Intel® C++ Compiler for Linux*
Intrinsics Reference

Disclaimer and Legal Information

The information in this manual is subject to change without notice and Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. This document and the software described in it are furnished under license and may only be used or copied in accordance with the terms of the license. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. The information in this document is provided in connection with Intel products and should not be construed as a commitment by Intel Corporation.

EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

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

The software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

Intel, the Intel logo, Intel SpeedStep, Intel NetBurst, Intel NetStructure, MMX, Intel386, Intel486, Celeron, Intel Centrino, Intel Xeon, Intel XScale, Itanium, Pentium, Pentium II Xeon, Pentium III Xeon, Pentium M, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright © 1998-2006, Intel Corporation.

Table Of Contents

Intel(R) C++ Intrinsics Reference

Introduction to Intel® C++ Compiler Intrinsics

Several Intel® processors enable development of optimized multimedia applications through extensions to previously implemented instructions. Applications with media-rich bit streams can significantly improve performance by using single instruction, multiple data (SIMD) instructions to process data elements in parallel.

The most direct way to use these instructions is to inline the assembly language instructions into your source code. However, this process can be time-consuming and tedious. In addition, your compiler may not support inline assembly language programming. The Intel® C++ Compiler enables easy implementation of these instructions through the use of API extension sets built into the compiler. These extension sets are referred to as intrinsic functions or intrinsics.

The Intel C++ Compiler supports both intrinsics that work on specific architectures and intrinsics that work across all IA-32 and Itanium®-based platforms.

Intrinsics provide you with several benefits:

- The compiler optimizes intrinsic instruction scheduling so that executables run faster.
- Intrinsics enable you to use the syntax of C function calls and C variables instead of assembly language or hardware registers.
- Intrinsics provide access to instructions that cannot be generated using the standard constructs of the C and C++ languages.

Availability of Intrinsics on Intel® Processors

The following table shows which intrinsics are supported on each processor listed in the left column.

On processors that do not support Streaming SIMD Extensions 2 (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
- Pentium® III Processor
- Pentium® II Processor
- Pentium® with MMX™ Technology

Processors:	MMX(TM) Technology Instructions	Streaming SIMD Extensions	Streaming SIMD Extensions 2	Streaming SIMD Extensions 3	Itanium Processor Instructions
Itanium®	Supported	Supported	Not		Supported

Processor			Supported		
Pentium® 4 Processor	Supported	Supported	Supported	Supported	Not Supported
Pentium® III Processor	Supported	Supported	Not Supported		Not Supported
Pentium® II Processor	Supported	Not Supported	Not Supported		Not Supported
Pentium® with MMX Technology	Supported	Not Supported	Not Supported		Not Supported
Pentium® Pro Processor	Not Supported	Not Supported	Not Supported		Not Supported
Pentium® Processor	Not Supported	Not Supported	Not Supported		Not Supported

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.

2

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 Itanium-based platforms, 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.

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.

Code Samples

Dot Product

This code sample demonstrates how to use C, MMX™ technology, and Streaming SIMD Extensions 3 (SSE3) intrinsics to calculate the dot product of two vectors. The following outputs are typical of this code when computed by C or SSE3 intrinsics: 506.000000 and when computed by MMX intrinsics: 506. Output may vary depending on your compiler version and the components of your computing platform.

 SSE3 intrinsics do not run on processors from the Pentium® III family or earlier.

```
/*

 * Copyright (C) 2006 Intel Corporation.  All rights reserved.

 *

 * The information and source code contained herein is the exclusive
property

 * of Intel Corporation and may not be disclosed, examined, or
reproduced in

 * whole or in part without explicit written authorization from the
Company.

 *

 * [Description]

 * This code sample demonstrates how to use C, MMX, and SSE3

 * instrinsics to calculate the dot product of two vectors.

 *

 * [Compile]

 * icc dot_prodcut.c (linux) | icl dot_product.c (windows)

 *

 * [Output]

 * Dot Product computed by C: 506.000000

 * Dot Product computed by SSE2 intrinsics:  506.000000

 * Dot Product computed by MMX intrinsics:  506
```

```
 */


#include <stdio.h>

#include <pmmintrin.h>

#define SIZE 12  //assumes size is a multiple of 4 because MMX and SSE

                 //registers will store 4 elements.

//Computes dot product using C

float dot_product(float *a, float *b);

//Computes dot product using intrinsics

float dot_product_intrin(float *a, float *b);

//Computes dot product using MMX intrinsics

short MMX_dot_product(short *a, short *b);

int main()

{

  float x[SIZE], y[SIZE];

  short a[SIZE], b[SIZE];

  int i;

  float product;

  short mmx_product;


  for(i=0; i<SIZE; i++)

  {

    x[i]=i;

    y[i]=i;

    a[i]=i;

    b[i]=i;

  }
```

```
  product= dot_product(x, y);

  printf("Dot Product computed by C: %f\n", product);


  #if __INTEL_COMPILER

  product =dot_product_intrin(x,y);

  printf("Dot Product computed by SSE2 intrinsics:  %f\n", product);

  mmx_product =MMX_dot_product(a,b);

  printf("Dot Product computed by MMX intrinsics:  %d\n", mmx_product);


  #else

  printf("Use INTEL compiler in order to calculate dot product\n");

  printf("usng intrinsics\n");


  #endif

  return 0;

}

float dot_product(float *a, float *b)

{

  int i;

  int sum=0;

  for(i=0; i<SIZE; i++)

  {

    sum += a[i]*b[i];

  }

  return sum;

}

#if __INTEL_COMPILER
```

```
float dot_product_intrin(float *a, float *b)

{

  float arr[4];

  float total;

  int i;

  __m128 num1, num2, num3, num4;

  num4= _mm_setzero_ps();  //sets sum to zero

  for(i=0; i<SIZE; i+=4)

  {

    num1 = _mm_loadu_ps(a+i);   //loads unaligned array a into num1
 num1= a[3]  a[2]  a[1]  a[0]

    num2 = _mm_loadu_ps(b+i);   //loads unaligned array b into num2
 num2= b[3]   b[2]   b[1]  b[0]

    num3 = _mm_mul_ps(num1, num2); //performs multiplication    num3 =
a[3]*b[3]  a[2]*b[2]  a[1]*b[1]  a[0]*b[0]

    num3 = _mm_hadd_ps(num3, num3); //performs horizontal addition

                              //num3=  a[3]*b[3]+ a[2]*b[2]
 a[1]*b[1]+a[0]*b[0]  a[3]*b[3]+ a[2]*b[2]  a[1]*b[1]+a[0]*b[0]

    num4 = _mm_add_ps(num4, num3);  //performs vertical addition

  }

  num4= _mm_hadd_ps(num4, num4);

  _mm_store_ss(&total,num4);

  return total;

}

//MMX technology cannot handle single precision floats

short MMX_dot_product(short *a, short *b)

{

  int i;

  short result, data;
```

```
    __m64 num3, sum;

    __m64 *ptr1, *ptr2;



  sum = _mm_setzero_si64(); //sets sum to zero

  for(i=0; i<SIZE; i+=4){

   ptr1 = (__m64*)&a[i];  //Converts array a to a pointer of type

                          //__m64 and stores four elements into MMX

                          //registers

   ptr2 = (__m64*)&b[i];

   num3 = _m_pmaddwd(*ptr1, *ptr2); //multiplies elements and adds lower

                                    //elements with lower element and

                                    //higher elements with higher

   sum = _m_paddw(sum, num3);

   }

   data = _m_to_int(sum);      //converts __m64 data type to an int

   sum= _m_psrlqi(sum,32);     //shifts sum

   result = _m_to_int(sum);

   result= result+data;

   _m_empty();  //clears the MMX registers and MMX state.

   return result;

}

#endif
```

Double Complex

This code sample demonstrates how to use C, Streaming SIMD Extensions 2 (SSE2) and Streaming SIMD Extensions 3 (SSE3) intrinsics to multiply two complex numbers. The following output is typical of this code: 23.00+ -2.00i. Output may vary depending on your compiler version and the components of your computing platform.

 SSE3 intrinsics do not run on processors from the Pentium® III family or earlier.

```
/*

 * Copyright (C) 2006 Intel Corporation.  All rights reserved.

 *

 * The information and source code contained herein is the exclusive

 * property  of Intel Corporation and may not be disclosed, examined,

 * or reproduced in  whole or in part without explicit written

 * authorization from the Company.

 *

 * [Description]

 * This code sample demonstrates the use of C in comparison with SSE2

 * and SSE3 instrinsics to multiply two complex numbers.

 *

 * [Compile]

 * icc double_complex.c (linux)

 * icl double_complex.c (windows)

 *

 * [Output]

 * Complex Product(C):    23.00+ -2.00i

 * Complex Product(SSE3): 23.00+ -2.00i

 * Complex Product(SSE2): 23.00+ -2.00i

 */
#include <stdio.h>

#include <pmmintrin.h>

typedef struct {

  double real;

  double img;
```

```
} complex_num;

// Multiplying complex numbers in C

void multiply_C(complex_num x, complex_num y, complex_num *z)

{

  z->real = (x.real*y.real) - (x.img*y.img);

  z->img  = (x.img*y.real)  + (y.img*x.real);

}

#if __INTEL_COMPILER

// Multiplying complex numbers using SSE3 intrinsics

void multiply_SSE3(complex_num x, complex_num y, complex_num *z)

{

  __m128d num1, num2, num3;

  // Duplicates lower vector element into upper vector element.

  //   num1: [x.real, x.real]

  num1 = _mm_loaddup_pd(&x.real);

  // Move y elements into a vector

  //   num2: [y.img, y.real]

  num2 = _mm_set_pd(y.img, y.real);

  // Multiplies vector elements

  //   num3: [(x.real*y.img), (x.real*y.real)]

  num3 = _mm_mul_pd(num2, num1);

  //   num1: [x.img, x.img]

  num1 = _mm_loaddup_pd(&x.img);

  // Swaps the vector elements

  //   num2: [y.real, y.img]

  num2 = _mm_shuffle_pd(num2, num2, 1);

  //   num2: [(x.img*y.real), (x.img*y.img)]
```

```
num2 = _mm_mul_pd(num2, num1);

// Adds upper vector element while subtracting lower vector element

//   num3: [((x.real *y.img)+(x.img*y.real)),

//          ((x.real*y.real)-(x.img*y.img))]

num3 = _mm_addsub_pd(num3, num2);

// Stores the elements of num3 into z

_mm_storeu_pd((double *)z, num3);

}

#endif

#if __INTEL_COMPILER

// Multiplying complex numbers using SSE2 intrinsics

void multiply_SSE2(complex_num x, complex_num y, complex_num *z)

{

  __m128d num1, num2, num3, num4;

  // Copies a single element into the vector

  //   num1:  [x.real, x.real]

  num1 = _mm_load1_pd(&x.real);

  // Move y elements into a vector

  //   num2: [y.img, y.real]

  num2 = _mm_set_pd(y.img, y.real);

  // Multiplies vector elements

  //   num3: [(x.real*y.img), (x.real*y.real)]

  num3 = _mm_mul_pd(num2, num1);

  //   num1: [x.img, x.img]

  num1 = _mm_load1_pd(&x.img);

  // Swaps the vector elements.

  //   num2: [y.real, y.img]
```

```
  num2 = _mm_shuffle_pd(num2, num2, 1);

  //   num2: [(x.img*y.real), (x.img*y.img)]

  num2 = _mm_mul_pd(num2, num1);

  num4 = _mm_add_pd(num3, num2);

  num3 = _mm_sub_pd(num3, num2);

  num4 = _mm_shuffle_pd(num3, num4, 2);

  // Stores the elements of num4 into z

  _mm_storeu_pd((double *)z, num4);

}

#endif

int main()

{

  complex_num a, b, c;

  // Initialize complex numbers

  a.real =  3;

  a.img  =  2;

  b.real =  5;

  b.img  = -4;

  // Output for each: 23.00+ -2.00i

  multiply_C(a, b, &c);

  printf("Complex Product(C):    %2.2f+ %2.2fi\n", c.real, c.img);

#if __INTEL_COMPILER

  multiply_SSE3(a, b, &c);

  printf("Complex Product(SSE3): %2.2f+ %2.2fi\n", c.real, c.img);

  multiply_SSE2(a, b, &c);

  printf("Complex Product(SSE2): %2.2f+ %2.2fi\n", c.real, c.img);

#endif
```

14

```
  return 0;

}
```

Time Stamp

This code sample demonstrates how to use the `_rdtsc()` intrinsic to read the time stamp counter. The output is the current value of the 64-bit time stamp counter, and therefore varies each time you compile the code.

```
/*

 * Copyright (C) 2006 Intel Corporation.  All rights reserved.

 *

 * The information and source code contained herein is the exclusive

 * property of Intel Corporation and may not be disclosed, examined,

 * or reproduced in whole or in part without explicit written

 * authorization from the Company.

 *

 * [Description]

 * This code sample demonstrates how to use intrinsics to

 * read the time stamp counter. The _rdtsc() function

 * returns the current value of the processor's 64-bit time stamp
counter.

 *

 * [Compile]

 * icc time_stamp.c (linux) | icl time_stamp.c (windows)

 *

 * [Output]

 * <varies>

 */
```

```c
#include <stdio.h>

int main()

{

 #if __INTEL_COMPILER

  __int64 start, stop, elaspe;

  int i;

  int arr[10000];

  start= _rdtsc();

  for(i=0; i<10000; i++)

  {

    arr[i]=i;

  }

  stop= _rdtsc();

  elaspe = stop -start;

  printf("Processor cycles\n %I64u\n", elaspe);

  #else

  printf("Use INTEL Compiler in order to implement\n");

  printf("_rdtsc intrinsic\n");

  #endif

  return 0;

}
```

Intrinsics for Use Across All IA

Overview: Intrinsics For All IA

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

- Integer Arithmetic 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)`	Rotates bits left for an unsigned long integer.
`unsigned long  lrotr(unsigned long value, int shift)`	Rotates bits right for an unsigned long integer.
`unsigned int _rotl(unsigned int value, int shift)`	Rotates bits left for an unsigned integer.
`unsigned int _rotr(unsigned int value, int shift)`	Rotates bits right for an unsigned integer.

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`	Computes absolute value of complex number. The

`cabs(struct_complex)`	`intrinsic argument` is a complex number made up of two double precision elements, one real and one imaginary part.
`double ceil(double)`	Computes smallest integral value of double precision argument not less than the argument.
`float ceilf(float)`	Computes smallest integral value of single precision argument not less than the argument.
`double cosh(double)`	Computes the hyperbolic cosine of double precison argument.
`float coshf(float)`	Computes the hyperbolic cosine of single precison argument.
`float fabsf(float)`	Computes absolute value of single precision argument.
`double floor(double)`	Computes the largest integral value of the double precision argument not greater than the argument.
`float floorf(float)`	Computes the largest integral value of the single precision argument not greater than the argument.
`double fmod(double)`	Computes the floating-point remainder of the division of the first argument by the second argument with double precison.
`float fmodf(float)`	Computes the floating-point remainder of the division of the first argument by the second argument with single precison.
`double hypot(double, double)`	Computes the length of the hypotenuse of a right angled triangle with double precision.
`float hypotf(float, 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 Itanium®-based platforms.

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 Itanium®-based systems. 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 Itanium®-based systems.

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

MMX Instruction Registers Need__ __m64 Data types

CM06881
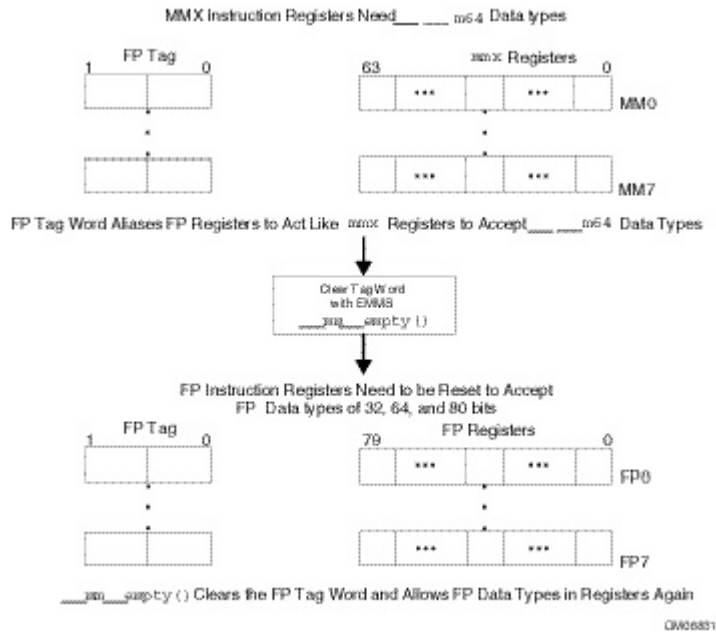
⚠️ Caution

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 Itanium®-based systems. There are no special registers (or overlay) for the MMX(TM) instructions or Streaming SIMD Extensions on Itanium-based systems even though the intrinsics are supported.
- 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.

To see detailed information about an intrinsic, click on that intrinsic in the following table.

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.

For detailed information about an intrinsic, click on the name of the intrinsic in the following table.

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.

For detailed information about an intrinsic, click on the name of the intrinsic in the following table.

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.

For detailed information about an intrinsic, click on that intrinsic in the following table.

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. For a more detailed description of an intrinsic, click on that intrinsic in the table.

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.

For detailed information about an intrinsic, click on that intrinsic in the following table.

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 Itanium® Architecture

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

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.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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

To see the details about an intrinsic, click on that intrinsic name in the following table.

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.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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 effecient 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.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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 Itanium®-based systems.

```
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.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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)`

80

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 & 0x01), where t = (unsigned char)a0 + (unsigned char)b0	(t >> 1) | (t & 0x01), where t = (unsigned char)a1 + (unsigned char)b1	...	((t >> 1) | (t & 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

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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 Itanium® Architecture

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

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

Compatibility versus Performance

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

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

- Any 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 Itanium instructions are required to implement these.
- Conversions in general are multi-instruction operations. These are particularly expensive: `_mm_cvtpi16_ps`, `_mm_cvtpu16_ps`, `_mm_cvtpi8_ps`, `_mm_cvtpu8_ps`, `_mm_cvtpi32x2_ps`, `_mm_cvtps_pi16`, `_mm_cvtps_pi8`
- 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

```
              127           0
  ; m1 =        a  b  c  d
              127           0
  ; m2 =        e  f  g  h
  m3 = _mm_shuffle_ps(m1, m2,
      _MM_SHUFFLE(1,0,3,2))
              127           0
  ; m3 =        g  h  a  b
```

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 Itanium Instructions topic.

Exception State Macros	Macro Arguments
_MM_SET_EXCEPTION_STATE(x)	_MM_EXCEPT_INVALID
_MM_GET_EXCEPTION_STATE()	_MM_EXCEPT_DIV_ZERO
	_MM_EXCEPT_DENORM
Macro Definitions Write to and read from the 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
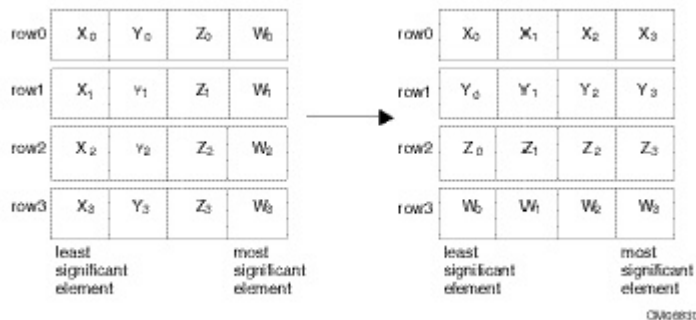
Streaming SIMD Extensions 2

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.

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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.

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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.

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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	`CMPLEPDr`
`_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 :	(a1 >= b1) ? 0xffffffffffffffff :

0x0	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.

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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.

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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.

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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)

120

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.

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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 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.

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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).

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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.

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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.

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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).

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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

146

(__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).

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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).

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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).

For detailed information about an intrinsic, click on that intrinsic name in the following table.

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	
`_mm_stream_si32`	Store	
`_mm_store_si128`	Store	`MOVDQA`
`_mm_storeu_si128`	Store	`MOVDQU`
`_mm_maskmoveu_si128`	Conditional store	`MASKMOVDQU`
`_mm_storel_epi64`	Store lowest	`MOVQ`

```
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

159

```
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	`MOVQ2DQ`

`_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`.

164

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(__m64 a)`

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

R0
a0

`__128i _mm_movpi64_pi64(__m128i 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(__128i 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)`

166

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


```
_MM_SHUFFLE2(x, y)

expands to the value of

(x<<1) | y
```

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

View of Original and Result Words with Shuffle Function Macro
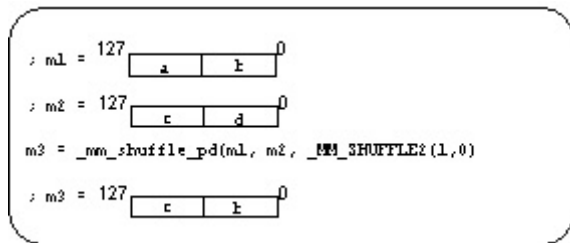
Streaming SIMD Extensions 3

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.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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;	a2;	a2;
a0;			

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.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

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.

Intrinsics for Itanium(R) Instructions

Overview: Intrinsics for Itanium® Instructions

This section lists and describes the native intrinsics for Itanium® instructions. These intrinsics cannot be used on the IA-32 architecture. The intrinsics for Itanium instructions give programmers access to Itanium instructions that cannot be generated using the standard constructs of the C and C++ languages.

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Itanium processor do not support SSE2 intrinsics. However, you can use the sse2mmx.h emulation pack to enable support for SSE2 instructions on Itanium architecture.

For information on how to use SSE intrinsics on Itanium architecture, see Using Streaming SIMD Extensions on Itanium(R) Architecture.

For information on how to use MMX (TM) technology intrinsics on Itanium architecture, see MMX(TM) Technology Intrinsics on Itanium Architecture

Native Intrinsics for Itanium® Instructions

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Integer Operations

176

Intrinsic	Operation	Corresponding Itanium 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(int amask, int omask)	Sets the control bits of FPSR.sf0. Maps to the fsetc.sf0 r, r instruction. There is no corresponding instruction to read the control bits. Use _mm_getfpsr().
void _fclrf(void)	Clears the floating point status flags (the 6-bit flags of FPSR.sf0). Maps to the fclrf.sf0 instruction.

```
__int64 _m64_dep_mr(__int64 r, __int64 s, const int pos, const int len)
```

The right-justified 64-bit value r is deposited into the value in s at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position pos and extends to the left (toward the most significant bit) the number of bits specified by len.

```
__int64 _m64_dep_mi(const int v, __int64 s, const int p, const int len)
```

The sign-extended value v (either all 1s or all 0s) is deposited into the value in s at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position p and extends to the left (toward the most significant bit) the number of bits specified by len.

```
__int64 _m64_dep_zr(__int64 s, const int pos, const int len)
```

The right-justified 64-bit value `s` is deposited into a 64-bit field of all zeros at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `pos` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

```
__int64 _m64_dep_zi(const int v, const int pos, const int len)
```

The sign-extended value `v` (either all 1s or all 0s) is deposited into a 64-bit field of all zeros at an arbitrary bit position and the result is returned. The deposited bit field begins at bit position `pos` and extends to the left (toward the most significant bit) the number of bits specified by `len`.

```
__int64 _m64_extr(__int64 r, const int pos, const int len)
```

A field is extracted from the 64-bit value `r` and is returned right-justified and sign extended. The extracted field begins at position `pos` and extends `len` bits to the left. The sign is taken from the most significant bit of the extracted field.

```
__int64 _m64_extru(__int64 r, const int pos, const int len)
```

A field is extracted from the 64-bit value `r` and is returned right-justified and zero extended. The extracted field begins at position `pos` and extends `len` bits to the left.

```
__int64 _m64_xmal(__int64 a, __int64 b, __int64 c)
```

The 64-bit values `a` and `b` are treated as signed integers and multiplied to produce a full 128-bit signed result. The 64-bit value `c` is zero-extended and added to the product. The least significant 64 bits of the sum are then returned.

```
__int64 _m64_xmalu(__int64 a, __int64 b, __int64 c)
```

The 64-bit values `a` and `b` are treated as signed integers and multiplied to produce a full 128-bit unsigned result. The 64-bit value `c` is zero-extended and added to the product. The least significant 64 bits of the sum are then returned.

```
__int64 _m64_xmah(__int64 a, __int64 b, __int64 c)
```

The 64-bit values `a` and `b` are treated as signed integers and multiplied to produce a full 128-bit signed result. The 64-bit value `c` is zero-extended and added to the product. The most significant 64 bits of the sum are then returned.

```
__int64 _m64_xmahu(__int64 a, __int64 b, __int64 c)
```

The 64-bit values `a` and `b` are treated as unsigned integers and multiplied to produce a full 128-bit unsigned result. The 64-bit value `c` is zero-extended and added to the product. The most significant 64 bits of the sum are then returned.

```
__int64 _m64_popcnt(__int64 a)
```

The number of bits in the 64-bit integer `a` that have the value 1 are counted, and the resulting sum is returned.

```
__int64 _m64_shladd(__int64 a, const int count, __int64 b)
```

`a` is shifted to the left by `count` bits and then added to `b`. The result is returned.

```
__int64 _m64_shrp(__int64 a, __int64 b, const int count)
```

`a` and `b` are concatenated to form a 128-bit value and shifted to the right `count` bits. The least significant 64 bits of the result are returned.

Lock and Atomic Operation Related Intrinsics

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

Intrinsic	Description

`unsigned __int64`
`_InterlockedExchange8(volatile unsigned char`
`*Target, unsigned __int64 value)`
	Map to the `xchg1` instruction. Atomically write the least significant byte of its 2nd argument to address specified by its 1st argument.

`unsigned __int64`
`_InterlockedCompareExchange8_rel(volatile`
`unsigned char *Destination, unsigned __int64`
`Exchange, unsigned __int64 Comparand)`
	Compare and exchange atomically the least significant byte at the address specified by its 1st argument. Maps to the `cmpxchg1.rel` instruction with appropriate setup.

`unsigned __int64`
`_InterlockedCompareExchange8_acq(volatile`
`unsigned char *Destination, unsigned __int64`
`Exchange, unsigned __int64 Comparand)`
	Same as 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 increment)`	Same as the previous 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 __int64 Exchange, unsigned __int64 Comparand)`	Maps to the `cmpxchg.acq` instruction with appropriate setup. Atomically compare and exchange the value specified by the first argument (a 64-bit pointer).
`__int64 _InterlockedCompareExchange64(volatile __int64 *Destination, __int64 Exchange, __int64 Comparand)`	Same as 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.

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.
`void __thash(__int64)`	Generates a translation hash entry address. Maps to the `thash r = r` instruction.
`void __ttag(__int64)`	Generates a translation hash entry tag. Maps to the `ttag r=r` instruction.
`void __itcd(__int64 pa)`	Insert an entry into the data translation cache (Map `itc.d` instruction).
`void __itci(__int64 pa)`	Insert an entry into the instruction translation cache (Map `itc.i`).
`void __itrd(__int64 whichTransReg, __int64 pa)`	Map the `itr.d` instruction.
`void __itri(__int64 whichTransReg, __int64 pa)`	Map the `itr.i` instruction.
`void __ptce(__int64 va)`	Map the `ptc.e` instruction.
`void __ptcl(__int64 va, __int64 pagesz)`	Purges the local translation cache. Maps to the `ptc.l r, r` instruction.
`void __ptcg(__int64 va, __int64 pagesz)`	Purges the global translation cache. Maps to the `ptc.g r, r` instruction.
`void __ptcga(__int64 va, __int64 pagesz)`	Purges the global translation cache and ALAT. Maps to the `ptc.ga r, r` instruction.
`void __ptri(__int64 va, __int64 pagesz)`	Purges the translation register. Maps to the `ptr.i r, r` instruction.

`void __ptrd(__int64 va, __int64 pagesz)`	Purges the translation register. Maps to the `ptr.d r, r` instruction.
`__int64 __tpa(__int64 va)`	Map the `tpa` instruction.
`void __invalat(void)`	Invalidates `ALAT`. Maps to the `invala` instruction.
`void __invala (void)`	Same as `void __invalat(void)`
`void __invala_gr(const int whichGeneralReg)`	`whichGeneralReg` = 0-127
`void __invala_fr(const int whichFloatReg)`	`whichFloatReg` = 0-127
`void __break(const int)`	Generates a break instruction with an immediate.
`void __nop(const int)`	Generate a `nop` instruction.
`void __debugbreak(void)`	Generates a Debug Break Instruction fault.
`void __fc(__int64)`	Flushes a cache line associated with the address given by the argument. Maps to the `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, void *y)`	Generate the `lfetch.lfhint` instruction. The value of the first argument specifies the hint type.
`void __lfetch_fault(int lfhint, void *y)`	Generate the `lfetch.fault.lfhint` instruction. The value of the first argument specifies the hint type.
`void __lfetch_excl(int lfhint, void *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 *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 on Itanium-based systems.
`__m64 _m_from_int64(__int64 a)`	Convert `a` of type `__int64` to type `__m64`. Translates to `nop` since both types reside in the same register on Itanium-based systems.
`__int64 _round_double_to_int64(double d)`	Convert its double precision argument to a signed integer.
`unsigned __int64 _getf_exp(double d)`	Map the `getf.exp` instruction and return the 16-bit exponent and the sign of its operand.

Register Names for getReg() and setReg()

The prototypes for getReg() and setReg() intrinsics are in the `ia64regs.h` header file.

Name	whichReg
`_IA64_REG_IP`	1016
`_IA64_REG_PSR`	1019
`_IA64_REG_PSR_L`	1019

General Integer Registers

Name	whichReg
`_IA64_REG_GP`	1025
`_IA64_REG_SP`	1036
`_IA64_REG_TP`	1037

Application Registers

Name	whichReg
`_IA64_REG_AR_KR0`	3072
`_IA64_REG_AR_KR1`	3073
`_IA64_REG_AR_KR2`	3074
`_IA64_REG_AR_KR3`	3075
`_IA64_REG_AR_KR4`	3076
`_IA64_REG_AR_KR5`	3077
`_IA64_REG_AR_KR6`	3078
`_IA64_REG_AR_KR7`	3079
`_IA64_REG_AR_RSC`	3088
`_IA64_REG_AR_BSP`	3089
`_IA64_REG_AR_BSPSTORE`	3090
`_IA64_REG_AR_RNAT`	3091
`_IA64_REG_AR_FCR`	3093
`_IA64_REG_AR_EFLAG`	3096
`_IA64_REG_AR_CSD`	3097
`_IA64_REG_AR_SSD`	3098
`_IA64_REG_AR_CFLAG`	3099
`_IA64_REG_AR_FSR`	3100
`_IA64_REG_AR_FIR`	3101
`_IA64_REG_AR_FDR`	3102
`_IA64_REG_AR_CCV`	3104
`_IA64_REG_AR_UNAT`	3108
`_IA64_REG_AR_FPSR`	3112
`_IA64_REG_AR_ITC`	3116
`_IA64_REG_AR_PFS`	3136
`_IA64_REG_AR_LC`	3137
`_IA64_REG_AR_EC`	3138

Control Registers

Name	whichReg
`_IA64_REG_CR_DCR`	4096

`_IA64_REG_CR_ITM`	4097
`_IA64_REG_CR_IVA`	4098
`_IA64_REG_CR_PTA`	4104
`_IA64_REG_CR_IPSR`	4112
`_IA64_REG_CR_ISR`	4113
`_IA64_REG_CR_IIP`	4115
`_IA64_REG_CR_IFA`	4116
`_IA64_REG_CR_ITIR`	4117
`_IA64_REG_CR_IIPA`	4118
`_IA64_REG_CR_IFS`	4119
`_IA64_REG_CR_IIM`	4120
`_IA64_REG_CR_IHA`	4121
`_IA64_REG_CR_LID`	4160
`_IA64_REG_CR_IVR`	4161 ^
`_IA64_REG_CR_TPR`	4162
`_IA64_REG_CR_EOI`	4163
`_IA64_REG_CR_IRR0`	4164 ^
`_IA64_REG_CR_IRR1`	4165 ^
`_IA64_REG_CR_IRR2`	4166 ^
`_IA64_REG_CR_IRR3`	4167 ^
`_IA64_REG_CR_ITV`	4168
`_IA64_REG_CR_PMV`	4169
`_IA64_REG_CR_CMCV`	4170
`_IA64_REG_CR_LRR0`	4176
`_IA64_REG_CR_LRR1`	4177

^ `getReg` only

Indirect Registers for getIndReg() and setIndReg()

Name	whichReg
`_IA64_REG_INDR_CPUID`	9000 ^
`_IA64_REG_INDR_DBR`	9001
`_IA64_REG_INDR_IBR`	9002

_IA64_REG_INDR_PKR	9003
_IA64_REG_INDR_PMC	9004
_IA64_REG_INDR_PMD	9005
_IA64_REG_INDR_RR	9006
_IA64_REG_INDR_RESERVED	9007

^ `getIndReg` only

Multimedia Additions

The prototypes for these intrinsics are in the `ia64intrin.h` header file.

For detailed information about an intrinsic, click on the intrinsic name in the following table.

Intrinsic	Operation	Corresponding Itanium 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

190

| `_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.
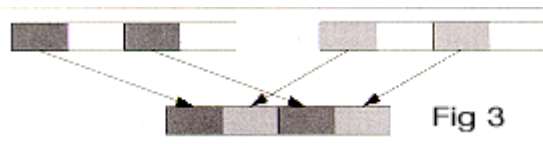

Fig 1
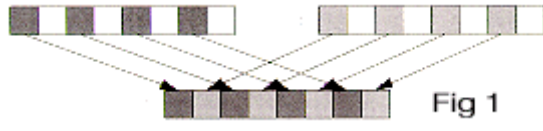
```
__m64 _m64_mix1r(__m64 a, __m64 b)
```

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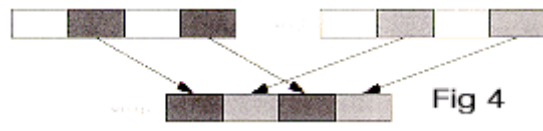

Fig 2
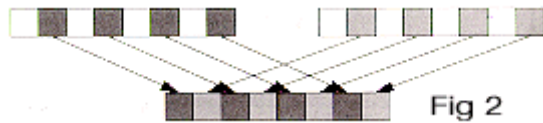
```
__m64 _m64_mix2l(__m64 a, __m64 b)
```

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

Fig 3

```
__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.

Fig 4

```
__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.

Fig 5

```
__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.

Fig 6

```
__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`.
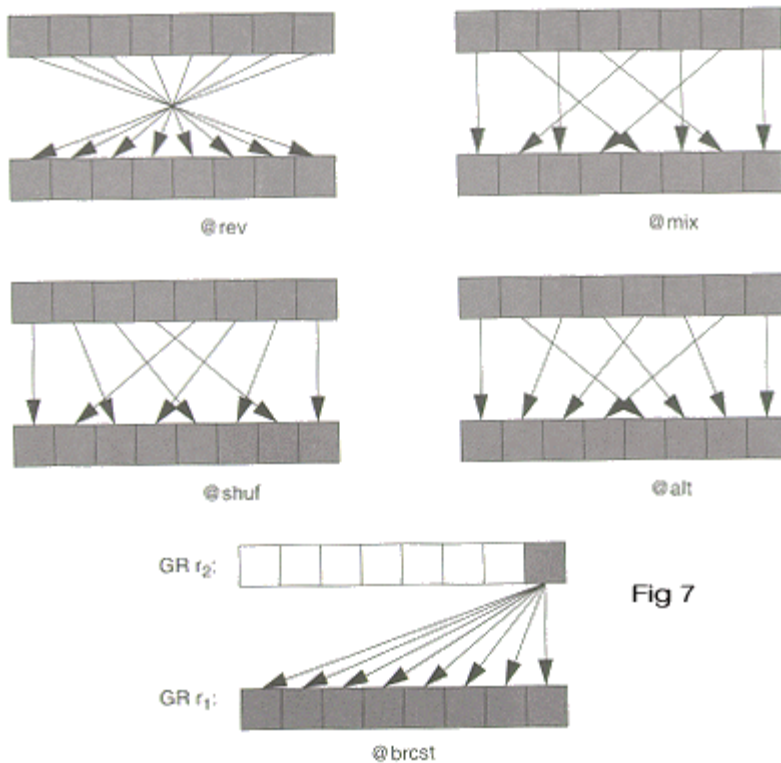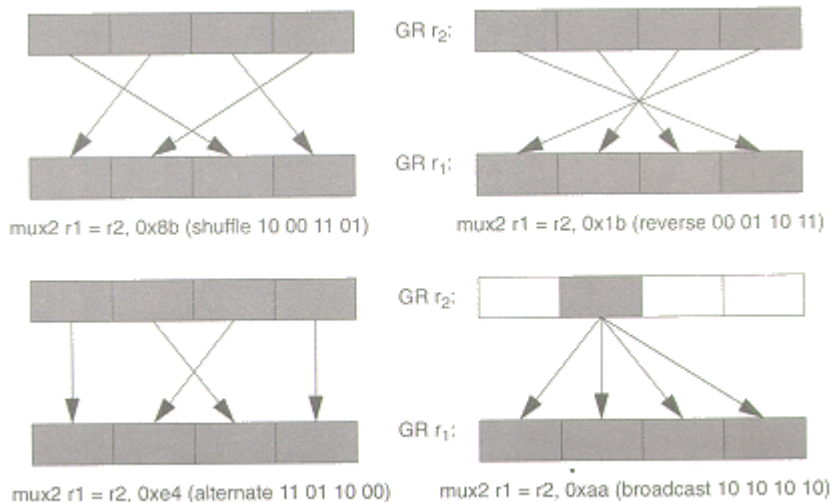

@rev

@mix

@shuf

@alt

GR r₂:

GR r₁:

@brcst

Fig 7

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.

mux2 r1 = r2, 0x8b (shuffle 10 00 11 01)   mux2 r1 = r2, 0x1b (reverse 00 01 10 11)

mux2 r1 = r2, 0xe4 (alternate 11 01 10 00)   mux2 r1 = r2, 0xaa (broadcast 10 10 10 10)

Fig 8

```
__m64 _m64_pavgsub1(__m64 a, __m64 b)
```

The unsigned data elements (bytes) of `b` are subtracted from the unsigned data elements (bytes) of `a` and the results of the subtraction are then each independently shifted to the right by one position. The high-order bits of each element are filled with the borrow bits of the subtraction.

```
__m64 _m64_pavgsub2(__m64 a, __m64 b)
```

The unsigned data elements (double bytes) of `b` are subtracted from the unsigned data elements (double bytes) of `a` and the results of the subtraction are then each

independently shifted to the right by one position. The high-order bits of each element are filled with the borrow bits of the subtraction.

`__m64 _m64_pmpy2l(__m64 a, __m64 b)`

Two signed 16-bit data elements of `a`, starting with the most significant data element, are multiplied by the corresponding two signed 16-bit data elements of `b`, and the two 32-bit results are returned as shown in Figure 9.

Fig 9

`__m64 _m64_pmpy2r(__m64 a, __m64 b)`

Two signed 16-bit data elements of `a`, starting with the least significant data element, are multiplied by the corresponding two signed 16-bit data elements of `b`, and the two 32-bit results are returned as shown in Figure 10.
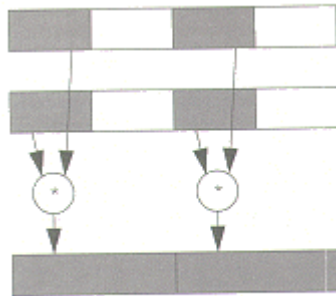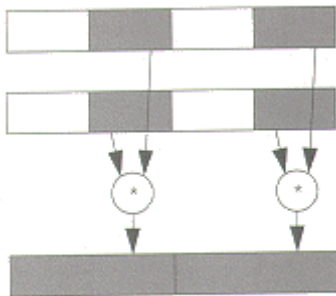

Fig 10

`__m64 _m64_pmpyshr2(__m64 a, __m64 b, const int count)`

The four signed 16-bit data elements of `a` are multiplied by the corresponding signed 16-bit data elements of `b`, yielding four 32-bit products. Each product is then shifted to the right count bits and the least significant 16 bits of each shifted product form 4 16-bit results, which are returned as one 64-bit word.

```
__m64 _m64_pmpyshr2u(__m64 a, __m64 b, const int count)
```

The four unsigned 16-bit data elements of `a` are multiplied by the corresponding unsigned 16-bit data elements of `b`, yielding four 32-bit products. Each product is then shifted to the right count bits and the least significant 16 bits of each shifted product form 4 16-bit results, which are returned as one 64-bit word.

```
__m64 _m64_pshladd2(__m64 a, const int count, __m64 b)
```

`a` is shifted to the left by count bits and then is added to `b`. The upper 32 bits of the result are forced to 0, and then bits [31:30] of `b` are copied to bits [62:61] of the result. The result is returned.

```
__m64 _m64_pshradd2(__m64 a, const int count, __m64 b)
```

The four signed 16-bit data elements of `a` are each independently shifted to the right by `count` bits (the high order bits of each element are filled with the initial value of the sign bits of the data elements in `a`); they are then added to the four signed 16-bit data elements of `b`. The result is returned.

```
__m64 _m64_padd1uus(__m64 a, __m64 b)
```

`a` is added to `b` as eight separate byte-wide elements. The elements of `a` are treated as unsigned, while the elements of `b` are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_padd2uus(__m64 a, __m64 b)
```

`a` is added to `b` as four separate 16-bit wide elements. The elements of `a` are treated as unsigned, while the elements of `b` are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_psub1uus(__m64 a, __m64 b)
```

`a` is subtracted from `b` as eight separate byte-wide elements. The elements of `a` are treated as unsigned, while the elements of `b` are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_psub2uus(__m64 a, __m64 b)
```

`a` is subtracted from `b` as four separate 16-bit wide elements. The elements of `a` are treated as unsigned, while the elements of `b` are treated as signed. The results are treated as unsigned and are returned as one 64-bit word.

```
__m64 _m64_pavg1_nraz(__m64 a, __m64 b)
```

The unsigned byte-wide data elements of `a` are added to the unsigned byte-wide data elements of `b` and the results of each add are then independently shifted to the right by one position. The high-order bits of each element are filled with the carry bits of the sums.

```
__m64 _m64_pavg2_nraz(__m64 a, __m64 b)
```

The unsigned 16-bit wide data elements of `a` are added to the unsigned 16-bit wide data elements of `b` and the results of each add are then independently shifted to the right by one position. The high-order bits of each element are filled with the carry bits of the sums.

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 Sequence

Intel® C++ Compiler for Linux* Intrinsics Reference

The Dual-Core Intel® Itanium® 2 Processor 9000 Sequence processor supports the intrinsics listed in the table below.

These intrinsics each generate Itanium 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 Itanium instruction.

For detailed information about an intrinsic, click on that intrinsic in the following table.

Click here for an example of several of these intrinsics.

For more information about the instructions these intrinsics generate, please see the Intel Itanium Architecture Software Developer Manual, Volume 3: Instruction Set Reference in the documentation area of the Itanium 2 processor website.

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 Itanium compare and exchange instruction.

Returns the original 64-bit value read from memory at the specified address.

Click here for an example of this intrinsic.

The following table describes each argument for this intrinsic.

sem	ldhint	addr	xchg_lo
Literal value between 0 and 1 that specifies the	Literal value between 0 and 2 that specifies the load hint	The address of the value	The least significant 8 bytes

semaphore completer (0==.acq, 1==.rel)	completer (0==.none, 1==.nt1, 2==.nta).	to read.	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 Itanium 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);]

Click here for an example of this intrinsic.

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 Itanium 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 Itanium 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 Itanium `hint@pause` instruction.

```
void __st16(const int <sttype>, const int <sthint>, void *<addr>,
__int64 <src_lo>)
```

Generates the Itanium instruction to store 16 bytes at the given address.

Click here for an example of this intrinsic.

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 Sequence

The Dual-Core Intel® Itanium® 2 Processor 9000 Sequence processor supports the intrinsics listed in the table below. These intrinsics are also compatible with the Microsoft compiler. These intrinsics each generate Itanium® instructions. The second alpha-numerical chain in the intrinsic name represents the Itanium instruction the intrinsic generates. For example, the intrinsic `_int64_cmp8xchg` generates the `cmp8xchg` Itanium instruction. For more information about the Itanium instructions that these intrinsics generate, see the Intel Itanium Architecture Software Developer Manual, Volume 3: Instruction Set Reference.

For detailed information about an intrinsic, click on that intrinsic name in the following table.

Intrinsic Name	Operation	Corresponding Itanium 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 Itanium 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 Itanium 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 Itanium 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 Itanium 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 Itanium 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 Itanium 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 Itanium 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

By default, the compiler inlines a number of standard C, C++, and math library functions. This usually results in faster execution of your program.

Sometimes inline expansion of library functions can cause unexpected results. The inlined library functions do not set the `errno` variable. So, in code that relies upon the setting of the `errno` variable, you should use the `-nolib_inline` option, which turns off inline expansion of library functions. Also, if one of your functions has the same name as one of the compiler's supplied library functions, the compiler assumes that it is one of the latter and replaces the call with the inlined version. Consequently, if the program defines a function with the same name as one of the known library routines, you must use the `-nolib_inline` option to ensure that the program's function is the one used.

Note

Automatic inline expansion of library functions is not related to the inline expansion that the compiler does during interprocedural optimizations. For example, the following command compiles the program sum.c without expanding the library functions, but with inline expansion from interprocedural optimizations (IPO):

```
prompt>icpc -ip -nolib_inline sum.cpp
```

For details on IPO, see Interprocedural Optimizations.

MASM* Style Inline Assembly

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

GNU*-like Style Inline Assembly (IA-32 only)

The Intel® C++ Compiler supports GNU-like style inline assembly. The syntax is as follows:

```
asm-keyword [ volatile-keyword ] ( asm-template [ asm-interface ] ) ;
```

⚠️**Caution**

The Intel C++ Compiler does not support the mixing UNIX and MASM style asms.

Syntax Element	Description
asm-keyword	`asm` statements begin with the keyword `asm`. Alternatively, either `__asm` or `__asm__` may be used for compatibility. See **Caution** statement.
volatile-keyword	If the optional keyword `volatile` is given, the `asm` is volatile. Two `volatile asm` statements will never be moved past each other, and a reference to a `volatile` variable will not be moved relative to a volatile `asm`. Alternate keywords `__volatile` and `__volatile__` may be used for compatibility.
asm-template	The `asm-template` is a C language ASCII string which specifies how to output the assembly code for an instruction. Most of the template is a fixed string; everything but the substitution-directives, if any, is passed through to the assembler. The syntax for a substitution directive is a `%` followed by one or two characters. The supported substitution directives are specified in a subsequent section.
asm-interface	The `asm-interface` consists of three parts: 1. an optional `output-list` 2. an optional `input-list` 3. an optional `clobber-list` These are separated by colon (:) characters. If the `output-list` is missing, but an `input-list` is given, the input list may be preceded by two

	colons (::)to take the place of the missing `output-list`. If the `asm-interface` is omitted altogether, the `asm` statement is considered `volatile` regardless of whether a `volatile-keyword` was specified.
`output-list`	An `output-list` consists of one or more `output-specs` separated by commas. For the purposes of substitution in the `asm-template`, each `output-spec` is numbered. The first operand in the `output-list` is numbered 0, the second is 1, and so on. Numbering is continuous through the `output-list` and into the `input-list`. The total number of operands is limited to 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.

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)
```

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 Streaming SIMD Extensions Streaming SIMD Extensions 2	Itanium® 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 Itanium 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 Itanium®-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	Streaming SIMD Extensions Streaming SIMD Extensions 2	Itanium® 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

The Pentium® 4 processor is the only Intel processor that supports Streaming SIMD Extensions 2 (SSE2) intrinsics.

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

Index