crc, unsigned int v);
```

Starting with an initial value in the first operand, accumulates a CRC32 value for the second operand and stores the result in the destination operand. Accumulates CRC32 on r/m32.

```
unsigned __int64 _mm_crc32_u64(unsigned __int64 crc, unsigned __int64
v);
```

Starting with an initial value in the first operand, accumulates a CRC32 value for the second operand and stores the result in the destination operand. Accumulates CRC32 on r/m64.

Intrinsics for IA-64 Instructions

Overview: Intrinsics for IA-64 Instructions

This section lists and describes the native intrinsics for IA-64 instructions. These intrinsics cannot be used on the IA-32 architecture. These intrinsics give programmers access to IA-64 instructions that cannot be generated using the standard constructs of the C and C++ languages.

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

The Intel® Itanium® processor does not support SSE2 intrinsics. However, you can use the `sse2mmx.h` emulation pack to enable support for SSE2 instructions on IA-64 architecture.

For information on how to use SSE intrinsics on IA-64 architecture, see Using Streaming SIMD Extensions on IA-64 Architecture.

For information on how to use MMX (TM) technology intrinsics on IA-64 architecture, see MMX(TM) Technology Intrinsics on IA-64 Architecture

Native Intrinsics for IA-64 Instructions

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Integer Operations

Intrinsic	Operation	Corresponding IA-64 Instruction
_m64_dep_mr	Deposit	dep
_m64_dep_mi	Deposit	dep
_m64_dep_zr	Deposit	dep.z
_m64_dep_zi	Deposit	dep.z
_m64_extr	Extract	extr
_m64_extru	Extract	extr.u
_m64_xmal	Multiply and add	xma.l
_m64_xmalu	Multiply and add	xma.lu
_m64_xmah	Multiply and add	xma.h
_m64_xmahu	Multiply and add	xma.hu
_m64_popcnt	Population Count	popcnt
_m64_shladd	Shift left and add	shladd
_m64_shrp	Shift right pair	shrp

FSR Operations

Intrinsic	Description
`void _fsetc(unsigned int amask, unsigned int omask)`	Sets the control bits of `FPSR.sf0`. Maps to the `fsetc.sf0 r, r` instruction. There is no corresponding instruction to read the control bits. Use `_mm_getfpsr()`.
`void _fclrf(void)`	Clears the floating point status flags (the 6-bit flags of `FPSR.sf0`). Maps to the `fclrf.sf0` instruction.

```
__int64 _m64_dep_mr(__int64 r, __int64 s, const int pos, const int len)
```

The right-justified 64-bit value `r` is deposited into the value in `s` at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `pos` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

```
__int64 _m64_dep_mi(const int v, __int64 s, const int p, const int len)
```

The sign-extended value `v` (either all 1s or all 0s) is deposited into the value in `s` at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `p` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

```
__int64 _m64_dep_zr(__int64 s, const int pos, const int len)
```

The right-justified 64-bit value `s` is deposited into a 64-bit field of all zeros at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `pos` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

```
__int64 _m64_dep_zi(const int v, const int pos, const int len)
```

The sign-extended value `v` (either all 1s or all 0s) is deposited into a 64-bit field of all zeros at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `pos` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

```
__int64 _m64_extr(__int64 r, const int pos, const int len)
```

A field is extracted from the 64-bit value `r` and is returned right-justified and sign extended. The extracted field begins at position `pos` and extends `len` bits to the left. The sign is taken from the most significant bit of the extracted field.

```
__int64 _m64_extru(__int64 r, const int pos, const int len)
```

A field is extracted from the 64-bit value `r` and is returned right-justified and zero extended. The extracted field begins at position `pos` and extends `len` bits to the left.

```
__int64 _m64_xmal(__int64 a, __int64 b, __int64 c)
```

The 64-bit values a and b are treated as signed integers and multiplied to produce a full 128-bit signed result. The 64-bit value c is zero-extended and added to the product. The least significant 64 bits of the sum are then returned.

```
__int64 _m64_xmalu(__int64 a, __int64 b, __int64 c)
```

The 64-bit values a and b are treated as signed integers and multiplied to produce a full 128-bit unsigned result. The 64-bit value c is zero-extended and added to the product. The least significant 64 bits of the sum are then returned.

```
__int64 _m64_xmah(__int64 a, __int64 b, __int64 c)
```

The 64-bit values a and b are treated as signed integers and multiplied to produce a full 128-bit signed result. The 64-bit value c is zero-extended and added to the product. The most significant 64 bits of the sum are then returned.

```
__int64 _m64_xmahu(__int64 a, __int64 b, __int64 c)
```

The 64-bit values a and b are treated as unsigned integers and multiplied to produce a full 128-bit unsigned result. The 64-bit value c is zero-extended and added to the product. The most significant 64 bits of the sum are then returned.

```
__int64 _m64_popcnt(__int64 a)
```

The number of bits in the 64-bit integer a that have the value 1 are counted, and the resulting sum is returned.

```
__int64 _m64_shladd(__int64 a, const int count, __int64 b)
```

a is shifted to the left by count bits and then added to b. The result is returned.

```
__int64 _m64_shrp(__int64 a, __int64 b, const int count)
```

a and b are concatenated to form a 128-bit value and shifted to the right count bits. The least significant 64 bits of the result are returned.

Lock and Atomic Operation Related Intrinsics

The prototypes for these intrinsics are in the ia64intrin.h header file.

Intrinsic	Description
`unsigned __int64 _InterlockedExchange8(volatile unsigned char *Target, unsigned __int64 value)`	Map to the xchg1 instruction. Atomically write the least significant byte of its 2nd argument to address specified by its 1st argument.
`unsigned __int64 _InterlockedCompareExchange8_rel(volatile unsigned char *Destination, unsigned __int64`	Compare and exchange atomically the least significant byte at the

Exchange, unsigned __int64 Comparand)	address specified by its 1st argument. Maps to the cmpxchg1.rel instruction with appropriate setup.
unsigned __int64 _InterlockedCompareExchange8_acq(volatile unsigned char *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	Same as the previous intrinsic, but using acquire semantic.
unsigned __int64 _InterlockedExchange16(volatile unsigned short *Target, unsigned __int64 value)	Map to the xchg2 instruction. Atomically write the least significant word of its 2nd argument to address specified by its 1st argument.
unsigned __int64 _InterlockedCompareExchange16_rel(volatile unsigned short *Destination, unsigned int64 Exchange, unsigned __int64 Comparand)	Compare and exchange atomically the least significant word at the address specified by its 1st argument. Maps to the cmpxchg2.rel instruction with appropriate setup.
unsigned __int64 _InterlockedCompareExchange16_acq(volatile unsigned short *Destination, unsigned int64 Exchange, unsigned __int64 Comparand)	Same as the previous intrinsic, but using acquire semantic.
int _InterlockedIncrement(volatile int *addend	Atomically increment by one the value specified by its argument. Maps to the fetchadd4 instruction.
int _InterlockedDecrement(volatile int *addend	Atomically decrement by one the value specified by its argument. Maps to the fetchadd4 instruction.
int _InterlockedExchange(volatile int *Target, long value	Do an exchange operation atomically. Maps to the xchg4 instruction.
int _InterlockedCompareExchange(volatile int *Destination, int Exchange, int Comparand	Do a compare and exchange operation atomically. Maps to the cmpxchg4 instruction with appropriate setup.
int _InterlockedExchangeAdd(volatile int *addend, int increment	Use compare and exchange to do an atomic add of the increment value to the addend. Maps to a loop with the cmpxchg4 instruction to guarantee atomicity.
int InterlockedAdd(volatile int *addend, int	Same as the previous

increment)	intrinsic, but returns new value, not the original one.
void * _InterlockedCompareExchangePointer(void * volatile *Destination, void *Exchange, void *Comparand)	Map the exch8 instruction; Atomically compare and exchange the pointer value specified by its first argument (all arguments are pointers)
unsigned __int64 _InterlockedExchangeU(volatile unsigned int *Target, unsigned __int64 value)	Atomically exchange the 32-bit quantity specified by the 1st argument. Maps to the xchg4 instruction.
unsigned __int64 _InterlockedCompareExchange_rel(volatile unsigned int *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	Maps to the cmpxchg4.rel instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 64-bit pointer).
unsigned __int64 _InterlockedCompareExchange_acq(volatile unsigned int *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	Same as the previous intrinsic, but map the cmpxchg4.acq instruction.
void _ReleaseSpinLock(volatile int *x)	Release spin lock.
__int64 _InterlockedIncrement64(volatile __int64 *addend)	Increment by one the value specified by its argument. Maps to the fetchadd instruction.
__int64 _InterlockedDecrement64(volatile __int64 *addend)	Decrement by one the value specified by its argument. Maps to the fetchadd instruction.
__int64 _InterlockedExchange64(volatile __int64 *Target, __int64 value)	Do an exchange operation atomically. Maps to the xchg instruction.
unsigned __int64 _InterlockedExchangeU64(volatile unsigned __int64 *Target, unsigned __int64 value)	Same as InterlockedExchange64 (for unsigned quantities).
unsigned __int64 _InterlockedCompareExchange64_rel(volatile unsigned __int64 *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	Maps to the cmpxchg.rel instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 64-bit pointer).
unsigned __int64 _InterlockedCompareExchange64_acq(volatile unsigned __int64 *Destination, unsigned	Maps to the cmpxchg.acq instruction with appropriate setup. Atomically compare

`__int64 Exchange, unsigned __int64 Comparand)`	and exchange the value specified by the first argument (a 64-bit pointer).
`__int64 _InterlockedCompareExchange64(volatile __int64 *Destination, __int64 Exchange, __int64 Comparand)`	Same as the previous intrinsic for signed quantities.
`__int64 _InterlockedExchangeAdd64(volatile __int64 *addend, __int64 increment)`	Use compare and exchange to do an atomic add of the increment value to the addend. Maps to a loop with the `cmpxchg` instruction to guarantee atomicity
`__int64 _InterlockedAdd64(volatile __int64 *addend, __int64 increment);`	Same as the previous intrinsic, but returns the new value, not the original value. See Note.

Note

`_InterlockedSub64` is provided as a macro definition based on `_InterlockedAdd64`.

```
#define _InterlockedSub64(target, incr)
_InterlockedAdd64((target),(-(incr))).
```

Uses `cmpxchg` to do an atomic sub of the `incr` value to the `target`. Maps to a loop with the `cmpxchg` instruction to guarantee atomicity.

Lock and Atomic Operation Related Intrinsics

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Intrinsic	Description
`unsigned __int64 _InterlockedExchange8(volatile unsigned char *Target, unsigned __int64 value)`	Map to the `xchg1` instruction. Atomically write the least significant byte of its 2nd argument to address specified by its 1st argument.
`unsigned __int64 _InterlockedCompareExchange8_rel(volatile unsigned char *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)`	Compare and exchange atomically the least significant byte at the address specified by its 1st argument. Maps to the `cmpxchg1.rel` instruction with appropriate setup.

unsigned __int64 _InterlockedCompareExchange8_acq(volatile unsigned char *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	Same as the previous intrinsic, but using acquire semantic.
unsigned __int64 _InterlockedExchange16(volatile unsigned short *Target, unsigned __int64 value)	Map to the xchg2 instruction. Atomically write the least significant word of its 2nd argument to address specified by its 1st argument.
unsigned __int64 _InterlockedCompareExchange16_rel(volatile unsigned short *Destination, unsigned int64 Exchange, unsigned __int64 Comparand)	Compare and exchange atomically the least significant word at the address specified by its 1st argument. Maps to the cmpxchg2.rel instruction with appropriate setup.
unsigned __int64 _InterlockedCompareExchange16_acq(volatile unsigned short *Destination, unsigned int64 Exchange, unsigned __int64 Comparand)	Same the previous intrinsic, but using acquire semantic.
long _InterlockedIncrement(volatile long *addend	Atomically increment by one the value specified by its argument. Maps to the fetchadd4 instruction.
long _InterlockedDecrement(volatile long *addend	Atomically decrement by one the value specified by its argument. Maps to the fetchadd4 instruction.
long _InterlockedExchange(volatile long *Target, long value	Do an exchange operation atomically. Maps to the xchg4 instruction.
long _InterlockedCompareExchange(volatile long *Destination, long Exchange, long Comparand	Do a compare and exchange operation atomically. Maps to the cmpxchg4 instruction with appropriate setup.
long _InterlockedExchangeAdd(volatile long *addend, long increment	Use compare and exchange to do an atomic add of the increment value to the addend. Maps to a loop with the cmpxchg4 instruction to guarantee atomicity.
long _InterlockedAdd(volatile long *addend, long increment)	Same the previous intrinsic, but returns new value, not the original one.
void * _InterlockedCompareExchangePointer(void *	Map the cmpxchg8.acq instruction; Atomically

volatile *Destination, void *Exchange, void *Comparand)	compare and exchange the pointer value specified by its first argument (all arguments are pointers)
unsigned __int64 _InterlockedExchangeU(volatile unsigned int *Target, unsigned __int64 value)	Atomically exchange the 32-bit quantity specified by the 1st argument. Maps to the xchg4 instruction.
unsigned __int64 _InterlockedCompareExchange_rel(volatile unsigned int *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	Maps to the cmpxchg4.rel instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 64-bit pointer).
unsigned __int64 _InterlockedCompareExchange_acq(volatile unsigned int *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	Same the previous intrinsic but map the cmpxchg4.acq instruction.
void _ReleaseSpinLock(volatile int *x)	Release spin lock.
__int64 _InterlockedIncrement64(volatile __int64 *addend)	Increment by one the value specified by its argument. Maps to the fetchadd instruction.
__int64 _InterlockedDecrement64(volatile __int64 *addend)	Decrement by one the value specified by its argument. Maps to the fetchadd instruction.
__int64 _InterlockedExchange64(volatile __int64 *Target, __int64 value)	Do an exchange operation atomically. Maps to the xchg instruction.
unsigned __int64 _InterlockedExchangeU64(volatile unsigned __int64 *Target, unsigned __int64 value)	Same as InterlockedExchange64 (for unsigned quantities).
unsigned __int64 _InterlockedCompareExchange64_rel(volatile unsigned __int64 *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	Maps to the cmpxchg.rel instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 64-bit pointer).
unsigned __int64 _InterlockedCompareExchange64_acq(volatile unsigned __int64 *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	Maps to the cmpxchg.acq instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 64-bit pointer).
__int64	Same as the previous

`_InterlockedCompareExchange64(volatile __int64 *Destination, __int64 Exchange, __int64 Comparand)`	intrinsic for signed quantities.
`__int64 _InterlockedExchangeAdd64(volatile __int64 *addend, __int64 increment)`	Use compare and exchange to do an atomic add of the increment value to the addend. Maps to a loop with the `cmpxchg` instruction to guarantee atomicity
`__int64 _InterlockedAdd64(volatile __int64 *addend, __int64 increment);`	Same as the previous intrinsic, but returns the new value, not the original value. See Note.

Note

`_InterlockedSub64` is provided as a macro definition based on `_InterlockedAdd64`.

```
#define _InterlockedSub64(target, incr)
_InterlockedAdd64((target),(-(incr))).
```

Uses `cmpxchg` to do an atomic sub of the `incr` value to the `target`. Maps to a loop with the `cmpxchg` instruction to guarantee atomicity.

Load and Store

You can use the load and store intrinsic to force the strict memory access ordering of specific data objects. This intended use is for the case when the user suppresses the strict memory access ordering by using the `-serialize-volatile-` option.

Intrinsic	Prototype	Description
`__st1_rel`	`void __st1_rel(void *dst, const char value);`	Generates an `st1.rel` instruction.
`__st2_rel`	`void __st2_rel(void *dst, const short value);`	Generates an `st2.rel` instruction.
`__st4_rel`	`void __st4_rel(void *dst, const int value);`	Generates an `st4.rel` instruction.
`__st8_rel`	`void __st8_rel(void *dst, const __int64 value);`	Generates an `st8.rel` instruction.
`__ld1_acq`	`unsigned char __ld1_acq(void *src);`	Generates an `ld1.acq` instruction.
`__ld2_acq`	`unsigned short __ld2_acq(void *src);`	Generates an `ld2.acq` instruction.
`__ld4_acq`	`unsigned int __ld4_acq(void *src);`	Generates an `ld4.acq`

		instruction.
__ld8_acq	unsigned __int64 __ld8_acq(void *src);	Generates an ld8.acq instruction.

Operating System Related Intrinsics

The prototypes for these intrinsics are in the ia64intrin.h header file.

Intrinsic	Description
unsigned __int64 __getReg(const int whichReg)	Gets the value from a hardware register based on the index passed in. Produces a corresponding mov = r instruction. Provides access to the following registers: See Register Names for getReg() and setReg().
void __setReg(const int whichReg, unsigned __int64 value)	Sets the value for a hardware register based on the index passed in. Produces a corresponding mov = r instruction. See Register Names for getReg() and setReg().
unsigned __int64 __getIndReg(const int whichIndReg, __int64 index)	Return the value of an indexed register. The index is the 2nd argument; the register file is the first argument.
void __setIndReg(const int whichIndReg, __int64 index, unsigned __int64 value)	Copy a value in an indexed register. The index is the 2nd argument; the register file is the first argument.
void *__ptr64 _rdteb(void)	Gets TEB address. The TEB address is kept in r13 and maps to the move r=tp instruction
void __isrlz(void)	Executes the serialize instruction. Maps to the srlz.i instruction.
void __dsrlz(void)	Serializes the data. Maps to the srlz.d instruction.
unsigned __int64 __fetchadd4_acq(unsigned int *addend, const int increment)	Map the fetchadd4.acq instruction.
unsigned __int64 __fetchadd4_rel(unsigned int *addend, const int increment)	Map the fetchadd4.rel instruction.
unsigned __int64 __fetchadd8_acq(unsigned __int64 *addend, const int increment)	Map the fetchadd8.acq instruction.
unsigned __int64 __fetchadd8_rel(unsigned __int64 *addend, const int increment)	Map the fetchadd8.rel instruction.

`void __fwb(void)`	Flushes the write buffers. Maps to the `fwb` instruction.
`void __ldfs(const int whichFloatReg, void *src)`	Map the `ldfs` instruction. Load a single precision value to the specified register.
`void __ldfd(const int whichFloatReg, void *src)`	Map the `ldfd` instruction. Load a double precision value to the specified register.
`void __ldfe(const int whichFloatReg, void *src)`	Map the `ldfe` instruction. Load an extended precision value to the specified register.
`void __ldf8(const int whichFloatReg, void *src)`	Map the `ldf8` instruction.
`void __ldf_fill(const int whichFloatReg, void *src)`	Map the `ldf.fill` instruction.
`void __stfs(void *dst, const int whichFloatReg)`	Map the `sfts` instruction.
`void __stfd(void *dst, const int whichFloatReg)`	Map the `stfd` instruction.
`void __stfe(void *dst, const int whichFloatReg)`	Map the `stfe` instruction.
`void __stf8(void *dst, const int whichFloatReg)`	Map the `stf8` instruction.
`void __stf_spill(void *dst, const int whichFloatReg)`	Map the `stf.spill` instruction.
`void __mf(void)`	Executes a memory fence instruction. Maps to the `mf` instruction.
`void __mfa(void)`	Executes a memory fence, acceptance form instruction. Maps to the `mf.a` instruction.
`void __synci(void)`	Enables memory synchronization. Maps to the `sync.i` instruction.
`unsigned __int64 __thash(__int64)`	Generates a translation hash entry address. Maps to the `thash r = r` instruction.
`unsigned __int64 __ttag(__int64)`	Generates a translation hash entry tag. Maps to the `ttag r=r` instruction.
`void __itcd(__int64 pa)`	Insert an entry into the data translation cache (Map `itc.d` instruction).
`void __itci(__int64 pa)`	Insert an entry into the instruction translation cache (Map `itc.i`).
`void __itrd(__int64 whichTransReg, __int64 pa)`	Map the `itr.d` instruction.
`void __itri(__int64 whichTransReg, __int64 pa)`	Map the `itr.i` instruction.
`void __ptce(__int64 va)`	Map the `ptc.e` instruction.

`void __ptcl(__int64 va, __int64 pagesz)`	Purges the local translation cache. Maps to the `ptc.l r, r` instruction.
`void __ptcg(__int64 va, __int64 pagesz)`	Purges the global translation cache. Maps to the `ptc.g r, r` instruction.
`void __ptcga(__int64 va, __int64 pagesz)`	Purges the global translation cache and `ALAT`. Maps to the `ptc.ga r, r` instruction.
`void __ptri(__int64 va, __int64 pagesz)`	Purges the translation register. Maps to the `ptr.i r, r` instruction.
`void __ptrd(__int64 va, __int64 pagesz)`	Purges the translation register. Maps to the `ptr.d r, r` instruction.
`__int64 __tpa(__int64 va)`	Map the `tpa` instruction.
`void __invalat(void)`	Invalidates `ALAT`. Maps to the `invala` instruction.
`void __invala (void)`	Same as `void __invalat(void)`
`void __invala_gr(const int whichGeneralReg)`	`whichGeneralReg` = 0-127
`void __invala_fr(const int whichFloatReg)`	`whichFloatReg` = 0-127
`void __break(const int)`	Generates a break instruction with an immediate.
`void __nop(const int)`	Generate a `nop` instruction.
`void __debugbreak(void)`	Generates a Debug Break Instruction fault.
`void __fc(void*)`	Flushes a cache line associated with the address given by the argument. Maps to the `fc` instruction.
`void __sum(int mask)`	Sets the user mask bits of `PSR`. Maps to the `sum imm24` instruction.
`void __rum(int mask)`	Resets the user mask.
`__int64 _ReturnAddress(void)`	Get the caller's address.
`void __lfetch(int lfhint, const *y)`	Generate the `lfetch.lfhint` instruction. The value of the first argument specifies the hint type.
`void __lfetch_fault(int lfhint, const *y)`	Generate the `lfetch.fault.lfhint` instruction. The value of the first argument specifies the hint type.
`void __lfetch_excl(int lfhint, const *y)`	Generate the `lfetch.excl.lfhint` instruction. The value {0|1|2|3} of the first argument specifies the hint type.
`void __lfetch_fault_excl(int lfhint, void const *y)`	Generate the `lfetch.fault.excl.lfhint` instruction. The value of the first argument

	specifies the hint type.
`unsigned int __cacheSize(unsigned int cacheLevel)`	`__cacheSize(n)` returns the size in bytes of the cache at level `n`. 1 represents the first-level cache. 0 is returned for a non-existent cache level. For example, an application may query the cache size and use it to select block sizes in algorithms that operate on matrices.
`void __memory_barrier(void)`	Creates a barrier across which the compiler will not schedule any data access instruction. The compiler may allocate local data in registers across a memory barrier, but not global data.
`void __ssm(int mask)`	Sets the system mask. Maps to the `ssm imm24` instruction.
`void __rsm(int mask)`	Resets the system mask bits of PSR. Maps to the `rsm imm24` instruction.

Conversion Intrinsics

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Intrinsic	Description
`__int64 _m_to_int64(__m64 a)`	Convert `a` of type `__m64` to type `__int64`. Translates to `nop` since both types reside in the same register for systems based on IA-64 architecture.
`__m64 _m_from_int64(__int64 a)`	Convert `a` of type `__int64` to type `__m64`. Translates to `nop` since both types reside in the same register for systems based on IA-64 architecture.
`__int64 __round_double_to_int64(double d)`	Convert its double precision argument to a signed integer.
`unsigned __int64 __getf_exp(double d)`	Map the `getf.exp` instruction and return the 16-bit exponent and the sign of its operand.

Register Names for getReg() and setReg()

The prototypes for `getReg()` and `setReg()` intrinsics are in the `ia64regs.h` header file.

Name	whichReg
`_IA64_REG_IP`	1016
`_IA64_REG_PSR`	1019

_IA64_REG_PSR_L	1019

General Integer Registers

Name	whichReg
_IA64_REG_GP	1025
_IA64_REG_SP	1036
_IA64_REG_TP	1037

Application Registers

Name	whichReg
_IA64_REG_AR_KR0	3072
_IA64_REG_AR_KR1	3073
_IA64_REG_AR_KR2	3074
_IA64_REG_AR_KR3	3075
_IA64_REG_AR_KR4	3076
_IA64_REG_AR_KR5	3077
_IA64_REG_AR_KR6	3078
_IA64_REG_AR_KR7	3079
_IA64_REG_AR_RSC	3088
_IA64_REG_AR_BSP	3089
_IA64_REG_AR_BSPSTORE	3090
_IA64_REG_AR_RNAT	3091
_IA64_REG_AR_FCR	3093
_IA64_REG_AR_EFLAG	3096
_IA64_REG_AR_CSD	3097
_IA64_REG_AR_SSD	3098
_IA64_REG_AR_CFLAG	3099
_IA64_REG_AR_FSR	3100
_IA64_REG_AR_FIR	3101
_IA64_REG_AR_FDR	3102
_IA64_REG_AR_CCV	3104
_IA64_REG_AR_UNAT	3108
_IA64_REG_AR_FPSR	3112
_IA64_REG_AR_ITC	3116

_IA64_REG_AR_PFS	3136
_IA64_REG_AR_LC	3137
_IA64_REG_AR_EC	3138

Control Registers

Name	whichReg
_IA64_REG_CR_DCR	4096
_IA64_REG_CR_ITM	4097
_IA64_REG_CR_IVA	4098
_IA64_REG_CR_PTA	4104
_IA64_REG_CR_IPSR	4112
_IA64_REG_CR_ISR	4113
_IA64_REG_CR_IIP	4115
_IA64_REG_CR_IFA	4116
_IA64_REG_CR_ITIR	4117
_IA64_REG_CR_IIPA	4118
_IA64_REG_CR_IFS	4119
_IA64_REG_CR_IIM	4120
_IA64_REG_CR_IHA	4121
_IA64_REG_CR_LID	4160
_IA64_REG_CR_IVR	4161 ^
_IA64_REG_CR_TPR	4162
_IA64_REG_CR_EOI	4163
_IA64_REG_CR_IRR0	4164 ^
_IA64_REG_CR_IRR1	4165 ^
_IA64_REG_CR_IRR2	4166 ^
_IA64_REG_CR_IRR3	4167 ^
_IA64_REG_CR_ITV	4168
_IA64_REG_CR_PMV	4169
_IA64_REG_CR_CMCV	4170
_IA64_REG_CR_LRR0	4176
_IA64_REG_CR_LRR1	4177

Indirect Registers for getIndReg() and setIndReg()

Name	whichReg
_IA64_REG_INDR_CPUID	9000 ^
_IA64_REG_INDR_DBR	9001
_IA64_REG_INDR_IBR	9002
_IA64_REG_INDR_PKR	9003
_IA64_REG_INDR_PMC	9004
_IA64_REG_INDR_PMD	9005
_IA64_REG_INDR_RR	9006
_IA64_REG_INDR_RESERVED	9007

Multimedia Additions

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Details about each intrinsic follows the table below.

Intrinsic	Operation	Corresponding IA-64 Instruction
_m64_czx1l	Compute Zero Index	czx1.l
_m64_czx1r	Compute Zero Index	czx1.r
_m64_czx2l	Compute Zero Index	czx2.l
_m64_czx2r	Compute Zero Index	czx2.r
_m64_mix1l	Mix	mix1.l
_m64_mix1r	Mix	mix1.r
_m64_mix2l	Mix	mix2.l
_m64_mix2r	Mix	mix2.r
_m64_mix4l	Mix	mix4.l
_m64_mix4r	Mix	mix4.r
_m64_mux1	Permutation	mux1
_m64_mux2	Permutation	mux2
_m64_padd1uus	Parallel add	padd1.uus
_m64_padd2uus	Parallel add	padd2.uus
_m64_pavg1_nraz	Parallel average	pavg1
_m64_pavg2_nraz	Parallel average	pavg2

_m64_pavgsub1	Parallel average subtract	pavgsub1
_m64_pavgsub2	Parallel average subtract	pavgsub2
_m64_pmpy2r	Parallel multiply	pmpy2.r
_m64_pmpy2l	Parallel multiply	pmpy2.l
_m64_pmpyshr2	Parallel multiply and shift right	pmpyshr2
_m64_pmpyshr2u	Parallel multiply and shift right	pmpyshr2.u
_m64_pshladd2	Parallel shift left and add	pshladd2
_m64_pshradd2	Parallel shift right and add	pshradd2
_m64_psub1uus	Parallel subtract	psub1.uus
_m64_psub2uus	Parallel subtract	psub2.uus

```
__int64 _m64_czx1l(__m64 a)
```

The 64-bit value `a` is scanned for a zero element from the most significant element to the least significant element, and the index of the first zero element is returned. The element width is 8 bits, so the range of the result is from 0 - 7. If no zero element is found, the default result is 8.

```
__int64 _m64_czx1r(__m64 a)
```

The 64-bit value `a` is scanned for a zero element from the least significant element to the most significant element, and the index of the first zero element is returned. The element width is 8 bits, so the range of the result is from 0 - 7. If no zero element is found, the default result is 8.

```
__int64 _m64_czx2l(__m64 a)
```

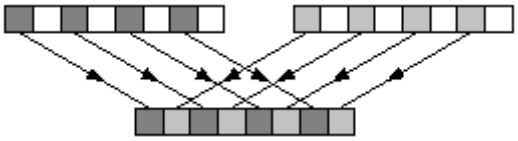
The 64-bit value `a` is scanned for a zero element from the most significant element to the least significant element, and the index of the first zero element is returned. The element width is 16 bits, so the range of the result is from 0 - 3. If no zero element is found, the default result is 4.

```
__int64 _m64_czx2r(__m64 a)
```

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

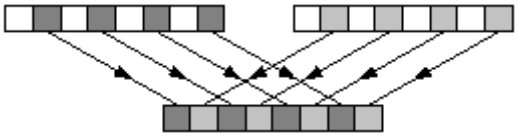
```
__m64 _m64_mix1l(__m64 a, __m64 b)
```

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



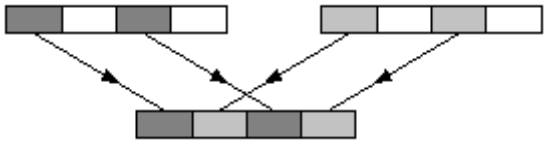
```
__m64 _m64_mix1r(__m64 a, __m64 b)
```

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



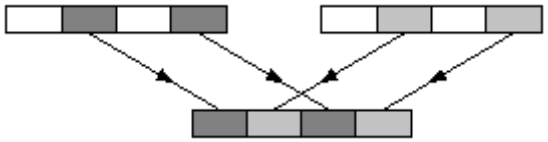
```
__m64 _m64_mix2l(__m64 a, __m64 b)
```

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



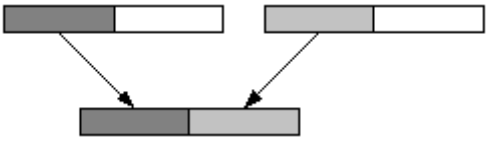
```
__m64 _m64_mix2r(__m64 a, __m64 b)
```

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



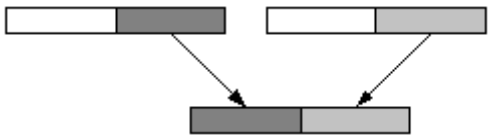
```
__m64 _m64_mix4l(__m64 a, __m64 b)
```

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



```
__m64 _m64_mix4r(__m64 a, __m64 b)
```

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



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

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

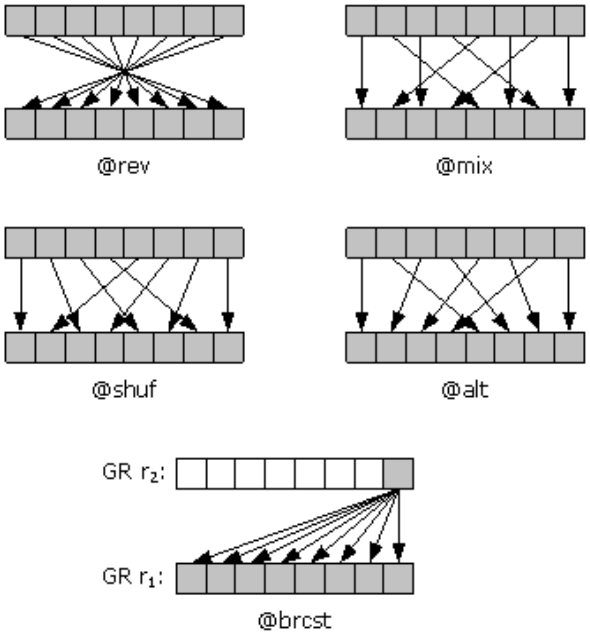
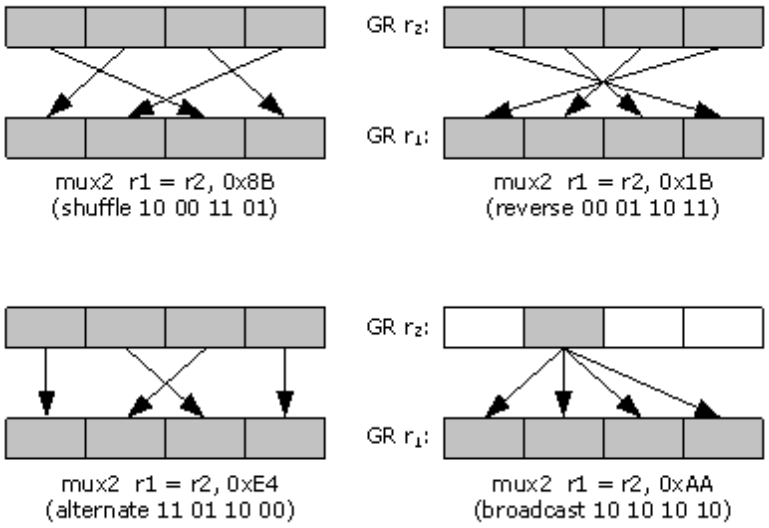


Table 1. Values of n for m64_mux1 Operation

	n
@brcst	0
@mix	8
@shuf	9
@alt	0xA
@rev	0xB

```
__m64 _m64_mux2(__m64 a, const int n)
```

Based on the value of `n`, a permutation is performed on a as shown in Figure 8, and the result is returned.



`__m64 _m64_pavgsub1(__m64 a, __m64 b)`

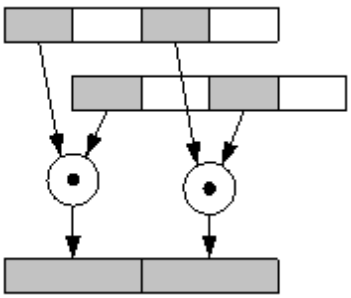
The unsigned data elements (bytes) of `b` are subtracted from the unsigned data elements (bytes) of `a` and the results of the subtraction are then each independently shifted to the right by one position. The high-order bits of each element are filled with the borrow bits of the subtraction.

`__m64 _m64_pavgsub2(__m64 a, __m64 b)`

The unsigned data elements (double bytes) of `b` are subtracted from the unsigned data elements (double bytes) of `a` and the results of the subtraction are then each independently shifted to the right by one position. The high-order bits of each element are filled with the borrow bits of the subtraction.

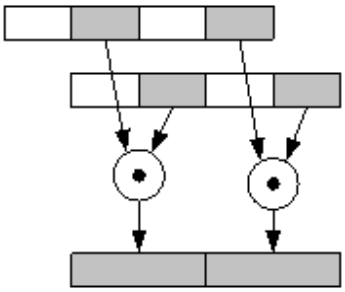
`__m64 _m64_pmpy2l(__m64 a, __m64 b)`

Two signed 16-bit data elements of `a`, starting with the most significant data element, are multiplied by the corresponding two signed 16-bit data elements of `b`, and the two 32-bit results are returned as shown in Figure 9.



```
__m64 _m64_pmpy2r(__m64 a, __m64 b)
```

Two signed 16-bit data elements of a, starting with the least significant data element, are multiplied by the corresponding two signed 16-bit data elements of b, and the two 32-bit results are returned as shown in Figure 10.



```
__m64 _m64_pmpyshr2(__m64 a, __m64 b, const int count)
```

The four signed 16-bit data elements of a are multiplied by the corresponding signed 16-bit data elements of b, yielding four 32-bit products. Each product is then shifted to the right count bits and the least significant 16 bits of each shifted product form 4 16-bit results, which are returned as one 64-bit word.

```
__m64 _m64_pmpyshr2u(__m64 a, __m64 b, const int count)
```

The four unsigned 16-bit data elements of a are multiplied by the corresponding unsigned 16-bit data elements of b, yielding four 32-bit products. Each product is then shifted to the right count bits and the least significant 16 bits of each shifted product form 4 16-bit results, which are returned as one 64-bit word.

```
__m64 _m64_pshladd2(__m64 a, const int count, __m64 b)
```

a is shifted to the left by count bits and then is added to b. The upper 32 bits of the result are forced to 0, and then bits [31:30] of b are copied to bits [62:61] of the result. The result is returned.

```
__m64 _m64_pshradd2(__m64 a, const int count, __m64 b)
```

The four signed 16-bit data elements of a are each independently shifted to the right by count bits (the high order bits of each element are filled with the initial value of the sign bits of the data elements in a); they are then added to the four signed 16-bit data elements of b. The result is returned.

```
__m64 _m64_padd1uus(__m64 a, __m64 b)
```

a is added to b as eight separate byte-wide elements. The elements of a are treated as unsigned, while the elements of b are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_padd2uus(__m64 a, __m64 b)
```

a is added to b as four separate 16-bit wide elements. The elements of a are treated as unsigned, while the elements of b are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_psub1uus(__m64 a, __m64 b)
```

a is subtracted from b as eight separate byte-wide elements. The elements of a are treated as unsigned, while the elements of b are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_psub2uus(__m64 a, __m64 b)
```

a is subtracted from b as four separate 16-bit wide elements. The elements of a are treated as unsigned, while the elements of b are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_pavg1_nraz(__m64 a, __m64 b)
```

The unsigned byte-wide data elements of a are added to the unsigned byte-wide data elements of b and the results of each add are then independently shifted to the right by one position. The high-order bits of each element are filled with the carry bits of the sums.

```
__m64 _m64_pavg2_nraz(__m64 a, __m64 b)
```

The unsigned 16-bit wide data elements of a are added to the unsigned 16-bit wide data elements of b and the results of each add are then independently shifted to the right by one position. The high-order bits of each element are filled with the carry bits of the sums.

Synchronization Primitives

The synchronization primitive intrinsics provide a variety of operations. Besides performing these operations, each intrinsic has two key properties:

- the function performed is guaranteed to be atomic
- associated with each intrinsic are certain memory barrier properties that restrict the movement of memory references to visible data across the intrinsic operation by either the compiler or the processor

For the following intrinsics, <type> is either a 32-bit or 64-bit integer.

Atomic Fetch-and-op Operations

```
<type> __sync_fetch_and_add(<type> *ptr,<type> val)
<type> __sync_fetch_and_and(<type> *ptr,<type> val)
<type> __sync_fetch_and_nand(<type> *ptr,<type> val)
<type> __sync_fetch_and_or(<type> *ptr,<type> val)
<type> __sync_fetch_and_sub(<type> *ptr,<type> val)
<type> __sync_fetch_and_xor(<type> *ptr,<type> val)
```

Atomic Op-and-fetch Operations

```
<type> __sync_add_and_fetch(<type> *ptr,<type> val)
<type> __sync_sub_and_fetch(<type> *ptr,<type> val)
<type> __sync_or_and_fetch(<type> *ptr,<type> val)
<type> __sync_and_and_fetch(<type> *ptr,<type> val)
<type> __sync_nand_and_fetch(<type> *ptr,<type> val)
<type> __sync_xor_and_fetch(<type> *ptr,<type> val)
```

Atomic Compare-and-swap Operations

```
<type> __sync_val_compare_and_swap(<type> *ptr, <type> old_val, <type>
new_val)
int __sync_bool_compare_and_swap(<type> *ptr, <type> old_val, <type>
new_val)
```

Atomic Synchronize Operation

```
void __sync_synchronize (void);
```

Atomic Lock-test-and-set Operation

```
<type> __sync_lock_test_and_set(<type> *ptr,<type> val)
```

Atomic Lock-release Operation

```
void __sync_lock_release(<type> *ptr)
```

Miscellaneous Intrinsics

```
void* __get_return_address(unsigned int level);
```

This intrinsic yields the return address of the current function. The `level` argument must be a constant value. A value of 0 yields the return address of the current function. Any other value yields a zero return address. On Linux systems, this intrinsic is synonymous with `__builtin_return_address`. The name and the argument are provided for compatibility with gcc*.

```
void __set_return_address(void* addr);
```

This intrinsic overwrites the default return address of the current function with the address indicated by its argument. On return from the current invocation, program execution continues at the address provided.

```
void* __get_frame_address(unsigned int level);
```

This intrinsic returns the frame address of the current function. The `level` argument must be a constant value. A value of 0 yields the frame address of the current function. Any other value yields a zero return value. On Linux systems, this intrinsic is synonymous with `__builtin_frame_address`. The name and the argument are provided for compatibility with gcc.

Intrinsics for Dual-Core Intel® Itanium® 2 processor 9000 series

The Dual-Core Intel® Itanium® 2 processor 9000 series supports the intrinsics listed in the table below.

These intrinsics each generate IA-64 instructions. The first alpha-numerical chain in the intrinsic name represents the return type, and the second alpha-numerical chain in the intrinsic name represents the instruction the intrinsic generates. For example, the intrinsic `_int64_cmp8xchg` generates the `_int64` return type and the `cmp8xchg` IA-64 instruction.

Detailed information about each intrinsic follows the table.

Click here for Examples of several of these intrinsics.

For more information about the instructions these intrinsics generate, please see the documentation area of the Itanium 2 processor website at http://developer.intel.com/products/processor/itanium2/index.htm


Note

Calling these intrinsics on any previous Itanium® processor causes an illegal instruction fault.

Intrinsic Name	Operation
`__cmp8xchg16`	Compare and Exchange
`__ld16`	Load
`__fc_i`	Flush cache
`__hint`	Provide performance hints
`__st16`	Store

```
__int64  __cmp8xchg16(const int <sem>, const int <ldhint>, void *<addr>,
__int64 <xchg_lo>)
```

Generates the 16-byte form of the IA-64 `compare` and `exchange` instruction.

Returns the original 64-bit value read from memory at the specified address.

The following table describes each argument for this intrinsic.

sem	ldhint	addr	xchg_lo
Literal value between 0	Literal value between 0	The	The least

and 1 that specifies the semaphore completer (0==.acq, 1==.rel)	and 2 that specifies the load hint completer (0==.none, 1==.nt1, 2==.nta).	address of the value to read.	significant 8 bytes of the exchange value.

The following table describes each implicit argument for this intrinsic.

xchg_hi	cmpnd
Highest 8 bytes of the exchange value. Use the setReg intrinsic to set the <xchg_hi> value in the register AR[CSD]. [__setReg (_IA64_REG_AR_CSD, <xchg_hi>);].	The 64-bit compare value. Use the __setReg intrinsic to set the <cmpnd> value in the register AR[CCV]. [__setReg (_IA64_REG_AR_CCV,<cmpnd>);]

```
__int64 __ld16(const int <ldtype>, const int <ldhint>, void *<addr>)
```

Generates the IA-64 instruction that loads 16 bytes from the given address.

Returns the lower 8 bytes of the quantity loaded from <addr>. The higher 8 bytes are loaded in register AR[CSD].

Generates implicit return of the higher 8 bytes to the register AR[CSD]. You can use the __getReg intrinsic to copy the value into a user variable. [foo = __getReg(_IA64_REG_AR_CSD);]

The following table describes each argument for this intrinsic.

ldtype	ldhint	addr
A literal value between 0 and 1 that specifies the load type (0==none, 1==.acq).	A literal value between 0 and 2 that specifies the hint completer (0==none, 1==.nt1, 2== .nta).	The address to load from.

```
void __fc_i(void *<addr>)
```

Generates the IA-64 instruction that flushes the cache line associated with the specified address and ensures coherency between instruction cache and data cache.

The following table describes the argument for this intrinsic.

cache_line
An address associated with the cache line you want to flush

```
void __hint(const int <hint_value>)
```

Generates the IA-64 instruction that provides performance hints about the program being executed.

The following table describes the argument for this intrinsic.

hint_value
A literal value that specifies the hint. Currently, zero is the only legal value. `__hint(0)` generates the IA-64 `hint@pause` instruction.

```
void __st16(const int <sttype>, const int <sthint>, void *<addr>,
__int64 <src_lo>)
```

Generates the IA-64 instruction to store 16 bytes at the given address.

The following table describes each argument for this intrinsic.

sttype	sthint	addr	src_lo
A literal value between 0 and 1 that specifies the store type completer (0==.none, 1==.rel).	A literal value between 0 and 1 that specifies the store hint completer (0==.none, 1==.nta).	The address where the 16-byte value is stored.	The lowest 8 bytes of the 16-byte value to store.

The following table describes the implicit argument for this intrinsic.

src_hi
The highest 8 bytes of the 16-byte value to store. Use the setReg intrinsic to set the `<src_hi>` value in the register AR[CSD]. [`__setReg(_IA64_REG_AR_CSD, <src_hi>);`]

Examples

The following examples show how to use the intrinsics listed above to generate the corresponding instructions. In all cases, use the __setReg (resp. __getReg) intrinsic to set up implicit arguments (resp. retrieve implicit return values).

```
// file foo.c

//

#include <ia64intrin.h>



void foo_ld16(__int64* lo, __int64* hi, void* addr)

{

    /**/
```

```
    // The following two calls load the 16-byte value at the given
address
    // into two (2) 64-bit integers
    // The higher 8 bytes are returned implicitly in the CSD register;
// The call to __getReg moves that value into a user variable (hi).
// The instruction generated is a plain ld16
//    ld16 Ra,ar.csd=[Rb]
    *lo = __ld16(__ldtype_none, __ldhint_none, addr);
    *hi = __getReg(_IA64_REG_AR_CSD);
    /**/
}


void foo_ld16_acq(__int64* lo, __int64* hi, void* addr)
{
    /**/
    // This is the same as the previous example, except that it uses the
    // __ldtype_acq completer to generate the acquire_from of the ld16:
    //      ld16.acq     Ra,ar.csd=[Rb]
    //
    *lo = __ld16(__ldtype_acq, __ldhint_none, addr);
    *hi = __getReg(_IA64_REG_AR_CSD);
    /**/
}
void foo_st16(__int64 lo, __int64 hi, void* addr)
{
    /**/
    // first set the highest 64-bits into CSD register. Then call
    // __st16 with the lowest 64-bits as argument
```

```
    //

    __setReg(_IA64_REG_AR_CSD, hi);

    __st16(__sttype_none, __sthint_none, addr, lo);

    /**/

}



__int64 foo_cmp8xchg16(__int64 xchg_lo, __int64 xchg_hi, __int64 cmpnd,
void* addr)

{

    __int64 old_value;

    /**/

    // set the highest bits of the exchange value and the comperand
value

    // respectively in CSD and CCV. Then, call the exchange intrinsic

    //

    __setReg(_IA64_REG_AR_CSD, xchg_hi);

    __setReg(_IA64_REG_AR_CCV, cmpnd);

    old_value = __cmp8xchg16(__semtype_acq, __ldhint_none, addr,
xchg_lo);

    /**/

    return old_value;

}

// end foo.c
```

Microsoft-compatible Intrinsics for Dual-Core Intel® Itanium® 2 processor 9000 series

The Dual-Core Intel® Itanium® 2 processor 9000 series supports the intrinsics listed in the table below. These intrinsics are also compatible with the Microsoft compiler. These intrinsics each generate IA-64 instructions. The second alpha-numerical chain in the intrinsic name represents the IA-64 instruction the intrinsic generates. For example, the intrinsic `_int64_cmp8xchg` generates the `cmp8xchg` IA-64 instruction.

For more information about the instructions these intrinsics generate, please see the documentation area of the Itanium 2 processor website at http://developer.intel.com/products/processor/itanium2/index.htm.

Detailed information about each intrinsic follows the table.

Intrinsic Name	Operation	Corresponding IA-64 Instruction
_InterlockedCompare64Exchange128	Compare and exchange	
_InterlockedCompare64Exchange128_acq	Compare and Exchange	
_InterlockedCompare64Exchange128_rel	Compare and Exchange	
__load128	Read	
__load128_acq	Read	
__store128	Store	
__store128_rel	Store	

```
__int64 _InterlockedCompare64Exchange128( __int64 volatile *
<Destination>, __int64 <ExchangeHigh>, __int64 <ExchangeLow>, __int64
<Comperand>)
```

Generates a compare and exchange IA-64 instruction.

Returns the lowest 64-bit value of the destination.

The following table describes each argument for this intrinsic.

Destination	ExchangeHigh	ExchangeLow	Comperand
Pointer to the 128-bit Destination value	Highest 64 bits of the Exchange value	Lowest 64 bits of the Exchange value	Value to compare with Destination

```
__int64 _InterlockedCompare64Exchange128_acq(  __int64 volatile *
<Destination>, __int64 <ExchangeHigh>,  __int64 <ExchangeLow>, __int64
<Comperand>)
```

Generates a compare and exchange IA-64 instruction. Same as `_InterlockedCompare64Exchange128,` but this intrinsic uses acquire semantics.

Returns the lowest 64-bit value of the destination.

The following table describes each argument for this intrinsic.

Destination	ExchangeHigh	ExchangeLow	Comperand
Pointer to the 128-bit Destination value	Highest 64 bits of the Exchange value	Lowest 64 bits of the Exchange value	Value to compare with Destination

```
__int64 _InterlockedCompare64Exchange128_rel(  __int64 volatile *
<Destination>,  __int64 <ExchangeHigh>,   __int64 <ExchangeLow>, __int64
<Comperand>
```

Generates a compare and exchange IA-64 instruction. Same as `_InterlockedCompare64Exchange128`, but this intrinsic uses release semantics.

Returns the lowest 64-bit value of the destination.

The following table describes each argument for this intrinsic.

Destination	ExchangeHigh	ExchangeLow	Comperand
Pointer to the 128-bit Destination value	Highest 64 bits of the Exchange value	Lowest 64 bits of the Exchange value	Value to compare with Destination

```
__int64 __load128(   __int64 volatile * Source, __int64
*<DestinationHigh>)
```

Generates the IA-64 instruction that atomically reads 128 bits from the memory location.

Returns the lowest 64-bit value of the 128-bit loaded value.

The following table describes each argument for this intrinsic.

Source	DestinationHigh
Pointer to the 128-bit Source value	Pointer to the location in memory that stores the highest 64 bits of the 128-bit loaded value

```
__int64 __load128_acq( __int64 volatile * <Source>, __int64
*<DestinationHigh>
```

Generates the IA-64 instruction that atomically reads 128 bits from the memory location. Same as `__load128`, but the this intrinsic uses acquire semantics.

Returns the lowest 64-bit value of the 128-bit loaded value.

The following table describes each argument for this intrinsic.

Source	DestinationHigh
Pointer to the 128-bit Source value	Pointer to the location in memory that stores the highest 64 bits of the 128-bit loaded value

```
void __store128( __int64 volatile * <Destination>, __int64 <SourceHigh>
__int64 <SourceLow>)
```

Generates the IA-64 instruction that atomically stores 128 bits at the destination memory location.

No returns.

Destination	SourceHigh	SourceLow
Pointer to the 128-bit Destination value	The highest 64 bits of the value to be stored	The lowest 64 bits of the value to be stored

```
void __store128_rel( __int64 volatile * <Destination>, __int64
<SourceHigh> __int64 <SourceLow>)
```

Generates the IA-64 instruction that atomically stores 128 bits at the destination memory location. Same as `__store128`, but this intrinsic uses release semantics.

No returns.

Destination	SourceHigh	SourceLow
Pointer to the 128-bit Destination value	The highest 64 bits of the value to be stored	The lowest 64 bits of the value to be stored

Data Alignment, Memory Allocation Intrinsics, and Inline Assembly

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

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

- Alignment Support
- Allocating and Freeing Aligned Memory Blocks
- Inline Assembly

Alignment Support

Aligning data improves the performance of intrinsics. When using the Streaming SIMD Extensions, you should align data to 16 bytes in memory operations. 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 would. For example, a data object of type int is allocated at a byte address which is a multiple of 4 by default. 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 restriction on IA-32:

- 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, up to 4096. The value specified is the requested alignment.


Caution

In this release, `__declspec(align(8))` does not function correctly. Use `__declspec(align(16))` instead.

Note

If a value is specified that is less than the alignment of the affected data type, it has no effect. In other words, data is aligned to the maximum of its own alignment or the alignment specified with `__declspec(align)`.

You can request alignments for individual variables, whether of static or automatic storage duration. (Global and static variables have static storage duration; local variables have automatic storage duration by default.) You cannot adjust the alignment of a parameter, nor a field of a `struct` or `class`. You can, however, increase the alignment of a `struct` (or `union` or `class`), in which case every object of that type is affected.

As an example, suppose that a function uses local variables `i` and `j` as subscripts into a 2-dimensional array. They might be declared as follows:

```
int i, j;
```

These variables are commonly used together. But they can fall in different cache lines, which could be detrimental to performance. You can instead declare them as follows:

```
__declspec(align(16)) struct { int i, j; } sub;
```

The compiler now ensures that they are allocated in the same cache line. In C++, you can omit the `struct` variable name (written as sub in the previous example). In C, however, it is required, and you must write references to `i` and `j` as `sub.i` and `sub.j`.

If you use many functions with such subscript pairs, it is more convenient to declare and use a `struct` type for them, as in the following example:

```
typedef struct __declspec(align(16)) { 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. Note that allocation of parameters is unaffected by `__declspec(align)`. (If necessary, you can assign the value of a parameter to a local variable with the appropriate alignment.)

You can also force alignment of global variables, such as arrays:

```
__declspec(align(16)) float array[1000];
```

Allocating and Freeing Aligned Memory Blocks

Use the `_mm_malloc` and `_mm_free` intrinsics to allocate and free aligned blocks of memory. These intrinsics are based on `malloc` and `free`, which are in the `libirc.a` library. You need to include `malloc.h`. The syntax for these intrinsics is as follows:

```
void* _mm_malloc (int size, int align)
```

```
void _mm_free (void *p)
```

The _mm_malloc routine takes an extra parameter, which is the alignment constraint. This constraint must be a power of two. The pointer that is returned from _mm_malloc is guaranteed to be aligned on the specified boundary.

Note

Memory that is allocated using _mm_malloc must be freed using _mm_free . Calling free on memory allocated with _mm_malloc or calling _mm_free on memory allocated with malloc will cause unpredictable behavior.

Inline Assembly

Microsoft Style Inline Assembly

The Intel® C++ Compiler supports Microsoft-style inline assembly with the -use-msasm compiler option. See your Microsoft documentation for the proper syntax.

GNU*-like Style Inline Assembly (IA-32 architecture and Intel® 64 architecture only)

The Intel® C++ Compiler supports GNU-like style inline assembly. The syntax is as follows:

```
asm-keyword [ volatile-keyword ] ( asm-template [ asm-interface ] ) ;
```

The Intel C++ Compiler also supports mixing UNIX and Microsoft style asms. Use the __asm__ keyword for GNU-style ASM when using the -use_msasm switch.

Note

The Intel C++ Compiler supports gcc-style inline ASM if the assembler code uses AT&T* System V/386 syntax.

Syntax Element	Description
asm-keyword	asm statements begin with the keyword asm. Alternatively, either __asm or __asm__ may be used for compatibility. When mixing UNIX and Microsoft style asm, use the __asm__ keyword. The compiler only accepts the __asm__ keyword. The asm and __asm keywords are reserved for Microsoft style assembly statements.
volatile-keyword	If the optional keyword volatile is given, the asm is volatile. Two volatile asm statements will never be moved past each other, and a reference to a volatile variable will not be moved relative to a volatile asm. Alternate keywords __volatile and __volatile__ may be used for compatibility.
asm-template	The asm-template is a C language ASCII string which specifies how to output the assembly code for an instruction. Most of the template is a

	fixed string; everything but the substitution-directives, if any, is passed through to the assembler. The syntax for a substitution directive is a % followed by one or two characters.
asm-interface	The asm-interface consists of three parts: 1. an optional output-list 2. an optional input-list 3. an optional clobber-list These are separated by colon (:) characters. If the output-list is missing, but an input-list is given, the input list may be preceded by two colons (::)to take the place of the missing output-list. If the asm-interface is omitted altogether, the asm statement is considered volatile regardless of whether a volatile-keyword was specified.
output-list	An output-list consists of one or more output-specs separated by commas. For the purposes of substitution in the asm-template, each output-spec is numbered. The first operand in the output-list is numbered 0, the second is 1, and so on. Numbering is continuous through the output-list and into the input-list. The total number of operands is limited to 30 (i.e. 0-29).
input-list	Similar to an output-list, an input-list consists of one or more input-specs separated by commas. For the purposes of substitution in the asm-template, each input-spec is numbered, with the numbers continuing from those in the output-list.
clobber-list	A clobber-list tells the compiler that the asm uses or changes a specific machine register that is either coded directly into the asm or is changed implicitly by the assembly instruction. The clobber-list is a comma-separated list of clobber-specs.
input-spec	The input-specs tell the compiler about expressions whose values may be needed by the inserted assembly instruction. In order to describe fully the input requirements of the asm, you can list input-specs that are not actually referenced in the asm-template.
clobber-spec	Each clobber-spec specifies the name of a single machine register that is clobbered. The register name may optionally be preceded by a %. You can specify any valid machine register name. It is also legal to specify "memory" in a clobber-spec. This prevents the compiler from keeping data cached in registers across the asm statement.

When compiling an assembly statement on Linux, the compiler simply emits the asm-template to the assembly file after making any necessary operand substitutions. The compiler then calls the GNU assembler to generate machine code. In contrast, on Windows the compiler itself must assemble the text contained in the asm-template string into machine code. In essence, the compiler contains a built-in assembler.

The compiler's built-in assembler does not support the full functionality of the GNU assembler, so there are limitations in the contents of the asm-template. In particular, the following assembler features are not currently supported.

- Directives

- Labels
- Symbols*

Note

* Direct symbol references in the asm-template are not supported. To access a C++ object, use the asm-interface with a substitution directive.

Example

Incorrect method for accessing a C++ object:

```
__asm__("addl $5, _x");
```

Proper method for accessing a C++ object

```
__asm__("addl  $5, %0" : "+rm" (x));
```

GNU-style inline assembly statements on Windows use the same assembly instruction format as on Linux. This means that destination operands are on the right and source operands are on the left. This operand order is the reverse of Intel assembly syntax.

Due to the limitations of the compiler's built-in assembler, many assembly statements that compile and run on Linux will not compile on Windows. On the other hand, assembly statements that compile and run on Windows should also compile and run on Linux.

This feature provides a high-performance alternative to Microsoft-style inline assembly statements when portability between Windows, Linux, and Mac OS is important. Its intended use is in small primitives where high-performance integration with the surrounding C++ code is essential.

Example

```
#ifdef  WIN64
#define INT64 PRINTF FORMAT "I64"
#else
#define   int64 long long
#define INT64 PRINTF FORMAT "L"
#endif
#include <stdio.h>
typedef struct {
      int64 lo64;
      int64 hi64;
} my i128;
#define ADD128(out, in1, in2)                        \
      asm  ("addq %2, %0; adcq %3, %1" :             \
            "=r"(out.lo64), "=r"(out.hi64) :         \
            "emr" (in2.lo64), "emr"(in2.hi64),       \
```

```
            "0" (in1.lo64), "1" (in1.hi64));
extern int
main()
{
    my_i128 val1, val2, result;
    val1.lo64 = ~0;
    val1.hi64 = 0;
    val2.lo64 = 1;
    val2.hi64 = 65;
    ADD128(result, val1, val2);
    printf("  0x%016" INT64 PRINTF FORMAT
"x%016"   INT64 PRINTF FORMAT "x\n",
           val1.hi64, val1.lo64);
    printf("+ 0x%016" INT64 PRINTF FORMAT "x%016" INT64 PRINTF FORMAT
"x\n",
           val2.hi64, val2.lo64);
    printf("-----------------------------------\n");
    printf("  0x%016" INT64 PRINTF FORMAT "x%016" INT64 PRINTF FORMAT
"x\n",
           result.hi64, result.lo64);
    return 0;
}
```

This example, written for Intel® 64 architecture, shows how to use a GNU-style inline assembly statement to add two 128-bit integers. In this example, a 128-bit integer is represented as two __int64 objects in the my_i128 structure. The inline assembly statement used to implement the addition is contained in the ADD128 macro, which takes 3 my_i128 arguments representing 3 128-bit integers. The first argument is the output. The next two arguments are the inputs. The example compiles and runs using the Intel Compiler on Linux or Windows, producing the following output.

```
  0x0000000000000000ffffffffffffffff

+ 0x0000000000000041000000000000001

-----------------------------------

+ 0x0000000000000042000000000000000
```

In the GNU-style inline assembly implementation, the asm interface specifies all the inputs, outputs, and side effects of the asm statement, enabling the compiler to generate very efficient code.

```
mov       r13, 0xffffffffffffffff

mov       r12, 0x000000000

add       r13, 1

adc       r12, 65
```

It is worth noting that when the compiler generates an assembly file on Windows, it uses Intel syntax even though the assembly statement was written using Linux assembly syntax.

The compiler moves in1.lo64 into a register to match the constraint of operand 4. Operand 4's constraint of "0" indicates that it must be assigned the same location as output operand 0. And operand 0's constraint is "=r", indicating that it must be assigned an integer register. In this case, the compiler chooses r13. In the same way, the compiler moves in1.hi64 into register r12.

The constraints for input operands 2 and 3 allow the operands to be assigned a register location ("r"), a memory location ("m"), or a constant signed 32-bit integer value ("e"). In this case, the compiler chooses to match operands 2 and 3 with the constant values 1 and 65, enabling the add and adc instructions to utilize the "register-immediate" forms.

The same operation is much more expensive using a Microsoft-style inline assembly statement, because the interface between the assembly statement and the surrounding C++ code is entirely through memory. Using Microsoft assembly, the ADD128 macro might be written as follows.

```
#define ADD128(out, in1, in2)                       \
    {                                               \
        __asm mov rax, in1.lo64                     \
        __asm mov rdx, in1.hi64                     \
        __asm add rax, in2.lo64                     \
        __asm adc rdx, in2.hi64                     \
        __asm mov out.lo64, rax                     \
        __asm mov out.hi64, rdx                     \
    }
```

The compiler must add code before the assembly statement to move the inputs into memory, and it must add code after the assembly statement to retrieve the outputs from memory. This prevents the compiler from exploiting some optimization opportunities. Thus, the following assembly code is produced.

```
        mov       QWORD PTR [rsp+32], -1
        mov       QWORD PTR [rsp+40], 0
        mov       QWORD PTR [rsp+48], 1
        mov       QWORD PTR [rsp+56], 65
; Begin ASM
        mov       rax, QWORD PTR [rsp+32]
        mov       rdx, QWORD PTR [rsp+40]
```

```
        add       rax, QWORD PTR [rsp+48]
        adc       rdx, QWORD PTR [rsp+56]
        mov       QWORD PTR [rsp+64], rax
        mov       QWORD PTR [rsp+72], rdx
; End ASM
        mov       rdx, QWORD PTR [rsp+72]
        mov       r8, QWORD PTR [rsp+64]
```

The operation that took only 4 instructions and 0 memory references using GNU-style inline assembly takes 12 instructions with 12 memory references using Microsoft-style inline assembly.

Intrinsics Cross-processor Implementation

Overview: Intrinsics Cross-processor Implementation

This section provides a series of tables that compare intrinsics performance across architectures. Before implementing intrinsics across architectures, please note the following.

- Instrinsics may generate code that does not run on all IA processors. You should therefore use `CPUID` to detect the processor and generate the appropriate code.
- Implement intrinsics by processor family, not by specific processor. The guiding principle for which family -- IA-32 or Itanium® processors -- the intrinsic is implemented on is performance, not compatibility. Where there is added performance on both families, the intrinsic will be identical.

Intrinsics For Implementation Across All IA

The following intrinsics provide significant performance gain over a non-intrinsic-based code equivalent.

`int abs(int)`
`long labs(long)`
`unsigned long _lrotl(unsigned long value, int shift)`
`unsigned long _lrotr(unsigned long value, int shift)`
`unsigned int _rotl(unsigned int value, int shift)`
`unsigned int _rotr(unsigned int value, int shift)`
`__int64 __i64_rotl(__int64 value, int shift)`
`__int64 __i64_rotr(__int64 value, int shift)`
`double fabs(double)`
`double log(double)`
`float logf(float)`
`double log10(double)`
`float log10f(float)`
`double exp(double)`
`float expf(float)`
`double pow(double, double)`
`float powf(float, float)`
`double sin(double)`
`float sinf(float)`
`double cos(double)`

float cosf(float)
double tan(double)
float tanf(float)
double acos(double)
float acosf(float)
double acosh(double)
float acoshf(float)
double asin(double)
float asinf(float)
double asinh(double)
float asinhf(float)
double atan(double)
float atanf(float)
double atanh(double)
float atanhf(float)
float cabs(double)*
double ceil(double)
float ceilf(float)
double cosh(double)
float coshf(float)
float fabsf(float)
double floor(double)
float floorf(float)
double fmod(double)
float fmodf(float)
double hypot(double, double)
float hypotf(float)
double rint(double)
float rintf(float)
double sinh(double)
float sinhf(float)
float sqrtf(float)
double tanh(double)
float tanhf(float)
char *_strset(char *, _int32)

void *memcmp(const void *cs, const void *ct, size_t n)
void *memcpy(void *s, const void *ct, size_t n)
void *memset(void * s, int c, size_t n)
char *Strcat(char * s, const char * ct)
int *strcmp(const char *, const char *)
char *strcpy(char * s, const char * ct)
size_t strlen(const char * cs)
int strncmp(char *, char *, int)
int strncpy(char *, char *, int)
void *__alloca(int)
int _setjmp(jmp_buf)
_exception_code(void)
_exception_info(void)
_abnormal_termination(void)
void _enable()
void _disable()
int _bswap(int)
int _in_byte(int)
int _in_dword(int)
int _in_word(int)
int _inp(int)
int _inpd(int)
int _inpw(int)
int _out_byte(int, int)
int _out_dword(int, int)
int _out_word(int, int)
int _outp(int, int)
int _outpd(int, int)
int _outpw(int, int)
unsigned short _rotwl(unsigned short val, int count)
unsigned short _rotwr(unsigned short val, int count)

MMX(TM) Technology Intrinsics Implementation

Key to the table entries

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

Intrinsic Name	MMX(TM) Technology SSE SSE2	IA-64 Architecture
_mm_empty	A	B
_mm_cvtsi32_si64	A	A
_mm_cvtsi64_si32	A	A
_mm_packs_pi16	A	A
_mm_packs_pi32	A	A
_mm_packs_pu16	A	A
_mm_unpackhi_pi8	A	A
_mm_unpackhi_pi16	A	A
_mm_unpackhi_pi32	A	A
_mm_unpacklo_pi8	A	A
_mm_unpacklo_pi16	A	A
_mm_unpacklo_pi32	A	A
_mm_add_pi8	A	A
_mm_add_pi16	A	A
_mm_add_pi32	A	A
_mm_adds_pi8	A	A
_mm_adds_pi16	A	A
_mm_adds_pu8	A	A
_mm_adds_pu16	A	A
_mm_sub_pi8	A	A

_mm_sub_pi16	A	A
_mm_sub_pi32	A	A
_mm_subs_pi8	A	A
_mm_subs_pi16	A	A
_mm_subs_pu8	A	A
_mm_subs_pu16	A	A
_mm_madd_pi16	A	C
_mm_mulhi_pi16	A	A
_mm_mullo_pi16	A	A
_mm_sll_pi16	A	A
_mm_slli_pi16	A	A
_mm_sll_pi32	A	A
_mm_slli_pi32	A	A
_mm_sll_pi64	A	A
_mm_slli_pi64	A	A
_mm_sra_pi16	A	A
_mm_srai_pi16	A	A
_mm_sra_pi32	A	A
_mm_srai_pi32	A	A
_mm_srl_pi16	A	A
_mm_srli_pi16	A	A
_mm_srl_pi32	A	A
_mm_srli_pi32	A	A
_mm_srl_si64	A	A
_mm_srli_si64	A	A
_mm_and_si64	A	A
_mm_andnot_si64	A	A
_mm_or_si64	A	A
_mm_xor_si64	A	A
_mm_cmpeq_pi8	A	A
_mm_cmpeq_pi16	A	A
_mm_cmpeq_pi32	A	A

_mm_cmpgt_pi8	A	A
_mm_cmpgt_pi16	A	A
_mm_cmpgt_pi32	A	A
_mm_setzero_si64	A	A
_mm_set_pi32	A	A
_mm_set_pi16	A	C
_mm_set_pi8	A	C
_mm_set1_pi32	A	A
_mm_set1_pi16	A	A
_mm_set1_pi8	A	A
_mm_setr_pi32	A	A
_mm_setr_pi16	A	C
_mm_setr_pi8	A	C

_mm_empty is implemented in IA-64 instructions as a NOP for source compatibility only.

Streaming SIMD Extensions Intrinsics Implementation

Regular Streaming SIMD Extensions (SSE) intrinsics work on 4 32-bit single precision values. On IA-64 architecture-based systems, basic operations like add and compare require two SIMD instructions. All can be executed in the same cycle so the throughput is one basic SSE operation per cycle or 4 32-bit single precision operations per cycle.

Key to the table entries

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

Intrinsic Name	MMX(TM Technology	SSE SSE2	IA-64 Architecture
_mm_add_ss	N/A	B	B
_mm_add_ps	N/A	A	A
_mm_sub_ss	N/A	B	B

_mm_sub_ps	N/A	A	A
_mm_mul_ss	N/A	B	B
_mm_mul_ps	N/A	A	A
_mm_div_ss	N/A	B	B
_mm_div_ps	N/A	A	A
_mm_sqrt_ss	N/A	B	B
_mm_sqrt_ps	N/A	A	A
_mm_rcp_ss	N/A	B	B
_mm_rcp_ps	N/A	A	A
_mm_rsqrt_ss	N/A	B	B
_mm_rsqrt_ps	N/A	A	A
_mm_min_ss	N/A	B	B
_mm_min_ps	N/A	A	A
_mm_max_ss	N/A	B	B
_mm_max_ps	N/A	A	A
_mm_and_ps	N/A	A	A
_mm_andnot_ps	N/A	A	A
_mm_or_ps	N/A	A	A
_mm_xor_ps	N/A	A	A
_mm_cmpeq_ss	N/A	B	B
_mm_cmpeq_ps	N/A	A	A
_mm_cmplt_ss	N/A	B	B
_mm_cmplt_ps	N/A	A	A
_mm_cmple_ss	N/A	B	B
_mm_cmple_ps	N/A	A	A
_mm_cmpgt_ss	N/A	B	B
_mm_cmpgt_ps	N/A	A	A
_mm_cmpge_ss	N/A	B	B
_mm_cmpge_ps	N/A	A	A
_mm_cmpneq_ss	N/A	B	B
_mm_cmpneq_ps	N/A	A	A
_mm_cmpnlt_ss	N/A	B	B

_mm_cmpnlt_ps	N/A	A	A
_mm_cmpnle_ss	N/A	B	B
_mm_cmpnle_ps	N/A	A	A
_mm_cmpngt_ss	N/A	B	B
_mm_cmpngt_ps	N/A	A	A
_mm_cmpnge_ss	N/A	B	B
_mm_cmpnge_ps	N/A	A	A
_mm_cmpord_ss	N/A	B	B
_mm_cmpord_ps	N/A	A	A
_mm_cmpunord_ss	N/A	B	B
_mm_cmpunord_ps	N/A	A	A
_mm_comieq_ss	N/A	B	B
_mm_comilt_ss	N/A	B	B
_mm_comile_ss	N/A	B	B
_mm_comigt_ss	N/A	B	B
_mm_comige_ss	N/A	B	B
_mm_comineq_ss	N/A	B	B
_mm_ucomieq_ss	N/A	B	B
_mm_ucomilt_ss	N/A	B	B
_mm_ucomile_ss	N/A	B	B
_mm_ucomigt_ss	N/A	B	B
_mm_ucomige_ss	N/A	B	B
_mm_ucomineq_ss	N/A	B	B
_mm_cvtss_si32	N/A	A	B
_mm_cvtps_pi32	N/A	A	A
_mm_cvttss_si32	N/A	A	B
_mm_cvttps_pi32	N/A	A	A
_mm_cvtsi32_ss	N/A	A	B
_mm_cvtpi32_ps	N/A	A	C
_mm_cvtpi16_ps	N/A	A	C
_mm_cvtpu16_ps	N/A	A	C
_mm_cvtpi8_ps	N/A	A	C

_mm_cvtpu8_ps	N/A	A	C
_mm_cvtpi32x2_ps	N/A	A	C
_mm_cvtps_pi16	N/A	A	C
_mm_cvtps_pi8	N/A	A	C
_mm_move_ss	N/A	A	A
_mm_shuffle_ps	N/A	A	A
_mm_unpackhi_ps	N/A	A	A
_mm_unpacklo_ps	N/A	A	A
_mm_movehl_ps	N/A	A	A
_mm_movelh_ps	N/A	A	A
_mm_movemask_ps	N/A	A	C
_mm_getcsr	N/A	A	A
_mm_setcsr	N/A	A	A
_mm_loadh_pi	N/A	A	A
_mm_loadl_pi	N/A	A	A
_mm_load_ss	N/A	A	B
_mm_load1_ps	N/A	A	A
_mm_load_ps	N/A	A	A
_mm_loadu_ps	N/A	A	A
_mm_loadr_ps	N/A	A	A
_mm_storeh_pi	N/A	A	A
_mm_storel_pi	N/A	A	A
_mm_store_ss	N/A	A	A
_mm_store_ps	N/A	A	A
_mm_store1_ps	N/A	A	A
_mm_storeu_ps	N/A	A	A
_mm_storer_ps	N/A	A	A
_mm_set_ss	N/A	A	A
_mm_set1_ps	N/A	A	A
_mm_set_ps	N/A	A	A
_mm_setr_ps	N/A	A	A
_mm_setzero_ps	N/A	A	A

_mm_prefetch	N/A	A	A
_mm_stream_pi	N/A	A	A
_mm_stream_ps	N/A	A	A
_mm_sfence	N/A	A	A
_mm_extract_pi16	N/A	A	A
_mm_insert_pi16	N/A	A	A
_mm_max_pi16	N/A	A	A
_mm_max_pu8	N/A	A	A
_mm_min_pi16	N/A	A	A
_mm_min_pu8	N/A	A	A
_mm_movemask_pi8	N/A	A	C
_mm_mulhi_pu16	N/A	A	A
_mm_shuffle_pi16	N/A	A	A
_mm_maskmove_si64	N/A	A	C
_mm_avg_pu8	N/A	A	A
_mm_avg_pu16	N/A	A	A
_mm_sad_pu8	N/A	A	A

Streaming SIMD Extensions 2 Intrinsics Implementation

On processors that do not support SSE2 instructions but do support MMX Technology, you can use the `sse2mmx.h` emulation pack to enable support for SSE2 instructions. You can use the `sse2mmx.h` header file for the following processors:

- Intel® Itanium® processor
- Intel® Pentium® III processor
- Intel® Pentium® II processor
- Intel® Pentium® processors with MMX™ Technology

Index