Contents

__readcr8

__readdr

__readeflags

__readfsbyte, __readfsdword, __readfsqword, __readfsword

__readgsbyte, __readgsdword, __readgsqword, __readgsword

__readmsr

__readpmc

_ReadWriteBarrier

_ReturnAddress

_rotl8, _rotl16

_rotr8, _rotr16

__segmentlimit

__shiftleft128

__shiftright128

__sidt

__stosb

__stosd

__stosq

__stosw

__svm_clgi

__svm_invlpga

__svm_skinit

__svm_stgi

__svm_vmload

__svm_vmrun

__svm_vmsave

__ud2

_udiv128

_udiv64

__ull_rshift

_umul128

__umulh

Compiler intrinsics

9/2/2022 • 2 minutes to read • Edit Online

Most functions are contained in libraries, but some functions are built in (that is, intrinsic) to the compiler. These are referred to as intrinsic functions or intrinsics.

Remarks

If a function is an intrinsic, the code for that function is usually inserted inline, avoiding the overhead of a function call and allowing highly efficient machine instructions to be emitted for that function. An intrinsic is often faster than the equivalent inline assembly, because the optimizer has a built-in knowledge of how many intrinsics behave, so some optimizations can be available that are not available when inline assembly is used. Also, the optimizer can expand the intrinsic differently, align buffers differently, or make other adjustments depending on the context and arguments of the call.

The use of intrinsics affects the portability of code, because intrinsics that are available in Visual C++ might not be available if the code is compiled with other compilers and some intrinsics that might be available for some target architectures are not available for all architectures. However, intrinsics are usually more portable than inline assembly. The intrinsics are required on 64-bit architectures where inline assembly is not supported.

Some intrinsics, such as `__assume` and `__ReadWriteBarrier`, provide information to the compiler, which affects the behavior of the optimizer.

Some intrinsics are available only as intrinsics, and some are available both in function and intrinsic implementations. You can instruct the compiler to use the intrinsic implementation in one of two ways, depending on whether you want to enable only specific functions or you want to enable all intrinsics. The first way is to use `#pragma intrinsic(` *intrinsic-function-name-list* `)`. The pragma can be used to specify a single intrinsic or multiple intrinsics separated by commas. The second is to use the /Oi (Generate intrinsic functions) compiler option, which makes all intrinsics on a given platform available. Under **/Oi**, use `#pragma function(` *intrinsic-function-name-list* `)` to force a function call to be used instead of an intrinsic. If the documentation for a specific intrinsic notes that the routine is only available as an intrinsic, then the intrinsic implementation is used regardless of whether **/Oi** or `#pragma intrinsic` is specified. In all cases, **/Oi** or `#pragma intrinsic` allows, but does not force, the optimizer to use the intrinsic. The optimizer can still call the function.

Some standard C/C++ library functions are available in intrinsic implementations on some architectures. When calling a CRT function, the intrinsic implementation is used if **/Oi** is specified on the command line.

A header file, <intrin.h>, is available that declares prototypes for the common intrinsic functions. Manufacturer-specific intrinsics are available in the <immintrin.h> and <ammintrin.h> header files. Additionally, certain Windows headers declare functions that map onto a compiler intrinsic.

The following sections list all intrinsics that are available on various architectures. For more information on how the intrinsics work on your particular target processor, refer to the manufacturer's reference documentation.

- ARM intrinsics
- ARM64 intrinsics
- x86 intrinsics list
- x64 (amd64) Intrinsics List
- Intrinsics available on all architectures

- Alphabetical listing of intrinsic functions

See also

ARM assembler reference
Microsoft Macro Assembler reference
Keywords
C run-time library reference

ARM intrinsics

9/2/2022 • 23 minutes to read • Edit Online

The Microsoft C++ compiler (MSVC) makes the following intrinsics available on the ARM architecture. For more information about ARM, see the Architecture and Software Development Tools sections of the ARM Developer Documentation website.

NEON

The NEON vector instruction set extensions for ARM provide Single Instruction Multiple Data (SIMD) capabilities that resemble the ones in the MMX and SSE vector instruction sets that are common to x86 and x64 architecture processors.

NEON intrinsics are supported, as provided in the header file `arm_neon.h`. The MSVC support for NEON intrinsics resembles that of the ARM compiler, which is documented in Appendix G of the ARM Compiler toolchain, Version 4.1 Compiler Reference on the ARM Infocenter website.

The primary difference between MSVC and the ARM compiler is that the MSVC adds `_ex` variants of the `vldX` and `vstX` vector load and store instructions. The `_ex` variants take an additional parameter that specifies the alignment of the pointer argument but are otherwise identical to their non-`_ex` counterparts.

ARM-specific Intrinsics Listing

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
_arm_smlal	SMLAL	__int64 _arm_smlal(__int64 _RdHiLo, int _Rn, int _Rm)
_arm_umlal	UMLAL	unsigned __int64 _arm_umlal(unsigned __int64 _RdHiLo, unsigned int _Rn, unsigned int _Rm)
_arm_clz	CLZ	unsigned int _arm_clz(unsigned int _Rm)
_arm_qadd	QADD	int _arm_qadd(int _Rm, int _Rn)
_arm_qdadd	QDADD	int _arm_qdadd(int _Rm, int _Rn)
_arm_qdsub	QDSUB	int _arm_qdsub(int _Rm, int _Rn)
_arm_qsub	QSUB	int _arm_qsub(int _Rm, int _Rn)
_arm_smlabb	SMLABB	int _arm_smlabb(int _Rn, int _Rm, int _Ra)
_arm_smlabt	SMLABT	int _arm_smlabt(int _Rn, int _Rm, int _Ra)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
_arm_smlatb	SMLATB	int _arm_smlatb(int _Rn, int _Rm, int _Ra)
_arm_smlatt	SMLATT	int _arm_smlatt(int _Rn, int _Rm, int _Ra)
_arm_smlalbb	SMLALBB	__int64 _arm_smlalbb(__int64 _RdHiLo, int _Rn, int _Rm)
_arm_smlalbt	SMLALBT	__int64 _arm_smlalbt(__int64 _RdHiLo, int _Rn, int _Rm)
_arm_smlaltb	SMLALTB	__int64 _arm_smlaltb(__int64 _RdHiLo, int _Rn, int _Rm)
_arm_smlaltt	SMLALTT	__int64 _arm_smlaltt(__int64 _RdHiLo, int _Rn, int _Rm)
_arm_smlawb	SMLAWB	int _arm_smlawb(int _Rn, int _Rm, int _Ra)
_arm_smlawt	SMLAWT	int _arm_smlawt(int _Rn, int _Rm, int _Ra)
_arm_smulbb	SMULBB	int _arm_smulbb(int _Rn, int _Rm)
_arm_smulbt	SMULBT	int _arm_smulbt(int _Rn, int _Rm)
_arm_smultb	SMULTB	int _arm_smultb(int _Rn, int _Rm)
_arm_smultt	SMULTT	int _arm_smultt(int _Rn, int _Rm)
_arm_smulwb	SMULWB	int _arm_smulwb(int _Rn, int _Rm)
_arm_smulwt	SMULWT	int _arm_smulwt(int _Rn, int _Rm)
_arm_sadd16	SADD16	int _arm_sadd16(int _Rn, int _Rm)
_arm_sadd8	SADD8	int _arm_sadd8(int _Rn, int _Rm)
_arm_sasx	SASX	int _arm_sasx(int _Rn, int _Rm)
_arm_ssax	SSAX	int _arm_ssax(int _Rn, int _Rm)
_arm_ssub16	SSUB16	int _arm_ssub16(int _Rn, int _Rm)
_arm_ssub8	SSUB8	int _arm_ssub8(int _Rn, int _Rm)
_arm_shadd16	SHADD16	int _arm_shadd16(int _Rn, int _Rm)
_arm_shadd8	SHADD8	int _arm_shadd8(int _Rn, int _Rm)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
_arm_shasx	SHASX	int _arm_shasx(int _Rn, int _Rm)
_arm_shsax	SHSAX	int _arm_shsax(int _Rn, int _Rm)
_arm_shsub16	SHSUB16	int _arm_shsub16(int _Rn, int _Rm)
_arm_shsub8	SHSUB8	int _arm_shsub8(int _Rn, int _Rm)
_arm_qadd16	QADD16	int _arm_qadd16(int _Rn, int _Rm)
_arm_qadd8	QADD8	int _arm_qadd8(int _Rn, int _Rm)
_arm_qasx	QASX	int _arm_qasx(int _Rn, int _Rm)
_arm_qsax	QSAX	int _arm_qsax(int _Rn, int _Rm)
_arm_qsub16	QSUB16	int _arm_qsub16(int _Rn, int _Rm)
_arm_qsub8	QSUB8	int _arm_qsub8(int _Rn, int _Rm)
_arm_uadd16	UADD16	unsigned int _arm_uadd16(unsigned int _Rn, unsigned int _Rm)
_arm_uadd8	UADD8	unsigned int _arm_uadd8(unsigned int _Rn, unsigned int _Rm)
_arm_uasx	UASX	unsigned int _arm_uasx(unsigned int _Rn, unsigned int _Rm)
_arm_usax	USAX	unsigned int _arm_usax(unsigned int _Rn, unsigned int _Rm)
_arm_usub16	USUB16	unsigned int _arm_usub16(unsigned int _Rn, unsigned int _Rm)
_arm_usub8	USUB8	unsigned int _arm_usub8(unsigned int _Rn, unsigned int _Rm)
_arm_uhadd16	UHADD16	unsigned int _arm_uhadd16(unsigned int _Rn, unsigned int _Rm)
_arm_uhadd8	UHADD8	unsigned int _arm_uhadd8(unsigned int _Rn, unsigned int _Rm)
_arm_uhasx	UHASX	unsigned int _arm_uhasx(unsigned int _Rn, unsigned int _Rm)
_arm_uhsax	UHSAX	unsigned int _arm_uhsax(unsigned int _Rn, unsigned int _Rm)
_arm_uhsub16	UHSUB16	unsigned int _arm_uhsub16(unsigned int _Rn, unsigned int _Rm)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
_arm_uhsub8	UHSUB8	unsigned int _arm_uhsub8(unsigned int _Rn, unsigned int _Rm)
_arm_uqadd16	UQADD16	unsigned int _arm_uqadd16(unsigned int _Rn, unsigned int _Rm)
_arm_uqadd8	UQADD8	unsigned int _arm_uqadd8(unsigned int _Rn, unsigned int _Rm)
_arm_uqasx	UQASX	unsigned int _arm_uqasx(unsigned int _Rn, unsigned int _Rm)
_arm_uqsax	UQSAX	unsigned int _arm_uqsax(unsigned int _Rn, unsigned int _Rm)
_arm_uqsub16	UQSUB16	unsigned int _arm_uqsub16(unsigned int _Rn, unsigned int _Rm)
_arm_uqsub8	UQSUB8	unsigned int _arm_uqsub8(unsigned int _Rn, unsigned int _Rm)
_arm_sxtab	SXTAB	int _arm_sxtab(int _Rn, int _Rm, unsigned int _Rotation)
_arm_sxtab16	SXTAB16	int _arm_sxtab16(int _Rn, int _Rm, unsigned int _Rotation)
_arm_sxtah	SXTAH	int _arm_sxtah(int _Rn, int _Rm, unsigned int _Rotation)
_arm_uxtab	UXTAB	unsigned int _arm_uxtab(unsigned int _Rn, unsigned int _Rm, unsigned int _Rotation)
_arm_uxtab16	UXTAB16	unsigned int _arm_uxta16b(unsigned int _Rn, unsigned int _Rm, unsigned int _Rotation)
_arm_uxtah	UXTAH	unsigned int _arm_uxtah(unsigned int _Rn, unsigned int _Rm, unsigned int _Rotation)
_arm_sxtb	SXTB	int _arm_sxtb(int _Rn, unsigned int _Rotation)
_arm_sxtb16	SXTB16	int _arm_sxtb16(int _Rn, unsigned int _Rotation)
_arm_sxth	SXTH	int _arm_sxth(int _Rn, unsigned int _Rotation)
_arm_uxtb	UXTB	unsigned int _arm_uxtb(unsigned int _Rn, unsigned int _Rotation)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
_arm_uxtb16	UXTB16	unsigned int _arm_uxtb16(unsigned int _Rn, unsigned int _Rotation)
_arm_uxth	UXTH	unsigned int _arm_uxth(unsigned int _Rn, unsigned int _Rotation)
_arm_pkhbt	PKHBT	int _arm_pkhbt(int _Rn, int _Rm, unsigned int _Lsl_imm)
_arm_pkhtb	PKHTB	int _arm_pkhtb(int _Rn, int _Rm, unsigned int _Asr_imm)
_arm_usad8	USAD8	unsigned int _arm_usad8(unsigned int _Rn, unsigned int _Rm)
_arm_usada8	USADA8	unsigned int _arm_usada8(unsigned int _Rn, unsigned int _Rm, unsigned int _Ra)
_arm_ssat	SSAT	int _arm_ssat(unsigned int _Sat_imm, _int _Rn, _ARMINTR_SHIFT_T _Shift_type, unsigned int _Shift_imm)
_arm_usat	USAT	int _arm_usat(unsigned int _Sat_imm, _int _Rn, _ARMINTR_SHIFT_T _Shift_type, unsigned int _Shift_imm)
_arm_ssat16	SSAT16	int _arm_ssat16(unsigned int _Sat_imm, _int _Rn)
_arm_usat16	USAT16	int _arm_usat16(unsigned int _Sat_imm, _int _Rn)
_arm_rev	REV	unsigned int _arm_rev(unsigned int _Rm)
_arm_rev16	REV16	unsigned int _arm_rev16(unsigned int _Rm)
_arm_revsh	REVSH	unsigned int _arm_revsh(unsigned int _Rm)
_arm_smlad	SMLAD	int _arm_smlad(int _Rn, int _Rm, int _Ra)
_arm_smladx	SMLADX	int _arm_smladx(int _Rn, int _Rm, int _Ra)
_arm_smlsd	SMLSD	int _arm_smlsd(int _Rn, int _Rm, int _Ra)
_arm_smlsdx	SMLSDX	int _arm_smlsdx(int _Rn, int _Rm, int _Ra)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
_arm_smmla	SMMLA	int _arm_smmla(int _Rn, int _Rm, int _Ra)
_arm_smmlar	SMMLAR	int _arm_smmlar(int _Rn, int _Rm, int _Ra)
_arm_smmls	SMMLS	int _arm_smmls(int _Rn, int _Rm, int _Ra)
_arm_smmlsr	SMMLSR	int _arm_smmlsr(int _Rn, int _Rm, int _Ra)
_arm_smmul	SMMUL	int _arm_smmul(int _Rn, int _Rm)
_arm_smmulr	SMMULR	int _arm_smmulr(int _Rn, int _Rm)
_arm_smlald	SMLALD	__int64 _arm_smlald(__int64 _RdHiLo, int _Rn, int _Rm)
_arm_smlaldx	SMLALDX	__int64 _arm_smlaldx(__int64 _RdHiLo, int _Rn, int _Rm)
_arm_smlsld	SMLSLD	__int64 _arm_smlsld(__int64 _RdHiLo, int _Rn, int _Rm)
_arm_smlsldx	SMLSLDX	__int64 _arm_smlsldx(__int64 _RdHiLo, int _Rn, int _Rm)
_arm_smuad	SMUAD	int _arm_smuad(int _Rn, int _Rm)
_arm_smuadx	SMUADX	int _arm_muadxs(int _Rn, int _Rm)
_arm_smusd	SMUSD	int _arm_smusd(int _Rn, int _Rm)
_arm_smusdx	SMUSDX	int _arm_smusdx(int _Rn, int _Rm)
_arm_smull	SMULL	__int64 _arm_smull(int _Rn, int _Rm)
_arm_umull	UMULL	unsigned __int64 _arm_umull(unsigned int _Rn, unsigned int _Rm)
_arm_umaal	UMAAL	unsigned __int64 _arm_umaal(unsigned int _RdLo, unsigned int _RdHi, unsigned int _Rn, unsigned int _Rm)
_arm_bfc	BFC	unsigned int _arm_bfc(unsigned int _Rd, unsigned int _Lsb, unsigned int _Width)
_arm_bfi	BFI	unsigned int _arm_bfi(unsigned int _Rd, unsigned int _Rn, unsigned int _Lsb, unsigned int _Width)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
_arm_rbit	RBIT	unsigned int _arm_rbit(unsigned int _Rm)
_arm_sbfx	SBFX	int _arm_sbfx(int _Rn, unsigned int _Lsb, unsigned int _Width)
_arm_ubfx	UBFX	unsigned int _arm_ubfx(unsigned int _Rn, unsigned int _Lsb, unsigned int _Width)
_arm_sdiv	SDIV	int _arm_sdiv(int _Rn, int _Rm)
_arm_udiv	UDIV	unsigned int _arm_udiv(unsigned int _Rn, unsigned int _Rm)
__cps	CPS	void __cps(unsigned int _Ops, unsigned int _Flags, unsigned int _Mode)
__dmb	DMB	void __dmb(unsigned int `_Type`) Inserts a memory barrier operation into the instruction stream. The parameter `_Type` specifies the kind of restriction that the barrier enforces. For more information about the kinds of restrictions that can be enforced, see Memory Barrier Restrictions.
__dsb	DSB	void __dsb(unsigned int _Type) Inserts a memory barrier operation into the instruction stream. The parameter `_Type` specifies the kind of restriction that the barrier enforces. For more information about the kinds of restrictions that can be enforced, see Memory Barrier Restrictions.
__isb	ISB	void __isb(unsigned int _Type) Inserts a memory barrier operation into the instruction stream. The parameter `_Type` specifies the kind of restriction that the barrier enforces. For more information about the kinds of restrictions that can be enforced, see Memory Barrier Restrictions.

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
__emit		void __emit(unsigned __int32 opcode) Inserts a specified instruction into the stream of instructions that is output by the compiler. The value of `opcode` must be a constant expression that is known at compile time. The size of an instruction word is 16 bits and the most significant 16 bits of `opcode` are ignored. The compiler makes no attempt to interpret the contents of `opcode` and doesn't guarantee a CPU or memory state before the inserted instruction is executed. The compiler assumes that the CPU and memory states are unchanged after the inserted instruction is executed. Therefore, instructions that do change state can have a detrimental impact on normal code that's generated by the compiler. For this reason, use `emit` only to insert instructions that affect a CPU state that the compiler doesn't normally process—for example, coprocessor state—or to implement functions that are declared by using `declspec(naked)`.
__hvc	HVC	unsigned int __hvc(unsigned int, ...)
__iso_volatile_load16		__int16 __iso_volatile_load16(const volatile __int16 *) For more information, see __iso_volatile_load/store intrinsics.
__iso_volatile_load32		__int32 __iso_volatile_load32(const volatile __int32 *) For more information, see __iso_volatile_load/store intrinsics.
__iso_volatile_load64		__int64 __iso_volatile_load64(const volatile __int64 *) For more information, see __iso_volatile_load/store intrinsics.

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
__iso_volatile_load8		__int8 __iso_volatile_load8(const volatile __int8 *) For more information, see __iso_volatile_load/store intrinsics.
__iso_volatile_store16		void __iso_volatile_store16(volatile __int16 *, __int16) For more information, see __iso_volatile_load/store intrinsics.
__iso_volatile_store32		void __iso_volatile_store32(volatile __int32 *, __int32) For more information, see __iso_volatile_load/store intrinsics.
__iso_volatile_store64		void __iso_volatile_store64(volatile __int64 *, __int64) For more information, see __iso_volatile_load/store intrinsics.
__iso_volatile_store8		void __iso_volatile_store8(volatile __int8 *, __int8) For more information, see __iso_volatile_load/store intrinsics.
__ldrexd	LDREXD	__int64 __ldrexd(const volatile __int64 *)
__prefetch	PLD	void __cdecl __prefetch(const void *) Provides a `PLD` memory hint to the system that memory at or near the specified address may be accessed soon. Some systems may choose to optimize for that memory access pattern to increase runtime performance. However, from the C++ language point of view, the function has no observable effect, and may do nothing at all.
__rdpmccntr64		unsigned __int64 __rdpmccntr64(void)
__sev	SEV	void __sev(void)
__static_assert		void __static_assert(int, const char *)
__swi	SVC	unsigned int __swi(unsigned int, ...)
__trap	BKPT	int __trap(int, ...)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
__wfe	WFE	void __wfe(void)
__wfi	WFI	void __wfi(void)
_AddSatInt	QADD	int _AddSatInt(int, int)
_CopyDoubleFromInt64		double _CopyDoubleFromInt64(__int64)
_CopyFloatFromInt32		float _CopyFloatFromInt32(__int32)
_CopyInt32FromFloat		__int32 _CopyInt32FromFloat(float)
_CopyInt64FromDouble		__int64 _CopyInt64FromDouble(double)
_CountLeadingOnes		unsigned int _CountLeadingOnes(unsigned long)
_CountLeadingOnes64		unsigned int _CountLeadingOnes64(unsigned __int64)
_CountLeadingSigns		unsigned int _CountLeadingSigns(long)
_CountLeadingSigns64		unsigned int _CountLeadingSigns64(__int64)
_CountLeadingZeros		unsigned int _CountLeadingZeros(unsigned long)
_CountLeadingZeros64		unsigned int _CountLeadingZeros64(unsigned __int64)
_CountOneBits		unsigned int _CountOneBits(unsigned long)
_CountOneBits64		unsigned int _CountOneBits64(unsigned __int64)
_DAddSatInt	QDADD	int _DAddSatInt(int, int)
_DSubSatInt	QDSUB	int _DSubSatInt(int, int)
_isunordered		int _isunordered(double, double)
_isunorderedf		int _isunorderedf(float, float)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
_MoveFromCoprocessor	MRC	unsigned int _MoveFromCoprocessor(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int) Reads data from an ARM coprocessor by using the coprocessor data transfer instructions. For more information, see _MoveFromCoprocessor, _MoveFromCoprocessor2.
_MoveFromCoprocessor2	MRC2	unsigned int _MoveFromCoprocessor2(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int) Reads data from an ARM coprocessor by using the coprocessor data transfer instructions. For more information, see _MoveFromCoprocessor, _MoveFromCoprocessor2.
_MoveFromCoprocessor64	MRRC	unsigned __int64 _MoveFromCoprocessor64(unsigned int, unsigned int, unsigned int) Reads data from an ARM coprocessor by using the coprocessor data transfer instructions. For more information, see _MoveFromCoprocessor64.
_MoveToCoprocessor	MCR	void _MoveToCoprocessor(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, unsigned int) Reads data from an ARM coprocessor by using the coprocessor data transfer instructions. For more information, see _MoveToCoprocessor, _MoveToCoprocessor2.
_MoveToCoprocessor2	MCR2	void _MoveToCoprocessor2(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, unsigned int) Reads data from an ARM coprocessor by using the coprocessor data transfer instructions. For more information, see _MoveToCoprocessor, _MoveToCoprocessor2.

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
_MoveToCoprocessor64	MCRR	void _MoveToCoprocessor64(unsigned __int64, unsigned int, unsigned int, unsigned int) Reads data from an ARM coprocessor by using the coprocessor data transfer instructions. For more information, see _MoveToCoprocessor64.
_MulHigh		long _MulHigh(long, long)
_MulUnsignedHigh		unsigned long _MulUnsignedHigh(unsigned long, unsigned long)
_ReadBankedReg	MRS	int _ReadBankedReg(int _Reg)
_ReadStatusReg	MRS	int _ReadStatusReg(int)
_SubSatInt	QSUB	int _SubSatInt(int, int)
_WriteBankedReg	MSR	void _WriteBankedReg(int _Value, int _Reg)
_WriteStatusReg	MSR	void _WriteStatusReg(int, int, int)

[Return to top]

Memory Barrier Restrictions

The intrinsic functions `__dmb` (data memory barrier), `__dsb` (data synchronization barrier), and `__isb` (instruction synchronization barrier) use the following predefined values to specify the memory barrier restriction in terms of the sharing domain and the kind of access that are affected by the operation.

RESTRICTION VALUE	DESCRIPTION
_ARM_BARRIER_SY	Full system, reads and writes.
_ARM_BARRIER_ST	Full system, writes only.
_ARM_BARRIER_ISH	Inner sharable, reads and writes.
_ARM_BARRIER_ISHST	Inner sharable, writes only.
_ARM_BARRIER_NSH	Non-sharable, reads and writes.
_ARM_BARRIER_NSHST	Non-sharable, writes only.
_ARM_BARRIER_OSH	Outer sharable, reads and writes.
_ARM_BARRIER_OSHST	Outer sharable, writes only.

For the `__isb` intrinsic, the only restriction that is currently valid is _ARM_BARRIER_SY; all other values are reserved by the architecture.

__iso_volatile_load/store intrinsics

These intrinsic functions explicitly perform loads and stores that aren't subject to compiler optimizations.

```
__int16 __iso_volatile_load16(const volatile __int16 * Location);
__int32 __iso_volatile_load32(const volatile __int32 * Location);
__int64 __iso_volatile_load64(const volatile __int64 * Location);
__int8 __iso_volatile_load8(const volatile __int8 * Location);

void __iso_volatile_store16(volatile __int16 * Location, __int16 Value);
void __iso_volatile_store32(volatile __int32 * Location, __int32 Value);
void __iso_volatile_store64(volatile __int64 * Location, __int64 Value);
void __iso_volatile_store8(volatile __int8 * Location, __int8 Value);
```

Parameters

Location
The address of a memory location to read from or write to.

Value
The value to write to the specified memory location (store intrinsics only).

Return value (load intrinsics only)

The value of the memory location that is specified by `Location`.

Remarks

You can use the `__iso_volatile_load8/16/32/64` and `__iso_volatile_store8/16/32/64` intrinsics to explicitly perform memory accesses that aren't subject to compiler optimizations. The compiler can't remove, synthetize, or change the relative order of these operations, but it doesn't generate implicit hardware memory barriers. Therefore, the hardware may still reorder the observable memory accesses across multiple threads. More precisely, these intrinsics are equivalent to the following expressions as compiled under **/volatile:iso**.

```
int a = __iso_volatile_load32(p);    // equivalent to: int a = *(const volatile __int32*)p;
__iso_volatile_store32(p, a);        // equivalent to: *(volatile __int32*)p = a;
```

Notice that the intrinsics take volatile pointers to accommodate volatile variables. However, there's no requirement or recommendation to use volatile pointers as arguments. The semantics of these operations are exactly the same if a regular, non-volatile type is used.

For more information about the **/volatile:iso** command-line argument, see /volatile (volatile Keyword Interpretation).

_MoveFromCoprocessor, _MoveFromCoprocessor2

These intrinsic functions read data from ARM coprocessors by using the coprocessor data transfer instructions.

```
int _MoveFromCoprocessor(
      unsigned int coproc,
      unsigned int opcode1,
      unsigned int crn,
      unsigned int crm,
      unsigned int opcode2
);

int _MoveFromCoprocessor2(
      unsigned int coproc,
      unsigned int opcode1,
      unsigned int crn,
      unsigned int crm,
      unsigned int opcode2
);
```

Parameters

coproc
Coprocessor number in the range 0 to 15.

opcode1
Coprocessor-specific opcode in the range 0 to 7

crn
Coprocessor register number, in the range 0 to 15, that specifies the first operand to the instruction.

crm
Coprocessor register number, in the range 0 to 15, that specifies an additional source or destination operand.

opcode2
Additional coprocessor-specific opcode in the range 0 to 7.

Return value

The value that is read from the coprocessor.

Remarks

The values of all five parameters of the intrinsic must be constant expressions that are known at compile time.

`_MoveFromCoprocessor` uses the MRC instruction; `_MoveFromCoprocessor2` uses MRC2. The parameters correspond to bitfields that are encoded directly into the instruction word. The interpretation of the parameters is coprocessor-dependent. For more information, see the manual for the coprocessor in question.

_MoveFromCoprocessor64

Reads data from ARM coprocessors by using the coprocessor data transfer instructions.

```
unsigned __int64 _MoveFromCoprocessor64(
      unsigned int coproc,
      unsigned int opcode1,
      unsigned int crm,
);
```

Parameters

coproc
Coprocessor number in the range 0 to 15.

opcode1
Coprocessor-specific opcode in the range 0 to 15.

crm
Coprocessor register number, in the range 0 to 15, that specifies an additional source or destination operand.

Return value

The value that is read from the coprocessor.

Remarks

The values of all three parameters of the intrinsic must be constant expressions that are known at compile time.

`_MoveFromCoprocessor64` uses the MRRC instruction. The parameters correspond to bitfields that are encoded directly into the instruction word. The interpretation of the parameters is coprocessor-dependent. For more information, see the manual for the coprocessor in question.

_MoveToCoprocessor, _MoveToCoprocessor2

These intrinsic functions write data to ARM coprocessors by using the coprocessor data transfer instructions.

```
void _MoveToCoprocessor(
      unsigned int value,
      unsigned int coproc,
      unsigned int opcode1,
      unsigned int crn,
      unsigned int crm,
      unsigned int opcode2
);

void _MoveToCoprocessor2(
      unsigned int value,
      unsigned int coproc,
      unsigned int opcode1,
      unsigned int crn,
      unsigned int crm,
      unsigned int opcode2
);
```

Parameters

value
The value to be written to the coprocessor.

coproc
Coprocessor number in the range 0 to 15.

opcode1
Coprocessor-specific opcode in the range 0 to 7.

crn
Coprocessor register number, in the range 0 to 15, that specifies the first operand to the instruction.

crm
Coprocessor register number, in the range 0 to 15, that specifies an additional source or destination operand.

opcode2
Additional coprocessor-specific opcode in the range 0 to 7.

Return value

None.

Remarks

The values of the `coproc`, `opcode1`, `crn`, `crm`, and `opcode2` parameters of the intrinsic must be constant expressions that are known at compile time.

`_MoveToCoprocessor` uses the MCR instruction; `_MoveToCoprocessor2` uses MCR2. The parameters correspond to bitfields that are encoded directly into the instruction word. The interpretation of the parameters is coprocessor-dependent. For more information, see the manual for the coprocessor in question.

_MoveToCoprocessor64

These intrinsic functions write data to ARM coprocessors by using the coprocessor data transfer instructions.

```
void _MoveFromCoprocessor64(
      unsigned __int64 value,
      unsigned int coproc,
      unsigned int opcode1,
      unsigned int crm,
);
```

Parameters

coproc

Coprocessor number in the range 0 to 15.

opcode1
Coprocessor-specific opcode in the range 0 to 15.

crm
Coprocessor register number, in the range 0 to 15, that specifies an additional source or destination operand.

Return value

None.

Remarks

The values of the `coproc`, `opcode1`, and `crm` parameters of the intrinsic must be constant expressions that are known at compile time.

`_MoveFromCoprocessor64` uses the MCRR instruction. The parameters correspond to bitfields that are encoded directly into the instruction word. The interpretation of the parameters is coprocessor-dependent. For more information, see the manual for the coprocessor in question.

ARM Support for Intrinsics from Other Architectures

The following table lists intrinsics from other architectures that are supported on ARM platforms. Where the behavior of an intrinsic on ARM differs from its behavior on other hardware architectures, additional details are noted.

FUNCTION NAME	FUNCTION PROTOTYPE
__assume	void __assume(int)
__code_seg	void __code_seg(const char *)
__debugbreak	void __cdecl __debugbreak(void)
__fastfail	__declspec(noreturn) void __fastfail(unsigned int)
__nop	void __nop(void) **Note:** On ARM platforms, this function generates a NOP instruction if one is implemented in the target architecture; otherwise, an alternative instruction that does not change the state of the program or CPU is generated—for example, `MOV r8, r8`. It's functionally equivalent to the __nop intrinsic for other hardware architectures. Because an instruction that has no effect on the state of the program or CPU might be ignored by the target architecture as an optimization, the instruction doesn't necessarily consume CPU cycles. Therefore, do not use the __nop intrinsic to manipulate the execution time of a code sequence unless you're certain about how the CPU will behave. Instead, you can use the __nop intrinsic to align the next instruction to a specific 32-bit boundary address.
__yield	void __yield(void) **Note:** On ARM platforms, this function generates the YIELD instruction, which indicates that the thread is performing a task that can be temporarily suspended from execution—for example, a spinlock—without adversely affecting the program. It enables the CPU to execute other tasks during execution cycles that would otherwise be wasted.

FUNCTION NAME	FUNCTION PROTOTYPE
_AddressOfReturnAddress	void * _AddressOfReturnAddress(void)
_BitScanForward	unsigned char _BitScanForward(unsigned long * _Index, unsigned long _Mask)
_BitScanReverse	unsigned char _BitScanReverse(unsigned long * _Index, unsigned long _Mask)
_bittest	unsigned char _bittest(long const *, long)
_bittestandcomplement	unsigned char _bittestandcomplement(long *, long)
_bittestandreset	unsigned char _bittestandreset(long *, long)
_bittestandset	unsigned char _bittestandset(long *, long)
_byteswap_uint64	unsigned __int64 __cdecl _byteswap_uint64(unsigned __int64)
_byteswap_ulong	unsigned long __cdecl _byteswap_ulong(unsigned long)
_byteswap_ushort	unsigned short __cdecl _byteswap_ushort(unsigned short)
_disable	void __cdecl _disable(void) **Note:** On ARM platforms, this function generates the CPSID instruction; it's only available as an intrinsic.
_enable	void __cdecl _enable(void) **Note:** On ARM platforms, this function generates the CPSIE instruction; it's only available as an intrinsic.
_lrotl	unsigned long __cdecl _lrotl(unsigned long, int)
_lrotr	unsigned long __cdecl _lrotr(unsigned long, int)
_ReadBarrier	void _ReadBarrier(void)
_ReadWriteBarrier	void _ReadWriteBarrier(void)
_ReturnAddress	void * _ReturnAddress(void)
_rotl	unsigned int __cdecl _rotl(unsigned int _Value, int _Shift)
_rotl16	unsigned short _rotl16(unsigned short _Value, unsigned char _Shift)
_rotl64	unsigned __int64 __cdecl _rotl64(unsigned __int64 _Value, int _Shift)
_rotl8	unsigned char _rotl8(unsigned char _Value, unsigned char _Shift)

FUNCTION NAME	FUNCTION PROTOTYPE
_rotr	unsigned int __cdecl _rotr(unsigned int _Value, int _Shift)
_rotr16	unsigned short _rotr16(unsigned short _Value, unsigned char _Shift)
_rotr64	unsigned __int64 __cdecl _rotr64(unsigned __int64 _Value, int _Shift)
_rotr8	unsigned char _rotr8(unsigned char _Value, unsigned char _Shift)
_setjmpex	int __cdecl _setjmpex(jmp_buf)
_WriteBarrier	void _WriteBarrier(void)

[Return to top]

Interlocked intrinsics

Interlocked intrinsics are a set of intrinsics that are used to perform atomic read-modify-write operations. Some of them are common to all platforms. They're listed separately here because there are a large number of them, but because their definitions are mostly redundant, it's easier to think about them in general terms. Their names can be used to derive the exact behaviors.

The following table summarizes the ARM support of the non-bittest interlocked intrinsics. Each cell in the table corresponds to a name that is derived by appending the operation name in the left-most cell of the row and the type name in the top-most cell of the column to `_Interlocked`. For example, the cell at the intersection of the `Xor` row and the `8` column corresponds to `_InterlockedXor8` and is fully supported. Most of the supported functions offer these optional suffixes: `_acq`, `_rel`, and `_nf`. The `_acq` suffix indicates an "acquire" semantic and the `_rel` suffix indicates a "release" semantic. The `_nf` or "no fence" suffix is unique to ARM and is discussed in the next section.

OPERATION	8	16	32	64	P
Add	None	None	Full	Full	None
And	Full	Full	Full	Full	None
CompareExchange	Full	Full	Full	Full	Full
Decrement	None	Full	Full	Full	None
Exchange	Partial	Partial	Partial	Partial	Partial
ExchangeAdd	Full	Full	Full	Full	None
Increment	None	Full	Full	Full	None
Or	Full	Full	Full	Full	None

OPERATION	8	16	32	64	P
Xor	Full	Full	Full	Full	None

Key:

- **Full**: supports plain, `_acq`, `_rel`, and `_nf` forms.
- **Partial**: supports plain, `_acq`, and `_nf` forms.
- **None**: Not supported

_nf (no fence) Suffix

The `_nf` or "no fence" suffix indicates that the operation doesn't behave as any kind of memory barrier, in contrast to the other three forms (plain, `_acq`, and `_rel`), which all behave as some kind of barrier. One possible use of the `_nf` forms is to maintain a statistic counter that is updated by multiple threads at the same time but whose value isn't otherwise used while multiple threads are executing.

List of interlocked intrinsics

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedAdd	long _InterlockedAdd(long _volatile *, long)
_InterlockedAdd64	__int64 _InterlockedAdd64(__int64 volatile *, __int64)
_InterlockedAdd64_acq	__int64 _InterlockedAdd64_acq(__int64 volatile *, __int64)
_InterlockedAdd64_nf	__int64 _InterlockedAdd64_nf(__int64 volatile *, __int64)
_InterlockedAdd64_rel	__int64 _InterlockedAdd64_rel(__int64 volatile *, __int64)
_InterlockedAdd_acq	long _InterlockedAdd_acq(long volatile *, long)
_InterlockedAdd_nf	long _InterlockedAdd_nf(long volatile *, long)
_InterlockedAdd_rel	long _InterlockedAdd_rel(long volatile *, long)
_InterlockedAnd	long _InterlockedAnd(long volatile *, long)
_InterlockedAnd16	short _InterlockedAnd16(short volatile *, short)
_InterlockedAnd16_acq	short _InterlockedAnd16_acq(short volatile *, short)
_InterlockedAnd16_nf	short _InterlockedAnd16_nf(short volatile *, short)
_InterlockedAnd16_rel	short _InterlockedAnd16_rel(short volatile *, short)
_InterlockedAnd64	__int64 _InterlockedAnd64(__int64 volatile *, __int64)
_InterlockedAnd64_acq	__int64 _InterlockedAnd64_acq(__int64 volatile *, __int64)
_InterlockedAnd64_nf	__int64 _InterlockedAnd64_nf(__int64 volatile *, __int64)

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedAnd64_rel	__int64 _InterlockedAnd64_rel(__int64 volatile *, __int64)
_InterlockedAnd8	char _InterlockedAnd8(char volatile *, char)
_InterlockedAnd8_acq	char _InterlockedAnd8_acq(char volatile *, char)
_InterlockedAnd8_nf	char _InterlockedAnd8_nf(char volatile *, char)
_InterlockedAnd8_rel	char _InterlockedAnd8_rel(char volatile *, char)
_InterlockedAnd_acq	long _InterlockedAnd_acq(long volatile *, long)
_InterlockedAnd_nf	long _InterlockedAnd_nf(long volatile *, long)
_InterlockedAnd_rel	long _InterlockedAnd_rel(long volatile *, long)
_InterlockedCompareExchange	long __cdecl _InterlockedCompareExchange(long volatile *, long, long)
_InterlockedCompareExchange16	short _InterlockedCompareExchange16(short volatile *, short, short)
_InterlockedCompareExchange16_acq	short _InterlockedCompareExchange16_acq(short volatile *, short, short)
_InterlockedCompareExchange16_nf	short _InterlockedCompareExchange16_nf(short volatile *, short, short)
_InterlockedCompareExchange16_rel	short _InterlockedCompareExchange16_rel(short volatile *, short, short)
_InterlockedCompareExchange64	__int64 _InterlockedCompareExchange64(__int64 volatile *, __int64, __int64)
_InterlockedCompareExchange64_acq	__int64 _InterlockedCompareExchange64_acq(__int64 volatile *, __int64, __int64)
_InterlockedCompareExchange64_nf	__int64 _InterlockedCompareExchange64_nf(__int64 volatile *, __int64, __int64)
_InterlockedCompareExchange64_rel	__int64 _InterlockedCompareExchange64_rel(__int64 volatile *, __int64, __int64)
_InterlockedCompareExchange8	char _InterlockedCompareExchange8(char volatile *, char, char)
_InterlockedCompareExchange8_acq	char _InterlockedCompareExchange8_acq(char volatile *, char, char)
_InterlockedCompareExchange8_nf	char _InterlockedCompareExchange8_nf(char volatile *, char, char)

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedCompareExchange8_rel	char _InterlockedCompareExchange8_rel(char volatile *, char, char)
_InterlockedCompareExchangePointer	void * _InterlockedCompareExchangePointer(void * volatile *, void *, void *)
_InterlockedCompareExchangePointer_acq	void * _InterlockedCompareExchangePointer_acq(void * volatile *, void *, void *)
_InterlockedCompareExchangePointer_nf	void * _InterlockedCompareExchangePointer_nf(void * volatile *, void *, void *)
_InterlockedCompareExchangePointer_rel	void * _InterlockedCompareExchangePointer_rel(void * volatile *, void *, void *)
_InterlockedCompareExchange_acq	long _InterlockedCompareExchange_acq(long volatile *, long, long)
_InterlockedCompareExchange_nf	long _InterlockedCompareExchange_nf(long volatile *, long, long)
_InterlockedCompareExchange_rel	long _InterlockedCompareExchange_rel(long volatile *, long, long)
_InterlockedDecrement	long __cdecl _InterlockedDecrement(long volatile *)
_InterlockedDecrement16	short _InterlockedDecrement16(short volatile *)
_InterlockedDecrement16_acq	short _InterlockedDecrement16_acq(short volatile *)
_InterlockedDecrement16_nf	short _InterlockedDecrement16_nf(short volatile *)
_InterlockedDecrement16_rel	short _InterlockedDecrement16_rel(short volatile *)
_InterlockedDecrement64	__int64 _InterlockedDecrement64(__int64 volatile *)
_InterlockedDecrement64_acq	__int64 _InterlockedDecrement64_acq(__int64 volatile *)
_InterlockedDecrement64_nf	__int64 _InterlockedDecrement64_nf(__int64 volatile *)
_InterlockedDecrement64_rel	__int64 _InterlockedDecrement64_rel(__int64 volatile *)
_InterlockedDecrement_acq	long _InterlockedDecrement_acq(long volatile *)
_InterlockedDecrement_nf	long _InterlockedDecrement_nf(long volatile *)
_InterlockedDecrement_rel	long _InterlockedDecrement_rel(long volatile *)
_InterlockedExchange	long __cdecl _InterlockedExchange(long volatile * _Target, long)
_InterlockedExchange16	short _InterlockedExchange16(short volatile * _Target, short)

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedExchange16_acq	short _InterlockedExchange16_acq(short volatile * _Target, short)
_InterlockedExchange16_nf	short _InterlockedExchange16_nf(short volatile * _Target, short)
_InterlockedExchange64	__int64 _InterlockedExchange64(__int64 volatile * _Target, __int64)
_InterlockedExchange64_acq	__int64 _InterlockedExchange64_acq(__int64 volatile * _Target, __int64)
_InterlockedExchange64_nf	__int64 _InterlockedExchange64_nf(__int64 volatile * _Target, __int64)
_InterlockedExchange8	char _InterlockedExchange8(char volatile * _Target, char)
_InterlockedExchange8_acq	char _InterlockedExchange8_acq(char volatile * _Target, char)
_InterlockedExchange8_nf	char _InterlockedExchange8_nf(char volatile * _Target, char)
_InterlockedExchangeAdd	long __cdecl _InterlockedExchangeAdd(long volatile *, long)
_InterlockedExchangeAdd16	short _InterlockedExchangeAdd16(short volatile *, short)
_InterlockedExchangeAdd16_acq	short _InterlockedExchangeAdd16_acq(short volatile *, short)
_InterlockedExchangeAdd16_nf	short _InterlockedExchangeAdd16_nf(short volatile *, short)
_InterlockedExchangeAdd16_rel	short _InterlockedExchangeAdd16_rel(short volatile *, short)
_InterlockedExchangeAdd64	__int64 _InterlockedExchangeAdd64(__int64 volatile *, __int64)
_InterlockedExchangeAdd64_acq	__int64 _InterlockedExchangeAdd64_acq(__int64 volatile *, __int64)
_InterlockedExchangeAdd64_nf	__int64 _InterlockedExchangeAdd64_nf(__int64 volatile *, __int64)
_InterlockedExchangeAdd64_rel	__int64 _InterlockedExchangeAdd64_rel(__int64 volatile *, __int64)
_InterlockedExchangeAdd8	char _InterlockedExchangeAdd8(char volatile *, char)
_InterlockedExchangeAdd8_acq	char _InterlockedExchangeAdd8_acq(char volatile *, char)
_InterlockedExchangeAdd8_nf	char _InterlockedExchangeAdd8_nf(char volatile *, char)
_InterlockedExchangeAdd8_rel	char _InterlockedExchangeAdd8_rel(char volatile *, char)

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedExchangeAdd_acq	long _InterlockedExchangeAdd_acq(long volatile *, long)
_InterlockedExchangeAdd_nf	long _InterlockedExchangeAdd_nf(long volatile *, long)
_InterlockedExchangeAdd_rel	long _InterlockedExchangeAdd_rel(long volatile *, long)
_InterlockedExchangePointer	void * _InterlockedExchangePointer(void * volatile * _Target, void *)
_InterlockedExchangePointer_acq	void * _InterlockedExchangePointer_acq(void * volatile * _Target, void *)
_InterlockedExchangePointer_nf	void * _InterlockedExchangePointer_nf(void * volatile * _Target, void *)
_InterlockedExchange_acq	long _InterlockedExchange_acq(long volatile * _Target, long)
_InterlockedExchange_nf	long _InterlockedExchange_nf(long volatile * _Target, long)
_InterlockedIncrement	long __cdecl _InterlockedIncrement(long volatile *)
_InterlockedIncrement16	short _InterlockedIncrement16(short volatile *)
_InterlockedIncrement16_acq	short _InterlockedIncrement16_acq(short volatile *)
_InterlockedIncrement16_nf	short _InterlockedIncrement16_nf(short volatile *)
_InterlockedIncrement16_rel	short _InterlockedIncrement16_rel(short volatile *)
_InterlockedIncrement64	__int64 _InterlockedIncrement64(__int64 volatile *)
_InterlockedIncrement64_acq	__int64 _InterlockedIncrement64_acq(__int64 volatile *)
_InterlockedIncrement64_nf	__int64 _InterlockedIncrement64_nf(__int64 volatile *)
_InterlockedIncrement64_rel	__int64 _InterlockedIncrement64_rel(__int64 volatile *)
_InterlockedIncrement_acq	long _InterlockedIncrement_acq(long volatile *)
_InterlockedIncrement_nf	long _InterlockedIncrement_nf(long volatile *)
_InterlockedIncrement_rel	long _InterlockedIncrement_rel(long volatile *)
_InterlockedOr	long _InterlockedOr(long volatile *, long)
_InterlockedOr16	short _InterlockedOr16(short volatile *, short)
_InterlockedOr16_acq	short _InterlockedOr16_acq(short volatile *, short)
_InterlockedOr16_nf	short _InterlockedOr16_nf(short volatile *, short)

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedOr16_rel	short _InterlockedOr16_rel(short volatile *, short)
_InterlockedOr64	__int64 _InterlockedOr64(__int64 volatile *, __int64)
_InterlockedOr64_acq	__int64 _InterlockedOr64_acq(__int64 volatile *, __int64)
_InterlockedOr64_nf	__int64 _InterlockedOr64_nf(__int64 volatile *, __int64)
_InterlockedOr64_rel	__int64 _InterlockedOr64_rel(__int64 volatile *, __int64)
_InterlockedOr8	char _InterlockedOr8(char volatile *, char)
_InterlockedOr8_acq	char _InterlockedOr8_acq(char volatile *, char)
_InterlockedOr8_nf	char _InterlockedOr8_nf(char volatile *, char)
_InterlockedOr8_rel	char _InterlockedOr8_rel(char volatile *, char)
_InterlockedOr_acq	long _InterlockedOr_acq(long volatile *, long)
_InterlockedOr_nf	long _InterlockedOr_nf(long volatile *, long)
_InterlockedOr_rel	long _InterlockedOr_rel(long volatile *, long)
_InterlockedXor	long _InterlockedXor(long volatile *, long)
_InterlockedXor16	short _InterlockedXor16(short volatile *, short)
_InterlockedXor16_acq	short _InterlockedXor16_acq(short volatile *, short)
_InterlockedXor16_nf	short _InterlockedXor16_nf(short volatile *, short)
_InterlockedXor16_rel	short _InterlockedXor16_rel(short volatile *, short)
_InterlockedXor64	__int64 _InterlockedXor64(__int64 volatile *, __int64)
_InterlockedXor64_acq	__int64 _InterlockedXor64_acq(__int64 volatile *, __int64)
_InterlockedXor64_nf	__int64 _InterlockedXor64_nf(__int64 volatile *, __int64)
_InterlockedXor64_rel	__int64 _InterlockedXor64_rel(__int64 volatile *, __int64)
_InterlockedXor8	char _InterlockedXor8(char volatile *, char)
_InterlockedXor8_acq	char _InterlockedXor8_acq(char volatile *, char)
_InterlockedXor8_nf	char _InterlockedXor8_nf(char volatile *, char)
_InterlockedXor8_rel	char _InterlockedXor8_rel(char volatile *, char)

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedXor_acq	long _InterlockedXor_acq(long volatile *, long)
_InterlockedXor_nf	long _InterlockedXor_nf(long volatile *, long)
_InterlockedXor_rel	long _InterlockedXor_rel(long volatile *, long)

[Return to top]

_interlockedbittest intrinsics

The plain interlocked bit test intrinsics are common to all platforms. ARM adds `_acq`, `_rel`, and `_nf` variants, which just modify the barrier semantics of an operation, as described in _nf (no fence) Suffix earlier in this article.

FUNCTION NAME	FUNCTION PROTOTYPE
_interlockedbittestandreset	unsigned char _interlockedbittestandreset(long volatile *, long)
_interlockedbittestandreset_acq	unsigned char _interlockedbittestandreset_acq(long volatile *, long)
_interlockedbittestandreset_nf	unsigned char _interlockedbittestandreset_nf(long volatile *, long)
_interlockedbittestandreset_rel	unsigned char _interlockedbittestandreset_rel(long volatile *, long)
_interlockedbittestandset	unsigned char _interlockedbittestandset(long volatile *, long)
_interlockedbittestandset_acq	unsigned char _interlockedbittestandset_acq(long volatile *, long)
_interlockedbittestandset_nf	unsigned char _interlockedbittestandset_nf(long volatile *, long)
_interlockedbittestandset_rel	unsigned char _interlockedbittestandset_rel(long volatile *, long)

[Return to top]

See also

Compiler intrinsics
ARM64 intrinsics
ARM assembler reference
C++ language reference

ARM64 intrinsics

9/2/2022 • 17 minutes to read • Edit Online

The Microsoft C++ compiler (MSVC) makes the following intrinsics available on the ARM64 architecture. For more information about ARM, see the Architecture and Software Development Tools sections of the ARM Developer Documentation website.

NEON

The NEON vector instruction set extensions for ARM64 provide Single Instruction Multiple Data (SIMD) capabilities. They resemble the ones in the MMX and SSE vector instruction sets that are common to x86 and x64 architecture processors.

NEON intrinsics are supported, as provided in the header file *arm64_neon.h*. The MSVC support for NEON intrinsics resembles that of the ARM64 compiler, which is documented in the ARM NEON Intrinsic Reference on the ARM Infocenter website.

ARM64-specific intrinsics listing

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
__break	BRK	void __break(int)
__addx18byte		void __addx18byte(unsigned long, unsigned char)
__addx18word		void __addx18word(unsigned long, unsigned short)
__addx18dword		void __addx18dword(unsigned long, unsigned long)
__addx18qword		void __addx18qword(unsigned long, unsigned __int64)
__cas8	CASB	unsigned __int8 __cas8(unsigned __int8 volatile* _Target, unsigned __int8 _Comp, unsigned __int8 _Value)
__cas16	CASH	unsigned __int16 __cas16(unsigned __int16 volatile* _Target, unsigned __int16 _Comp, unsigned __int16 _Value)
__cas32	CAS	unsigned __int32 __cas32(unsigned __int32 volatile* _Target, unsigned __int32 _Comp, unsigned __int32 _Value)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
__cas64	CAS	unsigned __int64 __cas64(unsigned __int64 volatile* _Target, unsigned __int64 _Comp, unsigned __int64 _Value)
__casa8	CASAB	unsigned __int8 __casa8(unsigned __int8 volatile* _Target, unsigned __int8 _Comp, unsigned __int8 _Value)
__casa16	CASAH	unsigned __int16 __casa16(unsigned __int16 volatile* _Target, unsigned __int16 _Comp, unsigned __int16 _Value)
__casa32	CASA	unsigned __int32 __casa32(unsigned __int32 volatile* _Target, unsigned __int32 _Comp, unsigned __int32 _Value)
__casa64	CASA	unsigned __int64 __casa64(unsigned __int64 volatile* _Target, unsigned __int64 _Comp, unsigned __int64 _Value)
__casl8	CASLB	unsigned __int8 __casl8(unsigned __int8 volatile* _Target, unsigned __int8 _Comp, unsigned __int8 _Value)
__casl16	CASLH	unsigned __int16 __casl16(unsigned __int16 volatile* _Target, unsigned __int16 _Comp, unsigned __int16 _Value)
__casl32	CASL	unsigned __int32 __casl32(unsigned __int32 volatile* _Target, unsigned __int32 _Comp, unsigned __int32 _Value)
__casl64	CASL	unsigned __int64 __casl64(unsigned __int64 volatile* _Target, unsigned __int64 _Comp, unsigned __int64 _Value)
__casal8	CASALB	unsigned __int8 __casal8(unsigned __int8 volatile* _Target, unsigned __int8 _Comp, unsigned __int8 _Value)
__casal16	CASALH	unsigned __int16 __casal16(unsigned __int16 volatile* _Target, unsigned __int16 _Comp, unsigned __int16 _Value)
__casal32	CASAL	unsigned __int32 __casal32(unsigned __int32 volatile* _Target, unsigned __int32 _Comp, unsigned __int32 _Value)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
__casal64	CASAL	unsigned __int64 __casal64(unsigned __int64 volatile* _Target, unsigned __int64 _Comp, unsigned __int64 _Value)
__crc32b	CRC32B	unsigned __int32 __crc32b(unsigned __int32, unsigned __int32)
__crc32h	CRC32H	unsigned __int32 __crc32h(unsigned __int32, unsigned __int32)
__crc32w	CRC32W	unsigned __int32 __crc32w(unsigned __int32, unsigned __int32)
__crc32d	CRC32X	unsigned __int32 __crc32d(unsigned __int32, unsigned __int64)
__crc32cb	CRC32CB	unsigned __int32 __crc32cb(unsigned __int32, unsigned __int32)
__crc32ch	CRC32CH	unsigned __int32 __crc32ch(unsigned __int32, unsigned __int32)
__crc32cw	CRC32CW	unsigned __int32 __crc32cw(unsigned __int32, unsigned __int32)
__crc32cd	CRC32CX	unsigned __int32 __crc32cd(unsigned __int32, unsigned __int64)
__dmb	DMB	void __dmb(unsigned int `_Type`) Inserts a memory barrier operation into the instruction stream. The parameter `_Type` specifies the kind of restriction that the barrier enforces. For more information about the kinds of restrictions that can be enforced, see Memory barrier restrictions.
__dsb	DSB	void __dsb(unsigned int _Type) Inserts a memory barrier operation into the instruction stream. The parameter `_Type` specifies the kind of restriction that the barrier enforces. For more information about the kinds of restrictions that can be enforced, see Memory barrier restrictions.

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
__isb	ISB	void __isb(unsigned int _Type) Inserts a memory barrier operation into the instruction stream. The parameter `_Type` specifies the kind of restriction that the barrier enforces. For more information about the kinds of restrictions that can be enforced, see Memory barrier restrictions.
__getReg		unsigned __int64 __getReg(int)
__getRegFp		double __getRegFp(int)
__getCallerReg		unsigned __int64 __getCallerReg(int)
__getCallerRegFp		double __getCallerRegFp(int)
__hvc	HVC	unsigned int __hvc(unsigned int, ...)
__hlt	HLT	int __hlt(unsigned int, ...)
__incx18byte		void __incx18byte(unsigned long)
__incx18word		void __incx18word(unsigned long)
__incx18dword		void __incx18dword(unsigned long)
__incx18qword		void __incx18qword(unsigned long)
__iso_volatile_load16		__int16 __iso_volatile_load16(const volatile __int16 *) For more information, see __iso_volatile_load/store intrinsics.
__iso_volatile_load32		__int32 __iso_volatile_load32(const volatile __int32 *) For more information, see __iso_volatile_load/store intrinsics.
__iso_volatile_load64		__int64 __iso_volatile_load64(const volatile __int64 *) For more information, see __iso_volatile_load/store intrinsics.
__iso_volatile_load8		__int8 __iso_volatile_load8(const volatile __int8 *) For more information, see __iso_volatile_load/store intrinsics.

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
__iso_volatile_store16		void __iso_volatile_store16(volatile __int16 *, __int16) For more information, see __iso_volatile_load/store intrinsics.
__iso_volatile_store32		void __iso_volatile_store32(volatile __int32 *, __int32) For more information, see __iso_volatile_load/store intrinsics.
__iso_volatile_store64		void __iso_volatile_store64(volatile __int64 *, __int64) For more information, see __iso_volatile_load/store intrinsics.
__iso_volatile_store8		void __iso_volatile_store8(volatile __int8 *, __int8) For more information, see __iso_volatile_load/store intrinsics.
__ldar8	LDARB	unsigned __int8 __ldar8(unsigned __int8 volatile* _Target)
__ldar16	LDARH	unsigned __int16 __ldar16(unsigned __int16 volatile* _Target)
__ldar32	LDAR	unsigned __int32 __ldar32(unsigned __int32 volatile* _Target)
__ldar64	LDAR	unsigned __int64 __ldar64(unsigned __int64 volatile* _Target)
__ldapr8	LDAPRB	unsigned __int8 __ldapr8(unsigned __int8 volatile* _Target)
__ldapr16	LDAPRH	unsigned __int16 __ldapr16(unsigned __int16 volatile* _Target)
__ldapr32	LDAPR	unsigned __int32 __ldapr32(unsigned __int32 volatile* _Target)
__ldapr64	LDAPR	unsigned __int64 __ldapr64(unsigned __int64 volatile* _Target)
__mulh		__int64 __mulh(__int64, __int64)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
__prefetch	PRFM	void __cdecl __prefetch(const void *) Provides a `PRFM` memory hint with the prefetch operation `PLDL1KEEP` to the system that memory at or near the specified address may be accessed soon. Some systems may choose to optimize for that memory access pattern to increase runtime performance. However, from the C++ language point of view, the function has no observable effect, and may do nothing at all.
__prefetch2	PRFM	void __cdecl __prefetch(const void *, uint8_t prfop) Provides a `PRFM` memory hint with the provided prefetch operation to the system that memory at or near the specified address may be accessed soon. Some systems may choose to optimize for that memory access pattern to increase runtime performance. However, from the C++ language point of view, the function has no observable effect, and may do nothing at all.
__readx18byte		unsigned char __readx18byte(unsigned long)
__readx18word		unsigned short __readx18word(unsigned long)
__readx18dword		unsigned long __readx18dword(unsigned long)
__readx18qword		unsigned __int64 __readx18qword(unsigned long)
__setReg		void __setReg(int, unsigned __int64)
__setRegFp		void __setRegFp(int, double)
__setCallerReg		void __setCallerReg(int, unsigned __int64)
__setCallerRegFp		void __setCallerRegFp(int, double)
__sev	SEV	void __sev(void)
__static_assert		void __static_assert(int, const char *)
__stlr8	STLRB	void __stlr8(unsigned __int8 volatile* _Target, unsigned __int8 _Value)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
__stlr16	STLRH	void __stlr16(unsigned __int16 volatile* _Target, unsigned __int16 _Value)
__stlr32	STLR	void __stlr32(unsigned __int32 volatile* _Target, unsigned __int32 _Value)
__stlr64	STLR	void __stlr64(unsigned __int64 volatile* _Target, unsigned __int64 _Value)
__swp8	SWPB	unsigned __int8 __swp8(unsigned __int8 volatile* _Target, unsigned __int8 _Value)
__swp16	SWPH	unsigned __int16 __swp16(unsigned __int16 volatile* _Target, unsigned __int16 _Value)
__swp32	SWP	unsigned __int32 __swp32(unsigned __int32 volatile* _Target, unsigned __int32 _Value)
__swp64	SWP	unsigned __int64 __swp64(unsigned __int64 volatile* _Target, unsigned __int64 _Value)
__swpa8	SWPAB	unsigned __int8 __swpa8(unsigned __int8 volatile* _Target, unsigned __int8 _Value)
__swpa16	SWPAH	unsigned __int16 __swpa16(unsigned __int16 volatile* _Target, unsigned __int16 _Value)
__swpa32	SWPA	unsigned __int32 __swpa32(unsigned __int32 volatile* _Target, unsigned __int32 _Value)
__swpa64	SWPA	unsigned __int64 __swpa64(unsigned __int64 volatile* _Target, unsigned __int64 _Value)
__swpl8	SWPLB	unsigned __int8 __swpl8(unsigned __int8 volatile* _Target, unsigned __int8 _Value)
__swpl16	SWPLH	unsigned __int16 __swpl16(unsigned __int16 volatile* _Target, unsigned __int16 _Value)
__swpl32	SWPL	unsigned __int32 __swpl32(unsigned __int32 volatile* _Target, unsigned __int32 _Value)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
__swpl64	SWPL	unsigned __int64 __swpl64(unsigned __int64 volatile* _Target, unsigned __int64 _Value)
__swpal8	SWPALB	unsigned __int8 __swpal8(unsigned __int8 volatile* _Target, unsigned __int8 _Value)
__swpal16	SWPALH	unsigned __int16 __swpal16(unsigned __int16 volatile* _Target, unsigned __int16 _Value)
__swpal32	SWPAL	unsigned __int32 __swpal32(unsigned __int32 volatile* _Target, unsigned __int32 _Value)
__swpal64	SWPAL	unsigned __int64 __swpal64(unsigned __int64 volatile* _Target, unsigned __int64 _Value)
__sys	SYS	unsigned int __sys(int, __int64)
__svc	SVC	unsigned int __svc(unsigned int, ...)
__wfe	WFE	void __wfe(void)
__wfi	WFI	void __wfi(void)
__writex18byte		void __writex18byte(unsigned long, unsigned char)
__writex18word		void __writex18word(unsigned long, unsigned short)
__writex18dword		void __writex18dword(unsigned long, unsigned long)
__writex18qword		void __writex18qword(unsigned long, unsigned __int64)
__umulh		unsigned __int64 __umulh(unsigned __int64, unsigned __int64)
_CopyDoubleFromInt64		double _CopyDoubleFromInt64(__int64)
_CopyFloatFromInt32		float _CopyFloatFromInt32(__int32)
_CopyInt32FromFloat		__int32 _CopyInt32FromFloat(float)
_CopyInt64FromDouble		__int64 _CopyInt64FromDouble(double)

FUNCTION NAME	INSTRUCTION	FUNCTION PROTOTYPE
_CountLeadingOnes		unsigned int _CountLeadingOnes(unsigned long)
_CountLeadingOnes64		unsigned int _CountLeadingOnes64(unsigned __int64)
_CountLeadingSigns		unsigned int _CountLeadingSigns(long)
_CountLeadingSigns64		unsigned int _CountLeadingSigns64(__int64)
_CountLeadingZeros		unsigned int _CountLeadingZeros(unsigned long)
_CountLeadingZeros64		unsigned int _CountLeadingZeros64(unsigned __int64)
_CountOneBits		unsigned int _CountOneBits(unsigned long)
_CountOneBits64		unsigned int _CountOneBits64(unsigned __int64)
_ReadStatusReg	MRS	__int64 _ReadStatusReg(int)
_WriteStatusReg	MSR	void _WriteStatusReg(int, __int64)

[Return to top]

Memory barrier restrictions

The intrinsic functions `__dmb` (data memory barrier), `__dsb` (data synchronization barrier), and `__isb` (instruction synchronization barrier) use the following predefined values to specify the memory barrier restriction in terms of the sharing domain and the kind of access that are affected by the operation.

RESTRICTION VALUE	DESCRIPTION
_ARM64_BARRIER_SY	Full system, reads and writes.
_ARM64_BARRIER_ST	Full system, writes only.
_ARM64_BARRIER_LD	Full system, read only.
_ARM64_BARRIER_ISH	Inner sharable, reads and writes.
_ARM64_BARRIER_ISHST	Inner sharable, writes only.
_ARM64_BARRIER_ISHLD	Inner sharable, read only.
_ARM64_BARRIER_NSH	Non-sharable, reads and writes.

RESTRICTION VALUE	DESCRIPTION
_ARM64_BARRIER_NSHST	Non-sharable, writes only.
_ARM64_BARRIER_NSHLD	Non-sharable, read only.
_ARM64_BARRIER_OSH	Outer sharable, reads and writes.
_ARM64_BARRIER_OSHST	Outer sharable, writes only.
_ARM64_BARRIER_OSHLD	Outer sharable, read only.

For the `__isb` intrinsic, the only restriction that is currently valid is _ARM64_BARRIER_SY; all other values are reserved by the architecture.

__iso_volatile_load/store intrinsics

These intrinsic functions explicitly perform loads and stores that aren't subject to compiler optimizations.

```
__int16 __iso_volatile_load16(const volatile __int16 * Location);
__int32 __iso_volatile_load32(const volatile __int32 * Location);
__int64 __iso_volatile_load64(const volatile __int64 * Location);
__int8 __iso_volatile_load8(const volatile __int8 * Location);

void __iso_volatile_store16(volatile __int16 * Location, __int16 Value);
void __iso_volatile_store32(volatile __int32 * Location, __int32 Value);
void __iso_volatile_store64(volatile __int64 * Location, __int64 Value);
void __iso_volatile_store8(volatile __int8 * Location, __int8 Value);
```

Parameters

Location
The address of a memory location to read from or write to.

Value
The value to write to the specified memory location (store intrinsics only).

Return value (load intrinsics only)

The value of the memory location that is specified by *Location*.

Remarks

You can use the `__iso_volatile_load8/16/32/64` and `__iso_volatile_store8/16/32/64` intrinsics to explicitly perform memory accesses that aren't subject to compiler optimizations. The compiler can't remove, synthetize, or change the relative order of these operations. However, it doesn't generate implicit hardware memory barriers. Therefore, the hardware may still reorder the observable memory accesses across multiple threads. More precisely, these intrinsics are equivalent to the following expressions as compiled under **/volatile:iso**.

```
int a = __iso_volatile_load32(p);    // equivalent to: int a = *(const volatile __int32*)p;
__iso_volatile_store32(p, a);        // equivalent to: *(volatile __int32*)p = a;
```

Notice that the intrinsics take volatile pointers to accommodate volatile variables. However, there's no requirement or recommendation to use volatile pointers as arguments. The semantics of these operations are exactly the same if a regular, non-volatile type is used.

For more information about the **/volatile:iso** command-line argument, see /volatile (volatile keyword interpretation).

ARM64 support for intrinsics from other architectures

The following table lists intrinsics from other architectures that are supported on ARM64 platforms. Where the behavior of an intrinsic on ARM64 differs from its behavior on other hardware architectures, additional details are noted.

FUNCTION NAME	FUNCTION PROTOTYPE
__assume	void __assume(int)
__code_seg	void __code_seg(const char *)
__debugbreak	void __cdecl __debugbreak(void)
__fastfail	__declspec(noreturn) void __fastfail(unsigned int)
__nop	void __nop(void)
__yield	void __yield(void) **Note:** On ARM64 platforms, this function generates the YIELD instruction. This instruction indicates that the thread is performing a task that may be temporarily suspended from execution—for example, a spinlock—without adversely affecting the program. It enables the CPU to execute other tasks during execution cycles that would otherwise be wasted.
_AddressOfReturnAddress	void * _AddressOfReturnAddress(void)
_BitScanForward	unsigned char _BitScanForward(unsigned long * _Index, unsigned long _Mask)
_BitScanForward64	unsigned char _BitScanForward64(unsigned long * _Index, unsigned __int64 _Mask)
_BitScanReverse	unsigned char _BitScanReverse(unsigned long * _Index, unsigned long _Mask)
_BitScanReverse64	unsigned char _BitScanReverse64(unsigned long * _Index, unsigned __int64 _Mask)
_bittest	unsigned char _bittest(long const *, long)
_bittest64	unsigned char _bittest64(__int64 const *, __int64)
_bittestandcomplement	unsigned char _bittestandcomplement(long *, long)
_bittestandcomplement64	unsigned char _bittestandcomplement64(__int64 *, __int64)
_bittestandreset	unsigned char _bittestandreset(long *, long)
_bittestandreset64	unsigned char _bittestandreset64(__int64 *, __int64)
_bittestandset	unsigned char _bittestandset(long *, long)

FUNCTION NAME	FUNCTION PROTOTYPE
_bittestandset64	unsigned char _bittestandset64(__int64 *, __int64)
_byteswap_uint64	unsigned __int64 __cdecl _byteswap_uint64(unsigned __int64)
_byteswap_ulong	unsigned long __cdecl _byteswap_ulong(unsigned long)
_byteswap_ushort	unsigned short __cdecl _byteswap_ushort(unsigned short)
_disable	void __cdecl _disable(void) **Note:** On ARM64 platforms, this function generates the instruction `MSR DAIFCLR,#2`; it's only available as an intrinsic.
_enable	void __cdecl _enable(void) **Note:** On ARM64 platforms, this function generates the instruction `MSR DAIFSET,#2`; it's only available as an intrinsic.
_lrotl	unsigned long __cdecl _lrotl(unsigned long, int)
_lrotr	unsigned long __cdecl _lrotr(unsigned long, int)
_ReadBarrier	void _ReadBarrier(void)
_ReadWriteBarrier	void _ReadWriteBarrier(void)
_ReturnAddress	void * _ReturnAddress(void)
_rotl	unsigned int __cdecl _rotl(unsigned int _Value, int _Shift)
_rotl16	unsigned short _rotl16(unsigned short _Value, unsigned char _Shift)
_rotl64	unsigned __int64 __cdecl _rotl64(unsigned __int64 _Value, int _Shift)
_rotl8	unsigned char _rotl8(unsigned char _Value, unsigned char _Shift)
_rotr	unsigned int __cdecl _rotr(unsigned int _Value, int _Shift)
_rotr16	unsigned short _rotr16(unsigned short _Value, unsigned char _Shift)
_rotr64	unsigned __int64 __cdecl _rotr64(unsigned __int64 _Value, int _Shift)
_rotr8	unsigned char _rotr8(unsigned char _Value, unsigned char _Shift)
_setjmpex	int __cdecl _setjmpex(jmp_buf)
_WriteBarrier	void _WriteBarrier(void)

[Return to top]

Interlocked intrinsics

Interlocked intrinsics are a set of intrinsics that are used to perform atomic read-modify-write operations. Some of them are common to all platforms. They're listed separately here because there are a large number of them. Because their definitions are mostly redundant, it's easier to think about them in general terms. Their names can be used to derive the exact behaviors.

The following table summarizes the ARM64 support of the non-bittest interlocked intrinsics. Each cell in the table corresponds to a name that is derived by appending the operation name in the left-most cell of the row and the type name in the top-most cell of the column to `_Interlocked`. For example, the cell at the intersection of the `Xor` row and the `8` column corresponds to `_InterlockedXor8` and is fully supported. Most of the supported functions offer these optional suffixes: `_acq`, `_rel`, and `_nf`. The `_acq` suffix indicates an "acquire" semantic and the `_rel` suffix indicates a "release" semantic. The `_nf` or "no fence" suffix is unique to ARM and ARM64 and is discussed in the next section.

OPERATION	8	16	32	64	128	P
Add	None	None	Full	Full	None	None
And	Full	Full	Full	Full	None	None
CompareExchange	Full	Full	Full	Full	Full	Full
Decrement	None	Full	Full	Full	None	None
Exchange	Full	Full	Full	Full	None	Full
ExchangeAdd	Full	Full	Full	Full	None	None
Increment	None	Full	Full	Full	None	None
Or	Full	Full	Full	Full	None	None
Xor	Full	Full	Full	Full	None	None

Key:

- **Full**: supports plain, `_acq`, `_rel`, and `_nf` forms.
- **None**: Not supported

_nf (no fence) suffix

The `_nf` or "no fence" suffix indicates that the operation doesn't behave as any kind of memory barrier, in contrast to the other three forms (plain, `_acq`, and `_rel`), which all behave as some kind of barrier. One possible use of the `_nf` forms is to maintain a statistic counter that is updated by multiple threads at the same time but whose value isn't otherwise used while multiple threads are executing.

List of interlocked intrinsics

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedAdd	long _InterlockedAdd(long _volatile *, long)
_InterlockedAdd64	__int64 _InterlockedAdd64(__int64 volatile *, __int64)
_InterlockedAdd64_acq	__int64 _InterlockedAdd64_acq(__int64 volatile *, __int64)
_InterlockedAdd64_nf	__int64 _InterlockedAdd64_nf(__int64 volatile *, __int64)
_InterlockedAdd64_rel	__int64 _InterlockedAdd64_rel(__int64 volatile *, __int64)
_InterlockedAdd_acq	long _InterlockedAdd_acq(long volatile *, long)
_InterlockedAdd_nf	long _InterlockedAdd_nf(long volatile *, long)
_InterlockedAdd_rel	long _InterlockedAdd_rel(long volatile *, long)
_InterlockedAnd	long _InterlockedAnd(long volatile *, long)
_InterlockedAnd16	short _InterlockedAnd16(short volatile *, short)
_InterlockedAnd16_acq	short _InterlockedAnd16_acq(short volatile *, short)
_InterlockedAnd16_nf	short _InterlockedAnd16_nf(short volatile *, short)
_InterlockedAnd16_rel	short _InterlockedAnd16_rel(short volatile *, short)
_InterlockedAnd64	__int64 _InterlockedAnd64(__int64 volatile *, __int64)
_InterlockedAnd64_acq	__int64 _InterlockedAnd64_acq(__int64 volatile *, __int64)
_InterlockedAnd64_nf	__int64 _InterlockedAnd64_nf(__int64 volatile *, __int64)
_InterlockedAnd64_rel	__int64 _InterlockedAnd64_rel(__int64 volatile *, __int64)
_InterlockedAnd8	char _InterlockedAnd8(char volatile *, char)
_InterlockedAnd8_acq	char _InterlockedAnd8_acq(char volatile *, char)
_InterlockedAnd8_nf	char _InterlockedAnd8_nf(char volatile *, char)
_InterlockedAnd8_rel	char _InterlockedAnd8_rel(char volatile *, char)
_InterlockedAnd_acq	long _InterlockedAnd_acq(long volatile *, long)
_InterlockedAnd_nf	long _InterlockedAnd_nf(long volatile *, long)
_InterlockedAnd_rel	long _InterlockedAnd_rel(long volatile *, long)
_InterlockedCompareExchange	long __cdecl _InterlockedCompareExchange(long volatile *, long, long)

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedCompareExchange_acq	long _InterlockedCompareExchange_acq(long volatile *, long, long)
_InterlockedCompareExchange_nf	long _InterlockedCompareExchange_nf(long volatile *, long, long)
_InterlockedCompareExchange_rel	long _InterlockedCompareExchange_rel(long volatile *, long, long)
_InterlockedCompareExchange16	short _InterlockedCompareExchange16(short volatile *, short, short)
_InterlockedCompareExchange16_acq	short _InterlockedCompareExchange16_acq(short volatile *, short, short)
_InterlockedCompareExchange16_nf	short _InterlockedCompareExchange16_nf(short volatile *, short, short)
_InterlockedCompareExchange16_rel	short _InterlockedCompareExchange16_rel(short volatile *, short, short)
_InterlockedCompareExchange64	__int64 _InterlockedCompareExchange64(__int64 volatile *, __int64, __int64)
_InterlockedCompareExchange64_acq	__int64 _InterlockedCompareExchange64_acq(__int64 volatile *, __int64, __int64)
_InterlockedCompareExchange64_nf	__int64 _InterlockedCompareExchange64_nf(__int64 volatile *, __int64, __int64)
_InterlockedCompareExchange64_rel	__int64 _InterlockedCompareExchange64_rel(__int64 volatile *, __int64, __int64)
_InterlockedCompareExchange8	char _InterlockedCompareExchange8(char volatile *, char, char)
_InterlockedCompareExchange8_acq	char _InterlockedCompareExchange8_acq(char volatile *, char, char)
_InterlockedCompareExchange8_nf	char _InterlockedCompareExchange8_nf(char volatile *, char, char)
_InterlockedCompareExchange8_rel	char _InterlockedCompareExchange8_rel(char volatile *, char, char)
_InterlockedCompareExchangePointer	void * _InterlockedCompareExchangePointer(void * volatile *, void *, void *)
_InterlockedCompareExchangePointer_acq	void * _InterlockedCompareExchangePointer_acq(void * volatile *, void *, void *)
_InterlockedCompareExchangePointer_nf	void * _InterlockedCompareExchangePointer_nf(void * volatile *, void *, void *)

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedCompareExchangePointer_rel	void * _InterlockedCompareExchangePointer_rel(void * volatile *, void *, void *)
_InterlockedCompareExchange128	unsigned char _InterlockedCompareExchange128(__int64 volatile * _Destination, __int64 _ExchangeHigh, __int64 _ExchangeLow, __int64 * _ComparandResult)
_InterlockedCompareExchange128_acq	unsigned char _InterlockedCompareExchange128_acq(__int64 volatile * _Destination, __int64 _ExchangeHigh, __int64 _ExchangeLow, __int64 * _ComparandResult)
_InterlockedCompareExchange128_nf	unsigned char _InterlockedCompareExchange128_nf(__int64 volatile * _Destination, __int64 _ExchangeHigh, __int64 _ExchangeLow, __int64 * _ComparandResult)
_InterlockedCompareExchange128_rel	unsigned char _InterlockedCompareExchange128_rel(__int64 volatile * _Destination, __int64 _ExchangeHigh, __int64 _ExchangeLow, __int64 * _ComparandResult)
_InterlockedDecrement	long __cdecl _InterlockedDecrement(long volatile *)
_InterlockedDecrement16	short _InterlockedDecrement16(short volatile *)
_InterlockedDecrement16_acq	short _InterlockedDecrement16_acq(short volatile *)
_InterlockedDecrement16_nf	short _InterlockedDecrement16_nf(short volatile *)
_InterlockedDecrement16_rel	short _InterlockedDecrement16_rel(short volatile *)
_InterlockedDecrement64	__int64 _InterlockedDecrement64(__int64 volatile *)
_InterlockedDecrement64_acq	__int64 _InterlockedDecrement64_acq(__int64 volatile *)
_InterlockedDecrement64_nf	__int64 _InterlockedDecrement64_nf(__int64 volatile *)
_InterlockedDecrement64_rel	__int64 _InterlockedDecrement64_rel(__int64 volatile *)
_InterlockedDecrement_acq	long _InterlockedDecrement_acq(long volatile *)
_InterlockedDecrement_nf	long _InterlockedDecrement_nf(long volatile *)
_InterlockedDecrement_rel	long _InterlockedDecrement_rel(long volatile *)
_InterlockedExchange	long __cdecl _InterlockedExchange(long volatile * _Target, long)
_InterlockedExchange_acq	long _InterlockedExchange_acq(long volatile * _Target, long)
_InterlockedExchange_nf	long _InterlockedExchange_nf(long volatile * _Target, long)
_InterlockedExchange_rel	long _InterlockedExchange_rel(long volatile * _Target, long)

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedExchange16	short _InterlockedExchange16(short volatile * _Target, short)
_InterlockedExchange16_acq	short _InterlockedExchange16_acq(short volatile * _Target, short)
_InterlockedExchange16_nf	short _InterlockedExchange16_nf(short volatile * _Target, short)
_InterlockedExchange16_rel	short _InterlockedExchange16_rel(short volatile * _Target, short)
_InterlockedExchange64	__int64 _InterlockedExchange64(__int64 volatile * _Target, __int64)
_InterlockedExchange64_acq	__int64 _InterlockedExchange64_acq(__int64 volatile * _Target, __int64)
_InterlockedExchange64_nf	__int64 _InterlockedExchange64_nf(__int64 volatile * _Target, __int64)
_InterlockedExchange64_rel	__int64 _InterlockedExchange64_rel(__int64 volatile * _Target, __int64)
_InterlockedExchange8	char _InterlockedExchange8(char volatile * _Target, char)
_InterlockedExchange8_acq	char _InterlockedExchange8_acq(char volatile * _Target, char)
_InterlockedExchange8_nf	char _InterlockedExchange8_nf(char volatile * _Target, char)
_InterlockedExchange8_rel	char _InterlockedExchange8_rel(char volatile * _Target, char)
_InterlockedExchangeAdd	long __cdecl _InterlockedExchangeAdd(long volatile *, long)
_InterlockedExchangeAdd16	short _InterlockedExchangeAdd16(short volatile *, short)
_InterlockedExchangeAdd16_acq	short _InterlockedExchangeAdd16_acq(short volatile *, short)
_InterlockedExchangeAdd16_nf	short _InterlockedExchangeAdd16_nf(short volatile *, short)
_InterlockedExchangeAdd16_rel	short _InterlockedExchangeAdd16_rel(short volatile *, short)
_InterlockedExchangeAdd64	__int64 _InterlockedExchangeAdd64(__int64 volatile *, __int64)
_InterlockedExchangeAdd64_acq	__int64 _InterlockedExchangeAdd64_acq(__int64 volatile *, __int64)
_InterlockedExchangeAdd64_nf	__int64 _InterlockedExchangeAdd64_nf(__int64 volatile *, __int64)

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedExchangeAdd64_rel	__int64 _InterlockedExchangeAdd64_rel(__int64 volatile *, __int64)
_InterlockedExchangeAdd8	char _InterlockedExchangeAdd8(char volatile *, char)
_InterlockedExchangeAdd8_acq	char _InterlockedExchangeAdd8_acq(char volatile *, char)
_InterlockedExchangeAdd8_nf	char _InterlockedExchangeAdd8_nf(char volatile *, char)
_InterlockedExchangeAdd8_rel	char _InterlockedExchangeAdd8_rel(char volatile *, char)
_InterlockedExchangeAdd_acq	long _InterlockedExchangeAdd_acq(long volatile *, long)
_InterlockedExchangeAdd_nf	long _InterlockedExchangeAdd_nf(long volatile *, long)
_InterlockedExchangeAdd_rel	long _InterlockedExchangeAdd_rel(long volatile *, long)
_InterlockedExchangePointer	void * _InterlockedExchangePointer(void * volatile * _Target, void *)
_InterlockedExchangePointer_acq	void * _InterlockedExchangePointer_acq(void * volatile * _Target, void *)
_InterlockedExchangePointer_nf	void * _InterlockedExchangePointer_nf(void * volatile * _Target, void *)
_InterlockedExchangePointer_rel	void * _InterlockedExchangePointer_rel(void * volatile * _Target, void *)
_InterlockedIncrement	long __cdecl _InterlockedIncrement(long volatile *)
_InterlockedIncrement16	short _InterlockedIncrement16(short volatile *)
_InterlockedIncrement16_acq	short _InterlockedIncrement16_acq(short volatile *)
_InterlockedIncrement16_nf	short _InterlockedIncrement16_nf(short volatile *)
_InterlockedIncrement16_rel	short _InterlockedIncrement16_rel(short volatile *)
_InterlockedIncrement64	__int64 _InterlockedIncrement64(__int64 volatile *)
_InterlockedIncrement64_acq	__int64 _InterlockedIncrement64_acq(__int64 volatile *)
_InterlockedIncrement64_nf	__int64 _InterlockedIncrement64_nf(__int64 volatile *)
_InterlockedIncrement64_rel	__int64 _InterlockedIncrement64_rel(__int64 volatile *)
_InterlockedIncrement_acq	long _InterlockedIncrement_acq(long volatile *)
_InterlockedIncrement_nf	long _InterlockedIncrement_nf(long volatile *)

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedIncrement_rel	long _InterlockedIncrement_rel(long volatile *)
_InterlockedOr	long _InterlockedOr(long volatile *, long)
_InterlockedOr16	short _InterlockedOr16(short volatile *, short)
_InterlockedOr16_acq	short _InterlockedOr16_acq(short volatile *, short)
_InterlockedOr16_nf	short _InterlockedOr16_nf(short volatile *, short)
_InterlockedOr16_rel	short _InterlockedOr16_rel(short volatile *, short)
_InterlockedOr64	__int64 _InterlockedOr64(__int64 volatile *, __int64)
_InterlockedOr64_acq	__int64 _InterlockedOr64_acq(__int64 volatile *, __int64)
_InterlockedOr64_nf	__int64 _InterlockedOr64_nf(__int64 volatile *, __int64)
_InterlockedOr64_rel	__int64 _InterlockedOr64_rel(__int64 volatile *, __int64)
_InterlockedOr8	char _InterlockedOr8(char volatile *, char)
_InterlockedOr8_acq	char _InterlockedOr8_acq(char volatile *, char)
_InterlockedOr8_nf	char _InterlockedOr8_nf(char volatile *, char)
_InterlockedOr8_rel	char _InterlockedOr8_rel(char volatile *, char)
_InterlockedOr_acq	long _InterlockedOr_acq(long volatile *, long)
_InterlockedOr_nf	long _InterlockedOr_nf(long volatile *, long)
_InterlockedOr_rel	long _InterlockedOr_rel(long volatile *, long)
_InterlockedXor	long _InterlockedXor(long volatile *, long)
_InterlockedXor16	short _InterlockedXor16(short volatile *, short)
_InterlockedXor16_acq	short _InterlockedXor16_acq(short volatile *, short)
_InterlockedXor16_nf	short _InterlockedXor16_nf(short volatile *, short)
_InterlockedXor16_rel	short _InterlockedXor16_rel(short volatile *, short)
_InterlockedXor64	__int64 _InterlockedXor64(__int64 volatile *, __int64)
_InterlockedXor64_acq	__int64 _InterlockedXor64_acq(__int64 volatile *, __int64)
_InterlockedXor64_nf	__int64 _InterlockedXor64_nf(__int64 volatile *, __int64)

FUNCTION NAME	FUNCTION PROTOTYPE
_InterlockedXor64_rel	__int64 _InterlockedXor64_rel(__int64 volatile *, __int64)
_InterlockedXor8	char _InterlockedXor8(char volatile *, char)
_InterlockedXor8_acq	char _InterlockedXor8_acq(char volatile *, char)
_InterlockedXor8_nf	char _InterlockedXor8_nf(char volatile *, char)
_InterlockedXor8_rel	char _InterlockedXor8_rel(char volatile *, char)
_InterlockedXor_acq	long _InterlockedXor_acq(long volatile *, long)
_InterlockedXor_nf	long _InterlockedXor_nf(long volatile *, long)
_InterlockedXor_rel	long _InterlockedXor_rel(long volatile *, long)

[Return to top]

_interlockedbittest intrinsics

The plain interlocked bit test intrinsics are common to all platforms. ARM64 adds `_acq`, `_rel`, and `_nf` variants, which just modify the barrier semantics of an operation, as described in _nf (no fence) Suffix earlier in this article.

FUNCTION NAME	FUNCTION PROTOTYPE
_interlockedbittestandreset	unsigned char _interlockedbittestandreset(long volatile *, long)
_interlockedbittestandreset_acq	unsigned char _interlockedbittestandreset_acq(long volatile *, long)
_interlockedbittestandreset_nf	unsigned char _interlockedbittestandreset_nf(long volatile *, long)
_interlockedbittestandreset_rel	unsigned char _interlockedbittestandreset_rel(long volatile *, long)
_interlockedbittestandreset64	unsigned char _interlockedbittestandreset64(__int64 volatile *, __int64)
_interlockedbittestandreset64_acq	unsigned char _interlockedbittestandreset64_acq(__int64 volatile *, __int64)
_interlockedbittestandreset64_nf	unsigned char _interlockedbittestandreset64_nf(__int64 volatile *, __int64)
_interlockedbittestandreset64_rel	unsigned char _interlockedbittestandreset64_rel(__int64 volatile *, __int64)
_interlockedbittestandset	unsigned char _interlockedbittestandset(long volatile *, long)

FUNCTION NAME	FUNCTION PROTOTYPE
_interlockedbittestandset_acq	unsigned char _interlockedbittestandset_acq(long volatile *, long)
_interlockedbittestandset_nf	unsigned char _interlockedbittestandset_nf(long volatile *, long)
_interlockedbittestandset_rel	unsigned char _interlockedbittestandset_rel(long volatile *, long)
_interlockedbittestandset64	unsigned char _interlockedbittestandset64(__int64 volatile *, __int64)
_interlockedbittestandset64_acq	unsigned char _interlockedbittestandset64_acq(__int64 volatile *, __int64)
_interlockedbittestandset64_nf	unsigned char _interlockedbittestandset64_nf(__int64 volatile *, __int64)
_interlockedbittestandset64_rel	unsigned char _interlockedbittestandset64_rel(__int64 volatile *, __int64)

[Return to top]

See also

Compiler intrinsics
ARM intrinsics
ARM assembler reference
C++ language reference

x86 intrinsics list

9/2/2022 • 43 minutes to read • Edit Online

This document lists intrinsics that the Microsoft C/C++ compiler supports when x86 is targeted.

For information about individual intrinsics, see these resources, as appropriate for the processor you're targeting:

- The header file. Many intrinsics are documented in comments in the header file.
- Intel Intrinsics Guide. Use the search box to find specific intrinsics.
- Intel 64 and IA-32 Architectures Software Developer Manuals
- Intel Architecture Instruction Set Extensions Programming Reference
- Introduction to Intel Advanced Vector Extensions
- AMD Developer Guides, Manuals & ISA Documents

x86 intrinsics

The following table lists the intrinsics available on x86 processors. The Technology column lists required instruction-set support. Use the `__cpuid` intrinsic to determine instruction-set support at run time. If two entries are in one row, they represent different entry points for the same intrinsic. [Macro] indicates the prototype is a macro. The header required for the function prototype is listed in the Header column. The `intrin.h` header includes both `immintrin.h` and `ammintrin.h` for simplicity.

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_addcarry_u16`		intrin.h	`unsigned char _addcarry_u16(unsigned char, unsigned short, unsigned short, unsigned short *);`
`_addcarry_u32`		intrin.h	`unsigned char _addcarry_u32(unsigned char, unsigned int, unsigned int, unsigned int *);`
`_addcarry_u8`		intrin.h	`unsigned char _addcarry_u8(unsigned char, unsigned char, unsigned char, unsigned char *);`
`_addcarryx_u32`	ADX	immintrin.h	`unsigned char _addcarryx_u32(unsigned char, unsigned int, unsigned int, unsigned int *);`
`__addfsbyte`		intrin.h	`void __addfsbyte(unsigned long, unsigned char);`
`__addfsdword`		intrin.h	`void __addfsdword(unsigned long, unsigned long);`
`__addfsword`		intrin.h	`void __addfsword(unsigned long, unsigned short);`
`_AddressOfReturnAddress`		intrin.h	`void * _AddressOfReturnAddress(void);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_andn_u32`	BMI	ammintrin.h	`unsigned int _andn_u32(unsigned int, unsigned int);`
`_bextr_u32`	BMI	ammintrin.h, immintrin.h	`unsigned int _bextr_u32(unsigned int, unsigned int, unsigned int);`
`_bextri_u32`	ABM	ammintrin.h	`unsigned int _bextri_u32(unsigned int, unsigned int);`
`_BitScanForward`		intrin.h	`unsigned char _BitScanForward(unsigned long*, unsigned long);`
`_BitScanReverse`		intrin.h	`unsigned char _BitScanReverse(unsigned long*, unsigned long);`
`_bittest`		intrin.h	`unsigned char _bittest(long const *, long);`
`_bittestandcomplement`		intrin.h	`unsigned char _bittestandcomplement(long *, long);`
`_bittestandreset`		intrin.h	`unsigned char _bittestandreset(long *, long);`
`_bittestandset`		intrin.h	`unsigned char _bittestandset(long *, long);`
`_blcfill_u32`	ABM	ammintrin.h	`unsigned int _blcfill_u32(unsigned int);`
`_blci_u32`	ABM	ammintrin.h	`unsigned int _blci_u32(unsigned int);`
`_blcic_u32`	ABM	ammintrin.h	`unsigned int _blcic_u32(unsigned int);`
`_blcmsk_u32`	ABM	ammintrin.h	`unsigned int _blcmsk_u32(unsigned int);`
`_blcs_u32`	ABM	ammintrin.h	`unsigned int _blcs_u32(unsigned int);`
`_blsfill_u32`	ABM	ammintrin.h	`unsigned int _blsfill_u32(unsigned int);`
`_blsi_u32`	BMI	ammintrin.h, immintrin.h	`unsigned int _blsi_u32(unsigned int);`
`_blsic_u32`	ABM	ammintrin.h	`unsigned int _blsic_u32(unsigned int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_blsmsk_u32`	BMI	ammintrin.h, immintrin.h	`unsigned int _blsmsk_u32(unsigned int);`
`_blsr_u32`	BMI	ammintrin.h, immintrin.h	`unsigned int _blsr_u32(unsigned int);`
`_bzhi_u32`	BMI	immintrin.h	`unsigned int _bzhi_u32(unsigned int, unsigned int);`
`_castf32_u32`		immintrin.h	`unsigned __int32 _castf32_u32 (float);`
`_castf64_u64`		immintrin.h	`unsigned __int64 _castf64_u64 (double);`
`_castu32_f32`		immintrin.h	`float _castu32_f32 (unsigned __int32);`
`_castu64_f64`		immintrin.h	`double _castu64_f64 (unsigned __int64 a);`
`_clac`	SMAP	intrin.h	`void _clac(void);`
`__cpuid`		intrin.h	`void __cpuid(int *, int);`
`__cpuidex`		intrin.h	`void __cpuidex(int *, int, int);`
`__debugbreak`		intrin.h	`void __debugbreak(void);`
`_disable`		intrin.h	`void _disable(void);`
`_div64`		intrin.h	`int _div64(__int64, int, int *);`
`__emul`		intrin.h	`__int64 [pascal/cdecl] __emul(int, int);`
`__emulu`		intrin.h	`unsigned __int64 [pascal/cdecl]__emulu(unsigned int, unsigned int);`
`_enable`		intrin.h	`void _enable(void);`
`__fastfail`		intrin.h	`void __fastfail(unsigned int);`
`_fxrstor`	FXSR	immintrin.h	`void _fxrstor(void const*);`
`_fxsave`	FXSR	immintrin.h	`void _fxsave(void*);`
`__getcallerseflags`		intrin.h	`(unsigned int __getcallerseflags());`
`__halt`		intrin.h	`void __halt(void);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__inbyte`		intrin.h	`unsigned char __inbyte(unsigned short);`
`__inbytestring`		intrin.h	`void __inbytestring(unsigned short, unsigned char *, unsigned long);`
`__incfsbyte`		intrin.h	`void __incfsbyte(unsigned long);`
`__incfsdword`		intrin.h	`void __incfsdword(unsigned long);`
`__incfsword`		intrin.h	`void __incfsword(unsigned long);`
`__indword`		intrin.h	`unsigned long __indword(unsigned short);`
`__indwordstring`		intrin.h	`void __indwordstring(unsigned short, unsigned long *, unsigned long);`
`__int2c`		intrin.h	`void __int2c(void);`
`_InterlockedAddLargeStatistic`		intrin.h	`long _InterlockedAddLargeStatistic(__int64 volatile *, long);`
`_InterlockedAnd`		intrin.h	`long _InterlockedAnd(long volatile *, long);`
`_InterlockedAnd_HLEAcquire`	HLE	immintrin.h	`long _InterlockedAnd_HLEAcquire(long volatile *, long);`
`_InterlockedAnd_HLERelease`	HLE	immintrin.h	`long _InterlockedAnd_HLERelease(long volatile *, long);`
`_InterlockedAnd16`		intrin.h	`short _InterlockedAnd16(short volatile *, short);`
`_InterlockedAnd8`		intrin.h	`char _InterlockedAnd8(char volatile *, char);`
`_interlockedbittestandreset`		intrin.h	`unsigned char _interlockedbittestandreset(long *, long);`
`_interlockedbittestandreset_HLEAcquire`	HLE	immintrin.h	`unsigned char _interlockedbittestandreset_HLEAcquire(lon *, long);`
`_interlockedbittestandreset_HLERelease`	HLE	immintrin.h	`unsigned char _interlockedbittestandreset_HLERelease(lon *, long);`
`_interlockedbittestandset`		intrin.h	`unsigned char _interlockedbittestandset(long *, long);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_interlockedbittestandset_HLEAcquire`	HLE	immintrin.h	`unsigned char _interlockedbittestandset_HLEAcquire(long *, long);`
`_interlockedbittestandset_HLERelease`	HLE	immintrin.h	`unsigned char _interlockedbittestandset_HLERelease(long *, long);`
`_InterlockedCompareExchange`		intrin.h	`long _InterlockedCompareExchange (long volatile *, long, long);`
`_InterlockedCompareExchange_HLEAcquire`	HLE	immintrin.h	`long _InterlockedCompareExchange_HLEAcquire(lon volatile *, long, long);`
`_InterlockedCompareExchange_HLERelease`	HLE	immintrin.h	`long _InterlockedCompareExchange_HLERelease(lon volatile *, long, long);`
`_InterlockedCompareExchange16`		intrin.h	`short _InterlockedCompareExchange16(short volatile *, short, short);`
`_InterlockedCompareExchange64`		intrin.h	`__int64 _InterlockedCompareExchange64(__int64 volatile *, __int64, __int64);`
`_InterlockedCompareExchange64_HLEAcquire`	HLE	immintrin.h	`__int64 _InterlockedCompareExchange64_HLEAcquire(_ volatile *, __int64, __int64);`
`_InterlockedCompareExchange64_HLERelease`	HLE	immintrin.h	`__int64 _InterlockedCompareExchange64_HLERelease(_ volatile *, __int64, __int64);`
`_InterlockedCompareExchange8`		intrin.h	`char _InterlockedCompareExchange8(char volatile *, char, char);`
`_InterlockedCompareExchangePointer`		intrin.h	`void *_InterlockedCompareExchangePointer (void *volatile *, void *, void *);`
`_InterlockedCompareExchangePointer_HLEAcquire`	HLE	immintrin.h	`void *_InterlockedCompareExchangePointer_HLEAcqu *volatile *, void *, void *);`
`_InterlockedCompareExchangePointer_HLERelease`	HLE	immintrin.h	`void *_InterlockedCompareExchangePointer_HLERel *volatile *, void *, void *);`
`_InterlockedDecrement`		intrin.h	`long _InterlockedDecrement(long volatile *);`
`_InterlockedDecrement16`		intrin.h	`short _InterlockedDecrement16(short volatile *);`
`_InterlockedExchange`		intrin.h	`long _InterlockedExchange(long volatile *, long);`
`_InterlockedExchange_HLEAcquire`	HLE	immintrin.h	`long _InterlockedExchange_HLEAcquire(long volatile *, long);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_InterlockedExchange_HLERelease`	HLE	immintrin.h	`long _InterlockedExchange_HLERelease(long volatile *, long);`
`_InterlockedExchange16`		intrin.h	`short _InterlockedExchange16(short volatile *, short);`
`_InterlockedExchange8`		intrin.h	`char _InterlockedExchange8(char volatile *, char);`
`_InterlockedExchangeAdd`		intrin.h	`long _InterlockedExchangeAdd(long volatile *, long);`
`_InterlockedExchangeAdd_HLEAcquire`	HLE	immintrin.h	`long _InterlockedExchangeAdd_HLEAcquire(long volatile *, long);`
`_InterlockedExchangeAdd_HLERelease`	HLE	immintrin.h	`long _InterlockedExchangeAdd_HLERelease(long volatile *, long);`
`_InterlockedExchangeAdd16`		intrin.h	`short _InterlockedExchangeAdd16(short volatile *, short);`
`_InterlockedExchangeAdd8`		intrin.h	`char _InterlockedExchangeAdd8(char volatile *, char);`
`_InterlockedExchangePointer`		intrin.h	`void * _InterlockedExchangePointer(void *volatile *, void *);`
`_InterlockedExchangePointer_HLEAcquire`	HLE	immintrin.h	`void * _InterlockedExchangePointer_HLEAcquire(voi *volatile *, void *);`
`_InterlockedExchangePointer_HLERelease`	HLE	immintrin.h	`void * _InterlockedExchangePointer_HLERelease(voi *volatile *, void *);`
`_InterlockedIncrement`		intrin.h	`long _InterlockedIncrement(long volatile *);`
`_InterlockedIncrement16`		intrin.h	`short _InterlockedIncrement16(short volatile *);`
`_InterlockedOr`		intrin.h	`long _InterlockedOr(long volatile *, long);`
`_InterlockedOr_HLEAcquire`	HLE	immintrin.h	`long _InterlockedOr_HLEAcquire(long volatile *, long);`
`_InterlockedOr_HLERelease`	HLE	immintrin.h	`long _InterlockedOr_HLERelease(long volatile *, long);`
`_InterlockedOr16`		intrin.h	`short _InterlockedOr16(short volatile *, short);`
`_InterlockedOr8`		intrin.h	`char _InterlockedOr8(char volatile *, char);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_InterlockedXor`		intrin.h	`long _InterlockedXor(long volatile *, long);`
`_InterlockedXor_HLEAcquire`	HLE	immintrin.h	`long _InterlockedXor_HLEAcquire(long volatile *, long);`
`_InterlockedXor_HLERelease`	HLE	immintrin.h	`long _InterlockedXor_HLERelease(long volatile *, long);`
`_InterlockedXor16`		intrin.h	`short _InterlockedXor16(short volatile *, short);`
`_InterlockedXor8`		intrin.h	`char _InterlockedXor8(char volatile *, char);`
`__invlpg`		intrin.h	`void __invlpg(void*);`
`_invpcid`	INVPCID	immintrin.h	`void _invpcid(unsigned int, void *);`
`__inword`		intrin.h	`unsigned short __inword(unsigned short);`
`__inwordstring`		intrin.h	`void __inwordstring(unsigned short, unsigned short *, unsigned long);`
`_lgdt`		intrin.h	`void _lgdt(void*);`
`__lidt`		intrin.h	`void __lidt(void*);`
`__ll_lshift`		intrin.h	`unsigned __int64 [pascal/cdecl] __ll_lshift(unsigned __int64, int);`
`__ll_rshift`		intrin.h	`__int64 [pascal/cdecl] __ll_rshift(__int64, int);`
`_loadbe_i16`	MOVBE	immintrin.h	`short _loadbe_i16(void const*);` [Macro]
`_loadbe_i32`	MOVBE	immintrin.h	`int _loadbe_i32(void const*);` [Macro]
`_load_be_u16`	MOVBE	immintrin.h	`unsigned short _load_be_u16(void const*);` [Macro]
`_load_be_u32`	MOVBE	immintrin.h	`unsigned int _load_be_u32(void const*);` [Macro]
`__llwpcb`	LWP	ammintrin.h	`void __llwpcb(void *);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__lwpins32`	LWP	ammintrin.h	`unsigned char __lwpins32(unsigned int, unsigned int, unsigned int);`
`__lwpval32`	LWP	ammintrin.h	`void __lwpval32(unsigned int, unsigned int, unsigned int);`
`__lzcnt`	LZCNT	intrin.h	`unsigned int __lzcnt(unsigned int);`
`_lzcnt_u32`	BMI	ammintrin.h, immintrin.h	`unsigned int _lzcnt_u32(unsigned int);`
`__lzcnt16`	LZCNT	intrin.h	`unsigned short __lzcnt16(unsigned short);`
`_m_empty`	MMX	intrin.h	`void _m_empty(void);`
`_m_femms`	3DNOW	intrin.h	`void _m_femms(void);`
`_m_from_float`	3DNOW	intrin.h	`__m64 _m_from_float(float);`
`_m_from_int`	MMX	intrin.h	`__m64 _m_from_int(int);`
`_m_maskmovq`	SSE	intrin.h	`void _m_maskmovq(__m64, __m64, char*);`
`_m_packssdw`	MMX	intrin.h	`__m64 _m_packssdw(__m64, __m64);`
`_m_packsswb`	MMX	intrin.h	`__m64 _m_packsswb(__m64, __m64);`
`_m_packuswb`	MMX	intrin.h	`__m64 _m_packuswb(__m64, __m64);`
`_m_paddb`	MMX	intrin.h	`__m64 _m_paddb(__m64, __m64);`
`_m_paddd`	MMX	intrin.h	`__m64 _m_paddd(__m64, __m64);`
`_m_paddsb`	MMX	intrin.h	`__m64 _m_paddsb(__m64, __m64);`
`_m_paddsw`	MMX	intrin.h	`__m64 _m_paddsw(__m64, __m64);`
`_m_paddusb`	MMX	intrin.h	`__m64 _m_paddusb(__m64, __m64);`
`_m_paddusw`	MMX	intrin.h	`__m64 _m_paddusw(__m64, __m64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_m_paddw`	MMX	intrin.h	`__m64 _m_paddw(__m64, __m64);`
`_m_pand`	MMX	intrin.h	`__m64 _m_pand(__m64, __m64);`
`_m_pandn`	MMX	intrin.h	`__m64 _m_pandn(__m64, __m64);`
`_m_pavgb`	SSE	intrin.h	`__m64 _m_pavgb(__m64, __m64);`
`_m_pavgusb`	3DNOW	intrin.h	`__m64 _m_pavgusb(__m64, __m64);`
`_m_pavgw`	SSE	intrin.h	`__m64 _m_pavgw(__m64, __m64);`
`_m_pcmpeqb`	MMX	intrin.h	`__m64 _m_pcmpeqb(__m64, __m64);`
`_m_pcmpeqd`	MMX	intrin.h	`__m64 _m_pcmpeqd(__m64, __m64);`
`_m_pcmpeqw`	MMX	intrin.h	`__m64 _m_pcmpeqw(__m64, __m64);`
`_m_pcmpgtb`	MMX	intrin.h	`__m64 _m_pcmpgtb(__m64, __m64);`
`_m_pcmpgtd`	MMX	intrin.h	`__m64 _m_pcmpgtd(__m64, __m64);`
`_m_pcmpgtw`	MMX	intrin.h	`__m64 _m_pcmpgtw(__m64, __m64);`
`_m_pextrw`	SSE	intrin.h	`int _m_pextrw(__m64, int);`
`_m_pf2id`	3DNOW	intrin.h	`__m64 _m_pf2id(__m64);`
`_m_pf2iw`	3DNOWEXT	intrin.h	`__m64 _m_pf2iw(__m64);`
`_m_pfacc`	3DNOW	intrin.h	`__m64 _m_pfacc(__m64, __m64);`
`_m_pfadd`	3DNOW	intrin.h	`__m64 _m_pfadd(__m64, __m64);`
`_m_pfcmpeq`	3DNOW	intrin.h	`__m64 _m_pfcmpeq(__m64, __m64);`
`_m_pfcmpge`	3DNOW	intrin.h	`__m64 _m_pfcmpge(__m64, __m64);`
`_m_pfcmpgt`	3DNOW	intrin.h	`__m64 _m_pfcmpgt(__m64, __m64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_m_pfmax`	3DNOW	intrin.h	`__m64 _m_pfmax(__m64, __m64);`
`_m_pfmin`	3DNOW	intrin.h	`__m64 _m_pfmin(__m64, __m64);`
`_m_pfmul`	3DNOW	intrin.h	`__m64 _m_pfmul(__m64, __m64);`
`_m_pfnacc`	3DNOWEXT	intrin.h	`__m64 _m_pfnacc(__m64, __m64);`
`_m_pfpnacc`	3DNOWEXT	intrin.h	`__m64 _m_pfpnacc(__m64, __m64);`
`_m_pfrcp`	3DNOW	intrin.h	`__m64 _m_pfrcp(__m64);`
`_m_pfrcpit1`	3DNOW	intrin.h	`__m64 _m_pfrcpit1(__m64, __m64);`
`_m_pfrcpit2`	3DNOW	intrin.h	`__m64 _m_pfrcpit2(__m64, __m64);`
`_m_pfrsqit1`	3DNOW	intrin.h	`__m64 _m_pfrsqit1(__m64, __m64);`
`_m_pfrsqrt`	3DNOW	intrin.h	`__m64 _m_pfrsqrt(__m64);`
`_m_pfsub`	3DNOW	intrin.h	`__m64 _m_pfsub(__m64, __m64);`
`_m_pfsubr`	3DNOW	intrin.h	`__m64 _m_pfsubr(__m64, __m64);`
`_m_pi2fd`	3DNOW	intrin.h	`__m64 _m_pi2fd(__m64);`
`_m_pi2fw`	3DNOWEXT	intrin.h	`__m64 _m_pi2fw(__m64);`
`_m_pinsrw`	SSE	intrin.h	`__m64 _m_pinsrw(__m64, int, int);`
`_m_pmaddwd`	MMX	intrin.h	`__m64 _m_pmaddwd(__m64, __m64);`
`_m_pmaxsw`	SSE	intrin.h	`__m64 _m_pmaxsw(__m64, __m64);`
`_m_pmaxub`	SSE	intrin.h	`__m64 _m_pmaxub(__m64, __m64);`
`_m_pminsw`	SSE	intrin.h	`__m64 _m_pminsw(__m64, __m64);`
`_m_pminub`	SSE	intrin.h	`__m64 _m_pminub(__m64, __m64);`
`_m_pmovmskb`	SSE	intrin.h	`int _m_pmovmskb(__m64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_m_pmulhrw`	3DNOW	intrin.h	`__m64 _m_pmulhrw(__m64, __m64);`
`_m_pmulhuw`	SSE	intrin.h	`__m64 _m_pmulhuw(__m64, __m64);`
`_m_pmulhw`	MMX	intrin.h	`__m64 _m_pmulhw(__m64, __m64);`
`_m_pmullw`	MMX	intrin.h	`__m64 _m_pmullw(__m64, __m64);`
`_m_por`	MMX	intrin.h	`__m64 _m_por(__m64, __m64);`
`_m_prefetch`	3DNOW	intrin.h	`void _m_prefetch(void*);`
`_m_prefetchw`	3DNOW	intrin.h	`void _m_prefetchw(void*);`
`_m_psadbw`	SSE	intrin.h	`__m64 _m_psadbw(__m64, __m64);`
`_m_pshufw`	SSE	intrin.h	`__m64 _m_pshufw(__m64, int);`
`_m_pslld`	MMX	intrin.h	`__m64 _m_pslld(__m64, __m64);`
`_m_pslldi`	MMX	intrin.h	`__m64 _m_pslldi(__m64, int);`
`_m_psllq`	MMX	intrin.h	`__m64 _m_psllq(__m64, __m64);`
`_m_psllqi`	MMX	intrin.h	`__m64 _m_psllqi(__m64, int);`
`_m_psllw`	MMX	intrin.h	`__m64 _m_psllw(__m64, __m64);`
`_m_psllwi`	MMX	intrin.h	`__m64 _m_psllwi(__m64, int);`
`_m_psrad`	MMX	intrin.h	`__m64 _m_psrad(__m64, __m64);`
`_m_psradi`	MMX	intrin.h	`__m64 _m_psradi(__m64, int);`
`_m_psraw`	MMX	intrin.h	`__m64 _m_psraw(__m64, __m64);`
`_m_psrawi`	MMX	intrin.h	`__m64 _m_psrawi(__m64, int);`
`_m_psrld`	MMX	intrin.h	`__m64 _m_psrld(__m64, __m64);`
`_m_psrldi`	MMX	intrin.h	`__m64 _m_psrldi(__m64, int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_m_psrlq`	MMX	intrin.h	`__m64 _m_psrlq(__m64, __m64);`
`_m_psrlqi`	MMX	intrin.h	`__m64 _m_psrlqi(__m64, int);`
`_m_psrlw`	MMX	intrin.h	`__m64 _m_psrlw(__m64, __m64);`
`_m_psrlwi`	MMX	intrin.h	`__m64 _m_psrlwi(__m64, int);`
`_m_psubb`	MMX	intrin.h	`__m64 _m_psubb(__m64, __m64);`
`_m_psubd`	MMX	intrin.h	`__m64 _m_psubd(__m64, __m64);`
`_m_psubsb`	MMX	intrin.h	`__m64 _m_psubsb(__m64, __m64);`
`_m_psubsw`	MMX	intrin.h	`__m64 _m_psubsw(__m64, __m64);`
`_m_psubusb`	MMX	intrin.h	`__m64 _m_psubusb(__m64, __m64);`
`_m_psubusw`	MMX	intrin.h	`__m64 _m_psubusw(__m64, __m64);`
`_m_psubw`	MMX	intrin.h	`__m64 _m_psubw(__m64, __m64);`
`_m_pswapd`	3DNOWEXT	intrin.h	`__m64 _m_pswapd(__m64);`
`_m_punpckhbw`	MMX	intrin.h	`__m64 _m_punpckhbw(__m64, __m64);`
`_m_punpckhdq`	MMX	intrin.h	`__m64 _m_punpckhdq(__m64, __m64);`
`_m_punpckhwd`	MMX	intrin.h	`__m64 _m_punpckhwd(__m64, __m64);`
`_m_punpcklbw`	MMX	intrin.h	`__m64 _m_punpcklbw(__m64, __m64);`
`_m_punpckldq`	MMX	intrin.h	`__m64 _m_punpckldq(__m64, __m64);`
`_m_punpcklwd`	MMX	intrin.h	`__m64 _m_punpcklwd(__m64, __m64);`
`_m_pxor`	MMX	intrin.h	`__m64 _m_pxor(__m64, __m64);`
`_m_to_float`	3DNOW	intrin.h	`float _m_to_float(__m64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_m_to_int`	MMX	intrin.h	`int _m_to_int(__m64);`
`_mm_abs_epi16`	SSSE3	intrin.h	`__m128i _mm_abs_epi16(__m128i);`
`_mm_abs_epi32`	SSSE3	intrin.h	`__m128i _mm_abs_epi32(__m128i);`
`_mm_abs_epi8`	SSSE3	intrin.h	`__m128i _mm_abs_epi8(__m128i);`
`_mm_abs_pi16`	SSSE3	intrin.h	`__m64 _mm_abs_pi16(__m64);`
`_mm_abs_pi32`	SSSE3	intrin.h	`__m64 _mm_abs_pi32(__m64);`
`_mm_abs_pi8`	SSSE3	intrin.h	`__m64 _mm_abs_pi8(__m64);`
`_mm_add_epi16`	SSE2	intrin.h	`__m128i _mm_add_epi16(__m128i, __m128i);`
`_mm_add_epi32`	SSE2	intrin.h	`__m128i _mm_add_epi32(__m128i, __m128i);`
`_mm_add_epi64`	SSE2	intrin.h	`__m128i _mm_add_epi64(__m128i, __m128i);`
`_mm_add_epi8`	SSE2	intrin.h	`__m128i _mm_add_epi8(__m128i, __m128i);`
`_mm_add_pd`	SSE2	intrin.h	`__m128d _mm_add_pd(__m128d, __m128d);`
`_mm_add_pi8`	MMX	mmintrin.h	`__m64 _mm_add_pi8(__m64, __m64);` [Macro]
`_mm_add_pi16`	MMX	mmintrin.h	`__m64 _mm_add_pi16(__m64, __m64);` [Macro]
`_mm_add_pi32`	MMX	mmintrin.h	`__m64 _mm_add_pi32(__m64, __m64);` [Macro]
`_mm_add_ps`	SSE	intrin.h	`__m128 _mm_add_ps(__m128, __m128);`
`_mm_add_sd`	SSE2	intrin.h	`__m128d _mm_add_sd(__m128d, __m128d);`
`_mm_add_si64`	SSE2	intrin.h	`__m64 _mm_add_si64(__m64, __m64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_add_ss`	SSE	intrin.h	`__m128 _mm_add_ss(__m128, __m128);`
`_mm_adds_epi16`	SSE2	intrin.h	`__m128i _mm_adds_epi16(__m128i, __m128i);`
`_mm_adds_epi8`	SSE2	intrin.h	`__m128i _mm_adds_epi8(__m128i, __m128i);`
`_mm_adds_epu16`	SSE2	intrin.h	`__m128i _mm_adds_epu16(__m128i, __m128i);`
`_mm_adds_epu8`	SSE2	intrin.h	`__m128i _mm_adds_epu8(__m128i, __m128i);`
`_mm_adds_pi8`	MMX	mmintrin.h	`__m64 _mm_adds_pi8(__m64, __m64);` [Macro]
`_mm_adds_pi16`	MMX	mmintrin.h	`__m64 _mm_adds_pi16(__m64, __m64);` [Macro]
`_mm_adds_pu8`	MMX	mmintrin.h	`__m64 _mm_adds_pu8(__m64, __m64);` [Macro]
`_mm_adds_pu16`	MMX	mmintrin.h	`__m64 _mm_adds_pu16(__m64, __m64);` [Macro]
`_mm_addsub_pd`	SSE3	intrin.h	`__m128d _mm_addsub_pd(__m128d, __m128d);`
`_mm_addsub_ps`	SSE3	intrin.h	`__m128 _mm_addsub_ps(__m128, __m128);`
`_mm_aesdec_si128`	AESNI	immintrin.h	`__m128i _mm_aesdec_si128(__m128i, __m128i);`
`_mm_aesdeclast_si128`	AESNI	immintrin.h	`__m128i _mm_aesdeclast_si128(__m128i, __m128i);`
`_mm_aesenc_si128`	AESNI	immintrin.h	`__m128i _mm_aesenc_si128(__m128i, __m128i);`
`_mm_aesenclast_si128`	AESNI	immintrin.h	`__m128i _mm_aesenclast_si128(__m128i, __m128i);`
`_mm_aesimc_si128`	AESNI	immintrin.h	`__m128i _mm_aesimc_si128 (__m128i);`
`_mm_aeskeygenassist_si128`	AESNI	immintrin.h	`__m128i _mm_aeskeygenassist_si128 (__m128i, const int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_alignr_epi8`	SSSE3	intrin.h	`__m128i _mm_alignr_epi8(__m128i, __m128i, int);`
`_mm_alignr_pi8`	SSSE3	intrin.h	`__m64 _mm_alignr_pi8(__m64, __m64, int);`
`_mm_and_pd`	SSE2	intrin.h	`__m128d _mm_and_pd(__m128d, __m128d);`
`_mm_and_ps`	SSE	intrin.h	`__m128 _mm_and_ps(__m128, __m128);`
`_mm_and_si64`	MMX	mmintrin.h	`__m64 _mm_and_si64(__m64, __m64);` [Macro]
`_mm_and_si128`	SSE2	intrin.h	`__m128i _mm_and_si128(__m128i, __m128i);`
`_mm_andnot_pd`	SSE2	intrin.h	`__m128d _mm_andnot_pd(__m128d, __m128d);`
`_mm_andnot_ps`	SSE	intrin.h	`__m128 _mm_andnot_ps(__m128, __m128);`
`_mm_andnot_si64`	MMX	mmintrin.h	`__m64 _mm_andnot_si64(__m64, __m64);` [Macro]
`_mm_andnot_si128`	SSE2	intrin.h	`__m128i _mm_andnot_si128(__m128i, __m128i);`
`_mm_avg_epu16`	SSE2	intrin.h	`__m128i _mm_avg_epu16(__m128i, __m128i);`
`_mm_avg_epu8`	SSE2	intrin.h	`__m128i _mm_avg_epu8(__m128i, __m128i);`
`_mm_blend_epi16`	SSE41	intrin.h	`__m128i _mm_blend_epi16 (__m128i, __m128i, const int);`
`_mm_blend_epi32`	AVX2	immintrin.h	`__m128i _mm_blend_epi32(__m128i, __m128i, const int);`
`_mm_blend_pd`	SSE41	intrin.h	`__m128d _mm_blend_pd (__m128d, __m128d, const int);`
`_mm_blend_ps`	SSE41	intrin.h	`__m128 _mm_blend_ps (__m128, __m128, const int);`
`_mm_blendv_epi8`	SSE41	intrin.h	`__m128i _mm_blendv_epi8 (__m128i, __m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_blendv_pd`	SSE41	intrin.h	`__m128d _mm_blendv_pd(__m128d, __m128d, __m128d);`
`_mm_blendv_ps`	SSE41	intrin.h	`__m128 _mm_blendv_ps(__m128, __m128, __m128);`
`_mm_broadcast_ss`	AVX	immintrin.h	`__m128 _mm_broadcast_ss(float const *);`
`_mm_broadcastb_epi8`	AVX2	immintrin.h	`__m128i _mm_broadcastb_epi8(__m128i);`
`_mm_broadcastd_epi32`	AVX2	immintrin.h	`__m128i _mm_broadcastd_epi32(__m128i);`
`_mm_broadcastq_epi64`	AVX2	immintrin.h	`__m128i _mm_broadcastq_epi64(__m128i);`
`_mm_broadcastsd_pd`	AVX2	immintrin.h	`__m128d _mm_broadcastsd_pd(__m128d);`
`_mm_broadcastss_ps`	AVX2	immintrin.h	`__m128 _mm_broadcastss_ps(__m128);`
`_mm_broadcastw_epi16`	AVX2	immintrin.h	`__m128i _mm_broadcastw_epi16(__m128i);`
`_mm_castpd_ps`	SSSE3	intrin.h	`__m128 _mm_castpd_ps(__m128d);`
`_mm_castpd_si128`	SSSE3	intrin.h	`__m128i _mm_castpd_si128(__m128d);`
`_mm_castps_pd`	SSSE3	intrin.h	`__m128d _mm_castps_pd(__m128);`
`_mm_castps_si128`	SSSE3	intrin.h	`__m128i _mm_castps_si128(__m128);`
`_mm_castsi128_pd`	SSSE3	intrin.h	`__m128d _mm_castsi128_pd(__m128i);`
`_mm_castsi128_ps`	SSSE3	intrin.h	`__m128 _mm_castsi128_ps(__m128i);`
`_mm_clflush`	SSE2	intrin.h	`void _mm_clflush(void const *);`
`_mm_clmulepi64_si128`	PCLMULQDQ	immintrin.h	`__m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int);`
`_mm_cmov_si128`	XOP	ammintrin.h	`__m128i _mm_cmov_si128(__m128i, __m128i, __m128i);`
`_mm_cmp_pd`	AVX	immintrin.h	`__m128d _mm_cmp_pd(__m128d, __m128d, const int);`
`_mm_cmp_ps`	AVX	immintrin.h	`__m128 _mm_cmp_ps(__m128, __m128, const int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cmp_sd`	AVX	immintrin.h	`__m128d _mm_cmp_sd(__m128d, __m128d, const int);`
`_mm_cmp_ss`	AVX	immintrin.h	`__m128 _mm_cmp_ss(__m128, __m128, const int);`
`_mm_cmpeq_epi16`	SSE2	intrin.h	`__m128i _mm_cmpeq_epi16(__m128i, __m128i);`
`_mm_cmpeq_epi32`	SSE2	intrin.h	`__m128i _mm_cmpeq_epi32(__m128i, __m128i);`
`_mm_cmpeq_epi64`	SSE41	intrin.h	`__m128i _mm_cmpeq_epi64(__m128i, __m128i);`
`_mm_cmpeq_epi8`	SSE2	intrin.h	`__m128i _mm_cmpeq_epi8(__m128i, __m128i);`
`_mm_cmpeq_pd`	SSE2	intrin.h	`__m128d _mm_cmpeq_pd(__m128d, __m128d);`
`_mm_cmpeq_pi8`	MMX	mmintrin.h	`__m64 _mm_cmpeq_pi8(__m64, __m64);` [Macro]
`_mm_cmpeq_pi16`	MMX	mmintrin.h	`__m64 _mm_cmpeq_pi16(__m64, __m64);` [Macro]
`_mm_cmpeq_pi32`	MMX	mmintrin.h	`__m64 _mm_cmpeq_pi32(__m64, __m64);` [Macro]
`_mm_cmpeq_ps`	SSE	intrin.h	`__m128 _mm_cmpeq_ps(__m128, __m128);`
`_mm_cmpeq_sd`	SSE2	intrin.h	`__m128d _mm_cmpeq_sd(__m128d, __m128d);`
`_mm_cmpeq_ss`	SSE	intrin.h	`__m128 _mm_cmpeq_ss(__m128, __m128);`
`_mm_cmpestra`	SSE42	intrin.h	`int _mm_cmpestra(__m128i, int, __m128i, int, const int);`
`_mm_cmpestrc`	SSE42	intrin.h	`int _mm_cmpestrc(__m128i, int, __m128i, int, const int);`
`_mm_cmpestri`	SSE42	intrin.h	`int _mm_cmpestri(__m128i, int, __m128i, int, const int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cmpestrm`	SSE42	intrin.h	`__m128i _mm_cmpestrm(__m128i, int, __m128i, int, const int);`
`_mm_cmpestro`	SSE42	intrin.h	`int _mm_cmpestro(__m128i, int, __m128i, int, const int);`
`_mm_cmpestrs`	SSE42	intrin.h	`int _mm_cmpestrs(__m128i, int, __m128i, int, const int);`
`_mm_cmpestrz`	SSE42	intrin.h	`int _mm_cmpestrz(__m128i, int, __m128i, int, const int);`
`_mm_cmpge_pd`	SSE2	intrin.h	`__m128d _mm_cmpge_pd(__m128d, __m128d);`
`_mm_cmpge_ps`	SSE	intrin.h	`__m128 _mm_cmpge_ps(__m128, __m128);`
`_mm_cmpge_sd`	SSE2	intrin.h	`__m128d _mm_cmpge_sd(__m128d, __m128d);`
`_mm_cmpge_ss`	SSE	intrin.h	`__m128 _mm_cmpge_ss(__m128, __m128);`
`_mm_cmpgt_epi16`	SSE2	intrin.h	`__m128i _mm_cmpgt_epi16(__m128i, __m128i);`
`_mm_cmpgt_epi32`	SSE2	intrin.h	`__m128i _mm_cmpgt_epi32(__m128i, __m128i);`
`_mm_cmpgt_epi64`	SSE42	intrin.h	`__m128i _mm_cmpgt_epi64(__m128i, __m128i);`
`_mm_cmpgt_epi8`	SSE2	intrin.h	`__m128i _mm_cmpgt_epi8(__m128i, __m128i);`
`_mm_cmpgt_pi8`	MMX	mmintrin.h	`__m64 _mm_cmpgt_pi8(__m64, __m64);` [Macro]
`_mm_cmpgt_pi16`	MMX	mmintrin.h	`__m64 _mm_cmpgt_pi16(__m64, __m64);` [Macro]
`_mm_cmpgt_pi32`	MMX	mmintrin.h	`__m64 _mm_cmpgt_pi32(__m64, __m64);` [Macro]
`_mm_cmpgt_pd`	SSE2	intrin.h	`__m128d _mm_cmpgt_pd(__m128d, __m128d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cmpgt_ps`	SSE	intrin.h	`__m128 _mm_cmpgt_ps(__m128, __m128);`
`_mm_cmpgt_sd`	SSE2	intrin.h	`__m128d _mm_cmpgt_sd(__m128d, __m128d);`
`_mm_cmpgt_ss`	SSE	intrin.h	`__m128 _mm_cmpgt_ss(__m128, __m128);`
`_mm_cmpistra`	SSE42	intrin.h	`int _mm_cmpistra(__m128i, __m128i, const int);`
`_mm_cmpistrc`	SSE42	intrin.h	`int _mm_cmpistrc(__m128i, __m128i, const int);`
`_mm_cmpistri`	SSE42	intrin.h	`int _mm_cmpistri(__m128i, __m128i, const int);`
`_mm_cmpistrm`	SSE42	intrin.h	`__m128i _mm_cmpistrm(__m128i, __m128i, const int);`
`_mm_cmpistro`	SSE42	intrin.h	`int _mm_cmpistro(__m128i, __m128i, const int);`
`_mm_cmpistrs`	SSE42	intrin.h	`int _mm_cmpistrs(__m128i, __m128i, const int);`
`_mm_cmpistrz`	SSE42	intrin.h	`int _mm_cmpistrz(__m128i, __m128i, const int);`
`_mm_cmple_pd`	SSE2	intrin.h	`__m128d _mm_cmple_pd(__m128d, __m128d);`
`_mm_cmple_ps`	SSE	intrin.h	`__m128 _mm_cmple_ps(__m128, __m128);`
`_mm_cmple_sd`	SSE2	intrin.h	`__m128d _mm_cmple_sd(__m128d, __m128d);`
`_mm_cmple_ss`	SSE	intrin.h	`__m128 _mm_cmple_ss(__m128, __m128);`
`_mm_cmplt_epi16`	SSE2	intrin.h	`__m128i _mm_cmplt_epi16(__m128i, __m128i);`
`_mm_cmplt_epi32`	SSE2	intrin.h	`__m128i _mm_cmplt_epi32(__m128i, __m128i);`
`_mm_cmplt_epi8`	SSE2	intrin.h	`__m128i _mm_cmplt_epi8(__m128i, __m128i);`
`_mm_cmplt_pd`	SSE2	intrin.h	`__m128d _mm_cmplt_pd(__m128d, __m128d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cmplt_ps`	SSE	intrin.h	`__m128 _mm_cmplt_ps(__m128, __m128);`
`_mm_cmplt_sd`	SSE2	intrin.h	`__m128d _mm_cmplt_sd(__m128d, __m128d);`
`_mm_cmplt_ss`	SSE	intrin.h	`__m128 _mm_cmplt_ss(__m128, __m128);`
`_mm_cmpneq_pd`	SSE2	intrin.h	`__m128d _mm_cmpneq_pd(__m128d, __m128d);`
`_mm_cmpneq_ps`	SSE	intrin.h	`__m128 _mm_cmpneq_ps(__m128, __m128);`
`_mm_cmpneq_sd`	SSE2	intrin.h	`__m128d _mm_cmpneq_sd(__m128d, __m128d);`
`_mm_cmpneq_ss`	SSE	intrin.h	`__m128 _mm_cmpneq_ss(__m128, __m128);`
`_mm_cmpnge_pd`	SSE2	intrin.h	`__m128d _mm_cmpnge_pd(__m128d, __m128d);`
`_mm_cmpnge_ps`	SSE	intrin.h	`__m128 _mm_cmpnge_ps(__m128, __m128);`
`_mm_cmpnge_sd`	SSE2	intrin.h	`__m128d _mm_cmpnge_sd(__m128d, __m128d);`
`_mm_cmpnge_ss`	SSE	intrin.h	`__m128 _mm_cmpnge_ss(__m128, __m128);`
`_mm_cmpngt_pd`	SSE2	intrin.h	`__m128d _mm_cmpngt_pd(__m128d, __m128d);`
`_mm_cmpngt_ps`	SSE	intrin.h	`__m128 _mm_cmpngt_ps(__m128, __m128);`
`_mm_cmpngt_sd`	SSE2	intrin.h	`__m128d _mm_cmpngt_sd(__m128d, __m128d);`
`_mm_cmpngt_ss`	SSE	intrin.h	`__m128 _mm_cmpngt_ss(__m128, __m128);`
`_mm_cmpnle_pd`	SSE2	intrin.h	`__m128d _mm_cmpnle_pd(__m128d, __m128d);`
`_mm_cmpnle_ps`	SSE	intrin.h	`__m128 _mm_cmpnle_ps(__m128, __m128);`
`_mm_cmpnle_sd`	SSE2	intrin.h	`__m128d _mm_cmpnle_sd(__m128d, __m128d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cmpnle_ss`	SSE	intrin.h	`__m128 _mm_cmpnle_ss(__m128, __m128);`
`_mm_cmpnlt_pd`	SSE2	intrin.h	`__m128d _mm_cmpnlt_pd(__m128d, __m128d);`
`_mm_cmpnlt_ps`	SSE	intrin.h	`__m128 _mm_cmpnlt_ps(__m128, __m128);`
`_mm_cmpnlt_sd`	SSE2	intrin.h	`__m128d _mm_cmpnlt_sd(__m128d, __m128d);`
`_mm_cmpnlt_ss`	SSE	intrin.h	`__m128 _mm_cmpnlt_ss(__m128, __m128);`
`_mm_cmpord_pd`	SSE2	intrin.h	`__m128d _mm_cmpord_pd(__m128d, __m128d);`
`_mm_cmpord_ps`	SSE	intrin.h	`__m128 _mm_cmpord_ps(__m128, __m128);`
`_mm_cmpord_sd`	SSE2	intrin.h	`__m128d _mm_cmpord_sd(__m128d, __m128d);`
`_mm_cmpord_ss`	SSE	intrin.h	`__m128 _mm_cmpord_ss(__m128, __m128);`
`_mm_cmpunord_pd`	SSE2	intrin.h	`__m128d _mm_cmpunord_pd(__m128d, __m128d);`
`_mm_cmpunord_ps`	SSE	intrin.h	`__m128 _mm_cmpunord_ps(__m128, __m128);`
`_mm_cmpunord_sd`	SSE2	intrin.h	`__m128d _mm_cmpunord_sd(__m128d, __m128d);`
`_mm_cmpunord_ss`	SSE	intrin.h	`__m128 _mm_cmpunord_ss(__m128, __m128);`
`_mm_com_epi16`	XOP	ammintrin.h	`__m128i _mm_com_epi16(__m128i, __m128i, int);`
`_mm_com_epi32`	XOP	ammintrin.h	`__m128i _mm_com_epi32(__m128i, __m128i, int);`
`_mm_com_epi64`	XOP	ammintrin.h	`__m128i _mm_com_epi32(__m128i, __m128i, int);`
`_mm_com_epi8`	XOP	ammintrin.h	`__m128i _mm_com_epi8(__m128i, __m128i, int);`
`_mm_com_epu16`	XOP	ammintrin.h	`__m128i _mm_com_epu16(__m128i, __m128i, int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_com_epu32`	XOP	ammintrin.h	`__m128i _mm_com_epu32(__m128i, __m128i, int);`
`_mm_com_epu64`	XOP	ammintrin.h	`__m128i _mm_com_epu32(__m128i, __m128i, int);`
`_mm_com_epu8`	XOP	ammintrin.h	`__m128i _mm_com_epu8(__m128i, __m128i, int);`
`_mm_comieq_sd`	SSE2	intrin.h	`int _mm_comieq_sd(__m128d, __m128d);`
`_mm_comieq_ss`	SSE	intrin.h	`int _mm_comieq_ss(__m128, __m128);`
`_mm_comige_sd`	SSE2	intrin.h	`int _mm_comige_sd(__m128d, __m128d);`
`_mm_comige_ss`	SSE	intrin.h	`int _mm_comige_ss(__m128, __m128);`
`_mm_comigt_sd`	SSE2	intrin.h	`int _mm_comigt_sd(__m128d, __m128d);`
`_mm_comigt_ss`	SSE	intrin.h	`int _mm_comigt_ss(__m128, __m128);`
`_mm_comile_sd`	SSE2	intrin.h	`int _mm_comile_sd(__m128d, __m128d);`
`_mm_comile_ss`	SSE	intrin.h	`int _mm_comile_ss(__m128, __m128);`
`_mm_comilt_sd`	SSE2	intrin.h	`int _mm_comilt_sd(__m128d, __m128d);`
`_mm_comilt_ss`	SSE	intrin.h	`int _mm_comilt_ss(__m128, __m128);`
`_mm_comineq_sd`	SSE2	intrin.h	`int _mm_comineq_sd(__m128d, __m128d);`
`_mm_comineq_ss`	SSE	intrin.h	`int _mm_comineq_ss(__m128, __m128);`
`_mm_crc32_u16`	SSE42	intrin.h	`unsigned int _mm_crc32_u16(unsigned int, unsigned short);`
`_mm_crc32_u32`	SSE42	intrin.h	`unsigned int _mm_crc32_u32(unsigned int, unsigned int);`
`_mm_crc32_u8`	SSE42	intrin.h	`unsigned int _mm_crc32_u8(unsigned int, unsigned char);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cvt_pi2ps`	SSE	intrin.h	`__m128 _mm_cvt_pi2ps(__m128, __m64);`
`_mm_cvt_ps2pi`	SSE	intrin.h	`__m64 _mm_cvt_ps2pi(__m128);`
`_mm_cvt_si2ss`	SSE	intrin.h	`__m128 _mm_cvt_si2ss(__m128, int);`
`_mm_cvt_ss2si`	SSE	intrin.h	`int _mm_cvt_ss2si(__m128);`
`_mm_cvtepi16_epi32`	SSE41	intrin.h	`__m128i _mm_cvtepi16_epi32(__m128i);`
`_mm_cvtepi16_epi64`	SSE41	intrin.h	`__m128i _mm_cvtepi16_epi64(__m128i);`
`_mm_cvtepi32_epi64`	SSE41	intrin.h	`__m128i _mm_cvtepi32_epi64(__m128i);`
`_mm_cvtepi32_pd`	SSE2	intrin.h	`__m128d _mm_cvtepi32_pd(__m128i);`
`_mm_cvtepi32_ps`	SSE2	intrin.h	`__m128 _mm_cvtepi32_ps(__m128i);`
`_mm_cvtepi8_epi16`	SSE41	intrin.h	`__m128i _mm_cvtepi8_epi16 (__m128i);`
`_mm_cvtepi8_epi32`	SSE41	intrin.h	`__m128i _mm_cvtepi8_epi32 (__m128i);`
`_mm_cvtepi8_epi64`	SSE41	intrin.h	`__m128i _mm_cvtepi8_epi64 (__m128i);`
`_mm_cvtepu16_epi32`	SSE41	intrin.h	`__m128i _mm_cvtepu16_epi32(__m128i);`
`_mm_cvtepu16_epi64`	SSE41	intrin.h	`__m128i _mm_cvtepu16_epi64(__m128i);`
`_mm_cvtepu32_epi64`	SSE41	intrin.h	`__m128i _mm_cvtepu32_epi64(__m128i);`
`_mm_cvtepu8_epi16`	SSE41	intrin.h	`__m128i _mm_cvtepu8_epi16 (__m128i);`
`_mm_cvtepu8_epi32`	SSE41	intrin.h	`__m128i _mm_cvtepu8_epi32 (__m128i);`
`_mm_cvtepu8_epi64`	SSE41	intrin.h	`__m128i _mm_cvtepu8_epi64 (__m128i);`
`_mm_cvtpd_epi32`	SSE2	intrin.h	`__m128i _mm_cvtpd_epi32(__m128d);`
`_mm_cvtpd_pi32`	SSE2	intrin.h	`__m64 _mm_cvtpd_pi32(__m128d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cvtpd_ps`	SSE2	intrin.h	`__m128 _mm_cvtpd_ps(__m128d);`
`_mm_cvtph_ps`	F16C	immintrin.h	`__m128 _mm_cvtph_ps(__m128i);`
`_mm_cvtpi32_pd`	SSE2	intrin.h	`__m128d _mm_cvtpi32_pd(__m64);`
`_mm_cvtps_epi32`	SSE2	intrin.h	`__m128i _mm_cvtps_epi32(__m128);`
`_mm_cvtps_pd`	SSE2	intrin.h	`__m128d _mm_cvtps_pd(__m128);`
`_mm_cvtps_ph`	F16C	immintrin.h	`__m128i _mm_cvtps_ph(__m128, const int);`
`_mm_cvtsd_f64`	SSSE3	intrin.h	`double _mm_cvtsd_f64(__m128d);`
`_mm_cvtsd_si32`	SSE2	intrin.h	`int _mm_cvtsd_si32(__m128d);`
`_mm_cvtsd_ss`	SSE2	intrin.h	`__m128 _mm_cvtsd_ss(__m128, __m128d);`
`_mm_cvtsi128_si32`	SSE2	intrin.h	`int _mm_cvtsi128_si32(__m128i);`
`_mm_cvtsi32_sd`	SSE2	intrin.h	`__m128d _mm_cvtsi32_sd(__m128d, int);`
`_mm_cvtsi32_si128`	SSE2	intrin.h	`__m128i _mm_cvtsi32_si128(int);`
`_mm_cvtsi32_si64`	MMX	mmintrin.h	`__m64 _mm_cvtsi32_si64(int);` [Macro]
`_mm_cvtsi64_si32`	MMX	mmintrin.h	`int _mm_cvtsi64_si32 (__m64);` [Macro]
`_mm_cvtss_f32`	SSSE3	intrin.h	`float _mm_cvtss_f32(__m128);`
`_mm_cvtss_sd`	SSE2	intrin.h	`__m128d _mm_cvtss_sd(__m128d, __m128);`
`_mm_cvtt_ps2pi`	SSE	intrin.h	`__m64 _mm_cvtt_ps2pi(__m128);`
`_mm_cvtt_ss2si`	SSE	intrin.h	`int _mm_cvtt_ss2si(__m128);`
`_mm_cvttpd_epi32`	SSE2	intrin.h	`__m128i _mm_cvttpd_epi32(__m128d);`
`_mm_cvttpd_pi32`	SSE2	intrin.h	`__m64 _mm_cvttpd_pi32(__m128d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cvttps_epi32`	SSE2	intrin.h	`__m128i _mm_cvttps_epi32(__m128);`
`_mm_cvttsd_si32`	SSE2	intrin.h	`int _mm_cvttsd_si32(__m128d);`
`_mm_div_pd`	SSE2	intrin.h	`__m128d _mm_div_pd(__m128d, __m128d);`
`_mm_div_ps`	SSE	intrin.h	`__m128 _mm_div_ps(__m128, __m128);`
`_mm_div_sd`	SSE2	intrin.h	`__m128d _mm_div_sd(__m128d, __m128d);`
`_mm_div_ss`	SSE	intrin.h	`__m128 _mm_div_ss(__m128, __m128);`
`_mm_dp_pd`	SSE41	intrin.h	`__m128d _mm_dp_pd(__m128d, __m128d, const int);`
`_mm_dp_ps`	SSE41	intrin.h	`__m128 _mm_dp_ps(__m128, __m128, const int);`
`_mm_empty`	MMX	mmintrin.h	`void _mm_empty (void);` [Macro]
`_mm_extract_epi16`	SSE2	intrin.h	`int _mm_extract_epi16(__m128i, int);`
`_mm_extract_epi32`	SSE41	intrin.h	`int _mm_extract_epi32(__m128i, const int);`
`_mm_extract_epi8`	SSE41	intrin.h	`int _mm_extract_epi8 (__m128i, const int);`
`_mm_extract_ps`	SSE41	intrin.h	`int _mm_extract_ps(__m128, const int);`
`_mm_extract_si64`	SSE4a	intrin.h	`__m128i _mm_extract_si64(__m128i, __m128i);`
`_mm_extracti_si64`	SSE4a	intrin.h	`__m128i _mm_extracti_si64(__m128i, int, int);`
`_mm_fmadd_pd`	FMA	immintrin.h	`__m128d _mm_fmadd_pd (__m128d, __m128d, __m128d);`
`_mm_fmadd_ps`	FMA	immintrin.h	`__m128 _mm_fmadd_ps (__m128, __m128, __m128);`
`_mm_fmadd_sd`	FMA	immintrin.h	`__m128d _mm_fmadd_sd (__m128d, __m128d, __m128d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_fmadd_ss`	FMA	immintrin.h	`__m128 _mm_fmadd_ss (__m128, __m128, __m128);`
`_mm_fmaddsub_pd`	FMA	immintrin.h	`__m128d _mm_fmaddsub_pd (__m128d, __m128d, __m128d);`
`_mm_fmaddsub_ps`	FMA	immintrin.h	`__m128 _mm_fmaddsub_ps (__m128, __m128, __m128);`
`_mm_fmsub_pd`	FMA	immintrin.h	`__m128d _mm_fmsub_pd (__m128d, __m128d, __m128d);`
`_mm_fmsub_ps`	FMA	immintrin.h	`__m128 _mm_fmsub_ps (__m128, __m128, __m128);`
`_mm_fmsub_sd`	FMA	immintrin.h	`__m128d _mm_fmsub_sd (__m128d, __m128d, __m128d);`
`_mm_fmsub_ss`	FMA	immintrin.h	`__m128 _mm_fmsub_ss (__m128, __m128, __m128);`
`_mm_fmsubadd_pd`	FMA	immintrin.h	`__m128d _mm_fmsubadd_pd (__m128d, __m128d, __m128d);`
`_mm_fmsubadd_ps`	FMA	immintrin.h	`__m128 _mm_fmsubadd_ps (__m128, __m128, __m128);`
`_mm_fnmadd_pd`	FMA	immintrin.h	`__m128d _mm_fnmadd_pd (__m128d, __m128d, __m128d);`
`_mm_fnmadd_ps`	FMA	immintrin.h	`__m128 _mm_fnmadd_ps (__m128, __m128, __m128);`
`_mm_fnmadd_sd`	FMA	immintrin.h	`__m128d _mm_fnmadd_sd (__m128d, __m128d, __m128d);`
`_mm_fnmadd_ss`	FMA	immintrin.h	`__m128 _mm_fnmadd_ss (__m128, __m128, __m128);`
`_mm_fnmsub_pd`	FMA	immintrin.h	`__m128d _mm_fnmsub_pd (__m128d, __m128d, __m128d);`
`_mm_fnmsub_ps`	FMA	immintrin.h	`__m128 _mm_fnmsub_ps (__m128, __m128, __m128);`
`_mm_fnmsub_sd`	FMA	immintrin.h	`__m128d _mm_fnmsub_sd (__m128d, __m128d, __m128d);`
`_mm_fnmsub_ss`	FMA	immintrin.h	`__m128 _mm_fnmsub_ss (__m128, __m128, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_frcz_pd`	XOP	ammintrin.h	`__m128d _mm_frcz_pd(__m128d);`
`_mm_frcz_ps`	XOP	ammintrin.h	`__m128 _mm_frcz_ps(__m128);`
`_mm_frcz_sd`	XOP	ammintrin.h	`__m128d _mm_frcz_sd(__m128d, __m128d);`
`_mm_frcz_ss`	XOP	ammintrin.h	`__m128 _mm_frcz_ss(__m128, __m128);`
`_mm_getcsr`	SSE	intrin.h	`unsigned int _mm_getcsr(void);`
`_mm_hadd_epi16`	SSSE3	intrin.h	`__m128i _mm_hadd_epi16(__m128i, __m128i);`
`_mm_hadd_epi32`	SSSE3	intrin.h	`__m128i _mm_hadd_epi32(__m128i, __m128i);`
`_mm_hadd_pd`	SSE3	intrin.h	`__m128d _mm_hadd_pd(__m128d, __m128d);`
`_mm_hadd_pi16`	SSSE3	intrin.h	`__m64 _mm_hadd_pi16(__m64, __m64);`
`_mm_hadd_pi32`	SSSE3	intrin.h	`__m64 _mm_hadd_pi32(__m64, __m64);`
`_mm_hadd_ps`	SSE3	intrin.h	`__m128 _mm_hadd_ps(__m128, __m128);`
`_mm_haddd_epi16`	XOP	ammintrin.h	`__m128i _mm_haddd_epi16(__m128i);`
`_mm_haddd_epi8`	XOP	ammintrin.h	`__m128i _mm_haddd_epi8(__m128i);`
`_mm_haddd_epu16`	XOP	ammintrin.h	`__m128i _mm_haddd_epu16(__m128i);`
`_mm_haddd_epu8`	XOP	ammintrin.h	`__m128i _mm_haddd_epu8(__m128i);`
`_mm_haddq_epi16`	XOP	ammintrin.h	`__m128i _mm_haddq_epi16(__m128i);`
`_mm_haddq_epi32`	XOP	ammintrin.h	`__m128i _mm_haddq_epi32(__m128i);`
`_mm_haddq_epi8`	XOP	ammintrin.h	`__m128i _mm_haddq_epi8(__m128i);`
`_mm_haddq_epu16`	XOP	ammintrin.h	`__m128i _mm_haddq_epu16(__m128i);`
`_mm_haddq_epu32`	XOP	ammintrin.h	`__m128i _mm_haddq_epu32(__m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_haddq_epu8`	XOP	ammintrin.h	`__m128i _mm_haddq_epu8(__m128i);`
`_mm_hadds_epi16`	SSSE3	intrin.h	`__m128i _mm_hadds_epi16(__m128i, __m128i);`
`_mm_hadds_pi16`	SSSE3	intrin.h	`__m64 _mm_hadds_pi16(__m64, __m64);`
`_mm_haddw_epi8`	XOP	ammintrin.h	`__m128i _mm_haddw_epi8(__m128i);`
`_mm_haddw_epu8`	XOP	ammintrin.h	`__m128i _mm_haddw_epu8(__m128i);`
`_mm_hsub_epi16`	SSSE3	intrin.h	`__m128i _mm_hsub_epi16(__m128i, __m128i);`
`_mm_hsub_epi32`	SSSE3	intrin.h	`__m128i _mm_hsub_epi32(__m128i, __m128i);`
`_mm_hsub_pd`	SSE3	intrin.h	`__m128d _mm_hsub_pd(__m128d, __m128d);`
`_mm_hsub_pi16`	SSSE3	intrin.h	`__m64 _mm_hsub_pi16(__m64, __m64);`
`_mm_hsub_pi32`	SSSE3	intrin.h	`__m64 _mm_hsub_pi32(__m64, __m64);`
`_mm_hsub_ps`	SSE3	intrin.h	`__m128 _mm_hsub_ps(__m128, __m128);`
`_mm_hsubd_epi16`	XOP	ammintrin.h	`__m128i _mm_hsubd_epi16(__m128i);`
`_mm_hsubq_epi32`	XOP	ammintrin.h	`__m128i _mm_hsubq_epi32(__m128i);`
`_mm_hsubs_epi16`	SSSE3	intrin.h	`__m128i _mm_hsubs_epi16(__m128i, __m128i);`
`_mm_hsubs_pi16`	SSSE3	intrin.h	`__m64 _mm_hsubs_pi16(__m64, __m64);`
`_mm_hsubw_epi8`	XOP	ammintrin.h	`__m128i _mm_hsubw_epi8(__m128i);`
`_mm_i32gather_epi32`	AVX2	immintrin.h	`__m128i _mm_i32gather_epi32(int const *, __m128i, const int);`
`_mm_i32gather_epi64`	AVX2	immintrin.h	`__m128i _mm_i32gather_epi64(__int64 const *, __m128i, const int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_i32gather_pd`	AVX2	immintrin.h	`__m128d _mm_i32gather_pd(double const *, __m128i, const int);`
`_mm_i32gather_ps`	AVX2	immintrin.h	`__m128 _mm_i32gather_ps(float const *, __m128i, const int);`
`_mm_i64gather_epi32`	AVX2	immintrin.h	`__m128i _mm_i64gather_epi32(int const *, __m128i, const int);`
`_mm_i64gather_epi64`	AVX2	immintrin.h	`__m128i _mm_i64gather_epi64(__int64 const *, __m128i, const int);`
`_mm_i64gather_pd`	AVX2	immintrin.h	`__m128d _mm_i64gather_pd(double const *, __m128i, const int);`
`_mm_i64gather_ps`	AVX2	immintrin.h	`__m128 _mm_i64gather_ps(float const *, __m128i, const int);`
`_mm_insert_epi16`	SSE2	intrin.h	`__m128i _mm_insert_epi16(__m128i, int, int);`
`_mm_insert_epi32`	SSE41	intrin.h	`__m128i _mm_insert_epi32(__m128i, int, const int);`
`_mm_insert_epi8`	SSE41	intrin.h	`__m128i _mm_insert_epi8 (__m128i, int, const int);`
`_mm_insert_ps`	SSE41	intrin.h	`__m128 _mm_insert_ps(__m128, __m128, const int);`
`_mm_insert_si64`	SSE4a	intrin.h	`__m128i _mm_insert_si64(__m128i, __m128i);`
`_mm_inserti_si64`	SSE4a	intrin.h	`__m128i _mm_inserti_si64(__m128i, __m128i, int, int);`
`_mm_lddqu_si128`	SSE3	intrin.h	`__m128i _mm_lddqu_si128(__m128i const*);`
`_mm_lfence`	SSE2	intrin.h	`void _mm_lfence(void);`
`_mm_load_pd`	SSE2	intrin.h	`__m128d _mm_load_pd(double*);`
`_mm_load_ps`	SSE	intrin.h	`__m128 _mm_load_ps(float*);`
`_mm_load_ps1`	SSE	intrin.h	`__m128 _mm_load_ps1(float*);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_load_sd`	SSE2	intrin.h	`__m128d _mm_load_sd(double*);`
`_mm_load_si128`	SSE2	intrin.h	`__m128i _mm_load_si128(__m128i*);`
`_mm_load_ss`	SSE	intrin.h	`__m128 _mm_load_ss(float*);`
`_mm_load1_pd`	SSE2	intrin.h	`__m128d _mm_load1_pd(double*);`
`_mm_loaddup_pd`	SSE3	intrin.h	`__m128d _mm_loaddup_pd(double const*);`
`_mm_loadh_pd`	SSE2	intrin.h	`__m128d _mm_loadh_pd(__m128d, double*);`
`_mm_loadh_pi`	SSE	intrin.h	`__m128 _mm_loadh_pi(__m128, __m64*);`
`_mm_loadl_epi64`	SSE2	intrin.h	`__m128i _mm_loadl_epi64(__m128i*);`
`_mm_loadl_pd`	SSE2	intrin.h	`__m128d _mm_loadl_pd(__m128d, double*);`
`_mm_loadl_pi`	SSE	intrin.h	`__m128 _mm_loadl_pi(__m128, __m64*);`
`_mm_loadr_pd`	SSE2	intrin.h	`__m128d _mm_loadr_pd(double*);`
`_mm_loadr_ps`	SSE	intrin.h	`__m128 _mm_loadr_ps(float*);`
`_mm_loadu_pd`	SSE2	intrin.h	`__m128d _mm_loadu_pd(double*);`
`_mm_loadu_ps`	SSE	intrin.h	`__m128 _mm_loadu_ps(float*);`
`_mm_loadu_si128`	SSE2	intrin.h	`__m128i _mm_loadu_si128(__m128i*);`
`_mm_macc_epi16`	XOP	ammintrin.h	`__m128i _mm_macc_epi16(__m128i, __m128i, __m128i);`
`_mm_macc_epi32`	XOP	ammintrin.h	`__m128i _mm_macc_epi32(__m128i, __m128i, __m128i);`
`_mm_macc_pd`	FMA4	ammintrin.h	`__m128d _mm_macc_pd(__m128d, __m128d, __m128d);`
`_mm_macc_ps`	FMA4	ammintrin.h	`__m128 _mm_macc_ps(__m128, __m128, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_macc_sd`	FMA4	ammintrin.h	`__m128d _mm_macc_sd(__m128d, __m128d, __m128d);`
`_mm_macc_ss`	FMA4	ammintrin.h	`__m128 _mm_macc_ss(__m128, __m128, __m128);`
`_mm_maccd_epi16`	XOP	ammintrin.h	`__m128i _mm_maccd_epi16(__m128i, __m128i, __m128i);`
`_mm_macchi_epi32`	XOP	ammintrin.h	`__m128i _mm_macchi_epi32(__m128i, __m128i, __m128i);`
`_mm_macclo_epi32`	XOP	ammintrin.h	`__m128i _mm_macclo_epi32(__m128i, __m128i, __m128i);`
`_mm_maccs_epi16`	XOP	ammintrin.h	`__m128i _mm_maccs_epi16(__m128i, __m128i, __m128i);`
`_mm_maccs_epi32`	XOP	ammintrin.h	`__m128i _mm_maccs_epi32(__m128i, __m128i, __m128i);`
`_mm_maccsd_epi16`	XOP	ammintrin.h	`__m128i _mm_maccsd_epi16(__m128i, __m128i, __m128i);`
`_mm_maccshi_epi32`	XOP	ammintrin.h	`__m128i _mm_maccshi_epi32(__m128i, __m128i, __m128i);`
`_mm_maccslo_epi32`	XOP	ammintrin.h	`__m128i _mm_maccslo_epi32(__m128i, __m128i, __m128i);`
`_mm_madd_epi16`	SSE2	intrin.h	`__m128i _mm_madd_epi16(__m128i, __m128i);`
`_mm_madd_pi16`	MMX	mmintrin.h	`__m64 _mm_madd_pi16(__m64, __m64);` [Macro]
`_mm_maddd_epi16`	XOP	ammintrin.h	`__m128i _mm_maddd_epi16(__m128i, __m128i, __m128i);`
`_mm_maddsd_epi16`	XOP	ammintrin.h	`__m128i _mm_maddsd_epi16(__m128i, __m128i, __m128i);`
`_mm_maddsub_pd`	FMA4	ammintrin.h	`__m128d _mm_maddsub_pd(__m128d, __m128d, __m128d);`
`_mm_maddsub_ps`	FMA4	ammintrin.h	`__m128 _mm_maddsub_ps(__m128, __m128, __m128);`
`_mm_maddubs_epi16`	SSSE3	intrin.h	`__m128i _mm_maddubs_epi16(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_maddubs_pi16`	SSSE3	intrin.h	`__m64 _mm_maddubs_pi16(__m64, __m64);`
`_mm_mask_i32gather_epi32`	AVX2	immintrin.h	`__m128i _mm_mask_i32gather_epi32(__m128i, int const *, __m128i, __m128i, const int);`
`_mm_mask_i32gather_epi64`	AVX2	immintrin.h	`__m128i _mm_mask_i32gather_epi64(__m128i, __int64 const *, __m128i, __m128i, const int);`
`_mm_mask_i32gather_pd`	AVX2	immintrin.h	`__m128d _mm_mask_i32gather_pd(__m128d, double const *, __m128i, __m128d, const int);`
`_mm_mask_i32gather_ps`	AVX2	immintrin.h	`__m128 _mm_mask_i32gather_ps(__m128, float const *, __m128i, __m128, const int);`
`_mm_mask_i64gather_epi32`	AVX2	immintrin.h	`__m128i _mm_mask_i64gather_epi32(__m128i, int const *, __m128i, __m128i, const int);`
`_mm_mask_i64gather_epi64`	AVX2	immintrin.h	`__m128i _mm_mask_i64gather_epi64(__m128i, __int64 const *, __m128i, __m128i, const int);`
`_mm_mask_i64gather_pd`	AVX2	immintrin.h	`__m128d _mm_mask_i64gather_pd(__m128d, double const *, __m128i, __m128d, const int);`
`_mm_mask_i64gather_ps`	AVX2	immintrin.h	`__m128 _mm_mask_i64gather_ps(__m128, float const *, __m128i, __m128, const int);`
`_mm_maskload_epi32`	AVX2	immintrin.h	`__m128i _mm_maskload_epi32(int const *, __m128i);`
`_mm_maskload_epi64`	AVX2	immintrin.h	`__m128i _mm_maskload_epi64(__int64 const *, __m128i);`
`_mm_maskload_pd`	AVX	immintrin.h	`__m128d _mm_maskload_pd(double const *, __m128i);`
`_mm_maskload_ps`	AVX	immintrin.h	`__m128 _mm_maskload_ps(float const *, __m128i);`
`_mm_maskmoveu_si128`	SSE2	intrin.h	`void _mm_maskmoveu_si128(__m128i, __m128i, char*);`
`_mm_maskstore_epi32`	AVX2	immintrin.h	`void _mm_maskstore_epi32(int *, __m128i, __m128i);`
`_mm_maskstore_epi64`	AVX2	immintrin.h	`void _mm_maskstore_epi64(__int64 *, __m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_maskstore_pd`	AVX	immintrin.h	`void _mm_maskstore_pd(double *, __m128i, __m128d);`
`_mm_maskstore_ps`	AVX	immintrin.h	`void _mm_maskstore_ps(float *, __m128i, __m128);`
`_mm_max_epi16`	SSE2	intrin.h	`__m128i _mm_max_epi16(__m128i, __m128i);`
`_mm_max_epi32`	SSE41	intrin.h	`__m128i _mm_max_epi32(__m128i, __m128i);`
`_mm_max_epi8`	SSE41	intrin.h	`__m128i _mm_max_epi8 (__m128i, __m128i);`
`_mm_max_epu16`	SSE41	intrin.h	`__m128i _mm_max_epu16(__m128i, __m128i);`
`_mm_max_epu32`	SSE41	intrin.h	`__m128i _mm_max_epu32(__m128i, __m128i);`
`_mm_max_epu8`	SSE2	intrin.h	`__m128i _mm_max_epu8(__m128i, __m128i);`
`_mm_max_pd`	SSE2	intrin.h	`__m128d _mm_max_pd(__m128d, __m128d);`
`_mm_max_ps`	SSE	intrin.h	`__m128 _mm_max_ps(__m128, __m128);`
`_mm_max_sd`	SSE2	intrin.h	`__m128d _mm_max_sd(__m128d, __m128d);`
`_mm_max_ss`	SSE	intrin.h	`__m128 _mm_max_ss(__m128, __m128);`
`_mm_mfence`	SSE2	intrin.h	`void _mm_mfence(void);`
`_mm_min_epi16`	SSE2	intrin.h	`__m128i _mm_min_epi16(__m128i, __m128i);`
`_mm_min_epi32`	SSE41	intrin.h	`__m128i _mm_min_epi32(__m128i, __m128i);`
`_mm_min_epi8`	SSE41	intrin.h	`__m128i _mm_min_epi8 (__m128i, __m128i);`
`_mm_min_epu16`	SSE41	intrin.h	`__m128i _mm_min_epu16(__m128i, __m128i);`
`_mm_min_epu32`	SSE41	intrin.h	`__m128i _mm_min_epu32(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_min_epu8`	SSE2	intrin.h	`__m128i _mm_min_epu8(__m128i, __m128i);`
`_mm_min_pd`	SSE2	intrin.h	`__m128d _mm_min_pd(__m128d, __m128d);`
`_mm_min_ps`	SSE	intrin.h	`__m128 _mm_min_ps(__m128, __m128);`
`_mm_min_sd`	SSE2	intrin.h	`__m128d _mm_min_sd(__m128d, __m128d);`
`_mm_min_ss`	SSE	intrin.h	`__m128 _mm_min_ss(__m128, __m128);`
`_mm_minpos_epu16`	SSE41	intrin.h	`__m128i _mm_minpos_epu16(__m128i);`
`_mm_monitor`	SSE3	intrin.h	`void _mm_monitor(void const*, unsigned int, unsigned int);`
`_mm_move_epi64`	SSE2	intrin.h	`__m128i _mm_move_epi64(__m128i);`
`_mm_move_sd`	SSE2	intrin.h	`__m128d _mm_move_sd(__m128d, __m128d);`
`_mm_move_ss`	SSE	intrin.h	`__m128 _mm_move_ss(__m128, __m128);`
`_mm_movedup_pd`	SSE3	intrin.h	`__m128d _mm_movedup_pd(__m128d);`
`_mm_movehdup_ps`	SSE3	intrin.h	`__m128 _mm_movehdup_ps(__m128);`
`_mm_movehl_ps`	SSE	intrin.h	`__m128 _mm_movehl_ps(__m128, __m128);`
`_mm_moveldup_ps`	SSE3	intrin.h	`__m128 _mm_moveldup_ps(__m128);`
`_mm_movelh_ps`	SSE	intrin.h	`__m128 _mm_movelh_ps(__m128, __m128);`
`_mm_movemask_epi8`	SSE2	intrin.h	`int _mm_movemask_epi8(__m128i);`
`_mm_movemask_pd`	SSE2	intrin.h	`int _mm_movemask_pd(__m128d);`
`_mm_movemask_ps`	SSE	intrin.h	`int _mm_movemask_ps(__m128);`
`_mm_movepi64_pi64`	SSE2	intrin.h	`__m64 _mm_movepi64_pi64(__m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_movpi64_epi64`	SSE2	intrin.h	`__m128i _mm_movpi64_epi64(__m64);`
`_mm_mpsadbw_epu8`	SSE41	intrin.h	`__m128i _mm_mpsadbw_epu8(__m128i, __m128i, const int);`
`_mm_msub_pd`	FMA4	ammintrin.h	`__m128d _mm_msub_pd(__m128d, __m128d, __m128d);`
`_mm_msub_ps`	FMA4	ammintrin.h	`__m128 _mm_msub_ps(__m128, __m128, __m128);`
`_mm_msub_sd`	FMA4	ammintrin.h	`__m128d _mm_msub_sd(__m128d, __m128d, __m128d);`
`_mm_msub_ss`	FMA4	ammintrin.h	`__m128 _mm_msub_ss(__m128, __m128, __m128);`
`_mm_msubadd_pd`	FMA4	ammintrin.h	`__m128d _mm_msubadd_pd(__m128d, __m128d, __m128d);`
`_mm_msubadd_ps`	FMA4	ammintrin.h	`__m128 _mm_msubadd_ps(__m128, __m128, __m128);`
`_mm_mul_epi32`	SSE41	intrin.h	`__m128i _mm_mul_epi32(__m128i, __m128i);`
`_mm_mul_epu32`	SSE2	intrin.h	`__m128i _mm_mul_epu32(__m128i, __m128i);`
`_mm_mul_pd`	SSE2	intrin.h	`__m128d _mm_mul_pd(__m128d, __m128d);`
`_mm_mul_ps`	SSE	intrin.h	`__m128 _mm_mul_ps(__m128, __m128);`
`_mm_mul_sd`	SSE2	intrin.h	`__m128d _mm_mul_sd(__m128d, __m128d);`
`_mm_mul_ss`	SSE	intrin.h	`__m128 _mm_mul_ss(__m128, __m128);`
`_mm_mul_su32`	SSE2	intrin.h	`__m64 _mm_mul_su32(__m64, __m64);`
`_mm_mulhi_epi16`	SSE2	intrin.h	`__m128i _mm_mulhi_epi16(__m128i, __m128i);`
`_mm_mulhi_epu16`	SSE2	intrin.h	`__m128i _mm_mulhi_epu16(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_mulhi_pi16`	MMX	mmintrin.h	`__m64 _mm_mulhi_pi16(__m64, __m64);` [Macro]
`_mm_mulhrs_epi16`	SSSE3	intrin.h	`__m128i _mm_mulhrs_epi16(__m128i, __m128i);`
`_mm_mulhrs_pi16`	SSSE3	intrin.h	`__m64 _mm_mulhrs_pi16(__m64, __m64);`
`_mm_mullo_epi16`	SSE2	intrin.h	`__m128i _mm_mullo_epi16(__m128i, __m128i);`
`_mm_mullo_epi32`	SSE41	intrin.h	`__m128i _mm_mullo_epi32(__m128i, __m128i);`
`_mm_mullo_pi16`	MMX	mmintrin.h	`__m64 _mm_mullo_pi16(__m64, __m64);` [Macro]
`_mm_mwait`	SSE3	intrin.h	`void _mm_mwait(unsigned int, unsigned int);`
`_mm_nmacc_pd`	FMA4	ammintrin.h	`__m128d _mm_nmacc_pd(__m128d, __m128d, __m128d);`
`_mm_nmacc_ps`	FMA4	ammintrin.h	`__m128 _mm_nmacc_ps(__m128, __m128, __m128);`
`_mm_nmacc_sd`	FMA4	ammintrin.h	`__m128d _mm_nmacc_sd(__m128d, __m128d, __m128d);`
`_mm_nmacc_ss`	FMA4	ammintrin.h	`__m128 _mm_nmacc_ss(__m128, __m128, __m128);`
`_mm_nmsub_pd`	FMA4	ammintrin.h	`__m128d _mm_nmsub_pd(__m128d, __m128d, __m128d);`
`_mm_nmsub_ps`	FMA4	ammintrin.h	`__m128 _mm_nmsub_ps(__m128, __m128, __m128);`
`_mm_nmsub_sd`	FMA4	ammintrin.h	`__m128d _mm_nmsub_sd(__m128d, __m128d, __m128d);`
`_mm_nmsub_ss`	FMA4	ammintrin.h	`__m128 _mm_nmsub_ss(__m128, __m128, __m128);`
`_mm_or_pd`	SSE2	intrin.h	`__m128d _mm_or_pd(__m128d, __m128d);`
`_mm_or_ps`	SSE	intrin.h	`__m128 _mm_or_ps(__m128, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_or_si64`	MMX	mmintrin.h	`__m64 _mm_or_si64(__m64, __m64);` [Macro]
`_mm_or_si128`	SSE2	intrin.h	`__m128i _mm_or_si128(__m128i, __m128i);`
`_mm_packs_epi16`	SSE2	intrin.h	`__m128i _mm_packs_epi16(__m128i, __m128i);`
`_mm_packs_epi32`	SSE2	intrin.h	`__m128i _mm_packs_epi32(__m128i, __m128i);`
`_mm_packs_pi16`	MMX	mmintrin.h	`__m64 _mm_packs_pi16 (__m64, __m64);` [Macro]
`_mm_packs_pi32`	MMX	mmintrin.h	`__m64 _mm_packs_pi32 (__m64, __m64);` [Macro]
`_mm_packs_pu16`	MMX	mmintrin.h	`__m64 _mm_packs_pu16 (__m64, __m64);` [Macro]
`_mm_packus_epi16`	SSE2	intrin.h	`__m128i _mm_packus_epi16(__m128i, __m128i);`
`_mm_packus_epi32`	SSE41	intrin.h	`__m128i _mm_packus_epi32(__m128i, __m128i);`
`_mm_pause`	SSE2	intrin.h	`void _mm_pause(void);`
`_mm_perm_epi8`	XOP	ammintrin.h	`__m128i _mm_perm_epi8(__m128i, __m128i, __m128i);`
`_mm_permute_pd`	AVX	immintrin.h	`__m128d _mm_permute_pd(__m128d, int);`
`_mm_permute_ps`	AVX	immintrin.h	`__m128 _mm_permute_ps(__m128, int);`
`_mm_permute2_pd`	XOP	ammintrin.h	`__m128d _mm_permute2_pd(__m128d, __m128d, __m128i, int);`
`_mm_permute2_ps`	XOP	ammintrin.h	`__m128 _mm_permute2_ps(__m128, __m128, __m128i, int);`
`_mm_permutevar_pd`	AVX	immintrin.h	`__m128d _mm_permutevar_pd(__m128d, __m128i);`
`_mm_permutevar_ps`	AVX	immintrin.h	`__m128 _mm_permutevar_ps(__m128, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_popcnt_u32`	POPCNT	intrin.h	`int _mm_popcnt_u32(unsigned int);`
`_mm_prefetch`	SSE	intrin.h	`void _mm_prefetch(char*, int);`
`_mm_rcp_ps`	SSE	intrin.h	`__m128 _mm_rcp_ps(__m128);`
`_mm_rcp_ss`	SSE	intrin.h	`__m128 _mm_rcp_ss(__m128);`
`_mm_rot_epi16`	XOP	ammintrin.h	`__m128i _mm_rot_epi16(__m128i, __m128i);`
`_mm_rot_epi32`	XOP	ammintrin.h	`__m128i _mm_rot_epi32(__m128i, __m128i);`
`_mm_rot_epi64`	XOP	ammintrin.h	`__m128i _mm_rot_epi64(__m128i, __m128i);`
`_mm_rot_epi8`	XOP	ammintrin.h	`__m128i _mm_rot_epi8(__m128i, __m128i);`
`_mm_roti_epi16`	XOP	ammintrin.h	`__m128i _mm_rot_epi16(__m128i, int);`
`_mm_roti_epi32`	XOP	ammintrin.h	`__m128i _mm_rot_epi32(__m128i, int);`
`_mm_roti_epi64`	XOP	ammintrin.h	`__m128i _mm_rot_epi64(__m128i, int);`
`_mm_roti_epi8`	XOP	ammintrin.h	`__m128i _mm_rot_epi8(__m128i, int);`
`_mm_round_pd`	SSE41	intrin.h	`__m128d _mm_round_pd(__m128d, const int);`
`_mm_round_ps`	SSE41	intrin.h	`__m128 _mm_round_ps(__m128, const int);`
`_mm_round_sd`	SSE41	intrin.h	`__m128d _mm_round_sd(__m128d, __m128d, const int);`
`_mm_round_ss`	SSE41	intrin.h	`__m128 _mm_round_ss(__m128, __m128, const int);`
`_mm_rsqrt_ps`	SSE	intrin.h	`__m128 _mm_rsqrt_ps(__m128);`
`_mm_rsqrt_ss`	SSE	intrin.h	`__m128 _mm_rsqrt_ss(__m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_sad_epu8`	SSE2	intrin.h	`__m128i _mm_sad_epu8(__m128i, __m128i);`
`_mm_set_epi16`	SSE2	intrin.h	`__m128i _mm_set_epi16(short, short, short, short, short, short, short, short);`
`_mm_set_epi32`	SSE2	intrin.h	`__m128i _mm_set_epi32(int, int, int, int);`
`_mm_set_epi64`	SSE2	intrin.h	`__m128i _mm_set_epi64(__m64, __m64);`
`_mm_set_epi8`	SSE2	intrin.h	`__m128i _mm_set_epi8(char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char);`
`_mm_set_pd`	SSE2	intrin.h	`__m128d _mm_set_pd(double, double);`
`_mm_set_pi16`	MMX	intrin.h	`__m64 _mm_set_pi16(short, short, short, short);`
`_mm_set_pi32`	MMX	intrin.h	`__m64 _mm_set_pi32(int, int);`
`_mm_set_pi8`	MMX	intrin.h	`__m64 _mm_set_pi8(char, char, char, char, char, char, char, char);`
`_mm_set_ps`	SSE	intrin.h	`__m128 _mm_set_ps(float, float, float, float);`
`_mm_set_ps1`	SSE	intrin.h	`__m128 _mm_set_ps1(float);`
`_mm_set_sd`	SSE2	intrin.h	`__m128d _mm_set_sd(double);`
`_mm_set_ss`	SSE	intrin.h	`__m128 _mm_set_ss(float);`
`_mm_set1_epi16`	SSE2	intrin.h	`__m128i _mm_set1_epi16(short);`
`_mm_set1_epi32`	SSE2	intrin.h	`__m128i _mm_set1_epi32(int);`
`_mm_set1_epi64`	SSE2	intrin.h	`__m128i _mm_set1_epi64(__m64);`
`_mm_set1_epi8`	SSE2	intrin.h	`__m128i _mm_set1_epi8(char);`
`_mm_set1_pd`	SSE2	intrin.h	`__m128d _mm_set1_pd(double);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_set1_pi16`	MMX	intrin.h	`__m64 _mm_set1_pi16(short);`
`_mm_set1_pi32`	MMX	intrin.h	`__m64 _mm_set1_pi32(int);`
`_mm_set1_pi8`	MMX	intrin.h	`__m64 _mm_set1_pi8(char);`
`_mm_setcsr`	SSE	intrin.h	`void _mm_setcsr(unsigned int);`
`_mm_setl_epi64`	SSE2	intrin.h	`__m128i _mm_setl_epi64(__m128i);`
`_mm_setr_epi16`	SSE2	intrin.h	`__m128i _mm_setr_epi16(short, short, short, short, short, short, short, short);`
`_mm_setr_epi32`	SSE2	intrin.h	`__m128i _mm_setr_epi32(int, int, int, int);`
`_mm_setr_epi64`	SSE2	intrin.h	`__m128i _mm_setr_epi64(__m64, __m64);`
`_mm_setr_epi8`	SSE2	intrin.h	`__m128i _mm_setr_epi8(char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char);`
`_mm_setr_pd`	SSE2	intrin.h	`__m128d _mm_setr_pd(double, double);`
`_mm_setr_pi16`	MMX	intrin.h	`__m64 _mm_setr_pi16(short, short, short, short);`
`_mm_setr_pi32`	MMX	intrin.h	`__m64 _mm_setr_pi32(int, int);`
`_mm_setr_pi8`	MMX	intrin.h	`__m64 _mm_setr_pi8(char, char, char, char, char, char, char, char);`
`_mm_setr_ps`	SSE	intrin.h	`__m128 _mm_setr_ps(float, float, float, float);`
`_mm_setzero_pd`	SSE2	intrin.h	`__m128d _mm_setzero_pd(void);`
`_mm_setzero_ps`	SSE	intrin.h	`__m128 _mm_setzero_ps(void);`
`_mm_setzero_si128`	SSE2	intrin.h	`__m128i _mm_setzero_si128(void);`
`_mm_setzero_si64`	MMX	intrin.h	`__m64 _mm_setzero_si64(void);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_sfence`	SSE	intrin.h	`void _mm_sfence(void);`
`_mm_sha_epi16`	XOP	ammintrin.h	`__m128i _mm_sha_epi16(__m128i, __m128i);`
`_mm_sha_epi32`	XOP	ammintrin.h	`__m128i _mm_sha_epi32(__m128i, __m128i);`
`_mm_sha_epi64`	XOP	ammintrin.h	`__m128i _mm_sha_epi64(__m128i, __m128i);`
`_mm_sha_epi8`	XOP	ammintrin.h	`__m128i _mm_sha_epi8(__m128i, __m128i);`
`_mm_shl_epi16`	XOP	ammintrin.h	`__m128i _mm_shl_epi16(__m128i, __m128i);`
`_mm_shl_epi32`	XOP	ammintrin.h	`__m128i _mm_shl_epi32(__m128i, __m128i);`
`_mm_shl_epi64`	XOP	ammintrin.h	`__m128i _mm_shl_epi64(__m128i, __m128i);`
`_mm_shl_epi8`	XOP	ammintrin.h	`__m128i _mm_shl_epi8(__m128i, __m128i);`
`_mm_shuffle_epi32`	SSE2	intrin.h	`__m128i _mm_shuffle_epi32(__m128i, int);`
`_mm_shuffle_epi8`	SSSE3	intrin.h	`__m128i _mm_shuffle_epi8(__m128i, __m128i);`
`_mm_shuffle_pd`	SSE2	intrin.h	`__m128d _mm_shuffle_pd(__m128d, __m128d, int);`
`_mm_shuffle_pi8`	SSSE3	intrin.h	`__m64 _mm_shuffle_pi8(__m64, __m64);`
`_mm_shuffle_ps`	SSE	intrin.h	`__m128 _mm_shuffle_ps(__m128, __m128, unsigned int);`
`_mm_shufflehi_epi16`	SSE2	intrin.h	`__m128i _mm_shufflehi_epi16(__m128i, int);`
`_mm_shufflelo_epi16`	SSE2	intrin.h	`__m128i _mm_shufflelo_epi16(__m128i, int);`
`_mm_sign_epi16`	SSSE3	intrin.h	`__m128i _mm_sign_epi16(__m128i, __m128i);`
`_mm_sign_epi32`	SSSE3	intrin.h	`__m128i _mm_sign_epi32(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_sign_epi8`	SSSE3	intrin.h	`__m128i _mm_sign_epi8(__m128i, __m128i);`
`_mm_sign_pi16`	SSSE3	intrin.h	`__m64 _mm_sign_pi16(__m64, __m64);`
`_mm_sign_pi32`	SSSE3	intrin.h	`__m64 _mm_sign_pi32(__m64, __m64);`
`_mm_sign_pi8`	SSSE3	intrin.h	`__m64 _mm_sign_pi8(__m64, __m64);`
`_mm_sll_epi16`	SSE2	intrin.h	`__m128i _mm_sll_epi16(__m128i, __m128i);`
`_mm_sll_epi32`	SSE2	intrin.h	`__m128i _mm_sll_epi32(__m128i, __m128i);`
`_mm_sll_epi64`	SSE2	intrin.h	`__m128i _mm_sll_epi64(__m128i, __m128i);`
`_mm_sll_pi16`	MMX	mmintrin.h	`__m64 _mm_sll_pi16(__m64, __m64);` [Macro]
`_mm_sll_pi32`	MMX	mmintrin.h	`__m64 _mm_sll_pi32(__m64, __m64);` [Macro]
`_mm_sll_si64`	MMX	mmintrin.h	`__m64 _mm_sll_si64(__m64, __m64);` [Macro]
`_mm_slli_epi16`	SSE2	intrin.h	`__m128i _mm_slli_epi16(__m128i, int);`
`_mm_slli_epi32`	SSE2	intrin.h	`__m128i _mm_slli_epi32(__m128i, int);`
`_mm_slli_epi64`	SSE2	intrin.h	`__m128i _mm_slli_epi64(__m128i, int);`
`_mm_slli_pi16`	MMX	mmintrin.h	`__m64 _mm_slli_pi16(__m64, int);` [Macro]
`_mm_slli_pi32`	MMX	mmintrin.h	`__m64 _mm_slli_pi32(__m64, int);` [Macro]
`_mm_slli_si64`	MMX	mmintrin.h	`__m64 _mm_slli_si64(__m64, int);` [Macro]

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_slli_si128`	SSE2	intrin.h	`__m128i _mm_slli_si128(__m128i, int);`
`_mm_sllv_epi32`	AVX2	immintrin.h	`__m128i _mm_sllv_epi32(__m128i, __m128i);`
`_mm_sllv_epi64`	AVX2	immintrin.h	`__m128i _mm_sllv_epi64(__m128i, __m128i);`
`_mm_sqrt_pd`	SSE2	intrin.h	`__m128d _mm_sqrt_pd(__m128d);`
`_mm_sqrt_ps`	SSE	intrin.h	`__m128 _mm_sqrt_ps(__m128);`
`_mm_sqrt_sd`	SSE2	intrin.h	`__m128d _mm_sqrt_sd(__m128d, __m128d);`
`_mm_sqrt_ss`	SSE	intrin.h	`__m128 _mm_sqrt_ss(__m128);`
`_mm_sra_epi16`	SSE2	intrin.h	`__m128i _mm_sra_epi16(__m128i, __m128i);`
`_mm_sra_epi32`	SSE2	intrin.h	`__m128i _mm_sra_epi32(__m128i, __m128i);`
`_mm_sra_pi16`	MMX	mmintrin.h	`__m64 _mm_sra_pi16(__m64, __m64);` [Macro]
`_mm_sra_pi32`	MMX	mmintrin.h	`__m64 _mm_sra_pi32(__m64, __m64);` [Macro]
`_mm_srai_epi16`	SSE2	intrin.h	`__m128i _mm_srai_epi16(__m128i, int);`
`_mm_srai_epi32`	SSE2	intrin.h	`__m128i _mm_srai_epi32(__m128i, int);`
`_mm_srai_pi16`	MMX	mmintrin.h	`__m64 _mm_srai_pi16(__m64, int);` [Macro]
`_mm_srai_pi32`	MMX	mmintrin.h	`__m64 _mm_srai_pi32(__m64, int);` [Macro]
`_mm_srav_epi32`	AVX2	immintrin.h	`__m128i _mm_srav_epi32(__m128i, __m128i);`
`_mm_srl_epi16`	SSE2	intrin.h	`__m128i _mm_srl_epi16(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_srl_epi32`	SSE2	intrin.h	`__m128i _mm_srl_epi32(__m128i, __m128i);`
`_mm_srl_epi64`	SSE2	intrin.h	`__m128i _mm_srl_epi64(__m128i, __m128i);`
`_mm_srl_pi16`	MMX	mmintrin.h	`__m64 _mm_srl_pi16(__m64, __m64);` [Macro]
`_mm_srl_pi32`	MMX	mmintrin.h	`__m64 _mm_srl_pi32(__m64, __m64);` [Macro]
`_mm_srl_si64`	MMX	mmintrin.h	`__m64 _mm_srl_si64(__m64, __m64);` [Macro]
`_mm_srli_epi16`	SSE2	intrin.h	`__m128i _mm_srli_epi16(__m128i, int);`
`_mm_srli_epi32`	SSE2	intrin.h	`__m128i _mm_srli_epi32(__m128i, int);`
`_mm_srli_epi64`	SSE2	intrin.h	`__m128i _mm_srli_epi64(__m128i, int);`
`_mm_srli_pi16`	MMX	mmintrin.h	`__m64 _mm_srli_pi16(__m64, int);` [Macro]
`_mm_srli_pi32`	MMX	mmintrin.h	`__m64 _mm_srli_pi32(__m64, int);` [Macro]
`_mm_srli_si64`	MMX	mmintrin.h	`__m64 _mm_srli_si64(__m64, int);` [Macro]
`_mm_srli_si128`	SSE2	intrin.h	`__m128i _mm_srli_si128(__m128i, int);`
`_mm_srlv_epi32`	AVX2	immintrin.h	`__m128i _mm_srlv_epi32(__m128i, __m128i);`
`_mm_srlv_epi64`	AVX2	immintrin.h	`__m128i _mm_srlv_epi64(__m128i, __m128i);`
`_mm_store_pd`	SSE2	intrin.h	`void _mm_store_pd(double*, __m128d);`
`_mm_store_ps`	SSE	intrin.h	`void _mm_store_ps(float*, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_store_ps1`	SSE	intrin.h	`void _mm_store_ps1(float*, __m128);`
`_mm_store_sd`	SSE2	intrin.h	`void _mm_store_sd(double*, __m128d);`
`_mm_store_si128`	SSE2	intrin.h	`void _mm_store_si128(__m128i*, __m128i);`
`_mm_store_ss`	SSE	intrin.h	`void _mm_store_ss(float*, __m128);`
`_mm_store1_pd`	SSE2	intrin.h	`void _mm_store1_pd(double*, __m128d);`
`_mm_storeh_pd`	SSE2	intrin.h	`void _mm_storeh_pd(double*, __m128d);`
`_mm_storeh_pi`	SSE	intrin.h	`void _mm_storeh_pi(__m64*, __m128);`
`_mm_storel_epi64`	SSE2	intrin.h	`void _mm_storel_epi64(__m128i*, __m128i);`
`_mm_storel_pd`	SSE2	intrin.h	`void _mm_storel_pd(double*, __m128d);`
`_mm_storel_pi`	SSE	intrin.h	`void _mm_storel_pi(__m64*, __m128);`
`_mm_storer_pd`	SSE2	intrin.h	`void _mm_storer_pd(double*, __m128d);`
`_mm_storer_ps`	SSE	intrin.h	`void _mm_storer_ps(float*, __m128);`
`_mm_storeu_pd`	SSE2	intrin.h	`void _mm_storeu_pd(double*, __m128d);`
`_mm_storeu_ps`	SSE	intrin.h	`void _mm_storeu_ps(float*, __m128);`
`_mm_storeu_si128`	SSE2	intrin.h	`void _mm_storeu_si128(__m128i*, __m128i);`
`_mm_stream_load_si128`	SSE41	intrin.h	`__m128i _mm_stream_load_si128(__m128i*);`
`_mm_stream_pd`	SSE2	intrin.h	`void _mm_stream_pd(double*, __m128d);`
`_mm_stream_pi`	SSE	intrin.h	`void _mm_stream_pi(__m64*, __m64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_stream_ps`	SSE	intrin.h	`void _mm_stream_ps(float*, __m128);`
`_mm_stream_sd`	SSE4a	intrin.h	`void _mm_stream_sd(double*, __m128d);`
`_mm_stream_si128`	SSE2	intrin.h	`void _mm_stream_si128(__m128i*, __m128i);`
`_mm_stream_si32`	SSE2	intrin.h	`void _mm_stream_si32(int*, int);`
`_mm_stream_ss`	SSE4a	intrin.h	`void _mm_stream_ss(float*, __m128);`
`_mm_sub_epi16`	SSE2	intrin.h	`__m128i _mm_sub_epi16(__m128i, __m128i);`
`_mm_sub_epi32`	SSE2	intrin.h	`__m128i _mm_sub_epi32(__m128i, __m128i);`
`_mm_sub_epi64`	SSE2	intrin.h	`__m128i _mm_sub_epi64(__m128i, __m128i);`
`_mm_sub_epi8`	SSE2	intrin.h	`__m128i _mm_sub_epi8(__m128i, __m128i);`
`_mm_sub_pd`	SSE2	intrin.h	`__m128d _mm_sub_pd(__m128d, __m128d);`
`_mm_sub_pi8`	MMX	mmintrin.h	`__m64 _mm_sub_pi8(__m64, __m64);` [Macro]
`_mm_sub_pi16`	MMX	mmintrin.h	`__m64 _mm_sub_pi16(__m64, __m64);` [Macro]
`_mm_sub_pi32`	MMX	mmintrin.h	`__m64 _mm_sub_pi32(__m64, __m64);` [Macro]
`_mm_sub_ps`	SSE	intrin.h	`__m128 _mm_sub_ps(__m128, __m128);`
`_mm_sub_sd`	SSE2	intrin.h	`__m128d _mm_sub_sd(__m128d, __m128d);`
`_mm_sub_si64`	SSE2	intrin.h	`__m64 _mm_sub_si64(__m64, __m64);`
`_mm_sub_ss`	SSE	intrin.h	`__m128 _mm_sub_ss(__m128, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_subs_epi16`	SSE2	intrin.h	`__m128i _mm_subs_epi16(__m128i, __m128i);`
`_mm_subs_epi8`	SSE2	intrin.h	`__m128i _mm_subs_epi8(__m128i, __m128i);`
`_mm_subs_epu16`	SSE2	intrin.h	`__m128i _mm_subs_epu16(__m128i, __m128i);`
`_mm_subs_epu8`	SSE2	intrin.h	`__m128i _mm_subs_epu8(__m128i, __m128i);`
`_mm_subs_pi8`	MMX	mmintrin.h	`__m64 _mm_subs_pi8(__m64, __m64);` [Macro]
`_mm_subs_pi16`	MMX	mmintrin.h	`__m64 _mm_subs_pi16(__m64, __m64);` [Macro]
`_mm_subs_pu8`	MMX	mmintrin.h	`__m64 _mm_subs_pu8(__m64, __m64);` [Macro]
`_mm_subs_pu16`	MMX	mmintrin.h	`__m64 _mm_subs_pu16(__m64, __m64);` [Macro]
`_mm_testc_pd`	AVX	immintrin.h	`int _mm_testc_pd(__m128d, __m128d);`
`_mm_testc_ps`	AVX	immintrin.h	`int _mm_testc_ps(__m128, __m128);`
`_mm_testc_si128`	SSE41	intrin.h	`int _mm_testc_si128(__m128i, __m128i);`
`_mm_testnzc_pd`	AVX	immintrin.h	`int _mm_testnzc_pd(__m128d, __m128d);`
`_mm_testnzc_ps`	AVX	immintrin.h	`int _mm_testnzc_ps(__m128, __m128);`
`_mm_testnzc_si128`	SSE41	intrin.h	`int _mm_testnzc_si128(__m128i, __m128i);`
`_mm_testz_pd`	AVX	immintrin.h	`int _mm_testz_pd(__m128d, __m128d);`
`_mm_testz_ps`	AVX	immintrin.h	`int _mm_testz_ps(__m128, __m128);`
`_mm_testz_si128`	SSE41	intrin.h	`int _mm_testz_si128(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_ucomieq_sd`	SSE2	intrin.h	`int _mm_ucomieq_sd(__m128d, __m128d);`
`_mm_ucomieq_ss`	SSE	intrin.h	`int _mm_ucomieq_ss(__m128, __m128);`
`_mm_ucomige_sd`	SSE2	intrin.h	`int _mm_ucomige_sd(__m128d, __m128d);`
`_mm_ucomige_ss`	SSE	intrin.h	`int _mm_ucomige_ss(__m128, __m128);`
`_mm_ucomigt_sd`	SSE2	intrin.h	`int _mm_ucomigt_sd(__m128d, __m128d);`
`_mm_ucomigt_ss`	SSE	intrin.h	`int _mm_ucomigt_ss(__m128, __m128);`
`_mm_ucomile_sd`	SSE2	intrin.h	`int _mm_ucomile_sd(__m128d, __m128d);`
`_mm_ucomile_ss`	SSE	intrin.h	`int _mm_ucomile_ss(__m128, __m128);`
`_mm_ucomilt_sd`	SSE2	intrin.h	`int _mm_ucomilt_sd(__m128d, __m128d);`
`_mm_ucomilt_ss`	SSE	intrin.h	`int _mm_ucomilt_ss(__m128, __m128);`
`_mm_ucomineq_sd`	SSE2	intrin.h	`int _mm_ucomineq_sd(__m128d, __m128d);`
`_mm_ucomineq_ss`	SSE	intrin.h	`int _mm_ucomineq_ss(__m128, __m128);`
`_mm_unpackhi_epi16`	SSE2	intrin.h	`__m128i _mm_unpackhi_epi16(__m128i, __m128i);`
`_mm_unpackhi_epi32`	SSE2	intrin.h	`__m128i _mm_unpackhi_epi32(__m128i, __m128i);`
`_mm_unpackhi_epi64`	SSE2	intrin.h	`__m128i _mm_unpackhi_epi64(__m128i, __m128i);`
`_mm_unpackhi_epi8`	SSE2	intrin.h	`__m128i _mm_unpackhi_epi8(__m128i, __m128i);`
`_mm_unpackhi_pd`	SSE2	intrin.h	`__m128d _mm_unpackhi_pd(__m128d, __m128d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_unpackhi_pi8`	MMX	mmintrin.h	`__m64 _mm_unpackhi_pi8 (__m64, __m64);` [Macro]
`_mm_unpackhi_pi16`	MMX	mmintrin.h	`__m64 _mm_unpackhi_pi16 (__m64, __m64);` [Macro]
`_mm_unpackhi_pi32`	MMX	mmintrin.h	`__m64 _mm_unpackhi_pi32 (__m64, __m64);` [Macro]
`_mm_unpackhi_ps`	SSE	intrin.h	`__m128 _mm_unpackhi_ps(__m128, __m128);`
`_mm_unpacklo_epi16`	SSE2	intrin.h	`__m128i _mm_unpacklo_epi16(__m128i, __m128i);`
`_mm_unpacklo_epi32`	SSE2	intrin.h	`__m128i _mm_unpacklo_epi32(__m128i, __m128i);`
`_mm_unpacklo_epi64`	SSE2	intrin.h	`__m128i _mm_unpacklo_epi64(__m128i, __m128i);`
`_mm_unpacklo_epi8`	SSE2	intrin.h	`__m128i _mm_unpacklo_epi8(__m128i, __m128i);`
`_mm_unpacklo_pd`	SSE2	intrin.h	`__m128d _mm_unpacklo_pd(__m128d, __m128d);`
`_mm_unpacklo_pi8`	MMX	mmintrin.h	`__m64 _mm_unpacklo_pi8 (__m64, __m64);` [Macro]
`_mm_unpacklo_pi16`	MMX	mmintrin.h	`__m64 _mm_unpacklo_pi16 (__m64, __m64);` [Macro]
`_mm_unpacklo_pi32`	MMX	mmintrin.h	`__m64 _mm_unpacklo_pi32 (__m64, __m64);` [Macro]
`_mm_unpacklo_ps`	SSE	intrin.h	`__m128 _mm_unpacklo_ps(__m128, __m128);`
`_mm_xor_pd`	SSE2	intrin.h	`__m128d _mm_xor_pd(__m128d, __m128d);`
`_mm_xor_ps`	SSE	intrin.h	`__m128 _mm_xor_ps(__m128, __m128);`
`_mm_xor_si64`	MMX	mmintrin.h	`__m64 _mm_xor_si64(__m64, __m64);` [Macro]

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_xor_si128`	SSE2	intrin.h	`__m128i _mm_xor_si128(__m128i, __m128i);`
`_mm256_abs_epi16`	AVX2	immintrin.h	`__m256i _mm256_abs_epi16(__m256i);`
`_mm256_abs_epi32`	AVX2	immintrin.h	`__m256i _mm256_abs_epi32(__m256i);`
`_mm256_abs_epi8`	AVX2	immintrin.h	`__m256i _mm256_abs_epi8(__m256i);`
`_mm256_add_epi16`	AVX2	immintrin.h	`__m256i _mm256_add_epi16(__m256i, __m256i);`
`_mm256_add_epi32`	AVX2	immintrin.h	`__m256i _mm256_add_epi32(__m256i, __m256i);`
`_mm256_add_epi64`	AVX2	immintrin.h	`__m256i _mm256_add_epi64(__m256i, __m256i);`
`_mm256_add_epi8`	AVX2	immintrin.h	`__m256i _mm256_add_epi8(__m256i, __m256i);`
`_mm256_add_pd`	AVX	immintrin.h	`__m256d _mm256_add_pd(__m256d, __m256d);`
`_mm256_add_ps`	AVX	immintrin.h	`__m256 _mm256_add_ps(__m256, __m256);`
`_mm256_adds_epi16`	AVX2	immintrin.h	`__m256i _mm256_adds_epi16(__m256i, __m256i);`
`_mm256_adds_epi8`	AVX2	immintrin.h	`__m256i _mm256_adds_epi8(__m256i, __m256i);`
`_mm256_adds_epu16`	AVX2	immintrin.h	`__m256i _mm256_adds_epu16(__m256i, __m256i);`
`_mm256_adds_epu8`	AVX2	immintrin.h	`__m256i _mm256_adds_epu8(__m256i, __m256i);`
`_mm256_addsub_pd`	AVX	immintrin.h	`__m256d _mm256_addsub_pd(__m256d, __m256d);`
`_mm256_addsub_ps`	AVX	immintrin.h	`__m256 _mm256_addsub_ps(__m256, __m256);`
`_mm256_alignr_epi8`	AVX2	immintrin.h	`__m256i _mm256_alignr_epi8(__m256i, __m256i, const int);`
`_mm256_and_pd`	AVX	immintrin.h	`__m256d _mm256_and_pd(__m256d, __m256d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_and_ps`	AVX	immintrin.h	`__m256 _mm256_and_ps(__m256, __m256);`
`_mm256_and_si256`	AVX2	immintrin.h	`__m256i _mm256_and_si256(__m256i, __m256i);`
`_mm256_andnot_pd`	AVX	immintrin.h	`__m256d _mm256_andnot_pd(__m256d, __m256d);`
`_mm256_andnot_ps`	AVX	immintrin.h	`__m256 _mm256_andnot_ps(__m256, __m256);`
`_mm256_andnot_si256`	AVX2	immintrin.h	`__m256i _mm256_andnot_si256(__m256i, __m256i);`
`_mm256_avg_epu16`	AVX2	immintrin.h	`__m256i _mm256_avg_epu16(__m256i, __m256i);`
`_mm256_avg_epu8`	AVX2	immintrin.h	`__m256i _mm256_avg_epu8(__m256i, __m256i);`
`_mm256_blend_epi16`	AVX2	immintrin.h	`__m256i _mm256_blend_epi16(__m256i, __m256i, const int);`
`_mm256_blend_epi32`	AVX2	immintrin.h	`__m256i _mm256_blend_epi32(__m256i, __m256i, const int);`
`_mm256_blend_pd`	AVX	immintrin.h	`__m256d _mm256_blend_pd(__m256d, __m256d, const int);`
`_mm256_blend_ps`	AVX	immintrin.h	`__m256 _mm256_blend_ps(__m256, __m256, const int);`
`_mm256_blendv_epi8`	AVX2	immintrin.h	`__m256i _mm256_blendv_epi8(__m256i, __m256i, __m256i);`
`_mm256_blendv_pd`	AVX	immintrin.h	`__m256d _mm256_blendv_pd(__m256d, __m256d, __m256d);`
`_mm256_blendv_ps`	AVX	immintrin.h	`__m256 _mm256_blendv_ps(__m256, __m256, __m256);`
`_mm256_broadcast_pd`	AVX	immintrin.h	`__m256d _mm256_broadcast_pd(__m128d const *);`
`_mm256_broadcast_ps`	AVX	immintrin.h	`__m256 _mm256_broadcast_ps(__m128 const *);`
`_mm256_broadcast_sd`	AVX	immintrin.h	`__m256d _mm256_broadcast_sd(double const *);`
`_mm256_broadcast_ss`	AVX	immintrin.h	`__m256 _mm256_broadcast_ss(float const *);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_broadcastb_epi8`	AVX2	immintrin.h	`__m256i _mm256_broadcastb_epi8 (__m128i);`
`_mm256_broadcastd_epi32`	AVX2	immintrin.h	`__m256i _mm256_broadcastd_epi32(__m128i);`
`_mm256_broadcastq_epi64`	AVX2	immintrin.h	`__m256i _mm256_broadcastq_epi64(__m128i);`
`_mm256_broadcastsd_pd`	AVX2	immintrin.h	`__m256d _mm256_broadcastsd_pd(__m128d);`
`_mm256_broadcastsi128_si256`	AVX2	immintrin.h	`__m256i _mm256_broadcastsi128_si256(__m128i);`
`_mm256_broadcastss_ps`	AVX2	immintrin.h	`__m256 _mm256_broadcastss_ps(__m128);`
`_mm256_broadcastw_epi16`	AVX2	immintrin.h	`__m256i _mm256_broadcastw_epi16(__m128i);`
`_mm256_castpd_ps`	AVX	immintrin.h	`__m256 _mm256_castpd_ps(__m256d);`
`_mm256_castpd_si256`	AVX	immintrin.h	`__m256i _mm256_castpd_si256(__m256d);`
`_mm256_castpd128_pd256`	AVX	immintrin.h	`__m256d _mm256_castpd128_pd256(__m128d);`
`_mm256_castpd256_pd128`	AVX	immintrin.h	`__m128d _mm256_castpd256_pd128(__m256d);`
`_mm256_castps_pd`	AVX	immintrin.h	`__m256d _mm256_castps_pd(__m256);`
`_mm256_castps_si256`	AVX	immintrin.h	`__m256i _mm256_castps_si256(__m256);`
`_mm256_castps128_ps256`	AVX	immintrin.h	`__m256 _mm256_castps128_ps256(__m128);`
`_mm256_castps256_ps128`	AVX	immintrin.h	`__m128 _mm256_castps256_ps128(__m256);`
`_mm256_castsi128_si256`	AVX	immintrin.h	`__m256i _mm256_castsi128_si256(__m128i);`
`_mm256_castsi256_pd`	AVX	immintrin.h	`__m256d _mm256_castsi256_pd(__m256i);`
`_mm256_castsi256_ps`	AVX	immintrin.h	`__m256 _mm256_castsi256_ps(__m256i);`
`_mm256_castsi256_si128`	AVX	immintrin.h	`__m128i _mm256_castsi256_si128(__m256i);`
`_mm256_cmov_si256`	XOP	ammintrin.h	`__m256i _mm256_cmov_si256(__m256i, __m256i, __m256i);`
`_mm256_cmp_pd`	AVX	immintrin.h	`__m256d _mm256_cmp_pd(__m256d, __m256d, const int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_cmp_ps`	AVX	immintrin.h	`__m256 _mm256_cmp_ps(__m256, __m256, const int);`
`_mm256_cmpeq_epi16`	AVX2	immintrin.h	`__m256i _mm256_cmpeq_epi16(__m256i, __m256i);`
`_mm256_cmpeq_epi32`	AVX2	immintrin.h	`__m256i _mm256_cmpeq_epi32(__m256i, __m256i);`
`_mm256_cmpeq_epi64`	AVX2	immintrin.h	`__m256i _mm256_cmpeq_epi64(__m256i, __m256i);`
`_mm256_cmpeq_epi8`	AVX2	immintrin.h	`__m256i _mm256_cmpeq_epi8(__m256i, __m256i);`
`_mm256_cmpgt_epi16`	AVX2	immintrin.h	`__m256i _mm256_cmpgt_epi16(__m256i, __m256i);`
`_mm256_cmpgt_epi32`	AVX2	immintrin.h	`__m256i _mm256_cmpgt_epi32(__m256i, __m256i);`
`_mm256_cmpgt_epi64`	AVX2	immintrin.h	`__m256i _mm256_cmpgt_epi64(__m256i, __m256i);`
`_mm256_cmpgt_epi8`	AVX2	immintrin.h	`__m256i _mm256_cmpgt_epi8(__m256i, __m256i);`
`_mm256_cvtepi16_epi32`	AVX2	immintrin.h	`__m256i _mm256_cvtepi16_epi32(__m128i);`
`_mm256_cvtepi16_epi64`	AVX2	immintrin.h	`__m256i _mm256_cvtepi16_epi64(__m128i);`
`_mm256_cvtepi32_epi64`	AVX2	immintrin.h	`__m256i _mm256_cvtepi32_epi64(__m128i);`
`_mm256_cvtepi32_pd`	AVX	immintrin.h	`__m256d _mm256_cvtepi32_pd(__m128i);`
`_mm256_cvtepi32_ps`	AVX	immintrin.h	`__m256 _mm256_cvtepi32_ps(__m256i);`
`_mm256_cvtepi8_epi16`	AVX2	immintrin.h	`__m256i _mm256_cvtepi8_epi16(__m128i);`
`_mm256_cvtepi8_epi32`	AVX2	immintrin.h	`__m256i _mm256_cvtepi8_epi32(__m128i);`
`_mm256_cvtepi8_epi64`	AVX2	immintrin.h	`__m256i _mm256_cvtepi8_epi64(__m128i);`
`_mm256_cvtepu16_epi32`	AVX2	immintrin.h	`__m256i _mm256_cvtepu16_epi32(__m128i);`
`_mm256_cvtepu16_epi64`	AVX2	immintrin.h	`__m256i _mm256_cvtepu16_epi64(__m128i);`
`_mm256_cvtepu32_epi64`	AVX2	immintrin.h	`__m256i _mm256_cvtepu32_epi64(__m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_cvtepu8_epi16`	AVX2	immintrin.h	`__m256i _mm256_cvtepu8_epi16(__m128i);`
`_mm256_cvtepu8_epi32`	AVX2	immintrin.h	`__m256i _mm256_cvtepu8_epi32(__m128i);`
`_mm256_cvtepu8_epi64`	AVX2	immintrin.h	`__m256i _mm256_cvtepu8_epi64(__m128i);`
`_mm256_cvtpd_epi32`	AVX	immintrin.h	`__m128i _mm256_cvtpd_epi32(__m256d);`
`_mm256_cvtpd_ps`	AVX	immintrin.h	`__m128 _mm256_cvtpd_ps(__m256d);`
`_mm256_cvtph_ps`	F16C	immintrin.h	`__m256 _mm256_cvtph_ps(__m128i);`
`_mm256_cvtps_epi32`	AVX	immintrin.h	`__m256i _mm256_cvtps_epi32(__m256);`
`_mm256_cvtps_pd`	AVX	immintrin.h	`__m256d _mm256_cvtps_pd(__m128);`
`_mm256_cvtps_ph`	F16C	immintrin.h	`__m128i _mm256_cvtps_ph(__m256, const int);`
`_mm256_cvttpd_epi32`	AVX	immintrin.h	`__m128i _mm256_cvttpd_epi32(__m256d);`
`_mm256_cvttps_epi32`	AVX	immintrin.h	`__m256i _mm256_cvttps_epi32(__m256);`
`_mm256_div_pd`	AVX	immintrin.h	`__m256d _mm256_div_pd(__m256d, __m256d);`
`_mm256_div_ps`	AVX	immintrin.h	`__m256 _mm256_div_ps(__m256, __m256);`
`_mm256_dp_ps`	AVX	immintrin.h	`__m256 _mm256_dp_ps(__m256, __m256, const int);`
`_mm256_extractf128_pd`	AVX	immintrin.h	`__m128d _mm256_extractf128_pd(__m256d, const int);`
`_mm256_extractf128_ps`	AVX	immintrin.h	`__m128 _mm256_extractf128_ps(__m256, const int);`
`_mm256_extractf128_si256`	AVX	immintrin.h	`__m128i _mm256_extractf128_si256(__m256i, const int);`
`_mm256_extracti128_si256`	AVX2	immintrin.h	`__m128i _mm256_extracti128_si256(__m256i, int);`
`_mm256_fmadd_pd`	FMA	immintrin.h	`__m256d _mm256_fmadd_pd (__m256d, __m256d, __m256d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_fmadd_ps`	FMA	immintrin.h	`__m256 _mm256_fmadd_ps (__m256, __m256, __m256);`
`_mm256_fmaddsub_pd`	FMA	immintrin.h	`__m256d _mm256_fmaddsub_pd (__m256d, __m256d, __m256d);`
`_mm256_fmaddsub_ps`	FMA	immintrin.h	`__m256 _mm256_fmaddsub_ps (__m256, __m256, __m256);`
`_mm256_fmsub_pd`	FMA	immintrin.h	`__m256d _mm256_fmsub_pd (__m256d, __m256d, __m256d);`
`_mm256_fmsub_ps`	FMA	immintrin.h	`__m256 _mm256_fmsub_ps (__m256, __m256, __m256);`
`_mm256_fmsubadd_pd`	FMA	immintrin.h	`__m256d _mm256_fmsubadd_pd (__m256d, __m256d, __m256d);`
`_mm256_fmsubadd_ps`	FMA	immintrin.h	`__m256 _mm256_fmsubadd_ps (__m256, __m256, __m256);`
`_mm256_fnmadd_pd`	FMA	immintrin.h	`__m256d _mm256_fnmadd_pd (__m256d, __m256d, __m256d);`
`_mm256_fnmadd_ps`	FMA	immintrin.h	`__m256 _mm256_fnmadd_ps (__m256, __m256, __m256);`
`_mm256_fnmsub_pd`	FMA	immintrin.h	`__m256d _mm256_fnmsub_pd (__m256d, __m256d, __m256d);`
`_mm256_fnmsub_ps`	FMA	immintrin.h	`__m256 _mm256_fnmsub_ps (__m256, __m256, __m256);`
`_mm256_frcz_pd`	XOP	ammintrin.h	`__m256d _mm256_frcz_pd(__m256d);`
`_mm256_frcz_ps`	XOP	ammintrin.h	`__m256 _mm256_frcz_ps(__m256);`
`_mm256_hadd_epi16`	AVX2	immintrin.h	`__m256i _mm256_hadd_epi16(__m256i, __m256i);`
`_mm256_hadd_epi32`	AVX2	immintrin.h	`__m256i _mm256_hadd_epi32(__m256i, __m256i);`
`_mm256_hadd_pd`	AVX	immintrin.h	`__m256d _mm256_hadd_pd(__m256d, __m256d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_hadd_ps`	AVX	immintrin.h	`__m256 _mm256_hadd_ps(__m256, __m256);`
`_mm256_hadds_epi16`	AVX2	immintrin.h	`__m256i _mm256_hadds_epi16(__m256i, __m256i);`
`_mm256_hsub_epi16`	AVX2	immintrin.h	`__m256i _mm256_hsub_epi16(__m256i, __m256i);`
`_mm256_hsub_epi32`	AVX2	immintrin.h	`__m256i _mm256_hsub_epi32(__m256i, __m256i);`
`_mm256_hsub_pd`	AVX	immintrin.h	`__m256d _mm256_hsub_pd(__m256d, __m256d);`
`_mm256_hsub_ps`	AVX	immintrin.h	`__m256 _mm256_hsub_ps(__m256, __m256);`
`_mm256_hsubs_epi16`	AVX2	immintrin.h	`__m256i _mm256_hsubs_epi16(__m256i, __m256i);`
`_mm256_i32gather_epi32`	AVX2	immintrin.h	`__m256i _mm256_i32gather_epi32(int const *, __m256i, const int);`
`_mm256_i32gather_epi64`	AVX2	immintrin.h	`__m256i _mm256_i32gather_epi64(__int64 const *, __m128i, const int);`
`_mm256_i32gather_pd`	AVX2	immintrin.h	`__m256d _mm256_i32gather_pd(double const *, __m128i, const int);`
`_mm256_i32gather_ps`	AVX2	immintrin.h	`__m256 _mm256_i32gather_ps(float const *, __m256i, const int);`
`_mm256_i64gather_epi32`	AVX2	immintrin.h	`__m256i _mm256_i64gather_epi32(int const *, __m256i, const int);`
`_mm256_i64gather_epi64`	AVX2	immintrin.h	`__m256i _mm256_i64gather_epi64(__int64 const *, __m256i, const int);`
`_mm256_i64gather_pd`	AVX2	immintrin.h	`__m256d _mm256_i64gather_pd(double const *, __m256i, const int);`
`_mm256_i64gather_ps`	AVX2	immintrin.h	`__m128 _mm256_i64gather_ps(float const *, __m256i, const int);`
`_mm256_insertf128_pd`	AVX	immintrin.h	`__m256d _mm256_insertf128_pd(__m256d, __m128d, int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_insertf128_ps`	AVX	immintrin.h	`__m256 _mm256_insertf128_ps(__m256, __m128, int);`
`_mm256_insertf128_si256`	AVX	immintrin.h	`__m256i _mm256_insertf128_si256(__m256i, __m128i, int);`
`_mm256_inserti128_si256`	AVX2	immintrin.h	`__m256i _mm256_inserti128_si256(__m256i, __m128i, int);`
`_mm256_lddqu_si256`	AVX	immintrin.h	`__m256i _mm256_lddqu_si256(__m256i *);`
`_mm256_load_pd`	AVX	immintrin.h	`__m256d _mm256_load_pd(double const *);`
`_mm256_load_ps`	AVX	immintrin.h	`__m256 _mm256_load_ps(float const *);`
`_mm256_load_si256`	AVX	immintrin.h	`__m256i _mm256_load_si256(__m256i *);`
`_mm256_loadu_pd`	AVX	immintrin.h	`__m256d _mm256_loadu_pd(double const *);`
`_mm256_loadu_ps`	AVX	immintrin.h	`__m256 _mm256_loadu_ps(float const *);`
`_mm256_loadu_si256`	AVX	immintrin.h	`__m256i _mm256_loadu_si256(__m256i *);`
`_mm256_macc_pd`	FMA4	ammintrin.h	`__m256d _mm_macc_pd(__m256d, __m256d, __m256d);`
`_mm256_macc_ps`	FMA4	ammintrin.h	`__m256 _mm_macc_ps(__m256, __m256, __m256);`
`_mm256_madd_epi16`	AVX2	immintrin.h	`__m256i _mm256_madd_epi16(__m256i, __m256i);`
`_mm256_maddsub_pd`	FMA4	ammintrin.h	`__m256d _mm_maddsub_pd(__m256d, __m256d, __m256d);`
`_mm256_maddsub_ps`	FMA4	ammintrin.h	`__m256 _mm_maddsub_ps(__m256, __m256, __m256);`
`_mm256_maddubs_epi16`	AVX2	immintrin.h	`__m256i _mm256_maddubs_epi16(__m256i, __m256i);`
`_mm256_mask_i32gather_epi32`	AVX2	immintrin.h	`__m256i _mm256_mask_i32gather_epi32(__m256i, int const *, __m256i, __m256i, const int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_mask_i32gather_epi64`	AVX2	immintrin.h	`__m256i _mm256_mask_i32gather_epi64(__m256i, __int64 const *, __m128i, __m256i, const int);`
`_mm256_mask_i32gather_pd`	AVX2	immintrin.h	`__m256d _mm256_mask_i32gather_pd(__m256d, double const *, __m128i, __m256d, const int);`
`_mm256_mask_i32gather_ps`	AVX2	immintrin.h	`__m256 _mm256_mask_i32gather_ps(__m256, float const *, __m256i, __m256, const int);`
`_mm256_mask_i64gather_epi32`	AVX2	immintrin.h	`__m128i _mm256_mask_i64gather_epi32(__m128i, int const *, __m256i, __m128i, const int);`
`_mm256_mask_i64gather_epi64`	AVX2	immintrin.h	`__m256i _mm256_mask_i64gather_epi64(__m256i, __int64 const *, __m256i, __m256i, const int);`
`_mm256_mask_i64gather_pd`	AVX2	immintrin.h	`__m256d _mm256_mask_i64gather_pd(__m256d, double const *, __m256i, __m256d, const int);`
`_mm256_mask_i64gather_ps`	AVX2	immintrin.h	`__m128 _mm256_mask_i64gather_ps(__m128, float const *, __m256i, __m128, const int);`
`_mm256_maskload_epi32`	AVX2	immintrin.h	`__m256i _mm256_maskload_epi32(int const *, __m256i);`
`_mm256_maskload_epi64`	AVX2	immintrin.h	`__m256i _mm256_maskload_epi64(__int64 const *, __m256i);`
`_mm256_maskload_pd`	AVX	immintrin.h	`__m256d _mm256_maskload_pd(double const *, __m256i);`
`_mm256_maskload_ps`	AVX	immintrin.h	`__m256 _mm256_maskload_ps(float const *, __m256i);`
`_mm256_maskstore_epi32`	AVX2	immintrin.h	`void _mm256_maskstore_epi32(int *, __m256i, __m256i);`
`_mm256_maskstore_epi64`	AVX2	immintrin.h	`void _mm256_maskstore_epi64(__int64 *, __m256i, __m256i);`
`_mm256_maskstore_pd`	AVX	immintrin.h	`void _mm256_maskstore_pd(double *, __m256i, __m256d);`
`_mm256_maskstore_ps`	AVX	immintrin.h	`void _mm256_maskstore_ps(float *, __m256i, __m256);`
`_mm256_max_epi16`	AVX2	immintrin.h	`__m256i _mm256_max_epi16(__m256i, __m256i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_max_epi32`	AVX2	immintrin.h	`__m256i _mm256_max_epi32(__m256i, __m256i);`
`_mm256_max_epi8`	AVX2	immintrin.h	`__m256i _mm256_max_epi8(__m256i, __m256i);`
`_mm256_max_epu16`	AVX2	immintrin.h	`__m256i _mm256_max_epu16(__m256i, __m256i);`
`_mm256_max_epu32`	AVX2	immintrin.h	`__m256i _mm256_max_epu32(__m256i, __m256i);`
`_mm256_max_epu8`	AVX2	immintrin.h	`__m256i _mm256_max_epu8(__m256i, __m256i);`
`_mm256_max_pd`	AVX	immintrin.h	`__m256d _mm256_max_pd(__m256d, __m256d);`
`_mm256_max_ps`	AVX	immintrin.h	`__m256 _mm256_max_ps(__m256, __m256);`
`_mm256_min_epi16`	AVX2	immintrin.h	`__m256i _mm256_min_epi16(__m256i, __m256i);`
`_mm256_min_epi32`	AVX2	immintrin.h	`__m256i _mm256_min_epi32(__m256i, __m256i);`
`_mm256_min_epi8`	AVX2	immintrin.h	`__m256i _mm256_min_epi8(__m256i, __m256i);`
`_mm256_min_epu16`	AVX2	immintrin.h	`__m256i _mm256_min_epu16(__m256i, __m256i);`
`_mm256_min_epu32`	AVX2	immintrin.h	`__m256i _mm256_min_epu32(__m256i, __m256i);`
`_mm256_min_epu8`	AVX2	immintrin.h	`__m256i _mm256_min_epu8(__m256i, __m256i);`
`_mm256_min_pd`	AVX	immintrin.h	`__m256d _mm256_min_pd(__m256d, __m256d);`
`_mm256_min_ps`	AVX	immintrin.h	`__m256 _mm256_min_ps(__m256, __m256);`
`_mm256_movedup_pd`	AVX	immintrin.h	`__m256d _mm256_movedup_pd(__m256d);`
`_mm256_movehdup_ps`	AVX	immintrin.h	`__m256 _mm256_movehdup_ps(__m256);`
`_mm256_moveldup_ps`	AVX	immintrin.h	`__m256 _mm256_moveldup_ps(__m256);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_movemask_epi8`	AVX2	immintrin.h	`int _mm256_movemask_epi8(__m256i);`
`_mm256_movemask_pd`	AVX	immintrin.h	`int _mm256_movemask_pd(__m256d);`
`_mm256_movemask_ps`	AVX	immintrin.h	`int _mm256_movemask_ps(__m256);`
`_mm256_mpsadbw_epu8`	AVX2	immintrin.h	`__m256i _mm256_mpsadbw_epu8(__m256i, __m256i, const int);`
`_mm256_msub_pd`	FMA4	ammintrin.h	`__m256d _mm_msub_pd(__m256d, __m256d, __m256d);`
`_mm256_msub_ps`	FMA4	ammintrin.h	`__m256 _mm_msub_ps(__m256, __m256, __m256);`
`_mm256_msubadd_pd`	FMA4	ammintrin.h	`__m256d _mm_msubadd_pd(__m256d, __m256d, __m256d);`
`_mm256_msubadd_ps`	FMA4	ammintrin.h	`__m256 _mm_msubadd_ps(__m256, __m256, __m256);`
`_mm256_mul_epi32`	AVX2	immintrin.h	`__m256i _mm256_mul_epi32(__m256i, __m256i);`
`_mm256_mul_epu32`	AVX2	immintrin.h	`__m256i _mm256_mul_epu32(__m256i, __m256i);`
`_mm256_mul_pd`	AVX	immintrin.h	`__m256d _mm256_mul_pd(__m256d, __m256d);`
`_mm256_mul_ps`	AVX	immintrin.h	`__m256 _mm256_mul_ps(__m256, __m256);`
`_mm256_mulhi_epi16`	AVX2	immintrin.h	`__m256i _mm256_mulhi_epi16(__m256i, __m256i);`
`_mm256_mulhi_epu16`	AVX2	immintrin.h	`__m256i _mm256_mulhi_epu16(__m256i, __m256i);`
`_mm256_mulhrs_epi16`	AVX2	immintrin.h	`__m256i _mm256_mulhrs_epi16(__m256i, __m256i);`
`_mm256_mullo_epi16`	AVX2	immintrin.h	`__m256i _mm256_mullo_epi16(__m256i, __m256i);`
`_mm256_mullo_epi32`	AVX2	immintrin.h	`__m256i _mm256_mullo_epi32(__m256i, __m256i);`
`_mm256_nmacc_pd`	FMA4	ammintrin.h	`__m256d _mm_nmacc_pd(__m256d, __m256d, __m256d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_nmacc_ps`	FMA4	ammintrin.h	`__m256 _mm_nmacc_ps(__m256, __m256, __m256);`
`_mm256_nmsub_pd`	FMA4	ammintrin.h	`__m256d _mm_nmsub_pd(__m256d, __m256d, __m256d);`
`_mm256_nmsub_ps`	FMA4	ammintrin.h	`__m256 _mm_nmsub_ps(__m256, __m256, __m256);`
`_mm256_or_pd`	AVX	immintrin.h	`__m256d _mm256_or_pd(__m256d, __m256d);`
`_mm256_or_ps`	AVX	immintrin.h	`__m256 _mm256_or_ps(__m256, __m256);`
`_mm256_or_si256`	AVX2	immintrin.h	`__m256i _mm256_or_si256(__m256i, __m256i);`
`_mm256_packs_epi16`	AVX2	immintrin.h	`__m256i _mm256_packs_epi16(__m256i, __m256i);`
`_mm256_packs_epi32`	AVX2	immintrin.h	`__m256i _mm256_packs_epi32(__m256i, __m256i);`
`_mm256_packus_epi16`	AVX2	immintrin.h	`__m256i _mm256_packus_epi16(__m256i, __m256i);`
`_mm256_packus_epi32`	AVX2	immintrin.h	`__m256i _mm256_packus_epi32(__m256i, __m256i);`
`_mm256_permute_pd`	AVX	immintrin.h	`__m256d _mm256_permute_pd(__m256d, int);`
`_mm256_permute_ps`	AVX	immintrin.h	`__m256 _mm256_permute_ps(__m256, int);`
`_mm256_permute2_pd`	XOP	ammintrin.h	`__m256d _mm256_permute2_pd(__m256d, __m256d, __m256i, int);`
`_mm256_permute2_ps`	XOP	ammintrin.h	`__m256 _mm256_permute2_ps(__m256, __m256, __m256i, int);`
`_mm256_permute2f128_pd`	AVX	immintrin.h	`__m256d _mm256_permute2f128_pd(__m256d, __m256d, int);`
`_mm256_permute2f128_ps`	AVX	immintrin.h	`__m256 _mm256_permute2f128_ps(__m256, __m256, int);`
`_mm256_permute2f128_si256`	AVX	immintrin.h	`__m256i _mm256_permute2f128_si256(__m256i, __m256i, int);`
`_mm256_permute2x128_si256`	AVX2	immintrin.h	`__m256i _mm256_permute2x128_si256(__m256i, __m256i, const int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_permute4x64_epi64`	AVX2	immintrin.h	`__m256i _mm256_permute4x64_epi64 (__m256i, const int);`
`_mm256_permute4x64_pd`	AVX2	immintrin.h	`__m256d _mm256_permute4x64_pd(__m256d, const int);`
`_mm256_permutevar_pd`	AVX	immintrin.h	`__m256d _mm256_permutevar_pd(__m256d, __m256i);`
`_mm256_permutevar_ps`	AVX	immintrin.h	`__m256 _mm256_permutevar_ps(__m256, __m256i);`
`_mm256_permutevar8x32_epi32`	AVX2	immintrin.h	`__m256i _mm256_permutevar8x32_epi32(__m256i, __m256i);`
`_mm256_permutevar8x32_ps`	AVX2	immintrin.h	`__m256 _mm256_permutevar8x32_ps (__m256, __m256i);`
`_mm256_rcp_ps`	AVX	immintrin.h	`__m256 _mm256_rcp_ps(__m256);`
`_mm256_round_pd`	AVX	immintrin.h	`__m256d _mm256_round_pd(__m256d, int);`
`_mm256_round_ps`	AVX	immintrin.h	`__m256 _mm256_round_ps(__m256, int);`
`_mm256_rsqrt_ps`	AVX	immintrin.h	`__m256 _mm256_rsqrt_ps(__m256);`
`_mm256_sad_epu8`	AVX2	immintrin.h	`__m256i _mm256_sad_epu8(__m256i, __m256i);`
`_mm256_set_epi16`	AVX	immintrin.h	`(__m256i _mm256_set_epi16(short, short, short, short, short, short, short, short, short, short, short, short, short, short, short, short);`
`_mm256_set_epi32`	AVX	immintrin.h	`__m256i _mm256_set_epi32(int, int, int, int, int, int, int, int);`
`_mm256_set_epi8`	AVX	immintrin.h	`__m256i _mm256_set_epi8(char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char);`
`_mm256_set_pd`	AVX	immintrin.h	`__m256d _mm256_set_pd(double, double, double, double);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_set_ps`	AVX	immintrin.h	`__m256 _mm256_set_ps(float, float, float, float, float, float, float, float);`
`_mm256_set1_epi16`	AVX	immintrin.h	`__m256i _mm256_set1_epi16(short);`
`_mm256_set1_epi32`	AVX	immintrin.h	`__m256i _mm256_set1_epi32(int);`
`_mm256_set1_epi8`	AVX	immintrin.h	`__m256i _mm256_set1_epi8(char);`
`_mm256_set1_pd`	AVX	immintrin.h	`__m256d _mm256_set1_pd(double);`
`_mm256_set1_ps`	AVX	immintrin.h	`__m256 _mm256_set1_ps(float);`
`_mm256_setr_epi16`	AVX	immintrin.h	`(__m256i _mm256_setr_epi16(short, short, short, short, short, short, short, short, short, short, short, short, short, short, short, short);`
`_mm256_setr_epi32`	AVX	immintrin.h	`__m256i _mm256_setr_epi32(int, int, int, int, int, int, int, int);`
`_mm256_setr_epi8`	AVX	immintrin.h	`(__m256i _mm256_setr_epi8(char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char,);`
`_mm256_setr_pd`	AVX	immintrin.h	`__m256d _mm256_setr_pd(double, double, double, double);`
`_mm256_setr_ps`	AVX	immintrin.h	`__m256 _mm256_setr_ps(float, float, float, float, float, float, float, float);`
`_mm256_setzero_pd`	AVX	immintrin.h	`__m256d _mm256_setzero_pd(void);`
`_mm256_setzero_ps`	AVX	immintrin.h	`__m256 _mm256_setzero_ps(void);`
`_mm256_setzero_si256`	AVX	immintrin.h	`__m256i _mm256_setzero_si256(void);`
`_mm256_shuffle_epi32`	AVX2	immintrin.h	`__m256i _mm256_shuffle_epi32(__m256i, const int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_shuffle_epi8`	AVX2	immintrin.h	`__m256i _mm256_shuffle_epi8(__m256i, __m256i);`
`_mm256_shuffle_pd`	AVX	immintrin.h	`__m256d _mm256_shuffle_pd(__m256d, __m256d, const int);`
`_mm256_shuffle_ps`	AVX	immintrin.h	`__m256 _mm256_shuffle_ps(__m256, __m256, const int);`
`_mm256_shufflehi_epi16`	AVX2	immintrin.h	`__m256i _mm256_shufflehi_epi16(__m256i, const int);`
`_mm256_shufflelo_epi16`	AVX2	immintrin.h	`__m256i _mm256_shufflelo_epi16(__m256i, const int);`
`_mm256_sign_epi16`	AVX2	immintrin.h	`__m256i _mm256_sign_epi16(__m256i, __m256i);`
`_mm256_sign_epi32`	AVX2	immintrin.h	`__m256i _mm256_sign_epi32(__m256i, __m256i);`
`_mm256_sign_epi8`	AVX2	immintrin.h	`__m256i _mm256_sign_epi8(__m256i, __m256i);`
`_mm256_sll_epi16`	AVX2	immintrin.h	`__m256i _mm256_sll_epi16(__m256i, __m128i);`
`_mm256_sll_epi32`	AVX2	immintrin.h	`__m256i _mm256_sll_epi32(__m256i, __m128i);`
`_mm256_sll_epi64`	AVX2	immintrin.h	`__m256i _mm256_sll_epi64(__m256i, __m128i);`
`_mm256_slli_epi16`	AVX2	immintrin.h	`__m256i _mm256_slli_epi16(__m256i, int);`
`_mm256_slli_epi32`	AVX2	immintrin.h	`__m256i _mm256_slli_epi32(__m256i, int);`
`_mm256_slli_epi64`	AVX2	immintrin.h	`__m256i _mm256_slli_epi64(__m256i, int);`
`_mm256_slli_si256`	AVX2	immintrin.h	`__m256i _mm256_slli_si256(__m256i, int);`
`_mm256_sllv_epi32`	AVX2	immintrin.h	`__m256i _mm256_sllv_epi32(__m256i, __m256i);`
`_mm256_sllv_epi64`	AVX2	immintrin.h	`__m256i _mm256_sllv_epi64(__m256i, __m256i);`
`_mm256_sqrt_pd`	AVX	immintrin.h	`__m256d _mm256_sqrt_pd(__m256d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_sqrt_ps`	AVX	immintrin.h	`__m256 _mm256_sqrt_ps(__m256);`
`_mm256_sra_epi16`	AVX2	immintrin.h	`__m256i _mm256_sra_epi16(__m256i, __m128i);`
`_mm256_sra_epi32`	AVX2	immintrin.h	`__m256i _mm256_sra_epi32(__m256i, __m128i);`
`_mm256_srai_epi16`	AVX2	immintrin.h	`__m256i _mm256_srai_epi16(__m256i, int);`
`_mm256_srai_epi32`	AVX2	immintrin.h	`__m256i _mm256_srai_epi32(__m256i, int);`
`_mm256_srav_epi32`	AVX2	immintrin.h	`__m256i _mm256_srav_epi32(__m256i, __m256i);`
`_mm256_srl_epi16`	AVX2	immintrin.h	`__m256i _mm256_srl_epi16(__m256i, __m128i);`
`_mm256_srl_epi32`	AVX2	immintrin.h	`__m256i _mm256_srl_epi32(__m256i, __m128i);`
`_mm256_srl_epi64`	AVX2	immintrin.h	`__m256i _mm256_srl_epi64(__m256i, __m128i);`
`_mm256_srli_epi16`	AVX2	immintrin.h	`__m256i _mm256_srli_epi16(__m256i, int);`
`_mm256_srli_epi32`	AVX2	immintrin.h	`__m256i _mm256_srli_epi32(__m256i, int);`
`_mm256_srli_epi64`	AVX2	immintrin.h	`__m256i _mm256_srli_epi64(__m256i, int);`
`_mm256_srli_si256`	AVX2	immintrin.h	`__m256i _mm256_srli_si256(__m256i, int);`
`_mm256_srlv_epi32`	AVX2	immintrin.h	`__m256i _mm256_srlv_epi32(__m256i, __m256i);`
`_mm256_srlv_epi64`	AVX2	immintrin.h	`__m256i _mm256_srlv_epi64(__m256i, __m256i);`
`_mm256_store_pd`	AVX	immintrin.h	`void _mm256_store_pd(double *, __m256d);`
`_mm256_store_ps`	AVX	immintrin.h	`void _mm256_store_ps(float *, __m256);`
`_mm256_store_si256`	AVX	immintrin.h	`void _mm256_store_si256(__m256i *, __m256i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_storeu_pd`	AVX	immintrin.h	`void _mm256_storeu_pd(double *, __m256d);`
`_mm256_storeu_ps`	AVX	immintrin.h	`void _mm256_storeu_ps(float *, __m256);`
`_mm256_storeu_si256`	AVX	immintrin.h	`void _mm256_storeu_si256(__m256i *, __m256i);`
`_mm256_stream_load_si256`	AVX2	immintrin.h	`__m256i _mm256_stream_load_si256(__m256i const *);`
`_mm256_stream_pd`	AVX	immintrin.h	`void __mm256_stream_pd(double *, __m256d);`
`_mm256_stream_ps`	AVX	immintrin.h	`void _mm256_stream_ps(float *, __m256);`
`_mm256_stream_si256`	AVX	immintrin.h	`void __mm256_stream_si256(__m256i *, __m256i);`
`_mm256_sub_epi16`	AVX2	immintrin.h	`__m256i _mm256_sub_epi16(__m256i, __m256i);`
`_mm256_sub_epi32`	AVX2	immintrin.h	`__m256i _mm256_sub_epi32(__m256i, __m256i);`
`_mm256_sub_epi64`	AVX2	immintrin.h	`__m256i _mm256_sub_epi64(__m256i, __m256i);`
`_mm256_sub_epi8`	AVX2	immintrin.h	`__m256i _mm256_sub_epi8(__m256i, __m256i);`
`_mm256_sub_pd`	AVX	immintrin.h	`__m256d _mm256_sub_pd(__m256d, __m256d);`
`_mm256_sub_ps`	AVX	immintrin.h	`__m256 _mm256_sub_ps(__m256, __m256);`
`_mm256_subs_epi16`	AVX2	immintrin.h	`__m256i _mm256_subs_epi16(__m256i, __m256i);`
`_mm256_subs_epi8`	AVX2	immintrin.h	`__m256i _mm256_subs_epi8(__m256i, __m256i);`
`_mm256_subs_epu16`	AVX2	immintrin.h	`__m256i _mm256_subs_epu16(__m256i, __m256i);`
`_mm256_subs_epu8`	AVX2	immintrin.h	`__m256i _mm256_subs_epu8(__m256i, __m256i);`
`_mm256_testc_pd`	AVX	immintrin.h	`int _mm256_testc_pd(__m256d, __m256d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_testc_ps`	AVX	immintrin.h	`int _mm256_testc_ps(__m256, __m256);`
`_mm256_testc_si256`	AVX	immintrin.h	`int _mm256_testc_si256(__m256i, __m256i);`
`_mm256_testnzc_pd`	AVX	immintrin.h	`int _mm256_testnzc_pd(__m256d, __m256d);`
`_mm256_testnzc_ps`	AVX	immintrin.h	`int _mm256_testnzc_ps(__m256, __m256);`
`_mm256_testnzc_si256`	AVX	immintrin.h	`int _mm256_testnzc_si256(__m256i, __m256i);`
`_mm256_testz_pd`	AVX	immintrin.h	`int _mm256_testz_pd(__m256d, __m256d);`
`_mm256_testz_ps`	AVX	immintrin.h	`int _mm256_testz_ps(__m256, __m256);`
`_mm256_testz_si256`	AVX	immintrin.h	`int _mm256_testz_si256(__m256i, __m256i);`
`_mm256_unpackhi_epi16`	AVX2	immintrin.h	`__m256i _mm256_unpackhi_epi16(__m256i, __m256i);`
`_mm256_unpackhi_epi32`	AVX2	immintrin.h	`__m256i _mm256_unpackhi_epi32(__m256i, __m256i);`
`_mm256_unpackhi_epi64`	AVX2	immintrin.h	`__m256i _mm256_unpackhi_epi64(__m256i, __m256i);`
`_mm256_unpackhi_epi8`	AVX2	immintrin.h	`__m256i _mm256_unpackhi_epi8(__m256i, __m256i);`
`_mm256_unpackhi_pd`	AVX	immintrin.h	`__m256d _mm256_unpackhi_pd(__m256d, __m256d);`
`_mm256_unpackhi_ps`	AVX	immintrin.h	`__m256 _mm256_unpackhi_ps(__m256, __m256);`
`_mm256_unpacklo_epi16`	AVX2	immintrin.h	`__m256i _mm256_unpacklo_epi16(__m256i, __m256i);`
`_mm256_unpacklo_epi32`	AVX2	immintrin.h	`__m256i _mm256_unpacklo_epi32(__m256i, __m256i);`
`_mm256_unpacklo_epi64`	AVX2	immintrin.h	`__m256i _mm256_unpacklo_epi64(__m256i, __m256i);`
`_mm256_unpacklo_epi8`	AVX2	immintrin.h	`__m256i _mm256_unpacklo_epi8(__m256i, __m256i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_unpacklo_pd`	AVX	immintrin.h	`__m256d _mm256_unpacklo_pd(__m256d, __m256d);`
`_mm256_unpacklo_ps`	AVX	immintrin.h	`__m256 _mm256_unpacklo_ps(__m256, __m256);`
`_mm256_xor_pd`	AVX	immintrin.h	`__m256d _mm256_xor_pd(__m256d, __m256d);`
`_mm256_xor_ps`	AVX	immintrin.h	`__m256 _mm256_xor_ps(__m256, __m256);`
`_mm256_xor_si256`	AVX2	immintrin.h	`__m256i _mm256_xor_si256(__m256i, __m256i);`
`_mm256_zeroall`	AVX	immintrin.h	`void _mm256_zeroall(void);`
`_mm256_zeroupper`	AVX	immintrin.h	`void _mm256_zeroupper(void);`
`__movsb`		intrin.h	`void __movsb(unsigned char *, unsigned char const *, size_t);`
`__movsd`		intrin.h	`void __movsd(unsigned long *, unsigned long const *, size_t);`
`__movsw`		intrin.h	`void __movsw(unsigned short *, unsigned short const *, size_t);`
`_mulx_u32`	BMI	immintrin.h	`unsigned int _mulx_u32(unsigned int, unsigned int, unsigned int*);`
`__nop`		intrin.h	`void __nop(void);`
`__nvreg_restore_fence`		intrin.h	`void __nvreg_restore_fence(void);`
`__nvreg_save_fence`		intrin.h	`void __nvreg_save_fence(void);`
`__outbyte`		intrin.h	`void __outbyte(unsigned short, unsigned char);`
`__outbytestring`		intrin.h	`void __outbytestring(unsigned short, unsigned char *, unsigned long);`
`__outdword`		intrin.h	`void __outdword(unsigned short, unsigned long);`
`__outdwordstring`		intrin.h	`void __outdwordstring(unsigned short, unsigned long *, unsigned long);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__outword`		intrin.h	`void __outword(unsigned short, unsigned short);`
`__outwordstring`		intrin.h	`void __outwordstring(unsigned short, unsigned short *, unsigned long);`
`_pdep_u32`	BMI	immintrin.h	`unsigned int _pdep_u32(unsigned int, unsigned int);`
`_pext_u32`	BMI	immintrin.h	`unsigned int _pext_u32(unsigned int, unsigned int);`
`__popcnt`	POPCNT	intrin.h	`unsigned int __popcnt(unsigned int);`
`__popcnt16`	POPCNT	intrin.h	`unsigned short __popcnt16(unsigned short);`
`_rdrand16_step`	RDRAND	immintrin.h	`int _rdrand16_step(unsigned short *);`
`_rdrand32_step`	RDRAND	immintrin.h	`int _rdrand32_step(unsigned int *);`
`_rdseed16_step`	RDSEED	immintrin.h	`int _rdseed16_step(unsigned short *);`
`_rdseed32_step`	RDSEED	immintrin.h	`int _rdseed32_step(unsigned int *);`
`__rdtsc`		intrin.h	`unsigned __int64 __rdtsc(void);`
`__rdtscp`	RDTSCP	intrin.h	`unsigned __int64 __rdtscp(unsigned int*);`
`_ReadBarrier`		intrin.h	`void _ReadBarrier(void);`
`__readcr0`		intrin.h	`unsigned long __readcr0(void);`
`__readcr2`		intrin.h	`unsigned long __readcr2(void);`
`__readcr3`		intrin.h	`unsigned long __readcr3(void);`
`__readcr4`		intrin.h	`unsigned long __readcr4(void);`
`__readcr8`		intrin.h	`unsigned long __readcr8(void);`
`__readdr`		intrin.h	`unsigned __readdr(unsigned);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__readeflags`		intrin.h	`unsigned __readeflags(void);`
`__readfsbyte`		intrin.h	`unsigned char __readfsbyte(unsigned long);`
`__readfsdword`		intrin.h	`unsigned long __readfsdword(unsigned long);`
`__readfsword`		intrin.h	`unsigned short __readfsword(unsigned long);`
`__readmsr`		intrin.h	`unsigned __int64 __readmsr(unsigned long);`
`__readpmc`		intrin.h	`unsigned __int64 __readpmc(unsigned long);`
`_ReadWriteBarrier`		intrin.h	`void _ReadWriteBarrier(void);`
`_ReturnAddress`		intrin.h	`void * _ReturnAddress(void);`
`_rorx_u32`	BMI	immintrin.h	`unsigned int _rorx_u32(unsigned int, const unsigned int);`
`_rotl16`		intrin.h	`unsigned short _rotl16(unsigned short, unsigned char);`
`_rotl8`		intrin.h	`unsigned char _rotl8(unsigned char, unsigned char);`
`_rotr16`		intrin.h	`unsigned short _rotr16(unsigned short, unsigned char);`
`_rotr8`		intrin.h	`unsigned char _rotr8(unsigned char, unsigned char);`
`_rsm`		intrin.h	`void _rsm(void);`
`_sarx_i32`	BMI	immintrin.h	`int _sarx_i32(int, unsigned int);`
`__segmentlimit`		intrin.h	`unsigned long __segmentlimit(unsigned long);`
`_sgdt`		intrin.h	`void _sgdt(void*);`
`_shlx_u32`	BMI	immintrin.h	`unsigned int _shlx_u32(unsigned int, unsigned int);`
`_shrx_u32`	BMI	immintrin.h	`unsigned int _shrx_u32(unsigned int, unsigned int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__sidt`		intrin.h	`void __sidt(void*);`
`__slwpcb`	LWP	ammintrin.h	`void *__slwpcb(void);`
`_stac`	SMAP	intrin.h	`void _stac(void);`
`_storebe_i16`	MOVBE	immintrin.h	`void _storebe_i16(void *, short);` [Macro]
`_storebe_i32`	MOVBE	immintrin.h	`void _storebe_i32(void *, int);` [Macro]
`_store_be_u16`	MOVBE	immintrin.h	`void _store_be_u16(void *, unsigned short);` [Macro]
`_store_be_u32`	MOVBE	immintrin.h	`void _store_be_u32(void *, unsigned int);` [Macro]
`_Store_HLERelease`	HLE	immintrin.h	`void _Store_HLERelease(long volatile *, long);`
`_StorePointer_HLERelease`	HLE	immintrin.h	`void _StorePointer_HLERelease(void * volatile *, void *);`
`__stosb`		intrin.h	`void __stosb(unsigned char *, unsigned char, size_t);`
`__stosd`		intrin.h	`void __stosd(unsigned long *, unsigned long, size_t);`
`__stosw`		intrin.h	`void __stosw(unsigned short *, unsigned short, size_t);`
`_subborrow_u16`		intrin.h	`unsigned char _subborrow_u16(unsigned char, unsigned short, unsigned short, unsigned short *);`
`_subborrow_u32`		intrin.h	`unsigned char _subborrow_u32(unsigned char, unsigned int, unsigned int, unsigned int *);`
`_subborrow_u8`		intrin.h	`unsigned char _subborrow_u8(unsigned char, unsigned char, unsigned char, unsigned char *);`
`__svm_clgi`		intrin.h	`void __svm_clgi(void);`
`__svm_invlpga`		intrin.h	`void __svm_invlpga(void*, int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__svm_skinit`		intrin.h	`void __svm_skinit(int);`
`__svm_stgi`		intrin.h	`void __svm_stgi(void);`
`__svm_vmload`		intrin.h	`void __svm_vmload(size_t);`
`__svm_vmrun`		intrin.h	`void __svm_vmrun(size_t);`
`__svm_vmsave`		intrin.h	`void __svm_vmsave(size_t);`
`_t1mskc_u32`	ABM	ammintrin.h	`unsigned int _t1mskc_u32(unsigned int);`
`_tzcnt_u32`	BMI	ammintrin.h, immintrin.h	`unsigned int _tzcnt_u32(unsigned int);`
`_tzmsk_u32`	ABM	ammintrin.h	`unsigned int _tzmsk_u32(unsigned int);`
`__ud2`		intrin.h	`void __ud2(void);`
`_udiv64`		intrin.h	`unsigned int _udiv64(unsigned __int64, unsigned int, unsigned int *);`
`__ull_rshift`		intrin.h	`unsigned __int64 [pascal/cdecl] __ull_rshift(unsigned __int64, int);`
`__vmx_off`		intrin.h	`void __vmx_off(void);`
`__vmx_vmptrst`		intrin.h	`void __vmx_vmptrst(unsigned __int64 *);`
`__wbinvd`		intrin.h	`void __wbinvd(void);`
`_WriteBarrier`		intrin.h	`void _WriteBarrier(void);`
`__writecr0`		intrin.h	`void __writecr0(unsigned long);`
`__writecr3`		intrin.h	`void __writecr3(unsigned long);`
`__writecr4`		intrin.h	`void __writecr4(unsigned long);`
`__writecr8`		intrin.h	`void __writecr8(unsigned long);`
`__writedr`		intrin.h	`void __writedr(unsigned, unsigned);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__writeeflags`		intrin.h	`void __writeeflags(unsigned);`
`__writefsbyte`		intrin.h	`void __writefsbyte(unsigned long, unsigned char);`
`__writefsdword`		intrin.h	`void __writefsdword(unsigned long, unsigned long);`
`__writefsword`		intrin.h	`void __writefsword(unsigned long, unsigned short);`
`__writemsr`		intrin.h	`void __writemsr(unsigned long, unsigned __int64);`
`_xabort`	RTM	immintrin.h	`void _xabort(unsigned int);`
`_xbegin`	RTM	immintrin.h	`unsigned _xbegin(void);`
`_xend`	RTM	immintrin.h	`void _xend(void);`
`_xgetbv`	XSAVE	immintrin.h	`unsigned __int64 _xgetbv(unsigned int);`
`_xrstor`	XSAVE	immintrin.h	`void _xrstor(void const*, unsigned __int64);`
`_xsave`	XSAVE	immintrin.h	`void _xsave(void*, unsigned __int64);`
`_xsaveopt`	XSAVEOPT	immintrin.h	`void _xsaveopt(void*, unsigned __int64);`
`_xsetbv`	XSAVE	immintrin.h	`void _xsetbv(unsigned int, unsigned __int64);`
`_xtest`	XTEST	immintrin.h	`unsigned char _xtest(void);`

See also

Compiler intrinsics
ARM intrinsics
ARM64 intrinsics
x64 (amd64) intrinsics

x64 (amd64) intrinsics list

9/2/2022 • 43 minutes to read • Edit Online

This document lists intrinsics that the Microsoft C++ compiler supports when x64 (also referred to as amd64) is targeted.

For information about individual intrinsics, see these resources, as appropriate for the processor you're targeting:

- The header file. Many intrinsics are documented in comments in the header file.
- Intel Intrinsics Guide. Use the search box to find specific intrinsics.
- Intel 64 and IA-32 Architectures Software Developer Manuals
- Intel Architecture Instruction Set Extensions Programming Reference
- Introduction to Intel Advanced Vector Extensions
- AMD Developer Guides, Manuals & ISA Documents

x64 intrinsics

The following table lists the intrinsics available on x64 processors. The Technology column lists required instruction-set support. Use the `__cpuid` intrinsic to determine instruction-set support at run time. If two entries are in one row, they represent different entry points for the same intrinsic. [Macro] indicates the prototype is a macro. The header required for the function prototype is listed in the Header column. The `intrin.h` header includes both `immintrin.h` and `ammintrin.h` for simplicity.

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_addcarry_u16`		intrin.h	`unsigned char _addcarry_u16(unsigned char, unsigned short, unsigned short, unsigned short *);`
`_addcarry_u32`		intrin.h	`unsigned char _addcarry_u32(unsigned char, unsigned int, unsigned int, unsigned int *);`
`_addcarry_u64`		intrin.h	`unsigned char _addcarry_u64(unsigned char, unsigned __int64, unsigned __int64, unsigned __int64 *);`
`_addcarry_u8`		intrin.h	`unsigned char _addcarry_u8(unsigned char, unsigned char, unsigned char, unsigned char *);`
`_addcarryx_u32`	ADX	immintrin.h	`unsigned char _addcarryx_u32(unsigned char, unsigned int, unsigned int, unsigned int *);`
`_addcarryx_u64`	ADX	immintrin.h	`unsigned char _addcarryx_u64(unsigned char, unsigned __int64, unsigned __int64, unsigned __int64 *);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__addgsbyte`		intrin.h	`void __addgsbyte(unsigned long, unsigned char);`
`__addgsdword`		intrin.h	`void __addgsdword(unsigned long, unsigned int);`
`__addgsqword`		intrin.h	`void __addgsqword(unsigned long, unsigned __int64);`
`__addgsword`		intrin.h	`void __addgsword(unsigned long, unsigned short);`
`_AddressOfReturnAddress`		intrin.h	`void * _AddressOfReturnAddress(void);`
`_andn_u32`	BMI	ammintrin.h	`unsigned int _andn_u32(unsigned int, unsigned int);`
`_andn_u64`	BMI	ammintrin.h	`unsigned __int64 _andn_u64(unsigned __int64, unsigned __int64);`
`_bextr_u32`	BMI	ammintrin.h, immintrin.h	`unsigned int _bextr_u32(unsigned int, unsigned int, unsigned int);`
`_bextr_u64`	BMI	ammintrin.h, immintrin.h	`unsigned __int64 _bextr_u64(unsigned __int64, unsigned int, unsigned int);`
`_bextri_u32`	ABM	ammintrin.h	`unsigned int _bextri_u32(unsigned int, unsigned int);`
`_bextri_u64`	ABM	ammintrin.h	`unsigned __int64 _bextri_u64(unsigned __int64, unsigned int);`
`_BitScanForward`		intrin.h	`unsigned char _BitScanForward(unsigned long*, unsigned long);`
`_BitScanForward64`		intrin.h	`unsigned char _BitScanForward64(unsigned long*, unsigned __int64);`
`_BitScanReverse`		intrin.h	`unsigned char _BitScanReverse(unsigned long*, unsigned long);`
`_BitScanReverse64`		intrin.h	`unsigned char _BitScanReverse64(unsigned long*, unsigned __int64);`
`_bittest`		intrin.h	`unsigned char _bittest(long const *, long);`
`_bittest64`		intrin.h	`unsigned char _bittest64(__int64 const *, __int64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_bittestandcomplement`		intrin.h	`unsigned char _bittestandcomplement(long *, long);`
`_bittestandcomplement64`		intrin.h	`unsigned char _bittestandcomplement64(__int64 *, __int64);`
`_bittestandreset`		intrin.h	`unsigned char _bittestandreset(long *, long);`
`_bittestandreset64`		intrin.h	`unsigned char _bittestandreset64(__int64 *, __int64);`
`_bittestandset`		intrin.h	`unsigned char _bittestandset(long *, long);`
`_bittestandset64`		intrin.h	`unsigned char _bittestandset64(__int64 *, __int64);`
`_blcfill_u32`	ABM	ammintrin.h	`unsigned int _blcfill_u32(unsigned int);`
`_blcfill_u64`	ABM	ammintrin.h	`unsigned __int64 _blcfill_u64(unsigned __int64);`
`_blci_u32`	ABM	ammintrin.h	`unsigned int _blci_u32(unsigned int);`
`_blci_u64`	ABM	ammintrin.h	`unsigned __int64 _blci_u64(unsigned __int64);`
`_blcic_u32`	ABM	ammintrin.h	`unsigned int _blcic_u32(unsigned int);`
`_blcic_u64`	ABM	ammintrin.h	`unsigned __int64 _blcic_u64(unsigned __int64);`
`_blcmsk_u32`	ABM	ammintrin.h	`unsigned int _blcmsk_u32(unsigned int);`
`_blcmsk_u64`	ABM	ammintrin.h	`unsigned __int64 _blcmsk_u64(unsigned __int64);`
`_blcs_u32`	ABM	ammintrin.h	`unsigned int _blcs_u32(unsigned int);`
`_blcs_u64`	ABM	ammintrin.h	`unsigned __int64 _blcs_u64(unsigned __int64);`
`_blsfill_u32`	ABM	ammintrin.h	`unsigned int _blsfill_u32(unsigned int);`
`_blsfill_u64`	ABM	ammintrin.h	`unsigned __int64 _blsfill_u64(unsigned __int64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_blsi_u32`	BMI	ammintrin.h, immintrin.h	`unsigned int _blsi_u32(unsigned int);`
`_blsi_u64`	BMI	ammintrin.h, immintrin.h	`unsigned __int64 _blsi_u64(unsigned __int64);`
`_blsic_u32`	ABM	ammintrin.h	`unsigned int _blsic_u32(unsigned int);`
`_blsic_u64`	ABM	ammintrin.h	`unsigned __int64 _blsic_u64(unsigned __int64);`
`_blsmsk_u32`	BMI	ammintrin.h, immintrin.h	`unsigned int _blsmsk_u32(unsigned int);`
`_blsmsk_u64`	BMI	ammintrin.h, immintrin.h	`unsigned __int64 _blsmsk_u64(unsigned __int64);`
`_blsr_u32`	BMI	ammintrin.h, immintrin.h	`unsigned int _blsr_u32(unsigned int);`
`_blsr_u64`	BMI	ammintrin.h, immintrin.h	`unsigned __int64 _blsr_u64(unsigned __int64);`
`_bzhi_u32`	BMI	immintrin.h	`unsigned int _bzhi_u32(unsigned int, unsigned int);`
`_bzhi_u64`	BMI	immintrin.h	`unsigned __int64 _bzhi_u64(unsigned __int64, unsigned int);`
`_castf32_u32`		immintrin.h	`unsigned __int32 _castf32_u32 (float);`
`_castf64_u64`		immintrin.h	`unsigned __int64 _castf64_u64 (double);`
`_castu32_f32`		immintrin.h	`float _castu32_f32 (unsigned __int32);`
`_castu64_f64`		immintrin.h	`double _castu64_f64 (unsigned __int64 a);`
`_clac`	SMAP	intrin.h	`void _clac(void);`
`__cpuid`		intrin.h	`void __cpuid(int *, int);`
`__cpuidex`		intrin.h	`void __cpuidex(int *, int, int);`
`__debugbreak`		intrin.h	`void __debugbreak(void);`
`_disable`		intrin.h	`void _disable(void);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_div128`		intrin.h	`__int64 _div128(__int64, __int64, __int64, __int64 *);`
`_div64`		intrin.h	`int _div64(__int64, int, int*);`
`__emul`		intrin.h	`__int64 [pascal/cdecl] __emul(int, int);`
`__emulu`		intrin.h	`unsigned __int64 [pascal/cdecl]__emulu(unsigned int, unsigned int);`
`_enable`		intrin.h	`void _enable(void);`
`__fastfail`		intrin.h	`void __fastfail(unsigned int);`
`__faststorefence`		intrin.h	`void __faststorefence(void);`
`_fxrstor`	FXSR	immintrin.h	`void _fxrstor(void const*);`
`_fxrstor64`	FXSR	immintrin.h	`void _fxrstor64(void const*);`
`_fxsave`	FXSR	immintrin.h	`void _fxsave(void*);`
`_fxsave64`	FXSR	immintrin.h	`void _fxsave64(void*);`
`__getcallerseflags`		intrin.h	`(unsigned int __getcallerseflags());`
`__halt`		intrin.h	`void __halt(void);`
`__inbyte`		intrin.h	`unsigned char __inbyte(unsigned short);`
`__inbytestring`		intrin.h	`void __inbytestring(unsigned short, unsigned char *, unsigned long);`
`__incgsbyte`		intrin.h	`void __incgsbyte(unsigned long);`
`__incgsdword`		intrin.h	`void __incgsdword(unsigned long);`
`__incgsqword`		intrin.h	`void __incgsqword(unsigned long);`
`__incgsword`		intrin.h	`void __incgsword(unsigned long);`
`__indword`		intrin.h	`unsigned long __indword(unsigned short);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__indwordstring`		intrin.h	`void __indwordstring(unsigned short, unsigned long *, unsigned long);`
`__int2c`		intrin.h	`void __int2c(void);`
`_InterlockedAnd`		intrin.h	`long _InterlockedAnd(long volatile *, long);`
`_InterlockedAnd_HLEAcquire`	HLE	immintrin.h	`long _InterlockedAnd_HLEAcquire(long volatile *, long);`
`_InterlockedAnd_HLERelease`	HLE	immintrin.h	`long _InterlockedAnd_HLERelease(long volatile *, long);`
`_InterlockedAnd_np`		intrin.h	`long _InterlockedAnd_np(long *, long);`
`_InterlockedAnd16`		intrin.h	`short _InterlockedAnd16(short volatile *, short);`
`_InterlockedAnd16_np`		intrin.h	`short _InterlockedAnd16_np(short *, short);`
`_InterlockedAnd64`		intrin.h	`__int64 _InterlockedAnd64(__int64 volatile *, __int64);`
`_InterlockedAnd64_HLEAcquire`	HLE	immintrin.h	`__int64 _InterlockedAnd64_HLEAcquire(__int64 volatile *, __int64);`
`_InterlockedAnd64_HLERelease`	HLE	immintrin.h	`__int64 _InterlockedAnd64_HLERelease(__int64 volatile *, __int64);`
`_InterlockedAnd64_np`		intrin.h	`__int64 _InterlockedAnd64_np(__int64 *, __int64);`
`_InterlockedAnd8`		intrin.h	`char _InterlockedAnd8(char volatile *, char);`
`_InterlockedAnd8_np`		intrin.h	`char _InterlockedAnd8_np(char *, char);`
`_interlockedbittestandreset`		intrin.h	`unsigned char _interlockedbittestandreset(long *, long);`
`_interlockedbittestandreset_HLEAcquire`	HLE	immintrin.h	`unsigned char _interlockedbittestandreset_HLEAcquire(lon *, long);`
`_interlockedbittestandreset_HLERelease`	HLE	immintrin.h	`unsigned char _interlockedbittestandreset_HLERelease(lon *, long);`
`_interlockedbittestandreset64`		intrin.h	`unsigned char _interlockedbittestandreset64(__int64 *, __int64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_interlockedbittestandreset64_HLEAcquire`	HLE	immintrin.h	`unsigned char _interlockedbittestandreset64_HLEAcquire(_ *, __int64);`
`_interlockedbittestandreset64_HLERelease`	HLE	immintrin.h	`unsigned char _interlockedbittestandreset64_HLERelease(_ *, __int64);`
`_interlockedbittestandset`		intrin.h	`unsigned char _interlockedbittestandset(long *, long);`
`_interlockedbittestandset_HLEAcquire`	HLE	immintrin.h	`unsigned char _interlockedbittestandset_HLEAcquire(long *, long);`
`_interlockedbittestandset_HLERelease`	HLE	immintrin.h	`unsigned char _interlockedbittestandset_HLERelease(long *, long);`
`_interlockedbittestandset64`		intrin.h	`unsigned char _interlockedbittestandset64(__int64 *, __int64);`
`_interlockedbittestandset64_HLEAcquire`	HLE	immintrin.h	`unsigned char _interlockedbittestandset64_HLEAcquire(__i *, __int64);`
`_interlockedbittestandset64_HLERelease`	HLE	immintrin.h	`unsigned char _interlockedbittestandset64_HLERelease(__i *, __int64);`
`_InterlockedCompareExchange`		intrin.h	`long _InterlockedCompareExchange (long volatile *, long, long);`
`_InterlockedCompareExchange_HLEAcquire`	HLE	immintrin.h	`long _InterlockedCompareExchange_HLEAcquire(lon volatile *, long, long);`
`_InterlockedCompareExchange_HLERelease`	HLE	immintrin.h	`long _InterlockedCompareExchange_HLERelease(lon volatile *, long, long);`
`_InterlockedCompareExchange_np`		intrin.h	`long _InterlockedCompareExchange_np (long *, long, long);`
`_InterlockedCompareExchange128`		intrin.h	`unsigned char _InterlockedCompareExchange128(__int64 volatile *, __int64, __int64, __int64*);`
`_InterlockedCompareExchange128_np`		intrin.h	`unsigned char _InterlockedCompareExchange128(__int64 volatile *, __int64, __int64, __int64*);`
`_InterlockedCompareExchange16`		intrin.h	`short _InterlockedCompareExchange16(short volatile *, short, short);`
`_InterlockedCompareExchange16_np`		intrin.h	`short _InterlockedCompareExchange16_np(short volatile *, short, short);`
`_InterlockedCompareExchange64`		intrin.h	`__int64 _InterlockedCompareExchange64(__int64 volatile *, __int64, __int64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_InterlockedCompareExchange64_HLEAcquire`	HLE	immintrin.h	`__int64 _InterlockedCompareExchange64_HLEAcquire(_ volatile *, __int64, __int64);`
`_InterlockedCompareExchange64_HLERelease`	HLE	immintrin.h	`__int64 _InterlockedCompareExchange64_HLERelease(_ volatile *, __int64, __int64);`
`_InterlockedCompareExchange64_np`		intrin.h	`__int64 _InterlockedCompareExchange64_np(__int64 *, __int64, __int64);`
`_InterlockedCompareExchange8`		intrin.h	`char _InterlockedCompareExchange8(char volatile *, char, char);`
`_InterlockedCompareExchangePointer`		intrin.h	`void *_InterlockedCompareExchangePointer (void *volatile *, void *, void *);`
`_InterlockedCompareExchangePointer_HLEAcquire`	HLE	immintrin.h	`void *_InterlockedCompareExchangePointer_HLEAcqu *volatile *, void *, void *);`
`_InterlockedCompareExchangePointer_HLERelease`	HLE	immintrin.h	`void *_InterlockedCompareExchangePointer_HLERel *volatile *, void *, void *);`
`_InterlockedCompareExchangePointer_np`		intrin.h	`void *_InterlockedCompareExchangePointer_np(voi **, void *, void *);`
`_InterlockedDecrement`		intrin.h	`long _InterlockedDecrement(long volatile *);`
`_InterlockedDecrement16`		intrin.h	`short _InterlockedDecrement16(short volatile *);`
`_InterlockedDecrement64`		intrin.h	`__int64 _InterlockedDecrement64(__int64 volatile *);`
`_InterlockedExchange`		intrin.h	`long _InterlockedExchange(long volatile *, long);`
`_InterlockedExchange_HLEAcquire`	HLE	immintrin.h	`long _InterlockedExchange_HLEAcquire(long volatile *, long);`
`_InterlockedExchange_HLERelease`	HLE	immintrin.h	`long _InterlockedExchange_HLERelease(long volatile *, long);`
`_InterlockedExchange16`		intrin.h	`short _InterlockedExchange16(short volatile *, short);`
`_InterlockedExchange64`		intrin.h	`__int64 _InterlockedExchange64(__int64 volatile *, __int64);`
`_InterlockedExchange64_HLEAcquire`	HLE	immintrin.h	`__int64 _InterlockedExchange64_HLEAcquire(__int64 volatile *, __int64);`
`_InterlockedExchange64_HLERelease`	HLE	immintrin.h	`__int64 _InterlockedExchange64_HLERelease(__int64 volatile *, __int64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_InterlockedExchange8`		intrin.h	`char _InterlockedExchange8(char volatile *, char);`
`_InterlockedExchangeAdd`		intrin.h	`long _InterlockedExchangeAdd(long volatile *, long);`
`_InterlockedExchangeAdd_HLEAcquire`	HLE	immintrin.h	`long _InterlockedExchangeAdd_HLEAcquire(long volatile *, long);`
`_InterlockedExchangeAdd_HLERelease`	HLE	immintrin.h	`long _InterlockedExchangeAdd_HLERelease(long volatile *, long);`
`_InterlockedExchangeAdd16`		intrin.h	`short _InterlockedExchangeAdd16(short volatile *, short);`
`_InterlockedExchangeAdd64`		intrin.h	`__int64 _InterlockedExchangeAdd64(__int64 volatile *, __int64);`
`_InterlockedExchangeAdd64_HLEAcquire`	HLE	immintrin.h	`__int64 _InterlockedExchangeAdd64_HLEAcquire(__int volatile *, __int64);`
`_InterlockedExchangeAdd64_HLERelease`	HLE	immintrin.h	`__int64 _InterlockedExchangeAdd64_HLERelease(__int volatile *, __int64);`
`_InterlockedExchangeAdd8`		intrin.h	`char _InterlockedExchangeAdd8(char volatile *, char);`
`_InterlockedExchangePointer`		intrin.h	`void * _InterlockedExchangePointer(void *volatile *, void *);`
`_InterlockedExchangePointer_HLEAcquire`	HLE	immintrin.h	`void * _InterlockedExchangePointer_HLEAcquire(voi *volatile *, void *);`
`_InterlockedExchangePointer_HLERelease`	HLE	immintrin.h	`void * _InterlockedExchangePointer_HLERelease(voi *volatile *, void *);`
`_InterlockedIncrement`		intrin.h	`long _InterlockedIncrement(long volatile *);`
`_InterlockedIncrement16`		intrin.h	`short _InterlockedIncrement16(short volatile *);`
`_InterlockedIncrement64`		intrin.h	`__int64 _InterlockedIncrement64(__int64 volatile *);`
`_InterlockedOr`		intrin.h	`long _InterlockedOr(long volatile *, long);`
`_InterlockedOr_HLEAcquire`	HLE	immintrin.h	`long _InterlockedOr_HLEAcquire(long volatile *, long);`
`_InterlockedOr_HLERelease`	HLE	immintrin.h	`long _InterlockedOr_HLERelease(long volatile *, long);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_InterlockedOr_np`		intrin.h	`long _InterlockedOr_np(long *, long);`
`_InterlockedOr16`		intrin.h	`short _InterlockedOr16(short volatile *, short);`
`_InterlockedOr16_np`		intrin.h	`short _InterlockedOr16_np(short *, short);`
`_InterlockedOr64`		intrin.h	`__int64 _InterlockedOr64(__int64 volatile *, __int64);`
`_InterlockedOr64_HLEAcquire`	HLE	immintrin.h	`__int64 _InterlockedOr64_HLEAcquire(__int64 volatile *, __int64);`
`_InterlockedOr64_HLERelease`	HLE	immintrin.h	`__int64 _InterlockedOr64_HLERelease(__int64 volatile *, __int64);`
`_InterlockedOr64_np`		intrin.h	`__int64 _InterlockedOr64_np(__int64 *, __int64);`
`_InterlockedOr8`		intrin.h	`char _InterlockedOr8(char volatile *, char);`
`_InterlockedOr8_np`		intrin.h	`char _InterlockedOr8_np(char *, char);`
`_InterlockedXor`		intrin.h	`long _InterlockedXor(long volatile *, long);`
`_InterlockedXor_HLEAcquire`	HLE	immintrin.h	`long _InterlockedXor_HLEAcquire(long volatile *, long);`
`_InterlockedXor_HLERelease`	HLE	immintrin.h	`long _InterlockedXor_HLERelease(long volatile *, long);`
`_InterlockedXor_np`		intrin.h	`long _InterlockedXor_np(long *, long);`
`_InterlockedXor16`		intrin.h	`short _InterlockedXor16(short volatile *, short);`
`_InterlockedXor16_np`		intrin.h	`short _InterlockedXor16_np(short *, short);`
`_InterlockedXor64`		intrin.h	`__int64 _InterlockedXor64(__int64 volatile *, __int64);`
`_InterlockedXor64_HLEAcquire`	HLE	immintrin.h	`__int64 _InterlockedXor64_HLEAcquire(__int64 volatile *, __int64);`
`_InterlockedXor64_HLERelease`	HLE	immintrin.h	`__int64 _InterlockedXor64_HLERelease(__int64 volatile *, __int64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_InterlockedXor64_np`		intrin.h	`__int64 _InterlockedXor64_np(__int64 *, __int64);`
`_InterlockedXor8`		intrin.h	`char _InterlockedXor8(char volatile *, char);`
`_InterlockedXor8_np`		intrin.h	`char _InterlockedXor8_np(char *, char);`
`__invlpg`		intrin.h	`void __invlpg(void*);`
`_invpcid`	INVPCID	immintrin.h	`void _invpcid(unsigned int, void *);`
`__inword`		intrin.h	`unsigned short __inword(unsigned short);`
`__inwordstring`		intrin.h	`void __inwordstring(unsigned short, unsigned short *, unsigned long);`
`_lgdt`		intrin.h	`void _lgdt(void*);`
`__lidt`		intrin.h	`void __lidt(void*);`
`__ll_lshift`		intrin.h	`unsigned __int64 [pascal/cdecl] __ll_lshift(unsigned __int64, int);`
`__ll_rshift`		intrin.h	`__int64 [pascal/cdecl] __ll_rshift(__int64, int);`
`__llwpcb`	LWP	ammintrin.h	`void __llwpcb(void *);`
`_loadbe_i16`	MOVBE	immintrin.h	`short _loadbe_i16(void const*);` [Macro]
`_loadbe_i32`	MOVBE	immintrin.h	`int _loadbe_i32(void const*);` [Macro]
`_loadbe_i64`	MOVBE	immintrin.h	`__int64 _loadbe_i64(void const*);` [Macro]
`_load_be_u16`	MOVBE	immintrin.h	`unsigned short _load_be_u16(void const*);` [Macro]
`_load_be_u32`	MOVBE	immintrin.h	`unsigned int _load_be_u32(void const*);` [Macro]
`_load_be_u64`	MOVBE	immintrin.h	`unsigned __int64 _load_be_u64(void const*);` [Macro]

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__lwpins32`	LWP	ammintrin.h	`unsigned char __lwpins32(unsigned int, unsigned int, unsigned int);`
`__lwpins64`	LWP	ammintrin.h	`unsigned char __lwpins64(unsigned __int64, unsigned int, unsigned int);`
`__lwpval32`	LWP	ammintrin.h	`void __lwpval32(unsigned int, unsigned int, unsigned int);`
`__lwpval64`	LWP	ammintrin.h	`void __lwpval64(unsigned __int64, unsigned int, unsigned int);`
`__lzcnt`	LZCNT	intrin.h	`unsigned int __lzcnt(unsigned int);`
`_lzcnt_u32`	BMI	ammintrin.h, immintrin.h	`unsigned int _lzcnt_u32(unsigned int);`
`_lzcnt_u64`	BMI	ammintrin.h, immintrin.h	`unsigned __int64 _lzcnt_u64(unsigned __int64);`
`__lzcnt16`	LZCNT	intrin.h	`unsigned short __lzcnt16(unsigned short);`
`__lzcnt64`	LZCNT	intrin.h	`unsigned __int64 __lzcnt64(unsigned __int64);`
`_m_prefetch`	3DNOW	intrin.h	`void _m_prefetch(void*);`
`_m_prefetchw`	3DNOW	intrin.h	`void _m_prefetchw(void*);`
`_mm_abs_epi16`	SSSE3	intrin.h	`__m128i _mm_abs_epi16(__m128i);`
`_mm_abs_epi32`	SSSE3	intrin.h	`__m128i _mm_abs_epi32(__m128i);`
`_mm_abs_epi8`	SSSE3	intrin.h	`__m128i _mm_abs_epi8(__m128i);`
`_mm_add_epi16`	SSE2	intrin.h	`__m128i _mm_add_epi16(__m128i, __m128i);`
`_mm_add_epi32`	SSE2	intrin.h	`__m128i _mm_add_epi32(__m128i, __m128i);`
`_mm_add_epi64`	SSE2	intrin.h	`__m128i _mm_add_epi64(__m128i, __m128i);`
`_mm_add_epi8`	SSE2	intrin.h	`__m128i _mm_add_epi8(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_add_pd`	SSE2	intrin.h	`__m128d _mm_add_pd(__m128d, __m128d);`
`_mm_add_ps`	SSE	intrin.h	`__m128 _mm_add_ps(__m128, __m128);`
`_mm_add_sd`	SSE2	intrin.h	`__m128d _mm_add_sd(__m128d, __m128d);`
`_mm_add_ss`	SSE	intrin.h	`__m128 _mm_add_ss(__m128, __m128);`
`_mm_adds_epi16`	SSE2	intrin.h	`__m128i _mm_adds_epi16(__m128i, __m128i);`
`_mm_adds_epi8`	SSE2	intrin.h	`__m128i _mm_adds_epi8(__m128i, __m128i);`
`_mm_adds_epu16`	SSE2	intrin.h	`__m128i _mm_adds_epu16(__m128i, __m128i);`
`_mm_adds_epu8`	SSE2	intrin.h	`__m128i _mm_adds_epu8(__m128i, __m128i);`
`_mm_addsub_pd`	SSE3	intrin.h	`__m128d _mm_addsub_pd(__m128d, __m128d);`
`_mm_addsub_ps`	SSE3	intrin.h	`__m128 _mm_addsub_ps(__m128, __m128);`
`_mm_aesdec_si128`	AESNI	immintrin.h	`__m128i _mm_aesdec_si128(__m128i, __m128i);`
`_mm_aesdeclast_si128`	AESNI	immintrin.h	`__m128i _mm_aesdeclast_si128(__m128i, __m128i);`
`_mm_aesenc_si128`	AESNI	immintrin.h	`__m128i _mm_aesenc_si128(__m128i, __m128i);`
`_mm_aesenclast_si128`	AESNI	immintrin.h	`__m128i _mm_aesenclast_si128(__m128i, __m128i);`
`_mm_aesimc_si128`	AESNI	immintrin.h	`__m128i _mm_aesimc_si128 (__m128i);`
`_mm_aeskeygenassist_si128`	AESNI	immintrin.h	`__m128i _mm_aeskeygenassist_si128 (__m128i, const int);`
`_mm_alignr_epi8`	SSSE3	intrin.h	`__m128i _mm_alignr_epi8(__m128i, __m128i, int);`
`_mm_and_pd`	SSE2	intrin.h	`__m128d _mm_and_pd(__m128d, __m128d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_and_ps`	SSE	intrin.h	`__m128 _mm_and_ps(__m128, __m128);`
`_mm_and_si128`	SSE2	intrin.h	`__m128i _mm_and_si128(__m128i, __m128i);`
`_mm_andnot_pd`	SSE2	intrin.h	`__m128d _mm_andnot_pd(__m128d, __m128d);`
`_mm_andnot_ps`	SSE	intrin.h	`__m128 _mm_andnot_ps(__m128, __m128);`
`_mm_andnot_si128`	SSE2	intrin.h	`__m128i _mm_andnot_si128(__m128i, __m128i);`
`_mm_avg_epu16`	SSE2	intrin.h	`__m128i _mm_avg_epu16(__m128i, __m128i);`
`_mm_avg_epu8`	SSE2	intrin.h	`__m128i _mm_avg_epu8(__m128i, __m128i);`
`_mm_blend_epi16`	SSE41	intrin.h	`__m128i _mm_blend_epi16 (__m128i, __m128i, const int);`
`_mm_blend_epi32`	AVX2	immintrin.h	`__m128i _mm_blend_epi32(__m128i, __m128i, const int);`
`_mm_blend_pd`	SSE41	intrin.h	`__m128d _mm_blend_pd (__m128d, __m128d, const int);`
`_mm_blend_ps`	SSE41	intrin.h	`__m128 _mm_blend_ps (__m128, __m128, const int);`
`_mm_blendv_epi8`	SSE41	intrin.h	`__m128i _mm_blendv_epi8 (__m128i, __m128i, __m128i);`
`_mm_blendv_pd`	SSE41	intrin.h	`__m128d _mm_blendv_pd(__m128d, __m128d, __m128d);`
`_mm_blendv_ps`	SSE41	intrin.h	`__m128 _mm_blendv_ps(__m128, __m128, __m128);`
`_mm_broadcast_ss`	AVX	immintrin.h	`__m128 _mm_broadcast_ss(float const *);`
`_mm_broadcastb_epi8`	AVX2	immintrin.h	`__m128i _mm_broadcastb_epi8(__m128i);`
`_mm_broadcastd_epi32`	AVX2	immintrin.h	`__m128i _mm_broadcastd_epi32(__m128i);`
`_mm_broadcastq_epi64`	AVX2	immintrin.h	`__m128i _mm_broadcastq_epi64(__m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_broadcastsd_pd`	AVX2	immintrin.h	`__m128d _mm_broadcastsd_pd(__m128d);`
`_mm_broadcastss_ps`	AVX2	immintrin.h	`__m128 _mm_broadcastss_ps(__m128);`
`_mm_broadcastw_epi16`	AVX2	immintrin.h	`__m128i _mm_broadcastw_epi16(__m128i);`
`_mm_castpd_ps`	SSSE3	intrin.h	`__m128 _mm_castpd_ps(__m128d);`
`_mm_castpd_si128`	SSSE3	intrin.h	`__m128i _mm_castpd_si128(__m128d);`
`_mm_castps_pd`	SSSE3	intrin.h	`__m128d _mm_castps_pd(__m128);`
`_mm_castps_si128`	SSSE3	intrin.h	`__m128i _mm_castps_si128(__m128);`
`_mm_castsi128_pd`	SSSE3	intrin.h	`__m128d _mm_castsi128_pd(__m128i);`
`_mm_castsi128_ps`	SSSE3	intrin.h	`__m128 _mm_castsi128_ps(__m128i);`
`_mm_clflush`	SSE2	intrin.h	`void _mm_clflush(void const *);`
`_mm_clmulepi64_si128`	PCLMULQDQ	immintrin.h	`__m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int);`
`_mm_cmov_si128`	XOP	ammintrin.h	`__m128i _mm_cmov_si128(__m128i, __m128i, __m128i);`
`_mm_cmp_pd`	AVX	immintrin.h	`__m128d _mm_cmp_pd(__m128d, __m128d, const int);`
`_mm_cmp_ps`	AVX	immintrin.h	`__m128 _mm_cmp_ps(__m128, __m128, const int);`
`_mm_cmp_sd`	AVX	immintrin.h	`__m128d _mm_cmp_sd(__m128d, __m128d, const int);`
`_mm_cmp_ss`	AVX	immintrin.h	`__m128 _mm_cmp_ss(__m128, __m128, const int);`
`_mm_cmpeq_epi16`	SSE2	intrin.h	`__m128i _mm_cmpeq_epi16(__m128i, __m128i);`
`_mm_cmpeq_epi32`	SSE2	intrin.h	`__m128i _mm_cmpeq_epi32(__m128i, __m128i);`
`_mm_cmpeq_epi64`	SSE41	intrin.h	`__m128i _mm_cmpeq_epi64(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cmpeq_epi8`	SSE2	intrin.h	`__m128i _mm_cmpeq_epi8(__m128i, __m128i);`
`_mm_cmpeq_pd`	SSE2	intrin.h	`__m128d _mm_cmpeq_pd(__m128d, __m128d);`
`_mm_cmpeq_ps`	SSE	intrin.h	`__m128 _mm_cmpeq_ps(__m128, __m128);`
`_mm_cmpeq_sd`	SSE2	intrin.h	`__m128d _mm_cmpeq_sd(__m128d, __m128d);`
`_mm_cmpeq_ss`	SSE	intrin.h	`__m128 _mm_cmpeq_ss(__m128, __m128);`
`_mm_cmpestra`	SSE42	intrin.h	`int _mm_cmpestra(__m128i, int, __m128i, int, const int);`
`_mm_cmpestrc`	SSE42	intrin.h	`int _mm_cmpestrc(__m128i, int, __m128i, int, const int);`
`_mm_cmpestri`	SSE42	intrin.h	`int _mm_cmpestri(__m128i, int, __m128i, int, const int);`
`_mm_cmpestrm`	SSE42	intrin.h	`__m128i _mm_cmpestrm(__m128i, int, __m128i, int, const int);`
`_mm_cmpestro`	SSE42	intrin.h	`int _mm_cmpestro(__m128i, int, __m128i, int, const int);`
`_mm_cmpestrs`	SSE42	intrin.h	`int _mm_cmpestrs(__m128i, int, __m128i, int, const int);`
`_mm_cmpestrz`	SSE42	intrin.h	`int _mm_cmpestrz(__m128i, int, __m128i, int, const int);`
`_mm_cmpge_pd`	SSE2	intrin.h	`__m128d _mm_cmpge_pd(__m128d, __m128d);`
`_mm_cmpge_ps`	SSE	intrin.h	`__m128 _mm_cmpge_ps(__m128, __m128);`
`_mm_cmpge_sd`	SSE2	intrin.h	`__m128d _mm_cmpge_sd(__m128d, __m128d);`
`_mm_cmpge_ss`	SSE	intrin.h	`__m128 _mm_cmpge_ss(__m128, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cmpgt_epi16`	SSE2	intrin.h	`__m128i _mm_cmpgt_epi16(__m128i, __m128i);`
`_mm_cmpgt_epi32`	SSE2	intrin.h	`__m128i _mm_cmpgt_epi32(__m128i, __m128i);`
`_mm_cmpgt_epi64`	SSE42	intrin.h	`__m128i _mm_cmpgt_epi64(__m128i, __m128i);`
`_mm_cmpgt_epi8`	SSE2	intrin.h	`__m128i _mm_cmpgt_epi8(__m128i, __m128i);`
`_mm_cmpgt_pd`	SSE2	intrin.h	`__m128d _mm_cmpgt_pd(__m128d, __m128d);`
`_mm_cmpgt_ps`	SSE	intrin.h	`__m128 _mm_cmpgt_ps(__m128, __m128);`
`_mm_cmpgt_sd`	SSE2	intrin.h	`__m128d _mm_cmpgt_sd(__m128d, __m128d);`
`_mm_cmpgt_ss`	SSE	intrin.h	`__m128 _mm_cmpgt_ss(__m128, __m128);`
`_mm_cmpistra`	SSE42	intrin.h	`int _mm_cmpistra(__m128i, __m128i, const int);`
`_mm_cmpistrc`	SSE42	intrin.h	`int _mm_cmpistrc(__m128i, __m128i, const int);`
`_mm_cmpistri`	SSE42	intrin.h	`int _mm_cmpistri(__m128i, __m128i, const int);`
`_mm_cmpistrm`	SSE42	intrin.h	`__m128i _mm_cmpistrm(__m128i, __m128i, const int);`
`_mm_cmpistro`	SSE42	intrin.h	`int _mm_cmpistro(__m128i, __m128i, const int);`
`_mm_cmpistrs`	SSE42	intrin.h	`int _mm_cmpistrs(__m128i, __m128i, const int);`
`_mm_cmpistrz`	SSE42	intrin.h	`int _mm_cmpistrz(__m128i, __m128i, const int);`
`_mm_cmple_pd`	SSE2	intrin.h	`__m128d _mm_cmple_pd(__m128d, __m128d);`
`_mm_cmple_ps`	SSE	intrin.h	`__m128 _mm_cmple_ps(__m128, __m128);`
`_mm_cmple_sd`	SSE2	intrin.h	`__m128d _mm_cmple_sd(__m128d, __m128d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cmple_ss`	SSE	intrin.h	`__m128 _mm_cmple_ss(__m128, __m128);`
`_mm_cmplt_epi16`	SSE2	intrin.h	`__m128i _mm_cmplt_epi16(__m128i, __m128i);`
`_mm_cmplt_epi32`	SSE2	intrin.h	`__m128i _mm_cmplt_epi32(__m128i, __m128i);`
`_mm_cmplt_epi8`	SSE2	intrin.h	`__m128i _mm_cmplt_epi8(__m128i, __m128i);`
`_mm_cmplt_pd`	SSE2	intrin.h	`__m128d _mm_cmplt_pd(__m128d, __m128d);`
`_mm_cmplt_ps`	SSE	intrin.h	`__m128 _mm_cmplt_ps(__m128, __m128);`
`_mm_cmplt_sd`	SSE2	intrin.h	`__m128d _mm_cmplt_sd(__m128d, __m128d);`
`_mm_cmplt_ss`	SSE	intrin.h	`__m128 _mm_cmplt_ss(__m128, __m128);`
`_mm_cmpneq_pd`	SSE2	intrin.h	`__m128d _mm_cmpneq_pd(__m128d, __m128d);`
`_mm_cmpneq_ps`	SSE	intrin.h	`__m128 _mm_cmpneq_ps(__m128, __m128);`
`_mm_cmpneq_sd`	SSE2	intrin.h	`__m128d _mm_cmpneq_sd(__m128d, __m128d);`
`_mm_cmpneq_ss`	SSE	intrin.h	`__m128 _mm_cmpneq_ss(__m128, __m128);`
`_mm_cmpnge_pd`	SSE2	intrin.h	`__m128d _mm_cmpnge_pd(__m128d, __m128d);`
`_mm_cmpnge_ps`	SSE	intrin.h	`__m128 _mm_cmpnge_ps(__m128, __m128);`
`_mm_cmpnge_sd`	SSE2	intrin.h	`__m128d _mm_cmpnge_sd(__m128d, __m128d);`
`_mm_cmpnge_ss`	SSE	intrin.h	`__m128 _mm_cmpnge_ss(__m128, __m128);`
`_mm_cmpngt_pd`	SSE2	intrin.h	`__m128d _mm_cmpngt_pd(__m128d, __m128d);`
`_mm_cmpngt_ps`	SSE	intrin.h	`__m128 _mm_cmpngt_ps(__m128, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cmpngt_sd`	SSE2	intrin.h	`__m128d _mm_cmpngt_sd(__m128d, __m128d);`
`_mm_cmpngt_ss`	SSE	intrin.h	`__m128 _mm_cmpngt_ss(__m128, __m128);`
`_mm_cmpnle_pd`	SSE2	intrin.h	`__m128d _mm_cmpnle_pd(__m128d, __m128d);`
`_mm_cmpnle_ps`	SSE	intrin.h	`__m128 _mm_cmpnle_ps(__m128, __m128);`
`_mm_cmpnle_sd`	SSE2	intrin.h	`__m128d _mm_cmpnle_sd(__m128d, __m128d);`
`_mm_cmpnle_ss`	SSE	intrin.h	`__m128 _mm_cmpnle_ss(__m128, __m128);`
`_mm_cmpnlt_pd`	SSE2	intrin.h	`__m128d _mm_cmpnlt_pd(__m128d, __m128d);`
`_mm_cmpnlt_ps`	SSE	intrin.h	`__m128 _mm_cmpnlt_ps(__m128, __m128);`
`_mm_cmpnlt_sd`	SSE2	intrin.h	`__m128d _mm_cmpnlt_sd(__m128d, __m128d);`
`_mm_cmpnlt_ss`	SSE	intrin.h	`__m128 _mm_cmpnlt_ss(__m128, __m128);`
`_mm_cmpord_pd`	SSE2	intrin.h	`__m128d _mm_cmpord_pd(__m128d, __m128d);`
`_mm_cmpord_ps`	SSE	intrin.h	`__m128 _mm_cmpord_ps(__m128, __m128);`
`_mm_cmpord_sd`	SSE2	intrin.h	`__m128d _mm_cmpord_sd(__m128d, __m128d);`
`_mm_cmpord_ss`	SSE	intrin.h	`__m128 _mm_cmpord_ss(__m128, __m128);`
`_mm_cmpunord_pd`	SSE2	intrin.h	`__m128d _mm_cmpunord_pd(__m128d, __m128d);`
`_mm_cmpunord_ps`	SSE	intrin.h	`__m128 _mm_cmpunord_ps(__m128, __m128);`
`_mm_cmpunord_sd`	SSE2	intrin.h	`__m128d _mm_cmpunord_sd(__m128d, __m128d);`
`_mm_cmpunord_ss`	SSE	intrin.h	`__m128 _mm_cmpunord_ss(__m128, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_com_epi16`	XOP	ammintrin.h	`__m128i _mm_com_epi16(__m128i, __m128i, int);`
`_mm_com_epi32`	XOP	ammintrin.h	`__m128i _mm_com_epi32(__m128i, __m128i, int);`
`_mm_com_epi64`	XOP	ammintrin.h	`__m128i _mm_com_epi32(__m128i, __m128i, int);`
`_mm_com_epi8`	XOP	ammintrin.h	`__m128i _mm_com_epi8(__m128i, __m128i, int);`
`_mm_com_epu16`	XOP	ammintrin.h	`__m128i _mm_com_epu16(__m128i, __m128i, int);`
`_mm_com_epu32`	XOP	ammintrin.h	`__m128i _mm_com_epu32(__m128i, __m128i, int);`
`_mm_com_epu64`	XOP	ammintrin.h	`__m128i _mm_com_epu32(__m128i, __m128i, int);`
`_mm_com_epu8`	XOP	ammintrin.h	`__m128i _mm_com_epu8(__m128i, __m128i, int);`
`_mm_comieq_sd`	SSE2	intrin.h	`int _mm_comieq_sd(__m128d, __m128d);`
`_mm_comieq_ss`	SSE	intrin.h	`int _mm_comieq_ss(__m128, __m128);`
`_mm_comige_sd`	SSE2	intrin.h	`int _mm_comige_sd(__m128d, __m128d);`
`_mm_comige_ss`	SSE	intrin.h	`int _mm_comige_ss(__m128, __m128);`
`_mm_comigt_sd`	SSE2	intrin.h	`int _mm_comigt_sd(__m128d, __m128d);`
`_mm_comigt_ss`	SSE	intrin.h	`int _mm_comigt_ss(__m128, __m128);`
`_mm_comile_sd`	SSE2	intrin.h	`int _mm_comile_sd(__m128d, __m128d);`
`_mm_comile_ss`	SSE	intrin.h	`int _mm_comile_ss(__m128, __m128);`
`_mm_comilt_sd`	SSE2	intrin.h	`int _mm_comilt_sd(__m128d, __m128d);`
`_mm_comilt_ss`	SSE	intrin.h	`int _mm_comilt_ss(__m128, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_comineq_sd`	SSE2	intrin.h	`int _mm_comineq_sd(__m128d, __m128d);`
`_mm_comineq_ss`	SSE	intrin.h	`int _mm_comineq_ss(__m128, __m128);`
`_mm_crc32_u16`	SSE42	intrin.h	`unsigned int _mm_crc32_u16(unsigned int, unsigned short);`
`_mm_crc32_u32`	SSE42	intrin.h	`unsigned int _mm_crc32_u32(unsigned int, unsigned int);`
`_mm_crc32_u64`	SSE42	intrin.h	`unsigned __int64 _mm_crc32_u64(unsigned __int64, unsigned __int64);`
`_mm_crc32_u8`	SSE42	intrin.h	`unsigned int _mm_crc32_u8(unsigned int, unsigned char);`
`_mm_cvt_si2ss`	SSE	intrin.h	`__m128 _mm_cvt_si2ss(__m128, int);`
`_mm_cvt_ss2si`	SSE	intrin.h	`int _mm_cvt_ss2si(__m128);`
`_mm_cvtepi16_epi32`	SSE41	intrin.h	`__m128i _mm_cvtepi16_epi32(__m128i);`
`_mm_cvtepi16_epi64`	SSE41	intrin.h	`__m128i _mm_cvtepi16_epi64(__m128i);`
`_mm_cvtepi32_epi64`	SSE41	intrin.h	`__m128i _mm_cvtepi32_epi64(__m128i);`
`_mm_cvtepi32_pd`	SSE2	intrin.h	`__m128d _mm_cvtepi32_pd(__m128i);`
`_mm_cvtepi32_ps`	SSE2	intrin.h	`__m128 _mm_cvtepi32_ps(__m128i);`
`_mm_cvtepi8_epi16`	SSE41	intrin.h	`__m128i _mm_cvtepi8_epi16 (__m128i);`
`_mm_cvtepi8_epi32`	SSE41	intrin.h	`__m128i _mm_cvtepi8_epi32 (__m128i);`
`_mm_cvtepi8_epi64`	SSE41	intrin.h	`__m128i _mm_cvtepi8_epi64 (__m128i);`
`_mm_cvtepu16_epi32`	SSE41	intrin.h	`__m128i _mm_cvtepu16_epi32(__m128i);`
`_mm_cvtepu16_epi64`	SSE41	intrin.h	`__m128i _mm_cvtepu16_epi64(__m128i);`
`_mm_cvtepu32_epi64`	SSE41	intrin.h	`__m128i _mm_cvtepu32_epi64(__m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cvtepu8_epi16`	SSE41	intrin.h	`__m128i _mm_cvtepu8_epi16 (__m128i);`
`_mm_cvtepu8_epi32`	SSE41	intrin.h	`__m128i _mm_cvtepu8_epi32 (__m128i);`
`_mm_cvtepu8_epi64`	SSE41	intrin.h	`__m128i _mm_cvtepu8_epi64 (__m128i);`
`_mm_cvtpd_epi32`	SSE2	intrin.h	`__m128i _mm_cvtpd_epi32(__m128d);`
`_mm_cvtpd_ps`	SSE2	intrin.h	`__m128 _mm_cvtpd_ps(__m128d);`
`_mm_cvtph_ps`	F16C	immintrin.h	`__m128 _mm_cvtph_ps(__m128i);`
`_mm_cvtps_epi32`	SSE2	intrin.h	`__m128i _mm_cvtps_epi32(__m128);`
`_mm_cvtps_pd`	SSE2	intrin.h	`__m128d _mm_cvtps_pd(__m128);`
`_mm_cvtps_ph`	F16C	immintrin.h	`__m128i _mm_cvtps_ph(__m128, const int);`
`_mm_cvtsd_f64`	SSSE3	intrin.h	`double _mm_cvtsd_f64(__m128d);`
`_mm_cvtsd_si32`	SSE2	intrin.h	`int _mm_cvtsd_si32(__m128d);`
`_mm_cvtsd_si64`	SSE2	intrin.h	`__int64 _mm_cvtsd_si64(__m128d);`
`_mm_cvtsd_si64x`	SSE2	intrin.h	`__int64 _mm_cvtsd_si64x(__m128d);`
`_mm_cvtsd_ss`	SSE2	intrin.h	`__m128 _mm_cvtsd_ss(__m128, __m128d);`
`_mm_cvtsi128_si32`	SSE2	intrin.h	`int _mm_cvtsi128_si32(__m128i);`
`_mm_cvtsi128_si64`	SSE2	intrin.h	`__int64 _mm_cvtsi128_si64(__m128i);`
`_mm_cvtsi128_si64x`	SSE2	intrin.h	`__int64 _mm_cvtsi128_si64x(__m128i);`
`_mm_cvtsi32_sd`	SSE2	intrin.h	`__m128d _mm_cvtsi32_sd(__m128d, int);`
`_mm_cvtsi32_si128`	SSE2	intrin.h	`__m128i _mm_cvtsi32_si128(int);`
`_mm_cvtsi64_sd`	SSE2	intrin.h	`__m128d _mm_cvtsi64_sd(__m128d, __int64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_cvtsi64_si128`	SSE2	intrin.h	`__m128i _mm_cvtsi64_si128(__int64);`
`_mm_cvtsi64_ss`	SSE	intrin.h	`__m128 _mm_cvtsi64_ss(__m128, __int64);`
`_mm_cvtsi64x_sd`	SSE2	intrin.h	`__m128d _mm_cvtsi64x_sd(__m128d, __int64);`
`_mm_cvtsi64x_si128`	SSE2	intrin.h	`__m128i _mm_cvtsi64x_si128(__int64);`
`_mm_cvtsi64x_ss`	SSE2	intrin.h	`__m128 _mm_cvtsi64x_ss(__m128, __int64);`
`_mm_cvtss_f32`	SSSE3	intrin.h	`float _mm_cvtss_f32(__m128);`
`_mm_cvtss_sd`	SSE2	intrin.h	`__m128d _mm_cvtss_sd(__m128d, __m128);`
`_mm_cvtss_si64`	SSE	intrin.h	`__int64 _mm_cvtss_si64(__m128);`
`_mm_cvtss_si64x`	SSE2	intrin.h	`__int64 _mm_cvtss_si64x(__m128);`
`_mm_cvtt_ss2si`	SSE	intrin.h	`int _mm_cvtt_ss2si(__m128);`
`_mm_cvttpd_epi32`	SSE2	intrin.h	`__m128i _mm_cvttpd_epi32(__m128d);`
`_mm_cvttps_epi32`	SSE2	intrin.h	`__m128i _mm_cvttps_epi32(__m128);`
`_mm_cvttsd_si32`	SSE2	intrin.h	`int _mm_cvttsd_si32(__m128d);`
`_mm_cvttsd_si64`	SSE2	intrin.h	`__int64 _mm_cvttsd_si64(__m128d);`
`_mm_cvttsd_si64x`	SSE2	intrin.h	`__int64 _mm_cvttsd_si64x(__m128d);`
`_mm_cvttss_si64`	SSE2	intrin.h	`__int64 _mm_cvttss_si64(__m128);`
`_mm_cvttss_si64x`	SSE2	intrin.h	`__int64 _mm_cvttss_si64x(__m128);`
`_mm_div_pd`	SSE2	intrin.h	`__m128d _mm_div_pd(__m128d, __m128d);`
`_mm_div_ps`	SSE	intrin.h	`__m128 _mm_div_ps(__m128, __m128);`
`_mm_div_sd`	SSE2	intrin.h	`__m128d _mm_div_sd(__m128d, __m128d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_div_ss`	SSE	intrin.h	`__m128 _mm_div_ss(__m128, __m128);`
`_mm_dp_pd`	SSE41	intrin.h	`__m128d _mm_dp_pd(__m128d, __m128d, const int);`
`_mm_dp_ps`	SSE41	intrin.h	`__m128 _mm_dp_ps(__m128, __m128, const int);`
`_mm_extract_epi16`	SSE2	intrin.h	`int _mm_extract_epi16(__m128i, int);`
`_mm_extract_epi32`	SSE41	intrin.h	`int _mm_extract_epi32(__m128i, const int);`
`_mm_extract_epi64`	SSE41	intrin.h	`__int64 _mm_extract_epi64(__m128i, const int);`
`_mm_extract_epi8`	SSE41	intrin.h	`int _mm_extract_epi8 (__m128i, const int);`
`_mm_extract_ps`	SSE41	intrin.h	`int _mm_extract_ps(__m128, const int);`
`_mm_extract_si64`	SSE4a	intrin.h	`__m128i _mm_extract_si64(__m128i, __m128i);`
`_mm_extracti_si64`	SSE4a	intrin.h	`__m128i _mm_extracti_si64(__m128i, int, int);`
`_mm_fmadd_pd`	FMA	immintrin.h	`__m128d _mm_fmadd_pd (__m128d, __m128d, __m128d);`
`_mm_fmadd_ps`	FMA	immintrin.h	`__m128 _mm_fmadd_ps (__m128, __m128, __m128);`
`_mm_fmadd_sd`	FMA	immintrin.h	`__m128d _mm_fmadd_sd (__m128d, __m128d, __m128d);`
`_mm_fmadd_ss`	FMA	immintrin.h	`__m128 _mm_fmadd_ss (__m128, __m128, __m128);`
`_mm_fmaddsub_pd`	FMA	immintrin.h	`__m128d _mm_fmaddsub_pd (__m128d, __m128d, __m128d);`
`_mm_fmaddsub_ps`	FMA	immintrin.h	`__m128 _mm_fmaddsub_ps (__m128, __m128, __m128);`
`_mm_fmsub_pd`	FMA	immintrin.h	`__m128d _mm_fmsub_pd (__m128d, __m128d, __m128d);`
`_mm_fmsub_ps`	FMA	immintrin.h	`__m128 _mm_fmsub_ps (__m128, __m128, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_fmsub_sd`	FMA	immintrin.h	`__m128d _mm_fmsub_sd (__m128d, __m128d, __m128d);`
`_mm_fmsub_ss`	FMA	immintrin.h	`__m128 _mm_fmsub_ss (__m128, __m128, __m128);`
`_mm_fmsubadd_pd`	FMA	immintrin.h	`__m128d _mm_fmsubadd_pd (__m128d, __m128d, __m128d);`
`_mm_fmsubadd_ps`	FMA	immintrin.h	`__m128 _mm_fmsubadd_ps (__m128, __m128, __m128);`
`_mm_fnmadd_pd`	FMA	immintrin.h	`__m128d _mm_fnmadd_pd (__m128d, __m128d, __m128d);`
`_mm_fnmadd_ps`	FMA	immintrin.h	`__m128 _mm_fnmadd_ps (__m128, __m128, __m128);`
`_mm_fnmadd_sd`	FMA	immintrin.h	`__m128d _mm_fnmadd_sd (__m128d, __m128d, __m128d);`
`_mm_fnmadd_ss`	FMA	immintrin.h	`__m128 _mm_fnmadd_ss (__m128, __m128, __m128);`
`_mm_fnmsub_pd`	FMA	immintrin.h	`__m128d _mm_fnmsub_pd (__m128d, __m128d, __m128d);`
`_mm_fnmsub_ps`	FMA	immintrin.h	`__m128 _mm_fnmsub_ps (__m128, __m128, __m128);`
`_mm_fnmsub_sd`	FMA	immintrin.h	`__m128d _mm_fnmsub_sd (__m128d, __m128d, __m128d);`
`_mm_fnmsub_ss`	FMA	immintrin.h	`__m128 _mm_fnmsub_ss (__m128, __m128, __m128);`
`_mm_frcz_pd`	XOP	ammintrin.h	`__m128d _mm_frcz_pd(__m128d);`
`_mm_frcz_ps`	XOP	ammintrin.h	`__m128 _mm_frcz_ps(__m128);`
`_mm_frcz_sd`	XOP	ammintrin.h	`__m128d _mm_frcz_sd(__m128d, __m128d);`
`_mm_frcz_ss`	XOP	ammintrin.h	`__m128 _mm_frcz_ss(__m128, __m128);`
`_mm_getcsr`	SSE	intrin.h	`unsigned int _mm_getcsr(void);`
`_mm_hadd_epi16`	SSSE3	intrin.h	`__m128i _mm_hadd_epi16(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_hadd_epi32`	SSSE3	intrin.h	`__m128i _mm_hadd_epi32(__m128i, __m128i);`
`_mm_hadd_pd`	SSE3	intrin.h	`__m128d _mm_hadd_pd(__m128d, __m128d);`
`_mm_hadd_ps`	SSE3	intrin.h	`__m128 _mm_hadd_ps(__m128, __m128);`
`_mm_haddd_epi16`	XOP	ammintrin.h	`__m128i _mm_haddd_epi16(__m128i);`
`_mm_haddd_epi8`	XOP	ammintrin.h	`__m128i _mm_haddd_epi8(__m128i);`
`_mm_haddd_epu16`	XOP	ammintrin.h	`__m128i _mm_haddd_epu16(__m128i);`
`_mm_haddd_epu8`	XOP	ammintrin.h	`__m128i _mm_haddd_epu8(__m128i);`
`_mm_haddq_epi16`	XOP	ammintrin.h	`__m128i _mm_haddq_epi16(__m128i);`
`_mm_haddq_epi32`	XOP	ammintrin.h	`__m128i _mm_haddq_epi32(__m128i);`
`_mm_haddq_epi8`	XOP	ammintrin.h	`__m128i _mm_haddq_epi8(__m128i);`
`_mm_haddq_epu16`	XOP	ammintrin.h	`__m128i _mm_haddq_epu16(__m128i);`
`_mm_haddq_epu32`	XOP	ammintrin.h	`__m128i _mm_haddq_epu32(__m128i);`
`_mm_haddq_epu8`	XOP	ammintrin.h	`__m128i _mm_haddq_epu8(__m128i);`
`_mm_hadds_epi16`	SSSE3	intrin.h	`__m128i _mm_hadds_epi16(__m128i, __m128i);`
`_mm_haddw_epi8`	XOP	ammintrin.h	`__m128i _mm_haddw_epi8(__m128i);`
`_mm_haddw_epu8`	XOP	ammintrin.h	`__m128i _mm_haddw_epu8(__m128i);`
`_mm_hsub_epi16`	SSSE3	intrin.h	`__m128i _mm_hsub_epi16(__m128i, __m128i);`
`_mm_hsub_epi32`	SSSE3	intrin.h	`__m128i _mm_hsub_epi32(__m128i, __m128i);`
`_mm_hsub_pd`	SSE3	intrin.h	`__m128d _mm_hsub_pd(__m128d, __m128d);`
`_mm_hsub_ps`	SSE3	intrin.h	`__m128 _mm_hsub_ps(__m128, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_hsubd_epi16`	XOP	ammintrin.h	`__m128i _mm_hsubd_epi16(__m128i);`
`_mm_hsubq_epi32`	XOP	ammintrin.h	`__m128i _mm_hsubq_epi32(__m128i);`
`_mm_hsubs_epi16`	SSSE3	intrin.h	`__m128i _mm_hsubs_epi16(__m128i, __m128i);`
`_mm_hsubw_epi8`	XOP	ammintrin.h	`__m128i _mm_hsubw_epi8(__m128i);`
`_mm_i32gather_epi32`	AVX2	immintrin.h	`__m128i _mm_i32gather_epi32(int const *, __m128i, const int);`
`_mm_i32gather_epi64`	AVX2	immintrin.h	`__m128i _mm_i32gather_epi64(__int64 const *, __m128i, const int);`
`_mm_i32gather_pd`	AVX2	immintrin.h	`__m128d _mm_i32gather_pd(double const *, __m128i, const int);`
`_mm_i32gather_ps`	AVX2	immintrin.h	`__m128 _mm_i32gather_ps(float const *, __m128i, const int);`
`_mm_i64gather_epi32`	AVX2	immintrin.h	`__m128i _mm_i64gather_epi32(int const *, __m128i, const int);`
`_mm_i64gather_epi64`	AVX2	immintrin.h	`__m128i _mm_i64gather_epi64(__int64 const *, __m128i, const int);`
`_mm_i64gather_pd`	AVX2	immintrin.h	`__m128d _mm_i64gather_pd(double const *, __m128i, const int);`
`_mm_i64gather_ps`	AVX2	immintrin.h	`__m128 _mm_i64gather_ps(float const *, __m128i, const int);`
`_mm_insert_epi16`	SSE2	intrin.h	`__m128i _mm_insert_epi16(__m128i, int, int);`
`_mm_insert_epi32`	SSE41	intrin.h	`__m128i _mm_insert_epi32(__m128i, int, const int);`
`_mm_insert_epi64`	SSE41	intrin.h	`__m128i _mm_insert_epi64(__m128i, __int64, const int);`
`_mm_insert_epi8`	SSE41	intrin.h	`__m128i _mm_insert_epi8 (__m128i, int, const int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_insert_ps`	SSE41	intrin.h	`__m128 _mm_insert_ps(__m128, __m128, const int);`
`_mm_insert_si64`	SSE4a	intrin.h	`__m128i _mm_insert_si64(__m128i, __m128i);`
`_mm_inserti_si64`	SSE4a	intrin.h	`__m128i _mm_inserti_si64(__m128i, __m128i, int, int);`
`_mm_lddqu_si128`	SSE3	intrin.h	`__m128i _mm_lddqu_si128(__m128i const*);`
`_mm_lfence`	SSE2	intrin.h	`void _mm_lfence(void);`
`_mm_load_pd`	SSE2	intrin.h	`__m128d _mm_load_pd(double*);`
`_mm_load_ps`	SSE	intrin.h	`__m128 _mm_load_ps(float*);`
`_mm_load_ps1`	SSE	intrin.h	`__m128 _mm_load_ps1(float*);`
`_mm_load_sd`	SSE2	intrin.h	`__m128d _mm_load_sd(double*);`
`_mm_load_si128`	SSE2	intrin.h	`__m128i _mm_load_si128(__m128i*);`
`_mm_load_ss`	SSE	intrin.h	`__m128 _mm_load_ss(float*);`
`_mm_load1_pd`	SSE2	intrin.h	`__m128d _mm_load1_pd(double*);`
`_mm_loaddup_pd`	SSE3	intrin.h	`__m128d _mm_loaddup_pd(double const*);`
`_mm_loadh_pd`	SSE2	intrin.h	`__m128d _mm_loadh_pd(__m128d, double*);`
`_mm_loadh_pi`	SSE	intrin.h	`__m128 _mm_loadh_pi(__m128, __m64*);`
`_mm_loadl_epi64`	SSE2	intrin.h	`__m128i _mm_loadl_epi64(__m128i*);`
`_mm_loadl_pd`	SSE2	intrin.h	`__m128d _mm_loadl_pd(__m128d, double*);`
`_mm_loadl_pi`	SSE	intrin.h	`__m128 _mm_loadl_pi(__m128, __m64*);`
`_mm_loadr_pd`	SSE2	intrin.h	`__m128d _mm_loadr_pd(double*);`
`_mm_loadr_ps`	SSE	intrin.h	`__m128 _mm_loadr_ps(float*);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_loadu_pd`	SSE2	intrin.h	`__m128d _mm_loadu_pd(double*);`
`_mm_loadu_ps`	SSE	intrin.h	`__m128 _mm_loadu_ps(float*);`
`_mm_loadu_si128`	SSE2	intrin.h	`__m128i _mm_loadu_si128(__m128i*);`
`_mm_macc_epi16`	XOP	ammintrin.h	`__m128i _mm_macc_epi16(__m128i, __m128i, __m128i);`
`_mm_macc_epi32`	XOP	ammintrin.h	`__m128i _mm_macc_epi32(__m128i, __m128i, __m128i);`
`_mm_macc_pd`	FMA4	ammintrin.h	`__m128d _mm_macc_pd(__m128d, __m128d, __m128d);`
`_mm_macc_ps`	FMA4	ammintrin.h	`__m128 _mm_macc_ps(__m128, __m128, __m128);`
`_mm_macc_sd`	FMA4	ammintrin.h	`__m128d _mm_macc_sd(__m128d, __m128d, __m128d);`
`_mm_macc_ss`	FMA4	ammintrin.h	`__m128 _mm_macc_ss(__m128, __m128, __m128);`
`_mm_maccd_epi16`	XOP	ammintrin.h	`__m128i _mm_maccd_epi16(__m128i, __m128i, __m128i);`
`_mm_macchi_epi32`	XOP	ammintrin.h	`__m128i _mm_macchi_epi32(__m128i, __m128i, __m128i);`
`_mm_macclo_epi32`	XOP	ammintrin.h	`__m128i _mm_macclo_epi32(__m128i, __m128i, __m128i);`
`_mm_maccs_epi16`	XOP	ammintrin.h	`__m128i _mm_maccs_epi16(__m128i, __m128i, __m128i);`
`_mm_maccs_epi32`	XOP	ammintrin.h	`__m128i _mm_maccs_epi32(__m128i, __m128i, __m128i);`
`_mm_maccsd_epi16`	XOP	ammintrin.h	`__m128i _mm_maccsd_epi16(__m128i, __m128i, __m128i);`
`_mm_maccshi_epi32`	XOP	ammintrin.h	`__m128i _mm_maccshi_epi32(__m128i, __m128i, __m128i);`
`_mm_maccslo_epi32`	XOP	ammintrin.h	`__m128i _mm_maccslo_epi32(__m128i, __m128i, __m128i);`
`_mm_madd_epi16`	SSE2	intrin.h	`__m128i _mm_madd_epi16(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_maddd_epi16`	XOP	ammintrin.h	`__m128i _mm_maddd_epi16(__m128i, __m128i, __m128i);`
`_mm_maddsd_epi16`	XOP	ammintrin.h	`__m128i _mm_maddsd_epi16(__m128i, __m128i, __m128i);`
`_mm_maddsub_pd`	FMA4	ammintrin.h	`__m128d _mm_maddsub_pd(__m128d, __m128d, __m128d);`
`_mm_maddsub_ps`	FMA4	ammintrin.h	`__m128 _mm_maddsub_ps(__m128, __m128, __m128);`
`_mm_maddubs_epi16`	SSSE3	intrin.h	`__m128i _mm_maddubs_epi16(__m128i, __m128i);`
`_mm_mask_i32gather_epi32`	AVX2	immintrin.h	`__m128i _mm_mask_i32gather_epi32(__m128i, int const *, __m128i, __m128i, const int);`
`_mm_mask_i32gather_epi64`	AVX2	immintrin.h	`__m128i _mm_mask_i32gather_epi64(__m128i, __int64 const *, __m128i, __m128i, const int);`
`_mm_mask_i32gather_pd`	AVX2	immintrin.h	`__m128d _mm_mask_i32gather_pd(__m128d, double const *, __m128i, __m128d, const int);`
`_mm_mask_i32gather_ps`	AVX2	immintrin.h	`__m128 _mm_mask_i32gather_ps(__m128, float const *, __m128i, __m128, const int);`
`_mm_mask_i64gather_epi32`	AVX2	immintrin.h	`__m128i _mm_mask_i64gather_epi32(__m128i, int const *, __m128i, __m128i, const int);`
`_mm_mask_i64gather_epi64`	AVX2	immintrin.h	`__m128i _mm_mask_i64gather_epi64(__m128i, __int64 const *, __m128i, __m128i, const int);`
`_mm_mask_i64gather_pd`	AVX2	immintrin.h	`__m128d _mm_mask_i64gather_pd(__m128d, double const *, __m128i, __m128d, const int);`
`_mm_mask_i64gather_ps`	AVX2	immintrin.h	`__m128 _mm_mask_i64gather_ps(__m128, float const *, __m128i, __m128, const int);`
`_mm_maskload_epi32`	AVX2	immintrin.h	`__m128i _mm_maskload_epi32(int const *, __m128i);`
`_mm_maskload_epi64`	AVX2	immintrin.h	`__m128i _mm_maskload_epi64(__int64 const *, __m128i);`
`_mm_maskload_pd`	AVX	immintrin.h	`__m128d _mm_maskload_pd(double const *, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_maskload_ps`	AVX	immintrin.h	`__m128 _mm_maskload_ps(float const *, __m128i);`
`_mm_maskmoveu_si128`	SSE2	intrin.h	`void _mm_maskmoveu_si128(__m128i, __m128i, char*);`
`_mm_maskstore_epi32`	AVX2	immintrin.h	`void _mm_maskstore_epi32(int *, __m128i, __m128i);`
`_mm_maskstore_epi64`	AVX2	immintrin.h	`void _mm_maskstore_epi64(__int64 *, __m128i, __m128i);`
`_mm_maskstore_pd`	AVX	immintrin.h	`void _mm_maskstore_pd(double *, __m128i, __m128d);`
`_mm_maskstore_ps`	AVX	immintrin.h	`void _mm_maskstore_ps(float *, __m128i, __m128);`
`_mm_max_epi16`	SSE2	intrin.h	`__m128i _mm_max_epi16(__m128i, __m128i);`
`_mm_max_epi32`	SSE41	intrin.h	`__m128i _mm_max_epi32(__m128i, __m128i);`
`_mm_max_epi8`	SSE41	intrin.h	`__m128i _mm_max_epi8 (__m128i, __m128i);`
`_mm_max_epu16`	SSE41	intrin.h	`__m128i _mm_max_epu16(__m128i, __m128i);`
`_mm_max_epu32`	SSE41	intrin.h	`__m128i _mm_max_epu32(__m128i, __m128i);`
`_mm_max_epu8`	SSE2	intrin.h	`__m128i _mm_max_epu8(__m128i, __m128i);`
`_mm_max_pd`	SSE2	intrin.h	`__m128d _mm_max_pd(__m128d, __m128d);`
`_mm_max_ps`	SSE	intrin.h	`__m128 _mm_max_ps(__m128, __m128);`
`_mm_max_sd`	SSE2	intrin.h	`__m128d _mm_max_sd(__m128d, __m128d);`
`_mm_max_ss`	SSE	intrin.h	`__m128 _mm_max_ss(__m128, __m128);`
`_mm_mfence`	SSE2	intrin.h	`void _mm_mfence(void);`
`_mm_min_epi16`	SSE2	intrin.h	`__m128i _mm_min_epi16(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_min_epi32`	SSE41	intrin.h	`__m128i _mm_min_epi32(__m128i, __m128i);`
`_mm_min_epi8`	SSE41	intrin.h	`__m128i _mm_min_epi8 (__m128i, __m128i);`
`_mm_min_epu16`	SSE41	intrin.h	`__m128i _mm_min_epu16(__m128i, __m128i);`
`_mm_min_epu32`	SSE41	intrin.h	`__m128i _mm_min_epu32(__m128i, __m128i);`
`_mm_min_epu8`	SSE2	intrin.h	`__m128i _mm_min_epu8(__m128i, __m128i);`
`_mm_min_pd`	SSE2	intrin.h	`__m128d _mm_min_pd(__m128d, __m128d);`
`_mm_min_ps`	SSE	intrin.h	`__m128 _mm_min_ps(__m128, __m128);`
`_mm_min_sd`	SSE2	intrin.h	`__m128d _mm_min_sd(__m128d, __m128d);`
`_mm_min_ss`	SSE	intrin.h	`__m128 _mm_min_ss(__m128, __m128);`
`_mm_minpos_epu16`	SSE41	intrin.h	`__m128i _mm_minpos_epu16(__m128i);`
`_mm_monitor`	SSE3	intrin.h	`void _mm_monitor(void const*, unsigned int, unsigned int);`
`_mm_move_epi64`	SSE2	intrin.h	`__m128i _mm_move_epi64(__m128i);`
`_mm_move_sd`	SSE2	intrin.h	`__m128d _mm_move_sd(__m128d, __m128d);`
`_mm_move_ss`	SSE	intrin.h	`__m128 _mm_move_ss(__m128, __m128);`
`_mm_movedup_pd`	SSE3	intrin.h	`__m128d _mm_movedup_pd(__m128d);`
`_mm_movehdup_ps`	SSE3	intrin.h	`__m128 _mm_movehdup_ps(__m128);`
`_mm_movehl_ps`	SSE	intrin.h	`__m128 _mm_movehl_ps(__m128, __m128);`
`_mm_moveldup_ps`	SSE3	intrin.h	`__m128 _mm_moveldup_ps(__m128);`
`_mm_movelh_ps`	SSE	intrin.h	`__m128 _mm_movelh_ps(__m128, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_movemask_epi8`	SSE2	intrin.h	`int _mm_movemask_epi8(__m128i);`
`_mm_movemask_pd`	SSE2	intrin.h	`int _mm_movemask_pd(__m128d);`
`_mm_movemask_ps`	SSE	intrin.h	`int _mm_movemask_ps(__m128);`
`_mm_mpsadbw_epu8`	SSE41	intrin.h	`__m128i _mm_mpsadbw_epu8(__m128i, __m128i, const int);`
`_mm_msub_pd`	FMA4	ammintrin.h	`__m128d _mm_msub_pd(__m128d, __m128d, __m128d);`
`_mm_msub_ps`	FMA4	ammintrin.h	`__m128 _mm_msub_ps(__m128, __m128, __m128);`
`_mm_msub_sd`	FMA4	ammintrin.h	`__m128d _mm_msub_sd(__m128d, __m128d, __m128d);`
`_mm_msub_ss`	FMA4	ammintrin.h	`__m128 _mm_msub_ss(__m128, __m128, __m128);`
`_mm_msubadd_pd`	FMA4	ammintrin.h	`__m128d _mm_msubadd_pd(__m128d, __m128d, __m128d);`
`_mm_msubadd_ps`	FMA4	ammintrin.h	`__m128 _mm_msubadd_ps(__m128, __m128, __m128);`
`_mm_mul_epi32`	SSE41	intrin.h	`__m128i _mm_mul_epi32(__m128i, __m128i);`
`_mm_mul_epu32`	SSE2	intrin.h	`__m128i _mm_mul_epu32(__m128i, __m128i);`
`_mm_mul_pd`	SSE2	intrin.h	`__m128d _mm_mul_pd(__m128d, __m128d);`
`_mm_mul_ps`	SSE	intrin.h	`__m128 _mm_mul_ps(__m128, __m128);`
`_mm_mul_sd`	SSE2	intrin.h	`__m128d _mm_mul_sd(__m128d, __m128d);`
`_mm_mul_ss`	SSE	intrin.h	`__m128 _mm_mul_ss(__m128, __m128);`
`_mm_mulhi_epi16`	SSE2	intrin.h	`__m128i _mm_mulhi_epi16(__m128i, __m128i);`
`_mm_mulhi_epu16`	SSE2	intrin.h	`__m128i _mm_mulhi_epu16(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_mulhrs_epi16`	SSSE3	intrin.h	`__m128i _mm_mulhrs_epi16(__m128i, __m128i);`
`_mm_mullo_epi16`	SSE2	intrin.h	`__m128i _mm_mullo_epi16(__m128i, __m128i);`
`_mm_mullo_epi32`	SSE41	intrin.h	`__m128i _mm_mullo_epi32(__m128i, __m128i);`
`_mm_mwait`	SSE3	intrin.h	`void _mm_mwait(unsigned int, unsigned int);`
`_mm_nmacc_pd`	FMA4	ammintrin.h	`__m128d _mm_nmacc_pd(__m128d, __m128d, __m128d);`
`_mm_nmacc_ps`	FMA4	ammintrin.h	`__m128 _mm_nmacc_ps(__m128, __m128, __m128);`
`_mm_nmacc_sd`	FMA4	ammintrin.h	`__m128d _mm_nmacc_sd(__m128d, __m128d, __m128d);`
`_mm_nmacc_ss`	FMA4	ammintrin.h	`__m128 _mm_nmacc_ss(__m128, __m128, __m128);`
`_mm_nmsub_pd`	FMA4	ammintrin.h	`__m128d _mm_nmsub_pd(__m128d, __m128d, __m128d);`
`_mm_nmsub_ps`	FMA4	ammintrin.h	`__m128 _mm_nmsub_ps(__m128, __m128, __m128);`
`_mm_nmsub_sd`	FMA4	ammintrin.h	`__m128d _mm_nmsub_sd(__m128d, __m128d, __m128d);`
`_mm_nmsub_ss`	FMA4	ammintrin.h	`__m128 _mm_nmsub_ss(__m128, __m128, __m128);`
`_mm_or_pd`	SSE2	intrin.h	`__m128d _mm_or_pd(__m128d, __m128d);`
`_mm_or_ps`	SSE	intrin.h	`__m128 _mm_or_ps(__m128, __m128);`
`_mm_or_si128`	SSE2	intrin.h	`__m128i _mm_or_si128(__m128i, __m128i);`
`_mm_packs_epi16`	SSE2	intrin.h	`__m128i _mm_packs_epi16(__m128i, __m128i);`
`_mm_packs_epi32`	SSE2	intrin.h	`__m128i _mm_packs_epi32(__m128i, __m128i);`
`_mm_packus_epi16`	SSE2	intrin.h	`__m128i _mm_packus_epi16(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_packus_epi32`	SSE41	intrin.h	`__m128i _mm_packus_epi32(__m128i, __m128i);`
`_mm_pause`	SSE2	intrin.h	`void _mm_pause(void);`
`_mm_perm_epi8`	XOP	ammintrin.h	`__m128i _mm_perm_epi8(__m128i, __m128i, __m128i);`
`_mm_permute_pd`	AVX	immintrin.h	`__m128d _mm_permute_pd(__m128d, int);`
`_mm_permute_ps`	AVX	immintrin.h	`__m128 _mm_permute_ps(__m128, int);`
`_mm_permute2_pd`	XOP	ammintrin.h	`__m128d _mm_permute2_pd(__m128d, __m128d, __m128i, int);`
`_mm_permute2_ps`	XOP	ammintrin.h	`__m128 _mm_permute2_ps(__m128, __m128, __m128i, int);`
`_mm_permutevar_pd`	AVX	immintrin.h	`__m128d _mm_permutevar_pd(__m128d, __m128i);`
`_mm_permutevar_ps`	AVX	immintrin.h	`__m128 _mm_permutevar_ps(__m128, __m128i);`
`_mm_popcnt_u32`	POPCNT	intrin.h	`int _mm_popcnt_u32(unsigned int);`
`_mm_popcnt_u64`	POPCNT	intrin.h	`__int64 _mm_popcnt_u64(unsigned __int64);`
`_mm_prefetch`	SSE	intrin.h	`void _mm_prefetch(char*, int);`
`_mm_rcp_ps`	SSE	intrin.h	`__m128 _mm_rcp_ps(__m128);`
`_mm_rcp_ss`	SSE	intrin.h	`__m128 _mm_rcp_ss(__m128);`
`_mm_rot_epi16`	XOP	ammintrin.h	`__m128i _mm_rot_epi16(__m128i, __m128i);`
`_mm_rot_epi32`	XOP	ammintrin.h	`__m128i _mm_rot_epi32(__m128i, __m128i);`
`_mm_rot_epi64`	XOP	ammintrin.h	`__m128i _mm_rot_epi64(__m128i, __m128i);`
`_mm_rot_epi8`	XOP	ammintrin.h	`__m128i _mm_rot_epi8(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_roti_epi16`	XOP	ammintrin.h	`__m128i _mm_rot_epi16(__m128i, int);`
`_mm_roti_epi32`	XOP	ammintrin.h	`__m128i _mm_rot_epi32(__m128i, int);`
`_mm_roti_epi64`	XOP	ammintrin.h	`__m128i _mm_rot_epi64(__m128i, int);`
`_mm_roti_epi8`	XOP	ammintrin.h	`__m128i _mm_rot_epi8(__m128i, int);`
`_mm_round_pd`	SSE41	intrin.h	`__m128d _mm_round_pd(__m128d, const int);`
`_mm_round_ps`	SSE41	intrin.h	`__m128 _mm_round_ps(__m128, const int);`
`_mm_round_sd`	SSE41	intrin.h	`__m128d _mm_round_sd(__m128d, __m128d, const int);`
`_mm_round_ss`	SSE41	intrin.h	`__m128 _mm_round_ss(__m128, __m128, const int);`
`_mm_rsqrt_ps`	SSE	intrin.h	`__m128 _mm_rsqrt_ps(__m128);`
`_mm_rsqrt_ss`	SSE	intrin.h	`__m128 _mm_rsqrt_ss(__m128);`
`_mm_sad_epu8`	SSE2	intrin.h	`__m128i _mm_sad_epu8(__m128i, __m128i);`
`_mm_set_epi16`	SSE2	intrin.h	`__m128i _mm_set_epi16(short, short, short, short, short, short, short, short);`
`_mm_set_epi32`	SSE2	intrin.h	`__m128i _mm_set_epi32(int, int, int, int);`
`_mm_set_epi64x`	SSE2	intrin.h	`__m128i _mm_set_epi64x(__int64, __int64);`
`_mm_set_epi8`	SSE2	intrin.h	`__m128i _mm_set_epi8(char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char);`
`_mm_set_pd`	SSE2	intrin.h	`__m128d _mm_set_pd(double, double);`
`_mm_set_ps`	SSE	intrin.h	`__m128 _mm_set_ps(float, float, float, float);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_set_ps1`	SSE	intrin.h	`__m128 _mm_set_ps1(float);`
`_mm_set_sd`	SSE2	intrin.h	`__m128d _mm_set_sd(double);`
`_mm_set_ss`	SSE	intrin.h	`__m128 _mm_set_ss(float);`
`_mm_set1_epi16`	SSE2	intrin.h	`__m128i _mm_set1_epi16(short);`
`_mm_set1_epi32`	SSE2	intrin.h	`__m128i _mm_set1_epi32(int);`
`_mm_set1_epi64x`	SSE2	intrin.h	`__m128i _mm_set1_epi64x(__int64);`
`_mm_set1_epi8`	SSE2	intrin.h	`__m128i _mm_set1_epi8(char);`
`_mm_set1_pd`	SSE2	intrin.h	`__m128d _mm_set1_pd(double);`
`_mm_setcsr`	SSE	intrin.h	`void _mm_setcsr(unsigned int);`
`_mm_setl_epi64`	SSE2	intrin.h	`__m128i _mm_setl_epi64(__m128i);`
`_mm_setr_epi16`	SSE2	intrin.h	`__m128i _mm_setr_epi16(short, short, short, short, short, short, short, short);`
`_mm_setr_epi32`	SSE2	intrin.h	`__m128i _mm_setr_epi32(int, int, int, int);`
`_mm_setr_epi8`	SSE2	intrin.h	`__m128i _mm_setr_epi8(char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char);`
`_mm_setr_pd`	SSE2	intrin.h	`__m128d _mm_setr_pd(double, double);`
`_mm_setr_ps`	SSE	intrin.h	`__m128 _mm_setr_ps(float, float, float, float);`
`_mm_setzero_pd`	SSE2	intrin.h	`__m128d _mm_setzero_pd(void);`
`_mm_setzero_ps`	SSE	intrin.h	`__m128 _mm_setzero_ps(void);`
`_mm_setzero_si128`	SSE2	intrin.h	`__m128i _mm_setzero_si128(void);`
`_mm_sfence`	SSE	intrin.h	`void _mm_sfence(void);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_sha_epi16`	XOP	ammintrin.h	`__m128i _mm_sha_epi16(__m128i, __m128i);`
`_mm_sha_epi32`	XOP	ammintrin.h	`__m128i _mm_sha_epi32(__m128i, __m128i);`
`_mm_sha_epi64`	XOP	ammintrin.h	`__m128i _mm_sha_epi64(__m128i, __m128i);`
`_mm_sha_epi8`	XOP	ammintrin.h	`__m128i _mm_sha_epi8(__m128i, __m128i);`
`_mm_shl_epi16`	XOP	ammintrin.h	`__m128i _mm_shl_epi16(__m128i, __m128i);`
`_mm_shl_epi32`	XOP	ammintrin.h	`__m128i _mm_shl_epi32(__m128i, __m128i);`
`_mm_shl_epi64`	XOP	ammintrin.h	`__m128i _mm_shl_epi64(__m128i, __m128i);`
`_mm_shl_epi8`	XOP	ammintrin.h	`__m128i _mm_shl_epi8(__m128i, __m128i);`
`_mm_shuffle_epi32`	SSE2	intrin.h	`__m128i _mm_shuffle_epi32(__m128i, int);`
`_mm_shuffle_epi8`	SSSE3	intrin.h	`__m128i _mm_shuffle_epi8(__m128i, __m128i);`
`_mm_shuffle_pd`	SSE2	intrin.h	`__m128d _mm_shuffle_pd(__m128d, __m128d, int);`
`_mm_shuffle_ps`	SSE	intrin.h	`__m128 _mm_shuffle_ps(__m128, __m128, unsigned int);`
`_mm_shufflehi_epi16`	SSE2	intrin.h	`__m128i _mm_shufflehi_epi16(__m128i, int);`
`_mm_shufflelo_epi16`	SSE2	intrin.h	`__m128i _mm_shufflelo_epi16(__m128i, int);`
`_mm_sign_epi16`	SSSE3	intrin.h	`__m128i _mm_sign_epi16(__m128i, __m128i);`
`_mm_sign_epi32`	SSSE3	intrin.h	`__m128i _mm_sign_epi32(__m128i, __m128i);`
`_mm_sign_epi8`	SSSE3	intrin.h	`__m128i _mm_sign_epi8(__m128i, __m128i);`
`_mm_sll_epi16`	SSE2	intrin.h	`__m128i _mm_sll_epi16(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_sll_epi32`	SSE2	intrin.h	`__m128i _mm_sll_epi32(__m128i, __m128i);`
`_mm_sll_epi64`	SSE2	intrin.h	`__m128i _mm_sll_epi64(__m128i, __m128i);`
`_mm_slli_epi16`	SSE2	intrin.h	`__m128i _mm_slli_epi16(__m128i, int);`
`_mm_slli_epi32`	SSE2	intrin.h	`__m128i _mm_slli_epi32(__m128i, int);`
`_mm_slli_epi64`	SSE2	intrin.h	`__m128i _mm_slli_epi64(__m128i, int);`
`_mm_slli_si128`	SSE2	intrin.h	`__m128i _mm_slli_si128(__m128i, int);`
`_mm_sllv_epi32`	AVX2	immintrin.h	`__m128i _mm_sllv_epi32(__m128i, __m128i);`
`_mm_sllv_epi64`	AVX2	immintrin.h	`__m128i _mm_sllv_epi64(__m128i, __m128i);`
`_mm_sqrt_pd`	SSE2	intrin.h	`__m128d _mm_sqrt_pd(__m128d);`
`_mm_sqrt_ps`	SSE	intrin.h	`__m128 _mm_sqrt_ps(__m128);`
`_mm_sqrt_sd`	SSE2	intrin.h	`__m128d _mm_sqrt_sd(__m128d, __m128d);`
`_mm_sqrt_ss`	SSE	intrin.h	`__m128 _mm_sqrt_ss(__m128);`
`_mm_sra_epi16`	SSE2	intrin.h	`__m128i _mm_sra_epi16(__m128i, __m128i);`
`_mm_sra_epi32`	SSE2	intrin.h	`__m128i _mm_sra_epi32(__m128i, __m128i);`
`_mm_srai_epi16`	SSE2	intrin.h	`__m128i _mm_srai_epi16(__m128i, int);`
`_mm_srai_epi32`	SSE2	intrin.h	`__m128i _mm_srai_epi32(__m128i, int);`
`_mm_srav_epi32`	AVX2	immintrin.h	`__m128i _mm_srav_epi32(__m128i, __m128i);`
`_mm_srl_epi16`	SSE2	intrin.h	`__m128i _mm_srl_epi16(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_srl_epi32`	SSE2	intrin.h	`__m128i _mm_srl_epi32(__m128i, __m128i);`
`_mm_srl_epi64`	SSE2	intrin.h	`__m128i _mm_srl_epi64(__m128i, __m128i);`
`_mm_srli_epi16`	SSE2	intrin.h	`__m128i _mm_srli_epi16(__m128i, int);`
`_mm_srli_epi32`	SSE2	intrin.h	`__m128i _mm_srli_epi32(__m128i, int);`
`_mm_srli_epi64`	SSE2	intrin.h	`__m128i _mm_srli_epi64(__m128i, int);`
`_mm_srli_si128`	SSE2	intrin.h	`__m128i _mm_srli_si128(__m128i, int);`
`_mm_srlv_epi32`	AVX2	immintrin.h	`__m128i _mm_srlv_epi32(__m128i, __m128i);`
`_mm_srlv_epi64`	AVX2	immintrin.h	`__m128i _mm_srlv_epi64(__m128i, __m128i);`
`_mm_store_pd`	SSE2	intrin.h	`void _mm_store_pd(double*, __m128d);`
`_mm_store_ps`	SSE	intrin.h	`void _mm_store_ps(float*, __m128);`
`_mm_store_ps1`	SSE	intrin.h	`void _mm_store_ps1(float*, __m128);`
`_mm_store_sd`	SSE2	intrin.h	`void _mm_store_sd(double*, __m128d);`
`_mm_store_si128`	SSE2	intrin.h	`void _mm_store_si128(__m128i*, __m128i);`
`_mm_store_ss`	SSE	intrin.h	`void _mm_store_ss(float*, __m128);`
`_mm_store1_pd`	SSE2	intrin.h	`void _mm_store1_pd(double*, __m128d);`
`_mm_storeh_pd`	SSE2	intrin.h	`void _mm_storeh_pd(double*, __m128d);`
`_mm_storeh_pi`	SSE	intrin.h	`void _mm_storeh_pi(__m64*, __m128);`
`_mm_storel_epi64`	SSE2	intrin.h	`void _mm_storel_epi64(__m128i*, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_storel_pd`	SSE2	intrin.h	`void _mm_storel_pd(double*, __m128d);`
`_mm_storel_pi`	SSE	intrin.h	`void _mm_storel_pi(__m64*, __m128);`
`_mm_storer_pd`	SSE2	intrin.h	`void _mm_storer_pd(double*, __m128d);`
`_mm_storer_ps`	SSE	intrin.h	`void _mm_storer_ps(float*, __m128);`
`_mm_storeu_pd`	SSE2	intrin.h	`void _mm_storeu_pd(double*, __m128d);`
`_mm_storeu_ps`	SSE	intrin.h	`void _mm_storeu_ps(float*, __m128);`
`_mm_storeu_si128`	SSE2	intrin.h	`void _mm_storeu_si128(__m128i*, __m128i);`
`_mm_stream_load_si128`	SSE41	intrin.h	`__m128i _mm_stream_load_si128(__m128i*);`
`_mm_stream_pd`	SSE2	intrin.h	`void _mm_stream_pd(double*, __m128d);`
`_mm_stream_ps`	SSE	intrin.h	`void _mm_stream_ps(float*, __m128);`
`_mm_stream_sd`	SSE4a	intrin.h	`void _mm_stream_sd(double*, __m128d);`
`_mm_stream_si128`	SSE2	intrin.h	`void _mm_stream_si128(__m128i*, __m128i);`
`_mm_stream_si32`	SSE2	intrin.h	`void _mm_stream_si32(int*, int);`
`_mm_stream_si64x`	SSE2	intrin.h	`void _mm_stream_si64x(__int64 *, __int64);`
`_mm_stream_ss`	SSE4a	intrin.h	`void _mm_stream_ss(float*, __m128);`
`_mm_sub_epi16`	SSE2	intrin.h	`__m128i _mm_sub_epi16(__m128i, __m128i);`
`_mm_sub_epi32`	SSE2	intrin.h	`__m128i _mm_sub_epi32(__m128i, __m128i);`
`_mm_sub_epi64`	SSE2	intrin.h	`__m128i _mm_sub_epi64(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_sub_epi8`	SSE2	intrin.h	`__m128i _mm_sub_epi8(__m128i, __m128i);`
`_mm_sub_pd`	SSE2	intrin.h	`__m128d _mm_sub_pd(__m128d, __m128d);`
`_mm_sub_ps`	SSE	intrin.h	`__m128 _mm_sub_ps(__m128, __m128);`
`_mm_sub_sd`	SSE2	intrin.h	`__m128d _mm_sub_sd(__m128d, __m128d);`
`_mm_sub_ss`	SSE	intrin.h	`__m128 _mm_sub_ss(__m128, __m128);`
`_mm_subs_epi16`	SSE2	intrin.h	`__m128i _mm_subs_epi16(__m128i, __m128i);`
`_mm_subs_epi8`	SSE2	intrin.h	`__m128i _mm_subs_epi8(__m128i, __m128i);`
`_mm_subs_epu16`	SSE2	intrin.h	`__m128i _mm_subs_epu16(__m128i, __m128i);`
`_mm_subs_epu8`	SSE2	intrin.h	`__m128i _mm_subs_epu8(__m128i, __m128i);`
`_mm_testc_pd`	AVX	immintrin.h	`int _mm_testc_pd(__m128d, __m128d);`
`_mm_testc_ps`	AVX	immintrin.h	`int _mm_testc_ps(__m128, __m128);`
`_mm_testc_si128`	SSE41	intrin.h	`int _mm_testc_si128(__m128i, __m128i);`
`_mm_testnzc_pd`	AVX	immintrin.h	`int _mm_testnzc_pd(__m128d, __m128d);`
`_mm_testnzc_ps`	AVX	immintrin.h	`int _mm_testnzc_ps(__m128, __m128);`
`_mm_testnzc_si128`	SSE41	intrin.h	`int _mm_testnzc_si128(__m128i, __m128i);`
`_mm_testz_pd`	AVX	immintrin.h	`int _mm_testz_pd(__m128d, __m128d);`
`_mm_testz_ps`	AVX	immintrin.h	`int _mm_testz_ps(__m128, __m128);`
`_mm_testz_si128`	SSE41	intrin.h	`int _mm_testz_si128(__m128i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_ucomieq_sd`	SSE2	intrin.h	`int _mm_ucomieq_sd(__m128d, __m128d);`
`_mm_ucomieq_ss`	SSE	intrin.h	`int _mm_ucomieq_ss(__m128, __m128);`
`_mm_ucomige_sd`	SSE2	intrin.h	`int _mm_ucomige_sd(__m128d, __m128d);`
`_mm_ucomige_ss`	SSE	intrin.h	`int _mm_ucomige_ss(__m128, __m128);`
`_mm_ucomigt_sd`	SSE2	intrin.h	`int _mm_ucomigt_sd(__m128d, __m128d);`
`_mm_ucomigt_ss`	SSE	intrin.h	`int _mm_ucomigt_ss(__m128, __m128);`
`_mm_ucomile_sd`	SSE2	intrin.h	`int _mm_ucomile_sd(__m128d, __m128d);`
`_mm_ucomile_ss`	SSE	intrin.h	`int _mm_ucomile_ss(__m128, __m128);`
`_mm_ucomilt_sd`	SSE2	intrin.h	`int _mm_ucomilt_sd(__m128d, __m128d);`
`_mm_ucomilt_ss`	SSE	intrin.h	`int _mm_ucomilt_ss(__m128, __m128);`
`_mm_ucomineq_sd`	SSE2	intrin.h	`int _mm_ucomineq_sd(__m128d, __m128d);`
`_mm_ucomineq_ss`	SSE	intrin.h	`int _mm_ucomineq_ss(__m128, __m128);`
`_mm_unpackhi_epi16`	SSE2	intrin.h	`__m128i _mm_unpackhi_epi16(__m128i, __m128i);`
`_mm_unpackhi_epi32`	SSE2	intrin.h	`__m128i _mm_unpackhi_epi32(__m128i, __m128i);`
`_mm_unpackhi_epi64`	SSE2	intrin.h	`__m128i _mm_unpackhi_epi64(__m128i, __m128i);`
`_mm_unpackhi_epi8`	SSE2	intrin.h	`__m128i _mm_unpackhi_epi8(__m128i, __m128i);`
`_mm_unpackhi_pd`	SSE2	intrin.h	`__m128d _mm_unpackhi_pd(__m128d, __m128d);`
`_mm_unpackhi_ps`	SSE	intrin.h	`__m128 _mm_unpackhi_ps(__m128, __m128);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm_unpacklo_epi16`	SSE2	intrin.h	`__m128i _mm_unpacklo_epi16(__m128i, __m128i);`
`_mm_unpacklo_epi32`	SSE2	intrin.h	`__m128i _mm_unpacklo_epi32(__m128i, __m128i);`
`_mm_unpacklo_epi64`	SSE2	intrin.h	`__m128i _mm_unpacklo_epi64(__m128i, __m128i);`
`_mm_unpacklo_epi8`	SSE2	intrin.h	`__m128i _mm_unpacklo_epi8(__m128i, __m128i);`
`_mm_unpacklo_pd`	SSE2	intrin.h	`__m128d _mm_unpacklo_pd(__m128d, __m128d);`
`_mm_unpacklo_ps`	SSE	intrin.h	`__m128 _mm_unpacklo_ps(__m128, __m128);`
`_mm_xor_pd`	SSE2	intrin.h	`__m128d _mm_xor_pd(__m128d, __m128d);`
`_mm_xor_ps`	SSE	intrin.h	`__m128 _mm_xor_ps(__m128, __m128);`
`_mm_xor_si128`	SSE2	intrin.h	`__m128i _mm_xor_si128(__m128i, __m128i);`
`_mm256_abs_epi16`	AVX2	immintrin.h	`__m256i _mm256_abs_epi16(__m256i);`
`_mm256_abs_epi32`	AVX2	immintrin.h	`__m256i _mm256_abs_epi32(__m256i);`
`_mm256_abs_epi8`	AVX2	immintrin.h	`__m256i _mm256_abs_epi8(__m256i);`
`_mm256_add_epi16`	AVX2	immintrin.h	`__m256i _mm256_add_epi16(__m256i, __m256i);`
`_mm256_add_epi32`	AVX2	immintrin.h	`__m256i _mm256_add_epi32(__m256i, __m256i);`
`_mm256_add_epi64`	AVX2	immintrin.h	`__m256i _mm256_add_epi64(__m256i, __m256i);`
`_mm256_add_epi8`	AVX2	immintrin.h	`__m256i _mm256_add_epi8(__m256i, __m256i);`
`_mm256_add_pd`	AVX	immintrin.h	`__m256d _mm256_add_pd(__m256d, __m256d);`
`_mm256_add_ps`	AVX	immintrin.h	`__m256 _mm256_add_ps(__m256, __m256);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_adds_epi16`	AVX2	immintrin.h	`__m256i _mm256_adds_epi16(__m256i, __m256i);`
`_mm256_adds_epi8`	AVX2	immintrin.h	`__m256i _mm256_adds_epi8(__m256i, __m256i);`
`_mm256_adds_epu16`	AVX2	immintrin.h	`__m256i _mm256_adds_epu16(__m256i, __m256i);`
`_mm256_adds_epu8`	AVX2	immintrin.h	`__m256i _mm256_adds_epu8(__m256i, __m256i);`
`_mm256_addsub_pd`	AVX	immintrin.h	`__m256d _mm256_addsub_pd(__m256d, __m256d);`
`_mm256_addsub_ps`	AVX	immintrin.h	`__m256 _mm256_addsub_ps(__m256, __m256);`
`_mm256_alignr_epi8`	AVX2	immintrin.h	`__m256i _mm256_alignr_epi8(__m256i, __m256i, const int);`
`_mm256_and_pd`	AVX	immintrin.h	`__m256d _mm256_and_pd(__m256d, __m256d);`
`_mm256_and_ps`	AVX	immintrin.h	`__m256 _mm256_and_ps(__m256, __m256);`
`_mm256_and_si256`	AVX2	immintrin.h	`__m256i _mm256_and_si256(__m256i, __m256i);`
`_mm256_andnot_pd`	AVX	immintrin.h	`__m256d _mm256_andnot_pd(__m256d, __m256d);`
`_mm256_andnot_ps`	AVX	immintrin.h	`__m256 _mm256_andnot_ps(__m256, __m256);`
`_mm256_andnot_si256`	AVX2	immintrin.h	`__m256i _mm256_andnot_si256(__m256i, __m256i);`
`_mm256_avg_epu16`	AVX2	immintrin.h	`__m256i _mm256_avg_epu16(__m256i, __m256i);`
`_mm256_avg_epu8`	AVX2	immintrin.h	`__m256i _mm256_avg_epu8(__m256i, __m256i);`
`_mm256_blend_epi16`	AVX2	immintrin.h	`__m256i _mm256_blend_epi16(__m256i, __m256i, const int);`
`_mm256_blend_epi32`	AVX2	immintrin.h	`__m256i _mm256_blend_epi32(__m256i, __m256i, const int);`
`_mm256_blend_pd`	AVX	immintrin.h	`__m256d _mm256_blend_pd(__m256d, __m256d, const int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_blend_ps`	AVX	immintrin.h	`__m256 _mm256_blend_ps(__m256, __m256, const int);`
`_mm256_blendv_epi8`	AVX2	immintrin.h	`__m256i _mm256_blendv_epi8(__m256i, __m256i, __m256i);`
`_mm256_blendv_pd`	AVX	immintrin.h	`__m256d _mm256_blendv_pd(__m256d, __m256d, __m256d);`
`_mm256_blendv_ps`	AVX	immintrin.h	`__m256 _mm256_blendv_ps(__m256, __m256, __m256);`
`_mm256_broadcast_pd`	AVX	immintrin.h	`__m256d _mm256_broadcast_pd(__m128d const *);`
`_mm256_broadcast_ps`	AVX	immintrin.h	`__m256 _mm256_broadcast_ps(__m128 const *);`
`_mm256_broadcast_sd`	AVX	immintrin.h	`__m256d _mm256_broadcast_sd(double const *);`
`_mm256_broadcast_ss`	AVX	immintrin.h	`__m256 _mm256_broadcast_ss(float const *);`
`_mm256_broadcastb_epi8`	AVX2	immintrin.h	`__m256i _mm256_broadcastb_epi8 (__m128i);`
`_mm256_broadcastd_epi32`	AVX2	immintrin.h	`__m256i _mm256_broadcastd_epi32(__m128i);`
`_mm256_broadcastq_epi64`	AVX2	immintrin.h	`__m256i _mm256_broadcastq_epi64(__m128i);`
`_mm256_broadcastsd_pd`	AVX2	immintrin.h	`__m256d _mm256_broadcastsd_pd(__m128d);`
`_mm256_broadcastsi128_si256`	AVX2	immintrin.h	`__m256i _mm256_broadcastsi128_si256(__m128i);`
`_mm256_broadcastss_ps`	AVX2	immintrin.h	`__m256 _mm256_broadcastss_ps(__m128);`
`_mm256_broadcastw_epi16`	AVX2	immintrin.h	`__m256i _mm256_broadcastw_epi16(__m128i);`
`_mm256_castpd_ps`	AVX	immintrin.h	`__m256 _mm256_castpd_ps(__m256d);`
`_mm256_castpd_si256`	AVX	immintrin.h	`__m256i _mm256_castpd_si256(__m256d);`
`_mm256_castpd128_pd256`	AVX	immintrin.h	`__m256d _mm256_castpd128_pd256(__m128d);`
`_mm256_castpd256_pd128`	AVX	immintrin.h	`__m128d _mm256_castpd256_pd128(__m256d);`
`_mm256_castps_pd`	AVX	immintrin.h	`__m256d _mm256_castps_pd(__m256);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_castps_si256`	AVX	immintrin.h	`__m256i _mm256_castps_si256(__m256);`
`_mm256_castps128_ps256`	AVX	immintrin.h	`__m256 _mm256_castps128_ps256(__m128);`
`_mm256_castps256_ps128`	AVX	immintrin.h	`__m128 _mm256_castps256_ps128(__m256);`
`_mm256_castsi128_si256`	AVX	immintrin.h	`__m256i _mm256_castsi128_si256(__m128i);`
`_mm256_castsi256_pd`	AVX	immintrin.h	`__m256d _mm256_castsi256_pd(__m256i);`
`_mm256_castsi256_ps`	AVX	immintrin.h	`__m256 _mm256_castsi256_ps(__m256i);`
`_mm256_castsi256_si128`	AVX	immintrin.h	`__m128i _mm256_castsi256_si128(__m256i);`
`_mm256_cmov_si256`	XOP	ammintrin.h	`__m256i _mm256_cmov_si256(__m256i, __m256i, __m256i);`
`_mm256_cmp_pd`	AVX	immintrin.h	`__m256d _mm256_cmp_pd(__m256d, __m256d, const int);`
`_mm256_cmp_ps`	AVX	immintrin.h	`__m256 _mm256_cmp_ps(__m256, __m256, const int);`
`_mm256_cmpeq_epi16`	AVX2	immintrin.h	`__m256i _mm256_cmpeq_epi16(__m256i, __m256i);`
`_mm256_cmpeq_epi32`	AVX2	immintrin.h	`__m256i _mm256_cmpeq_epi32(__m256i, __m256i);`
`_mm256_cmpeq_epi64`	AVX2	immintrin.h	`__m256i _mm256_cmpeq_epi64(__m256i, __m256i);`
`_mm256_cmpeq_epi8`	AVX2	immintrin.h	`__m256i _mm256_cmpeq_epi8(__m256i, __m256i);`
`_mm256_cmpgt_epi16`	AVX2	immintrin.h	`__m256i _mm256_cmpgt_epi16(__m256i, __m256i);`
`_mm256_cmpgt_epi32`	AVX2	immintrin.h	`__m256i _mm256_cmpgt_epi32(__m256i, __m256i);`
`_mm256_cmpgt_epi64`	AVX2	immintrin.h	`__m256i _mm256_cmpgt_epi64(__m256i, __m256i);`
`_mm256_cmpgt_epi8`	AVX2	immintrin.h	`__m256i _mm256_cmpgt_epi8(__m256i, __m256i);`
`_mm256_cvtepi16_epi32`	AVX2	immintrin.h	`__m256i _mm256_cvtepi16_epi32(__m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_cvtepi16_epi64`	AVX2	immintrin.h	`__m256i _mm256_cvtepi16_epi64(__m128i);`
`_mm256_cvtepi32_epi64`	AVX2	immintrin.h	`__m256i _mm256_cvtepi32_epi64(__m128i);`
`_mm256_cvtepi32_pd`	AVX	immintrin.h	`__m256d _mm256_cvtepi32_pd(__m128i);`
`_mm256_cvtepi32_ps`	AVX	immintrin.h	`__m256 _mm256_cvtepi32_ps(__m256i);`
`_mm256_cvtepi8_epi16`	AVX2	immintrin.h	`__m256i _mm256_cvtepi8_epi16(__m128i);`
`_mm256_cvtepi8_epi32`	AVX2	immintrin.h	`__m256i _mm256_cvtepi8_epi32(__m128i);`
`_mm256_cvtepi8_epi64`	AVX2	immintrin.h	`__m256i _mm256_cvtepi8_epi64(__m128i);`
`_mm256_cvtepu16_epi32`	AVX2	immintrin.h	`__m256i _mm256_cvtepu16_epi32(__m128i);`
`_mm256_cvtepu16_epi64`	AVX2	immintrin.h	`__m256i _mm256_cvtepu16_epi64(__m128i);`
`_mm256_cvtepu32_epi64`	AVX2	immintrin.h	`__m256i _mm256_cvtepu32_epi64(__m128i);`
`_mm256_cvtepu8_epi16`	AVX2	immintrin.h	`__m256i _mm256_cvtepu8_epi16(__m128i);`
`_mm256_cvtepu8_epi32`	AVX2	immintrin.h	`__m256i _mm256_cvtepu8_epi32(__m128i);`
`_mm256_cvtepu8_epi64`	AVX2	immintrin.h	`__m256i _mm256_cvtepu8_epi64(__m128i);`
`_mm256_cvtpd_epi32`	AVX	immintrin.h	`__m128i _mm256_cvtpd_epi32(__m256d);`
`_mm256_cvtpd_ps`	AVX	immintrin.h	`__m128 _mm256_cvtpd_ps(__m256d);`
`_mm256_cvtph_ps`	F16C	immintrin.h	`__m256 _mm256_cvtph_ps(__m128i);`
`_mm256_cvtps_epi32`	AVX	immintrin.h	`__m256i _mm256_cvtps_epi32(__m256);`
`_mm256_cvtps_pd`	AVX	immintrin.h	`__m256d _mm256_cvtps_pd(__m128);`
`_mm256_cvtps_ph`	F16C	immintrin.h	`__m128i _mm256_cvtps_ph(__m256, const int);`
`_mm256_cvttpd_epi32`	AVX	immintrin.h	`__m128i _mm256_cvttpd_epi32(__m256d);`
`_mm256_cvttps_epi32`	AVX	immintrin.h	`__m256i _mm256_cvttps_epi32(__m256);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_div_pd`	AVX	immintrin.h	`__m256d _mm256_div_pd(__m256d, __m256d);`
`_mm256_div_ps`	AVX	immintrin.h	`__m256 _mm256_div_ps(__m256, __m256);`
`_mm256_dp_ps`	AVX	immintrin.h	`__m256 _mm256_dp_ps(__m256, __m256, const int);`
`_mm256_extractf128_pd`	AVX	immintrin.h	`__m128d _mm256_extractf128_pd(__m256d, const int);`
`_mm256_extractf128_ps`	AVX	immintrin.h	`__m128 _mm256_extractf128_ps(__m256, const int);`
`_mm256_extractf128_si256`	AVX	immintrin.h	`__m128i _mm256_extractf128_si256(__m256i, const int);`
`_mm256_extracti128_si256`	AVX2	immintrin.h	`__m128i _mm256_extracti128_si256(__m256i, int);`
`_mm256_fmadd_pd`	FMA	immintrin.h	`__m256d _mm256_fmadd_pd (__m256d, __m256d, __m256d);`
`_mm256_fmadd_ps`	FMA	immintrin.h	`__m256 _mm256_fmadd_ps (__m256, __m256, __m256);`
`_mm256_fmaddsub_pd`	FMA	immintrin.h	`__m256d _mm256_fmaddsub_pd (__m256d, __m256d, __m256d);`
`_mm256_fmaddsub_ps`	FMA	immintrin.h	`__m256 _mm256_fmaddsub_ps (__m256, __m256, __m256);`
`_mm256_fmsub_pd`	FMA	immintrin.h	`__m256d _mm256_fmsub_pd (__m256d, __m256d, __m256d);`
`_mm256_fmsub_ps`	FMA	immintrin.h	`__m256 _mm256_fmsub_ps (__m256, __m256, __m256);`
`_mm256_fmsubadd_pd`	FMA	immintrin.h	`__m256d _mm256_fmsubadd_pd (__m256d, __m256d, __m256d);`
`_mm256_fmsubadd_ps`	FMA	immintrin.h	`__m256 _mm256_fmsubadd_ps (__m256, __m256, __m256);`
`_mm256_fnmadd_pd`	FMA	immintrin.h	`__m256d _mm256_fnmadd_pd (__m256d, __m256d, __m256d);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_fnmadd_ps`	FMA	immintrin.h	`__m256 _mm256_fnmadd_ps (__m256, __m256, __m256);`
`_mm256_fnmsub_pd`	FMA	immintrin.h	`__m256d _mm256_fnmsub_pd (__m256d, __m256d, __m256d);`
`_mm256_fnmsub_ps`	FMA	immintrin.h	`__m256 _mm256_fnmsub_ps (__m256, __m256, __m256);`
`_mm256_frcz_pd`	XOP	ammintrin.h	`__m256d _mm256_frcz_pd(__m256d);`
`_mm256_frcz_ps`	XOP	ammintrin.h	`__m256 _mm256_frcz_ps(__m256);`
`_mm256_hadd_epi16`	AVX2	immintrin.h	`__m256i _mm256_hadd_epi16(__m256i, __m256i);`
`_mm256_hadd_epi32`	AVX2	immintrin.h	`__m256i _mm256_hadd_epi32(__m256i, __m256i);`
`_mm256_hadd_pd`	AVX	immintrin.h	`__m256d _mm256_hadd_pd(__m256d, __m256d);`
`_mm256_hadd_ps`	AVX	immintrin.h	`__m256 _mm256_hadd_ps(__m256, __m256);`
`_mm256_hadds_epi16`	AVX2	immintrin.h	`__m256i _mm256_hadds_epi16(__m256i, __m256i);`
`_mm256_hsub_epi16`	AVX2	immintrin.h	`__m256i _mm256_hsub_epi16(__m256i, __m256i);`
`_mm256_hsub_epi32`	AVX2	immintrin.h	`__m256i _mm256_hsub_epi32(__m256i, __m256i);`
`_mm256_hsub_pd`	AVX	immintrin.h	`__m256d _mm256_hsub_pd(__m256d, __m256d);`
`_mm256_hsub_ps`	AVX	immintrin.h	`__m256 _mm256_hsub_ps(__m256, __m256);`
`_mm256_hsubs_epi16`	AVX2	immintrin.h	`__m256i _mm256_hsubs_epi16(__m256i, __m256i);`
`_mm256_i32gather_epi32`	AVX2	immintrin.h	`__m256i _mm256_i32gather_epi32(int const *, __m256i, const int);`
`_mm256_i32gather_epi64`	AVX2	immintrin.h	`__m256i _mm256_i32gather_epi64(__int64 const *, __m128i, const int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_i32gather_pd`	AVX2	immintrin.h	`__m256d _mm256_i32gather_pd(double const *, __m128i, const int);`
`_mm256_i32gather_ps`	AVX2	immintrin.h	`__m256 _mm256_i32gather_ps(float const *, __m256i, const int);`
`_mm256_i64gather_epi32`	AVX2	immintrin.h	`__m256i _mm256_i64gather_epi32(int const *, __m256i, const int);`
`_mm256_i64gather_epi64`	AVX2	immintrin.h	`__m256i _mm256_i64gather_epi64(__int64 const *, __m256i, const int);`
`_mm256_i64gather_pd`	AVX2	immintrin.h	`__m256d _mm256_i64gather_pd(double const *, __m256i, const int);`
`_mm256_i64gather_ps`	AVX2	immintrin.h	`__m128 _mm256_i64gather_ps(float const *, __m256i, const int);`
`_mm256_insertf128_pd`	AVX	immintrin.h	`__m256d _mm256_insertf128_pd(__m256d, __m128d, int);`
`_mm256_insertf128_ps`	AVX	immintrin.h	`__m256 _mm256_insertf128_ps(__m256, __m128, int);`
`_mm256_insertf128_si256`	AVX	immintrin.h	`__m256i _mm256_insertf128_si256(__m256i, __m128i, int);`
`_mm256_inserti128_si256`	AVX2	immintrin.h	`__m256i _mm256_inserti128_si256(__m256i, __m128i, int);`
`_mm256_lddqu_si256`	AVX	immintrin.h	`__m256i _mm256_lddqu_si256(__m256i *);`
`_mm256_load_pd`	AVX	immintrin.h	`__m256d _mm256_load_pd(double const *);`
`_mm256_load_ps`	AVX	immintrin.h	`__m256 _mm256_load_ps(float const *);`
`_mm256_load_si256`	AVX	immintrin.h	`__m256i _mm256_load_si256(__m256i *);`
`_mm256_loadu_pd`	AVX	immintrin.h	`__m256d _mm256_loadu_pd(double const *);`
`_mm256_loadu_ps`	AVX	immintrin.h	`__m256 _mm256_loadu_ps(float const *);`
`_mm256_loadu_si256`	AVX	immintrin.h	`__m256i _mm256_loadu_si256(__m256i *);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_macc_pd`	FMA4	ammintrin.h	`__m256d _mm_macc_pd(__m256d, __m256d, __m256d);`
`_mm256_macc_ps`	FMA4	ammintrin.h	`__m256 _mm_macc_ps(__m256, __m256, __m256);`
`_mm256_madd_epi16`	AVX2	immintrin.h	`__m256i _mm256_madd_epi16(__m256i, __m256i);`
`_mm256_maddsub_pd`	FMA4	ammintrin.h	`__m256d _mm_maddsub_pd(__m256d, __m256d, __m256d);`
`_mm256_maddsub_ps`	FMA4	ammintrin.h	`__m256 _mm_maddsub_ps(__m256, __m256, __m256);`
`_mm256_maddubs_epi16`	AVX2	immintrin.h	`__m256i _mm256_maddubs_epi16(__m256i, __m256i);`
`_mm256_mask_i32gather_epi32`	AVX2	immintrin.h	`__m256i _mm256_mask_i32gather_epi32(__m256i, int const *, __m256i, __m256i, const int);`
`_mm256_mask_i32gather_epi64`	AVX2	immintrin.h	`__m256i _mm256_mask_i32gather_epi64(__m256i, __int64 const *, __m128i, __m256i, const int);`
`_mm256_mask_i32gather_pd`	AVX2	immintrin.h	`__m256d _mm256_mask_i32gather_pd(__m256d, double const *, __m128i, __m256d, const int);`
`_mm256_mask_i32gather_ps`	AVX2	immintrin.h	`__m256 _mm256_mask_i32gather_ps(__m256, float const *, __m256i, __m256, const int);`
`_mm256_mask_i64gather_epi32`	AVX2	immintrin.h	`__m128i _mm256_mask_i64gather_epi32(__m128i, int const *, __m256i, __m128i, const int);`
`_mm256_mask_i64gather_epi64`	AVX2	immintrin.h	`__m256i _mm256_mask_i64gather_epi64(__m256i, __int64 const *, __m256i, __m256i, const int);`
`_mm256_mask_i64gather_pd`	AVX2	immintrin.h	`__m256d _mm256_mask_i64gather_pd(__m256d, double const *, __m256i, __m256d, const int);`
`_mm256_mask_i64gather_ps`	AVX2	immintrin.h	`__m128 _mm256_mask_i64gather_ps(__m128, float const *, __m256i, __m128, const int);`
`_mm256_maskload_epi32`	AVX2	immintrin.h	`__m256i _mm256_maskload_epi32(int const *, __m256i);`
`_mm256_maskload_epi64`	AVX2	immintrin.h	`__m256i _mm256_maskload_epi64(__int64 const *, __m256i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_maskload_pd`	AVX	immintrin.h	`__m256d _mm256_maskload_pd(double const *, __m256i);`
`_mm256_maskload_ps`	AVX	immintrin.h	`__m256 _mm256_maskload_ps(float const *, __m256i);`
`_mm256_maskstore_epi32`	AVX2	immintrin.h	`void _mm256_maskstore_epi32(int *, __m256i, __m256i);`
`_mm256_maskstore_epi64`	AVX2	immintrin.h	`void _mm256_maskstore_epi64(__int64 *, __m256i, __m256i);`
`_mm256_maskstore_pd`	AVX	immintrin.h	`void _mm256_maskstore_pd(double *, __m256i, __m256d);`
`_mm256_maskstore_ps`	AVX	immintrin.h	`void _mm256_maskstore_ps(float *, __m256i, __m256);`
`_mm256_max_epi16`	AVX2	immintrin.h	`__m256i _mm256_max_epi16(__m256i, __m256i);`
`_mm256_max_epi32`	AVX2	immintrin.h	`__m256i _mm256_max_epi32(__m256i, __m256i);`
`_mm256_max_epi8`	AVX2	immintrin.h	`__m256i _mm256_max_epi8(__m256i, __m256i);`
`_mm256_max_epu16`	AVX2	immintrin.h	`__m256i _mm256_max_epu16(__m256i, __m256i);`
`_mm256_max_epu32`	AVX2	immintrin.h	`__m256i _mm256_max_epu32(__m256i, __m256i);`
`_mm256_max_epu8`	AVX2	immintrin.h	`__m256i _mm256_max_epu8(__m256i, __m256i);`
`_mm256_max_pd`	AVX	immintrin.h	`__m256d _mm256_max_pd(__m256d, __m256d);`
`_mm256_max_ps`	AVX	immintrin.h	`__m256 _mm256_max_ps(__m256, __m256);`
`_mm256_min_epi16`	AVX2	immintrin.h	`__m256i _mm256_min_epi16(__m256i, __m256i);`
`_mm256_min_epi32`	AVX2	immintrin.h	`__m256i _mm256_min_epi32(__m256i, __m256i);`
`_mm256_min_epi8`	AVX2	immintrin.h	`__m256i _mm256_min_epi8(__m256i, __m256i);`
`_mm256_min_epu16`	AVX2	immintrin.h	`__m256i _mm256_min_epu16(__m256i, __m256i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_min_epu32`	AVX2	immintrin.h	`__m256i _mm256_min_epu32(__m256i, __m256i);`
`_mm256_min_epu8`	AVX2	immintrin.h	`__m256i _mm256_min_epu8(__m256i, __m256i);`
`_mm256_min_pd`	AVX	immintrin.h	`__m256d _mm256_min_pd(__m256d, __m256d);`
`_mm256_min_ps`	AVX	immintrin.h	`__m256 _mm256_min_ps(__m256, __m256);`
`_mm256_movedup_pd`	AVX	immintrin.h	`__m256d _mm256_movedup_pd(__m256d);`
`_mm256_movehdup_ps`	AVX	immintrin.h	`__m256 _mm256_movehdup_ps(__m256);`
`_mm256_moveldup_ps`	AVX	immintrin.h	`__m256 _mm256_moveldup_ps(__m256);`
`_mm256_movemask_epi8`	AVX2	immintrin.h	`int _mm256_movemask_epi8(__m256i);`
`_mm256_movemask_pd`	AVX	immintrin.h	`int _mm256_movemask_pd(__m256d);`
`_mm256_movemask_ps`	AVX	immintrin.h	`int _mm256_movemask_ps(__m256);`
`_mm256_mpsadbw_epu8`	AVX2	immintrin.h	`__m256i _mm256_mpsadbw_epu8(__m256i, __m256i, const int);`
`_mm256_msub_pd`	FMA4	ammintrin.h	`__m256d _mm_msub_pd(__m256d, __m256d, __m256d);`
`_mm256_msub_ps`	FMA4	ammintrin.h	`__m256 _mm_msub_ps(__m256, __m256, __m256);`
`_mm256_msubadd_pd`	FMA4	ammintrin.h	`__m256d _mm_msubadd_pd(__m256d, __m256d, __m256d);`
`_mm256_msubadd_ps`	FMA4	ammintrin.h	`__m256 _mm_msubadd_ps(__m256, __m256, __m256);`
`_mm256_mul_epi32`	AVX2	immintrin.h	`__m256i _mm256_mul_epi32(__m256i, __m256i);`
`_mm256_mul_epu32`	AVX2	immintrin.h	`__m256i _mm256_mul_epu32(__m256i, __m256i);`
`_mm256_mul_pd`	AVX	immintrin.h	`__m256d _mm256_mul_pd(__m256d, __m256d);`
`_mm256_mul_ps`	AVX	immintrin.h	`__m256 _mm256_mul_ps(__m256, __m256);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_mulhi_epi16`	AVX2	immintrin.h	`__m256i _mm256_mulhi_epi16(__m256i, __m256i);`
`_mm256_mulhi_epu16`	AVX2	immintrin.h	`__m256i _mm256_mulhi_epu16(__m256i, __m256i);`
`_mm256_mulhrs_epi16`	AVX2	immintrin.h	`__m256i _mm256_mulhrs_epi16(__m256i, __m256i);`
`_mm256_mullo_epi16`	AVX2	immintrin.h	`__m256i _mm256_mullo_epi16(__m256i, __m256i);`
`_mm256_mullo_epi32`	AVX2	immintrin.h	`__m256i _mm256_mullo_epi32(__m256i, __m256i);`
`_mm256_nmacc_pd`	FMA4	ammintrin.h	`__m256d _mm_nmacc_pd(__m256d, __m256d, __m256d);`
`_mm256_nmacc_ps`	FMA4	ammintrin.h	`__m256 _mm_nmacc_ps(__m256, __m256, __m256);`
`_mm256_nmsub_pd`	FMA4	ammintrin.h	`__m256d _mm_nmsub_pd(__m256d, __m256d, __m256d);`
`_mm256_nmsub_ps`	FMA4	ammintrin.h	`__m256 _mm_nmsub_ps(__m256, __m256, __m256);`
`_mm256_or_pd`	AVX	immintrin.h	`__m256d _mm256_or_pd(__m256d, __m256d);`
`_mm256_or_ps`	AVX	immintrin.h	`__m256 _mm256_or_ps(__m256, __m256);`
`_mm256_or_si256`	AVX2	immintrin.h	`__m256i _mm256_or_si256(__m256i, __m256i);`
`_mm256_packs_epi16`	AVX2	immintrin.h	`__m256i _mm256_packs_epi16(__m256i, __m256i);`
`_mm256_packs_epi32`	AVX2	immintrin.h	`__m256i _mm256_packs_epi32(__m256i, __m256i);`
`_mm256_packus_epi16`	AVX2	immintrin.h	`__m256i _mm256_packus_epi16(__m256i, __m256i);`
`_mm256_packus_epi32`	AVX2	immintrin.h	`__m256i _mm256_packus_epi32(__m256i, __m256i);`
`_mm256_permute_pd`	AVX	immintrin.h	`__m256d _mm256_permute_pd(__m256d, int);`
`_mm256_permute_ps`	AVX	immintrin.h	`__m256 _mm256_permute_ps(__m256, int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_permute2_pd`	XOP	ammintrin.h	`__m256d _mm256_permute2_pd(__m256d, __m256d, __m256i, int);`
`_mm256_permute2_ps`	XOP	ammintrin.h	`__m256 _mm256_permute2_ps(__m256, __m256, __m256i, int);`
`_mm256_permute2f128_pd`	AVX	immintrin.h	`__m256d _mm256_permute2f128_pd(__m256d, __m256d, int);`
`_mm256_permute2f128_ps`	AVX	immintrin.h	`__m256 _mm256_permute2f128_ps(__m256, __m256, int);`
`_mm256_permute2f128_si256`	AVX	immintrin.h	`__m256i _mm256_permute2f128_si256(__m256i, __m256i, int);`
`_mm256_permute2x128_si256`	AVX2	immintrin.h	`__m256i _mm256_permute2x128_si256(__m256i, __m256i, const int);`
`_mm256_permute4x64_epi64`	AVX2	immintrin.h	`__m256i _mm256_permute4x64_epi64 (__m256i, const int);`
`_mm256_permute4x64_pd`	AVX2	immintrin.h	`__m256d _mm256_permute4x64_pd(__m256d, const int);`
`_mm256_permutevar_pd`	AVX	immintrin.h	`__m256d _mm256_permutevar_pd(__m256d, __m256i);`
`_mm256_permutevar_ps`	AVX	immintrin.h	`__m256 _mm256_permutevar_ps(__m256, __m256i);`
`_mm256_permutevar8x32_epi32`	AVX2	immintrin.h	`__m256i _mm256_permutevar8x32_epi32(__m256i, __m256i);`
`_mm256_permutevar8x32_ps`	AVX2	immintrin.h	`__m256 _mm256_permutevar8x32_ps (__m256, __m256i);`
`_mm256_rcp_ps`	AVX	immintrin.h	`__m256 _mm256_rcp_ps(__m256);`
`_mm256_round_pd`	AVX	immintrin.h	`__m256d _mm256_round_pd(__m256d, int);`
`_mm256_round_ps`	AVX	immintrin.h	`__m256 _mm256_round_ps(__m256, int);`
`_mm256_rsqrt_ps`	AVX	immintrin.h	`__m256 _mm256_rsqrt_ps(__m256);`
`_mm256_sad_epu8`	AVX2	immintrin.h	`__m256i _mm256_sad_epu8(__m256i, __m256i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_set_epi16`	AVX	immintrin.h	`(__m256i _mm256_set_epi16(short, short, short, short, short, short, short, short, short, short, short, short, short, short, short, short);`
`_mm256_set_epi32`	AVX	immintrin.h	`__m256i _mm256_set_epi32(int, int, int, int, int, int, int, int);`
`_mm256_set_epi64x`	AVX	immintrin.h	`__m256i _mm256_set_epi64x(long long, long long, long long, long long);`
`_mm256_set_epi8`	AVX	immintrin.h	`__m256i _mm256_set_epi8(char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char);`
`_mm256_set_pd`	AVX	immintrin.h	`__m256d _mm256_set_pd(double, double, double, double);`
`_mm256_set_ps`	AVX	immintrin.h	`__m256 _mm256_set_ps(float, float, float, float, float, float, float, float);`
`_mm256_set1_epi16`	AVX	immintrin.h	`__m256i _mm256_set1_epi16(short);`
`_mm256_set1_epi32`	AVX	immintrin.h	`__m256i _mm256_set1_epi32(int);`
`_mm256_set1_epi64x`	AVX	immintrin.h	`__m256i _mm256_set1_epi64x(long long);`
`_mm256_set1_epi8`	AVX	immintrin.h	`__m256i _mm256_set1_epi8(char);`
`_mm256_set1_pd`	AVX	immintrin.h	`__m256d _mm256_set1_pd(double);`
`_mm256_set1_ps`	AVX	immintrin.h	`__m256 _mm256_set1_ps(float);`
`_mm256_setr_epi16`	AVX	immintrin.h	`(__m256i _mm256_setr_epi16(short, short, short, short, short, short, short, short, short, short, short, short, short, short, short, short);`
`_mm256_setr_epi32`	AVX	immintrin.h	`__m256i _mm256_setr_epi32(int, int, int, int, int, int, int, int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_setr_epi64x`	AVX	immintrin.h	`__m256i _mm256_setr_epi64x(long long, long long, long long, long long);`
`_mm256_setr_epi8`	AVX	immintrin.h	`(__m256i _mm256_setr_epi8(char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char, char);`
`_mm256_setr_pd`	AVX	immintrin.h	`__m256d _mm256_setr_pd(double, double, double, double);`
`_mm256_setr_ps`	AVX	immintrin.h	`__m256 _mm256_setr_ps(float, float, float, float, float, float, float, float);`
`_mm256_setzero_pd`	AVX	immintrin.h	`__m256d _mm256_setzero_pd(void);`
`_mm256_setzero_ps`	AVX	immintrin.h	`__m256 _mm256_setzero_ps(void);`
`_mm256_setzero_si256`	AVX	immintrin.h	`__m256i _mm256_setzero_si256(void);`
`_mm256_shuffle_epi32`	AVX2	immintrin.h	`__m256i _mm256_shuffle_epi32(__m256i, const int);`
`_mm256_shuffle_epi8`	AVX2	immintrin.h	`__m256i _mm256_shuffle_epi8(__m256i, __m256i);`
`_mm256_shuffle_pd`	AVX	immintrin.h	`__m256d _mm256_shuffle_pd(__m256d, __m256d, const int);`
`_mm256_shuffle_ps`	AVX	immintrin.h	`__m256 _mm256_shuffle_ps(__m256, __m256, const int);`
`_mm256_shufflehi_epi16`	AVX2	immintrin.h	`__m256i _mm256_shufflehi_epi16(__m256i, const int);`
`_mm256_shufflelo_epi16`	AVX2	immintrin.h	`__m256i _mm256_shufflelo_epi16(__m256i, const int);`
`_mm256_sign_epi16`	AVX2	immintrin.h	`__m256i _mm256_sign_epi16(__m256i, __m256i);`
`_mm256_sign_epi32`	AVX2	immintrin.h	`__m256i _mm256_sign_epi32(__m256i, __m256i);`
`_mm256_sign_epi8`	AVX2	immintrin.h	`__m256i _mm256_sign_epi8(__m256i, __m256i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_sll_epi16`	AVX2	immintrin.h	`__m256i _mm256_sll_epi16(__m256i, __m128i);`
`_mm256_sll_epi32`	AVX2	immintrin.h	`__m256i _mm256_sll_epi32(__m256i, __m128i);`
`_mm256_sll_epi64`	AVX2	immintrin.h	`__m256i _mm256_sll_epi64(__m256i, __m128i);`
`_mm256_slli_epi16`	AVX2	immintrin.h	`__m256i _mm256_slli_epi16(__m256i, int);`
`_mm256_slli_epi32`	AVX2	immintrin.h	`__m256i _mm256_slli_epi32(__m256i, int);`
`_mm256_slli_epi64`	AVX2	immintrin.h	`__m256i _mm256_slli_epi64(__m256i, int);`
`_mm256_slli_si256`	AVX2	immintrin.h	`__m256i _mm256_slli_si256(__m256i, int);`
`_mm256_sllv_epi32`	AVX2	immintrin.h	`__m256i _mm256_sllv_epi32(__m256i, __m256i);`
`_mm256_sllv_epi64`	AVX2	immintrin.h	`__m256i _mm256_sllv_epi64(__m256i, __m256i);`
`_mm256_sqrt_pd`	AVX	immintrin.h	`__m256d _mm256_sqrt_pd(__m256d);`
`_mm256_sqrt_ps`	AVX	immintrin.h	`__m256 _mm256_sqrt_ps(__m256);`
`_mm256_sra_epi16`	AVX2	immintrin.h	`__m256i _mm256_sra_epi16(__m256i, __m128i);`
`_mm256_sra_epi32`	AVX2	immintrin.h	`__m256i _mm256_sra_epi32(__m256i, __m128i);`
`_mm256_srai_epi16`	AVX2	immintrin.h	`__m256i _mm256_srai_epi16(__m256i, int);`
`_mm256_srai_epi32`	AVX2	immintrin.h	`__m256i _mm256_srai_epi32(__m256i, int);`
`_mm256_srav_epi32`	AVX2	immintrin.h	`__m256i _mm256_srav_epi32(__m256i, __m256i);`
`_mm256_srl_epi16`	AVX2	immintrin.h	`__m256i _mm256_srl_epi16(__m256i, __m128i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_srl_epi32`	AVX2	immintrin.h	`__m256i _mm256_srl_epi32(__m256i, __m128i);`
`_mm256_srl_epi64`	AVX2	immintrin.h	`__m256i _mm256_srl_epi64(__m256i, __m128i);`
`_mm256_srli_epi16`	AVX2	immintrin.h	`__m256i _mm256_srli_epi16(__m256i, int);`
`_mm256_srli_epi32`	AVX2	immintrin.h	`__m256i _mm256_srli_epi32(__m256i, int);`
`_mm256_srli_epi64`	AVX2	immintrin.h	`__m256i _mm256_srli_epi64(__m256i, int);`
`_mm256_srli_si256`	AVX2	immintrin.h	`__m256i _mm256_srli_si256(__m256i, int);`
`_mm256_srlv_epi32`	AVX2	immintrin.h	`__m256i _mm256_srlv_epi32(__m256i, __m256i);`
`_mm256_srlv_epi64`	AVX2	immintrin.h	`__m256i _mm256_srlv_epi64(__m256i, __m256i);`
`_mm256_store_pd`	AVX	immintrin.h	`void _mm256_store_pd(double *, __m256d);`
`_mm256_store_ps`	AVX	immintrin.h	`void _mm256_store_ps(float *, __m256);`
`_mm256_store_si256`	AVX	immintrin.h	`void _mm256_store_si256(__m256i *, __m256i);`
`_mm256_storeu_pd`	AVX	immintrin.h	`void _mm256_storeu_pd(double *, __m256d);`
`_mm256_storeu_ps`	AVX	immintrin.h	`void _mm256_storeu_ps(float *, __m256);`
`_mm256_storeu_si256`	AVX	immintrin.h	`void _mm256_storeu_si256(__m256i *, __m256i);`
`_mm256_stream_load_si256`	AVX2	immintrin.h	`__m256i _mm256_stream_load_si256(__m256i const *);`
`_mm256_stream_pd`	AVX	immintrin.h	`void __mm256_stream_pd(double *, __m256d);`
`_mm256_stream_ps`	AVX	immintrin.h	`void _mm256_stream_ps(float *, __m256);`
`_mm256_stream_si256`	AVX	immintrin.h	`void __mm256_stream_si256(__m256i *, __m256i);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_sub_epi16`	AVX2	immintrin.h	`__m256i _mm256_sub_epi16(__m256i, __m256i);`
`_mm256_sub_epi32`	AVX2	immintrin.h	`__m256i _mm256_sub_epi32(__m256i, __m256i);`
`_mm256_sub_epi64`	AVX2	immintrin.h	`__m256i _mm256_sub_epi64(__m256i, __m256i);`
`_mm256_sub_epi8`	AVX2	immintrin.h	`__m256i _mm256_sub_epi8(__m256i, __m256i);`
`_mm256_sub_pd`	AVX	immintrin.h	`__m256d _mm256_sub_pd(__m256d, __m256d);`
`_mm256_sub_ps`	AVX	immintrin.h	`__m256 _mm256_sub_ps(__m256, __m256);`
`_mm256_subs_epi16`	AVX2	immintrin.h	`__m256i _mm256_subs_epi16(__m256i, __m256i);`
`_mm256_subs_epi8`	AVX2	immintrin.h	`__m256i _mm256_subs_epi8(__m256i, __m256i);`
`_mm256_subs_epu16`	AVX2	immintrin.h	`__m256i _mm256_subs_epu16(__m256i, __m256i);`
`_mm256_subs_epu8`	AVX2	immintrin.h	`__m256i _mm256_subs_epu8(__m256i, __m256i);`
`_mm256_testc_pd`	AVX	immintrin.h	`int _mm256_testc_pd(__m256d, __m256d);`
`_mm256_testc_ps`	AVX	immintrin.h	`int _mm256_testc_ps(__m256, __m256);`
`_mm256_testc_si256`	AVX	immintrin.h	`int _mm256_testc_si256(__m256i, __m256i);`
`_mm256_testnzc_pd`	AVX	immintrin.h	`int _mm256_testnzc_pd(__m256d, __m256d);`
`_mm256_testnzc_ps`	AVX	immintrin.h	`int _mm256_testnzc_ps(__m256, __m256);`
`_mm256_testnzc_si256`	AVX	immintrin.h	`int _mm256_testnzc_si256(__m256i, __m256i);`
`_mm256_testz_pd`	AVX	immintrin.h	`int _mm256_testz_pd(__m256d, __m256d);`
`_mm256_testz_ps`	AVX	immintrin.h	`int _mm256_testz_ps(__m256, __m256);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_mm256_testz_si256`	AVX	immintrin.h	`int _mm256_testz_si256(__m256i, __m256i);`
`_mm256_unpackhi_epi16`	AVX2	immintrin.h	`__m256i _mm256_unpackhi_epi16(__m256i, __m256i);`
`_mm256_unpackhi_epi32`	AVX2	immintrin.h	`__m256i _mm256_unpackhi_epi32(__m256i, __m256i);`
`_mm256_unpackhi_epi64`	AVX2	immintrin.h	`__m256i _mm256_unpackhi_epi64(__m256i, __m256i);`
`_mm256_unpackhi_epi8`	AVX2	immintrin.h	`__m256i _mm256_unpackhi_epi8(__m256i, __m256i);`
`_mm256_unpackhi_pd`	AVX	immintrin.h	`__m256d _mm256_unpackhi_pd(__m256d, __m256d);`
`_mm256_unpackhi_ps`	AVX	immintrin.h	`__m256 _mm256_unpackhi_ps(__m256, __m256);`
`_mm256_unpacklo_epi16`	AVX2	immintrin.h	`__m256i _mm256_unpacklo_epi16(__m256i, __m256i);`
`_mm256_unpacklo_epi32`	AVX2	immintrin.h	`__m256i _mm256_unpacklo_epi32(__m256i, __m256i);`
`_mm256_unpacklo_epi64`	AVX2	immintrin.h	`__m256i _mm256_unpacklo_epi64(__m256i, __m256i);`
`_mm256_unpacklo_epi8`	AVX2	immintrin.h	`__m256i _mm256_unpacklo_epi8(__m256i, __m256i);`
`_mm256_unpacklo_pd`	AVX	immintrin.h	`__m256d _mm256_unpacklo_pd(__m256d, __m256d);`
`_mm256_unpacklo_ps`	AVX	immintrin.h	`__m256 _mm256_unpacklo_ps(__m256, __m256);`
`_mm256_xor_pd`	AVX	immintrin.h	`__m256d _mm256_xor_pd(__m256d, __m256d);`
`_mm256_xor_ps`	AVX	immintrin.h	`__m256 _mm256_xor_ps(__m256, __m256);`
`_mm256_xor_si256`	AVX2	immintrin.h	`__m256i _mm256_xor_si256(__m256i, __m256i);`
`_mm256_zeroall`	AVX	immintrin.h	`void _mm256_zeroall(void);`
`_mm256_zeroupper`	AVX	immintrin.h	`void _mm256_zeroupper(void);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__movsb`		intrin.h	`VOID __movsb(unsigned char *, unsigned char const *, size_t);`
`__movsd`		intrin.h	`VOID __movsd(unsigned long *, unsigned long const *, size_t);`
`__movsq`		intrin.h	`VOID __movsq(unsigned __int64 *, unsigned __int64 const *, size_t);`
`__movsw`		intrin.h	`VOID __movsw(unsigned short *, unsigned short const *, size_t);`
`_mul128`		intrin.h	`__int64 _mul128(__int64, __int64, __int64 *);`
`__mulh`		intrin.h	`__int64 __mulh(__int64, __int64);`
`_mulx_u32`	BMI	immintrin.h	`unsigned int _mulx_u32(unsigned int, unsigned int, unsigned int*);`
`_mulx_u64`	BMI	immintrin.h	`unsigned __int64 _mulx_u64(unsigned __int64, unsigned __int64, unsigned __int64*);`
`__nop`		intrin.h	`void __nop(void);`
`__nvreg_restore_fence`		intrin.h	`void __nvreg_restore_fence(void);`
`__nvreg_save_fence`		intrin.h	`void __nvreg_save_fence(void);`
`__outbyte`		intrin.h	`void __outbyte(unsigned short, unsigned char);`
`__outbytestring`		intrin.h	`void __outbytestring(unsigned short, unsigned char *, unsigned long);`
`__outdword`		intrin.h	`void __outdword(unsigned short, unsigned long);`
`__outdwordstring`		intrin.h	`void __outdwordstring(unsigned short, unsigned long *, unsigned long);`
`__outword`		intrin.h	`void __outword(unsigned short, unsigned short);`
`__outwordstring`		intrin.h	`void __outwordstring(unsigned short, unsigned short *, unsigned long);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_pdep_u32`	BMI	immintrin.h	`unsigned int _pdep_u32(unsigned int, unsigned int);`
`_pdep_u64`	BMI	immintrin.h	`unsigned __int64 _pdep_u64(unsigned __int64, unsigned __int64);`
`_pext_u32`	BMI	immintrin.h	`unsigned int _pext_u32(unsigned int, unsigned int);`
`_pext_u64`	BMI	immintrin.h	`unsigned __int64 _pext_u64(unsigned __int64, unsigned __int64);`
`__popcnt`	POPCNT	intrin.h	`unsigned int __popcnt(unsigned int);`
`__popcnt16`	POPCNT	intrin.h	`unsigned short __popcnt16(unsigned short);`
`__popcnt64`	POPCNT	intrin.h	`unsigned __int64 __popcnt64(unsigned __int64);`
`_rdrand16_step`	RDRAND	immintrin.h	`int _rdrand16_step(unsigned short *);`
`_rdrand32_step`	RDRAND	immintrin.h	`int _rdrand32_step(unsigned int *);`
`_rdrand64_step`	RDRAND	immintrin.h	`int _rdrand64_step(unsigned __int64 *);`
`_rdseed16_step`	RDSEED	immintrin.h	`int _rdseed16_step(unsigned short *);`
`_rdseed32_step`	RDSEED	immintrin.h	`int _rdseed32_step(unsigned int *);`
`_rdseed64_step`	RDSEED	immintrin.h	`int _rdseed64_step(unsigned __int64 *);`
`__rdtsc`		intrin.h	`unsigned __int64 __rdtsc(void);`
`__rdtscp`	RDTSCP	intrin.h	`unsigned __int64 __rdtscp(unsigned int*);`
`_ReadBarrier`		intrin.h	`void _ReadBarrier(void);`
`__readcr0`		intrin.h	`unsigned __int64 __readcr0(void);`
`__readcr2`		intrin.h	`unsigned __int64 __readcr2(void);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__readcr3`		intrin.h	`unsigned __int64 __readcr3(void);`
`__readcr4`		intrin.h	`unsigned __int64 __readcr4(void);`
`__readcr8`		intrin.h	`unsigned __int64 __readcr8(void);`
`__readdr`		intrin.h	`unsigned __int64 __readdr(unsigned);`
`__readeflags`		intrin.h	`unsigned __int64 __readeflags(void);`
`_readfsbase_u32`	FSGSBASE	immintrin.h	`unsigned int _readfsbase_u32(void);`
`_readfsbase_u64`	FSGSBASE	immintrin.h	`unsigned __int64 _readfsbase_u64(void);`
`_readgsbase_u32`	FSGSBASE	immintrin.h	`unsigned int _readgsbase_u32(void);`
`_readgsbase_u64`	FSGSBASE	immintrin.h	`unsigned __int64 _readgsbase_u64(void);`
`__readgsbyte`		intrin.h	`unsigned char __readgsbyte(unsigned long);`
`__readgsdword`		intrin.h	`unsigned long __readgsdword(unsigned long);`
`__readgsqword`		intrin.h	`unsigned __int64 __readgsqword(unsigned long);`
`__readgsword`		intrin.h	`unsigned short __readgsword(unsigned long);`
`__readmsr`		intrin.h	`unsigned __int64 __readmsr(unsigned long);`
`__readpmc`		intrin.h	`unsigned __int64 __readpmc(unsigned long);`
`_ReadWriteBarrier`		intrin.h	`void _ReadWriteBarrier(void);`
`_ReturnAddress`		intrin.h	`void * _ReturnAddress(void);`
`_rorx_u32`	BMI	immintrin.h	`unsigned int _rorx_u32(unsigned int, const unsigned int);`
`_rorx_u64`	BMI	immintrin.h	`unsigned __int64 _rorx_u64(unsigned __int64, const unsigned int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_rotl16`		intrin.h	`unsigned short _rotl16(unsigned short, unsigned char);`
`_rotl8`		intrin.h	`unsigned char _rotl8(unsigned char, unsigned char);`
`_rotr16`		intrin.h	`unsigned short _rotr16(unsigned short, unsigned char);`
`_rotr8`		intrin.h	`unsigned char _rotr8(unsigned char, unsigned char);`
`_rsm`		intrin.h	`void _rsm(void);`
`_sarx_i32`	BMI	immintrin.h	`int _sarx_i32(int, unsigned int);`
`_sarx_i64`	BMI	immintrin.h	`__int64 _sarx_i64(__int64, unsigned int);`
`__segmentlimit`		intrin.h	`unsigned long __segmentlimit(unsigned long);`
`_sgdt`		intrin.h	`void _sgdt(void*);`
`__shiftleft128`		intrin.h	`unsigned __int64 __shiftleft128(unsigned __int64, unsigned __int64, unsigned char);`
`__shiftright128`		intrin.h	`unsigned __int64 __shiftright128(unsigned __int64, unsigned __int64, unsigned char);`
`_shlx_u32`	BMI	immintrin.h	`unsigned int _shlx_u32(unsigned int, unsigned int);`
`_shlx_u64`	BMI	immintrin.h	`unsigned __int64 _shlx_u64(unsigned __int64, unsigned int);`
`_shrx_u32`	BMI	immintrin.h	`unsigned int _shrx_u32(unsigned int, unsigned int);`
`_shrx_u64`	BMI	immintrin.h	`unsigned __int64 _shrx_u64(unsigned __int64, unsigned int);`
`__sidt`		intrin.h	`void __sidt(void*);`
`__slwpcb`	LWP	ammintrin.h	`void *__slwpcb(void);`
`_stac`	SMAP	intrin.h	`void _stac(void);`
`_storebe_i16`	MOVBE	immintrin.h	`void _storebe_i16(void *, short);` [Macro]

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_storebe_i32`	MOVBE	immintrin.h	`void _storebe_i32(void *, int);` [Macro]
`_storebe_i64`	MOVBE	immintrin.h	`void _storebe_i64(void *, __int64);` [Macro]
`_store_be_u16`	MOVBE	immintrin.h	`void _store_be_u16(void *, unsigned short);` [Macro]
`_store_be_u32`	MOVBE	immintrin.h	`void _store_be_u32(void *, unsigned int);` [Macro]
`_store_be_u64`	MOVBE	immintrin.h	`void _store_be_u64(void *, unsigned __int64);` [Macro]
`_Store_HLERelease`	HLE	immintrin.h	`void _Store_HLERelease(long volatile *, long);`
`_Store64_HLERelease`	HLE	immintrin.h	`void _Store64_HLERelease(__int64 volatile *, __int64);`
`_StorePointer_HLERelease`	HLE	immintrin.h	`void _StorePointer_HLERelease(void * volatile *, void *);`
`__stosb`		intrin.h	`void __stosb(unsigned char *, unsigned char, size_t);`
`__stosd`		intrin.h	`void __stosd(unsigned long *, unsigned long, size_t);`
`__stosq`		intrin.h	`void __stosq(unsigned __int64 *, unsigned __int64, size_t);`
`__stosw`		intrin.h	`void __stosw(unsigned short *, unsigned short, size_t);`
`_subborrow_u16`		intrin.h	`unsigned char _subborrow_u16(unsigned char, unsigned short, unsigned short, unsigned short *);`
`_subborrow_u32`		intrin.h	`unsigned char _subborrow_u32(unsigned char, unsigned int, unsigned int, unsigned int *);`
`_subborrow_u64`		intrin.h	`unsigned char _subborrow_u64(unsigned char, unsigned __int64, unsigned __int64, unsigned __int64 *);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_subborrow_u8`		intrin.h	`unsigned char _subborrow_u8(unsigned char, unsigned char, unsigned char, unsigned char *);`
`__svm_clgi`		intrin.h	`void __svm_clgi(void);`
`__svm_invlpga`		intrin.h	`void __svm_invlpga(void*, int);`
`__svm_skinit`		intrin.h	`void __svm_skinit(int);`
`__svm_stgi`		intrin.h	`void __svm_stgi(void);`
`__svm_vmload`		intrin.h	`void __svm_vmload(size_t);`
`__svm_vmrun`		intrin.h	`void __svm_vmrun(size_t);`
`__svm_vmsave`		intrin.h	`void __svm_vmsave(size_t);`
`_t1mskc_u32`	ABM	ammintrin.h	`unsigned int _t1mskc_u32(unsigned int);`
`_t1mskc_u64`	ABM	ammintrin.h	`unsigned __int64 _t1mskc_u64(unsigned __int64);`
`_tzcnt_u32`	BMI	ammintrin.h, immintrin.h	`unsigned int _tzcnt_u32(unsigned int);`
`_tzcnt_u64`	BMI	ammintrin.h, immintrin.h	`unsigned __int64 _tzcnt_u64(unsigned __int64);`
`_tzmsk_u32`	ABM	ammintrin.h	`unsigned int _tzmsk_u32(unsigned int);`
`_tzmsk_u64`	ABM	ammintrin.h	`unsigned __int64 _tzmsk_u64(unsigned __int64);`
`__ud2`		intrin.h	`void __ud2(void);`
`_udiv128`		intrin.h	`unsigned __int64 _udiv128(unsigned __int64, unsigned __int64, unsigned __int64, unsigned __int64 *);`
`_udiv64`		intrin.h	`unsigned int _udiv64(unsigned __int64, unsigned int, unsigned int*);`
`__ull_rshift`		intrin.h	`unsigned __int64 [pascal/cdecl] __ull_rshift(unsigned __int64, int);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_umul128`		intrin.h	`unsigned __int64 _umul128(unsigned __int64, unsigned __int64, unsigned __int64 *);`
`__umulh`		intrin.h	`unsigned __int64 __umulh(unsigned __int64, unsigned __int64);`
`__vmx_off`		intrin.h	`void __vmx_off(void);`
`__vmx_on`		intrin.h	`unsigned char __vmx_on(unsigned __int64*);`
`__vmx_vmclear`		intrin.h	`unsigned char __vmx_vmclear(unsigned __int64*);`
`__vmx_vmlaunch`		intrin.h	`unsigned char __vmx_vmlaunch(void);`
`__vmx_vmptrld`		intrin.h	`unsigned char __vmx_vmptrld(unsigned __int64*);`
`__vmx_vmptrst`		intrin.h	`void __vmx_vmptrst(unsigned __int64 *);`
`__vmx_vmread`		intrin.h	`unsigned char __vmx_vmread(size_t, size_t*);`
`__vmx_vmresume`		intrin.h	`unsigned char __vmx_vmresume(void);`
`__vmx_vmwrite`		intrin.h	`unsigned char __vmx_vmwrite(size_t, size_t);`
`__wbinvd`		intrin.h	`void __wbinvd(void);`
`_WriteBarrier`		intrin.h	`void _WriteBarrier(void);`
`__writecr0`		intrin.h	`void __writecr0(unsigned __int64);`
`__writecr3`		intrin.h	`void __writecr3(unsigned __int64);`
`__writecr4`		intrin.h	`void __writecr4(unsigned __int64);`
`__writecr8`		intrin.h	`void __writecr8(unsigned __int64);`
`__writedr`		intrin.h	`void __writedr(unsigned, unsigned __int64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`__writeeflags`		intrin.h	`void __writeeflags(unsigned __int64);`
`_writefsbase_u32`	FSGSBASE	immintrin.h	`void _writefsbase_u32(unsigned int);`
`_writefsbase_u64`	FSGSBASE	immintrin.h	`void _writefsbase_u64(unsigned __int64);`
`_writegsbase_u32`	FSGSBASE	immintrin.h	`void _writegsbase_u32(unsigned int);`
`_writegsbase_u64`	FSGSBASE	immintrin.h	`void _writegsbase_u64(unsigned __int64);`
`__writegsbyte`		intrin.h	`void __writegsbyte(unsigned long, unsigned char);`
`__writegsdword`		intrin.h	`void __writegsdword(unsigned long, unsigned long);`
`__writegsqword`		intrin.h	`void __writegsqword(unsigned long, unsigned __int64);`
`__writegsword`		intrin.h	`void __writegsword(unsigned long, unsigned short);`
`__writemsr`		intrin.h	`void __writemsr(unsigned long, unsigned __int64);`
`_xabort`	RTM	immintrin.h	`void _xabort(unsigned int);`
`_xbegin`	RTM	immintrin.h	`unsigned _xbegin(void);`
`_xend`	RTM	immintrin.h	`void _xend(void);`
`_xgetbv`	XSAVE	immintrin.h	`unsigned __int64 _xgetbv(unsigned int);`
`_xrstor`	XSAVE	immintrin.h	`void _xrstor(void const*, unsigned __int64);`
`_xrstor64`	XSAVE	immintrin.h	`void _xrstor64(void const*, unsigned __int64);`
`_xsave`	XSAVE	immintrin.h	`void _xsave(void*, unsigned __int64);`
`_xsave64`	XSAVE	immintrin.h	`void _xsave64(void*, unsigned __int64);`
`_xsaveopt`	XSAVEOPT	immintrin.h	`void _xsaveopt(void*, unsigned __int64);`

INTRINSIC NAME	TECHNOLOGY	HEADER	FUNCTION PROTOTYPE
`_xsaveopt64`	XSAVEOPT	immintrin.h	`void _xsaveopt64(void*, unsigned __int64);`
`_xsetbv`	XSAVE	immintrin.h	`void _xsetbv(unsigned int, unsigned __int64);`
`_xtest`	XTEST	immintrin.h	`unsigned char _xtest(void);`

See also

Compiler intrinsics
ARM intrinsics
ARM64 intrinsics
x86 intrinsics

Intrinsics available on all architectures

9/2/2022 • 2 minutes to read • Edit Online

The Microsoft C/C++ compiler and the Universal C Runtime Library (UCRT) make some intrinsics available on all architectures.

Compiler intrinsics

The following intrinsics are available with the x86, AMD64, ARM, and ARM64 architectures:

INTRINSIC	HEADER
`_AddressOfReturnAddress`	intrin.h
`_BitScanForward`	intrin.h
`_BitScanReverse`	intrin.h
`_bittest`	intrin.h
`_bittestandcomplement`	intrin.h
`_bittestandreset`	intrin.h
`_bittestandset`	intrin.h
`__code_seg`	intrin.h
`__debugbreak`	intrin.h
`_disable`	intrin.h
`_enable`	intrin.h
`__fastfail`	intrin.h
`_InterlockedAnd`	intrin.h
`_InterlockedAnd16`	intrin.h
`_InterlockedAnd8`	intrin.h
`_interlockedbittestandreset`	intrin.h
`_interlockedbittestandset`	intrin.h
`_InterlockedCompareExchange`	intrin.h

INTRINSIC	HEADER
`_InterlockedCompareExchange16`	intrin.h
`_InterlockedCompareExchange8`	intrin.h
`_InterlockedCompareExchangePointer`	intrin.h
`_InterlockedDecrement`	intrin.h
`_InterlockedDecrement16`	intrin.h
`_InterlockedExchange`	intrin.h
`_InterlockedExchange16`	intrin.h
`_InterlockedExchange8`	intrin.h
`_InterlockedExchangeAdd`	intrin.h
`_InterlockedExchangeAdd16`	intrin.h
`_InterlockedExchangeAdd8`	intrin.h
`_InterlockedExchangePointer`	intrin.h
`_InterlockedIncrement`	intrin.h
`_InterlockedIncrement16`	intrin.h
`_InterlockedOr`	intrin.h
`_InterlockedOr16`	intrin.h
`_InterlockedOr8`	intrin.h
`_InterlockedXor`	intrin.h
`_InterlockedXor16`	intrin.h
`_InterlockedXor8`	intrin.h
`__nop`	intrin.h
`_ReadBarrier`	intrin.h
`_ReadWriteBarrier`	intrin.h
`_ReturnAddress`	intrin.h

INTRINSIC	HEADER
`_rotl16`	intrin.h
`_rotl8`	intrin.h
`_rotr16`	intrin.h
`_rotr8`	intrin.h
`_WriteBarrier`	intrin.h

UCRT intrinsics

The following UCRT functions have intrinsic forms on all architectures:

INTRINSIC	HEADER
`abs`	stdlib.h
`_abs64`	stdlib.h
`acos`	math.h
`acosf`	math.h
`acosl`	math.h
`_alloca`	malloc.h
`asin`	math.h
`asinf`	math.h
`asinl`	math.h
`atan`	math.h
`atan2`	math.h
`atan2f`	math.h
`atan2l`	math.h
`atanf`	math.h
`atanl`	math.h
`_byteswap_uint64`	stdlib.h

INTRINSIC	HEADER
`_byteswap_ulong`	stdlib.h
`_byteswap_ushort`	stdlib.h
`ceil`	math.h
`ceilf`	math.h
`ceill`	math.h
`cos`	math.h
`cosf`	math.h
`cosh`	math.h
`coshf`	math.h
`coshl`	math.h
`cosl`	math.h
`exp`	math.h
`expf`	math.h
`expl`	math.h
`fabs`	math.h
`fabsf`	math.h
`floor`	math.h
`floorf`	math.h
`floorl`	math.h
`fmod`	math.h
`fmodf`	math.h
`fmodl`	math.h
`labs`	stdlib.h
`llabs`	stdlib.h

INTRINSIC	HEADER
`log`	math.h
`log10`	math.h
`log10f`	math.h
`log10l`	math.h
`logf`	math.h
`logl`	math.h
`_lrotl`	stdlib.h
`_lrotr`	stdlib.h
`memcmp`	string.h
`memcpy`	string.h
`memset`	string.h
`pow`	math.h
`powf`	math.h
`powl`	math.h
`_rotl`	stdlib.h
`_rotl64`	stdlib.h
`_rotr`	stdlib.h
`_rotr64`	stdlib.h
`sin`	math.h
`sinf`	math.h
`sinh`	math.h
`sinhf`	math.h
`sinhl`	math.h
`sinl`	math.h

INTRINSIC	HEADER
`sqrt`	math.h
`sqrtf`	math.h
`sqrtl`	math.h
`strcat`	string.h
`strcmp`	string.h
`strcpy`	string.h
`strlen`	string.h
`_strset`	string.h
`strset`	string.h
`tan`	math.h
`tanf`	math.h
`tanh`	math.h
`tanhf`	math.h
`tanhl`	math.h
`tanl`	math.h
`wcscat`	string.h
`wcscmp`	string.h
`wcscpy`	string.h
`wcslen`	string.h
`_wcsset`	string.h

In Visual Studio 2022 version 17.2 and later, these functions have intrinsic forms on x64 and ARM64 platforms:

INTRINSIC	HEADER
`log2`	math.h
`log2f`	math.h

See also

ARM intrinsics

ARM64 intrinsics

x86 intrinsics list

x64 (amd64) intrinsics list

Alphabetical listing of intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

The following sections describe the Microsoft-specific intrinsic functions available on some or all architectures. Other supported intrinsics are documented by processor manufacturers, either in the header files or on their websites. For more information, and links to manufacturer documentation, see these articles: ARM intrinsics, ARM64 intrinsics, x86 intrinsics, and x64 intrinsics. C Runtime Library (CRT) functions implemented as intrinsics aren't documented here. CRT intrinsic functions are documented in the C Runtime Library Reference.

`__addfsbyte`, `__addfsword`, `__addfsdword`

`__addgsbyte`, `__addgsword`, `__addgsdword`, `__addgsqword`

`_AddressOfReturnAddress`

`__assume`

`_BitScanForward`, `_BitScanForward64`

`_BitScanReverse`, `_BitScanReverse64`

`_bittest`, `_bittest64`

`_bittestandcomplement`, `_bittestandcomplement64`

`_bittestandreset`, `_bittestandreset64`

`_bittestandset`, `_bittestandset64`

`__cpuid`, `__cpuidex`

`_cvt_ftoi_fast`, `_cvt_ftoll_fast`, `_cvt_ftoui_fast`, `_cvt_ftoull_fast`, `_cvt_dtoi_fast`, `_cvt_dtoll_fast`, `_cvt_dtoui_fast`, `_cvt_dtoull_fast`

`_cvt_ftoi_sat`, `_cvt_ftoll_sat`, `_cvt_ftoui_sat`, `_cvt_ftoull_sat`, `_cvt_dtoi_sat`, `_cvt_dtoll_sat`, `_cvt_dtoui_sat`, `_cvt_dtoull_sat`

`_cvt_ftoi_sent`, `_cvt_ftoll_sent`, `_cvt_ftoui_sent`, `_cvt_ftoull_sent`, `_cvt_dtoi_sent`, `_cvt_dtoll_sent`, `_cvt_dtoui_sent`, `_cvt_dtoull_sent`

`__debugbreak`

`_disable`

`__emul`, `__emulu`

`_enable`

`__fastfail`

`__faststorefence`

`__getcallerseflags`

`__halt`

`__inbyte`

`__inbytestring`

`__incfsbyte`, `__incfsword`, `__incfsdword`

`__incgsbyte`, `__incgsword`, `__incgsdword`, `__incgsqword`

`__indword`

`__indwordstring`

`__int2c`

`_InterlockedAdd` intrinsic functions

`_InterlockedAddLargeStatistic`

`_InterlockedAnd` intrinsic functions

`_interlockedbittestandreset` intrinsic functions

`_interlockedbittestandset` intrinsic functions

`_InterlockedCompareExchange` intrinsic functions

`_InterlockedCompareExchange128`

`_InterlockedCompareExchangePointer` intrinsic functions

`_InterlockedDecrement` intrinsic functions

`_InterlockedExchange` intrinsic functions

`_InterlockedExchangeAdd` intrinsic functions

`_InterlockedExchangePointer` intrinsic functions

`_InterlockedIncrement` intrinsic functions

`_InterlockedOr` intrinsic functions

`_InterlockedXor` intrinsic functions

`__invlpg`

`__inword`

`__inwordstring`

`__lidt`

`__ll_lshift`

`__ll_rshift`

`__lzcnt16`, `__lzcnt`, `__lzcnt64`

`_mm_cvtsi64x_ss`

`_mm_cvtss_si64x`

`_mm_cvttss_si64x`

`_mm_extract_si64`, `_mm_extracti_si64`

`_mm_insert_si64`, `_mm_inserti_si64`

`_mm_stream_sd`

`_mm_stream_si64x`

`_mm_stream_ss`

`__movsb`

`__movsd`

`__movsq`

`__movsw`

`__mul128`

`__mulh`

`__noop`

`__nop`

`__outbyte`

`__outbytestring`

`__outdword`

`__outdwordstring`

`__outword`

`__outwordstring`

`__popcnt16`, `__popcnt`, `__popcnt64`

`__rdtsc`

`__rdtscp`

`_ReadBarrier`

`__readcr0`

`__readcr2`

`__readcr3`

`__readcr4`

`__readcr8`

`__readdr`

`__readeflags`

`__readfsbyte`, `__readfsdword`, `__readfsqword`, `__readfsword`

`__readgsbyte`, `__readgsdword`, `__readgsqword`, `__readgsword`

`__readmsr`

`__readpmc`

`_ReadWriteBarrier`

`_ReturnAddress`

`_rotl8`, `_rotl16`

`_rotr8`, `_rotr16`

`__segmentlimit`

`__shiftleft128`

`__shiftright128`

`__sidt`

`__stosb`

`__stosd`

`__stosq`

`__stosw`

`__svm_clgi`

`__svm_invlpga`

`__svm_skinit`

`__svm_stgi`

`__svm_vmload`

`__svm_vmrun`

`__svm_vmsave`

`__ud2`

`__ull_rshift`

`_umul128`

`__umulh`

`__vmx_off`

`__vmx_on`

`__vmx_vmclear`

`__vmx_vmlaunch`

`__vmx_vmptrld`

`__vmx_vmptrst`

`__vmx_vmread`

`__vmx_vmresume`

`__vmx_vmwrite`

`__wbinvd`

`_WriteBarrier`

`__writecr0`

`__writecr3`

`__writecr4`

`__writecr8`

`__writedr`

`__writeeflags`

`__writefsbyte`, `__writefsdword`, `__writefsqword`, `__writefsword`

`__writegsbyte`, `__writegsdword`, `__writegsqword`, `__writegsword`

`__writemsr`

See also

Compiler intrinsics

__addfsbyte, __addfsword, __addfsdword

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Add a value to a memory location specified by an offset relative to the beginning of the `FS` segment.

Syntax

```
void __addfsbyte(
   unsigned long Offset,
   unsigned char Data
);
void __addfsword(
   unsigned long Offset,
   unsigned short Data
);
void __addfsdword(
   unsigned long Offset,
   unsigned long Data
);
```

Parameters

Offset
[in] The offset from the beginning of `FS`.

Data
[in] The value to add to the memory location.

Requirements

INTRINSIC	ARCHITECTURE
`__addfsbyte`	x86
`__addfsword`	x86
`__addfsdword`	x86

Header file <intrin.h>

Remarks

These routines are available only as intrinsics.

END Microsoft Specific

See also

__incfsbyte, __incfsword, __incfsdword
__readfsbyte, __readfsdword, __readfsqword, __readfsword
__writefsbyte, __writefsdword, __writefsqword, __writefsword

__addgsbyte, __addgsword, __addgsdword, __addgsqword

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Add a value to a memory location specified by an offset relative to the beginning of the `GS` segment.

Syntax

```
void __addgsbyte(
   unsigned long Offset,
   unsigned char Data
);
void __addgsword(
   unsigned long Offset,
   unsigned short Data
);
void __addgsdword(
   unsigned long Offset,
   unsigned long Data
);
void __addgsqword(
   unsigned long Offset,
   unsigned __int64 Data
);
```

Parameters

Offset
[in] The offset from the beginning of `GS`.

Data
[in] The value to add to the memory location.

Requirements

INTRINSIC	ARCHITECTURE
`__addgsbyte`	x64
`__addgsword`	x64
`__addgsdword`	x64
`__addgsqword`	x64

Header file <intrin.h>

Remarks

These routines are only available as an intrinsic.

END Microsoft Specific

See also

__incgsbyte, __incgsword, __incgsdword, __incgsqword
__readgsbyte, __readgsdword, __readgsqword, __readgsword
__writegsbyte, __writegsdword, __writegsqword, __writegsword
Compiler intrinsics

_AddressOfReturnAddress

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Provides the address of the memory location that holds the return address of the current function. This address may not be used to access other memory locations (for example, the function's arguments).

Syntax

```
void * _AddressOfReturnAddress();
```

Requirements

INTRINSIC	ARCHITECTURE
`_AddressOfReturnAddress`	x86, x64, ARM, ARM64

Header file <intrin.h>

Remarks

When `_AddressOfReturnAddress` is used in a program compiled with /clr, the function containing the `_AddressOfReturnAddress` call is compiled as a native function. When a function compiled as managed calls into the function containing `_AddressOfReturnAddress`, `_AddressOfReturnAddress` might not behave as expected.

This routine is only available as an intrinsic.

Example

```
// compiler_intrinsics_AddressOfReturnAddress.cpp
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

// This function will print three values:
//   (1) The address retrieved from _AddressOfReturnAdress
//   (2) The return address stored at the location returned in (1)
//   (3) The return address retrieved the _ReturnAddress* intrinsic
// Note that (2) and (3) should be the same address.
__declspec(noinline)
void func() {
   void* pvAddressOfReturnAddress = _AddressOfReturnAddress();
   printf_s("%p\n", pvAddressOfReturnAddress);
   printf_s("%p\n", *((void**) pvAddressOfReturnAddress));
   printf_s("%p\n", _ReturnAddress());
}

int main() {
   func();
}
```

```
0012FF78
00401058
00401058
```

END Microsoft Specific

See also

Compiler intrinsics
Keywords

__assume

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Passes a hint to the optimizer.

Syntax

```
__assume(
   expression
)
```

Parameters

`expression`
For reachable code, any expression that is assumed to evaluate to `true`. Use `0` to indicate unreachable code to the optimizer.

Remarks

The optimizer assumes that the condition represented by `expression` is `true` at the point where the keyword appears and remains true until `expression` is modified (for example, by assignment to a variable). Selective use of hints passed to the optimizer by `__assume` can improve optimization.

If the `__assume` statement is written as a contradiction (an expression that always evaluates to `false`), it's always treated as `__assume(0)`. If your code isn't behaving as expected, ensure that the `expression` you defined is valid and `true`, as described earlier. The `__assume(0)` statement is a special case. Use `__assume(0)` to indicate a code path that can't be reached.

WARNING
A program must not contain an invalid `__assume` statement on a reachable path. If the compiler can reach an invalid `__assume` statement, the program might cause unpredictable and potentially dangerous behavior.

For compatibility with previous versions, `_assume` is a synonym for `__assume` unless compiler option `/Za` (Disable language extensions) is specified.

`__assume` isn't a genuine intrinsic. It doesn't have to be declared as a function and it can't be used in a `#pragma intrinsic` directive. Although no code is generated, the code generated by the optimizer is affected.

Use `__assume` in an `ASSERT` only when the assertion isn't recoverable. Don't use `__assume` in an assertion for which you have subsequent error recovery code because the compiler might optimize away the error-handling code.

Requirements

INTRINSIC	ARCHITECTURE
`__assume`	x86, ARM, x64, ARM64, ARM64EC

Example

The following example shows how to use `__assume(0)` to indicate that the `default` case of a `switch` statement can't be reached. It's the most typical use of `__assume(0)`. Here, the programmer knows that the only possible inputs for `p` will be 1 or 2. If another value is passed in for `p`, the program becomes invalid and causes unpredictable behavior.

```
// compiler_intrinsics__assume.cpp

void func1(int /*ignored*/)
{
}

int main(int p)
{
   switch(p)
   {
   case 1:
      func1(1);
      break;
   case 2:
      func1(-1);
      break;
   default:
      __assume(0);
      // This tells the optimizer that the default
      // cannot be reached. As so, it does not have to generate
      // the extra code to check that 'p' has a value
      // not represented by a case arm. This makes the switch
      // run faster.
   }
}
```

As a result of the `__assume(0)` statement, the compiler doesn't generate code to test whether `p` has a value that isn't represented in a case statement.

If you aren't sure that the expression will always be `true` at runtime, you can use the `assert` function to protect the code. This macro definition wraps the `__assume` statement with a check:

```
#define ASSUME(e) (((e) || (assert(e), (e))), __assume(e))
```

For the `default` case optimization to work, the `__assume(0)` statement must be the first statement in the body of the `default` case. Unfortunately, the `assert` in the `ASSUME` macro prevents the compiler from performing this optimization. As an alternative, you can use a separate macro, as shown here:

```
#ifdef DEBUG
// This code is supposed to be unreachable, so assert
# define NODEFAULT   assert(0)
#else
# define NODEFAULT   __assume(0)
#endif
// . . .
   default:
      NODEFAULT;
```

END Microsoft Specific

See also

Compiler intrinsics
Keywords

_BitScanForward, _BitScanForward64

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Search the mask data from least significant bit (LSB) to the most significant bit (MSB) for a set bit (1).

Syntax

```
unsigned char _BitScanForward(
   unsigned long * Index,
   unsigned long Mask
);
unsigned char _BitScanForward64(
   unsigned long * Index,
   unsigned __int64 Mask
);
```

Parameters

Index
[out] Loaded with the bit position of the first set bit (1) found.

Mask
[in] The 32-bit or 64-bit value to search.

Return value

0 if the mask is zero; nonzero otherwise.

Remarks

If a set bit is found, the bit position of the first set bit found is returned in the first parameter. If no set bit is found, 0 is returned; otherwise, 1 is returned.

Requirements

INTRINSIC	ARCHITECTURE
`_BitScanForward`	x86, ARM, x64, ARM64
`_BitScanForward64`	ARM64, x64

Header file <intrin.h>

Example

```
// BitScanForward.cpp
// compile with: /EHsc
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(_BitScanForward)

int main()
{
   unsigned long mask = 0x1000;
   unsigned long index;
   unsigned char isNonzero;

   cout << "Enter a positive integer as the mask: " << flush;
   cin >> mask;
   isNonzero = _BitScanForward(&index, mask);
   if (isNonzero)
   {
      cout << "Mask: " << mask << " Index: " << index << endl;
   }
   else
   {
      cout << "No set bits found.  Mask is zero." << endl;
   }
}
```

```
12
```

```
Enter a positive integer as the mask:
Mask: 12 Index: 2
```

END Microsoft Specific

See also

Compiler intrinsics

_BitScanReverse, _BitScanReverse64

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Search the mask data from most significant bit (MSB) to least significant bit (LSB) for a set bit (1).

Syntax

```
unsigned char _BitScanReverse(
   unsigned long * Index,
   unsigned long Mask
);
unsigned char _BitScanReverse64(
   unsigned long * Index,
   unsigned __int64 Mask
);
```

Parameters

Index
[out] Loaded with the bit position of the first set bit (1) found. Otherwise, undefined.

Mask
[in] The 32-bit or 64-bit value to search.

Return value

Nonzero if any bit was set in `Mask`, or 0 if no set bits were found.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_BitScanReverse`	x86, ARM, x64, ARM64	<intrin.h>
`_BitScanReverse64`	ARM64, x64	<intrin.h>

Example

```
// BitScanReverse.cpp
// compile with: /EHsc
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(_BitScanReverse)

int main()
{
   unsigned long mask = 0x1000;
   unsigned long index;
   unsigned char isNonzero;

   cout << "Enter a positive integer as the mask: " << flush;
   cin >> mask;
   isNonzero = _BitScanReverse(&index, mask);
   if (isNonzero)
   {
      cout << "Mask: " << mask << " Index: " << index << endl;
   }
   else
   {
      cout << "No set bits found.  Mask is zero." << endl;
   }
}
```

```
12
```

```
Enter a positive integer as the mask:
Mask: 12 Index: 3
```

END Microsoft Specific

See also

Compiler intrinsics

_bittest, _bittest64

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `bt` instruction, which examines the bit in position `b` of address `a`, and returns the value of that bit.

Syntax

```
unsigned char _bittest(
   long const *a,
   long b
);
unsigned char _bittest64(
   __int64 const *a,
   __int64 b
);
```

Parameters

a
[in] A pointer to the memory to examine.

b
[in] The bit position to test.

Return value

The bit at the position specified.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_bittest`	x86, ARM, x64, ARM64	<intrin.h>
`_bittest64`	ARM64, x64	<intrin.h>

Remarks

This routine is only available as an intrinsic.

Example

```
// bittest.cpp
// processor: x86, ARM, x64

#include <stdio.h>
#include <intrin.h>

long num = 78002;

int main()
{
    unsigned char bits[32];
    long nBit;

    printf_s("Number: %d\n", num);

    for (nBit = 0; nBit < 31; nBit++)
    {
        bits[nBit] = _bittest(&num, nBit);
    }

    printf_s("Binary representation:\n");
    while (nBit--)
    {
        if (bits[nBit])
            printf_s("1");
        else
            printf_s("0");
    }
}
```

```
Number: 78002
Binary representation:
0000000000000010011000010110010
```

END Microsoft Specific

See also

Compiler intrinsics

_bittestandcomplement, _bittestandcomplement64

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generate an instruction which examines bit `b` of the address `a`, returns its current value, and sets the bit to its complement.

Syntax

```
unsigned char _bittestandcomplement(
   long *a,
   long b
);
unsigned char _bittestandcomplement64(
   __int64 *a,
   __int64 b
);
```

Parameters

a
[in, out] A pointer to the memory to examine.

b
[in] The bit position to test.

Return value

The bit at the position specified.

Requirements

INTRINSIC	ARCHITECTURE
`_bittestandcomplement`	x86, ARM, x64, ARM64
`_bittestandcomplement64`	x64, ARM64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

Example

```
// bittestandcomplement.cpp
// processor: x86, IPF, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_bittestandcomplement)
#ifdef _M_AMD64
#pragma intrinsic(_bittestandcomplement64)
#endif

int main()
{
   long i = 1;
   __int64 i64 = 0x1I64;
   unsigned char result;
   printf("Initial value: %d\n", i);
   printf("Testing bit 1\n");
   result = _bittestandcomplement(&i, 1);
   printf("Value changed to %d, Result: %d\n", i, result);
#ifdef _M_AMD64
   printf("Testing bit 0\n");
   result = _bittestandcomplement64(&i64, 0);
   printf("Value changed to %I64d, Result: %d\n", i64, result);
#endif
}
```

```
Initial value: 1
Testing bit 1
Value changed to 3, Result: 0
Testing bit 0
Value changed to 0, Result: 1
```

END Microsoft Specific

See also

Compiler intrinsics

_bittestandreset, _bittestandreset64

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generate the instruction to examine bit `b` of the address `a`, return its current value, and reset the bit to 0.

Syntax

```
unsigned char _bittestandreset(
   long *a,
   long b
);
unsigned char _bittestandreset64(
   __int64 *a,
   __int64 b
);
```

Parameters

a
[in, out] A pointer to the memory to examine.

b
[in] The bit position to test.

Return value

The bit at the position specified.

Requirements

INTRINSIC	ARCHITECTURE
`_bittestandreset`	x86, ARM, x64, ARM64
`_bittestandreset64`	x64, ARM64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

Example

```
// bittestandreset.cpp
// processor: x86, IPF, x64
#include <stdio.h>
#include <limits.h>
#include <intrin.h>

#pragma intrinsic(_bittestandreset)

// Check the sign bit and reset to 0 (taking the absolute value)
// Returns 0 if the number is positive or zero
// Returns 1 if the number is negative
unsigned char absolute_value(long* p)
{
   const int SIGN_BIT = 31;
   return _bittestandreset(p, SIGN_BIT);
}

int main()
{
    long i = -112;
    unsigned char result;

    // Check the sign bit and reset to 0 (taking the absolute value)

    result = absolute_value(&i);
    if (result == 1)
        printf_s("The number was negative.\n");
}
```

```
The number was negative.
```

END Microsoft Specific

See also

Compiler intrinsics

_bittestandset, _bittestandset64

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generate an instruction to examine bit `b` of the address `a`, return its current value, and set the bit to 1.

Syntax

```
unsigned char _bittestandset(
   long *a,
   long b
);
unsigned char _bittestandset64(
   __int64 *a,
   __int64 b
);
```

Parameters

a
[in, out] A pointer to the memory to examine.

b
[in] The bit position to test.

Return value

The bit at the position specified.

Requirements

INTRINSIC	ARCHITECTURE
`_bittestandset`	x86, ARM, x64, ARM64
`_bittestandset64`	x64, ARM64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

Example

```
// bittestandset.cpp
// processor: x86, ARM, x64
// This example uses several of the _bittest family of intrinsics
// to implement a Flags class that allows bit level access to an
// integer field.
#include <stdio.h>
#include <intrin.h>
```

```
#include <intrin.h>

#pragma intrinsic(_bittestandset, _bittestandreset,\
                  _bittestandcomplement, _bittest)

class Flags
{
private:
    long flags;
    long* oldValues;

public:
    Flags() : flags(0)
    {
        oldValues = new long[32];
    }

    ~Flags()
    {
        delete oldValues;
    }

    void SetFlagBit(long nBit)
    {
        // We omit range checks on the argument
        oldValues[nBit] = _bittestandset(&flags, nBit);
        printf_s("Flags: 0x%x\n", flags);
    }
    void ClearFlagBit(long nBit)
    {
        oldValues[nBit] = _bittestandreset(&flags, nBit);
        printf_s("Flags: 0x%x\n", flags);
    }
    unsigned char GetFlagBit(long nBit)
    {
        unsigned char result = _bittest(&flags, nBit);
        printf_s("Flags: 0x%x\n", flags);
        return result;
    }
    void RestoreFlagBit(long nBit)
    {
        if (oldValues[nBit])
            oldValues[nBit] = _bittestandset(&flags, nBit);
        else
            oldValues[nBit] = _bittestandreset(&flags, nBit);
        printf_s("Flags: 0x%x\n", flags);
    }
    unsigned char ToggleBit(long nBit)
    {
        unsigned char result = _bittestandcomplement(&flags, nBit);
        printf_s("Flags: 0x%x\n", flags);
        return result;
    }
};

int main()
{
    Flags f;
    f.SetFlagBit(1);
    f.SetFlagBit(2);
    f.SetFlagBit(3);
    f.ClearFlagBit(3);
    f.ToggleBit(1);
    f.RestoreFlagBit(2);
}
```

```
Flags: 0x2
Flags: 0x6
Flags: 0xe
Flags: 0x6
Flags: 0x4
Flags: 0x0
```

END Microsoft Specific

See also

Compiler intrinsics

__cpuid, __cpuidex

9/2/2022 • 7 minutes to read • Edit Online

Microsoft Specific

Generates the `cpuid` instruction that is available on x86 and x64. This instruction queries the processor for information about supported features and the CPU type.

Syntax

```
void __cpuid(
   int cpuInfo[4],
   int function_id
);

void __cpuidex(
   int cpuInfo[4],
   int function_id,
   int subfunction_id
);
```

Parameters

cpuInfo
[out] An array of four integers that contains the information returned in EAX, EBX, ECX, and EDX about supported features of the CPU.

function_id
[in] A code that specifies the information to retrieve, passed in EAX.

subfunction_id
[in] An additional code that specifies information to retrieve, passed in ECX.

Requirements

INTRINSIC	ARCHITECTURE
`__cpuid`	x86, x64
`__cpuidex`	x86, x64

Header file <intrin.h>

Remarks

This intrinsic stores the supported features and CPU information returned by the `cpuid` instruction in *cpuInfo*, an array of four 32-bit integers that's filled with the values of the EAX, EBX, ECX, and EDX registers (in that order). The information returned has a different meaning depending on the value passed as the *function_id* parameter. The information returned with various values of *function_id* is processor-dependent.

The `__cpuid` intrinsic clears the ECX register before calling the `cpuid` instruction. The `__cpuidex` intrinsic sets the value of the ECX register to *subfunction_id* before it generates the `cpuid` instruction. It enables you to gather additional information about the processor.

For more information about the specific parameters to use and the values returned by these intrinsics on Intel processors, see the documentation for the `cpuid` instruction in Intel 64 and IA-32 Architectures Software Developers Manual Volume 2: Instruction Set Reference and Intel Architecture Instruction Set Extensions Programming Reference. Intel documentation uses the terms "leaf" and "subleaf" for the *function_id* and *subfunction_id* parameters passed in EAX and ECX.

For more information about the specific parameters to use and the values returned by these intrinsics on AMD processors, see the documentation for the `cpuid` instruction in AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions, and in the Revision Guides for specific processor families. For links to these documents and other information, see the AMD Developer Guides, Manuals & ISA Documents page. AMD documentation uses the terms "function number" and "subfunction number" for the *function_id* and *subfunction_id* parameters passed in EAX and ECX.

When the *function_id* argument is 0, *cpuInfo*[0] returns the highest available non-extended *function_id* value supported by the processor. The processor manufacturer is encoded in *cpuInfo*[1], *cpuInfo*[2], and *cpuInfo*[3].

Support for specific instruction set extensions and CPU features is encoded in the *cpuInfo* results returned for higher *function_id* values. For more information, see the manuals linked above, and the following example code.

Some processors support Extended Function CPUID information. When it's supported, *function_id* values from 0x80000000 might be used to return information. To determine the maximum meaningful value allowed, set *function_id* to 0x80000000. The maximum value of *function_id* supported for extended functions will be written to *cpuInfo*[0].

Example

This example shows some of the information available through the `__cpuid` and `__cpuidex` intrinsics. The app lists the instruction set extensions supported by the current processor. The output shows a possible result for a particular processor.

```
// InstructionSet.cpp
// Compile by using: cl /EHsc /W4 InstructionSet.cpp
// processor: x86, x64
// Uses the __cpuid intrinsic to get information about
// CPU extended instruction set support.

#include <iostream>
#include <vector>
#include <bitset>
#include <array>
#include <string>
#include <intrin.h>

class InstructionSet
{
    // forward declarations
    class InstructionSet_Internal;

public:
    // getters
    static std::string Vendor(void) { return CPU_Rep.vendor_; }
    static std::string Brand(void) { return CPU_Rep.brand_; }

    static bool SSE3(void) { return CPU_Rep.f_1_ECX_[0]; }
    static bool PCLMULQDQ(void) { return CPU_Rep.f_1_ECX_[1]; }
    static bool MONITOR(void) { return CPU_Rep.f_1_ECX_[3]; }
    static bool SSSE3(void) { return CPU_Rep.f_1_ECX_[9]; }
    static bool FMA(void) { return CPU_Rep.f_1_ECX_[12]; }
    static bool CMPXCHG16B(void) { return CPU_Rep.f_1_ECX_[13]; }
    static bool SSE41(void) { return CPU_Rep.f_1_ECX_[19]; }
    static bool SSE42(void) { return CPU_Rep.f_1_ECX_[20]; }
    static bool MOVBE(void) { return CPU_Rep.f_1_ECX_[22]; }
    static bool POPCNT(void) { return CPU_Rep.f_1_ECX_[23]; }
    static bool AES(void) { return CPU_Rep.f_1_ECX_[25]; }
    static bool XSAVE(void) { return CPU_Rep.f_1_ECX_[26]; }
    static bool OSXSAVE(void) { return CPU_Rep.f_1_ECX_[27]; }
    static bool AVX(void) { return CPU_Rep.f_1_ECX_[28]; }
    static bool F16C(void) { return CPU_Rep.f_1_ECX_[29]; }
    static bool RDRAND(void) { return CPU_Rep.f_1_ECX_[30]; }

    static bool MSR(void) { return CPU_Rep.f_1_EDX_[5]; }
    static bool CX8(void) { return CPU_Rep.f_1_EDX_[8]; }
    static bool SEP(void) { return CPU_Rep.f_1_EDX_[11]; }
    static bool CMOV(void) { return CPU_Rep.f_1_EDX_[15]; }
    static bool CLFSH(void) { return CPU_Rep.f_1_EDX_[19]; }
    static bool MMX(void) { return CPU_Rep.f_1_EDX_[23]; }
    static bool FXSR(void) { return CPU_Rep.f_1_EDX_[24]; }
    static bool SSE(void) { return CPU_Rep.f_1_EDX_[25]; }
    static bool SSE2(void) { return CPU_Rep.f_1_EDX_[26]; }

    static bool FSGSBASE(void) { return CPU_Rep.f_7_EBX_[0]; }
    static bool BMI1(void) { return CPU_Rep.f_7_EBX_[3]; }
    static bool HLE(void) { return CPU_Rep.isIntel_ && CPU_Rep.f_7_EBX_[4]; }
    static bool AVX2(void) { return CPU_Rep.f_7_EBX_[5]; }
    static bool BMI2(void) { return CPU_Rep.f_7_EBX_[8]; }
    static bool ERMS(void) { return CPU_Rep.f_7_EBX_[9]; }
    static bool INVPCID(void) { return CPU_Rep.f_7_EBX_[10]; }
    static bool RTM(void) { return CPU_Rep.isIntel_ && CPU_Rep.f_7_EBX_[11]; }
    static bool AVX512F(void) { return CPU_Rep.f_7_EBX_[16]; }
    static bool RDSEED(void) { return CPU_Rep.f_7_EBX_[18]; }
    static bool ADX(void) { return CPU_Rep.f_7_EBX_[19]; }
    static bool AVX512PF(void) { return CPU_Rep.f_7_EBX_[26]; }
    static bool AVX512ER(void) { return CPU_Rep.f_7_EBX_[27]; }
    static bool AVX512CD(void) { return CPU_Rep.f_7_EBX_[28]; }
    static bool SHA(void) { return CPU_Rep.f_7_EBX_[29]; }

    static bool PREFETCHWT1(void) { return CPU_Rep.f_7_ECX_[0]; }

    static bool LAHF(void) { return CPU_Rep.f_81_ECX_[0]; }
    static bool LZCNT(void) { return CPU_Rep.isIntel_ && CPU_Rep.f_81_ECX_[5]; }
    static bool ABM(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_ECX_[5]; }
    static bool SSE4a(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_ECX_[6]; }
    static bool XOP(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_ECX_[11]; }
    static bool TBM(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_ECX_[21]; }

    static bool SYSCALL(void) { return CPU_Rep.isIntel_ && CPU_Rep.f_81_EDX_[11]; }
    static bool MMXEXT(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_EDX_[22]; }
    static bool RDTSCP(void) { return CPU_Rep.isIntel_ && CPU_Rep.f_81_EDX_[27]; }
    static bool _3DNOWEXT(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_EDX_[30]; }
    static bool _3DNOW(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_EDX_[31]; }

private:
    static const InstructionSet_Internal CPU_Rep;

    class InstructionSet_Internal
    {
    public:
        InstructionSet_Internal()
            : nIds_{ 0 },
            nExIds_{ 0 },
            isIntel_{ false },
            isAMD_{ false },
            f_1_ECX_{ 0 },
            f_1_EDX_{ 0 },
            f_7_EBX_{ 0 },
            f_7_ECX_{ 0 },
            f_81_ECX_{ 0 },
            f_81_EDX_{ 0 },
            data_{},
            extdata_{}
        {
            //int cpuInfo[4] = {-1};
            std::array<int, 4> cpui;

            // Calling __cpuid with 0x0 as the function_id argument
            // gets the number of the highest valid function ID.
            __cpuid(cpui.data(), 0);
            nIds_ = cpui[0];

            for (int i = 0; i <= nIds_; ++i)
            {
                __cpuidex(cpui.data(), i, 0);
                data_.push_back(cpui);
            }

            // Capture vendor string
            char vendor[0x20];
            memset(vendor, 0, sizeof(vendor));
            *reinterpret_cast<int*>(vendor) = data_[0][1];
            *reinterpret_cast<int*>(vendor + 4) = data_[0][3];
            *reinterpret_cast<int*>(vendor + 8) = data_[0][2];
            vendor_ = vendor;
            if (vendor_ == "GenuineIntel")
            {
                isIntel_ = true;
            }
            else if (vendor_ == "AuthenticAMD")
            {
                isAMD_ = true;
            }

            // load bitset with flags for function 0x00000001
            if (nIds_ >= 1)
            {
                f_1_ECX_ = data_[1][2];
                f_1_EDX_ = data_[1][3];
            }

            // load bitset with flags for function 0x00000007
            if (nIds_ >= 7)
            {
                f_7_EBX_ = data_[7][1];
                f_7_ECX_ = data_[7][2];
            }

            // Calling __cpuid with 0x80000000 as the function_id argument
            // gets the number of the highest valid extended ID.
            __cpuid(cpui.data(), 0x80000000);
            nExIds_ = cpui[0];

            char brand[0x40];
            memset(brand, 0, sizeof(brand));

            for (int i = 0x80000000; i <= nExIds_; ++i)
            {
                __cpuidex(cpui.data(), i, 0);
                extdata_.push_back(cpui);
            }

            // load bitset with flags for function 0x80000001
            if (nExIds_ >= 0x80000001)
            {
                f_81_ECX_ = extdata_[1][2];
                f_81_EDX_ = extdata_[1][3];
            }

            // Interpret CPU brand string if reported
            if (nExIds_ >= 0x80000004)
            {
                memcpy(brand, extdata_[2].data(), sizeof(cpui));
                memcpy(brand + 16, extdata_[3].data(), sizeof(cpui));
                memcpy(brand + 32, extdata_[4].data(), sizeof(cpui));
                brand_ = brand;
            }
        };

        int nIds_;
        int nExIds_;
        std::string vendor_;
        std::string brand_;
        bool isIntel_;
        bool isAMD_;
        std::bitset<32> f_1_ECX_;
        std::bitset<32> f_1_EDX_;
        std::bitset<32> f_7_EBX_;
        std::bitset<32> f_7_ECX_;
        std::bitset<32> f_81_ECX_;
        std::bitset<32> f_81_EDX_;
        std::vector<std::array<int, 4>> data_;
        std::vector<std::array<int, 4>> extdata_;
    };
};

// Initialize static member data
const InstructionSet::InstructionSet_Internal InstructionSet::CPU_Rep;

// Print out supported instruction set extensions
int main()
{
    auto& outstream = std::cout;

    auto support_message = [&outstream](std::string isa_feature, bool is_supported) {
        outstream << isa_feature << (is_supported ? " supported" : " not supported") << std::endl;
    };

    std::cout << InstructionSet::Vendor() << std::endl;
    std::cout << InstructionSet::Brand() << std::endl;

    support_message("3DNOW",       InstructionSet::_3DNOW());
    support_message("3DNOWEXT",    InstructionSet::_3DNOWEXT());
    support_message("ABM",         InstructionSet::ABM());
    support_message("ADX",         InstructionSet::ADX());
    support_message("AES",         InstructionSet::AES());
    support_message("AVX",         InstructionSet::AVX());
    support_message("AVX2",        InstructionSet::AVX2());
    support_message("AVX512CD",    InstructionSet::AVX512CD());
    support_message("AVX512ER",    InstructionSet::AVX512ER());
    support_message("AVX512F",     InstructionSet::AVX512F());
    support_message("AVX512PF",    InstructionSet::AVX512PF());
    support_message("BMI1",        InstructionSet::BMI1());
    support_message("BMI2",        InstructionSet::BMI2());
    support_message("CLFSH",       InstructionSet::CLFSH());
    support_message("CMPXCHG16B",  InstructionSet::CMPXCHG16B());
    support_message("CX8",         InstructionSet::CX8());
    support_message("ERMS",        InstructionSet::ERMS());
    support_message("F16C",        InstructionSet::F16C());
    support_message("FMA",         InstructionSet::FMA());
    support_message("FSGSBASE",    InstructionSet::FSGSBASE());
    support_message("FXSR",        InstructionSet::FXSR());
    support_message("HLE",         InstructionSet::HLE());
    support_message("INVPCID",     InstructionSet::INVPCID());
    support_message("LAHF",        InstructionSet::LAHF());
    support_message("LZCNT",       InstructionSet::LZCNT());
    support_message("MMX",         InstructionSet::MMX());
    support_message("MMXEXT",      InstructionSet::MMXEXT());
    support_message("MONITOR",     InstructionSet::MONITOR());
    support_message("MOVBE",       InstructionSet::MOVBE());
    support_message("MSR",         InstructionSet::MSR());
    support_message("OSXSAVE",     InstructionSet::OSXSAVE());
    support_message("PCLMULQDQ",   InstructionSet::PCLMULQDQ());
    support_message("POPCNT",      InstructionSet::POPCNT());
    support_message("PREFETCHWT1", InstructionSet::PREFETCHWT1());
    support_message("RDRAND",      InstructionSet::RDRAND());
    support_message("RDSEED",      InstructionSet::RDSEED());
    support_message("RDTSCP",      InstructionSet::RDTSCP());
    support_message("RTM",         InstructionSet::RTM());
    support_message("SEP",         InstructionSet::SEP());
    support_message("SHA",         InstructionSet::SHA());
    support_message("SSE",         InstructionSet::SSE());
    support_message("SSE2",        InstructionSet::SSE2());
    support_message("SSE3",        InstructionSet::SSE3());
    support_message("SSE4.1",      InstructionSet::SSE41());
    support_message("SSE4.2",      InstructionSet::SSE42());
    support_message("SSE4a",       InstructionSet::SSE4a());
    support_message("SSSE3",       InstructionSet::SSSE3());
    support_message("SYSCALL",     InstructionSet::SYSCALL());
    support_message("TBM",         InstructionSet::TBM());
    support_message("XOP",         InstructionSet::XOP());
    support_message("XSAVE",       InstructionSet::XSAVE());
}
```

```
GenuineIntel
        Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz
3DNOW not supported
3DNOWEXT not supported
ABM not supported
ADX not supported
AES supported
AVX supported
AVX2 not supported
AVX512CD not supported
AVX512ER not supported
AVX512F not supported
AVX512PF not supported
BMI1 not supported
BMI2 not supported
CLFSH supported
CMPXCHG16B supported
CX8 supported
ERMS not supported
F16C not supported
FMA not supported
FSGSBASE not supported
FXSR supported
HLE not supported
INVPCID not supported
LAHF supported
LZCNT not supported
MMX supported
MMXEXT not supported
MONITOR not supported
MOVBE not supported
MSR supported
OSXSAVE supported
PCLMULQDQ supported
POPCNT supported
PREFETCHWT1 not supported
RDRAND not supported
RDSEED not supported
RDTSCP supported
RTM not supported
SEP supported
SHA not supported
SSE supported
SSE2 supported
SSE3 supported
SSE4.1 supported
SSE4.2 supported
SSE4a not supported
SSSE3 supported
SYSCALL supported
TBM not supported
XOP not supported
XSAVE supported
```

END Microsoft Specific

See also

Compiler intrinsics

__debugbreak

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Causes a breakpoint in your code, where the user will be prompted to run the debugger.

Syntax

```
void __debugbreak();
```

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`__debugbreak`	x86, x64, ARM, ARM64	<intrin.h>

Remarks

The `__debugbreak` compiler intrinsic, similar to DebugBreak, is a portable Win32 way to cause a breakpoint.

NOTE

When compiling with **/clr**, a function containing `__debugbreak` will be compiled to MSIL. `asm int 3` causes a function to be compiled to native. For more information, see __asm.

For example:

```
main() {
   __debugbreak();
}
```

is similar to:

```
main() {
   __asm {
      int 3
   }
}
```

on an x86 computer.

On ARM64, the `__debugbreak` intrinsic is compiled into the instruction `brk #0xF000`.

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

Keywords

_disable

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Disables interrupts.

Syntax

```
void _disable(void);
```

Requirements

INTRINSIC	ARCHITECTURE
`_disable`	x86, ARM, x64, ARM64

Header file <intrin.h>

Remarks

`_disable` instructs the processor to clear the interrupt flag. On x86 systems, this function generates the Clear Interrupt Flag (`cli`) instruction.

This function is only available in kernel mode. If used in user mode, a Privileged Instruction exception is thrown at run time.

On ARM and ARM64 platforms, this routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

_div128

9/2/2022 • 2 minutes to read • Edit Online

The `_div128` intrinsic divides a 128-bit integer by a 64-bit integer. The return value holds the quotient, and the intrinsic returns the remainder through a pointer parameter. `_div128` is **Microsoft-specific**.

Syntax

```
__int64 _div128(
   __int64 highDividend,
   __int64 lowDividend,
   __int64 divisor,
   __int64 *remainder
);
```

Parameters

highDividend
[in] The high 64 bits of the dividend.

lowDividend
[in] The low 64 bits of the dividend.

divisor
[in] The 64-bit integer to divide by.

remainder
[out] The 64-bit integer bits of the remainder.

Return value

The 64 bits of the quotient.

Remarks

Pass the upper 64 bits of the 128-bit dividend in *highDividend*, and the lower 64 bits in *lowDividend*. The intrinsic divides this value by *divisor*. It stores the remainder in the 64-bit integer pointed to by *remainder*, and returns the 64 bits of the quotient.

The `_div128` intrinsic is available starting in Visual Studio 2019 RTM.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_div128`	x64	<immintrin.h>

See also

_udiv128
Compiler intrinsics

_div64

9/2/2022 • 2 minutes to read • Edit Online

The `_div64` intrinsic divides a 64-bit integer by a 32-bit integer. The return value holds the quotient, and the intrinsic returns the remainder through a pointer parameter. `_div64` is **Microsoft-specific**.

Syntax

```
int _div64(
   __int64 dividend,
   int divisor,
   int* remainder
);
```

Parameters

dividend
[in] The 64-bit integer to divide.

divisor
[in] The 32-bit integer to divide by.

remainder
[out] The 32-bit integer bits of the remainder.

Return value

The 32 bits of the quotient.

Remarks

The `_div64` intrinsic divides *dividend* by *divisor*. It stores the remainder in the 32-bit integer pointed to by *remainder*, and returns the 32 bits of the quotient.

The `_div64` intrinsic is available starting in Visual Studio 2019 RTM.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_div64`	x86, x64	<immintrin.h>

See also

_udiv64
Compiler intrinsics

__emul, __emulu

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Performs multiplications that overflow what a 32-bit integer can hold.

Syntax

```
__int64 __emul(
   int a,
   int b
);
unsigned __int64 __emulu(
   unsigned int a,
   unsigned int b
);
```

Parameters

a
[in] The first integer operand of the multiplication.

b
[in] The second integer operand of the multiplication.

Return value

The result of the multiplication.

Requirements

INTRINSIC	ARCHITECTURE
`__emul`	x86, x64
`__emulu`	x86, x64

Header file <intrin.h>

Remarks

`__emul` takes two 32-bit signed values and returns the result of the multiplication as a 64-bit signed integer value.

`__emulu` takes two 32-bit unsigned integer values and returns the result of the multiplication as a 64-bit unsigned integer value.

Example

```
// emul.cpp
// compile with: /EHsc
// processor: x86, x64
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(__emul)
#pragma intrinsic(__emulu)

int main()
{
   int a = -268435456;
   int b = 2;

   __int64 result = __emul(a, b);

   cout << a << " * " << b << " = " << result << endl;

   unsigned int ua = 0xFFFFFFFF; // Dec value: 4294967295
   unsigned int ub = 0xF000000;  // Dec value: 251658240

   unsigned __int64 uresult = __emulu(ua, ub);

   cout << ua << " * " << ub << " = " << uresult << endl;

}
```

Output

```
-268435456 * 2 = -536870912
4294967295 * 251658240 = 1080863910317260800
```

END Microsoft Specific

See also

Compiler intrinsics

_enable

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Enables interrupts.

Syntax

```
void _enable(void);
```

Requirements

INTRINSIC	ARCHITECTURE
`_enable`	x86, ARM, x64, ARM64

Header file <intrin.h>

Remarks

`_enable` instructs the processor to set the interrupt flag. On x86 systems, this function generates the Set Interrupt Flag (`sti`) instruction.

This function is only available in kernel mode. If used in user mode, a Privileged Instruction exception is thrown.

END Microsoft Specific

See also

Compiler intrinsics

__fastfail

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Immediately terminates the calling process with minimum overhead.

Syntax

```
void __fastfail(unsigned int code);
```

Parameters

code
[in] A `FAST_FAIL_<description>` symbolic constant from winnt.h or wdm.h that indicates the reason for process termination.

Return value

The `__fastfail` intrinsic does not return.

Remarks

The `__fastfail` intrinsic provides a mechanism for a *fast fail* request—a way for a potentially corrupted process to request immediate process termination. Critical failures that may have corrupted program state and stack beyond recovery cannot be handled by the regular exception handling facility. Use `__fastfail` to terminate the process using minimal overhead.

Internally, `__fastfail` is implemented by using several architecture-specific mechanisms:

ARCHITECTURE	INSTRUCTION	LOCATION OF CODE ARGUMENT
x86	int 0x29	`ecx`
x64	int 0x29	`rcx`
ARM	Opcode 0xDEFB	`r0`
ARM64	Opcode 0xF003	`x0`

A fast fail request is self-contained and typically requires just two instructions to execute. After a fast fail request has been executed, the kernel then takes the appropriate action. In user-mode code, there are no memory dependencies beyond the instruction pointer itself when a fast fail event is raised. That maximizes its reliability, even in cases of severe memory corruption.

The `code` argument, one of the `FAST_FAIL_<description>` symbolic constants from winnt.h or wdm.h, describes the type of failure condition. It's incorporated into failure reports in an environment-specific manner.

User-mode fast fail requests appear as a second chance non-continuable exception with exception code 0xC0000409, and with at least one exception parameter. The first exception parameter is the `code` value. This exception code indicates to the Windows Error Reporting (WER) and debugging infrastructure that the process is corrupted, and that minimal in-process actions should be taken in response to the failure. Kernel-mode fast fail requests are implemented by using a dedicated bugcheck code, `KERNEL_SECURITY_CHECK_FAILURE` (0x139). In both cases, no exception handlers are invoked because the program is expected to be in a corrupted state. If a debugger is present, it's given an opportunity to examine the state of the program before termination.

Support for the native fast fail mechanism began in Windows 8. Windows operating systems that don't support the fast fail instruction natively will typically treat a fast fail request as an access violation, or as an `UNEXPECTED_KERNEL_MODE_TRAP` bugcheck. In these cases, the program is still terminated, but not necessarily as quickly.

`__fastfail` is only available as an intrinsic.

Requirements

INTRINSIC	ARCHITECTURE
`__fastfail`	x86, x64, ARM, ARM64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics

__faststorefence

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Guarantees that every previous memory reference, including both load and store memory references, is globally visible before any subsequent memory reference.

Syntax

```
void __faststorefence();
```

Requirements

INTRINSIC	ARCHITECTURE
`__faststorefence`	x64

Header file <intrin.h>

Remarks

Generates a full memory barrier instruction sequence that guarantees load and store operations issued before the intrinsic are globally visible before execution continues. The effect is comparable to but faster than the `_mm_mfence` intrinsic on all x64 platforms.

On the AMD64 platform, this routine generates an instruction that is a faster store fence than the `sfence` instruction. For time-critical code, use this intrinsic instead of `_mm_sfence` only on AMD64 platforms. On Intel x64 platforms, the `_mm_sfence` instruction is faster.

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

Fast floating-point conversion functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Fast conversion functions between floating-point types and integral types.

Syntax

```
int _cvt_ftoi_fast(float value);
long long _cvt_ftoll_fast(float value);
unsigned _cvt_ftoui_fast(float value);
unsigned long long _cvt_ftoull_fast(float value);
int _cvt_dtoi_fast(double value);
long long _cvt_dtoll_fast(double value);
unsigned _cvt_dtoui_fast(double value);
unsigned long long _cvt_dtoull_fast(double value);
```

Parameters

value
[in] A floating-point value to convert.

Return value

The integer-typed result of the conversion.

Requirements

Header: <intrin.h>

Architecture: x86, x64

Remarks

These intrinsics are fast conversion functions that execute as quickly as possible for valid conversions. As in Standard C++, fast conversions aren't fully defined. They may generate different values or exceptions for invalid conversions. The results depend on the target platform, compiler options, and context. These functions can be useful for handling values that have already been range-checked. Or, for values generated in a way that can never cause an invalid conversion.

The fast conversion intrinsics are available starting in Visual Studio 2022.

END Microsoft Specific

See also

Compiler intrinsics
Saturation floating-point conversion functions
Sentinel floating-point conversion functions

Saturation floating-point conversion functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Conversion functions between floating-point types and integral types that use an ARM processor-compatible saturation strategy.

Syntax

```
int _cvt_ftoi_sat(float value);
long long _cvt_ftoll_sat(float value);
unsigned _cvt_ftoui_sat(float value);
unsigned long long _cvt_ftoull_sat(float value);
int _cvt_dtoi_sat(double value);
long long _cvt_dtoll_sat(double value);
unsigned _cvt_dtoui_sat(double value);
unsigned long long _cvt_dtoull_sat(double value);
```

Parameters

`value`
[in] A floating-point value to convert.

Return value

The integer-typed result of the conversion.

Requirements

Header: <intrin.h>

Architecture: x86, x64

Remarks

These intrinsics are floating-point to integral type conversion functions that use a *saturation* strategy: Any floating-point value too high to fit in the destination type is mapped to the highest possible destination value. Any value too low to fit maps to the lowest possible value. And if the source value is NaN, zero is returned for the result.

The saturation conversion intrinsics are available starting in Visual Studio 2019 version 16.10.

END Microsoft Specific

See also

Compiler intrinsics
Fast floating-point conversion functions
Sentinel floating-point conversion functions

Sentinel floating-point conversion functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Conversion functions between floating-point types and integral types that use an Intel Architecture (IA) AVX-512 compatible sentinel strategy.

Syntax

```
int _cvt_ftoi_sent(float value);
long long _cvt_ftoll_sent(float value);
unsigned _cvt_ftoui_sent(float value);
unsigned long long _cvt_ftoull_sent(float value);
int _cvt_dtoi_sent(double value);
long long _cvt_dtoll_sent(double value);
unsigned _cvt_dtoui_sent(double value);
unsigned long long _cvt_dtoull_sent(double value);
```

Parameters

value
[in] A floating-point value to convert.

Return value

The integer-typed result of the conversion.

Requirements

Header: <intrin.h>

Architecture: x86, x64

Remarks

These intrinsics are floating-point to integral type conversion functions that use a *sentinel* strategy: They return the result value farthest from zero as a proxy sentinel value for NaN. Any invalid conversion returns this sentinel value. The specific sentinel value returned depends on the result type.

RESULT TYPE	SENTINEL	`<LIMITS.H>` CONSTANT
`int`	-2147483648 (0xFFFFFFFF)	`INT_MIN`
`unsigned int`	4294967295 (0xFFFFFFFF)	`UINT_MAX`
`long long`	-9223372036854775808 (0xFFFFFFFF'FFFFFFFF)	`LLONG_MIN`
`unsigned long long`	18446744073709551615 (0xFFFFFFFF'FFFFFFFF)	`ULLONG_MAX`

The sentinel conversion intrinsics are available starting in Visual Studio 2019 version 16.10.

END Microsoft Specific

See also

Compiler intrinsics
Fast floating-point conversion functions
Saturation floating-point conversion functions

__getcallerseflags

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Returns the EFLAGS value from the caller's context.

Syntax

```
unsigned int __getcallerseflags(void);
```

Return value

EFLAGS value from the caller's context.

Requirements

INTRINSIC	ARCHITECTURE
`__getcallerseflags`	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

Example

```
// getcallerseflags.cpp
// processor: x86, x64

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__getcallerseflags)

unsigned int g()
{
    unsigned int EFLAGS = __getcallerseflags();
    printf_s("EFLAGS 0x%x\n", EFLAGS);
    return EFLAGS;
}
unsigned int f()
{
    return g();
}

int main()
{
    unsigned int i;
    i = f();
    i = g();
    return 0;
}
```

```
EFLAGS 0x202
EFLAGS 0x206
```

END Microsoft Specific

See also

Compiler intrinsics

__halt

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Halts the microprocessor until an enabled interrupt, a nonmaskable interrupt (NMI), or a reset occurs.

Syntax

```
void __halt( void );
```

Requirements

INTRINSIC	ARCHITECTURE
`__halt`	x86, x64

Header file <intrin.h>

Remarks

The `__halt` function is equivalent to the `HLT` machine instruction, and is available only in kernel mode. For more information, search for the document, "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference," at the Intel Corporation site.

END Microsoft Specific

See also

Compiler intrinsics

_InterlockedAdd intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

These functions perform an atomic addition, which makes sure that the operation completes successfully when more than one thread has access to a shared variable.

Syntax

```
long _InterlockedAdd(
   long volatile * Addend,
   long Value
);
long _InterlockedAdd_acq(
   long volatile * Addend,
   long Value
);
long _InterlockedAdd_nf(
   long volatile * Addend,
   long Value
);
long _InterlockedAdd_rel(
   long volatile * Addend,
   long Value
);
__int64 _InterlockedAdd64(
   __int64 volatile * Addend,
   __int64 Value
);
__int64 _InterlockedAdd64_acq(
   __int64 volatile * Addend,
   __int64 Value
);
__int64 _InterlockedAdd64_nf (
   __int64 volatile * Addend,
   __int64 Value
);
__int64 _InterlockedAdd64_rel(
   __int64 volatile * Addend,
   __int64 Value
);
```

Parameters

Addend
[in, out] Pointer to the integer to be added to; replaced by the result of the addition.

Value
[in] The value to add.

Return value

Both functions return the result of the addition.

Requirements

INTRINSIC	ARCHITECTURE
`_InterlockedAdd`	ARM, ARM64
`_InterlockedAdd_acq`	ARM, ARM64
`_InterlockedAdd_nf`	ARM, ARM64
`_InterlockedAdd_rel`	ARM, ARM64
`_InterlockedAdd64`	ARM, ARM64
`_InterlockedAdd64_acq`	ARM, ARM64
`_InterlockedAdd64_nf`	ARM, ARM64
`_InterlockedAdd64_rel`	ARM, ARM64

Header file <intrin.h>

Remarks

The versions of these functions with the `_acq` or `_rel` suffixes perform an interlocked addition following acquire or release semantics. *Acquire semantics* means that the result of the operation is made visible to all threads and processors before any later memory reads and writes. Acquire is useful when entering a critical section. *Release semantics* means that all memory reads and writes are forced to be made visible to all threads and processors before the result of the operation is made visible itself. Release is useful when leaving a critical section. The intrinsics with an `_nf` ("no fence") suffix don't act as a memory barrier.

These routines are only available as intrinsics.

Example: `_InterlockedAdd`

```
// interlockedadd.cpp
// Compile with: /Oi /EHsc
// processor: ARM
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_InterlockedAdd)

int main()
{
        long data1 = 0xFF00FF00;
        long data2 = 0x00FF0000;
        long retval;
        retval = _InterlockedAdd(&data1, data2);
        printf("0x%x 0x%x 0x%x", data1, data2, retval);
}
```

Output: `_InterlockedAdd`

```
0xffffff00 0xff0000 0xffffff00
```

Example: `_InterlockedAdd64`

```
// interlockedadd64.cpp
// compile with: /Oi /EHsc
// processor: ARM
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(_InterlockedAdd64)

int main()
{
        __int64 data1 = 0x0000FF0000000000;
        __int64 data2 = 0x00FF0000FFFFFFFF;
        __int64 retval;
        cout << hex << data1 << " + " << data2 << " = " ;
        retval = _InterlockedAdd64(&data1, data2);
        cout << data1 << endl;
        cout << "Return value: " << retval << endl;
}
```

Output: `_InterlockedAdd64`

```
ff0000000000 + ff0000ffffffff = ffff00ffffffff
Return value: ffff00ffffffff
```

END Microsoft Specific

See also

Compiler intrinsics
Conflicts with the x86 Compiler

_InterlockedAddLargeStatistic

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Performs an interlocked addition in which the first operand is a 64-bit value.

Syntax

```
long _InterlockedAddLargeStatistic(
   __int64 volatile * Addend,
   long Value
);
```

Parameters

Addend
[in, out] A pointer to the first operand to the add operation. The value pointed to is replaced by the result of the addition.

Value
[in] The second operand; value to add to the first operand.

Return value

The value of the second operand.

Requirements

INTRINSIC	ARCHITECTURE
`_InterlockedAddLargeStatistic`	x86

Header file <intrin.h>

Remarks

The `_InterlockedAddLargeStatistic` intrinsic isn't atomic, because it's implemented as two separate locked instructions. An atomic 64-bit read that occurs on another thread during the execution of the intrinsic could result in a read of an inconsistent value.

`_InterlockedAddLargeStatistic` behaves as a read-write barrier. For more information, see _ReadWriteBarrier.

END Microsoft Specific

See also

Compiler intrinsics
Conflicts with the x86 Compiler

_InterlockedAnd intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Used to perform an atomic bitwise AND operation on a variable shared by multiple threads.

Syntax

```
long _InterlockedAnd(
   long volatile * value,
   long mask
);
long _InterlockedAnd_acq(
   long volatile * value,
   long mask
);
long _InterlockedAnd_HLEAcquire(
   long volatile * value,
   long mask
);
long _InterlockedAnd_HLERelease(
   long volatile * value,
   long mask
);
long _InterlockedAnd_nf(
   long volatile * value,
   long mask
);
long _InterlockedAnd_np(
   long volatile * value,
   long mask
);
long _InterlockedAnd_rel(
   long volatile * value,
   long mask
);
char _InterlockedAnd8(
   char volatile * value,
   char mask
);
char _InterlockedAnd8_acq(
   char volatile * value,
   char mask
);
char _InterlockedAnd8_nf(
   char volatile * value,
   char mask
);
char _InterlockedAnd8_np(
   char volatile * value,
   char mask
);
char _InterlockedAnd8_rel(
   char volatile * value,
   char mask
);
short _InterlockedAnd16(
   short volatile * value,
   short mask
);
```

```
short _InterlockedAnd16_acq(
   short volatile * value,
   short mask
);
short _InterlockedAnd16_nf(
   short volatile * value,
   short mask
);
short _InterlockedAnd16_np(
   short volatile * value,
   short mask
);
short _InterlockedAnd16_rel(
   short volatile * value,
   short mask
);
__int64 _InterlockedAnd64(
   __int64 volatile* value,
   __int64 mask
);
__int64 _InterlockedAnd64_acq(
   __int64 volatile* value,
   __int64 mask
);
__int64 _InterlockedAnd64_HLEAcquire(
   __int64 volatile* value,
   __int64 mask
);
__int64 _InterlockedAnd64_HLERelease(
   __int64 volatile* value,
   __int64 mask
);
__int64 _InterlockedAnd64_nf(
   __int64 volatile* value,
   __int64 mask
);
__int64 _InterlockedAnd64_np(
   __int64 volatile* value,
   __int64 mask
);
__int64 _InterlockedAnd64_rel(
   __int64 volatile* value,
   __int64 mask
);
```

Parameters

value
[in, out] A pointer to the first operand, to be replaced by the result.

mask
[in] The second operand.

Return value

The original value of the first operand.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedAnd`, `_InterlockedAnd8`, `_InterlockedAnd16`	x86, ARM, x64, ARM64	<intrin.h>
`_InterlockedAnd64`	ARM, x64, ARM64	<intrin.h>
`_InterlockedAnd_acq`, `_InterlockedAnd_nf`, `_InterlockedAnd_rel`, `_InterlockedAnd8_acq`, `_InterlockedAnd8_nf`, `_InterlockedAnd8_rel`, `_InterlockedAnd16_acq`, `_InterlockedAnd16_nf`, `_InterlockedAnd16_rel`, `_InterlockedAnd64_acq`, `_InterlockedAnd64_nf`, `_InterlockedAnd64_rel`	ARM, ARM64	<intrin.h>
`_InterlockedAnd_np`, `_InterlockedAnd8_np`, `_InterlockedAnd16_np`, `_InterlockedAnd64_np`	x64	<intrin.h>
`_InterlockedAnd_HLEAcquire`, `_InterlockedAnd_HLERelease`, `_InterlockedAnd64_HLEAcquire`, `_InterlockedAnd64_HLERelease`	x86, x64	<immintrin.h>

Remarks

The number in the name of each function specifies the bit size of the arguments.

On ARM and ARM64 platforms, use the intrinsics with `_acq` and `_rel` suffixes for acquire and release semantics, such as at the beginning and end of a critical section. The intrinsics with an `_nf` ("no fence") suffix do not act as a memory barrier.

The intrinsics with an `_np` ("no prefetch") suffix prevent a possible prefetch operation from being inserted by the compiler.

On Intel platforms that support Hardware Lock Elision (HLE) instructions, the intrinsics with `_HLEAcquire` and `_HLERelease` suffixes include a hint to the processor that can accelerate performance by eliminating a lock write step in hardware. If these intrinsics are called on platforms that do not support HLE, the hint is ignored.

Example

```
// InterlockedAnd.cpp
// Compile with: /Oi
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_InterlockedAnd)

int main()
{
        long data1 = 0xFF00FF00;
        long data2 = 0x00FFFF00;
        long retval;
        retval = _InterlockedAnd(&data1, data2);
        printf_s("0x%x 0x%x 0x%x", data1, data2, retval);
}
```

```
0xff00 0xffff00 0xff00ff00
```

END Microsoft Specific

See also

Compiler intrinsics
Conflicts with the x86 Compiler

_interlockedbittestandreset intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates an instruction to set bit `b` of the address `a` to zero and return its original value.

Syntax

```
unsigned char _interlockedbittestandreset(
   long *a,
   long b
);
unsigned char _interlockedbittestandreset_acq(
   long *a,
   long b
);
unsigned char _interlockedbittestandreset_HLEAcquire(
   long *a,
   long b
);
unsigned char _interlockedbittestandreset_HLERelease(
   long *a,
   long b
);
unsigned char _interlockedbittestandreset_nf(
   long *a,
   long b
);
unsigned char _interlockedbittestandreset_rel(
   long *a,
   long b
);
unsigned char _interlockedbittestandreset64(
   __int64 *a,
   __int64 b
);
unsigned char _interlockedbittestandreset64_acq(
   __int64 *a,
   __int64 b
);
unsigned char _interlockedbittestandreset64_nf(
   __int64 *a,
   __int64 b
);
unsigned char _interlockedbittestandreset64_rel(
   __int64 *a,
   __int64 b
);
unsigned char _interlockedbittestandreset64_HLEAcquire(
   __int64 *a,
   __int64 b
);
unsigned char _interlockedbittestandreset64_HLERelease(
   __int64 *a,
   __int64 b
);
```

Parameters

a
[in] A pointer to the memory to examine.

b
[in] The bit position to test.

Return value

The original value of the bit at the position specified by `b`.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_interlockedbittestandreset`	x86, ARM, x64, ARM64	<intrin.h>
`_interlockedbittestandreset_acq`, `_interlockedbittestandreset_nf`, `_interlockedbittestandreset_rel`	ARM, ARM64	<intrin.h>
`_interlockedbittestandreset64_acq`, `_interlockedbittestandreset64_nf`, `_interlockedbittestandreset64_rel`	ARM64	<intrin.h>
`_interlockedbittestandreset_HLEAcquire`, `_interlockedbittestandreset_HLERelease`	x86, x64	<immintrin.h>
`_interlockedbittestandreset64`	x64, ARM64	<intrin.h>
`_interlockedbittestandreset64_HLEAcquire`, `_interlockedbittestandreset64_HLERelease`	x64	<immintrin.h>

Remarks

On x86 and x64 processors, these intrinsics use the `lock btr` instruction, that reads and sets the specified bit to zero in an atomic operation.

On ARM processors, use the intrinsics with `_acq` and `_rel` suffixes for acquire and release semantics, such as at the beginning and end of a critical section. The ARM intrinsics with an `_nf` ("no fence") suffix don't act as a memory barrier.

On Intel processors that support Hardware Lock Elision (HLE) instructions, the intrinsics with `_HLEAcquire` and `_HLERelease` suffixes include a hint to the processor that can accelerate performance by eliminating a lock write step in hardware. If these intrinsics are called on processors that don't support HLE, the hint is ignored.

These routines are only available as intrinsics.

END Microsoft Specific

See also

Compiler intrinsics

Conflicts with the x86 Compiler

_interlockedbittestandset intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generate an instruction to examine bit `b` of the address `a` and return its current value before setting it to 1.

Syntax

```
unsigned char _interlockedbittestandset(
   long *a,
   long b
);
unsigned char _interlockedbittestandset_acq(
   long *a,
   long b
);
unsigned char _interlockedbittestandset_HLEAcquire(
   long *a,
   long b
);
unsigned char _interlockedbittestandset_HLERelease(
   long *a,
   long b
);
unsigned char _interlockedbittestandset_nf(
   long *a,
   long b
);
unsigned char _interlockedbittestandset_rel(
   long *a,
   long b
);
unsigned char _interlockedbittestandset64(
   __int64 *a,
   __int64 b
);
unsigned char _interlockedbittestandset64_acq(
   __int64 *a,
   __int64 b
);
unsigned char _interlockedbittestandset64_nf(
   __int64 *a,
   __int64 b
);
unsigned char _interlockedbittestandset64_rel(
   __int64 *a,
   __int64 b
);
unsigned char _interlockedbittestandset64_HLEAcquire(
   __int64 *a,
   __int64 b
);
unsigned char _interlockedbittestandset64_HLERelease(
   __int64 *a,
   __int64 b
);
```

Parameters

a
[in] A pointer to the memory to examine.

b
[in] The bit position to test.

Return value

The value of the bit at position `b` before it's set.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_interlockedbittestandset`	x86, ARM, x64, ARM64	<intrin.h>
`_interlockedbittestandset_acq`, `_interlockedbittestandset_nf`, `_interlockedbittestandset_rel`	ARM, ARM64	<intrin.h>
`_interlockedbittestandset64_acq`, `_interlockedbittestandset64_nf`, `_interlockedbittestandset64_rel`	ARM64	<intrin.h>
`_interlockedbittestandset_HLEAcquire`, `_interlockedbittestandset_HLERelease`	x86, x64	<immintrin.h>
`_interlockedbittestandset64`	x64, ARM64	<intrin.h>
`_interlockedbittestandset64_HLEAcquire`, `_interlockedbittestandset64_HLERelease`	x64	<immintrin.h>

Remarks

On x86 and x64 processors, these intrinsics use the `lock bts` instruction to read and set the specified bit to 1. The operation is atomic.

On ARM and ARM64 processors, use the intrinsics with `_acq` and `_rel` suffixes for acquire and release semantics, such as at the beginning and end of a critical section. The ARM intrinsics with an `_nf` ("no fence") suffix don't act as a memory barrier.

On Intel processors that support Hardware Lock Elision (HLE) instructions, the intrinsics with `_HLEAcquire` and `_HLERelease` suffixes include a hint to the processor that can accelerate performance by eliminating a lock write step in hardware. If these intrinsics are called on processors that don't support HLE, the hint is ignored.

These routines are only available as intrinsics.

END Microsoft Specific

See also

[Compiler intrinsics]
[Conflicts with the x86 Compiler]

`_InterlockedCompareExchange` intrinsic functions

9/2/2022 • 5 minutes to read • Edit Online

Microsoft Specific

Does an interlocked compare and exchange.

Syntax

```
long _InterlockedCompareExchange(
   long volatile * Destination,
   long Exchange,
   long Comparand
);
long _InterlockedCompareExchange_acq(
   long volatile * Destination,
   long Exchange,
   long Comparand
);
long _InterlockedCompareExchange_HLEAcquire(
   long volatile * Destination,
   long Exchange,
   long Comparand
);
long _InterlockedCompareExchange_HLERelease(
   long volatile * Destination,
   long Exchange,
   long Comparand
);
long _InterlockedCompareExchange_nf(
   long volatile * Destination,
   long Exchange,
   long Comparand
);
long _InterlockedCompareExchange_np(
   long volatile * Destination,
   long Exchange,
   long Comparand
);
long _InterlockedCompareExchange_rel(
   long volatile * Destination,
   long Exchange,
   long Comparand
);
char _InterlockedCompareExchange8(
   char volatile * Destination,
   char Exchange,
   char Comparand
);
char _InterlockedCompareExchange8_acq(
   char volatile * Destination,
   char Exchange,
   char Comparand
);
char _InterlockedCompareExchange8_nf(
   char volatile * Destination,
   char Exchange,
   char Comparand
);
char _InterlockedCompareExchange8_rel(
   char volatile * Destination,
   char Exchange,
   char Comparand
);
short _InterlockedCompareExchange16(
   short volatile * Destination,
   short Exchange,
   short Comparand
);
short _InterlockedCompareExchange16_acq(
   short volatile * Destination,
   short Exchange,
   short Comparand
);
short _InterlockedCompareExchange16_nf(
   short volatile * Destination,
   short Exchange,
   short Comparand
);
short _InterlockedCompareExchange16_np(
   short volatile * Destination,
   short Exchange,
   short Comparand
);
short _InterlockedCompareExchange16_rel(
   short volatile * Destination,
   short Exchange,
   short Comparand
);
__int64 _InterlockedCompareExchange64(
   __int64 volatile * Destination,
   __int64 Exchange,
   __int64 Comparand
);
__int64 _InterlockedCompareExchange64_acq(
   __int64 volatile * Destination,
   __int64 Exchange,
   __int64 Comparand
);
__int64 _InterlockedCompareExchange64_HLEAcquire (
   __int64 volatile * Destination,
   __int64 Exchange,
   __int64 Comparand
);
__int64 _InterlockedCompareExchange64_HLERelease(
   __int64 volatile * Destination,
   __int64 Exchange,
   __int64 Comparand
);
__int64 _InterlockedCompareExchange64_nf(
   __int64 volatile * Destination,
   __int64 Exchange,
   __int64 Comparand
);
__int64 _InterlockedCompareExchange64_np(
   __int64 volatile * Destination,
   __int64 Exchange,
   __int64 Comparand
);
__int64 _InterlockedCompareExchange64_rel(
   __int64 volatile * Destination,
   __int64 Exchange,
   __int64 Comparand
);
```

Parameters

`Destination`

[in, out] Pointer to the destination value. The sign is ignored.

`Exchange`
[in] Exchange value. The sign is ignored.

`Comparand`
[in] Value to compare to the value pointed at by `Destination`. The sign is ignored.

Return value

The return value is the initial value pointed at by the `Destination` pointer.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedCompareExchange`, `_InterlockedCompareExchange8`, `_InterlockedCompareExchange16`, `_InterlockedCompareExchange64`	x86, ARM, x64, ARM64	<intrin.h>
`_InterlockedCompareExchange_acq`, `_InterlockedCompareExchange_nf`, `_InterlockedCompareExchange_rel`, `_InterlockedCompareExchange8_acq`, `_InterlockedCompareExchange8_nf`, `_InterlockedCompareExchange8_rel`, `_InterlockedCompareExchange16_acq`, `_InterlockedCompareExchange16_nf`, `_InterlockedCompareExchange16_rel`, `_InterlockedCompareExchange64_acq`, `_InterlockedCompareExchange64_nf`, `_InterlockedCompareExchange64_rel`,	ARM, ARM64	<intrin.h>
`_InterlockedCompareExchange_np`, `_InterlockedCompareExchange16_np`, `_InterlockedCompareExchange64_np`	x64	<intrin.h>
`_InterlockedCompareExchange_HLEAcquire`, `_InterlockedCompareExchange_HLERelease`, `_InterlockedCompareExchange64_HLEAcquire`, `_InterlockedCompareExchange64_HLERelease`	x86, x64	<immintrin.h>

Remarks

`_InterlockedCompareExchange` does an atomic comparison of the value pointed at by `Destination` with the `Comparand` value. If the `Destination` value is equal to the `Comparand` value, the `Exchange` value is stored in the address specified by `Destination`. Otherwise, does no operation.

`_InterlockedCompareExchange` provides compiler intrinsic support for the Win32 Windows SDK `InterlockedCompareExchange` function.

There are several variations on `_InterlockedCompareExchange` that vary based on the data types they involve and whether processor-specific acquire or release semantics are used.

While the `_InterlockedCompareExchange` function operates on 32-bit `long` integer values, `_InterlockedCompareExchange8` operates on 8-bit integer values, `_InterlockedCompareExchange16` operates on 16-bit `short` integer values, and `_InterlockedCompareExchange64` operates on 64-bit integer values. For more information on similar intrinsics for 128-bit values, see `_InterlockedCompareExchange128`.

On all ARM platforms, use the intrinsics with `_acq` and `_rel` suffixes for acquire and release semantics, such as at the beginning and end of a critical section. The ARM intrinsics with an `_nf` ("no fence") suffix don't act as a memory barrier.

The intrinsics with an `_np` ("no prefetch") suffix prevent a possible prefetch operation from being inserted by the compiler.

On Intel platforms that support Hardware Lock Elision (HLE) instructions, the intrinsics with `_HLEAcquire` and `_HLERelease` suffixes include a hint to the processor that can accelerate performance by eliminating a lock write step in hardware. If these intrinsics are called on platforms that don't support HLE, the hint is ignored.

These routines are only available as intrinsics.

Example

In the following example, `_InterlockedCompareExchange` is used for simple low-level thread synchronization. The approach has its limitations as a basis for multithreaded programming; it's presented to illustrate the typical use of the interlocked intrinsics. For best results, use the Windows API. For more information about multithreaded programming, see Writing a Multithreaded Win32 Program.

```
// intrinExample.cpp
// compile with: /EHsc /O2
// Simple example of using _Interlocked* intrinsics to
// do manual synchronization
//
// Add [-DSKIP_LOCKING] to the command line to disable
// the locking. This will cause the threads to execute out
// of sequence.

#define _CRT_RAND_S

#include "windows.h"

#include <iostream>
#include <queue>
#include <intrin.h>

using namespace std;

// --------------------------------------------------------------------

// if defined, will not do any locking on shared data
//#define SKIP_LOCKING

// A common way of locking using _InterlockedCompareExchange.
// Refer to other sources for a discussion of the many issues
// involved. For example, this particular locking scheme performs well
// when lock contention is low, as the while loop overhead is small and
// locks are acquired very quickly, but degrades as many callers want
// the lock and most threads are doing a lot of interlocked spinning.
// There are also no guarantees that a caller will ever acquire the
// lock.
namespace MyInterlockedIntrinsicLock
{
    typedef unsigned LOCK, *PLOCK;

#pragma intrinsic(_InterlockedCompareExchange, _InterlockedExchange)

    enum {LOCK_IS_FREE = 0, LOCK_IS_TAKEN = 1};

    void Lock(PLOCK pl)
    {
#if !defined(SKIP_LOCKING)
        // If *pl == LOCK_IS_FREE, it is set to LOCK_IS_TAKEN
        // atomically, so only 1 caller gets the lock.
        // If *pl == LOCK_IS_TAKEN,
        // the result is LOCK_IS_TAKEN, and the while loop keeps spinning.
        while (_InterlockedCompareExchange((long *)pl,
                                           LOCK_IS_TAKEN, // exchange
                                           LOCK_IS_FREE)  // comparand
               == LOCK_IS_TAKEN)
        {
            // spin!
        }
        // This will also work.
        //while (_InterlockedExchange(pl, LOCK_IS_TAKEN) ==
        //                             LOCK_IS_TAKEN)
        //{
        //    // spin!
        //}

        // At this point, the lock is acquired.
#endif
    }

    void Unlock(PLOCK pl) {
#if !defined(SKIP_LOCKING)
        _InterlockedExchange((long *)pl, LOCK_IS_FREE);
#endif
    }
}

// ------------------------------------------------------------------
// Data shared by threads

queue<int> SharedQueue;
MyInterlockedIntrinsicLock::LOCK SharedLock;
int TicketNumber;

// ------------------------------------------------------------------

DWORD WINAPI
ProducerThread(
    LPVOID unused
    )
{
    unsigned int randValue;
    while (1) {
        // Acquire shared data. Enter critical section.
        MyInterlockedIntrinsicLock::Lock(&SharedLock);

        //cout << ">" << TicketNumber << endl;
        SharedQueue.push(TicketNumber++);

        // Release shared data. Leave critical section.
        // Release shared data. Leave critical section.
        MyInterlockedIntrinsicLock::Unlock(&SharedLock);

        rand_s(&randValue);
        Sleep(randValue % 20);
    }

    return 0;
}

DWORD WINAPI
ConsumerThread(
    LPVOID unused
    )
{
    while (1) {
        // Acquire shared data. Enter critical section
        MyInterlockedIntrinsicLock::Lock(&SharedLock);

        if (!SharedQueue.empty()) {
            int x = SharedQueue.front();
            cout << "<" << x << endl;
            SharedQueue.pop();
        }

        // Release shared data. Leave critical section
        MyInterlockedIntrinsicLock::Unlock(&SharedLock);

        unsigned int randValue;
        rand_s(&randValue);
        Sleep(randValue % 20);
    }
    return 0;
}

int main(
    void
    )
{
    const int timeoutTime = 500;
    int unused1, unused2;
    HANDLE threads[4];

    // The program creates 4 threads:
    // two producer threads adding to the queue
    // and two consumers taking data out and printing it.
    threads[0] = CreateThread(NULL,
                              0,
                              ProducerThread,
                              &unused1,
                              0,
                              (LPDWORD)&unused2);

    threads[1] = CreateThread(NULL,
                              0,
                              ConsumerThread,
                              &unused1,
                              0,
                              (LPDWORD)&unused2);

    threads[2] = CreateThread(NULL,
                              0,
                              ProducerThread,
                              &unused1,
                              0,
                              (LPDWORD)&unused2);

    threads[3] = CreateThread(NULL,
                              0,
                              ConsumerThread,
                                 ConsumerThread,
                                 &unused1,
                                 0,
                                 (LPDWORD)&unused2);

    WaitForMultipleObjects(4, threads, TRUE, timeoutTime);

    return 0;
}
```

```
<0
<1
<2
<3
<4
<5
<6
<7
<8
<9
<10
<11
<12
<13
<14
<15
<16
<17
<18
<19
<20
<21
<22
<23
<24
<25
<26
<27
<28
<29
```

END Microsoft Specific

See also

`_InterlockedCompareExchange128`
`_InterlockedCompareExchangePointer` intrinsic functions
Compiler intrinsics
Keywords
Conflicts with the x86 Compiler

_InterlockedCompareExchange128 intrinsic functions

9/2/2022 • 3 minutes to read • Edit Online

Microsoft Specific

Performs a 128-bit interlocked compare and exchange.

Syntax

```
unsigned char _InterlockedCompareExchange128(
   __int64 volatile * Destination,
   __int64 ExchangeHigh,
   __int64 ExchangeLow,
   __int64 * ComparandResult
);
unsigned char _InterlockedCompareExchange128_acq(
   __int64 volatile * Destination,
   __int64 ExchangeHigh,
   __int64 ExchangeLow,
   __int64 * ComparandResult
);
unsigned char _InterlockedCompareExchange128_nf(
   __int64 volatile * Destination,
   __int64 ExchangeHigh,
   __int64 ExchangeLow,
   __int64 * ComparandResult
);
unsigned char _InterlockedCompareExchange128_np(
   __int64 volatile * Destination,
   __int64 ExchangeHigh,
   __int64 ExchangeLow,
   __int64 * ComparandResult
);
unsigned char _InterlockedCompareExchange128_rel(
   __int64 volatile * Destination,
   __int64 ExchangeHigh,
   __int64 ExchangeLow,
   __int64 * ComparandResult
);
```

Parameters

Destination
[in, out] Pointer to the destination, which is an array of two 64-bit integers considered as a 128-bit field. The destination data must be 16-byte aligned to avoid a general protection fault.

ExchangeHigh
[in] A 64-bit integer that may be exchanged with the high part of the destination.

ExchangeLow
[in] A 64-bit integer that may be exchanged with the low part of the destination.

ComparandResult
[in, out] Pointer to an array of two 64-bit integers (considered as a 128-bit field) to compare with the destination. On output, this array is overwritten with the original value of the destination.

Return value

1 if the 128-bit comparand equals the original value of the destination. `ExchangeHigh` and `ExchangeLow` overwrite the 128-bit destination.

0 if the comparand doesn't equal the original value of the destination. The value of the destination is unchanged, and the value of the comparand is overwritten with the value of the destination.

Requirements

INTRINSIC	ARCHITECTURE
`_InterlockedCompareExchange128`	x64, ARM64
`_InterlockedCompareExchange128_acq`, `_InterlockedCompareExchange128_nf`, `_InterlockedCompareExchange128_rel`	ARM64
`_InterlockedCompareExchange128_np`	x64

Header file <intrin.h>

Remarks

The `_InterlockedCompareExchange128` intrinsic generates the `cmpxchg16b` instruction (with the `lock` prefix) to perform a 128-bit locked compare and exchange. Early versions of AMD 64-bit hardware don't support this instruction. To check for hardware support for the `cmpxchg16b` instruction, call the `__cpuid` intrinsic with `InfoType=0x00000001 (standard function 1)`. Bit 13 of `CPUInfo[2]` (ECX) is 1 if the instruction is supported.

NOTE

The value of `ComparandResult` is always overwritten. After the `lock` instruction, this intrinsic immediately copies the initial value of `Destination` to `ComparandResult`. For this reason, `ComparandResult` and `Destination` should point to separate memory locations to avoid unexpected behavior.

Although you can use `_InterlockedCompareExchange128` for low-level thread synchronization, you don't need to synchronize over 128 bits if you can use smaller synchronization functions (such as the other `_InterlockedCompareExchange` intrinsics) instead. Use `_InterlockedCompareExchange128` if you want atomic access to a 128-bit value in memory.

If you run code that uses the intrinsic on hardware that doesn't support the `cmpxchg16b` instruction, the results are unpredictable.

On ARM platforms, use the intrinsics with `_acq` and `_rel` suffixes for acquire and release semantics, such as at the beginning and end of a critical section. The ARM intrinsics with an `_nf` ("no fence") suffix don't act as a memory barrier.

The intrinsics with an `_np` ("no prefetch") suffix prevent a possible prefetch operation from being inserted by the compiler.

This routine is available only as an intrinsic.

Example

This example uses `_InterlockedCompareExchange128` to replace the high word of an array of two 64-bit integers with the sum of its high and low words and to increment the low word. The access to the `BigInt.Int` array is atomic, but this example uses a single thread and ignores the locking for simplicity.

```
// cmpxchg16b.c
// processor: x64
// compile with: /EHsc /O2
#include <stdio.h>
#include <intrin.h>

typedef struct _LARGE_INTEGER_128 {
    __int64 Int[2];
} LARGE_INTEGER_128, *PLARGE_INTEGER_128;

volatile LARGE_INTEGER_128 BigInt;

// This AtomicOp() function atomically performs:
//   BigInt.Int[1] += BigInt.Int[0]
//   BigInt.Int[0] += 1
void AtomicOp ()
{
    LARGE_INTEGER_128 Comparand;
    Comparand.Int[0] = BigInt.Int[0];
    Comparand.Int[1] = BigInt.Int[1];
    do {
        ; // nothing
    } while (_InterlockedCompareExchange128(BigInt.Int,
                                            Comparand.Int[0] + Comparand.Int[1],
                                            Comparand.Int[0] + 1,
                                            Comparand.Int) == 0);
}

// In a real application, several threads contend for the value
// of BigInt.
// Here we focus on the compare and exchange for simplicity.
int main(void)
{
   BigInt.Int[1] = 23;
   BigInt.Int[0] = 11;
   AtomicOp();
   printf("BigInt.Int[1] = %d, BigInt.Int[0] = %d\n",
      BigInt.Int[1],BigInt.Int[0]);
}
```

```
BigInt.Int[1] = 34, BigInt.Int[0] = 12
```

END Microsoft Specific

See also

Compiler intrinsics
_InterlockedCompareExchange intrinsic functions
Conflicts with the x86 Compiler

_InterlockedCompareExchangePointer intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Performs an atomic operation that stores the `Exchange` address in the `Destination` address if the `Comparand` and the `Destination` address are equal.

Syntax

```
void * _InterlockedCompareExchangePointer (
   void * volatile * Destination,
   void * Exchange,
   void * Comparand
);
void * _InterlockedCompareExchangePointer_acq (
   void * volatile * Destination,
   void * Exchange,
   void * Comparand
);
void * _InterlockedCompareExchangePointer_HLEAcquire (
   void * volatile * Destination,
   void * Exchange,
   void * Comparand
);
void * _InterlockedCompareExchangePointer_HLERelease (
   void * volatile * Destination,
   void * Exchange,
   void * Comparand
);
void * _InterlockedCompareExchangePointer_nf (
   void * volatile * Destination,
   void * Exchange,
   void * Comparand
);
void * _InterlockedCompareExchangePointer_np (
   void * volatile * Destination,
   void * Exchange,
   void * Comparand
);
void * _InterlockedCompareExchangePointer_rel (
   void * volatile * Destination,
   void * Exchange,
   void * Comparand
);
```

Parameters

Destination
[in, out] Pointer to a pointer to the destination value. The sign is ignored.

Exchange
[in] Exchange pointer. The sign is ignored.

Comparand
[in] Pointer to compare to destination. The sign is ignored.

Return value

The return value is the initial value of the destination.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedCompareExchangePointer`	x86, ARM, x64, ARM64	<intrin.h>
`_InterlockedCompareExchangePointer_acq`, `_InterlockedCompareExchangePointer_nf`, `_InterlockedCompareExchangePointer_rel`	ARM, ARM64	<iiintrin.h>
`_InterlockedCompareExchangePointer_HLEAcquire`, `_InterlockedCompareExchangePointer_HLERelease`	x86, x64	<immintrin.h>

Remarks

`_InterlockedCompareExchangePointer` performs an atomic comparison of the `Destination` address with the `Comparand` address. If the `Destination` address is equal to the `Comparand` address, the `Exchange` address is stored in the address specified by `Destination`. Otherwise, no operation is performed.

`_InterlockedCompareExchangePointer` provides compiler intrinsic support for the Win32 Windows SDK InterlockedCompareExchangePointer function.

For an example of how to use `_InterlockedCompareExchangePointer`, see _InterlockedDecrement.

On ARM platforms, use the intrinsics with `_acq` and `_rel` suffixes if you need acquire and release semantics, such as at the beginning and end of a critical section. ARM intrinsics with an `_nf` ("no fence") suffix don't act as a memory barrier.

The intrinsics with an `_np` ("no prefetch") suffix prevent a possible prefetch operation from being inserted by the compiler.

On Intel platforms that support Hardware Lock Elision (HLE) instructions, the intrinsics with `_HLEAcquire` and `_HLERelease` suffixes include a hint to the processor that can accelerate performance by eliminating a lock write step in hardware. If these intrinsics are called on platforms that don't support HLE, the hint is ignored.

These routines are only available as intrinsics.

END Microsoft Specific

See also

Compiler intrinsics
Keywords

`_InterlockedDecrement` intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

Provides compiler intrinsic support for the Win32 Windows SDK InterlockedDecrement function. The `_InterlockedDecrement` intrinsic functions are **Microsoft-specific**.

Syntax

```
long _InterlockedDecrement(
   long volatile * lpAddend
);
long _InterlockedDecrement_acq(
   long volatile * lpAddend
);
long _InterlockedDecrement_rel(
   long volatile * lpAddend
);
long _InterlockedDecrement_nf(
   long volatile * lpAddend
);
short _InterlockedDecrement16(
   short volatile * lpAddend
);
short _InterlockedDecrement16_acq(
   short volatile * lpAddend
);
short _InterlockedDecrement16_rel(
   short volatile * lpAddend
);
short _InterlockedDecrement16_nf(
   short volatile * lpAddend
);
__int64 _InterlockedDecrement64(
   __int64 volatile * lpAddend
);
__int64 _InterlockedDecrement64_acq(
   __int64 volatile * lpAddend
);
__int64 _InterlockedDecrement64_rel(
   __int64 volatile * lpAddend
);
__int64 _InterlockedDecrement64_nf(
   __int64 volatile * lpAddend
);
```

Parameters

lpAddend
[in, out] Volatile pointer to the variable to be decremented.

Return value

The return value is the resulting decremented value.

Requirements

INTRINSIC	ARCHITECTURE
`_InterlockedDecrement`, `_InterlockedDecrement16`	x86, ARM, x64, ARM64
`_InterlockedDecrement64`	ARM, x64, ARM64
`_InterlockedDecrement_acq`, `_InterlockedDecrement_rel`, `_InterlockedDecrement_nf`, `_InterlockedDecrement16_acq`, `_InterlockedDecrement16_rel`, `_InterlockedDecrement16_nf`, `_InterlockedDecrement64_acq`, `_InterlockedDecrement64_rel`, `_InterlockedDecrement64_nf`,	ARM, ARM64

Header file <intrin.h>

Remarks

There are several variations on `_InterlockedDecrement` that vary based on the data types they involve and whether processor-specific acquire or release semantics is used.

While the `_InterlockedDecrement` function operates on 32-bit integer values, `_InterlockedDecrement16` operates on 16-bit integer values and `_InterlockedDecrement64` operates on 64-bit integer values.

On ARM platforms, use the intrinsics with `_acq` and `_rel` suffixes if you need acquire and release semantics, such as at the beginning and end of a critical section. The intrinsics with an `_nf` ("no fence") suffix don't act as a memory barrier.

The variable pointed to by the `lpAddend` parameter must be aligned on a 32-bit boundary; otherwise, this function fails on multiprocessor x86 systems and any non-x86 systems. For more information, see align.

These routines are only available as intrinsics.

Example

```
// compiler_intrinsics_interlocked.cpp
// compile with: /Oi
#define _CRT_RAND_S

#include <cstdlib>
#include <cstdio>
#include <process.h>
#include <windows.h>

// To declare an interlocked function for use as an intrinsic,
// include intrin.h and put the function in a #pragma intrinsic
// statement.
#include <intrin.h>

#pragma intrinsic (_InterlockedIncrement)

// Data to protect with the interlocked functions.
volatile LONG data = 1;

void __cdecl SimpleThread(void* pParam);

const int THREAD_COUNT = 6;

int main() {
   DWORD num;
   HANDLE threads[THREAD_COUNT];
   int args[THREAD_COUNT];
   int i;

   for (i = 0; i < THREAD_COUNT; i++) {
      args[i] = i + 1;
      threads[i] = reinterpret_cast<HANDLE>(_beginthread(SimpleThread, 0,
                           args + i));
      if (threads[i] == reinterpret_cast<HANDLE>(-1))
         // error creating threads
         break;
   }

   WaitForMultipleObjects(i, threads, true, INFINITE);
}

// Code for our simple thread
void __cdecl SimpleThread(void* pParam) {
   int threadNum = *((int*)pParam);
   int counter;
   unsigned int randomValue;
   unsigned int time;
   errno_t err = rand_s(&randomValue);

   if (err == 0) {
      time = (unsigned int) ((double) randomValue / (double) UINT_MAX * 500);
      while (data < 100) {
         if (data < 100) {
            _InterlockedIncrement(&data);
            printf_s("Thread %d: %d\n", threadNum, data);
         }

         Sleep(time);   // wait up to half of a second
      }
   }

   printf_s("Thread %d complete: %d\n", threadNum, data);
}
```

See also

Compiler intrinsics
Keywords
Conflicts with the x86 Compiler

_InterlockedExchange intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates an atomic instruction to set a specified value.

Syntax

```
long _InterlockedExchange(
   long volatile * Target,
   long Value
);
long _InterlockedExchange_acq(
   long volatile * Target,
   long Value
);
long _InterlockedExchange_HLEAcquire(
   long volatile * Target,
   long Value
);
long _InterlockedExchange_HLERelease(
   long volatile * Target,
   long Value
);
long _InterlockedExchange_nf(
   long volatile * Target,
   long Value
);
long _InterlockedExchange_rel(
   long volatile * Target,
   long Value
);
char _InterlockedExchange8(
   char volatile * Target,
   char Value
);
char _InterlockedExchange8_acq(
   char volatile * Target,
   char Value
);
char _InterlockedExchange8_nf(
   char volatile * Target,
   char Value
);
char _InterlockedExchange8_rel(
   char volatile * Target,
   char Value
);
short _InterlockedExchange16(
   short volatile * Target,
   short Value
);
short _InterlockedExchange16_acq(
   short volatile * Target,
   short Value
);
short _InterlockedExchange16_nf(
   short volatile * Target,
   short Value
);
```

```
short _InterlockedExchange16_rel(
   short volatile * Target,
   short Value
);
__int64 _InterlockedExchange64(
   __int64 volatile * Target,
   __int64 Value
);
__int64 _InterlockedExchange64_acq(
   __int64 volatile * Target,
   __int64 Value
);
__int64 _InterlockedExchange64_HLEAcquire(
   __int64 volatile * Target,
   __int64 Value
);
__int64 _InterlockedExchange64_HLERelease(
   __int64 volatile * Target,
   __int64 Value
);
__int64 _InterlockedExchange64_nf(
   __int64 volatile * Target,
   __int64 Value
);
__int64 _InterlockedExchange64_rel(
   __int64 volatile * Target,
   __int64 Value
);
```

Parameters

Target
[in, out] Pointer to the value to be exchanged. The function sets this variable to `Value` and returns its prior value.

Value
[in] Value to be exchanged with the value pointed to by `Target`.

Return value

Returns the initial value pointed to by `Target`.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedExchange`, `_InterlockedExchange8`, `_InterlockedExchange16`	x86, ARM, x64, ARM64	<intrin.h>
`_InterlockedExchange64`	ARM, x64, ARM64	<intrin.h>

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedExchange_acq`, `_InterlockedExchange_nf`, `_InterlockedExchange_rel`, `_InterlockedExchange8_acq`, `_InterlockedExchange8_nf`, `_InterlockedExchange8_rel`, `_InterlockedExchange16_acq`, `_InterlockedExchange16_nf`, `_InterlockedExchange16_rel`, `_InterlockedExchange64_acq`, `_InterlockedExchange64_nf`, `_InterlockedExchange64_rel`,	ARM, ARM64	<intrin.h>
`_InterlockedExchange_HLEAcquire`, `_InterlockedExchange_HLERelease`	x86, x64	<immintrin.h>
`_InterlockedExchange64_HLEAcquire` , `_InterlockedExchange64_HLERelease`	x64	<immintrin.h>

Remarks

`_InterlockedExchange` provides compiler intrinsic support for the Win32 Windows SDK InterlockedExchange function.

There are several variations on `_InterlockedExchange` that vary based on the data types they involve and whether processor-specific acquire or release semantics is used.

While the `_InterlockedExchange` function operates on 32-bit integer values, `_InterlockedExchange8` operates on 8-bit integer values, `_InterlockedExchange16` operates on 16-bit integer values, and `_InterlockedExchange64` operates on 64-bit integer values.

On ARM platforms, use the intrinsics with `_acq` and `_rel` suffixes for acquire and release semantics, such as at the beginning and end of a critical section. The intrinsics with an `_nf` ("no fence") suffix don't act as a memory barrier.

On Intel platforms that support Hardware Lock Elision (HLE) instructions, the intrinsics with `_HLEAcquire` and `_HLERelease` suffixes include a hint to the processor that can accelerate performance by eliminating a lock write step in hardware. If these intrinsics are called on platforms that do not support HLE, the hint is ignored.

These routines are only available as intrinsics.

Example

For a sample of how to use `_InterlockedExchange`, see _InterlockedDecrement.

END Microsoft Specific

See also

Compiler intrinsics
Keywords
Conflicts with the x86 Compiler

_InterlockedExchangeAdd intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Provide compiler intrinsic support for the Win32 Windows SDK _InterlockedExchangeAdd intrinsic functions function.

Syntax

```
long _InterlockedExchangeAdd(
   long volatile * Addend,
   long Value
);
long _InterlockedExchangeAdd_acq(
   long volatile * Addend,
   long Value
);
long _InterlockedExchangeAdd_rel(
   long volatile * Addend,
   long Value
);
long _InterlockedExchangeAdd_nf(
   long volatile * Addend,
   long Value
);
long _InterlockedExchangeAdd_HLEAcquire(
   long volatile * Addend,
   long Value
);
long _InterlockedExchangeAdd_HLERelease(
   long volatile * Addend,
   long Value
);
char _InterlockedExchangeAdd8(
   char volatile * Addend,
   char Value
);
char _InterlockedExchangeAdd8_acq(
   char volatile * Addend,
   char Value
);
char _InterlockedExchangeAdd8_rel(
   char volatile * Addend,
   char Value
);
char _InterlockedExchangeAdd8_nf(
   char volatile * Addend,
   char Value
);
short _InterlockedExchangeAdd16(
   short volatile * Addend,
   short Value
);
short _InterlockedExchangeAdd16_acq(
   short volatile * Addend,
   short Value
);
short _InterlockedExchangeAdd16_rel(
   short volatile * Addend,
   short Value
);
short _InterlockedExchangeAdd16_nf(
   short volatile * Addend,
   short Value
);
__int64 _InterlockedExchangeAdd64(
   __int64 volatile * Addend,
   __int64 Value
);
__int64 _InterlockedExchangeAdd64_acq(
   __int64 volatile * Addend,
   __int64 Value
);
__int64 _InterlockedExchangeAdd64_rel(
   __int64 volatile * Addend,
   __int64 Value
);
__int64 _InterlockedExchangeAdd64_nf(
   __int64 volatile * Addend,
   __int64 Value
);
__int64 _InterlockedExchangeAdd64_HLEAcquire(
   __int64 volatile * Addend,
   __int64 Value
);
__int64 _InterlockedExchangeAdd64_HLERelease(
   __int64 volatile * Addend,
   __int64 Value
);
```

Parameters

Addend
[in, out] The value to be added to; replaced by the result of the addition.

Value
[in] The value to add.

Return value

The return value is the initial value of the variable pointed to by the `Addend` parameter.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedExchangeAdd`, `_InterlockedExchangeAdd8`, `_InterlockedExchangeAdd16`	x86, ARM, x64, ARM64	<intrin.h>
`_InterlockedExchangeAdd64`	ARM, x64, ARM64	<intrin.h>

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedExchangeAdd_acq` , `_InterlockedExchangeAdd_rel` , `_InterlockedExchangeAdd_nf` , `_InterlockedExchangeAdd8_acq` , `_InterlockedExchangeAdd8_rel` , `_InterlockedExchangeAdd8_nf` , `_InterlockedExchangeAdd16_acq` , `_InterlockedExchangeAdd16_rel` , `_InterlockedExchangeAdd16_nf` , `_InterlockedExchangeAdd64_acq` , `_InterlockedExchangeAdd64_rel` , `_InterlockedExchangeAdd64_nf`	ARM, ARM64	<intrin.h>
`_InterlockedExchangeAdd_HLEAcquire` , `_InterlockedExchangeAdd_HLERelease`	x86, x64	<immintrin.h>
`_InterlockedExchangeAdd64_HLEAcquire` , `_InterlockedExchangeAdd64_HLErelease`	x64	<immintrin.h>

Remarks

There are several variations on `_InterlockedExchangeAdd` that vary based on the data types they involve and whether processor-specific acquire or release semantics is used.

While the `_InterlockedExchangeAdd` function operates on 32-bit integer values, `_InterlockedExchangeAdd8` operates on 8-bit integer values, `_InterlockedExchangeAdd16` operates on 16-bit integer values, and `_InterlockedExchangeAdd64` operates on 64-bit integer values.

On ARM platforms, use the intrinsics with `_acq` and `_rel` suffixes if you need acquire and release semantics, such as at the beginning and end of a critical section. The intrinsics with an `_nf` ("no fence") suffix don't act as a memory barrier.

On Intel platforms that support Hardware Lock Elision (HLE) instructions, the intrinsics with `_HLEAcquire` and `_HLERelease` suffixes include a hint to the processor that can accelerate performance by eliminating a lock write step in hardware. If these intrinsics are called on platforms that don't support HLE, the hint is ignored.

These routines are only available as intrinsics. They're intrinsic even when /Oi or #pragma intrinsic is used. It isn't possible to use #pragma function on these intrinsics.

Example

For a sample of how to use `_InterlockedExchangeAdd`, see _InterlockedDecrement.

END Microsoft Specific

See also

Compiler intrinsics
Keywords
Conflicts with the x86 Compiler

_InterlockedExchangePointer intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Performs an atomic exchange operation, which copies the address passed in as the second argument into the first argument, and returns the original address of the first.

Syntax

```
void * _InterlockedExchangePointer(
   void * volatile * Target,
   void * Value
);
void * _InterlockedExchangePointer_acq(
   void * volatile * Target,
   void * Value
);
void * _InterlockedExchangePointer_rel(
   void * volatile * Target,
   void * Value
);
void * _InterlockedExchangePointer_nf(
   void * volatile * Target,
   void * Value
);
void * _InterlockedExchangePointer_HLEAcquire(
   void * volatile * Target,
   void * Value
);
void * _InterlockedExchangePointer_HLERelease(
   void * volatile * Target,
   void * Value
);
```

Parameters

Target
[in, out] Pointer to the pointer to the value to exchange. The function sets the value to *Value* and returns its previous value.

Value
[in] Value to be exchanged with the value pointed to by *Target*.

Return value

The function returns the initial value pointed to by *Target*.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedExchangePointer`	x86, ARM, x64, ARM64	<intrin.h>

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedExchangePointer_acq`, `_InterlockedExchangePointer_rel`, `_InterlockedExchangePointer_nf`	ARM, ARM64	<intrin.h>
`_InterlockedExchangePointer_HLEAcquire`, `_InterlockedExchangePointer_HLERelease`	x64	<immintrin.h>

On the x86 architecture, `_InterlockedExchangePointer` is a macro that calls `_InterlockedExchange`.

Remarks

On a 64-bit system, the parameters are 64 bits and must be aligned on 64-bit boundaries. Otherwise, the function fails. On a 32-bit system, the parameters are 32 bits and must be aligned on 32-bit boundaries. For more information, see align.

On ARM platforms, use the intrinsics with `_acq` and `_rel` suffixes if you need acquire and release semantics, such as at the beginning and end of a critical section. The intrinsic with an `_nf` ("no fence") suffix doesn't act as a memory barrier.

On Intel platforms that support Hardware Lock Elision (HLE) instructions, the intrinsics with `_HLEAcquire` and `_HLERelease` suffixes include a hint to the processor that can accelerate performance by eliminating a lock write step in hardware. If these intrinsics are called on platforms that don't support HLE, the hint is ignored.

These routines are only available as intrinsics.

END Microsoft Specific

See also

Compiler intrinsics
Conflicts with the x86 Compiler

`_InterlockedIncrement` intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

Provide compiler intrinsic support for the Win32 Windows SDK InterlockedIncrement function. The `_InterlockedIncrement` intrinsic functions are **Microsoft-specific**.

Syntax

```
long _InterlockedIncrement(
   long volatile * lpAddend
);
long _InterlockedIncrement_acq(
   long volatile * lpAddend
);
long _InterlockedIncrement_rel(
   long volatile * lpAddend
);
long _InterlockedIncrement_nf(
   long volatile * lpAddend
);
short _InterlockedIncrement16(
   short volatile * lpAddend
);
short _InterlockedIncrement16_acq(
   short volatile * lpAddend
);
short _InterlockedIncrement16_rel(
   short volatile * lpAddend
);
short _InterlockedIncrement16_nf (
   short volatile * lpAddend
);
__int64 _InterlockedIncrement64(
   __int64 volatile * lpAddend
);
__int64 _InterlockedIncrement64_acq(
   __int64 volatile * lpAddend
);
__int64 _InterlockedIncrement64_rel(
   __int64 volatile * lpAddend
);
__int64 _InterlockedIncrement64_nf(
   __int64 volatile * lpAddend
);
```

Parameters

lpAddend
[in, out] Pointer to the variable to be incremented.

Return value

The return value is the resulting incremented value.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedIncrement`, `_InterlockedIncrement16`	x86, ARM, x64, ARM64	<intrin.h>
`_InterlockedIncrement64`	ARM, x64, ARM64	<intrin.h>
`_InterlockedIncrement_acq`, `_InterlockedIncrement_rel`, `_InterlockedIncrement_nf`, `_InterlockedIncrement16_acq`, `_InterlockedIncrement16_rel`, `_InterlockedIncrement16_nf`, `_InterlockedIncrement64_acq`, `_InterlockedIncrement64_rel`, `_InterlockedIncrement64_nf`	ARM, ARM64	<intrin.h>

Remarks

There are several variations on `_InterlockedIncrement` that vary based on the data types they involve and whether processor-specific acquire or release semantics is used.

While the `_InterlockedIncrement` function operates on 32-bit integer values, `_InterlockedIncrement16` operates on 16-bit integer values and `_InterlockedIncrement64` operates on 64-bit integer values.

On ARM platforms, use the intrinsics with `_acq` and `_rel` suffixes if you need acquire and release semantics, such as at the beginning and end of a critical section. The intrinsic with an `_nf` ("no fence") suffix doesn't act as a memory barrier.

The variable pointed to by the `lpAddend` parameter must be aligned on a 32-bit boundary; otherwise, this function fails on multiprocessor x86 systems and any non-x86 systems. For more information, see align.

The Win32 function is declared in `Wdm.h` or `Ntddk.h`.

These routines are only available as intrinsics.

Example

For a sample of how to use `_InterlockedIncrement`, see _InterlockedDecrement.

See also

Compiler intrinsics
Keywords
Conflicts with the x86 Compiler

_InterlockedOr intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Perform an atomic bitwise or operation on a variable shared by multiple threads.

Syntax

```
long _InterlockedOr(
   long volatile * Value,
   long Mask
);
long _InterlockedOr_acq(
   long volatile * Value,
   long Mask
);
long _InterlockedOr_HLEAcquire(
   long volatile * Value,
   long Mask
);
long _InterlockedOr_HLERelease(
   long volatile * Value,
   long Mask
);
long _InterlockedOr_nf(
   long volatile * Value,
   long Mask
);
long _InterlockedOr_np(
   long volatile * Value,
   long Mask
);
long _InterlockedOr_rel(
   long volatile * Value,
   long Mask
);
char _InterlockedOr8(
   char volatile * Value,
   char Mask
);
char _InterlockedOr8_acq(
   char volatile * Value,
   char Mask
);
char _InterlockedOr8_nf(
   char volatile * Value,
   char Mask
);
char _InterlockedOr8_np(
   char volatile * Value,
   char Mask
);
char _InterlockedOr8_rel(
   char volatile * Value,
   char Mask
);
short _InterlockedOr16(
   short volatile * Value,
   short Mask
);
```

```
short _InterlockedOr16_acq(
   short volatile * Value,
   short Mask
);
short _InterlockedOr16_nf(
   short volatile * Value,
   short Mask
);
short _InterlockedOr16_np(
   short volatile * Value,
   short Mask
);
short _InterlockedOr16_rel(
   short volatile * Value,
   short Mask
);
__int64 _InterlockedOr64(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedOr64_acq(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedOr64_HLEAcquire(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedOr64_HLERelease(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedOr64_nf(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedOr64_np(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedOr64_rel(
   __int64 volatile * Value,
   __int64 Mask
);
```

Parameters

Value
[in, out] A pointer to the first operand, to be replaced by the result.

Mask
[in] The second operand.

Return value

The original value pointed to by the first parameter.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedOr`, `_InterlockedOr8`, `_InterlockedOr16`	x86, ARM, x64, ARM64	<intrin.h>

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedOr64`	ARM, x64, ARM64	<intrin.h>
`_InterlockedOr_acq`, `_InterlockedOr_nf`, `_InterlockedOr_rel`, `_InterlockedOr8_acq`, `_InterlockedOr8_nf`, `_InterlockedOr8_rel`, `_InterlockedOr16_acq`, `_InterlockedOr16_nf`, `_InterlockedOr16_rel`, `_InterlockedOr64_acq`, `_InterlockedOr64_nf`, `_InterlockedOr64_rel`	ARM, ARM64	<intrin.h>
`_InterlockedOr_np`, `_InterlockedOr8_np`, `_InterlockedOr16_np`, `_InterlockedOr64_np`	x64	<intrin.h>
`_InterlockedOr_HLEAcquire`, `_InterlockedOr_HLERelease`	x86, x64	<immintrin.h>
`_InterlockedOr64_HLEAcquire`, `_InterlockedOr64_HLERelease`	x64	<immintrin.h>

Remarks

The number in the name of each function specifies the bit size of the arguments.

On ARM platforms, use the intrinsics with `_acq` and `_rel` suffixes if you need acquire and release semantics, such as at the beginning and end of a critical section. The ARM intrinsics with an `_nf` ("no fence") suffix don't act as a memory barrier.

The intrinsics with an `_np` ("no prefetch") suffix prevent a possible prefetch operation from being inserted by the compiler.

On Intel platforms that support Hardware Lock Elision (HLE) instructions, the intrinsics with `_HLEAcquire` and `_HLERelease` suffixes include a hint to the processor that can accelerate performance by eliminating a lock write step in hardware. If these intrinsics are called on platforms that don't support HLE, the hint is ignored.

Example

```
// _InterlockedOr.cpp
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_InterlockedOr)

int main()
{
        long data1 = 0xFF00FF00;
        long data2 = 0x00FFFF00;
        long retval;
        retval = _InterlockedOr(&data1, data2);
        printf_s("0x%x 0x%x 0x%x", data1, data2, retval);
}
```

```
0xffffff00 0xffff00 0xff00ff00
```

END Microsoft Specific

See also

Compiler intrinsics
Conflicts with the x86 Compiler

_InterlockedXor intrinsic functions

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Perform an atomic bitwise exclusive or (XOR) operation on a variable shared by multiple threads.

Syntax

```
long _InterlockedXor(
   long volatile * Value,
   long Mask
);
long _InterlockedXor_acq(
   long volatile * Value,
   long Mask
);
long _InterlockedXor_HLEAcquire(
   long volatile * Value,
   long Mask
);
long _InterlockedXor_HLERelease(
   long volatile * Value,
   long Mask
);
long _InterlockedXor_nf(
   long volatile * Value,
   long Mask
);
long _InterlockedXor_np(
   long volatile * Value,
   long Mask
);
long _InterlockedXor_rel(
   long volatile * Value,
   long Mask
);
char _InterlockedXor8(
   char volatile * Value,
   char Mask
);
char _InterlockedXor8_acq(
   char volatile * Value,
   char Mask
);
char _InterlockedXor8_nf(
   char volatile * Value,
   char Mask
);
char _InterlockedXor8_np(
   char volatile * Value,
   char Mask
);
char _InterlockedXor8_rel(
   char volatile * Value,
   char Mask
);
short _InterlockedXor16(
   short volatile * Value,
   short Mask
);
```

```
short _InterlockedXor16_acq(
   short volatile * Value,
   short Mask
);
short _InterlockedXor16_nf (
   short volatile * Value,
   short Mask
);
short _InterlockedXor16_np (
   short volatile * Value,
   short Mask
);
short _InterlockedXor16_rel(
   short volatile * Value,
   short Mask
);
__int64 _InterlockedXor64(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedXor64_acq(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedXor64_HLEAcquire(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedXor64_HLERelease(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedXor64_nf(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedXor64_np(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedXor64_rel(
   __int64 volatile * Value,
   __int64 Mask
);
```

Parameters

Value
[in, out] A pointer to the first operand, to be replaced by the result.

Mask
[in] The second operand.

Return value

The original value of the first operand.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_InterlockedXor`, `_InterlockedXor8`, `_InterlockedXor16`	x86, ARM, x64, ARM64	<intrin.h>
`_InterlockedXor64`	ARM, x64, ARM64	<intrin.h>
`_InterlockedXor_acq`, `_InterlockedXor_nf`, `_InterlockedXor_rel`, `_InterlockedXor8_acq`, `_InterlockedXor8_nf`, `_InterlockedXor8_rel`, `_InterlockedXor16_acq`, `_InterlockedXor16_nf`, `_InterlockedXor16_rel`, `_InterlockedXor64_acq`, `_InterlockedXor64_nf`, `_InterlockedXor64_rel`,	ARM, ARM64	<intrin.h>
`_InterlockedXor_np`, `_InterlockedXor8_np`, `_InterlockedXor16_np`, `_InterlockedXor64_np`	x64	<intrin.h>
`_InterlockedXor_HLEAcquire`, `_InterlockedXor_HLERelease`	x86, x64	<immintrin.h>
`_InterlockedXor64_HLEAcquire`, `_InterlockedXor64_HLERelease`	x64	<immintrin.h>

Remarks

The number in the name of each function specifies the bit size of the arguments.

On ARM platforms, use the intrinsics with `_acq` and `_rel` suffixes if you need acquire and release semantics, such as at the beginning and end of a critical section. The ARM intrinsics with an `_nf` ("no fence") suffix don't act as a memory barrier.

The intrinsics with an `_np` ("no prefetch") suffix prevent a possible prefetch operation from being inserted by the compiler.

On Intel platforms that support Hardware Lock Elision (HLE) instructions, the intrinsics with `_HLEAcquire` and `_HLERelease` suffixes include a hint to the processor that can accelerate performance by eliminating a lock write step in hardware. If these intrinsics are called on platforms that don't support HLE, the hint is ignored.

Example

```
// _InterLockedXor.cpp
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_InterlockedXor)

int main()
{
        long data1 = 0xFF00FF00;
        long data2 = 0x00FFFF00;
        long retval;
        retval = _InterlockedXor(&data1, data2);
        printf_s("0x%x 0x%x 0x%x", data1, data2, retval);
}
```

```
0xffff0000 0xffff00 0xff00ff00
```

END Microsoft Specific

See also

Compiler intrinsics
Conflicts with the x86 Compiler

__inbyte

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `in` instruction, returning a single byte read from the port specified by `Port`.

Syntax

```
unsigned char __inbyte(
   unsigned short Port
);
```

Parameters

Port
[in] The port to read from.

Return value

The byte read from the specified port.

Requirements

INTRINSIC	ARCHITECTURE
`__inbyte`	x86, x64

Header file <intrin.h>

END Microsoft Specific

Remarks

This routine is only available as an intrinsic.

See also

Compiler intrinsics

__inbytestring

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads data from the specified port using the `rep insb` instruction.

Syntax

```
void __inbytestring(
   unsigned short Port,
   unsigned char* Buffer,
   unsigned long Count
);
```

Parameters

Port
[in] The port to read from.

Buffer
[out] The data read from the port is written here.

Count
[in] The number of bytes of data to read.

Requirements

INTRINSIC	ARCHITECTURE
`__inbytestring`	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__incfsbyte, __incfsword, __incfsdword

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Add one to the value at a memory location specified by an offset relative to the beginning of the `FS` segment.

Syntax

```
void __incfsbyte(
   unsigned long Offset
);
void __incfsword(
   unsigned long Offset
);
void __incfsdword(
   unsigned long Offset
);
```

Parameters

Offset
[in] The offset from the beginning of `FS`.

Requirements

INTRINSIC	ARCHITECTURE
`__incfsbyte`	x86
`__incfsword`	x86
`__incfsdword`	x86

Header file <intrin.h>

Remarks

These intrinsics are only available in kernel mode, and the routines are only available as intrinsics.

END Microsoft Specific

See also

__addfsbyte, __addfsword, __addfsdword
__readfsbyte, __readfsdword, __readfsqword, __readfsword
__writefsbyte, __writefsdword, __writefsqword, __writefsword
Compiler intrinsics

__incgsbyte, __incgsword, __incgsdword, __incgsqword

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Add one to the value at a memory location specified by an offset relative to the beginning of the `GS` segment.

Syntax

```
void __incgsbyte(
   unsigned long Offset
);
void __incgsword(
   unsigned long Offset
);
void __incgsdword(
   unsigned long Offset
);
void __incgsqword(
   unsigned long Offset
);
```

Parameters

Offset
[in] The offset from the beginning of `GS`.

Requirements

INTRINSIC	ARCHITECTURE
`__incgsbyte`	x64
`__incgsword`	x64
`__incgsdword`	x64
`__incgsqword`	x64

Header file <intrin.h>

Remarks

These routines are only available as an intrinsic.

END Microsoft Specific

See also

__addgsbyte, __addgsword, __addgsdword, __addgsqword

__readgsbyte, __readgsdword, __readgsqword, __readgsword
__writegsbyte, __writegsdword, __writegsqword, __writegsword
Compiler intrinsics

__indword

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads one double word of data from the specified port using the `in` instruction.

Syntax

```
unsigned long __indword(
   unsigned short Port
);
```

Parameters

Port
[in] The port to read from.

Return value

The word read from the port.

Requirements

INTRINSIC	ARCHITECTURE
`__indword`	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__indwordstring

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads data from the specified port using the `rep insd` instruction.

Syntax

```
void __indwordstring(
   unsigned short Port,
   unsigned long* Buffer,
   unsigned long Count
);
```

Parameters

Port
[in] The port to read from.

Buffer
[out] The data read from the port is written here.

Count
[in] The number of bytes of data to read.

Requirements

INTRINSIC	ARCHITECTURE
`__indwordstring`	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__int2c

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `int 2c` instruction, which triggers the `2c` interrupt.

Syntax

```
void __int2c(void);
```

Requirements

INTRINSIC	ARCHITECTURE
`__int2c`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics

__invlpg

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the x86 `invlpg` instruction, which invalidates the translation lookaside buffer (TLB) for the page associated with memory pointed to by *Address*.

Syntax

```
void __invlpg(
   void* Address
);
```

Parameters

Address
[in] A 64-bit address.

Requirements

INTRINSIC	ARCHITECTURE
`__invlpg`	x86, x64

Header file <intrin.h>

Remarks

The intrinsic `__invlpg` emits a privileged instruction, and is only available in kernel mode with a privilege level (CPL) of 0.

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__inword

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads data from the specified port using the `in` instruction.

Syntax

```
unsigned short __inword(
   unsigned short Port
);
```

Parameters

Port
[in] The port to read from.

Return value

The word of data read.

Requirements

INTRINSIC	ARCHITECTURE
`__inword`	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__inwordstring

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads data from the specified port using the `rep insw` instruction.

Syntax

```
void __inwordstring(
   unsigned short Port,
   unsigned short* Buffer,
   unsigned long Count
);
```

Parameters

Port
[in] The port to read from.

Buffer
[out] The data read from the port is written here.

Count
[in] The number of words of data to read.

Requirements

INTRINSIC	ARCHITECTURE
`__inwordstring`	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__lidt

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Loads the interrupt descriptor table register (IDTR) with the value in the specified memory location.

Syntax

```
void __lidt(void * Source);
```

Parameters

Source
[in] Pointer to the value to be copied to the IDTR.

Requirements

INTRINSIC	ARCHITECTURE
`__lidt`	x86, x64

Header file <intrin.h>

Remarks

The `__lidt` function is equivalent to the `LIDT` machine instruction, and is available only in kernel mode. For more information, search for the document, "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference," at the Intel Corporation site.

END Microsoft Specific

See also

Compiler intrinsics
__sidt

__ll_lshift

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Shifts the supplied 64-bit value to the left by the specified number of bits.

Syntax

```
unsigned __int64 __ll_lshift(
   unsigned __int64 Mask,
   int nBit
);
```

Parameters

Mask
[in] The 64-bit integer value to shift left.

nBit
[in] The number of bits to shift.

Return value

The mask shifted left by `nBit` bits.

Requirements

INTRINSIC	ARCHITECTURE
`__ll_lshift`	x86, x64

Header file <intrin.h>

Remarks

If you compile your program for the 64-bit architecture, and `nBit` is larger than 63, the number of bits to shift is `nBit` modulo 64. If you compile your program for the 32-bit architecture, and `nBit` is larger than 31, the number of bits to shift is `nBit` modulo 32.

The `ll` in the name indicates that it's an operation on `long long` (`__int64`).

Example

```
// ll_lshift.cpp
// compile with: /EHsc
// processor: x86, x64
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(__ll_lshift)

int main()
{
   unsigned __int64 Mask = 0x100;
   int nBit = 8;
   Mask = __ll_lshift(Mask, nBit);
   cout << hex << Mask << endl;
}
```

Output

```
10000
```

NOTE

There is no unsigned version of the left shift operation. This is because `__ll_lshift` already uses an unsigned input parameter. Unlike the right shift, there is no sign dependence for the left shift, because the least significant bit in the result is always set to zero regardless of the sign of the value shifted.

END Microsoft Specific

See also

__ll_rshift
__ull_rshift
Compiler intrinsics

__ll_rshift

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Shifts a 64-bit value specified by the first parameter to the right, by a number of bits specified by the second parameter.

Syntax

```
__int64 __ll_rshift(
   __int64 Mask,
   int nBit
);
```

Parameters

Mask
[in] The 64-bit integer value to shift right.

nBit
[in] The number of bits to shift, modulo 64 on x64, and modulo 32 on x86.

Return value

The mask shifted by `nBit` bits.

Requirements

INTRINSIC	ARCHITECTURE
`__ll_rshift`	x86, x64

Header file <intrin.h>

Remarks

If the second parameter is greater than 64 on x64 (32 on x86), that number is taken modulo 64 (32 on x86) to determine the number of bits to shift. The `ll` prefix indicates that it's an operation on `long long`, another name for `__int64`, the 64-bit signed integral type.

Example

```
// ll_rshift.cpp
// compile with: /EHsc
// processor: x86, x64
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(__ll_rshift)

int main()
{
   __int64 Mask = - 0x100;
   int nBit = 4;
   cout << hex << Mask << endl;
   cout << " - " << (- Mask) << endl;
   Mask = __ll_rshift(Mask, nBit);
   cout << hex << Mask << endl;
   cout << " - " << (- Mask) << endl;
}
```

Output

```
ffffffffffffff00
- 100
fffffffffffffff0
- 10
```

NOTE

If `_ull_rshift` has been used, the MSB of the right-shifted value would have been zero, so the desired result would not have been obtained in the case of a negative value.

END Microsoft Specific

See also

Compiler intrinsics
__ll_lshift
__ull_rshift

__lzcnt16, __lzcnt, __lzcnt64

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Counts the number of leading zeros in a 16-, 32-, or 64-bit integer.

Syntax

```
unsigned short __lzcnt16(
   unsigned short value
);
unsigned int __lzcnt(
   unsigned int value
);
unsigned __int64 __lzcnt64(
   unsigned __int64 value
);
```

Parameters

value
[in] The 16-, 32-, or 64-bit unsigned integer to scan for leading zeros.

Return value

The number of leading zero bits in the `value` parameter. If `value` is zero, the return value is the size of the input operand (16, 32, or 64). If the most significant bit of `value` is one, the return value is zero.

Requirements

INTRINSIC	ARCHITECTURE
`__lzcnt16`	AMD: Advanced Bit Manipulation (ABM) Intel: Haswell
`__lzcnt`	AMD: Advanced Bit Manipulation (ABM) Intel: Haswell
`__lzcnt64`	AMD: Advanced Bit Manipulation (ABM) in 64-bit mode. Intel: Haswell

Header file <intrin.h>

Remarks

Each of the intrinsics generates the `lzcnt` instruction. The size of the value that the `lzcnt` instruction returns is the same as the size of its argument. In 32-bit mode, there are no 64-bit general-purpose registers, so the 64-bit `lzcnt` isn't supported.

To determine hardware support for the `lzcnt` instruction, call the `__cpuid` intrinsic with `InfoType=0x80000001` and check bit 5 of `CPUInfo[2] (ECX)`. This bit will be 1 if the instruction is supported, and 0 otherwise. If you run code that uses the intrinsic on hardware that doesn't support the `lzcnt` instruction, the results are unpredictable.

On Intel processors that don't support the `lzcnt` instruction, the instruction byte encoding is executed as `bsr` (bit scan reverse). If code portability is a concern, consider use of the `_BitScanReverse` intrinsic instead. For more information, see _BitScanReverse, _BitScanReverse64.

Example

```
// Compile this test with: /EHsc
#include <iostream>
#include <intrin.h>
using namespace std;

int main()
{
  unsigned short us[3] = {0, 0xFF, 0xFFFF};
  unsigned short usr;
  unsigned int   ui[4] = {0, 0xFF, 0xFFFF, 0xFFFFFFFF};
  unsigned int   uir;

  for (int i=0; i<3; i++) {
    usr = __lzcnt16(us[i]);
    cout << "__lzcnt16(0x" << hex << us[i] << ") = " << dec << usr << endl;
  }

  for (int i=0; i<4; i++) {
    uir = __lzcnt(ui[i]);
    cout << "__lzcnt(0x" << hex << ui[i] << ") = " << dec << uir << endl;
  }
}
```

```
__lzcnt16(0x0) = 16
__lzcnt16(0xff) = 8
__lzcnt16(0xffff) = 0
__lzcnt(0x0) = 32
__lzcnt(0xff) = 24
__lzcnt(0xffff) = 16
__lzcnt(0xffffffff) = 0
```

END Microsoft Specific

See also

Compiler intrinsics

_mm_cvtsi64x_ss

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the x64 extended version of the Convert 64-Bit Integer to Scalar Single-Precision Floating-Point Value (`cvtsi2ss`) instruction.

Syntax

```
__m128 _mm_cvtsi64x_ss(
   __m128 a,
   __int64 b
);
```

Parameters

a
[in] An `__m128` structure containing four single-precision floating-point values.

b
[in] A 64-bit integer to be converted into a floating-point value.

Return value

An `__m128` structure whose first floating-point value is the result of the conversion. The other three values are copied unchanged from *a*.

Requirements

INTRINSIC	ARCHITECTURE
`_mm_cvtsi64x_ss`	x64

Header file <intrin.h>

Remarks

The `__m128` structure represents an XMM register, so the intrinsic allows the value *b* from system memory to be moved into an XMM register.

This routine is only available as an intrinsic.

Example

```
// _mm_cvtsi64x_ss.cpp
// processor: x64

#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvtsi64x_ss)

int main()
{
    __m128 a;
    __int64 b = 54;

    a.m128_f32[0] = 0;
    a.m128_f32[1] = 0;
    a.m128_f32[2] = 0;
    a.m128_f32[3] = 0;
    a = _mm_cvtsi64x_ss(a, b);

    printf_s( "%lf %lf %lf %lf\n",
              a.m128_f32[0], a.m128_f32[1],
              a.m128_f32[2], a.m128_f32[3] );
}
```

```
54.000000 0.000000 0.000000 0.000000
```

END Microsoft Specific

See also

__m128
Compiler intrinsics

_mm_cvtss_si64x

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the x64 extended version of the Convert Scalar Single Precision Floating Point Number to 64-bit Integer (`cvtss2si`) instruction.

Syntax

```
__int64 _mm_cvtss_si64x(
   __m128 value
);
```

Parameters

value
[in] An `__m128` structure containing floating point-values.

Return value

A 64-bit integer, the result of the conversion of the first floating-point value to an integer.

Requirements

INTRINSIC	ARCHITECTURE
`_mm_cvtss_si64x`	x64

Header file <intrin.h>

Remarks

The first element of the structure value is converted to an integer and returned. The rounding control bits in MXCSR are used to determine the rounding behavior. The default rounding mode is round to nearest, rounding to the even number if the decimal part is 0.5. Because the `__m128` structure represents an XMM register, the intrinsic takes a value from the XMM register and writes it to system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_cvtss_si64x.cpp
// processor: x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvtss_si64x)

int main()
{
    __m128 a;
    __int64 b = 54;

    // _mm_load_ps requires an aligned buffer.
    __declspec(align(16)) float af[4] =
                           { 101.25, 200.75, 300.5, 400.5 };

    // Load a with the floating point values.
    // The values will be copied to the XMM registers.
    a = _mm_load_ps(af);

    // Extract the first element of a and convert to an integer
    b = _mm_cvtss_si64x(a);

    printf_s("%I64d\n", b);
}
```

```
101
```

END Microsoft Specific

See also

__m128d
Compiler intrinsics

_mm_cvttss_si64x

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Emits the x64 extended version of the Convert with Truncation Single-Precision Floating-Point Number to 64-Bit Integer (`cvttss2si`) instruction.

Syntax

```
__int64 _mm_cvttss_si64x(
   __m128 value
);
```

Parameters

value
[in] An `__m128` structure containing single-precision floating-point values.

Return value

The result of the conversion of the first floating-point value to a 64-bit integer.

Requirements

INTRINSIC	ARCHITECTURE
`_mm_cvttss_si64x`	x64

Header file <intrin.h>

Remarks

The intrinsic differs from `_mm_cvtss_si64x` only in that inexact conversions are truncated toward zero. Because the `__m128` structure represents an XMM register, the instruction generated moves data from an XMM register into system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_cvttss_si64x.cpp
// processor: x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvttss_si64x)

int main()
{
    __m128 a;
    __int64 b = 54;

    // _mm_load_ps requires an aligned buffer.
    __declspec(align(16)) float af[4] = { 101.5, 200.75,
                                          300.5, 400.5 };

    // Load a with the floating point values.
    // The values will be copied to the XMM registers.
    a = _mm_load_ps(af);

    // Extract the first element of a and convert to an integer
    b = _mm_cvttss_si64x(a);

    printf_s("%I64d\n", b);
}
```

```
101
```

END Microsoft Specific

See also

__m128

Compiler intrinsics

_mm_extract_si64, _mm_extracti_si64

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `extrq` instruction to extract specified bits from the low 64 bits of its first argument.

Syntax

```
__m128i _mm_extract_si64(
   __m128i Source,
   __m128i Descriptor
);
__m128i _mm_extracti_si64(
   __m128i Source,
   int Length,
   int Index
);
```

Parameters

Source
[in] A 128-bit field with input data in its lower 64 bits.

Descriptor
[in] A 128-bit field that describes the bit field to extract.

Length
[in] An integer that specifies the length of the field to extract.

Index
[in] An integer that specifies the index of the field to extract

Return value

A 128-bit field with the extracted field in its least significant bits.

Requirements

INTRINSIC	ARCHITECTURE
`_mm_extract_si64`	SSE4a
`_mm_extracti_si64`	SSE4a

Header file <intrin.h>

Remarks

These intrinsics generate the `extrq` instruction to extract bits from *Source*. There are two versions: `_mm_extracti_si64` is the immediate version, and `_mm_extract_si64` is the non-immediate one. Each version extracts from *Source* a bit field defined by its length and the index of its least significant bit. The values of the length and index are taken mod 64, so both -1 and 127 are interpreted as 63. If the sum of the (reduced) index and (reduced) field length is greater than 64, the results are undefined. A value of zero for field length is interpreted as 64. If the field length and bit index are both zero, bits 63:0 of *Source* are extracted. If the field length is zero but the bit index is non-zero, the results are undefined.

In a call to `_mm_extract_si64`, the *Descriptor* contains the index in bits 13:8 and the field length of the data to be extracted in bits 5:0.

If you call `_mm_extracti_si64` with arguments that the compiler can't determine to be integer constants, the compiler generates code to pack those values into an XMM register (*Descriptor*) and to call `_mm_extract_si64`.

To determine hardware support for the `extrq` instruction, call the `__cpuid` intrinsic with `InfoType=0x80000001` and check bit 6 of `CPUInfo[2] (ECX)`. This bit will be 1 if the instruction is supported, and 0 otherwise. If you run code that uses this intrinsic hardware that doesn't support the `extrq` instruction, the results are unpredictable.

Example

```
// Compile this sample with: /EHsc
#include <iostream>
#include <intrin.h>
using namespace std;

union {
    __m128i m;
    unsigned __int64 ui64[2];
} source, descriptor, result1, result2, result3;

int
main()
{
    source.ui64[0] =     0xfedcba9876543210ll;
    descriptor.ui64[0] = 0x0000000000000b1bll;

    result1.m = _mm_extract_si64 (source.m, descriptor.m);
    result2.m = _mm_extracti_si64(source.m, 27, 11);
    result3.ui64[0] = (source.ui64[0] >> 11) & 0x7ffffff;

    cout << hex << "result1 = 0x" << result1.ui64[0] << endl;
    cout << "result2 = 0x" << result2.ui64[0] << endl;
    cout << "result3 = 0x" << result3.ui64[0] << endl;
}
```

```
result1 = 0x30eca86
result2 = 0x30eca86
result3 = 0x30eca86
```

END Microsoft Specific

Portions Copyright 2007 by Advanced Micro Devices, Inc. All rights reserved. Reproduced with permission from Advanced Micro Devices, Inc.

See also

_mm_insert_si64, _mm_inserti_si64
Compiler intrinsics

_mm_insert_si64, _mm_inserti_si64

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `insertq` instruction to insert bits from its second operand into its first operand.

Syntax

```
__m128i _mm_insert_si64(
   __m128i Source1,
   __m128i Source2
);
__m128i _mm_inserti_si64(
   __m128i Source1,
   __m128i Source2
   int Length,
   int Index
);
```

Parameters

Source1
[in] A 128-bit field that has input data in its lower 64 bits, into which a field will be inserted.

Source2
[in] A 128-bit field that has the data to insert in its low bits. For `_mm_insert_si64`, also contains a field descriptor in its high bits.

Length
[in] An integer constant that specifies the length of the field to insert.

Index
[in] An integer constant that specifies the index of the least significant bit of the field into which data will be inserted.

Return value

A 128-bit field, whose lower 64 bits contain the original low 64 bits of *Source1*, with the specified bit field replaced by the low bits of *Source2*. The upper 64 bits of the return value are undefined.

Requirements

INTRINSIC	ARCHITECTURE
`_mm_insert_si64`	SSE4a
`_mm_inserti_si64`	SSE4a

Header file <intrin.h>

Remarks

These intrinsics generate the `insertq` instruction to insert bits from *Source2* into *Source1*. There are two versions: `_mm_inserti_si64`, is the immediate version, and `_mm_insert_si64` is the non-immediate one. Each version extracts a bit field of a given length from Source2 and inserts it into Source1. The extracted bits are the least significant bits of Source2. The field Source1 into which these bits will be inserted is defined by the length and the index of its least significant bit. The values of the length and index are taken mod 64, so both -1 and 127 are interpreted as 63. If the sum of the (reduced) bit index and (reduced) field length is larger than 64, the results are undefined. A value of zero for field length is interpreted as 64. If the field length and bit index are both zero, bits 63:0 of *Source2* are inserted into *Source1*. If the field length is zero, but the bit index is non-zero, the results are undefined.

In a call to _mm_insert_si64, the field length is contained in bits 77:72 of Source2 and the index in bits 69:64.

If you call `_mm_inserti_si64` with arguments that the compiler can't determine to be integer constants, the compiler generates code to pack those values into an XMM register and to call `_mm_insert_si64`.

To determine hardware support for the `insertq` instruction, call the `__cpuid` intrinsic with `InfoType=0x80000001` and check bit 6 of `CPUInfo[2] (ECX)`. This bit is 1 if the instruction is supported, and 0 otherwise. If you run code that uses the intrinsic on hardware that doesn't support the `insertq` instruction, the results are unpredictable.

Example

```
// Compile this sample with: /EHsc
#include <iostream>
#include <intrin.h>
using namespace std;

union {
    __m128i m;
    unsigned __int64 ui64[2];
} source1, source2, source3, result1, result2, result3;

int
main()
{

    __int64 mask;

    source1.ui64[0] = 0xffffffffffffffffll;
    source2.ui64[0] = 0xfedcba9876543210ll;
    source2.ui64[1] = 0xc10;
    source3.ui64[0] = source2.ui64[0];

    result1.m = _mm_insert_si64 (source1.m, source2.m);
    result2.m = _mm_inserti_si64(source1.m, source3.m, 16, 12);
    mask = 0xffff << 12;
    mask = ~mask;
    result3.ui64[0] = (source1.ui64[0] & mask) |
                      ((source2.ui64[0] & 0xffff) << 12);

    cout << hex << "result1 = 0x" << result1.ui64[0] << endl;
    cout << "result2 = 0x" << result2.ui64[0] << endl;
    cout << "result3 = 0x" << result3.ui64[0] << endl;

}
```

```
result1 = 0xfffffffff3210fff
result2 = 0xfffffffff3210fff
result3 = 0xfffffffff3210fff
```

END Microsoft Specific

See also

_mm_extract_si64, _mm_extracti_si64
Compiler intrinsics

_mm_stream_sd

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Writes 64-bit data to a memory location without polluting the caches.

Syntax

```
void _mm_stream_sd(
   double * Dest,
   __m128d Source
);
```

Parameters

Dest
[out] A pointer to the location where the source data will be written.

Source
[in] A 128-bit value containing the `double` value to be written in its bottom 64 bits.

Return value

None.

Requirements

INTRINSIC	ARCHITECTURE
`_mm_stream_sd`	SSE4a

Header file <intrin.h>

Remarks

The intrinsic generates the `movntsd` instruction. To determine hardware support for this instruction, call the `__cpuid` intrinsic with `InfoType=0x80000001` and check bit 6 of `CPUInfo[2] (ECX)`. This bit is 1 if the hardware supports this instruction, and 0 otherwise.

If you run code that uses the `_mm_stream_sd` intrinsic on hardware that doesn't support the `movntsd` instruction, the results are unpredictable.

Example

```
// Compile this sample with: /EHsc
#include <iostream>
#include <intrin.h>
using namespace std;

int main()
{
    __m128d vals;
    double d[2];

    d[0] = -1.;
    d[1] = -2.;
    vals.m128d_f64[0] = 0.;
    vals.m128d_f64[1] = 1.;
    _mm_stream_sd(&d[1], vals);
    cout << "d[0] = " << d[0] << ", d[1] = " << d[1] << endl;
}
```

```
d[0] = -1, d[1] = 1
```

END Microsoft Specific

See also

_mm_stream_ss
_mm_store_sd
_mm_sfence
Compiler intrinsics

_mm_stream_si64x

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the MOVNTI instruction. Writes the data in *Source* to a memory location specified by *Destination*, without polluting the caches.

Syntax

```
void _mm_stream_si64x(
   __int64 * Destination,
   __int64 Source
);
```

Parameters

Destination
[out] A pointer to the location to write the source data to.

Source
[in] The data to write.

Requirements

INTRINSIC	ARCHITECTURE
`_mm_stream_si64x`	x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

Example

```
// _mm_stream_si64x.c
// processor: x64

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_mm_stream_si64x)

int main()
{
    __int64 val = 0xFFFFFFFFFFFFI64;
    __int64 a[10];

    memset(a, 0, sizeof(a));
    _mm_stream_si64x(a+1, val);
    printf_s( "%I64x %I64x %I64x %I64x", a[0], a[1], a[2], a[3]);
}
```

```
0 ffffffffffff 0 0
```

END Microsoft Specific

See also

Compiler intrinsics

_mm_stream_ss

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Writes 32-bit data to a memory location without polluting the caches.

Syntax

```
void _mm_stream_ss(
   float * Destination,
   __m128 Source
);
```

Parameters

Destination
[out] A pointer to the location where the source data is written.

Source
[in] A 128-bit number that contains the `float` value to be written in its bottom 32 bits.

Return value

None.

Requirements

INTRINSIC	ARCHITECTURE
`_mm_stream_ss`	SSE4a

Header file <intrin.h>

Remarks

The intrinsic generates the `movntss` instruction. To determine hardware support for this instruction, call the `__cpuid` intrinsic with `InfoType=0x80000001` and check bit 6 of `CPUInfo[2] (ECX)`. This bit is 1 when the instruction is supported, and 0 otherwise.

If you run code that uses the `_mm_stream_ss` intrinsic on hardware that doesn't support the `movntss` instruction, the results are unpredictable.

Example

```
// Compile this sample with: /EHsc
#include <iostream>
#include <intrin.h>
using namespace std;

int main()
{
    __m128 vals;
    float f[4];

    f[0] = -1.;
    f[1] = -2.;
    f[2] = -3.;
    f[3] = -4.;
    vals.m128_f32[0] = 0.;
    vals.m128_f32[1] = 1.;
    vals.m128_f32[2] = 2.;
    vals.m128_f32[3] = 3.;
    _mm_stream_ss(&f[3], vals);
    cout << "f[0] = " << f[0] << ", f[1] = " << f[1] << endl;
    cout << "f[1] = " << f[1] << ", f[3] = " << f[3] << endl;
}
```

```
f[0] = -1, f[1] = -2
f[2] = -3, f[3] = 3
```

END Microsoft Specific

See also

_mm_stream_sd
_mm_stream_ps
_mm_store_ss
_mm_sfence
Compiler intrinsics

__movsb

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates a Move String (`rep movsb`) instruction.

Syntax

```
void __movsb(
   unsigned char* Destination,
   unsigned const char* Source,
   size_t Count
);
```

Parameters

Destination
[out] A pointer to the destination of the copy.

Source
[in] A pointer to the source of the copy.

Count
[in] The number of bytes to copy.

Requirements

INTRINSIC	ARCHITECTURE
`__movsb`	x86, x64

Header file <intrin.h>

Remarks

The result is that the first `Count` bytes pointed to by `Source` are copied to the `Destination` string.

This routine is only available as an intrinsic.

Example

```
// movsb.cpp
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__movsb)

int main()
{
    unsigned char s1[100];
    unsigned char s2[100] = "A big black dog.";
    __movsb(s1, s2, 100);

    printf_s("%s %s", s1, s2);
}
```

```
A big black dog. A big black dog.
```

END Microsoft Specific

See also

Compiler intrinsics

__movsd

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates a Move String (`rep movsd`) instruction.

Syntax

```
void __movsd(
   unsigned long* Destination,
   unsigned long* Source,
   size_t Count
);
```

Parameters

Destination
[out] The destination of the operation.

Source
[in] The source of the operation.

Count
[in] The number of doublewords to copy.

Requirements

INTRINSIC	ARCHITECTURE
`__movsd`	x86, x64

Header file <intrin.h>

Remarks

The result is that the first *Count* doublewords pointed to by *Source* are copied to the *Destination* string.

This routine is only available as an intrinsic.

Example

```
// movsd.cpp
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__movsd)

int main()
{
    unsigned long a1[10];
    unsigned long a2[10] = {950, 850, 750, 650, 550, 450, 350,
                            250, 150, 50};
    __movsd(a1, a2, 10);

    for (int i = 0; i < 10; i++)
        printf_s("%d ", a1[i]);
    printf_s("\n");
}
```

```
950 850 750 650 550 450 350 250 150 50
```

END Microsoft Specific

See also

Compiler intrinsics

__movsq

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates a repeated Move String (`rep movsq`) instruction.

Syntax

```
void __movsq(
   unsigned long long* Destination,
   unsigned long long const* Source,
   size_t Count
);
```

Parameters

Destination
[out] The destination of the operation.

Source
[in] The source of the operation.

Count
[in] The number of quadwords to copy.

Requirements

INTRINSIC	ARCHITECTURE
`__movsq`	x64

Header file <intrin.h>

Remarks

The result is that the first *Count* quadwords pointed to by *Source* are copied to the *Destination* string.

This routine is only available as an intrinsic.

Example

```
// movsq.cpp
// processor: x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__movsq)

int main()
{
    unsigned __int64 a1[10];
    unsigned __int64 a2[10] = {950, 850, 750, 650, 550, 450, 350, 250,
                               150, 50};
    __movsq(a1, a2, 10);

    for (int i = 0; i < 10; i++)
       printf_s("%d ", a1[i]);
    printf_s("\n");
}
```

```
950 850 750 650 550 450 350 250 150 50
```

END Microsoft Specific

See also

Compiler intrinsics

__movsw

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates a Move String (`rep movsw`) instruction.

Syntax

```
void __movsw(
   unsigned short* Destination,
   unsigned short* Source,
   size_t Count
);
```

Parameters

Destination
[out] The destination of the operation.

Source
[in] The source of the operation.

Count
[in] The number of words to copy.

Requirements

INTRINSIC	ARCHITECTURE
`__movsw`	x86, x64

Header file <intrin.h>

Remarks

The result is that the first *Count* words pointed to by *Source* are copied to the *Destination* string.

This routine is only available as an intrinsic.

Example

```
// movsw.cpp
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__movsw)

int main()
{
    unsigned short s1[10];
    unsigned short s2[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    __movsw(s1, s2, 10);

    for (int i = 0; i < 10; i++)
        printf_s("%d ", s1[i]);
    printf_s("\n");
}
```

```
0 1 2 3 4 5 6 7 8 9
```

END Microsoft Specific

See also

Compiler intrinsics

_mul128

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Multiplies two 64-bit integers passed in as the first two arguments and puts the high 64 bits of the product in the 64-bit integer pointed to by `HighProduct` and returns the low 64 bits of the product.

Syntax

```
__int64 _mul128(
   __int64 Multiplier,
   __int64 Multiplicand,
   __int64 *HighProduct
);
```

Parameters

Multiplier
[in] The first 64-bit integer to multiply.

Multiplicand
[in] The second 64-bit integer to multiply.

HighProduct
[out] The high 64 bits of the product.

Return value

The low 64 bits of the product.

Requirements

INTRINSIC	ARCHITECTURE
`_mul128`	x64

Header file <intrin.h>

Example

```
// mul128.c
// processor: x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_mul128)

int main()
{
    __int64 a = 0x0fffffffffffffffI64;
    __int64 b = 0xf0000000I64;
    __int64 c, d;

    d = _mul128(a, b, &c);

    printf_s("%#I64x * %#I64x = %#I64x%I64x\n", a, b, c, d);
}
```

```
0xfffffffffffffff * 0xf0000000 = 0xeffffffffffffff10000000
```

END Microsoft Specific

See also

Compiler intrinsics

__mulh

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Returns the high 64 bits of the product of two 64-bit signed integers.

Syntax

```
__int64 __mulh(
   __int64 a,
   __int64 b
);
```

Parameters

a
[in] The first number to multiply.

b
[in] The second number to multiply.

Return value

The high 64 bits of the 128-bit result of the multiplication.

Requirements

INTRINSIC	ARCHITECTURE
`__mulh`	x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

Example

```
// mulh.cpp
// processor: x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic (__mulh)

int main()
{
    __int64 a = 0x0fffffffffffffffI64;
    __int64 b = 0xf0000000I64;

    __int64 result = __mulh(a, b); // the high 64 bits
    __int64 result2 = a * b; // the low 64 bits

    printf_s(" %#I64x * %#I64x = %#I64x%I64x\n",
             a, b, result, result2);
}
```

```
0xfffffffffffffff * 0xf0000000 = 0xeffffffffffffff10000000
```

END Microsoft Specific

See also

Compiler intrinsics

__noop

9/2/2022 • 2 minutes to read • Edit Online

The **Microsoft-specific** `__noop` intrinsic specifies that a function should be ignored. The argument list is parsed, but no code is generated for the arguments. The compiler considers the arguments as referenced for the purposes of compiler warning C4100 and similar analysis. The `__noop` intrinsic is intended for use in global debug functions that take a variable number of arguments.

The compiler converts the `__noop` intrinsic to 0 at compile time.

Example

The following code shows how you could use `__noop`.

```
// compiler_intrinsics__noop.cpp
// compile using: cl /EHsc /W4 compiler_intrinsics__noop.cpp
// compile with or without /DDEBUG
#include <stdio.h>

#if DEBUG
   #define PRINT   printf_s
#else
   #define PRINT   __noop
#endif

#define IGNORE(x) { __noop(x); }

int main(int argv, char ** argc)
{
   IGNORE(argv);
   IGNORE(argc);
   PRINT("\nDEBUG is defined\n");
}
```

See also

Compiler intrinsics
Keywords

__nop

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates platform-specific machine code that performs no operation.

Syntax

```
void __nop();
```

Requirements

INTRINSIC	ARCHITECTURE
`__nop`	x86, ARM, x64, ARM64

Header file <intrin.h>

END Microsoft Specific

Remarks

The `__nop` function is equivalent to the `NOP` machine instruction. For more information on x86 and x64, search for the document, "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference," at the Intel Corporation site.

See also

Compiler intrinsics
__noop

__outbyte

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `out` instruction, which sends 1 byte specified by `Data` out the I/O port specified by `Port`.

Syntax

```
void __outbyte(
   unsigned short Port,
   unsigned char Data
);
```

Parameters

Port
[in] The port to send the data to.

Data
[in] The byte to be sent out the specified port.

Requirements

INTRINSIC	ARCHITECTURE
`__outbyte`	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__outbytestring

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `rep outsb` instruction, which sends the first `Count` bytes of data pointed to by `Buffer` to the port specified by `Port`.

Syntax

```
void __outbytestring(
   unsigned short Port,
   unsigned char* Buffer,
   unsigned long Count
);
```

Parameters

Port
[in] The port to send the data to.

Buffer
[in] The data to be sent out the specified port.

Count
[in] The number of bytes of data to be sent.

Requirements

INTRINSIC	ARCHITECTURE
`__outbytestring`	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__outdword

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `out` instruction to send a doubleword *Data* out the port *Port*.

Syntax

```
void __outdword(
   unsigned short Port,
   unsigned long Data
);
```

Parameters

Port
[in] The port to send the data to.

Data
[in] The doubleword to be sent.

Requirements

INTRINSIC	ARCHITECTURE
`__outdword`	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__outdwordstring

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `rep outsd` instruction, which sends `Count` doublewords starting at `Buffer` out the I/O port specified by `Port`.

Syntax

```
void __outdwordstring(
   unsigned short Port,
   unsigned long* Buffer,
   unsigned long Count
);
```

Parameters

Port
[in] The port to send the data to.

Buffer
[in] A pointer to the data to be sent out the specified port.

Count
[in] The number of doublewords to send.

Requirements

INTRINSIC	ARCHITECTURE
`__outdwordstring`	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__outword

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `out` instruction, which sends the word *Data* out the I/O port specified by *Port*.

Syntax

```
void __outword(
   unsigned short Port,
   unsigned short Data
);
```

Parameters

Port
[in] The port to send the data to.

Data
[in] The data to be sent.

Requirements

INTRINSIC	ARCHITECTURE
`__outword`	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__outwordstring

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `rep outsw` instruction, which sends *Count* words starting at *Buffer* out the I/O port specified by *Port*.

Syntax

```
void __outwordstring(
   unsigned short Port,
   unsigned short* Buffer,
   unsigned long Count
);
```

Parameters

Port
[in] The port to send the data to.

Buffer
[in] A pointer to the data to be sent out the specified port.

Count
[in] The number of words to send.

Requirements

INTRINSIC	ARCHITECTURE
`__outwordstring`	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__popcnt16, __popcnt, __popcnt64

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Counts the number of `1` bits (population count) in a 16-, 32-, or 64-bit unsigned integer.

Syntax

```
unsigned short __popcnt16(
   unsigned short value
);
unsigned int __popcnt(
   unsigned int value
);
unsigned __int64 __popcnt64(
   unsigned __int64 value
);
```

Parameters

value
[in] The 16-, 32-, or 64-bit unsigned integer for which we want the population count.

Return value

The number of `1` bits in the *value* parameter.

Requirements

INTRINSIC	ARCHITECTURE
`__popcnt16`	Advanced Bit Manipulation
`__popcnt`	Advanced Bit Manipulation
`__popcnt64`	Advanced Bit Manipulation in 64-bit mode.

Header file <intrin.h>

Remarks

Each of the intrinsics generates the `popcnt` instruction. In 32-bit mode, there are no 64-bit general-purpose registers, so 64-bit `popcnt` isn't supported.

To determine hardware support for the `popcnt` instruction, call the `__cpuid` intrinsic with `InfoType=0x00000001` and check bit 23 of `CPUInfo[2] (ECX)`. This bit is 1 if the instruction is supported, and 0 otherwise. If you run code that uses these intrinsics on hardware that doesn't support the `popcnt` instruction, the results are unpredictable.

Example

```
#include <iostream>
#include <intrin.h>
using namespace std;

int main()
{
  unsigned short us[3] = {0, 0xFF, 0xFFFF};
  unsigned short usr;
  unsigned int   ui[4] = {0, 0xFF, 0xFFFF, 0xFFFFFFFF};
  unsigned int   uir;

  for (int i=0; i<3; i++) {
    usr = __popcnt16(us[i]);
    cout << "__popcnt16(0x" << hex << us[i] << ") = " << dec << usr << endl;
  }

  for (int i=0; i<4; i++) {
    uir = __popcnt(ui[i]);
    cout << "__popcnt(0x" << hex << ui[i] << ") = " << dec << uir << endl;
  }
}
```

```
__popcnt16(0x0) = 0
__popcnt16(0xff) = 8
__popcnt16(0xffff) = 16
__popcnt(0x0) = 0
__popcnt(0xff) = 8
__popcnt(0xffff) = 16
__popcnt(0xffffffff) = 32
```

END Microsoft Specific

See also

Compiler intrinsics

__rdtsc

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `rdtsc` instruction, which returns the processor time stamp. The processor time stamp records the number of clock cycles since the last reset.

Syntax

```
unsigned __int64 __rdtsc();
```

Return value

A 64-bit unsigned integer representing a tick count.

Requirements

INTRINSIC	ARCHITECTURE
`__rdtsc`	x86, x64

Header file <intrin.h>

Remarks

This routine is available only as an intrinsic.

The interpretation of the TSC value in later generations of hardware differs from that in earlier versions of x64. For more information, see the hardware manuals.

Example

```
// rdtsc.cpp
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__rdtsc)

int main()
{
    unsigned __int64 i;
    i = __rdtsc();
    printf_s("%I64d ticks\n", i);
}
```

```
3363423610155519 ticks
```

END Microsoft Specific

See also

Compiler intrinsics

__rdtscp

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `rdtscp` instruction, writes `TSC_AUX[31:0`] to memory, and returns the 64-bit Time Stamp Counter (`TSC)` result.

Syntax

```
unsigned __int64 __rdtscp(
   unsigned int * AUX
);
```

Parameters

AUX
[out] Pointer to a location that will contain the contents of the machine-specific register `TSC_AUX[31:0]`.

Return value

A 64-bit unsigned integer tick count.

Requirements

INTRINSIC	ARCHITECTURE
`__rdtscp`	x86, x64

Header file <intrin.h>

Remarks

The `__rdtscp` intrinsic generates the `rdtscp` instruction. To determine hardware support for this instruction, call the `__cpuid` intrinsic with `InfoType=0x80000001` and check bit 27 of `CPUInfo[3] (EDX)`. This bit is 1 if the instruction is supported, and 0 otherwise. If you run code that uses the intrinsic on hardware that doesn't support the `rdtscp` instruction, the results are unpredictable.

This instruction waits until all previous instructions have executed and all previous loads are globally visible. However, it isn't a serializing instruction. For more information, see the Intel and AMD manuals.

The meaning of the value in `TSC_AUX[31:0]` depends on the operating system.

Example

```
#include <intrin.h>
#include <stdio.h>
int main()
{
    unsigned __int64 i;
    unsigned int ui;
    i = __rdtscp(&ui);
    printf_s("%I64d ticks\n", i);
    printf_s("TSC_AUX was %x\n", ui);
}
```

```
3363423610155519 ticks
TSC_AUX was 0
```

END Microsoft Specific

See also

__rdtsc
Compiler intrinsics

_ReadBarrier

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Limits the compiler optimizations that can reorder memory access operations across the point of the call.

Caution

The `_ReadBarrier`, `_WriteBarrier`, and `_ReadWriteBarrier` compiler intrinsics and the `MemoryBarrier` macro are all deprecated and should not be used. For inter-thread communication, use mechanisms such as atomic_thread_fence and std::atomic<T> that are defined in the C++ Standard Library. For hardware access, use the /volatile:iso compiler option together with the volatile keyword.

Syntax

```
void _ReadBarrier(void);
```

Requirements

INTRINSIC	ARCHITECTURE
`_ReadBarrier`	x86, x64

Header file <intrin.h>

Remarks

The `_ReadBarrier` intrinsic limits the compiler optimizations that can remove or reorder memory access operations across the point of the call.

END Microsoft Specific

See also

Compiler intrinsics
Keywords

__readcr0

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads the CR0 register and returns its value.

Syntax

```
unsigned long __readcr0(void);  /* X86 */
unsigned __int64 __readcr0(void);  /* X64 */
```

Return value

The value in the CR0 register.

Requirements

INTRINSIC	ARCHITECTURE
`__readcr0`	x86, x64

Header file <intrin.h>

Remarks

The intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__readcr2

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads the CR2 register and returns its value.

Syntax

```
unsigned __int64 __readcr2(void);
```

Return value

The value in the CR2 register.

Requirements

INTRINSIC	ARCHITECTURE
`__readcr2`	x86, x64

Header file <intrin.h>

Remarks

The intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__readcr3

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads the CR3 register and returns its value.

Syntax

```
unsigned __int64 __readcr3(void);
```

Return value

The value in the CR3 register.

Requirements

INTRINSIC	ARCHITECTURE
`__readcr3`	x86, x64

Header file <intrin.h>

Remarks

The intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__readcr4

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads the CR4 register and returns its value.

Syntax

```
unsigned __int64 __readcr4(void);
```

Return value

The value in the CR4 register.

Requirements

INTRINSIC	ARCHITECTURE
`__readcr4`	x86, x64

Header file <intrin.h>

Remarks

The intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__readcr8

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads the CR8 register and returns its value.

Syntax

```
unsigned __int64 __readcr8(void);
```

Return value

The value in the CR8 register.

Requirements

INTRINSIC	ARCHITECTURE
`__readcr8`	x64

Header file <intrin.h>

Remarks

The intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__readdr

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads the value of the specified debug register.

Syntax

```
unsigned         __readdr(unsigned int DebugRegister); /* x86 */
unsigned __int64 __readdr(unsigned int DebugRegister); /* x64 */
```

Parameters

DebugRegister
[in] A constant from 0 through 7 that identifies the debug register.

Return value

The value of the specified debug register.

Remarks

These intrinsics are available only in kernel mode, and the routines are available only as intrinsics.

Requirements

INTRINSIC	ARCHITECTURE
`__readdr`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__readeflags

__readeflags

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads the program status and control (EFLAGS) register.

Syntax

```
unsigned     int __readeflags(void); /* x86 */
unsigned __int64 __readeflags(void); /* x64 */
```

Return value

The value of the EFLAGS register. The return value is 32 bits long on a 32-bit platform, and 64 bits long on a 64-bit platform.

Remarks

These routines are available only as intrinsics.

Requirements

INTRINSIC	ARCHITECTURE
`__readeflags`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__writeeflags

__readfsbyte, __readfsdword, __readfsqword, __readfsword

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Read memory from a location specified by an offset relative to the beginning of the FS segment.

Syntax

```
unsigned char __readfsbyte(
   unsigned long Offset
);
unsigned short __readfsword(
   unsigned long Offset
);
unsigned long __readfsdword(
   unsigned long Offset
);
unsigned __int64 __readfsqword(
   unsigned long Offset
);
```

Parameters

Offset
[in] The offset from the beginning of `FS` to read from.

Return value

The memory contents of the byte, word, doubleword, or quadword (as indicated by the name of the function called) at the location `FS:[Offset]`.

Requirements

INTRINSIC	ARCHITECTURE
`__readfsbyte`	x86
`__readfsdword`	x86
`__readfsqword`	x86
`__readfsword`	x86

Header file <intrin.h>

Remarks

These routines are available only as intrinsics.

END Microsoft Specific

See also

__writefsbyte, __writefsdword, __writefsqword, __writefsword
Compiler intrinsics

__readgsbyte, __readgsdword, __readgsqword, __readgsword

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Read memory from a location specified by an offset relative to the beginning of the GS segment.

Syntax

```
unsigned char __readgsbyte(
   unsigned long Offset
);
unsigned short __readgsword(
   unsigned long Offset
);
unsigned long __readgsdword(
   unsigned long Offset
);
unsigned __int64 __readgsqword(
   unsigned long Offset
);
```

Parameters

Offset
[in] The offset from the beginning of `GS` to read from.

Return value

The memory contents of the byte, word, double word, or quadword (as indicated by the name of the function called) at the location `GS:[Offset]`.

Requirements

INTRINSIC	ARCHITECTURE
`__readgsbyte`	x64
`__readgsdword`	x64
`__readgsqword`	x64
`__readgsword`	x64

Header file <intrin.h>

Remarks

These routines are only available as an intrinsic.

END Microsoft Specific

See also

__writegsbyte, __writegsdword, __writegsqword, __writegsword
Compiler intrinsics

__readmsr

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `rdmsr` instruction, which reads the model-specific register specified by `register` and returns its value.

Syntax

```
__int64 __readmsr(
   int register
);
```

Parameters

register
[in] The model-specific register to read.

Return value

The value in the specified register.

Requirements

INTRINSIC	ARCHITECTURE
`__readmsr`	x86, x64

Header file <intrin.h>

Remarks

This function is only available in kernel mode, and the routine is only available as an intrinsic.

For more information, see the AMD documentation.

END Microsoft Specific

See also

Compiler intrinsics

__readpmc

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `rdpmc` instruction, which reads the performance monitoring counter specified by *counter*.

Syntax

```
unsigned __int64 __readpmc(
   unsigned long counter
);
```

Parameters

counter
[in] The performance counter to read.

Return value

The value of the specified performance counter.

Requirements

INTRINSIC	ARCHITECTURE
`__readpmc`	x86, x64

Header file <intrin.h>

Remarks

The intrinsic is available in kernel mode only, and the routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

_ReadWriteBarrier

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Limits the compiler optimizations that can reorder memory accesses across the point of the call.

Caution

The `_ReadBarrier`, `_WriteBarrier`, and `_ReadWriteBarrier` compiler intrinsics and the `MemoryBarrier` macro are all deprecated and should not be used. For inter-thread communication, use mechanisms such as atomic_thread_fence and std::atomic<T>, which are defined in the C++ Standard Library. For hardware access, use the /volatile:iso compiler option together with the volatile keyword.

Syntax

```
void _ReadWriteBarrier(void);
```

Requirements

INTRINSIC	ARCHITECTURE
`_ReadWriteBarrier`	x86, x64

Header file <intrin.h>

Remarks

The `_ReadWriteBarrier` intrinsic limits the compiler optimizations that can remove or reorder memory accesses across the point of the call.

END Microsoft Specific

See also

_ReadBarrier
_WriteBarrier
Compiler intrinsics
Keywords

_ReturnAddress

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

The `_ReturnAddress` intrinsic provides the address of the instruction in the calling function that will be executed after control returns to the caller.

Build the following program and step through it in the debugger. As you step through the program, note the address that is returned from `_ReturnAddress`. Then, immediately after returning from the function where `_ReturnAddress` was used, open the How to: Use the Disassembly Window and note that the address of the next instruction to be executed matches the address returned from `_ReturnAddress`.

Optimizations such as inlining may affect the return address. For example, if the sample program below is compiled with /Ob1, `inline_func` will be inlined into the calling function, `main`. Therefore, the calls to `_ReturnAddress` from `inline_func` and `main` will each produce the same value.

When `_ReturnAddress` is used in a program compiled with /clr, the function containing the `_ReturnAddress` call will be compiled as a native function. When a function compiled as managed calls into the function containing `_ReturnAddress`, `_ReturnAddress` may not behave as expected.

Requirements

Header file <intrin.h>

Example

```
// compiler_intrinsics__ReturnAddress.cpp
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_ReturnAddress)

__declspec(noinline)
void noinline_func(void)
{
   printf("Return address from %s: %p\n", __FUNCTION__, _ReturnAddress());
}

__forceinline
void inline_func(void)
{
   printf("Return address from %s: %p\n", __FUNCTION__, _ReturnAddress());
}

int main(void)
{
   noinline_func();
   inline_func();
   printf("Return address from %s: %p\n", __FUNCTION__, _ReturnAddress());

   return 0;
}
```

END Microsoft Specific

See also

_AddressOfReturnAddress

Compiler intrinsics

Keywords

_rotl8, _rotl16

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Rotate the input values to the left to the most significant bit (MSB) by a specified number of bit positions.

Syntax

```
unsigned char _rotl8(
   unsigned char value,
   unsigned char shift
);
unsigned short _rotl16(
   unsigned short value,
   unsigned char shift
);
```

Parameters

value
[in] The value to rotate.

shift
[in] The number of bits to rotate.

Return value

The rotated value.

Requirements

INTRINSIC	ARCHITECTURE
`_rotl8`	x86, ARM, x64, ARM64
`_rotl16`	x86, ARM, x64, ARM64

Header file <intrin.h>

Remarks

Unlike a left-shift operation, when executing a left rotation, the high-order bits that fall off the high end are moved into the least significant bit positions.

Example

```
// rotl.cpp
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_rotl8, _rotl16)

int main()
{
    unsigned char c = 'A', c1, c2;

    for (int i = 0; i < 8; i++)
    {
       printf_s("Rotating 0x%x left by %d bits gives 0x%x\n", c,
               i, _rotl8(c, i));
    }

    unsigned short s = 0x12;
    int nBit = 10;

    printf_s("Rotating unsigned short 0x%x left by %d bits gives 0x%x\n",
            s, nBit, _rotl16(s, nBit));
}
```

```
Rotating 0x41 left by 0 bits gives 0x41
Rotating 0x41 left by 1 bits gives 0x82
Rotating 0x41 left by 2 bits gives 0x5
Rotating 0x41 left by 3 bits gives 0xa
Rotating 0x41 left by 4 bits gives 0x14
Rotating 0x41 left by 5 bits gives 0x28
Rotating 0x41 left by 6 bits gives 0x50
Rotating 0x41 left by 7 bits gives 0xa0
Rotating unsigned short 0x12 left by 10 bits gives 0x4800
```

END Microsoft Specific

See also

_rotr8, _rotr16

Compiler intrinsics

_rotr8, _rotr16

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Rotate the input values to the right to the least significant bit (LSB) by a specified number of bit positions.

Syntax

```
unsigned char _rotr8(
   unsigned char value,
   unsigned char shift
);
unsigned short _rotr16(
   unsigned short value,
   unsigned char shift
);
```

Parameters

value
[in] The value to rotate.

shift
[in] The number of bits to rotate.

Return value

The rotated value.

Requirements

INTRINSIC	ARCHITECTURE
`_rotr8`	x86, ARM, x64, ARM64
`_rotr16`	x86, ARM, x64, ARM64

Header file <intrin.h>

Remarks

Unlike a right-shift operation, when executing a right rotation, the low-order bits that fall off the low end are moved into the high-order bit positions.

Example

```
// rotr.cpp
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_rotr8, _rotr16)

int main()
{
    unsigned char c = 'A', c1, c2;

    for (int i = 0; i < 8; i++)
    {
       printf_s("Rotating 0x%x right by %d bits gives 0x%x\n", c,
                i, _rotr8(c, i));
    }

    unsigned short s = 0x12;
    int nBit = 10;

    printf_s("Rotating unsigned short 0x%x right by %d bits "
             "gives 0x%x\n",
            s, nBit, _rotr16(s, nBit));
}
```

```
Rotating 0x41 right by 0 bits gives 0x41
Rotating 0x41 right by 1 bits gives 0xa0
Rotating 0x41 right by 2 bits gives 0x50
Rotating 0x41 right by 3 bits gives 0x28
Rotating 0x41 right by 4 bits gives 0x14
Rotating 0x41 right by 5 bits gives 0xa
Rotating 0x41 right by 6 bits gives 0x5
Rotating 0x41 right by 7 bits gives 0x82
Rotating unsigned short 0x12 right by 10 bits gives 0x480
```

END Microsoft Specific

See also

_rotl8, _rotl16
Compiler intrinsics

__segmentlimit

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the `lsl` (Load Segment Limit) instruction.

Syntax

```
unsigned long __segmentlimit(
   unsigned long a
);
```

Parameters

a
[in] A constant that specifies the segment selector.

Return value

The segment limit of the segment selector specified by *a*, if the selector is valid and visible at the current permission level.

Requirements

INTRINSIC	ARCHITECTURE
`__segmentlimit`	x86, x64

Header file <intrin.h>

Remarks

If the segment limit can't be retrieved, this instruction fails. On failure, this instruction clears the ZF flag and the return value is undefined.

This routine is only available as an intrinsic.

Example

```
#include <stdio.h>

#ifdef _M_IX86
typedef unsigned int READETYPE;
#else
typedef unsigned __int64 READETYPE;
#endif

#define EFLAGS_ZF      0x00000040
#define KGDT_R3_DATA    0x0020
#define RPL_MASK        0x3

extern "C"
{
unsigned long __segmentlimit (unsigned long);
READETYPE __readeflags();
}

#pragma intrinsic(__readeflags)
#pragma intrinsic(__segmentlimit)

int main(void)
{
   const unsigned long initsl = 0xbaadbabe;
   READETYPE eflags = 0;
   unsigned long sl = initsl;

   printf("Before: segment limit =0x%x eflags =0x%x\n", sl, eflags);
   sl = __segmentlimit(KGDT_R3_DATA + RPL_MASK);

   eflags = __readeflags();

   printf("After: segment limit =0x%x eflags =0x%x eflags.zf = %s\n", sl, eflags, (eflags & EFLAGS_ZF) ?
"set" : "clear");

   // If ZF is set, the call to lsl succeeded; if ZF is clear, the call failed.
   printf("%s\n", eflags & EFLAGS_ZF ? "Success!": "Fail!");

   // You can verify the value of sl to make sure that the instruction wrote to it
   printf("sl was %s\n", (sl == initsl) ? "unchanged" : "changed");

   return 0;
}
```

```
Before: segment limit =0xbaadbabe eflags =0x0
After: segment limit =0xffffffff eflags =0x256 eflags.zf = set
Success!
sl was changed
```

END Microsoft Specific

See also

Compiler intrinsics

__shiftleft128

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Shifts a 128-bit quantity, represented as two 64-bit quantities `LowPart` and `HighPart`, to the left by a number of bits specified by `Shift` and returns the high 64 bits of the result.

Syntax

```
unsigned __int64 __shiftleft128(
   unsigned __int64 LowPart,
   unsigned __int64 HighPart,
   unsigned char Shift
);
```

Parameters

LowPart
[in] The low 64 bits of the 128-bit quantity to shift.

HighPart
[in] The high 64 bits of the 128-bit quantity to shift.

Shift
[in] The number of bits to shift.

Return value

The high 64 bits of the result.

Requirements

INTRINSIC	ARCHITECTURE
`__shiftleft128`	x64

Header file <intrin.h>

Remarks

The *Shift* value is always modulo 64 so that, for example, if you call `__shiftleft128(1, 0, 64)`, the function will shift the low part `0` bits left and return a high part of `0` and not `1` as might otherwise be expected.

Example

```
// shiftleft128.c
// processor: IPF, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic (__shiftleft128, __shiftright128)

int main()
{
    unsigned __int64 i = 0x1I64;
    unsigned __int64 j = 0x10I64;
    unsigned __int64 ResultLowPart;
    unsigned __int64 ResultHighPart;

    ResultLowPart = i << 1;
    ResultHighPart = __shiftleft128(i, j, 1);

    // concatenate the low and high parts padded with 0's
    // to display correct hexadecimal 128 bit values
    printf_s("0x%02I64x%016I64x << 1 = 0x%02I64x%016I64x\n",
             j, i, ResultHighPart, ResultLowPart);

    ResultHighPart = j >> 1;
    ResultLowPart = __shiftright128(i, j, 1);

    printf_s("0x%02I64x%016I64x >> 1 = 0x%02I64x%016I64x\n",
             j, i, ResultHighPart, ResultLowPart);
}
```

```
0x100000000000000001 << 1 = 0x200000000000000002
0x100000000000000001 >> 1 = 0x080000000000000000
```

END Microsoft Specific

See also

__shiftright128
Compiler intrinsics

__shiftright128

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Shifts a 128-bit quantity, represented as two 64-bit quantities `LowPart` and `HighPart`, to the right by a number of bits specified by `Shift` and returns the low 64 bits of the result.

Syntax

```
unsigned __int64 __shiftright128(
   unsigned __int64 LowPart,
   unsigned __int64 HighPart,
   unsigned char Shift
);
```

Parameters

LowPart
[in] The low 64 bits of the 128-bit quantity to shift.

HighPart
[in] The high 64 bits of the 128-bit quantity to shift.

Shift
[in] The number of bits to shift.

Return value

The low 64 bits of the result.

Requirements

INTRINSIC	ARCHITECTURE
`__shiftright128`	x64

Header file <intrin.h>

Remarks

The `Shift` value is always modulo 64 so that, for example, if you call `__shiftright128(0, 1, 64)`, the function will shift the high part `0` bits right and return a low part of `0` and not `1` as might otherwise be expected.

Example

For an example, see __shiftleft128.

END Microsoft Specific

See also

__shiftleft128

Compiler intrinsics

__sidt

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Stores the value of the interrupt descriptor table register (IDTR) in the specified memory location.

Syntax

```
void __sidt(void * Destination);
```

Parameters

Destination
[in] A pointer to the memory location where the IDTR is stored.

Requirements

INTRINSIC	ARCHITECTURE
`__sidt`	x86, x64

Header file <intrin.h>

Remarks

The `__sidt` function is equivalent to the `SIDT` machine instruction. For more information, search for the document, "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference," at the Intel Corporation site.

END Microsoft Specific

See also

Compiler intrinsics
__lidt

__stosb

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates a store string instruction (`rep stosb`).

Syntax

```
void __stosb(
   unsigned char* Destination,
   unsigned char Data,
   size_t Count
);
```

Parameters

Destination
[out] The destination of the operation.

Data
[in] The data to store.

Count
[in] The length of the block of bytes to write.

Requirements

INTRINSIC	ARCHITECTURE
`__stosb`	x86, x64

Header file <intrin.h>

Remarks

The result is that the character *Data* is written into a block of *Count* bytes in the *Destination* string.

This routine is only available as an intrinsic.

Example

```
// stosb.c
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__stosb)

int main()
{
    unsigned char c = 0x40; /* '@' character */
    unsigned char s[] = "*********************************";

    printf_s("%s\n", s);
    __stosb((unsigned char*)s+1, c, 6);
    printf_s("%s\n", s);

}
```

```
*********************************
*@@@@@@**************************
```

END Microsoft Specific

See also

Compiler intrinsics

__stosd

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates a store string instruction (`rep stosd`).

Syntax

```
void __stosd(
   unsigned long* Destination,
   unsigned long Data,
   size_t Count
);
```

Parameters

Destination
[out] The destination of the operation.

Data
[in] The data to store.

Count
[in] The length of the block of doublewords to write.

Requirements

INTRINSIC	ARCHITECTURE
`__stosd`	x86, x64

Header file <intrin.h>

Remarks

The result is that the doubleword *Data* is written into a block of *Count* doublewords at the memory location pointed to by *Destination*.

This routine is only available as an intrinsic.

Example

```
// stosd.c
// processor: x86, x64

#include <stdio.h>
#include <memory.h>
#include <intrin.h>

#pragma intrinsic(__stosd)

int main()
{
    unsigned long val = 99999;
    unsigned long a[10];

    memset(a, 0, sizeof(a));
    __stosd(a+1, val, 2);

printf_s( "%u %u %u %u",
              a[0], a[1], a[2], a[3]);
}
```

```
0 99999 99999 0
```

END Microsoft Specific

See also

Compiler intrinsics

__stosq

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates a store string instruction (`rep stosq`).

Syntax

```
void __stosb(
   unsigned __int64* Destination,
   unsigned __int64 Data,
   size_t Count
);
```

Parameters

Destination
[out] The destination of the operation.

Data
[in] The data to store.

Count
[in] The length of the block of quadwords to write.

Requirements

INTRINSIC	ARCHITECTURE
`__stosq`	AMD64

Header file <intrin.h>

Remarks

The result is that the quadword *Data* is written into a block of *Count* quadwords in the *Destination* string.

This routine is only available as an intrinsic.

Example

```
// stosq.c
// processor: x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__stosq)

int main()
{
   unsigned __int64 val = 0xFFFFFFFFFFFFI64;
   unsigned __int64 a[10];
   memset(a, 0, sizeof(a));
   __stosq(a+1, val, 2);
   printf("%I64x %I64x %I64x %I64x", a[0], a[1], a[2], a[3]);
}
```

```
0 ffffffffffff ffffffffffff 0
```

END Microsoft Specific

See also

Compiler intrinsics

__stosw

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates a store string instruction (`rep stosw`).

Syntax

```
void __stosw(
   unsigned short* Destination,
   unsigned short Data,
   size_t Count
);
```

Parameters

Destination
[out] The destination of the operation.

Data
[in] The data to store.

Count
[in] The length of the block of words to write.

Requirements

INTRINSIC	ARCHITECTURE
`__stosw`	x86, x64

Header file <intrin.h>

Remarks

The result is that the word *Data* is written into a block of *Count* words in the *Destination* string.

This routine is only available as an intrinsic.

Example

```
// stosw.c
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__stosw)

int main()
{
    unsigned short val = 128;
    unsigned short a[100];
    memset(a, 0, sizeof(a));
    __stosw(a+10, val, 2);
    printf_s("%u %u %u %u", a[9], a[10], a[11], a[12]);
}
```

```
0 128 128 0
```

END Microsoft Specific

See also

Compiler intrinsics

__svm_clgi

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Clears the global interrupt flag.

Syntax

```
void __svm_clgi( void );
```

Remarks

The `__svm_clgi` function is equivalent to the `CLGI` machine instruction. The global interrupt flag determines whether the microprocessor ignores, postpones, or handles interrupts, due to events such as an I/O completion, a hardware temperature alert, or a debug exception.

This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for "AMD64 Architecture Programmer's Manual Volume 2: System Programming," at the AMD corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__svm_clgi`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__svm_stgi

__svm_invlpga

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Invalidates the address mapping entry in the computer's translation look-aside buffer. Parameters specify the virtual address and address space identifier of the page to invalidate.

Syntax

```
void __svm_invlpga(void *Vaddr, int as_id);
```

Parameters

Vaddr
[in] The virtual address of the page to invalidate.

as_id
[in] The address space identifier (ASID) of the page to invalidate.

Remarks

The `__svm_invlpga` function is equivalent to the `INVLPGA` machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," document number 24593, revision 3.11, at the AMD corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__svm_invlpga`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics

__svm_skinit

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Initiates the loading of verifiably secure software, such as a virtual machine monitor.

Syntax

```
void __svm_skinit(
   int block_address
);
```

Parameters

block_address
The 32-bit physical address of a 64K byte Secure Loader Block (SLB).

Remarks

The `__svm_skinit` function is equivalent to the `SKINIT` machine instruction. This function is part of a security system that uses the processor and a Trusted Platform Module (TPM), to verify and load trusted software, called a *security kernel* (SK). A virtual machine monitor is an example of a security kernel. The security system verifies program components loaded during the initialization process. It protects components from tampering by interrupts, device access, or another program if the computer is a multiprocessor.

The *block_address* parameter specifies the physical address of a 64K block of memory called the *Secure Loader Block* (SLB). The SLB contains a program called the *secure loader*. It establishes the operating environment for the computer, and then loads the security kernel.

This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for "AMD64 Architecture Programmer's Manual Volume 2: System Programming," at the AMD corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__svm_skinit`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics

__svm_stgi

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Sets the global interrupt flag.

Syntax

```
void __svm_stgi(void);
```

Remarks

The `__svm_stgi` function is equivalent to the `STGI` machine instruction. The global interrupt flag determines whether the microprocessor ignores, postpones, or handles interrupts, due to events such as an I/O completion, a hardware temperature alert, or a debug exception.

This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for "AMD64 Architecture Programmer's Manual Volume 2: System Programming," at the AMD corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__svm_stgi`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__svm_clgi

__svm_vmload

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Loads a subset of processor state from the specified virtual machine control block (VMCB).

Syntax

```
void __svm_vmload(
   size_t VmcbPhysicalAddress
);
```

Parameters

VmcbPhysicalAddress
[in] The physical address of the VMCB.

Remarks

The `__svm_vmload` function is equivalent to the `VMLOAD` machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," document number 24593, revision 3.11, at the AMD corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__svm_vmload`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__svm_vmrun
__svm_vmsave

__svm_vmrun

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Starts execution of the virtual machine guest code that corresponds to the specified virtual machine control block (VMCB).

Syntax

```
void __svm_vmrun(
   size_t VmcbPhysicalAddress
);
```

Parameters

VmcbPhysicalAddress
[in] The physical address of the VMCB.

Remarks

The `__svm_vmrun` function uses a minimal amount of information in the VMCB to begin executing the virtual machine guest code. Use the __svm_vmsave or __svm_vmload function if you require more information to handle a complex interrupt, or to switch to another guest.

The `__svm_vmrun` function is equivalent to the `VMRUN` machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," document number 24593, revision 3.11 or later, at the AMD corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__svm_vmrun`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__svm_vmsave
__svm_vmload

__svm_vmsave

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Stores a subset of processor state in the specified virtual machine control block (VMCB).

Syntax

```
void __svm_vmsave(
   size_t VmcbPhysicalAddress
);
```

Parameters

VmcbPhysicalAddress
[in] The physical address of the VMCB.

Remarks

The `__svm_vmsave` function is equivalent to the `VMSAVE` machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," document number 24593, revision 3.11 or later, at the AMD Corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__svm_vmsave`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__svm_vmrun
__svm_vmload

__ud2

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates an undefined instruction.

Syntax

```
void __ud2();
```

Remarks

The processor raises an invalid opcode exception if you execute an undefined instruction.

The `__ud2` function is equivalent to the `UD2` machine instruction. For more information, search for the document, "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference," at the Intel Corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__ud2`	x86, x64

Header file <intrin.h>

END Microsoft Specific

Example

The following example executes an undefined instruction, which raises an exception. The exception handler then changes the return code from zero to one.

```
// __ud2_intrinsic.cpp
#include <stdio.h>
#include <intrin.h>
#include <excpt.h>
// compile with /EHa

int main() {

// Initialize the return code to 0.
int ret = 0;

// Attempt to execute an undefined instruction.
  printf("Before __ud2(). Return code = %d.\n", ret);
  __try {
  __ud2();
  }

// Catch any exceptions and set the return code to 1.
  __except(EXCEPTION_EXECUTE_HANDLER){
  printf("  In the exception handler.\n");
  ret = 1;
  }

// Report the value of the return code.
  printf("After __ud2().  Return code = %d.\n", ret);
  return ret;
}
```

```
Before __ud2(). Return code = 0.
  In the exception handler.
After __ud2().  Return code = 1.
```

See also

Compiler intrinsics

_udiv128

9/2/2022 • 2 minutes to read • Edit Online

The `_udiv128` intrinsic divides a 128-bit unsigned integer by a 64-bit unsigned integer. The return value holds the quotient, and the intrinsic returns the remainder through a pointer parameter. `_udiv128` is **Microsoft-specific**.

Syntax

```
unsigned __int64 _udiv128(
   unsigned __int64 highDividend,
   unsigned __int64 lowDividend,
   unsigned __int64 divisor,
   unsigned __int64 *remainder
);
```

Parameters

highDividend
[in] The high 64 bits of the dividend.

lowDividend
[in] The low 64 bits of the dividend.

divisor
[in] The 64-bit integer to divide by.

remainder
[out] The 64-bit integer bits of the remainder.

Return value

The 64 bits of the quotient.

Remarks

Pass the upper 64 bits of the 128-bit dividend in *highDividend*, and the lower 64 bits in *lowDividend*. The intrinsic divides this value by *divisor*. It stores the remainder in the 64-bit unsigned integer pointed to by *remainder*, and returns the 64 bits of the quotient.

The `_udiv128` intrinsic is available starting in Visual Studio 2019 RTM.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_udiv128`	x64	<immintrin.h>

See also

_div128

_udiv64

9/2/2022 • 2 minutes to read • Edit Online

The `_udiv64` intrinsic divides a 64-bit unsigned integer by a 32-bit unsigned integer. The return value holds the quotient, and the intrinsic returns the remainder through a pointer parameter. `_udiv64` is **Microsoft-specific**.

Syntax

```
unsigned int _udiv64(
   unsigned __int64 dividend,
   unsigned int divisor,
   unsigned int* remainder
);
```

Parameters

dividend
[in] The 64-bit unsigned integer to divide.

divisor
[in] The 32-bit unsigned integer to divide by.

remainder
[out] The 32-bit unsigned integer remainder.

Return value

The 32 bits of the quotient.

Remarks

The `_udiv64` intrinsic divides *dividend* by *divisor*. It stores the remainder in the 32-bit unsigned integer pointed to by *remainder*, and returns the 32 bits of the quotient.

The `_udiv64` intrinsic is available starting in Visual Studio 2019 RTM.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_udiv64`	x86, x64	<immintrin.h>

See also

_div64
Compiler intrinsics

__ull_rshift

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

on x64, shifts a 64-bit value specified by the first parameter to the right by a number of bits specified by the second parameter.

Syntax

```
unsigned __int64 __ull_rshift(
   unsigned __int64 mask,
   int nBit
);
```

Parameters

mask
[in] The 64-bit integer value to shift right.

nBit
[in] The number of bits to shift, modulo 32 on x86, and modulo 64 on x64.

Return value

The mask shifted by `nBit` bits.

Requirements

INTRINSIC	ARCHITECTURE
`__ull_rshift`	x86, x64

Header file <intrin.h>

Remarks

If the second parameter is greater than 31 on x86 (63 on x64), that number is taken modulo 32 (64 on x64) to determine the number of bits to shift. The `ull` in the name indicates `unsigned long long (unsigned __int64)`.

Example

```
// ull_rshift.cpp
// compile with: /EHsc
// processor: x86, x64
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(__ull_rshift)

int main()
{
   unsigned __int64 mask = 0x100;
   int nBit = 8;
   mask = __ull_rshift(mask, nBit);
   cout << hex << mask << endl;
}
```

```
1
```

END Microsoft Specific

See also

__ll_lshift
__ll_rshift
Compiler intrinsics

_umul128

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Multiplies two 64-bit unsigned integers passed in as the first two arguments and puts the high 64 bits of the product in the 64-bit unsigned integer pointed to by `HighProduct` and returns the low 64 bits of the product.

Syntax

```
unsigned __int64 _umul128(
   unsigned __int64 Multiplier,
   unsigned __int64 Multiplicand,
   unsigned __int64 *HighProduct
);
```

Parameters

Multiplier
[in] The first 64-bit integer to multiply.

Multiplicand
[in] The second 64-bit integer to multiply.

HighProduct
[out] The high 64 bits of the product.

Return value

The low 64 bits of the product.

Requirements

INTRINSIC	ARCHITECTURE	HEADER
`_umul128`	x64	<intrin.h>

Example

```
// umul128.c
// processor: x64

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_umul128)

int main()
{
    unsigned __int64 a = 0x0fffffffffffffffI64;
    unsigned __int64 b = 0xf0000000I64;
    unsigned __int64 c, d;

    d = _umul128(a, b, &c);

    printf_s("%#I64x * %#I64x = %#I64x%I64x\n", a, b, c, d);
}
```

```
0xfffffffffffffff * 0xf0000000 = 0xeffffffffffffff10000000
```

END Microsoft Specific

See also

Compiler intrinsics

__umulh

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Return the high 64 bits of the product of two 64-bit unsigned integers.

Syntax

```
unsigned __int64 __umulh(
   unsigned __int64 a,
   unsigned __int64 b
);
```

Parameters

a
[in] The first number to multiply.

b
[in] The second number to multiply.

Return value

The high 64 bits of the 128-bit result of the multiplication.

Requirements

INTRINSIC	ARCHITECTURE
`__umulh`	x64

Header file <intrin.h>

Remarks

These routines are only available as intrinsics.

Example

```
// umulh.cpp
// processor: X64
#include <cstdio>
#include <intrin.h>

int main()
{
    unsigned __int64 i = 0x10;
    unsigned __int64 j = 0xFEDCBA9876543210;
    unsigned __int64 k = i * j; // k has the low 64 bits
                                // of the product.
    unsigned __int64 result;
    result = __umulh(i, j); // result has the high 64 bits
                            // of the product.
    printf_s("0x%I64x * 0x%I64x = 0x%I64x%I64x \n", i, j, result, k);
    return 0;
}
```

```
0x10 * 0xfedcba9876543210 = 0xfedcba98765432100
```

END Microsoft Specific

See also

Compiler intrinsics

__vmx_off

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Deactivates virtual machine extensions (VMX) operation in the processor.

Syntax

```
void __vmx_off();
```

Remarks

The `__vmx_off` function is equivalent to the `VMXOFF` machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__vmx_off`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics

__vmx_on

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Activates virtual machine extensions (VMX) operation in the processor.

Syntax

```
unsigned char __vmx_on(
   unsigned __int64 *VmxonRegionPhysicalAddress
);
```

Parameters

`VmxonRegionPhysicalAddress`
[in] A pointer to a 64-bit, 4KB-aligned physical address that points to a VMXON region.

Return value

VALUE	MEANING
0	The operation succeeded.
1	The operation failed with extended status available in the `VM-instruction error field` of the current VMCS.
2	The operation failed without status available.

Remarks

The `__vmx_on` function corresponds to the `VMXON` machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, see "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3" in the Intel 64 and IA-32 Architecture Developer Manuals.

Requirements

INTRINSIC	ARCHITECTURE
`__vmx_on`	x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics

__vmx_vmclear

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Initializes the specified virtual machine control structure (VMCS) and sets its launch state to `Clear`.

Syntax

```
unsigned char __vmx_vmclear(
   unsigned __int64 *VmcsPhysicalAddress
);
```

Parameters

VmcsPhysicalAddress
[in] A pointer to a 64-bit memory location that contains the physical address of the VMCS to clear.

Return value

VALUE	MEANING
0	The operation succeeded.
1	The operation failed with extended status available in the `VM-instruction error field` of the current VMCS.
2	The operation failed without status available.

Remarks

An application can perform a VM-enter operation by using either the __vmx_vmlaunch or __vmx_vmresume function. The __vmx_vmlaunch function can be used only with a VMCS whose launch state is `Clear`, and the __vmx_vmresume function can be used only with a VMCS whose launch state is `Launched`. Consequently, use the __vmx_vmclear function to set the launch state of a VMCS to `Clear`. Use the __vmx_vmlaunch function for your first VM-enter operation and the __vmx_vmresume function for subsequent VM-enter operations.

The `__vmx_vmclear` function is equivalent to the `VMCLEAR` machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__vmx_vmclear`	x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__vmx_vmlaunch
__vmx_vmresume

__vmx_vmlaunch

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Places the calling application in VMX non-root operation state (VM enter) by using the current virtual-machine control structure (VMCS).

Syntax

```
unsigned char __vmx_vmlaunch(void);
```

Return value

VALUE	MEANING
0	The operation succeeded.
1	The operation failed with extended status available in the `VM-instruction error field` of the current VMCS.
2	The operation failed without status available.

Remarks

An application can perform a VM-enter operation by using either the __vmx_vmlaunch or __vmx_vmresume function. The __vmx_vmlaunch function can be used only with a VMCS whose launch state is `Clear`, and the __vmx_vmresume function can be used only with a VMCS whose launch state is `Launched`. Consequently, use the __vmx_vmclear function to set the launch state of a VMCS to `Clear`, and then use the __vmx_vmlaunch function for your first VM-enter operation and the __vmx_vmresume function for subsequent VM-enter operations.

The `__vmx_vmlaunch` function is equivalent to the `VMLAUNCH` machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__vmx_vmlaunch`	x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__vmx_vmresume
__vmx_vmclear

__vmx_vmptrld

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Loads the pointer to the current virtual-machine control structure (VMCS) from the specified address.

Syntax

```
int __vmx_vmptrld(
   unsigned __int64 *VmcsPhysicalAddress
);
```

Parameters

VmcsPhysicalAddress
[in] The address where the VMCS pointer is stored.

Return value

0
The operation succeeded.

1
The operation failed with extended status available in the `VM-instruction error field` of the current VMCS.

2
The operation failed without status available.

Remarks

The VMCS pointer is a 64-bit physical address.

The `__vmx_vmptrld` function is equivalent to the `VMPTRLD` machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__vmx_vmptrld`	x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__vmx_vmptrst

__vmx_vmptrst

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Stores the pointer to the current virtual-machine control structure (VMCS) at the specified address.

Syntax

```
void __vmx_vmptrst(
   unsigned __int64 *VmcsPhysicalAddress
);
```

Parameters

VmcsPhysicalAddress
[in] The address where the current VMCS pointer is stored.

Remarks

The VMCS pointer is a 64-bit physical address.

The `__vmx_vmptrst` function is equivalent to the `VMPTRST` machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__vmx_vmptrst`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__vmx_vmptrld

__vmx_vmread

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Reads a specified field from the current virtual machine control structure (VMCS) and places it in the specified location.

Syntax

```
unsigned char __vmx_vmread(
   size_t Field,
   size_t *FieldValue
);
```

Parameters

Field
[in] The VMCS field to read.

FieldValue
[in] A pointer to the location to store the value read from the VMCS field specified by the `Field` parameter.

Return value

VALUE	MEANING
0	The operation succeeded.
1	The operation failed with extended status available in the `VM-instruction error field` of the current VMCS.
2	The operation failed without status available.

Remarks

The `__vmx_vmread` function is equivalent to the `VMREAD` machine instruction. The value of the `Field` parameter is an encoded field index that is described in Intel documentation. For more information, search for Appendix C of "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," at the Intel Corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__vmx_vmread`	x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics

__vmx_vmwrite

__vmx_vmresume

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Resumes VMX non-root operation by using the current virtual machine control structure (VMCS).

Syntax

```
unsigned char __vmx_vmresume(
   void);
```

Return value

VALUE	MEANING
0	The operation succeeded.
1	The operation failed with extended status available in the `VM-instruction error field` of the current VMCS.
2	The operation failed without status available.

Remarks

An application can perform a VM-enter operation by using either the __vmx_vmlaunch or `__vmx_vmresume` function. The `__vmx_vmlaunch` function can be used only with a VMCS whose launch state is `Clear`, and the `__vmx_vmresume` function can be used only with a VMCS whose launch state is `Launched`. Consequently, use the __vmx_vmclear function to set the launch state of a VMCS to `Clear`, and then use the `__vmx_vmlaunch` function for your first VM-enter operation and the `__vmx_vmresume` function for subsequent VM-enter operations.

The `__vmx_vmresume` function is equivalent to the `VMRESUME` machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the PDF document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__vmx_vmresume`	x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics

__vmx_vmlaunch

__vmx_vmclear

__vmx_vmwrite

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Writes the specified value to the specified field in the current virtual machine control structure (VMCS).

Syntax

```
unsigned char __vmx_vmwrite(
   size_t Field,
   size_t FieldValue
);
```

Parameters

Field
[in] The VMCS field to write.

FieldValue
[in] The value to write to the VMCS field.

Return value

0
The operation succeeded.

1
The operation failed with extended status available in the `VM-instruction error field` of the current VMCS.

2
The operation failed without status available.

Remarks

The `__vmx_vmwrite` function is equivalent to the `VMWRITE` machine instruction. The value of the `Field` parameter is an encoded field index that is described in Intel documentation. For more information, search for Appendix C of "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," at the Intel Corporation site.

Requirements

INTRINSIC	ARCHITECTURE
`__vmx_vmwrite`	x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__vmx_vmread

__wbinvd

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the Write Back and Invalidate Cache (`wbinvd`) instruction.

Syntax

```
void __wbinvd(void);
```

Requirements

INTRINSIC	ARCHITECTURE
`__wbinvd`	x86, x64

Header file <intrin.h>

Remarks

This function is only available in kernel mode with a privilege level (CPL) of 0, and the routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

_WriteBarrier

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Limits the compiler optimizations that can reorder memory access operations across the point of the call.

Caution

The `_ReadBarrier`, `_WriteBarrier`, and `_ReadWriteBarrier` compiler intrinsics and the `MemoryBarrier` macro are all deprecated and should not be used. For inter-thread communication, use mechanisms such as atomic_thread_fence and std::atomic<T>, which are defined in the C++ Standard Library. For hardware access, use the /volatile:iso compiler option together with the volatile keyword.

Syntax

```
void _WriteBarrier(void);
```

Requirements

INTRINSIC	ARCHITECTURE
`_WriteBarrier`	x86, x64

Header file <intrin.h>

Remarks

The `_WriteBarrier` intrinsic limits the compiler optimizations that can remove or reorder memory access operations across the point of the call.

END Microsoft Specific

See also

_ReadBarrier
_ReadWriteBarrier
Compiler intrinsics
Keywords

__writecr0

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Writes the value `Data` to the CR0 register.

Syntax

```
void writecr0(
   unsigned __int64 Data
);
```

Parameters

Data
[in] The value to write to the CR0 register.

Requirements

INTRINSIC	ARCHITECTURE
`__writecr0`	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__writecr3

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Writes the value `Data` to the CR3 register.

Syntax

```
void writecr3(
   unsigned __int64 Data
);
```

Parameters

Data
[in] The value to write to the CR3 register.

Requirements

INTRINSIC	ARCHITECTURE
`__writecr3`	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__writecr4

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Writes the value `Data` to the CR4 register.

Syntax

```
void writecr4(
   unsigned __int64 Data
);
```

Parameters

Data
[in] The value to write to the CR4 register.

Requirements

INTRINSIC	ARCHITECTURE
`__writecr4`	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__writecr8

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Writes the value `Data` to the CR8 register.

Syntax

```
void writecr8(
   unsigned __int64 Data
);
```

Parameters

Data
[in] The value to write to the CR8 register.

Requirements

INTRINSIC	ARCHITECTURE
`__writecr8`	x64

Header file <intrin.h>

Remarks

The `__writecr8` intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

__writedr

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Writes the specified value to the specified debug register.

Syntax

```
void __writedr(unsigned DebugRegister, unsigned DebugValue); /* x86 */
void __writedr(unsigned DebugRegister, unsigned __int64 DebugValue); /* x64 */
```

Parameters

DebugRegister
[in] A number from 0 through 7 that identifies the debug register.

DebugValue
[in] A value to write to the debug register.

Remarks

These intrinsics are available only in kernel mode, and the routines are available only as intrinsics.

Requirements

INTRINSIC	ARCHITECTURE
`__writedr`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__readdr

__writeeflags

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Writes the specified value to the program status and control (EFLAGS) register.

Syntax

```
void __writeeflags(unsigned Value); /* x86 */
void __writeeflags(unsigned __int64 Value); /* x64 */
```

Parameters

Value
[in] The value to write to the EFLAGS register. The `Value` parameter is 32 bits long for a 32-bit platform and 64 bits long for a 64-bit platform.

Remarks

These routines are available only as intrinsics.

Requirements

INTRINSIC	ARCHITECTURE
`__writeeflags`	x86, x64

Header file <intrin.h>

END Microsoft Specific

See also

Compiler intrinsics
__readeflags

__writefsbyte, __writefsdword, __writefsqword, __writefsword

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Write memory to a location specified by an offset relative to the beginning of the FS segment.

Syntax

```
void __writefsbyte(
   unsigned long Offset,
   unsigned char Data
);
void __writefsword(
   unsigned long Offset,
   unsigned short Data
);
void __writefsdword(
   unsigned long Offset,
   unsigned long Data
);
void __writefsqword(
   unsigned long Offset,
   unsigned __int64 Data
);
```

Parameters

Offset
[in] The offset from the beginning of FS to write to.

Data
[in] The value to write.

Requirements

INTRINSIC	ARCHITECTURE
`__writefsbyte`	x86
`__writefsword`	x86
`__writefsdword`	x86
`__writefsqword`	x86

Header file <intrin.h>

Remarks

These routines are available only as intrinsics.

END Microsoft Specific

See also

__readfsbyte, __readfsdword, __readfsqword, __readfsword
Compiler intrinsics

__writegsbyte, __writegsdword, __writegsqword, __writegsword

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Write memory to a location specified by an offset relative to the beginning of the GS segment.

Syntax

```
void __writegsbyte(
   unsigned long Offset,
   unsigned char Data
);
void __writegsword(
   unsigned long Offset,
   unsigned short Data
);
void __writegsdword(
   unsigned long Offset,
   unsigned long Data
);
void __writegsqword(
   unsigned long Offset,
   unsigned __int64 Data
);
```

Parameters

Offset
[in] The offset from the beginning of GS to write to.

Data
[in] The value to write.

Requirements

INTRINSIC	ARCHITECTURE
`__writegsbyte`	x64
`__writegsdword`	x64
`__writegsqword`	x64
`__writegsword`	x64

Header file <intrin.h>

Remarks

These routines are only available as an intrinsic.

END Microsoft Specific

See also

__readgsbyte, __readgsdword, __readgsqword, __readgsword

Compiler intrinsics

__writemsr

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Generates the Write to Model Specific Register (`wrmsr`) instruction.

Syntax

```
void __writemsr(
   unsigned long Register,
   unsigned __int64 Value
);
```

Parameters

Register
[in] The model-specific register.

Value
[in] The value to write.

Requirements

INTRINSIC	ARCHITECTURE
`__writemsr`	x86, x64

Header file <intrin.h>

Remarks

This function may only be used in kernel mode, and this routine is only available as an intrinsic.

END Microsoft Specific

See also

Compiler intrinsics

Inline Assembler

9/2/2022 • 2 minutes to read • Edit Online

Microsoft Specific

Assembly language serves many purposes, such as improving program speed, reducing memory needs, and controlling hardware. You can use the inline assembler to embed assembly-language instructions directly in your C and C++ source programs without extra assembly and link steps. The inline assembler is built into the compiler, so you don't need a separate assembler such as the Microsoft Macro Assembler (MASM).

> **NOTE**
> Programs with inline assembler code are not fully portable to other hardware platforms. If you are designing for portability, avoid using inline assembler.

Inline assembly is not supported on the ARM and x64 processors. The following topics explain how to use the Visual C/C++ inline assembler with x86 processors:

- Inline Assembler Overview
- Advantages of Inline Assembly
- __asm
- Using Assembly Language in __asm Blocks
- Using C or C++ in __asm Blocks
- Using and Preserving Registers in Inline Assembly
- Jumping to Labels in Inline Assembly
- Calling C Functions in Inline Assembly
- Calling C++ Functions in Inline Assembly
- Defining __asm Blocks as C Macros
- Optimizing Inline Assembly

END Microsoft Specific

See also

Compiler Intrinsics and Assembly Language
C++ Language Reference

ARM Assembler reference

9/2/2022 • 2 minutes to read • Edit Online

The articles in this section of the documentation provide reference material for the Microsoft ARM assembler (`armasm` or `armasm64`) and related tools.

Related articles

TITLE	DESCRIPTION
ARM Assembler command-line reference	Describes the Microsoft armasm and armasm64 command-line options.
ARM Assembler diagnostic messages	Describes commonly seen armasm and armasm64 warning and error messages.
ARM Assembler directives	Describes the ARM directives that are different in Microsoft armasm and armasm64.
ARM Architecture Reference Manual on the ARM Developer website.	Choose the relevant manual for your ARM architecture. Each contains reference sections about ARM, Thumb, NEON, and VFP, and additional information about the ARM assembly language.
ARM Compiler armasm User Guide on the ARM Developer website.	Choose a recent version to find up-to-date information about the ARM assembly language.

IMPORTANT

The armasm assembler that the ARM Developer website describes isn't the same as the Microsoft armasm assembler that's included in Visual Studio and is documented in this section.

See also

ARM intrinsics
ARM64 intrinsics
Compiler intrinsics

Microsoft Macro Assembler reference

9/2/2022 • 2 minutes to read • Edit Online

The Microsoft Macro Assembler (MASM) provides several advantages over inline assembly. MASM contains a macro language that has features such as looping, arithmetic, and text string processing. MASM gives you greater control over the hardware. By using MASM, you also can reduce time and memory overhead in your code.

In This Section

ML and ML64 command-line option
Describes the ML and ML64 command-line options.

MASM for x64 (ml64.exe)
Information about how to create output files for x64.

Instruction Format
Describes basic instruction format and instruction prefixes for MASM.

Directives reference
Provides links to articles that discuss the use of directives in MASM.

Symbols Reference
Provides links to articles that discuss the use of symbols in MASM.

Operators Reference
Provides links to articles that discuss the use of operators in MASM.

ML error messages
Describes fatal and nonfatal error messages and warnings.

Processor Manufacturer Programming Manuals
Provides links to programming information about processors not manufactured, sold, or supported by Microsoft.

MASM BNF Grammar

Formal BNF description of MASM for x64.

Related Sections

C++ in Visual Studio
Provides links to different areas of the Visual Studio and Visual C++ documentation.

See also

Compiler Intrinsics
x86 Intrinsics
x64 (amd64) Intrinsics