

# 编译器内部函数和程序集语言

了解 Visual Studio 如何创建处理器特定的低级别代码以实现最优性能和控制。

## 使用编译器内部函数

### 概述

[编译器内部函数概述](#)

### 参考

[ARM 内部函数](#)

[ARM64 内部函数](#)

[x86 内部函数列表](#)

[x64 \(amd64\) 内部函数列表](#)

[在所有体系结构上都可用的内部函数](#)

[按字母顺序排列的内部函数列表](#)

## 在 Microsoft C/C++ 中使用适用于 x86 的内联程序集

### 概述

[内联汇编程序概述](#)

### 概念

[\\_asm 关键字](#)

### 操作指南

[在 \\_asm 块中使用汇编语言](#)

[在 \\_asm 块中使用 C 或 C++](#)

[使用和保留寄存器](#)

[在内联程序集中调用 C++ 函数](#)

## 使用 ARM 和 ARM64 汇编程序

### 参考

[ARM 和 ARM64 汇编程序引用](#)

[ARM 和 ARM64 汇编程序命令行参考](#)

[ARM 和 ARM64 汇编程序诊断消息](#)

[ARM 和 ARM64 汇编程序指令](#)

## 使用 x86 和 x64 汇编程序

### 参考

[适用于 x86 和 x64 的 Microsoft 宏汇编程序参考](#)

[MASM for x64 \(ml64.exe\)](#)

[ML 和 ML64 命令行参考](#)

[指令参考](#)

[符号参考](#)

[运算符参考](#)

[MASM BNF 语法](#)

[ML 和 ML64 错误消息](#)

[处理器制造商编程手册](#)

# 编译器内部函数

项目 · 2023/06/16

大多数函数都包含在库中，但也有一些函数是在编译器中生成的（即内部函数）。这些被称为内联函数或内部函数。

## 注解

如果一个函数是内部函数，在通常会采用内联方式插入该函数的代码，从而避免函数调用的开销并可发出该函数的高效率计算机指令。内部函数通常比等效的内联程序集速度更快，因为优化程序拥有众多内部函数行为方式的内置知识，因此可以优化使用内联程序集无法优化的内容。此外，优化程序还可以采用不同的方式扩展内部函数、对齐缓冲区或根据上下文和调用自变量进行其他方面的调整。

使用内部函数会影响到代码的可移植性，因为在 Visual C++ 中可用的内部函数如果用其他编译器编译代码则可能不可用，并且对于某些目标体系结构可用的部分内部函数并非对所有体系结构都可用。但是，内部函数通常比内联程序集可移植性更大。64 位体系结构要求内部函数，但不支持内联程序集。

某些内部函数（例如 `_assume` 和 `_ReadWriteBarrier`）向编译器提供信息，但这会影响优化程序的行为。

某些内部函数只能用作内部函数，某些内部函数可以同时用于函数和内部函数实现。你可以指示编译器使用这两种方式中的一种来使用内部函数实现，具体取决于你是想仅启用特定函数还是想启用所有内部函数。第一种方法是使用 `#pragma intrinsic(intrinsic-function-name-list)`。杂注可用于指定单个内部函数或用逗号分隔的多个内部函数。第二种方法是使用 `/Oi (生成内部函数)` 编译器选项，让指定平台的所有内部函数可用。在 `/Oi` 下，使用 `#pragma function(intrinsic-function-name-list)` 强制使用函数调用，而不是内部函数。如果特定内部函数的文档规定例程只能用作内部函数，则不管是指定 `/Oi` 还是 `#pragma intrinsic` 都会使用内部函数实现。在所有情况下，`/Oi` 或 `#pragma intrinsic` 允许但不是强制优化程序使用内部函数。优化程序仍然可以调用函数。

一些标准的 C/C++ 库函数在某些体系结构上可用于内部函数实现。调用 CRT 函数时，如果在命令行指定了 `/Oi`，则会使用内部函数实现。

头文件 `<intrin.h>` 可用，其用于声明常用内部函数的原型。制造商指定的内部函数可用于 `<immintrin.h>` 和 `<ammintrin.h>` 头文件。此外，某些 Windows 头文件还可声明在编译器内部函数上映射的函数。

以下部分列出了可用于各种体系结构的所有内部函数。有关内部函数在特定目标处理器上的工作方式的详细信息，请参阅制造商参考文档。

- ARM 内部函数
- ARM64 内部函数
- x86 内部函数列表
- x64 (amd64) 内部函数列表
- 在所有体系结构上都可用的内部函数
- 按字母顺序排序的内部函数列表

## 另请参阅

[ARM 汇编程序参考](#)

[Microsoft 宏汇编程序参考](#)

[关键字](#)

[C 运行时库参考](#)

# ARM 内部函数

项目 • 2023/06/16

Microsoft C++ 编译器 (MSVC) 使以下内部函数可用于 ARM 体系结构。有关 ARM 的详细信息，请参阅 [ARM 开发人员文档](#) 网站的体系结构和软件开发工具部分。

## NEON

ARM 的 NEON 向量指令集扩展提供 Single Instruction Multiple Data (SIMD) 功能，类似于 x86 和 x64 架构处理器通用的 MMX 和 SSE 向量指令集中的功能。

根据头文件 `arm_neon.h`，霓虹灯内部函数受到支持。MSVC 对 NEON 内部函数的支持类似于 ARM 编译器的相应支持，ARM 信息中心网站上的 [ARM 编译器工具链，版本 4.1 编译器参考](#) 的附录 G 中记录了相关内容。

MSVC 和 ARM 编译器之间的主要区别在于 MSVC 添加了 `vldx` 和 `vstx` 矢量加载和存储指令的 `_ex` 变体。`_ex` 变量采用附加参数，该参数指定指针参数的对齐情况但在其他方面与其非 `_ex` 对应项相同。

## 特定于 ARM 的内部函数列表

函数名称	指令	函数原型
<code>_arm_smlal</code>	SMLAL	<code>_int64 _arm_smlal(_int64 _RdHiLo, int _Rn, int _Rm)</code>
<code>_arm_umal</code>	UMLAL	<code>unsigned _int64 _arm_umal(unsigned _int64 _RdHiLo, unsigned int _Rn, unsigned int _Rm)</code>
<code>_arm_clz</code>	CLZ	<code>unsigned int _arm_clz(unsigned int _Rm)</code>
<code>_arm_qadd</code>	QADD	<code>int _arm_qadd(int _Rm, int _Rn)</code>
<code>_arm_qdadd</code>	QDADD	<code>int _arm_qdadd(int _Rm, int _Rn)</code>
<code>_arm_qdsub</code>	QDSUB	<code>int _arm_qdsub(int _Rm, int _Rn)</code>
<code>_arm_qsub</code>	QSUB	<code>int _arm_qsub(int _Rm, int _Rn)</code>
<code>_arm_smlabb</code>	SMLABB	<code>int _arm_smlabb(int _Rn, int _Rm, int _Ra)</code>
<code>_arm_smlabt</code>	SMLABT	<code>int _arm_smlabt(int _Rn, int _Rm, int _Ra)</code>
<code>_arm_smlatb</code>	SMLATB	<code>int _arm_smlatb(int _Rn, int _Rm, int _Ra)</code>
<code>_arm_smlatt</code>	SMLATT	<code>int _arm_smlatt(int _Rn, int _Rm, int _Ra)</code>

函数名称	指令	函数原型
_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)
_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)

函数名称	指令	函数原型
_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)
_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)

函数名称	指令	函数原型
_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)
_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)

函数名称	指令	函数原型
_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)
_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_smlaldux	SMLALDX	_int64 _arm_smlaldux(_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)

函数名称	指令	函数原型
_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)
_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)
		将一个内存屏障操作插入指令流中。参数 <code>_Type</code> 指定屏障强制执行的限制类型。
		要详细了解可以强制执行的限制类型，请参阅 <a href="#">内存屏障限制</a> 。

函数名称	指令	函数原型
__dsb	DSB	void __dsb(unsigned int _Type)
		<p>将一个内存屏障操作插入指令流中。参数 <code>_Type</code> 指定屏障强制执行的限制类型。</p> <p>要详细了解可以强制执行的限制类型，请参阅<a href="#">内存屏障限制</a>。</p>
__isb	ISB	<p>void __isb(unsigned int _Type)</p> <p>将一个内存屏障操作插入指令流中。参数 <code>_Type</code> 指定屏障强制执行的限制类型。</p> <p>要详细了解可以强制执行的限制类型，请参阅<a href="#">内存屏障限制</a>。</p>
__emit		<p>void __emit(unsigned __int32 opcode)</p> <p>将指定指令插入由编译器输出的指令流中。</p> <p><code>opcode</code> 的值必须是在编译时已知的常量表达式。指令字大小为 16 位且忽略 <code>opcode</code> 最重要的 16 位。</p> <p>编译器在执行插入的指令前不尝试解释 <code>opcode</code> 的内容且不保证 CPU 或内存状态。</p> <p>编译器假定 CPU 和内存状态在执行插入的指令后保持不变。因此，更改状态的指令会对由编译器生成的正常代码产生不利影响。</p> <p>出于此原因，仅使用 <code>emit</code> 插入影响编译未正常处理的 CPU 状态（如协处理器状态）的指令或实现使用 <code>declspec(naked)</code> 声明的函数。</p>
__hvc	HVC	unsigned int __hvc(unsigned int, ...)
__iso_volatile_load16		<p><code>_int16 __iso_volatile_load16(const volatile __int16 *)</code></p> <p>有关详细信息，请参阅<a href="#">__iso_volatile_load/store 内部函数</a>。</p>
__iso_volatile_load32		<p><code>_int32 __iso_volatile_load32(const volatile __int32 *)</code></p> <p>有关详细信息，请参阅<a href="#">__iso_volatile_load/store 内部函数</a>。</p>

函数名称	指令	函数原型
<code>_iso_volatile_load64</code>		<code>_int64 __iso_volatile_load64(const volatile __int64 *)</code>
		有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
<code>_iso_volatile_load8</code>		<code>_int8 __iso_volatile_load8(const volatile __int8 *)</code>
		有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
<code>_iso_volatile_store16</code>		<code>void __iso_volatile_store16(volatile __int16 *, __int16)</code>
		有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
<code>_iso_volatile_store32</code>		<code>void __iso_volatile_store32(volatile __int32 *, __int32)</code>
		有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
<code>_iso_volatile_store64</code>		<code>void __iso_volatile_store64(volatile __int64 *, __int64)</code>
		有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
<code>_iso_volatile_store8</code>		<code>void __iso_volatile_store8(volatile __int8 *, __int8)</code>
		有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
<code>_ldrex</code>	LDREXD	<code>_int64 __ldrex(const volatile __int64 *)</code>
<code>_prefetch</code>	PLD	<code>void __cdecl __prefetch(const void *)</code>
		为系统提供一个 <code>PLD</code> 内存提示，便可很快访问指定地址处或指定地址附近的内存。某些系统可能会选择优化此内存访问模式以提高运行时性能。但是，从 c++ 语言的角度来看，此功能没有明显的影响，可能不执行任何操作。
<code>_rdpmccntr64</code>		<code>unsigned __int64 __rdpmccntr64(void)</code>
<code>_sev</code>	SEV	<code>void __sev(void)</code>
<code>_static_assert</code>		<code>void __static_assert(int, const char *)</code>
<code>_swi</code>	SVC	<code>unsigned int __swi(unsigned int, ...)</code>
<code>_trap</code>	BKPT	<code>int __trap(int, ...)</code>

函数名称	指令	函数原型
__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)
_MoveFromCoprocessor	MRC	unsigned int _MoveFromCoprocessor(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int)
		使用协处理器数据传输指令，读取 ARM 协处理器中的 数据。有关详细信息，请参阅 <a href="#">_MoveFromCoprocessor</a> 、 <a href="#">_MoveFromCoprocessor2</a> 。
_MoveFromCoprocessor2	MRC2	unsigned int _MoveFromCoprocessor2(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int)
		使用协处理器数据传输指令，读取 ARM 协处理器中的 数据。有关详细信息，请参阅 <a href="#">_MoveFromCoprocessor</a> 、 <a href="#">_MoveFromCoprocessor2</a> 。

函数名称	指令	函数原型
_MoveFromCoprocessor64	MRRC	unsigned __int64 _MoveFromCoprocessor64(unsigned int, unsigned int, unsigned int)
		使用协处理器数据传输指令，读取 ARM 协处理器中的数据。有关详细信息，请参阅 <a href="#">_MoveFromCoprocessor64</a> 。
_MoveToCoprocessor	MCR	void _MoveToCoprocessor(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int)
		使用协处理器数据传输指令，读取 ARM 协处理器中的数据。有关详细信息，请参阅 <a href="#">_MoveToCoprocessor</a> 、 <a href="#">_MoveToCoprocessor2</a> 。
_MoveToCoprocessor2	MCR2	void _MoveToCoprocessor2(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int)
		使用协处理器数据传输指令，读取 ARM 协处理器中的数据。有关详细信息，请参阅 <a href="#">_MoveToCoprocessor</a> 、 <a href="#">_MoveToCoprocessor2</a> 。
_MoveToCoprocessor64	MCRR	void _MoveToCoprocessor64(unsigned __int64, unsigned int, unsigned int, unsigned int)
		使用协处理器数据传输指令，读取 ARM 协处理器中的数据。有关详细信息，请参阅 <a href="#">_MoveToCoprocessor64</a> 。
_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)

[[返回顶部](#)]

## 内存屏障限制

内部函数 `_dmb` (数据内存屏障)、`_dsb` (数据同步屏障) 和 `_isb` (指令同步屏障) 使用以下预定义值以根据共享域和受操作影响的访问类型对内存屏障限制进行指定。

限制值	说明
<code>_ARM_BARRIER_SY</code>	完整系统，读取和写入操作。
<code>_ARM_BARRIER_ST</code>	完整系统，只写操作。
<code>_ARM_BARRIER_ISH</code>	内部可共享，读取和写入操作。
<code>_ARM_BARRIER_ISHST</code>	内部可共享，只写操作。
<code>_ARM_BARRIER_NSH</code>	不可共享，读取和写入操作。
<code>_ARM_BARRIER_NSHST</code>	不可共享，只写操作。
<code>_ARM_BARRIER_OSH</code>	外部可共享，读取和写入操作。
<code>_ARM_BARRIER_OSHST</code>	外部可共享，只写操作。

对于 `_isb` 内部函数，当前有效的唯一限制是 `_ARM_BARRIER_SY`；其他所有值由体系结构进行保留。

## `_iso_volatile_load/store` 内部函数

这些内部函数显式执行不进行编译器优化的加载和存储。

C

```
_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);
```

## 参数

### 位置

要从中读取或为其写入的内存位置的地址。

### 值

要写入指定内存位置的值（仅存储内部函数）。

## 返回值（仅加载内部函数）

由 `Location` 指定的内存位置的值。

### 注解

你可以使用 `_iso_volatile_load8/16/32/64` 和 `_iso_volatile_store8/16/32/64` 内部函数显式执行不进行编译器优化的内存访问。编译器不能移除、同步或更改这些操作的相对顺序，但不会生成隐式硬件内存屏障。因此，硬件仍可能对跨多个线程的可观察内存访问进行重新排序。更准确地说，这些内部函数等效于在 `/volatile:iso` 下编译的以下表达式。

C++

```
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;
```

请注意内部函数采用易失性指针来适应易失性变量。但是，没有要求或建议使用 `volatile` 指针作为参数。如果使用常规的非易失性类型，这些操作的语义完全相同。

有关 `/volatile:iso` 命令行参数的详细信息，请参阅 [/volatile \(volatile 关键字解释\)](#)。

## `_MoveFromCoprocessor, _MoveFromCoprocessor2`

这些内部函数通过使用协处理器数据传输指令从 ARM 协处理器读取数据。

C

```
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
);
```

## 参数

coproc

0 到 15 之间的协处理器编号。

opcode1

0 到 7 之间特定于协处理器的操作码

crn

协处理器寄存器编号，在 0 到 15 之间，用于向指令指定第一个操作数。

crm

协处理器寄存器编号，在 0 到 15 之间，用于指定附加的源或目标操作数。

opcode2

0 到 7 之间特定于附加协处理器的操作码。

## 返回值

从协处理器中读取的值。

## 注解

此内部函数的所有五个参数的值必须是在编译时已知的常数表达式。

`_MoveFromCoprocessor` 使用 MRC 指令；`_MoveFromCoprocessor2` 使用 MRC2。此参数对应于直接编码到指令字中的位字段。参数的解释与协处理器相关。有关详细信息，请参见有问题的协处理器的手册。

## `_MoveFromCoprocessor64`

使用协处理器数据传输指令，读取 ARM 协处理器中的数据。

C

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

## 参数

coproc

0 到 15 之间的协处理器编号。

opcode1

0 到 15 之间特定于协处理器的操作码。

crm

协处理器寄存器编号，在 0 到 15 之间，用于指定附加的源或目标操作数。

## 返回值

从协处理器中读取的值。

## 注解

此内部函数的所有三个参数的值必须是在编译时已知的常数表达式。

`_MoveFromCoprocessor64` 使用 MRRC 指令。此参数对应于直接编码到指令字中的位字段。参数的解释与协处理器相关。有关详细信息，请参见有问题的协处理器的手册。

## `_MoveToCoprocessor, _MoveToCoprocessor2`

这些内部函数通过使用协处理器数据传输指令将数据写入到 ARM 协处理器。

C

```
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
);
```

## 参数

*value*

要写入到协处理器的值。

*coproc*

0 到 15 之间的协处理器编号。

*opcode1*

0 到 7 之间特定于协处理器的操作码。

*crn*

协处理器寄存器编号，在 0 到 15 之间，用于向指令指定第一个操作数。

*crm*

协处理器寄存器编号，在 0 到 15 之间，用于指定附加的源或目标操作数。

*opcode2*

0 到 7 之间特定于附加协处理器的操作码。

## 返回值

无。

## 备注

此内部函数的 *coproc*、*opcode1*、*crn*、*crm* 和 *opcode2* 参数的值必须是在编译时已知的常数表达式。

`_MoveToCoprocessor` 使用 MCR 指令；`_MoveToCoprocessor2` 使用 MCR2。此参数对应于直接编码到指令字中的位字段。参数的解释与协处理器相关。有关详细信息，请参见有问题的协处理器的手册。

## `_MoveToCoprocessor64`

这些内部函数通过使用协处理器数据传输指令将数据写入到 ARM 协处理器。

C

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

## 参数

`coproc`

0 到 15 之间的协处理器编号。

`opcode1`

0 到 15 之间特定于协处理器的操作码。

`crm`

协处理器寄存器编号，在 0 到 15 之间，用于指定附加的源或目标操作数。

## 返回值

无。

## 备注

内部函数的 `coproc`、`opcode1` 和 `crm` 参数的值必须是在编译时已知的常数表达式。

`_MoveFromCoprocessor64` 使用 MCRR 指令。此参数对应于直接编码到指令字中的位字段。参数的解释与协处理器相关。有关详细信息，请参见有问题的协处理器的手册。

## ARM 支持来自其他体系结构的内部函数

下表列出了来自在 ARM 平台上受支持的其他体系结构的内部函数。如果 ARM 上的内部函数行为不同于它在其他硬件体系结构上的行为，则记录其他详细信息。

函数名称	函数原型
<code>_assume</code>	<code>void __assume(int)</code>
<code>_code_seg</code>	<code>void __code_seg(const char *)</code>
<code>_debugbreak</code>	<code>void __cdecl __debugbreak(void)</code>
<code>_fastfail</code>	<code>_declspec(noreturn) void __fastfail(unsigned int)</code>
<code>_nop</code>	<code>void __nop(void)</code> 注意：如果在目标体系结构中实现了一个指令，则该函数会在 ARM 平台上生成一条 NOP 指令；否则，将生成一条不会改变程序或 CPU 状态的替代指令，例如 <code>MOV r8, r8</code> 。它在功能上等同于其他硬件体系结构的 <code>_nop</code> 内部函数。因为对项目或 CPU 的状态无影响的指令可能会因优化而被目标体系结构忽略，所以指令不一定消耗 CPU 周期。因此，不要使用 <code>_nop</code> 内部函数来操作代码序列的执行时间，除非确信 CPU 的运行方式。相反，你可以使用 <code>_nop</code> 内部函数将下一个指令对齐到特定的 32 位边界地址。

函数名称	函数原型
_yield	void __yield(void) 注意：在 ARM 平台上，此函数生成 YIELD 指令，这表明线程正在执行可以暂时暂停执行的任务（如旋转锁），而不会对该程序产生不利影响。它使 CPU 能够在执行周期内执行其他任务，这些任务如果不执行则会被浪费掉。
_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) 注意：在 ARM 平台上，此函数生成 CPSID 指令；它只能用作内部函数。
_enable	void __cdecl __enable(void) 注意：在 ARM 平台上，此函数生成 CPSIE 指令；它只能用作内部函数。
_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)

函数名称	函数原型
_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)

[[返回顶部](#)]

## 互锁内部函数

互锁内部函数是用于执行原子读取-修改-写入操作的一组内部函数。其中一些互锁内部函数通用于所有平台。此处单独将其列出不是因为它们数量庞大，而是因为它们的定义通常是多余的，一般情况下较容易就想到它们。它们的名称可用于派生确切行为。

下表总结了非 bittest 互锁内部函数的 ARM 支持。表中的每个单元格都对应一个名称，这些名称的派生方式是将该行的最左侧单元格中的操作名和该列的最上面单元格中的类型名附加到 `_Interlocked`。例如，`xor` 行和 `8` 列交叉处的单元格对应于 `_InterlockedXor8` 并且完全受支持。大部分受支持的函数提供以下可选后缀：`_acq`、`_rel` 和 `_nf`。`_acq` 后缀表示“获取”语义，而 `_rel` 后缀表示“发布”语义。`_nf` 或“no fence”后缀对于 ARM 来说是唯一的，我们将在下一节进行讨论。

操作	8	16	32	64	P
添加	无	无	完全	完全	无
且	完全	完全	完全	完全	无
CompareExchange	完全	完全	完全	完全	完全
递减	无	完全	完全	完全	无
Exchange	部分	部分	部分	部分	部分
ExchangeAdd	完全	完全	完全	完全	无
增量	无	完全	完全	完全	无
或	完全	完全	完全	完全	无
Xor	完全	完全	完全	完全	无

密钥:

- **完全**: 支持普通、`_acq`、`_rel` 和 `_nf` 窗体。
- **部分**: 支持普通、`_acq` 和 `_nf` 窗体。
- **无**: 不支持

## `_nf (no fence) 后缀`

`_nf` 或“no fence”后缀表示该操作不表现为任何类型的内存屏障，与表现为某种屏障的其他三种窗体（普通、`_acq` 和 `_rel`）相反。`_nf` 形式的一种可能用途是维护统计信息计数器，该计数器由多个线程同时更新，但在执行多个线程时不会使用该计数器的值。

## 互锁内部函数列表

函数名称	函数原型
<code>_InterlockedAdd</code>	<code>long _InterlockedAdd(long volatile *, long)</code>
<code>_InterlockedAdd64</code>	<code>_int64 _InterlockedAdd64(_int64 volatile *, _int64)</code>
<code>_InterlockedAdd64_acq</code>	<code>_int64 _InterlockedAdd64_acq(_int64 volatile *, _int64)</code>
<code>_InterlockedAdd64_nf</code>	<code>_int64 _InterlockedAdd64_nf(_int64 volatile *, _int64)</code>
<code>_InterlockedAdd64_rel</code>	<code>_int64 _InterlockedAdd64_rel(_int64 volatile *, _int64)</code>
<code>_InterlockedAdd_acq</code>	<code>long _InterlockedAdd_acq(long volatile *, long)</code>
<code>_InterlockedAdd_nf</code>	<code>long _InterlockedAdd_nf(long volatile *, long)</code>
<code>_InterlockedAdd_rel</code>	<code>long _InterlockedAdd_rel(long volatile *, long)</code>
<code>_InterlockedAnd</code>	<code>long _InterlockedAnd(long volatile *, long)</code>
<code>_InterlockedAnd16</code>	<code>short _InterlockedAnd16(short volatile *, short)</code>
<code>_InterlockedAnd16_acq</code>	<code>short _InterlockedAnd16_acq(short volatile *, short)</code>
<code>_InterlockedAnd16_nf</code>	<code>short _InterlockedAnd16_nf(short volatile *, short)</code>
<code>_InterlockedAnd16_rel</code>	<code>short _InterlockedAnd16_rel(short volatile *, short)</code>
<code>_InterlockedAnd64</code>	<code>_int64 _InterlockedAnd64(_int64 volatile *, _int64)</code>

函数名称	函数原型
_InterlockedAnd64_acq	<code>_int64 _InterlockedAnd64_acq(_int64 volatile *, _int64)</code>
_InterlockedAnd64_nf	<code>_int64 _InterlockedAnd64_nf(_int64 volatile *, _int64)</code>
_InterlockedAnd64_rel	<code>_int64 _InterlockedAnd64_rel(_int64 volatile *, _int64)</code>
_InterlockedAnd8	<code>char _InterlockedAnd8(char volatile *, char)</code>
_InterlockedAnd8_acq	<code>char _InterlockedAnd8_acq(char volatile *, char)</code>
_InterlockedAnd8_nf	<code>char _InterlockedAnd8_nf(char volatile *, char)</code>
_InterlockedAnd8_rel	<code>char _InterlockedAnd8_rel(char volatile *, char)</code>
_InterlockedAnd_acq	<code>long _InterlockedAnd_acq(long volatile *, long)</code>
_InterlockedAnd_nf	<code>long _InterlockedAnd_nf(long volatile *, long)</code>
_InterlockedAnd_rel	<code>long _InterlockedAnd_rel(long volatile *, long)</code>
_InterlockedCompareExchange	<code>long __cdecl _InterlockedCompareExchange(long volatile *, long, long)</code>
_InterlockedCompareExchange16	<code>short _InterlockedCompareExchange16(short volatile *, short, short)</code>
_InterlockedCompareExchange16_acq	<code>short _InterlockedCompareExchange16_acq(short volatile *, short, short)</code>
_InterlockedCompareExchange16_nf	<code>short _InterlockedCompareExchange16_nf(short volatile *, short, short)</code>
_InterlockedCompareExchange16_rel	<code>short _InterlockedCompareExchange16_rel(short volatile *, short, short)</code>
_InterlockedCompareExchange64	<code>_int64 _InterlockedCompareExchange64(_int64 volatile *, _int64, _int64)</code>
_InterlockedCompareExchange64_acq	<code>_int64 _InterlockedCompareExchange64_acq(_int64 volatile *, _int64, _int64)</code>
_InterlockedCompareExchange64_nf	<code>_int64 _InterlockedCompareExchange64_nf(_int64 volatile *, _int64, _int64)</code>
_InterlockedCompareExchange64_rel	<code>_int64 _InterlockedCompareExchange64_rel(_int64 volatile *, _int64, _int64)</code>

函数名称	函数原型
_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 *)
_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 *)

函数名称	函数原型
_InterlockedDecrement64_nf	<code>_int64 _InterlockedDecrement64_nf(_int64 volatile *)</code>
_InterlockedDecrement64_rel	<code>_int64 _InterlockedDecrement64_rel(_int64 volatile *)</code>
_InterlockedDecrement_acq	<code>long _InterlockedDecrement_acq(long volatile *)</code>
_InterlockedDecrement_nf	<code>long _InterlockedDecrement_nf(long volatile *)</code>
_InterlockedDecrement_rel	<code>long _InterlockedDecrement_rel(long volatile *)</code>
_InterlockedExchange	<code>long __cdecl _InterlockedExchange(long volatile * _Target, long)</code>
_InterlockedExchange16	<code>short _InterlockedExchange16(short volatile * _Target, short)</code>
_InterlockedExchange16_acq	<code>short _InterlockedExchange16_acq(short volatile * _Target, short)</code>
_InterlockedExchange16_nf	<code>short _InterlockedExchange16_nf(short volatile * _Target, short)</code>
_InterlockedExchange64	<code>_int64 _InterlockedExchange64(_int64 volatile * _Target, _int64)</code>
_InterlockedExchange64_acq	<code>_int64 _InterlockedExchange64_acq(_int64 volatile * _Target, _int64)</code>
_InterlockedExchange64_nf	<code>_int64 _InterlockedExchange64_nf(_int64 volatile * _Target, _int64)</code>
_InterlockedExchange8	<code>char _InterlockedExchange8(char volatile * _Target, char)</code>
_InterlockedExchange8_acq	<code>char _InterlockedExchange8_acq(char volatile * _Target, char)</code>
_InterlockedExchange8_nf	<code>char _InterlockedExchange8_nf(char volatile * _Target, char)</code>
_InterlockedExchangeAdd	<code>long __cdecl _InterlockedExchangeAdd(long volatile *, long)</code>
_InterlockedExchangeAdd16	<code>short _InterlockedExchangeAdd16(short volatile *, short)</code>
_InterlockedExchangeAdd16_acq	<code>short _InterlockedExchangeAdd16_acq(short volatile *, short)</code>

函数名称	函数原型
_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)
_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)

函数名称	函数原型
_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)
_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)

函数名称	函数原型
_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)
_InterlockedXor_acq	long _InterlockedXor_acq(long volatile *, long)
_InterlockedXor_nf	long _InterlockedXor_nf(long volatile *, long)
_InterlockedXor_rel	long _InterlockedXor_rel(long volatile *, long)

[[返回顶部](#)]

## \_interlockedbittest 内部函数

纯互锁的 bit 测试内部函数适用于所有平台。ARM 添加 `_acq`、`_rel` 和 `_nf` 变量，它们只修改操作的屏障语义，如本文前面的 `_nf (no fence)` 后缀所述。

函数名称	函数原型
<code>_interlockedbittestandreset</code>	<code>unsigned char _interlockedbittestandreset(long volatile *, long)</code>
<code>_interlockedbittestandreset_acq</code>	<code>unsigned char _interlockedbittestandreset_acq(long volatile *, long)</code>
<code>_interlockedbittestandreset_nf</code>	<code>unsigned char _interlockedbittestandreset_nf(long volatile *, long)</code>
<code>_interlockedbittestandreset_rel</code>	<code>unsigned char _interlockedbittestandreset_rel(long volatile *, long)</code>
<code>_interlockedbittestandset</code>	<code>unsigned char _interlockedbittestandset(long volatile *, long)</code>
<code>_interlockedbittestandset_acq</code>	<code>unsigned char _interlockedbittestandset_acq(long volatile *, long)</code>
<code>_interlockedbittestandset_nf</code>	<code>unsigned char _interlockedbittestandset_nf(long volatile *, long)</code>
<code>_interlockedbittestandset_rel</code>	<code>unsigned char _interlockedbittestandset_rel(long volatile *, long)</code>

[[返回顶部](#)]

## 另请参阅

[编译器内部函数](#)

[ARM64 内部函数](#)

[ARM 汇编程序参考](#)

[C++ 语言参考](#)

# ARM64 内部函数

项目 • 2023/06/16

Microsoft C++ 编译器 (MSVC) 使以下内部函数在 ARM64 体系结构上可用。有关 ARM 的详细信息，请参阅 [ARM 开发人员文档](#) 网站的体系结构和软件开发工具部分。

## NEON

适用于 ARM64 的 NEON 矢量指令集扩展提供 Single Instruction Multiple Data (SIMD) 功能。它们类似于 x86 和 x64 体系结构处理器通用的 MMX 和 SSE 矢量指令集中的指令集。

支持 NEON 内部函数，如头文件 arm64\_neon.h 中所提供的。MSVC 对 NEON 内部函数的支持类似于 ARM64 编译器，ARM 信息中心网站上的 [ARM NEON 内部函数参考](#) 中对此进行了记录。

## 特定于 ARM64 的内部函数列表

函数名称	指令	函数原型
_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)
_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)

函数名称	指令	函数原型
_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)
_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)

函数名称	指令	函数原型
_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)
		将一个内存屏障操作插入指令流中。参数 <code>_Type</code> 指定屏障强制执行的限制类型。
		有关可以强制执行的限制类型的详细信息，请参阅 <a href="#">内存屏障限制</a> 。
_dsb	DSB	void __dsb(unsigned int _Type)
		将一个内存屏障操作插入指令流中。参数 <code>_Type</code> 指定屏障强制执行的限制类型。
		有关可以强制执行的限制类型的详细信息，请参阅 <a href="#">内存屏障限制</a> 。
_isb	ISB	void __isb(unsigned int _Type)
		将一个内存屏障操作插入指令流中。参数 <code>_Type</code> 指定屏障强制执行的限制类型。
		有关可以强制执行的限制类型的详细信息，请参阅 <a href="#">内存屏障限制</a> 。
_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)

函数名称	指令	函数原型
_iso_volatile_load16		_int16 _iso_volatile_load16(const volatile _int16 *)  有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
_iso_volatile_load32		_int32 _iso_volatile_load32(const volatile _int32 *)  有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
_iso_volatile_load64		_int64 _iso_volatile_load64(const volatile _int64 *)  有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
_iso_volatile_load8		_int8 _iso_volatile_load8(const volatile _int8 *)  有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
_iso_volatile_store16		void _iso_volatile_store16(volatile _int16 *, _int16)  有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
_iso_volatile_store32		void _iso_volatile_store32(volatile _int32 *, _int32)  有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
_iso_volatile_store64		void _iso_volatile_store64(volatile _int64 *, _int64)  有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
_iso_volatile_store8		void _iso_volatile_store8(volatile _int8 *, _int8)  有关详细信息，请参阅 <a href="#">_iso_volatile_load/store 内部函数</a> 。
_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)

函数名称	指令	函数原型
_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)
_prefetch	PRFM	void __cdecl __prefetch(const void *)
		采用预提取操作 <code>PLDL1KEEP</code> 为系统提供一个 <code>PRFM</code> 内存提示，即可快速访问指定地址处或附近的内存。某些系统可能会选择优化此内存访问模式以提高运行时性能。但是，从 c++ 语言的角度来看，此功能没有明显的影响，可能不执行任何操作。
_prefetch2	PRFM	void __cdecl __prefetch(const void *, uint8_t prfop)
		采用所提供的预提取操作为系统提供一个 <code>PRFM</code> 内存提示，即可快速访问指定地址处或附近的内存。某些系统可能会选择优化此内存访问模式以提高运行时性能。但是，从 c++ 语言的角度来看，此功能没有明显的影响，可能不执行任何操作。
_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)

函数名称	指令	函数原型
__static_assert		void __static_assert(int, const char *)
_stlr8	STLRB	void _stlr8(unsigned _int8 volatile* _Target, unsigned _int8 _Value)
_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)
_swpl64	SWPL	unsigned _int64 _swpl64(unsigned _int64 volatile* _Target, unsigned _int64 _Value)

函数名称	指令	函数原型
_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)
_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)

函数名称	指令	函数原型
_CountOneBits		unsigned int _CountOneBits(unsigned long)
_CountOneBits64		unsigned int _CountOneBits64(unsigned __int64)
_ReadStatusReg	MRS	__int64 _ReadStatusReg(int)
_WriteStatusReg	MSR	void _WriteStatusReg(int, __int64)

[[返回顶部](#)]

## 内存屏障限制

内部函数 `__dmb` (数据内存屏障)、`__dsb` (数据同步屏障) 和 `__isb` (指令同步屏障) 使用以下预定义值以根据共享域和受操作影响的访问类型对内存屏障限制进行指定。

限制值	说明
<code>_ARM64_BARRIER_SY</code>	完整系统，读取和写入操作。
<code>_ARM64_BARRIER_ST</code>	完整系统，只写操作。
<code>_ARM64_BARRIER_LD</code>	完整系统，只读操作。
<code>_ARM64_BARRIER_ISH</code>	内部可共享，读取和写入操作。
<code>_ARM64_BARRIER_ISHST</code>	内部可共享，只写操作。
<code>_ARM64_BARRIER_ISHLD</code>	内部可共享，只读操作。
<code>_ARM64_BARRIER_NSH</code>	不可共享，读取和写入操作。
<code>_ARM64_BARRIER_NSHST</code>	不可共享，只写操作。
<code>_ARM64_BARRIER_NSHLD</code>	不可共享，只读操作。
<code>_ARM64_BARRIER_OSH</code>	外部可共享，读取和写入操作。
<code>_ARM64_BARRIER_OSHST</code>	外部可共享，只写操作。
<code>_ARM64_BARRIER_OSHLD</code>	外部可共享，只读操作。

对于 `__isb` 内部函数，当前有效的唯一限制是 `_ARM64_BARRIER_SY`；其他所有值由体系结构进行保留。

## `__iso_volatile_load/store` 内部函数

这些内部函数显式执行不进行编译器优化的加载和存储。

C

```
_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);
```

## 参数

### 位置

要从中读取或为其写入的内存位置的地址。

### 值

要写入指定内存位置的值（仅存储内部函数）。

## 返回值（仅加载内部函数）

由 Location 指定的内存位置的值。

## 注解

你可以使用 `__iso_volatile_load8/16/32/64` 和 `__iso_volatile_store8/16/32/64` 内部函数显式执行不进行编译器优化的内存访问。编译器无法删除、合成或更改这些操作的相对顺序。但是，它不会生成隐式硬件内存屏障。因此，硬件仍可能对跨多个线程的可观察内存访问进行重新排序。更准确地说，这些内部函数等效于在 /volatile:iso 下编译的以下表达式。

C++

```
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;
```

请注意内部函数采用易失性指针来适应易失性变量。但是，没有要求或建议使用 volatile 指针作为参数。如果使用常规的非易失类型，这些操作的语义完全相同。

有关 `/volatile:iso` 命令行参数的详细信息, 请参阅 [/volatile \(volatile 关键字解释\)](#)。

## ARM64 支持来自其他体系结构的内部函数

下表列出了来自在 ARM64 平台上受支持的其他体系结构的内部函数。如果 ARM64 上的内部函数的行为不同于它在其他硬件体系结构上的行为, 将注明附加的详细信息。

函数名称	函数原型
<code>_assume</code>	<code>void __assume(int)</code>
<code>_code_seg</code>	<code>void __code_seg(const char *)</code>
<code>_debugbreak</code>	<code>void __cdecl __debugbreak(void)</code>
<code>_fastfail</code>	<code>_declspec(noreturn) void __fastfail(unsigned int)</code>
<code>_nop</code>	<code>void __nop(void)</code>
<code>_yield</code>	<code>void __yield(void)</code> 注意: 在 ARM64 平台上, 此函数生成 YIELD 指令。此指令表示线程正在执行可从执行暂时挂起的任务, 例如旋转锁, 而不会对程序产生负面影响。它使 CPU 能够在执行周期内执行其他任务, 这些任务如果不执行则会被浪费掉。
<code>_AddressOfReturnAddress</code>	<code>void * __AddressOfReturnAddress(void)</code>
<code>_BitScanForward</code>	<code>unsigned char __BitScanForward(unsigned long * _Index, unsigned long _Mask)</code>
<code>_BitScanForward64</code>	<code>unsigned char __BitScanForward64(unsigned long * _Index, unsigned __int64 _Mask)</code>
<code>_BitScanReverse</code>	<code>unsigned char __BitScanReverse(unsigned long * _Index, unsigned long _Mask)</code>
<code>_BitScanReverse64</code>	<code>unsigned char __BitScanReverse64(unsigned long * _Index, unsigned __int64 _Mask)</code>
<code>_bittest</code>	<code>unsigned char __bittest(long const *, long)</code>
<code>_bittest64</code>	<code>unsigned char __bittest64(__int64 const *, __int64)</code>
<code>_bittestandcomplement</code>	<code>unsigned char __bittestandcomplement(long *, long)</code>
<code>_bittestandcomplement64</code>	<code>unsigned char __bittestandcomplement64(__int64 *, __int64)</code>
<code>_bittestandreset</code>	<code>unsigned char __bittestandreset(long *, long)</code>
<code>_bittestandreset64</code>	<code>unsigned char __bittestandreset64(__int64 *, __int64)</code>
<code>_bittestandset</code>	<code>unsigned char __bittestandset(long *, long)</code>

函数名称	函数原型
_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) 注意：在 ARM64 平台上，此函数生成指令 令 MSR DAIFCLR,#2；该指令仅作为内部函数提供。
_enable	void __cdecl _enable(void) 注意：在 ARM64 平台上，此函数生成指令 令 MSR DAIFSET,#2；该指令仅作为内部函数提供。
_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)

[[返回顶部](#)]

## 互锁内部函数

互锁内部函数是用于执行原子读取-修改-写入操作的一组内部函数。其中一些互锁内部函数通用于所有平台。因为它们数量很多，所以在这里单独列出。由于它们的定义大多是冗余的，因此，用一般术语来考虑更加容易。它们的名称可用于派生确切行为。

下表总结了非 bittest 互锁内部函数的 ARM64 支持。表中的每个单元格都对应一个名称，这些名称的派生方式是将该行的最左侧单元格中的操作名和该列的最上面单元格中的类型名附加到 `_Interlocked`。例如，`Xor` 行和 `8` 列交叉处的单元格对应于 `_InterlockedXor8` 并且完全受支持。大部分受支持的函数提供以下可选后缀：`_acq`、`_rel` 和 `_nf`。`_acq` 后缀表示“获取”语义，而 `_rel` 后缀表示“发布”语义。`_nf` 或“no fence”后缀是 ARM 和 ARM64 独有的，将在下一节中进行讨论。

操作	8	16	32	64	128	P
添加	无	无	完全	完全	无	无
且	完全	完全	完全	完全	无	无
CompareExchange	完全	完全	完全	完全	完全	完全
递减	无	完全	完全	完全	无	无
Exchange	完全	完全	完全	完全	无	完全
ExchangeAdd	完全	完全	完全	完全	无	无
增量	无	完全	完全	完全	无	无
或	完全	完全	完全	完全	无	无
Xor	完全	完全	完全	完全	无	无

密钥：

- **完全**：支持无格式、`_acq`、`_rel` 和 `_nf` 形式。
- **无**：不支持

## `_nf (no fence)` 后缀

`_nf` 或“no fence”后缀表示操作不表现为任何类型的内存屏障，与表现为某种类型屏障的其他三种形式（无格式、`_acq` 和 `_rel`）相反。`_nf` 形式的一种可能用途是维护统计信息计数器，该计数器由多个线程同时更新，但在执行多个线程时不会使用该计数器的值。

## 互锁内部函数列表

函数名称	函数原型
_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)

函数名称	函数原型
_InterlockedAnd_rel	long _InterlockedAnd_rel(long volatile *, long)
_InterlockedCompareExchange	long __cdecl _InterlockedCompareExchange(long volatile *, long, long)
_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)

函数名称	函数原型
_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 *)
_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 *)

函数名称	函数原型
_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)
_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)

函数名称	函数原型
_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)
_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)

函数名称	函数原型
_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 *)
_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)

函数名称	函数原型
_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)

函数名称	函数原型
_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)

[[返回顶部](#)]

## \_interlockedbittest 内部函数

纯互锁的 bit 测试内部函数通用于所有平台。 ARM64 添加了 `_acq`、`_rel` 和 `_nf` 变体，它们只是修改操作的屏障语义，如本文前面的 [\\_nf \(no fence\) 后缀](#) 中所述。

函数名称	函数原型
_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)
_interlockedbittestandset_acq	unsigned char _interlockedbittestandset_acq(long volatile *, long)
_interlockedbittestandset_nf	unsigned char _interlockedbittestandset_nf(long volatile *, long)

函数名称	函数原型
_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)

[[返回顶部](#)]

## 另请参阅

[编译器内部函数](#)

[ARM 内部函数](#)

[ARM 汇编程序参考](#)

[C++ 语言参考](#)

# x86 内部函数列表

项目 · 2023/06/16

本文档列出了 Microsoft C/C++ 编译器在面向 x86 时支持的内部函数。

有关各个内部函数的信息，请参见适用于你面向的处理器的资源：

- 头文件。 头文件的注释中记录了很多内部函数。
- [Intel 内部函数指南](#)。 使用搜索框查找特定的内部函数。
- [Intel 64 和 IA-32 体系结构软件开发人员手册](#)
- [Intel 体系结构指令集扩展编程参考](#)
- [Intel 高级矢量扩展](#)
- [AMD 开发人员指南、手册和 ISA 文档](#)

## x86 内部函数

下表列出了 x86 处理器上可用的内部函数。“技术”列列出了所需的指令集支持。使用 `_cpuid` 内部函数来确定运行时的指令集支持。如果两个条目均在一行中，则它们表示同一内部函数的不同入口点。[宏] 指示该原型是一个宏。“标题”列中列出了函数原型所需的标题。为简单起见，`intrin.h` 标头同时包含 `immintrin.h` 和 `ammintrin.h`。

内部函数名称	技术	标头	函数原型
<code>_addcarry_u16</code>		intrin.h	<code>unsigned char _addcarry_u16(unsigned char, unsigned short, unsigned short, unsigned short *);</code>
<code>_addcarry_u32</code>		intrin.h	<code>unsigned char _addcarry_u32(unsigned char, unsigned int, unsigned int, unsigned int *);</code>
<code>_addcarry_u8</code>		intrin.h	<code>unsigned char _addcarry_u8(unsigned char, unsigned char, unsigned char, unsigned char *);</code>
<code>_addcarryx_u32</code>	ADX	immintrin.h	<code>unsigned char _addcarryx_u32(unsigned char, unsigned int, unsigned int, unsigned int *);</code>
<code>_addfsbyte</code>		intrin.h	<code>void __addfsbyte(unsigned long, unsigned char);</code>
<code>_addfsdword</code>		intrin.h	<code>void __addfsdword(unsigned long, unsigned long);</code>
<code>_addfsword</code>		intrin.h	<code>void __addfsword(unsigned long, unsigned short);</code>
<code>_AddressOfReturnAddress</code>		intrin.h	<code>void * __AddressOfReturnAddress(void);</code>
<code>_andn_u32</code>	BMI	ammintrin.h	<code>unsigned int __andn_u32(unsigned int, unsigned int);</code>
<code>_bextr_u32</code>	BMI	ammintrin.h, immintrin.h	<code>unsigned int __bextr_u32(unsigned int, unsigned int, unsigned int);</code>
<code>_bextri_u32</code>	ABM	ammintrin.h	<code>unsigned int __bextri_u32(unsigned int, unsigned int);</code>
<code>_BitScanForward</code>		intrin.h	<code>unsigned char __BitScanForward(unsigned long*, unsigned long);</code>

内部函数名称	技术	标头	函数原型
_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);
_blsmask_u32 ↗	BMI	ammintrin.h, immintrin.h	unsigned int _blsmask_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);

内部函数名称	技术	标头	函数原型
_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);
_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(long *, long);
_interlockedbittestandreset_HLERelease	HLE	immintrin.h	unsigned char _interlockedbittestandreset_HLERelease(long *, long);
_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);

内部函数名称	技术	标头	函数原型
_InterlockedCompareExchange		intrin.h	long _InterlockedCompareExchange (long volatile *, long, long);
_InterlockedCompareExchange_HLEAcquire	HLE	immintrin.h	long _InterlockedCompareExchange_HLEAcquire(long volatile *, long, long);
_InterlockedCompareExchange_HLERelease	HLE	immintrin.h	long _InterlockedCompareExchange_HLERelease(long 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(__int64 volatile *, __int64, __int64);
_InterlockedCompareExchange64_HLERelease	HLE	immintrin.h	__int64 _InterlockedCompareExchange64_HLERelease(__int64 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_HLEAcquire(void *volatile *, void *, void *);
_InterlockedCompareExchangePointer_HLERelease	HLE	immintrin.h	void *_InterlockedCompareExchangePointer_HLERelease(void *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);
_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);

内部函数名称	技术	标头	函数原型
_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(void *volatile *, void *);
_InterlockedExchangePointer_HLERelease	HLE	immintrin.h	void * _InterlockedExchangePointer_HLERelease(void *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);
_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*); [宏]
_loadbe_i32 ↴	MOVBE	immintrin.h	int _loadbe_i32(void const*); [宏]

内部函数名称	技术	标头	函数原型
_load_be_u16	MOVBE	immintrin.h	unsigned short _load_be_u16(void const*); [宏]
_load_be_u32	MOVBE	immintrin.h	unsigned int _load_be_u32(void const*); [宏]
__llwpcb	LWP	ammintrin.h	void __llwpcb(void *);
__lwpins32	LWP	ammintrin.h	unsigned char __lwpins32(unsigned int, unsigned int, unsigned int);
__lpval32	LWP	ammintrin.h	void __lpval32(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);
_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_pccmpeqb	MMX	intrin.h	__m64 __m_pccmpeqb(__m64, __m64);
_m_pcmeqd	MMX	intrin.h	__m64 __m_pcmeqd(__m64, __m64);
_m_pccmpeqw	MMX	intrin.h	__m64 __m_pccmpeqw(__m64, __m64);

内部函数名称	技术	标头	函数原型
<a href="#">_m_pcmpgtb</a>	MMX	intrin.h	<code>__m64 _m_pcmpgtb(__m64, __m64);</code>
<a href="#">_m_pcmpgtd</a>	MMX	intrin.h	<code>__m64 _m_pcmpgtd(__m64, __m64);</code>
<a href="#">_m_pcmpgtw</a>	MMX	intrin.h	<code>__m64 _m_pcmpgtw(__m64, __m64);</code>
<a href="#">_m_pextrw</a>	SSE	intrin.h	<code>int _m_pextrw(__m64, int);</code>
<a href="#">_m_pf2id</a>	3DNow	intrin.h	<code>__m64 _m_pf2id(__m64);</code>
<a href="#">_m_pf2iw</a>	3DNowEXT	intrin.h	<code>__m64 _m_pf2iw(__m64);</code>
<a href="#">_m_pfacc</a>	3DNow	intrin.h	<code>__m64 _m_pfacc(__m64, __m64);</code>
<a href="#">_m_pfadd</a>	3DNow	intrin.h	<code>__m64 _m_pfadd(__m64, __m64);</code>
<a href="#">_m_pfccmpeq</a>	3DNow	intrin.h	<code>__m64 _m_pfccmpeq(__m64, __m64);</code>
<a href="#">_m_pfcmpge</a>	3DNow	intrin.h	<code>__m64 _m_pfcmpge(__m64, __m64);</code>
<a href="#">_m_pfcmpgt</a>	3DNow	intrin.h	<code>__m64 _m_pfcmpgt(__m64, __m64);</code>
<a href="#">_m_pfmax</a>	3DNow	intrin.h	<code>__m64 _m_pfmax(__m64, __m64);</code>
<a href="#">_m_pfmin</a>	3DNow	intrin.h	<code>__m64 _m_pfmin(__m64, __m64);</code>
<a href="#">_m_pfmul</a>	3DNow	intrin.h	<code>__m64 _m_pfmul(__m64, __m64);</code>
<a href="#">_m_pfnacc</a>	3DNowEXT	intrin.h	<code>__m64 _m_pfnacc(__m64, __m64);</code>
<a href="#">_m_pfpnacc</a>	3DNowEXT	intrin.h	<code>__m64 _m_pfpnacc(__m64, __m64);</code>
<a href="#">_m_pfrcp</a>	3DNow	intrin.h	<code>__m64 _m_pfrcp(__m64);</code>
<a href="#">_m_pfrcpit1</a>	3DNow	intrin.h	<code>__m64 _m_pfrcpit1(__m64, __m64);</code>
<a href="#">_m_pfrcpit2</a>	3DNow	intrin.h	<code>__m64 _m_pfrcpit2(__m64, __m64);</code>
<a href="#">_m_pfrsqit1</a>	3DNow	intrin.h	<code>__m64 _m_pfrsqit1(__m64, __m64);</code>
<a href="#">_m_pfrsqrt</a>	3DNow	intrin.h	<code>__m64 _m_pfrsqrt(__m64);</code>
<a href="#">_m_pfsub</a>	3DNow	intrin.h	<code>__m64 _m_pfsub(__m64, __m64);</code>
<a href="#">_m_pfsubr</a>	3DNow	intrin.h	<code>__m64 _m_pfsubr(__m64, __m64);</code>
<a href="#">_m_pi2fd</a>	3DNow	intrin.h	<code>__m64 _m_pi2fd(__m64);</code>
<a href="#">_m_pi2fw</a>	3DNowEXT	intrin.h	<code>__m64 _m_pi2fw(__m64);</code>
<a href="#">_m_pinsrw</a>	SSE	intrin.h	<code>__m64 _m_pinsrw(__m64, int, int);</code>
<a href="#">_m_pmaddwd</a>	MMX	intrin.h	<code>__m64 _m_pmaddwd(__m64, __m64);</code>
<a href="#">_m_pmaxsw</a>	SSE	intrin.h	<code>__m64 _m_pmaxsw(__m64, __m64);</code>
<a href="#">_m_pmaxub</a>	SSE	intrin.h	<code>__m64 _m_pmaxub(__m64, __m64);</code>
<a href="#">_m_pminsw</a>	SSE	intrin.h	<code>__m64 _m_pminsw(__m64, __m64);</code>
<a href="#">_m_pminub</a>	SSE	intrin.h	<code>__m64 _m_pminub(__m64, __m64);</code>
<a href="#">_m_pmovmskb</a>	SSE	intrin.h	<code>int _m_pmovmskb(__m64);</code>
<a href="#">_m_pmulhrw</a>	3DNow	intrin.h	<code>__m64 _m_pmulhrw(__m64, __m64);</code>

内部函数名称	技术	标头	函数原型
<a href="#">_m_pmulluw</a>	SSE	intrin.h	<code>__m64 _m_pmulluw(__m64, __m64);</code>
<a href="#">_m_pmullw</a>	MMX	intrin.h	<code>__m64 _m_pmullw(__m64, __m64);</code>
<a href="#">_m_pmulw</a>	MMX	intrin.h	<code>__m64 _m_pmulw(__m64, __m64);</code>
<a href="#">_m_por</a>	MMX	intrin.h	<code>__m64 _m_por(__m64, __m64);</code>
<a href="#">_m_prefetch</a>	3DNOW	intrin.h	<code>void _m_prefetch(void*);</code>
<a href="#">_m_prefetchw</a>	3DNOW	intrin.h	<code>void _m_prefetchw(void*);</code>
<a href="#">_m_psadbw</a>	SSE	intrin.h	<code>__m64 _m_psadbw(__m64, __m64);</code>
<a href="#">_m_pshufw</a>	SSE	intrin.h	<code>__m64 _m_pshufw(__m64, int);</code>
<a href="#">_m_pslld</a>	MMX	intrin.h	<code>__m64 _m_pslld(__m64, __m64);</code>
<a href="#">_m_pslldi</a>	MMX	intrin.h	<code>__m64 _m_pslldi(__m64, int);</code>
<a href="#">_m_psllq</a>	MMX	intrin.h	<code>__m64 _m_psllq(__m64, __m64);</code>
<a href="#">_m_psllqi</a>	MMX	intrin.h	<code>__m64 _m_psllqi(__m64, int);</code>
<a href="#">_m_psllw</a>	MMX	intrin.h	<code>__m64 _m_psllw(__m64, __m64);</code>
<a href="#">_m_psllwi</a>	MMX	intrin.h	<code>__m64 _m_psllwi(__m64, int);</code>
<a href="#">_m_psrad</a>	MMX	intrin.h	<code>__m64 _m_psrad(__m64, __m64);</code>
<a href="#">_m_psradi</a>	MMX	intrin.h	<code>__m64 _m_psradi(__m64, int);</code>
<a href="#">_m_psraw</a>	MMX	intrin.h	<code>__m64 _m_psraw(__m64, __m64);</code>
<a href="#">_m_psrawi</a>	MMX	intrin.h	<code>__m64 _m_psrawi(__m64, int);</code>
<a href="#">_m_psrlld</a>	MMX	intrin.h	<code>__m64 _m_psrlld(__m64, __m64);</code>
<a href="#">_m_psrlldi</a>	MMX	intrin.h	<code>__m64 _m_psrlldi(__m64, int);</code>
<a href="#">_m_psrlrq</a>	MMX	intrin.h	<code>__m64 _m_psrlrq(__m64, __m64);</code>
<a href="#">_m_psrlqi</a>	MMX	intrin.h	<code>__m64 _m_psrlqi(__m64, int);</code>
<a href="#">_m_psrlw</a>	MMX	intrin.h	<code>__m64 _m_psrlw(__m64, __m64);</code>
<a href="#">_m_psrlwi</a>	MMX	intrin.h	<code>__m64 _m_psrlwi(__m64, int);</code>
<a href="#">_m_psubb</a>	MMX	intrin.h	<code>__m64 _m_psubb(__m64, __m64);</code>
<a href="#">_m_psubd</a>	MMX	intrin.h	<code>__m64 _m_psubd(__m64, __m64);</code>
<a href="#">_m_psubsb</a>	MMX	intrin.h	<code>__m64 _m_psubsb(__m64, __m64);</code>
<a href="#">_m_psubsw</a>	MMX	intrin.h	<code>__m64 _m_psubsw(__m64, __m64);</code>
<a href="#">_m_psubusb</a>	MMX	intrin.h	<code>__m64 _m_psubusb(__m64, __m64);</code>
<a href="#">_m_psubusw</a>	MMX	intrin.h	<code>__m64 _m_psubusw(__m64, __m64);</code>
<a href="#">_m_psubw</a>	MMX	intrin.h	<code>__m64 _m_psubw(__m64, __m64);</code>
<a href="#">_m_pswapd</a>	3DNOWEXT	intrin.h	<code>__m64 _m_pswapd(__m64);</code>
<a href="#">_m_punpckhbw</a>	MMX	intrin.h	<code>__m64 _m_punpckhbw(__m64, __m64);</code>

内部函数名称	技术	标头	函数原型
_m_punpckhdq ↗	MMX	intrin.h	<code>__m64 _m_punpckhdq(__m64, __m64);</code>
_m_punpckhwd ↗	MMX	intrin.h	<code>__m64 _m_punpckhwd(__m64, __m64);</code>
_m_punpcklbw ↗	MMX	intrin.h	<code>__m64 _m_punpcklbw(__m64, __m64);</code>
_m_punpckldq ↗	MMX	intrin.h	<code>__m64 _m_punpckldq(__m64, __m64);</code>
_m_punpcklwd ↗	MMX	intrin.h	<code>__m64 _m_punpcklwd(__m64, __m64);</code>
_m_pxor ↗	MMX	intrin.h	<code>__m64 _m_pxor(__m64, __m64);</code>
_m_to_float ↗	3DNOW	intrin.h	<code>float _m_to_float(__m64);</code>
_m_to_int ↗	MMX	intrin.h	<code>int _m_to_int(__m64);</code>
_mm_abs_epi16 ↗	SSSE3	intrin.h	<code>__m128i _mm_abs_epi16(__m128i);</code>
_mm_abs_epi32 ↗	SSSE3	intrin.h	<code>__m128i _mm_abs_epi32(__m128i);</code>
_mm_abs_epi8 ↗	SSSE3	intrin.h	<code>__m128i _mm_abs_epi8(__m128i);</code>
_mm_abs_pi16 ↗	SSSE3	intrin.h	<code>__m64 _mm_abs_pi16(__m64);</code>
_mm_abs_pi32 ↗	SSSE3	intrin.h	<code>__m64 _mm_abs_pi32(__m64);</code>
_mm_abs_pi8 ↗	SSSE3	intrin.h	<code>__m64 _mm_abs_pi8(__m64);</code>
_mm_add_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_add_epi16(__m128i, __m128i);</code>
_mm_add_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_add_epi32(__m128i, __m128i);</code>
_mm_add_epi64 ↗	SSE2	intrin.h	<code>__m128i _mm_add_epi64(__m128i, __m128i);</code>
_mm_add_epi8 ↗	SSE2	intrin.h	<code>__m128i _mm_add_epi8(__m128i, __m128i);</code>
_mm_add_pd ↗	SSE2	intrin.h	<code>__m128d _mm_add_pd(__m128d, __m128d);</code>
_mm_add_pi8 ↗	MMX	mmmintrin.h	<code>__m64 _mm_add_pi8(__m64, __m64); [宏]</code>
_mm_add_pi16 ↗	MMX	mmmintrin.h	<code>__m64 _mm_add_pi16(__m64, __m64); [宏]</code>
_mm_add_pi32 ↗	MMX	mmmintrin.h	<code>__m64 _mm_add_pi32(__m64, __m64); [宏]</code>
_mm_add_ps ↗	SSE	intrin.h	<code>__m128 _mm_add_ps(__m128, __m128);</code>
_mm_add_sd ↗	SSE2	intrin.h	<code>__m128d _mm_add_sd(__m128d, __m128d);</code>
_mm_add_si64 ↗	SSE2	intrin.h	<code>__m64 _mm_add_si64(__m64, __m64);</code>
_mm_add_ss ↗	SSE	intrin.h	<code>__m128 _mm_add_ss(__m128, __m128);</code>
_mm_adds_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_adds_epi16(__m128i, __m128i);</code>
_mm_adds_epi8 ↗	SSE2	intrin.h	<code>__m128i _mm_adds_epi8(__m128i, __m128i);</code>
_mm_adds_epu16 ↗	SSE2	intrin.h	<code>__m128i _mm_adds_epu16(__m128i, __m128i);</code>
_mm_adds_epu8 ↗	SSE2	intrin.h	<code>__m128i _mm_adds_epu8(__m128i, __m128i);</code>
_mm_adds_pi8 ↗	MMX	mmmintrin.h	<code>__m64 _mm_adds_pi8(__m64, __m64); [宏]</code>
_mm_adds_pi16 ↗	MMX	mmmintrin.h	<code>__m64 _mm_adds_pi16(__m64, __m64); [宏]</code>
_mm_adds_pu8 ↗	MMX	mmmintrin.h	<code>__m64 _mm_adds_pu8(__m64, __m64); [宏]</code>

内部函数名称	技术	标头	函数原型
_mm_adds_ps	MMX	mmmintrin.h	<code>__m64 _mm_adds_ps(__m64, __m64); [宏]</code>
_mm_addsub_pd	SSE3	intrin.h	<code>__m128d _mm_addsub_pd(__m128d, __m128d);</code>
_mm_addsub_ps	SSE3	intrin.h	<code>__m128 _mm_addsub_ps(__m128, __m128);</code>
_mm_aesdec_si128	AESNI	immintrin.h	<code>__m128i _mm_aesdec_si128(__m128i, __m128i);</code>
_mm_aesdeclast_si128	AESNI	immintrin.h	<code>__m128i _mm_aesdeclast_si128(__m128i, __m128i);</code>
_mm_aesenc_si128	AESNI	immintrin.h	<code>__m128i _mm_aesenc_si128(__m128i, __m128i);</code>
_mm_aesenclast_si128	AESNI	immintrin.h	<code>__m128i _mm_aesenclast_si128(__m128i, __m128i);</code>
_mm_aesimc_si128	AESNI	immintrin.h	<code>__m128i _mm_aesimc_si128 (__m128i);</code>
_mm_aeskeygenassist_si128	AESNI	immintrin.h	<code>__m128i _mm_aeskeygenassist_si128 (__m128i, const int);</code>
_mm_alignr_epi8	SSSE3	intrin.h	<code>__m128i _mm_alignr_epi8(__m128i, __m128i, int);</code>
_mm_alignr_pi8	SSSE3	intrin.h	<code>__m64 _mm_alignr_pi8(__m64, __m64, int);</code>
_mm_and_pd	SSE2	intrin.h	<code>__m128d _mm_and_pd(__m128d, __m128d);</code>
_mm_and_ps	SSE	intrin.h	<code>__m128 _mm_and_ps(__m128, __m128);</code>
_mm_and_si64	MMX	mmmintrin.h	<code>__m64 _mm_and_si64(__m64, __m64); [宏]</code>
_mm_and_si128	SSE2	intrin.h	<code>__m128i _mm_and_si128(__m128i, __m128i);</code>
_mm_andnot_pd	SSE2	intrin.h	<code>__m128d _mm_andnot_pd(__m128d, __m128d);</code>
_mm_andnot_ps	SSE	intrin.h	<code>__m128 _mm_andnot_ps(__m128, __m128);</code>
_mm_andnot_si64	MMX	mmmintrin.h	<code>__m64 _mm_andnot_si64(__m64, __m64); [宏]</code>
_mm_andnot_si128	SSE2	intrin.h	<code>__m128i _mm_andnot_si128(__m128i, __m128i);</code>
_mm_avg_epu16	SSE2	intrin.h	<code>__m128i _mm_avg_epu16(__m128i, __m128i);</code>
_mm_avg_epu8	SSE2	intrin.h	<code>__m128i _mm_avg_epu8(__m128i, __m128i);</code>
_mm_blend_epi16	SSE41	intrin.h	<code>__m128i _mm_blend_epi16 (__m128i, __m128i, const int);</code>
_mm_blend_epi32	AVX2	immintrin.h	<code>__m128i _mm_blend_epi32(__m128i, __m128i, const int);</code>
_mm_blend_pd	SSE41	intrin.h	<code>__m128d _mm_blend_pd (__m128d, __m128d, const int);</code>
_mm_blend_ps	SSE41	intrin.h	<code>__m128 _mm_blend_ps (__m128, __m128, const int);</code>
_mm_blendv_epi8	SSE41	intrin.h	<code>__m128i _mm_blendv_epi8 (__m128i, __m128i, __m128i);</code>
_mm_blendv_pd	SSE41	intrin.h	<code>__m128d _mm_blendv_pd(__m128d, __m128d, __m128d);</code>
_mm_blendv_ps	SSE41	intrin.h	<code>__m128 _mm_blendv_ps(__m128, __m128, __m128);</code>
_mm_broadcast_ss	AVX	immintrin.h	<code>__m128 _mm_broadcast_ss(float const *);</code>
_mm_broadcastb_epi8	AVX2	immintrin.h	<code>__m128i _mm_broadcastb_epi8(__m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_broadcastd_epi32 ↗	AVX2	immintrin.h	<code>__m128i _mm_broadcastd_epi32(__m128i);</code>
_mm_broadcastq_epi64 ↗	AVX2	immintrin.h	<code>__m128i _mm_broadcastq_epi64(__m128i);</code>
_mm_broadcastsd_pd ↗	AVX2	immintrin.h	<code>__m128d _mm_broadcastsd_pd(__m128d);</code>
_mm_broadcastss_ps ↗	AVX2	immintrin.h	<code>__m128 _mm_broadcastss_ps(__m128);</code>
_mm_broadcastw_epi16 ↗	AVX2	immintrin.h	<code>__m128i _mm_broadcastw_epi16(__m128i);</code>
_mm_castpd_ps ↗	SSSE3	intrin.h	<code>__m128 _mm_castpd_ps(__m128d);</code>
_mm_castpd_si128 ↗	SSSE3	intrin.h	<code>__m128i _mm_castpd_si128(__m128d);</code>
_mm_castps_pd ↗	SSSE3	intrin.h	<code>__m128d _mm_castps_pd(__m128);</code>
_mm_castps_si128 ↗	SSSE3	intrin.h	<code>__m128i _mm_castps_si128(__m128);</code>
_mm_castsi128_pd ↗	SSSE3	intrin.h	<code>__m128d _mm_castsi128_pd(__m128i);</code>
_mm_castsi128_ps ↗	SSSE3	intrin.h	<code>__m128 _mm_castsi128_ps(__m128i);</code>
_mm_clflush ↗	SSE2	intrin.h	<code>void _mm_clflush(void const *);</code>
_mm_clmulepi64_si128 ↗	PCLMULQDQ	immintrin.h	<code>__m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int);</code>
_mm_cmov_si128	XOP	ammintrin.h	<code>__m128i _mm_cmov_si128(__m128i, __m128i, __m128i);</code>
_mm_cmp_pd ↗	AVX	immintrin.h	<code>__m128d _mm_cmp_pd(__m128d, __m128d, const int);</code>
_mm_cmp_ps ↗	AVX	immintrin.h	<code>__m128 _mm_cmp_ps(__m128, __m128, const int);</code>
_mm_cmp_sd ↗	AVX	immintrin.h	<code>__m128d _mm_cmp_sd(__m128d, __m128d, const int);</code>
_mm_cmp_ss ↗	AVX	immintrin.h	<code>__m128 _mm_cmp_ss(__m128, __m128, const int);</code>
_mm_cmpeq_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_cmpeq_epi16(__m128i, __m128i);</code>
_mm_cmpeq_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_cmpeq_epi32(__m128i, __m128i);</code>
_mm_cmpeq_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cmpeq_epi64(__m128i, __m128i);</code>
_mm_cmpeq_epi8 ↗	SSE2	intrin.h	<code>__m128i _mm_cmpeq_epi8(__m128i, __m128i);</code>
_mm_cmpeq_pd ↗	SSE2	intrin.h	<code>__m128d _mm_cmpeq_pd(__m128d, __m128d);</code>
_mm_cmpeq_pi8 ↗	MMX	mmmintrin.h	<code>__m64 _mm_cmpeq_pi8(__m64, __m64); [宏]</code>
_mm_cmpeq_pi16 ↗	MMX	mmmintrin.h	<code>__m64 _mm_cmpeq_pi16(__m64, __m64); [宏]</code>
_mm_cmpeq_pi32 ↗	MMX	mmmintrin.h	<code>__m64 _mm_cmpeq_pi32(__m64, __m64); [宏]</code>
_mm_cmpeq_ps ↗	SSE	intrin.h	<code>__m128 _mm_cmpeq_ps(__m128, __m128);</code>
_mm_cmpeq_sd ↗	SSE2	intrin.h	<code>__m128d _mm_cmpeq_sd(__m128d, __m128d);</code>
_mm_cmpeq_ss ↗	SSE	intrin.h	<code>__m128 _mm_cmpeq_ss(__m128, __m128);</code>
_mm_cmpestra ↗	SSE42	intrin.h	<code>int _mm_cmpestra(__m128i, int, __m128i, int, const int);</code>
_mm_cmpestrc ↗	SSE42	intrin.h	<code>int _mm_cmpestrc(__m128i, int, __m128i, int, const int);</code>

内部函数名称	技术	标头	函数原型
_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);
_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	mmmintrin.h	_m64 _mm_cmpgt_pi8(_m64, _m64); [宏]
_mm_cmpgt_pi16 ↗	MMX	mmmintrin.h	_m64 _mm_cmpgt_pi16(_m64, _m64); [宏]
_mm_cmpgt_pi32 ↗	MMX	mmmintrin.h	_m64 _mm_cmpgt_pi32(_m64, _m64); [宏]
_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_cmilstra ↗	SSE42	intrin.h	int _mm_cmilstra(_m128i, _m128i, const int);
_mm_cmilstrc ↗	SSE42	intrin.h	int _mm_cmilstrc(_m128i, _m128i, const int);
_mm_cmilstri ↗	SSE42	intrin.h	int _mm_cmilstri(_m128i, _m128i, const int);
_mm_cmilstrm ↗	SSE42	intrin.h	_m128i _mm_cmilstrm(_m128i, _m128i, const int);
_mm_cmilstro ↗	SSE42	intrin.h	int _mm_cmilstro(_m128i, _m128i, const int);
_mm_cmilstrs ↗	SSE42	intrin.h	int _mm_cmilstrs(_m128i, _m128i, const int);
_mm_cmilstrz ↗	SSE42	intrin.h	int _mm_cmilstrz(_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);

内部函数名称	技术	标头	函数原型
<a href="#">_mm_cmple_ss</a>	SSE	intrin.h	<code>__m128 _mm_cmple_ss(__m128, __m128);</code>
<a href="#">_mm_cmplt_epi16</a>	SSE2	intrin.h	<code>__m128i _mm_cmplt_epi16(__m128i, __m128i);</code>
<a href="#">_mm_cmplt_epi32</a>	SSE2	intrin.h	<code>__m128i _mm_cmplt_epi32(__m128i, __m128i);</code>
<a href="#">_mm_cmplt_epi8</a>	SSE2	intrin.h	<code>__m128i _mm_cmplt_epi8(__m128i, __m128i);</code>
<a href="#">_mm_cmplt_pd</a>	SSE2	intrin.h	<code>__m128d _mm_cmplt_pd(__m128d, __m128d);</code>
<a href="#">_mm_cmplt_ps</a>	SSE	intrin.h	<code>__m128 _mm_cmplt_ps(__m128, __m128);</code>
<a href="#">_mm_cmplt_sd</a>	SSE2	intrin.h	<code>__m128d _mm_cmplt_sd(__m128d, __m128d);</code>
<a href="#">_mm_cmplt_ss</a>	SSE	intrin.h	<code>__m128 _mm_cmplt_ss(__m128, __m128);</code>
<a href="#">_mm_cmpneq_pd</a>	SSE2	intrin.h	<code>__m128d _mm_cmpneq_pd(__m128d, __m128d);</code>
<a href="#">_mm_cmpneq_ps</a>	SSE	intrin.h	<code>__m128 _mm_cmpneq_ps(__m128, __m128);</code>
<a href="#">_mm_cmpneq_sd</a>	SSE2	intrin.h	<code>__m128d _mm_cmpneq_sd(__m128d, __m128d);</code>
<a href="#">_mm_cmpneq_ss</a>	SSE	intrin.h	<code>__m128 _mm_cmpneq_ss(__m128, __m128);</code>
<a href="#">_mm_cmpnge_pd</a>	SSE2	intrin.h	<code>__m128d _mm_cmpnge_pd(__m128d, __m128d);</code>
<a href="#">_mm_cmpnge_ps</a>	SSE	intrin.h	<code>__m128 _mm_cmpnge_ps(__m128, __m128);</code>
<a href="#">_mm_cmpnge_sd</a>	SSE2	intrin.h	<code>__m128d _mm_cmpnge_sd(__m128d, __m128d);</code>
<a href="#">_mm_cmpnge_ss</a>	SSE	intrin.h	<code>__m128 _mm_cmpnge_ss(__m128, __m128);</code>
<a href="#">_mm_cmpngt_pd</a>	SSE2	intrin.h	<code>__m128d _mm_cmpngt_pd(__m128d, __m128d);</code>
<a href="#">_mm_cmpngt_ps</a>	SSE	intrin.h	<code>__m128 _mm_cmpngt_ps(__m128, __m128);</code>
<a href="#">_mm_cmpngt_sd</a>	SSE2	intrin.h	<code>__m128d _mm_cmpngt_sd(__m128d, __m128d);</code>
<a href="#">_mm_cmpngt_ss</a>	SSE	intrin.h	<code>__m128 _mm_cmpngt_ss(__m128, __m128);</code>
<a href="#">_mm_cmple_pd</a>	SSE2	intrin.h	<code>__m128d _mm_cmple_pd(__m128d, __m128d);</code>
<a href="#">_mm_cmple_ps</a>	SSE	intrin.h	<code>__m128 _mm_cmple_ps(__m128, __m128);</code>
<a href="#">_mm_cmple_sd</a>	SSE2	intrin.h	<code>__m128d _mm_cmple_sd(__m128d, __m128d);</code>
<a href="#">_mm_cmple_ss</a>	SSE	intrin.h	<code>__m128 _mm_cmple_ss(__m128, __m128);</code>
<a href="#">_mm_cmplt_pd</a>	SSE2	intrin.h	<code>__m128d _mm_cmplt_pd(__m128d, __m128d);</code>
<a href="#">_mm_cmplt_ps</a>	SSE	intrin.h	<code>__m128 _mm_cmplt_ps(__m128, __m128);</code>
<a href="#">_mm_cmplt_sd</a>	SSE2	intrin.h	<code>__m128d _mm_cmplt_sd(__m128d, __m128d);</code>
<a href="#">_mm_cmplt_ss</a>	SSE	intrin.h	<code>__m128 _mm_cmplt_ss(__m128, __m128);</code>
<a href="#">_mm_cmpord_pd</a>	SSE2	intrin.h	<code>__m128d _mm_cmpord_pd(__m128d, __m128d);</code>
<a href="#">_mm_cmpord_ps</a>	SSE	intrin.h	<code>__m128 _mm_cmpord_ps(__m128, __m128);</code>
<a href="#">_mm_cmpord_sd</a>	SSE2	intrin.h	<code>__m128d _mm_cmpord_sd(__m128d, __m128d);</code>
<a href="#">_mm_cmpord_ss</a>	SSE	intrin.h	<code>__m128 _mm_cmpord_ss(__m128, __m128);</code>
<a href="#">_mm_cmputord_pd</a>	SSE2	intrin.h	<code>__m128d _mm_cmputord_pd(__m128d, __m128d);</code>

内部函数名称	技术	标头	函数原型
<a href="#">_mm_cmppunord_ps</a>	SSE	intrin.h	<code>__m128 _mm_cmppunord_ps(__m128, __m128);</code>
<a href="#">_mm_cmppunord_sd</a>	SSE2	intrin.h	<code>__m128d _mm_cmppunord_sd(__m128d, __m128d);</code>
<a href="#">_mm_cmppunord_ss</a>	SSE	intrin.h	<code>__m128 _mm_cmppunord_ss(__m128, __m128);</code>
<a href="#">_mm_com_epi16</a>	XOP	ammintrin.h	<code>__m128i _mm_com_epi16(__m128i, __m128i, int);</code>
<a href="#">_mm_com_epi32</a>	XOP	ammintrin.h	<code>__m128i _mm_com_epi32(__m128i, __m128i, int);</code>
<a href="#">_mm_com_epi64</a>	XOP	ammintrin.h	<code>__m128i _mm_com_epi32(__m128i, __m128i, int);</code>
<a href="#">_mm_com_epi8</a>	XOP	ammintrin.h	<code>__m128i _mm_com_epi8(__m128i, __m128i, int);</code>
<a href="#">_mm_com_epu16</a>	XOP	ammintrin.h	<code>__m128i _mm_com_epu16(__m128i, __m128i, int);</code>
<a href="#">_mm_com_epu32</a>	XOP	ammintrin.h	<code>__m128i _mm_com_epu32(__m128i, __m128i, int);</code>
<a href="#">_mm_com_epu64</a>	XOP	ammintrin.h	<code>__m128i _mm_com_epu32(__m128i, __m128i, int);</code>
<a href="#">_mm_com_epu8</a>	XOP	ammintrin.h	<code>__m128i _mm_com_epu8(__m128i, __m128i, int);</code>
<a href="#">_mm_comieq_sd</a>	SSE2	intrin.h	<code>int _mm_comieq_sd(__m128d, __m128d);</code>
<a href="#">_mm_comieq_ss</a>	SSE	intrin.h	<code>int _mm_comieq_ss(__m128, __m128);</code>
<a href="#">_mm_comige_sd</a>	SSE2	intrin.h	<code>int _mm_comige_sd(__m128d, __m128d);</code>
<a href="#">_mm_comige_ss</a>	SSE	intrin.h	<code>int _mm_comige_ss(__m128, __m128);</code>
<a href="#">_mm_comigt_sd</a>	SSE2	intrin.h	<code>int _mm_comigt_sd(__m128d, __m128d);</code>
<a href="#">_mm_comigt_ss</a>	SSE	intrin.h	<code>int _mm_comigt_ss(__m128, __m128);</code>
<a href="#">_mm_comile_sd</a>	SSE2	intrin.h	<code>int _mm_comile_sd(__m128d, __m128d);</code>
<a href="#">_mm_comile_ss</a>	SSE	intrin.h	<code>int _mm_comile_ss(__m128, __m128);</code>
<a href="#">_mm_comilt_sd</a>	SSE2	intrin.h	<code>int _mm_comilt_sd(__m128d, __m128d);</code>
<a href="#">_mm_comilt_ss</a>	SSE	intrin.h	<code>int _mm_comilt_ss(__m128, __m128);</code>
<a href="#">_mm_comineq_sd</a>	SSE2	intrin.h	<code>int _mm_comineq_sd(__m128d, __m128d);</code>
<a href="#">_mm_comineq_ss</a>	SSE	intrin.h	<code>int _mm_comineq_ss(__m128, __m128);</code>
<a href="#">_mm_crc32_u16</a>	SSE42	intrin.h	<code>unsigned int _mm_crc32_u16(unsigned int, unsigned short);</code>
<a href="#">_mm_crc32_u32</a>	SSE42	intrin.h	<code>unsigned int _mm_crc32_u32(unsigned int, unsigned int);</code>
<a href="#">_mm_crc32_u8</a>	SSE42	intrin.h	<code>unsigned int _mm_crc32_u8(unsigned int, unsigned char);</code>
<a href="#">_mm_cvtpi2ps</a>	SSE	intrin.h	<code>__m128 _mm_cvtpi2ps(__m128, __m64);</code>
<a href="#">_mm_cvtps2pi</a>	SSE	intrin.h	<code>__m64 _mm_cvtps2pi(__m128);</code>
<a href="#">_mm_cvt_si2ss</a>	SSE	intrin.h	<code>__m128 _mm_cvt_si2ss(__m128, int);</code>
<a href="#">_mm_cvt_ss2si</a>	SSE	intrin.h	<code>int _mm_cvt_ss2si(__m128);</code>
<a href="#">_mm_cvtepi16_epi32</a>	SSE41	intrin.h	<code>__m128i _mm_cvtepi16_epi32(__m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_cvtepi16_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepi16_epi64(__m128i);</code>
_mm_cvtepi32_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepi32_epi64(__m128i);</code>
_mm_cvtepi32_pd ↗	SSE2	intrin.h	<code>__m128d _mm_cvtepi32_pd(__m128i);</code>
_mm_cvtepi32_ps ↗	SSE2	intrin.h	<code>__m128 _mm_cvtepi32_ps(__m128i);</code>
_mm_cvtepi8_epi16 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepi8_epi16 (__m128i);</code>
_mm_cvtepi8_epi32 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepi8_epi32 (__m128i);</code>
_mm_cvtepi8_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepi8_epi64 (__m128i);</code>
_mm_cvtepu16_epi32 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepu16_epi32(__m128i);</code>
_mm_cvtepu16_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepu16_epi64(__m128i);</code>
_mm_cvtepu32_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepu32_epi64(__m128i);</code>
_mm_cvtepu8_epi16 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepu8_epi16 (__m128i);</code>
_mm_cvtepu8_epi32 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepu8_epi32 (__m128i);</code>
_mm_cvtepu8_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepu8_epi64 (__m128i);</code>
_mm_cvtpd_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_cvtpd_epi32(__m128d);</code>
_mm_cvtpd_pi32 ↗	SSE2	intrin.h	<code>__m64 _mm_cvtpd_pi32(__m128d);</code>
_mm_cvtpd_ps ↗	SSE2	intrin.h	<code>__m128 _mm_cvtpd_ps(__m128d);</code>
_mm_cvtph_ps ↗	F16C	immintrin.h	<code>__m128 _mm_cvtph_ps(__m128i);</code>
_mm_cvtpi32_pd ↗	SSE2	intrin.h	<code>__m128d _mm_cvtpi32_pd(__m64);</code>
_mm_cvtps_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_cvtps_epi32(__m128);</code>
_mm_cvtps_pd ↗	SSE2	intrin.h	<code>__m128d _mm_cvtps_pd(__m128);</code>
_mm_cvtps_ph ↗	F16C	immintrin.h	<code>__m128i _mm_cvtps_ph(__m128, const int);</code>
_mm_cvtsd_f64 ↗	SSSE3	intrin.h	<code>double _mm_cvtsd_f64(__m128d);</code>
_mm_cvtsd_si32 ↗	SSE2	intrin.h	<code>int _mm_cvtsd_si32(__m128d);</code>
_mm_cvtsd_ss ↗	SSE2	intrin.h	<code>__m128 _mm_cvtsd_ss(__m128, __m128d);</code>
_mm_cvtsi128_si32 ↗	SSE2	intrin.h	<code>int _mm_cvtsi128_si32(__m128i);</code>
_mm_cvtsi32_sd ↗	SSE2	intrin.h	<code>__m128d _mm_cvtsi32_sd(__m128d, int);</code>
_mm_cvtsi32_si128 ↗	SSE2	intrin.h	<code>__m128i _mm_cvtsi32_si128(int);</code>
_mm_cvtsi32_si64 ↗	MMX	mmmintrin.h	<code>__m64 _mm_cvtsi32_si64(int); [宏]</code>
_mm_cvtsi64_si32 ↗	MMX	mmmintrin.h	<code>int _mm_cvtsi64_si32 (__m64); [宏]</code>
_mm_cvtss_f32 ↗	SSSE3	intrin.h	<code>float _mm_cvtss_f32(__m128);</code>
_mm_cvtss_sd ↗	SSE2	intrin.h	<code>__m128d _mm_cvtss_sd(__m128d, __m128);</code>
_mm_cvtt_ps2pi ↗	SSE	intrin.h	<code>__m64 _mm_cvtt_ps2pi(__m128);</code>
_mm_cvtt_ss2si ↗	SSE	intrin.h	<code>int _mm_cvtt_ss2si(__m128);</code>

内部函数名称	技术	标头	函数原型
_mm_cvtpd_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_cvtpd_epi32(__m128d);</code>
_mm_cvtpd_pi32 ↗	SSE2	intrin.h	<code>__m64 _mm_cvtpd_pi32(__m128d);</code>
_mm_cvtps_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_cvtps_epi32(__m128);</code>
_mm_cvtsd_si32 ↗	SSE2	intrin.h	<code>int _mm_cvtsd_si32(__m128d);</code>
_mm_div_pd ↗	SSE2	intrin.h	<code>__m128d _mm_div_pd(__m128d, __m128d);</code>
_mm_div_ps ↗	SSE	intrin.h	<code>__m128 _mm_div_ps(__m128, __m128);</code>
_mm_div_sd ↗	SSE2	intrin.h	<code>__m128d _mm_div_sd(__m128d, __m128d);</code>
_mm_div_ss ↗	SSE	intrin.h	<code>__m128 _mm_div_ss(__m128, __m128);</code>
_mm_dp_pd ↗	SSE41	intrin.h	<code>__m128d _mm_dp_pd(__m128d, __m128d, const int);</code>
_mm_dp_ps ↗	SSE41	intrin.h	<code>__m128 _mm_dp_ps(__m128, __m128, const int);</code>
_mm_empty ↗	MMX	mmmintrin.h	<code>void _mm_empty (void); [宏]</code>
_mm_extract_epi16 ↗	SSE2	intrin.h	<code>int _mm_extract_epi16(__m128i, int);</code>
_mm_extract_epi32 ↗	SSE41	intrin.h	<code>int _mm_extract_epi32(__m128i, const int);</code>
_mm_extract_epi8 ↗	SSE41	intrin.h	<code>int _mm_extract_epi8 (__m128i, const int);</code>
_mm_extract_ps ↗	SSE41	intrin.h	<code>int _mm_extract_ps(__m128, const int);</code>
_mm_extract_si64	SSE4a	intrin.h	<code>__m128i _mm_extract_si64(__m128i, __m128i);</code>
_mm_extracti_si64	SSE4a	intrin.h	<code>__m128i _mm_extracti_si64(__m128i, int, int);</code>
_mm_fmadd_pd ↗	FMA	immintrin.h	<code>__m128d _mm_fmadd_pd (__m128d, __m128d, __m128d);</code>
_mm_fmadd_ps ↗	FMA	immintrin.h	<code>__m128 _mm_fmadd_ps (__m128, __m128, __m128);</code>
_mm_fmadd_sd ↗	FMA	immintrin.h	<code>__m128d _mm_fmadd_sd (__m128d, __m128d, __m128d);</code>
_mm_fmadd_ss ↗	FMA	immintrin.h	<code>__m128 _mm_fmadd_ss (__m128, __m128, __m128);</code>
_mm_fmaddsub_pd ↗	FMA	immintrin.h	<code>__m128d _mm_fmaddsub_pd (__m128d, __m128d, __m128d);</code>
_mm_fmaddsub_ps ↗	FMA	immintrin.h	<code>__m128 _mm_fmaddsub_ps (__m128, __m128, __m128);</code>
_mm_fmsub_pd ↗	FMA	immintrin.h	<code>__m128d _mm_fmsub_pd (__m128d, __m128d, __m128d);</code>
_mm_fmsub_ps ↗	FMA	immintrin.h	<code>__m128 _mm_fmsub_ps (__m128, __m128, __m128);</code>
_mm_fmsub_sd ↗	FMA	immintrin.h	<code>__m128d _mm_fmsub_sd (__m128d, __m128d, __m128d);</code>
_mm_fmsub_ss ↗	FMA	immintrin.h	<code>__m128 _mm_fmsub_ss (__m128, __m128, __m128);</code>
_mm_fmsubadd_pd ↗	FMA	immintrin.h	<code>__m128d _mm_fmsubadd_pd (__m128d, __m128d, __m128d);</code>
_mm_fmsubadd_ps ↗	FMA	immintrin.h	<code>__m128 _mm_fmsubadd_ps (__m128, __m128, __m128);</code>
_mm_fnmadd_pd ↗	FMA	immintrin.h	<code>__m128d _mm_fnmadd_pd (__m128d, __m128d, __m128d);</code>
_mm_fnmadd_ps ↗	FMA	immintrin.h	<code>__m128 _mm_fnmadd_ps (__m128, __m128, __m128);</code>
_mm_fnmadd_sd ↗	FMA	immintrin.h	<code>__m128d _mm_fnmadd_sd (__m128d, __m128d, __m128d);</code>

内部函数名称	技术	标头	函数原型
_mm_fnmadd_ss	FMA	immintrin.h	<code>__m128 _mm_fnmadd_ss (__m128, __m128, __m128);</code>
_mm_fnmsub_pd	FMA	immintrin.h	<code>__m128d _mm_fnmsub_pd (__m128d, __m128d, __m128d);</code>
_mm_fnmsub_ps	FMA	immintrin.h	<code>__m128 _mm_fnmsub_ps (__m128, __m128, __m128);</code>
_mm_fnmsub_sd	FMA	immintrin.h	<code>__m128d _mm_fnmsub_sd (__m128d, __m128d, __m128d);</code>
_mm_fnmsub_ss	FMA	immintrin.h	<code>__m128 _mm_fnmsub_ss (__m128, __m128, __m128);</code>
_mm_frcz_pd	XOP	ammintrin.h	<code>__m128d _mm_frcz_pd(__m128d);</code>
_mm_frcz_ps	XOP	ammintrin.h	<code>__m128 _mm_frcz_ps(__m128);</code>
_mm_frcz_sd	XOP	ammintrin.h	<code>__m128d _mm_frcz_sd(__m128d, __m128d);</code>
_mm_frcz_ss	XOP	ammintrin.h	<code>__m128 _mm_frcz_ss(__m128, __m128);</code>
_mm_getcsr	SSE	intrin.h	<code>unsigned int _mm_getcsr(void);</code>
_mm_hadd_epi16	SSSE3	intrin.h	<code>__m128i _mm_hadd_epi16(__m128i, __m128i);</code>
_mm_hadd_epi32	SSSE3	intrin.h	<code>__m128i _mm_hadd_epi32(__m128i, __m128i);</code>
_mm_hadd_pd	SSE3	intrin.h	<code>__m128d _mm_hadd_pd(__m128d, __m128d);</code>
_mm_hadd_pi16	SSSE3	intrin.h	<code>__m64 _mm_hadd_pi16(__m64, __m64);</code>
_mm_hadd_pi32	SSSE3	intrin.h	<code>__m64 _mm_hadd_pi32(__m64, __m64);</code>
_mm_hadd_ps	SSE3	intrin.h	<code>__m128 _mm_hadd_ps(__m128, __m128);</code>
_mm_hadddd_epi16	XOP	ammintrin.h	<code>__m128i _mm_hadddd_epi16(__m128i);</code>
_mm_hadddd_epi8	XOP	ammintrin.h	<code>__m128i _mm_hadddd_epi8(__m128i);</code>
_mm_hadddd_epu16	XOP	ammintrin.h	<code>__m128i _mm_hadddd_epu16(__m128i);</code>
_mm_hadddd_epu8	XOP	ammintrin.h	<code>__m128i _mm_hadddd_epu8(__m128i);</code>
_mm_haddq_epi16	XOP	ammintrin.h	<code>__m128i _mm_haddq_epi16(__m128i);</code>
_mm_haddq_epi32	XOP	ammintrin.h	<code>__m128i _mm_haddq_epi32(__m128i);</code>
_mm_haddq_epi8	XOP	ammintrin.h	<code>__m128i _mm_haddq_epi8(__m128i);</code>
_mm_haddq_epu16	XOP	ammintrin.h	<code>__m128i _mm_haddq_epu16(__m128i);</code>
_mm_haddq_epu32	XOP	ammintrin.h	<code>__m128i _mm_haddq_epu32(__m128i);</code>
_mm_haddq_epu8	XOP	ammintrin.h	<code>__m128i _mm_haddq_epu8(__m128i);</code>
_mm_hadds_epi16	SSSE3	intrin.h	<code>__m128i _mm_hadds_epi16(__m128i, __m128i);</code>
_mm_hadds_pi16	SSSE3	intrin.h	<code>__m64 _mm_hadds_pi16(__m64, __m64);</code>
_mm_haddw_epi8	XOP	ammintrin.h	<code>__m128i _mm_haddw_epi8(__m128i);</code>
_mm_haddw_epu8	XOP	ammintrin.h	<code>__m128i _mm_haddw_epu8(__m128i);</code>
_mm_hsub_epi16	SSSE3	intrin.h	<code>__m128i _mm_hsub_epi16(__m128i, __m128i);</code>
_mm_hsub_epi32	SSSE3	intrin.h	<code>__m128i _mm_hsub_epi32(__m128i, __m128i);</code>
_mm_hsub_pd	SSE3	intrin.h	<code>__m128d _mm_hsub_pd(__m128d, __m128d);</code>

内部函数名称	技术	标头	函数原型
_mm_hsub_pi16 ↗	SSSE3	intrin.h	<code>__m64 _mm_hsub_pi16(__m64, __m64);</code>
_mm_hsub_pi32 ↗	SSSE3	intrin.h	<code>__m64 _mm_hsub_pi32(__m64, __m64);</code>
_mm_hsub_ps ↗	SSE3	intrin.h	<code>__m128 _mm_hsub_ps(__m128, __m128);</code>
_mm_hsubd_epi16	XOP	ammintrin.h	<code>__m128i _mm_hsubd_epi16(__m128i);</code>
_mm_hsubq_epi32	XOP	ammintrin.h	<code>__m128i _mm_hsubq_epi32(__m128i);</code>
_mm_hsubs_epi16 ↗	SSSE3	intrin.h	<code>__m128i _mm_hsubs_epi16(__m128i, __m128i);</code>
_mm_hsubs_pi16 ↗	SSSE3	intrin.h	<code>__m64 _mm_hsubs_pi16(__m64, __m64);</code>
_mm_hsubw_epi8	XOP	ammintrin.h	<code>__m128i _mm_hsubw_epi8(__m128i);</code>
_mm_i32gather_epi32 ↗	AVX2	immintrin.h	<code>__m128i _mm_i32gather_epi32(int const *, __m128i, const int);</code>
_mm_i32gather_epi64 ↗	AVX2	immintrin.h	<code>__m128i _mm_i32gather_epi64(__int64 const *, __m128i, const int);</code>
_mm_i32gather_pd ↗	AVX2	immintrin.h	<code>__m128d _mm_i32gather_pd(double const *, __m128i, const int);</code>
_mm_i32gather_ps ↗	AVX2	immintrin.h	<code>__m128 _mm_i32gather_ps(float const *, __m128i, const int);</code>
_mm_i64gather_epi32 ↗	AVX2	immintrin.h	<code>__m128i _mm_i64gather_epi32(int const *, __m128i, const int);</code>
_mm_i64gather_epi64 ↗	AVX2	immintrin.h	<code>__m128i _mm_i64gather_epi64(__int64 const *, __m128i, const int);</code>
_mm_i64gather_pd ↗	AVX2	immintrin.h	<code>__m128d _mm_i64gather_pd(double const *, __m128i, const int);</code>
_mm_i64gather_ps ↗	AVX2	immintrin.h	<code>__m128 _mm_i64gather_ps(float const *, __m128i, const int);</code>
_mm_insert_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_insert_epi16(__m128i, int, int);</code>
_mm_insert_epi32 ↗	SSE41	intrin.h	<code>__m128i _mm_insert_epi32(__m128i, int, const int);</code>
_mm_insert_epi8 ↗	SSE41	intrin.h	<code>__m128i _mm_insert_epi8 (__m128i, int, const int);</code>
_mm_insert_ps ↗	SSE41	intrin.h	<code>__m128 _mm_insert_ps(__m128, __m128, const int);</code>
_mm_insert_si64	SSE4a	intrin.h	<code>__m128i _mm_insert_si64(__m128i, __m128i);</code>
_mm_inserti_si64	SSE4a	intrin.h	<code>__m128i _mm_inserti_si64(__m128i, __m128i, int, int);</code>
_mm_lddqu_si128 ↗	SSE3	intrin.h	<code>__m128i _mm_lddqu_si128(__m128i const*);</code>
_mm_lfence ↗	SSE2	intrin.h	<code>void _mm_lfence(void);</code>
_mm_load_pd ↗	SSE2	intrin.h	<code>__m128d _mm_load_pd(double*);</code>
_mm_load_ps ↗	SSE	intrin.h	<code>__m128 _mm_load_ps(float*);</code>
_mm_load_ps1 ↗	SSE	intrin.h	<code>__m128 _mm_load_ps1(float*);</code>
_mm_load_sd ↗	SSE2	intrin.h	<code>__m128d _mm_load_sd(double*);</code>

内部函数名称	技术	标头	函数原型
_mm_load_si128	SSE2	intrin.h	<code>__m128i _mm_load_si128(__m128i*);</code>
_mm_load_ss	SSE	intrin.h	<code>__m128 _mm_load_ss(float*);</code>
_mm_load1_pd	SSE2	intrin.h	<code>__m128d _mm_load1_pd(double*);</code>
_mm_loaddup_pd	SSE3	intrin.h	<code>__m128d _mm_loaddup_pd(double const*);</code>
_mm_loadh_pd	SSE2	intrin.h	<code>__m128d _mm_loadh_pd(__m128d, double*);</code>
_mm_loadh_pi	SSE	intrin.h	<code>__m128 _mm_loadh_pi(__m128, __m64*);</code>
_mm_loadl_epi64	SSE2	intrin.h	<code>__m128i _mm_loadl_epi64(__m128i*);</code>
_mm_loadl_pd	SSE2	intrin.h	<code>__m128d _mm_loadl_pd(__m128d, double*);</code>
_mm_loadl_pi	SSE	intrin.h	<code>__m128 _mm_loadl_pi(__m128, __m64*);</code>
_mm_loadr_pd	SSE2	intrin.h	<code>__m128d _mm_loadr_pd(double*);</code>
_mm_loadr_ps	SSE	intrin.h	<code>__m128 _mm_loadr_ps(float*);</code>
_mm_loadu_pd	SSE2	intrin.h	<code>__m128d _mm_loadu_pd(double*);</code>
_mm_loadu_ps	SSE	intrin.h	<code>__m128 _mm_loadu_ps(float*);</code>
_mm_loadu_si128	SSE2	intrin.h	<code>__m128i _mm_loadu_si128(__m128i*);</code>
_mm_macc_epi16	XOP	ammintrin.h	<code>__m128i _mm_macc_epi16(__m128i, __m128i, __m128i);</code>
_mm_macc_epi32	XOP	ammintrin.h	<code>__m128i _mm_macc_epi32(__m128i, __m128i, __m128i);</code>
_mm_macc_pd	FMA4	ammintrin.h	<code>__m128d _mm_macc_pd(__m128d, __m128d, __m128d);</code>
_mm_macc_ps	FMA4	ammintrin.h	<code>__m128 _mm_macc_ps(__m128, __m128, __m128);</code>
_mm_macc_sd	FMA4	ammintrin.h	<code>__m128d _mm_macc_sd(__m128d, __m128d, __m128d);</code>
_mm_macc_ss	FMA4	ammintrin.h	<code>__m128 _mm_macc_ss(__m128, __m128, __m128);</code>
_mm_maccd_epi16	XOP	ammintrin.h	<code>__m128i _mm_maccd_epi16(__m128i, __m128i, __m128i);</code>
_mm_macchi_epi32	XOP	ammintrin.h	<code>__m128i _mm_macchi_epi32(__m128i, __m128i, __m128i);</code>
_mm_macclo_epi32	XOP	ammintrin.h	<code>__m128i _mm_macclo_epi32(__m128i, __m128i, __m128i);</code>
_mm_maccs_epi16	XOP	ammintrin.h	<code>__m128i _mm_maccs_epi16(__m128i, __m128i, __m128i);</code>
_mm_maccs_epi32	XOP	ammintrin.h	<code>__m128i _mm_maccs_epi32(__m128i, __m128i, __m128i);</code>
_mm_maccsd_epi16	XOP	ammintrin.h	<code>__m128i _mm_maccsd_epi16(__m128i, __m128i, __m128i);</code>
_mm_maccshi_epi32	XOP	ammintrin.h	<code>__m128i _mm_maccshi_epi32(__m128i, __m128i, __m128i);</code>
_mm_maccslo_epi32	XOP	ammintrin.h	<code>__m128i _mm_maccslo_epi32(__m128i, __m128i, __m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_madd_epi16	SSE2	intrin.h	<code>__m128i _mm_madd_epi16(__m128i, __m128i);</code>
_mm_madd_pi16	MMX	mmmintrin.h	<code>__m64 _mm_madd_pi16(__m64, __m64); [宏]</code>
_mm_maddd_epi16	XOP	ammintrin.h	<code>__m128i _mm_maddd_epi16(__m128i, __m128i, __m128i);</code>
_mm_maddsd_epi16	XOP	ammintrin.h	<code>__m128i _mm_maddsd_epi16(__m128i, __m128i, __m128i);</code>
_mm_maddsub_pd	FMA4	ammintrin.h	<code>__m128d _mm_maddsub_pd(__m128d, __m128d, __m128d);</code>
_mm_maddsub_ps	FMA4	ammintrin.h	<code>__m128 _mm_maddsub_ps(__m128, __m128, __m128);</code>
_mm_maddubs_epi16	SSSE3	intrin.h	<code>__m128i _mm_maddubs_epi16(__m128i, __m128i);</code>
_mm_maddubs_pi16	SSSE3	intrin.h	<code>__m64 _mm_maddubs_pi16(__m64, __m64);</code>
_mm_mask_i32gather_epi32	AVX2	immintrin.h	<code>__m128i _mm_mask_i32gather_epi32(__m128i, int const *, __m128i, __m128i, const int);</code>
_mm_mask_i32gather_epi64	AVX2	immintrin.h	<code>__m128i _mm_mask_i32gather_epi64(__m128i, __int64 const *, __m128i, __m128i, const int);</code>
_mm_mask_i32gather_pd	AVX2	immintrin.h	<code>__m128d _mm_mask_i32gather_pd(__m128d, double const *, __m128i, __m128d, const int);</code>
_mm_mask_i32gather_ps	AVX2	immintrin.h	<code>__m128 _mm_mask_i32gather_ps(__m128, float const *, __m128i, __m128, const int);</code>
_mm_mask_i64gather_epi32	AVX2	immintrin.h	<code>__m128i _mm_mask_i64gather_epi32(__m128i, int const *, __m128i, __m128i, const int);</code>
_mm_mask_i64gather_epi64	AVX2	immintrin.h	<code>__m128i _mm_mask_i64gather_epi64(__m128i, __int64 const *, __m128i, __m128i, const int);</code>
_mm_mask_i64gather_pd	AVX2	immintrin.h	<code>__m128d _mm_mask_i64gather_pd(__m128d, double const *, __m128i, __m128d, const int);</code>
_mm_mask_i64gather_ps	AVX2	immintrin.h	<code>__m128 _mm_mask_i64gather_ps(__m128, float const *, __m128i, __m128, const int);</code>
_mm_maskload_epi32	AVX2	immintrin.h	<code>__m128i _mm_maskload_epi32(int const *, __m128i);</code>
_mm_maskload_epi64	AVX2	immintrin.h	<code>__m128i _mm_maskload_epi64(__int64 const *, __m128i);</code>
_mm_maskload_pd	AVX	immintrin.h	<code>__m128d _mm_maskload_pd(double const *, __m128i);</code>
_mm_maskload_ps	AVX	immintrin.h	<code>__m128 _mm_maskload_ps(float const *, __m128i);</code>
_mm_maskmoveu_si128	SSE2	intrin.h	<code>void _mm_maskmoveu_si128(__m128i, __m128i, char*);</code>
_mm_maskstore_epi32	AVX2	immintrin.h	<code>void _mm_maskstore_epi32(int *, __m128i, __m128i);</code>
_mm_maskstore_epi64	AVX2	immintrin.h	<code>void _mm_maskstore_epi64(__int64 *, __m128i, __m128i);</code>
_mm_maskstore_pd	AVX	immintrin.h	<code>void _mm_maskstore_pd(double *, __m128i, __m128d);</code>
_mm_maskstore_ps	AVX	immintrin.h	<code>void _mm_maskstore_ps(float *, __m128i, __m128);</code>
_mm_max_epi16	SSE2	intrin.h	<code>__m128i _mm_max_epi16(__m128i, __m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_max_epi32 ↗	SSE41	intrin.h	<code>__m128i _mm_max_epi32(__m128i, __m128i);</code>
_mm_max_epi8 ↗	SSE41	intrin.h	<code>__m128i _mm_max_epi8 (__m128i, __m128i);</code>
_mm_max_epu16 ↗	SSE41	intrin.h	<code>__m128i _mm_max_epu16(__m128i, __m128i);</code>
_mm_max_epu32 ↗	SSE41	intrin.h	<code>__m128i _mm_max_epu32(__m128i, __m128i);</code>
_mm_max_epu8 ↗	SSE2	intrin.h	<code>__m128i _mm_max_epu8(__m128i, __m128i);</code>
_mm_max_pd ↗	SSE2	intrin.h	<code>__m128d _mm_max_pd(__m128d, __m128d);</code>
_mm_max_ps ↗	SSE	intrin.h	<code>__m128 _mm_max_ps(__m128, __m128);</code>
_mm_max_sd ↗	SSE2	intrin.h	<code>__m128d _mm_max_sd(__m128d, __m128d);</code>
_mm_max_ss ↗	SSE	intrin.h	<code>__m128 _mm_max_ss(__m128, __m128);</code>
_mm_mfence ↗	SSE2	intrin.h	<code>void _mm_mfence(void);</code>
_mm_min_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_min_epi16(__m128i, __m128i);</code>
_mm_min_epi32 ↗	SSE41	intrin.h	<code>__m128i _mm_min_epi32(__m128i, __m128i);</code>
_mm_min_epi8 ↗	SSE41	intrin.h	<code>__m128i _mm_min_epi8 (__m128i, __m128i);</code>
_mm_min_epu16 ↗	SSE41	intrin.h	<code>__m128i _mm_min_epu16(__m128i, __m128i);</code>
_mm_min_epu32 ↗	SSE41	intrin.h	<code>__m128i _mm_min_epu32(__m128i, __m128i);</code>
_mm_min_epu8 ↗	SSE2	intrin.h	<code>__m128i _mm_min_epu8(__m128i, __m128i);</code>
_mm_min_pd ↗	SSE2	intrin.h	<code>__m128d _mm_min_pd(__m128d, __m128d);</code>
_mm_min_ps ↗	SSE	intrin.h	<code>__m128 _mm_min_ps(__m128, __m128);</code>
_mm_min_sd ↗	SSE2	intrin.h	<code>__m128d _mm_min_sd(__m128d, __m128d);</code>
_mm_min_ss ↗	SSE	intrin.h	<code>__m128 _mm_min_ss(__m128, __m128);</code>
_mm_minpos_epu16 ↗	SSE41	intrin.h	<code>__m128i _mm_minpos_epu16(__m128i);</code>
_mm_monitor ↗	SSE3	intrin.h	<code>void _mm_monitor(void const*, unsigned int, unsigned int);</code>
_mm_move_epi64 ↗	SSE2	intrin.h	<code>__m128i _mm_move_epi64(__m128i);</code>
_mm_move_sd ↗	SSE2	intrin.h	<code>__m128d _mm_move_sd(__m128d, __m128d);</code>
_mm_move_ss ↗	SSE	intrin.h	<code>__m128 _mm_move_ss(__m128, __m128);</code>
_mm_movedup_pd ↗	SSE3	intrin.h	<code>__m128d _mm_movedup_pd(__m128d);</code>
_mm_movehdup_ps ↗	SSE3	intrin.h	<code>__m128 _mm_movehdup_ps(__m128);</code>
_mm_movehl_ps ↗	SSE	intrin.h	<code>__m128 _mm_movehl_ps(__m128, __m128);</code>
_mm_moveldup_ps ↗	SSE3	intrin.h	<code>__m128 _mm_moveldup_ps(__m128);</code>
_mm_movelh_ps ↗	SSE	intrin.h	<code>__m128 _mm_movelh_ps(__m128, __m128);</code>
_mm_movemask_epi8 ↗	SSE2	intrin.h	<code>int _mm_movemask_epi8(__m128i);</code>
_mm_movemask_pd ↗	SSE2	intrin.h	<code>int _mm_movemask_pd(__m128d);</code>

内部函数名称	技术	标头	函数原型
_mm_movemask_ps ↗	SSE	intrin.h	int _mm_movemask_ps(__m128);
_mm_movepi64_pi64 ↗	SSE2	intrin.h	__m64 _mm_movepi64_pi64(__m128i);
_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);
_mm_mulhi_pi16 ↗	MMX	mmmintrin.h	__m64 __mm_mulhi_pi16(__m64, __m64); [宏]
_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	mmmintrin.h	__m64 __mm_mullo_pi16(__m64, __m64); [宏]
_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);

内部函数名称	技术	标头	函数原型
<code>_mm_nmsub_sd</code>	FMA4	ammintrin.h	<code>__m128d _mm_nmsub_sd(__m128d, __m128d, __m128d);</code>
<code>_mm_nmsub_ss</code>	FMA4	ammintrin.h	<code>__m128 _mm_nmsub_ss(__m128, __m128, __m128);</code>
<code>_mm_or_pd</code>	SSE2	intrin.h	<code>__m128d _mm_or_pd(__m128d, __m128d);</code>
<code>_mm_or_ps</code>	SSE	intrin.h	<code>__m128 _mm_or_ps(__m128, __m128);</code>
<code>_mm_or_si64</code>	MMX	mmmintrin.h	<code>__m64 _mm_or_si64(__m64, __m64); [宏]</code>
<code>_mm_or_si128</code>	SSE2	intrin.h	<code>__m128i _mm_or_si128(__m128i, __m128i);</code>
<code>_mm_packs_epi16</code>	SSE2	intrin.h	<code>__m128i _mm_packs_epi16(__m128i, __m128i);</code>
<code>_mm_packs_epi32</code>	SSE2	intrin.h	<code>__m128i _mm_packs_epi32(__m128i, __m128i);</code>
<code>_mm_packs_pi16</code>	MMX	mmmintrin.h	<code>__m64 _mm_packs_pi16 (__m64, __m64); [宏]</code>
<code>_mm_packs_pi32</code>	MMX	mmmintrin.h	<code>__m64 _mm_packs_pi32 (__m64, __m64); [宏]</code>
<code>_mm_packs_pu16</code>	MMX	mmmintrin.h	<code>__m64 _mm_packs_pu16 (__m64, __m64); [宏]</code>
<code>_mm_packus_epi16</code>	SSE2	intrin.h	<code>__m128i _mm_packus_epi16(__m128i, __m128i);</code>
<code>_mm_packus_epi32</code>	SSE41	intrin.h	<code>__m128i _mm_packus_epi32(__m128i, __m128i);</code>
<code>_mm_pause</code>	SSE2	intrin.h	<code>void _mm_pause(void);</code>
<code>_mm_perm_epi8</code>	XOP	ammintrin.h	<code>__m128i _mm_perm_epi8(__m128i, __m128i, __m128i);</code>
<code>_mm_permute_pd</code>	AVX	immintrin.h	<code>__m128d _mm_permute_pd(__m128d, int);</code>
<code>_mm_permute_ps</code>	AVX	immintrin.h	<code>__m128 _mm_permute_ps(__m128, int);</code>
<code>_mm_permute2_pd</code>	XOP	ammintrin.h	<code>__m128d _mm_permute2_pd(__m128d, __m128d, __m128i, int);</code>
<code>_mm_permute2_ps</code>	XOP	ammintrin.h	<code>__m128 _mm_permute2_ps(__m128, __m128, __m128i, int);</code>
<code>_mm_permutevar_pd</code>	AVX	immintrin.h	<code>__m128d _mm_permutevar_pd(__m128d, __m128i);</code>
<code>_mm_permutevar_ps</code>	AVX	immintrin.h	<code>__m128 _mm_permutevar_ps(__m128, __m128i);</code>
<code>_mm_popcnt_u32</code>	POPCNT	intrin.h	<code>int _mm_popcnt_u32(unsigned int);</code>
<code>_mm_prefetch</code>	SSE	intrin.h	<code>void _mm_prefetch(char*, int);</code>
<code>_mm_rcp_ps</code>	SSE	intrin.h	<code>__m128 _mm_rcp_ps(__m128);</code>
<code>_mm_rcp_ss</code>	SSE	intrin.h	<code>__m128 _mm_rcp_ss(__m128);</code>
<code>_mm_rot_epi16</code>	XOP	ammintrin.h	<code>__m128i _mm_rot_epi16(__m128i, __m128i);</code>
<code>_mm_rot_epi32</code>	XOP	ammintrin.h	<code>__m128i _mm_rot_epi32(__m128i, __m128i);</code>
<code>_mm_rot_epi64</code>	XOP	ammintrin.h	<code>__m128i _mm_rot_epi64(__m128i, __m128i);</code>
<code>_mm_rot_epi8</code>	XOP	ammintrin.h	<code>__m128i _mm_rot_epi8(__m128i, __m128i);</code>
<code>_mm_roti_epi16</code>	XOP	ammintrin.h	<code>__m128i _mm_roti_epi16(__m128i, int);</code>
<code>_mm_roti_epi32</code>	XOP	ammintrin.h	<code>__m128i _mm_roti_epi32(__m128i, int);</code>
<code>_mm_roti_epi64</code>	XOP	ammintrin.h	<code>__m128i _mm_roti_epi64(__m128i, int);</code>

内部函数名称	技术	标头	函数原型
_mm_roti_epi8	XOP	ammintrin.h	<code>__m128i _mm_roti_epi8(__m128i, int);</code>
_mm_round_pd	SSE41	intrin.h	<code>__m128d _mm_round_pd(__m128d, const int);</code>
_mm_round_ps	SSE41	intrin.h	<code>__m128 _mm_round_ps(__m128, const int);</code>
_mm_round_sd	SSE41	intrin.h	<code>__m128d _mm_round_sd(__m128d, __m128d, const int);</code>
_mm_round_ss	SSE41	intrin.h	<code>__m128 _mm_round_ss(__m128, __m128, const int);</code>
_mm_rsqrt_ps	SSE	intrin.h	<code>__m128 _mm_rsqrt_ps(__m128);</code>
_mm_rsqrt_ss	SSE	intrin.h	<code>__m128 _mm_rsqrt_ss(__m128);</code>
_mm_sad_epu8	SSE2	intrin.h	<code>__m128i _mm_sad_epu8(__m128i, __m128i);</code>
_mm_set_epi16	SSE2	intrin.h	<code>__m128i _mm_set_epi16(short, short, short, short, short, short, short, short);</code>
_mm_set_epi32	SSE2	intrin.h	<code>__m128i _mm_set_epi32(int, int, int, int);</code>
_mm_set_epi64	SSE2	intrin.h	<code>__m128i _mm_set_epi64(__m64, __m64);</code>
_mm_set_epi8	SSE2	intrin.h	<code>__m128i _mm_set_epi8(char, char, char, char, char, char, char, char, char, char, char);</code>
_mm_set_pd	SSE2	intrin.h	<code>__m128d _mm_set_pd(double, double);</code>
_mm_set_pi16	MMX	intrin.h	<code>__m64 _mm_set_pi16(short, short, short, short);</code>
_mm_set_pi32	MMX	intrin.h	<code>__m64 _mm_set_pi32(int, int);</code>
_mm_set_pi8	MMX	intrin.h	<code>__m64 _mm_set_pi8(char, char, char, char, char, char, char, char);</code>
_mm_set_ps	SSE	intrin.h	<code>__m128 _mm_set_ps(float, float, float, float);</code>
_mm_set_ps1	SSE	intrin.h	<code>__m128 _mm_set_ps1(float);</code>
_mm_set_sd	SSE2	intrin.h	<code>__m128d _mm_set_sd(double);</code>
_mm_set_ss	SSE	intrin.h	<code>__m128 _mm_set_ss(float);</code>
_mm_set1_epi16	SSE2	intrin.h	<code>__m128i _mm_set1_epi16(short);</code>
_mm_set1_epi32	SSE2	intrin.h	<code>__m128i _mm_set1_epi32(int);</code>
_mm_set1_epi64	SSE2	intrin.h	<code>__m128i _mm_set1_epi64(__m64);</code>
_mm_set1_epi8	SSE2	intrin.h	<code>__m128i _mm_set1_epi8(char);</code>
_mm_set1_pd	SSE2	intrin.h	<code>__m128d _mm_set1_pd(double);</code>
_mm_set1_pi16	MMX	intrin.h	<code>__m64 _mm_set1_pi16(short);</code>
_mm_set1_pi32	MMX	intrin.h	<code>__m64 _mm_set1_pi32(int);</code>
_mm_set1_pi8	MMX	intrin.h	<code>__m64 _mm_set1_pi8(char);</code>
_mm_setscsr	SSE	intrin.h	<code>void _mm_setscsr(unsigned int);</code>
_mm_setl_epi64	SSE2	intrin.h	<code>__m128i _mm_setl_epi64(__m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_setr_epi16	SSE2	intrin.h	<code>__m128i _mm_setr_epi16(short, short, short, short, short, short, short, short);</code>
_mm_setr_epi32	SSE2	intrin.h	<code>__m128i _mm_setr_epi32(int, int, int, int);</code>
_mm_setr_epi64	SSE2	intrin.h	<code>__m128i _mm_setr_epi64(__m64, __m64);</code>
_mm_setr_epi8	SSE2	intrin.h	<code>__m128i _mm_setr_epi8(char, char, char);</code>
_mm_setr_pd	SSE2	intrin.h	<code>__m128d _mm_setr_pd(double, double);</code>
_mm_setr_pi16	MMX	intrin.h	<code>__m64 _mm_setr_pi16(short, short, short, short);</code>
_mm_setr_pi32	MMX	intrin.h	<code>__m64 _mm_setr_pi32(int, int);</code>
_mm_setr_pi8	MMX	intrin.h	<code>__m64 _mm_setr_pi8(char, char, char, char, char, char);</code>
_mm_setr_ps	SSE	intrin.h	<code>__m128 _mm_setr_ps(float, float, float, float);</code>
_mm_setzero_pd	SSE2	intrin.h	<code>__m128d _mm_setzero_pd(void);</code>
_mm_setzero_ps	SSE	intrin.h	<code>__m128 _mm_setzero_ps(void);</code>
_mm_setzero_si128	SSE2	intrin.h	<code>__m128i _mm_setzero_si128(void);</code>
_mm_setzero_si64	MMX	intrin.h	<code>__m64 _mm_setzero_si64(void);</code>
_mm_sfence	SSE	intrin.h	<code>void _mm_sfence(void);</code>
_mm_sha_epi16	XOP	ammintrin.h	<code>__m128i _mm_sha_epi16(__m128i, __m128i);</code>
_mm_sha_epi32	XOP	ammintrin.h	<code>__m128i _mm_sha_epi32(__m128i, __m128i);</code>
_mm_sha_epi64	XOP	ammintrin.h	<code>__m128i _mm_sha_epi64(__m128i, __m128i);</code>
_mm_sha_epi8	XOP	ammintrin.h	<code>__m128i _mm_sha_epi8(__m128i, __m128i);</code>
_mm_shl_epi16	XOP	ammintrin.h	<code>__m128i _mm_shl_epi16(__m128i, __m128i);</code>
_mm_shl_epi32	XOP	ammintrin.h	<code>__m128i _mm_shl_epi32(__m128i, __m128i);</code>
_mm_shl_epi64	XOP	ammintrin.h	<code>__m128i _mm_shl_epi64(__m128i, __m128i);</code>
_mm_shl_epi8	XOP	ammintrin.h	<code>__m128i _mm_shl_epi8(__m128i, __m128i);</code>
_mm_shuffle_epi32	SSE2	intrin.h	<code>__m128i _mm_shuffle_epi32(__m128i, int);</code>
_mm_shuffle_epi8	SSSE3	intrin.h	<code>__m128i _mm_shuffle_epi8(__m128i, __m128i);</code>
_mm_shuffle_pd	SSE2	intrin.h	<code>__m128d _mm_shuffle_pd(__m128d, __m128d, int);</code>
_mm_shuffle_pi8	SSSE3	intrin.h	<code>__m64 _mm_shuffle_pi8(__m64, __m64);</code>
_mm_shuffle_ps	SSE	intrin.h	<code>__m128 _mm_shuffle_ps(__m128, __m128, unsigned int);</code>
_mm_shufflehi_epi16	SSE2	intrin.h	<code>__m128i _mm_shufflehi_epi16(__m128i, int);</code>
_mm_shufflelo_epi16	SSE2	intrin.h	<code>__m128i _mm_shufflelo_epi16(__m128i, int);</code>
_mm_sign_epi16	SSSE3	intrin.h	<code>__m128i _mm_sign_epi16(__m128i, __m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_sign_epi32 ↗	SSSE3	intrin.h	<code>__m128i _mm_sign_epi32(__m128i, __m128i);</code>
_mm_sign_epi8 ↗	SSSE3	intrin.h	<code>__m128i _mm_sign_epi8(__m128i, __m128i);</code>
_mm_sign_pi16 ↗	SSSE3	intrin.h	<code>__m64 _mm_sign_pi16(__m64, __m64);</code>
_mm_sign_pi32 ↗	SSSE3	intrin.h	<code>__m64 _mm_sign_pi32(__m64, __m64);</code>
_mm_sign_pi8 ↗	SSSE3	intrin.h	<code>__m64 _mm_sign_pi8(__m64, __m64);</code>
_mm_sll_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_sll_epi16(__m128i, __m128i);</code>
_mm_sll_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_sll_epi32(__m128i, __m128i);</code>
_mm_sll_epi64 ↗	SSE2	intrin.h	<code>__m128i _mm_sll_epi64(__m128i, __m128i);</code>
_mm_sll_pi16 ↗	MMX	mmmintrin.h	<code>__m64 _mm_sll_pi16(__m64, __m64); [宏]</code>
_mm_sll_pi32 ↗	MMX	mmmintrin.h	<code>__m64 _mm_sll_pi32(__m64, __m64); [宏]</code>
_mm_sll_si64 ↗	MMX	mmmintrin.h	<code>__m64 _mm_sll_si64(__m64, __m64); [宏]</code>
_mm_slli_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_slli_epi16(__m128i, int);</code>
_mm_slli_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_slli_epi32(__m128i, int);</code>
_mm_slli_epi64 ↗	SSE2	intrin.h	<code>__m128i _mm_slli_epi64(__m128i, int);</code>
_mm_slli_pi16 ↗	MMX	mmmintrin.h	<code>__m64 _mm_slli_pi16(__m64, int); [宏]</code>
_mm_slli_pi32 ↗	MMX	mmmintrin.h	<code>__m64 _mm_slli_pi32(__m64, int); [宏]</code>
_mm_slli_si64 ↗	MMX	mmmintrin.h	<code>__m64 _mm_slli_si64(__m64, int); [宏]</code>
_mm_slli_si128 ↗	SSE2	intrin.h	<code>__m128i _mm_slli_si128(__m128i, int);</code>
_mm_sllv_epi32 ↗	AVX2	immintrin.h	<code>__m128i _mm_sllv_epi32(__m128i, __m128i);</code>
_mm_sllv_epi64 ↗	AVX2	immintrin.h	<code>__m128i _mm_sllv_epi64(__m128i, __m128i);</code>
_mm_sqrt_pd ↗	SSE2	intrin.h	<code>__m128d _mm_sqrt_pd(__m128d);</code>
_mm_sqrt_ps ↗	SSE	intrin.h	<code>__m128 _mm_sqrt_ps(__m128);</code>
_mm_sqrt_sd ↗	SSE2	intrin.h	<code>__m128d _mm_sqrt_sd(__m128d, __m128d);</code>
_mm_sqrt_ss ↗	SSE	intrin.h	<code>__m128 _mm_sqrt_ss(__m128);</code>
_mm_sra_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_sra_epi16(__m128i, __m128i);</code>
_mm_sra_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_sra_epi32(__m128i, __m128i);</code>
_mm_sra_pi16 ↗	MMX	mmmintrin.h	<code>__m64 _mm_sra_pi16(__m64, __m64); [宏]</code>
_mm_sra_pi32 ↗	MMX	mmmintrin.h	<code>__m64 _mm_sra_pi32(__m64, __m64); [宏]</code>
_mm_srai_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_srai_epi16(__m128i, int);</code>
_mm_srai_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_srai_epi32(__m128i, int);</code>
_mm_srai_pi16 ↗	MMX	mmmintrin.h	<code>__m64 _mm_srai_pi16(__m64, int); [宏]</code>
_mm_srai_pi32 ↗	MMX	mmmintrin.h	<code>__m64 _mm_srai_pi32(__m64, int); [宏]</code>
_mm_srav_epi32 ↗	AVX2	immintrin.h	<code>__m128i _mm_srav_epi32(__m128i, __m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_srl_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_srl_epi16(__m128i, __m128i);</code>
_mm_srl_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_srl_epi32(__m128i, __m128i);</code>
_mm_srl_epi64 ↗	SSE2	intrin.h	<code>__m128i _mm_srl_epi64(__m128i, __m128i);</code>
_mm_srl_pi16 ↗	MMX	mmmintrin.h	<code>__m64 _mm_srl_pi16(__m64, __m64); [宏]</code>
_mm_srl_pi32 ↗	MMX	mmmintrin.h	<code>__m64 _mm_srl_pi32(__m64, __m64); [宏]</code>
_mm_srl_si64 ↗	MMX	mmmintrin.h	<code>__m64 _mm_srl_si64(__m64, __m64); [宏]</code>
_mm_srli_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_srli_epi16(__m128i, int);</code>
_mm_srli_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_srli_epi32(__m128i, int);</code>
_mm_srli_epi64 ↗	SSE2	intrin.h	<code>__m128i _mm_srli_epi64(__m128i, int);</code>
_mm_srli_pi16 ↗	MMX	mmmintrin.h	<code>__m64 _mm_srli_pi16(__m64, int); [宏]</code>
_mm_srli_pi32 ↗	MMX	mmmintrin.h	<code>__m64 _mm_srli_pi32(__m64, int); [宏]</code>
_mm_srli_si64 ↗	MMX	mmmintrin.h	<code>__m64 _mm_srli_si64(__m64, int); [宏]</code>
_mm_srli_si128 ↗	SSE2	intrin.h	<code>__m128i _mm_srli_si128(__m128i, int);</code>
_mm_srlv_epi32 ↗	AVX2	immintrin.h	<code>__m128i _mm_srlv_epi32(__m128i, __m128i);</code>
_mm_srlv_epi64 ↗	AVX2	immintrin.h	<code>__m128i _mm_srlv_epi64(__m128i, __m128i);</code>
_mm_store_pd ↗	SSE2	intrin.h	<code>void _mm_store_pd(double*, __m128d);</code>
_mm_store_ps ↗	SSE	intrin.h	<code>void _mm_store_ps(float*, __m128);</code>
_mm_store_ps1 ↗	SSE	intrin.h	<code>void _mm_store_ps1(float*, __m128);</code>
_mm_store_sd ↗	SSE2	intrin.h	<code>void _mm_store_sd(double*, __m128d);</code>
_mm_store_si128 ↗	SSE2	intrin.h	<code>void _mm_store_si128(__m128i*, __m128i);</code>
_mm_store_ss ↗	SSE	intrin.h	<code>void _mm_store_ss(float*, __m128);</code>
_mm_store1_pd ↗	SSE2	intrin.h	<code>void _mm_store1_pd(double*, __m128d);</code>
_mm_storeh_pd ↗	SSE2	intrin.h	<code>void _mm_storeh_pd(double*, __m128d);</code>
_mm_storeh_pi ↗	SSE	intrin.h	<code>void _mm_storeh_pi(__m64*, __m128);</code>
_mm_storel_epi64 ↗	SSE2	intrin.h	<code>void _mm_storel_epi64(__m128i*, __m128i);</code>
_mm_storel_pd ↗	SSE2	intrin.h	<code>void _mm_storel_pd(double*, __m128d);</code>
_mm_storel_pi ↗	SSE	intrin.h	<code>void _mm_storel_pi(__m64*, __m128);</code>
_mm_storer_pd ↗	SSE2	intrin.h	<code>void _mm_storer_pd(double*, __m128d);</code>
_mm_storer_ps ↗	SSE	intrin.h	<code>void _mm_storer_ps(float*, __m128);</code>
_mm_storeu_pd ↗	SSE2	intrin.h	<code>void _mm_storeu_pd(double*, __m128d);</code>
_mm_storeu_ps ↗	SSE	intrin.h	<code>void _mm_storeu_ps(float*, __m128);</code>
_mm_storeu_si128 ↗	SSE2	intrin.h	<code>void _mm_storeu_si128(__m128i*, __m128i);</code>
_mm_stream_load_si128 ↗	SSE41	intrin.h	<code>__m128i _mm_stream_load_si128(__m128i*);</code>

内部函数名称	技术	标头	函数原型
_mm_stream_pd	SSE2	intrin.h	void _mm_stream_pd(double*, __m128d);
_mm_stream_pi	SSE	intrin.h	void _mm_stream_pi(__m64*, __m64);
_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	mmmintrin.h	__m64 _mm_sub_pi8(__m64, __m64); [宏]
_mm_sub_pi16	MMX	mmmintrin.h	__m64 _mm_sub_pi16(__m64, __m64); [宏]
_mm_sub_pi32	MMX	mmmintrin.h	__m64 _mm_sub_pi32(__m64, __m64); [宏]
_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);
_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	mmmintrin.h	__m64 _mm_subs_pi8(__m64, __m64); [宏]
_mm_subs_pi16	MMX	mmmintrin.h	__m64 _mm_subs_pi16(__m64, __m64); [宏]
_mm_subs_pu8	MMX	mmmintrin.h	__m64 _mm_subs_pu8(__m64, __m64); [宏]
_mm_subs_pu16	MMX	mmmintrin.h	__m64 _mm_subs_pu16(__m64, __m64); [宏]
_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);

内部函数名称	技术	标头	函数原型
_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);
_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_pi8 ↗	MMX	mmmintrin.h	_m64 _mm_unpackhi_pi8 (_m64, _m64); [宏]
_mm_unpackhi_pi16 ↗	MMX	mmmintrin.h	_m64 _mm_unpackhi_pi16 (_m64, _m64); [宏]
_mm_unpackhi_pi32 ↗	MMX	mmmintrin.h	_m64 _mm_unpackhi_pi32 (_m64, _m64); [宏]
_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	mmmintrin.h	_m64 _mm_unpacklo_pi8 (_m64, _m64); [宏]
_mm_unpacklo_pi16 ↗	MMX	mmmintrin.h	_m64 _mm_unpacklo_pi16 (_m64, _m64); [宏]
_mm_unpacklo_pi32 ↗	MMX	mmmintrin.h	_m64 _mm_unpacklo_pi32 (_m64, _m64); [宏]
_mm_unpacklo_ps ↗	SSE	intrin.h	_m128 _mm_unpacklo_ps(_m128, _m128);

内部函数名称	技术	标头	函数原型
<a href="#">_mm_xor_pd</a>	SSE2	intrin.h	<code>__m128d _mm_xor_pd(__m128d, __m128d);</code>
<a href="#">_mm_xor_ps</a>	SSE	intrin.h	<code>__m128 _mm_xor_ps(__m128, __m128);</code>
<a href="#">_mm_xor_si64</a>	MMX	mmmintrin.h	<code>__m64 _mm_xor_si64(__m64, __m64); [宏]</code>
<a href="#">_mm_xor_si128</a>	SSE2	intrin.h	<code>__m128i _mm_xor_si128(__m128i, __m128i);</code>
<a href="#">_mm256_abs_epi16</a>	AVX2	immintrin.h	<code>__m256i _mm256_abs_epi16(__m256i);</code>
<a href="#">_mm256_abs_epi32</a>	AVX2	immintrin.h	<code>__m256i _mm256_abs_epi32(__m256i);</code>
<a href="#">_mm256_abs_epi8</a>	AVX2	immintrin.h	<code>__m256i _mm256_abs_epi8(__m256i);</code>
<a href="#">_mm256_add_epi16</a>	AVX2	immintrin.h	<code>__m256i _mm256_add_epi16(__m256i, __m256i);</code>
<a href="#">_mm256_add_epi32</a>	AVX2	immintrin.h	<code>__m256i _mm256_add_epi32(__m256i, __m256i);</code>
<a href="#">_mm256_add_epi64</a>	AVX2	immintrin.h	<code>__m256i _mm256_add_epi64(__m256i, __m256i);</code>
<a href="#">_mm256_add_epi8</a>	AVX2	immintrin.h	<code>__m256i _mm256_add_epi8(__m256i, __m256i);</code>
<a href="#">_mm256_add_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_add_pd(__m256d, __m256d);</code>
<a href="#">_mm256_add_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_add_ps(__m256, __m256);</code>
<a href="#">_mm256_adds_epi16</a>	AVX2	immintrin.h	<code>__m256i _mm256_adds_epi16(__m256i, __m256i);</code>
<a href="#">_mm256_adds_epi8</a>	AVX2	immintrin.h	<code>__m256i _mm256_adds_epi8(__m256i, __m256i);</code>
<a href="#">_mm256_adds_epu16</a>	AVX2	immintrin.h	<code>__m256i _mm256_adds_epu16(__m256i, __m256i);</code>
<a href="#">_mm256_adds_epu8</a>	AVX2	immintrin.h	<code>__m256i _mm256_adds_epu8(__m256i, __m256i);</code>
<a href="#">_mm256_addsub_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_addsub_pd(__m256d, __m256d);</code>
<a href="#">_mm256_addsub_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_addsub_ps(__m256, __m256);</code>
<a href="#">_mm256_alignr_epi8</a>	AVX2	immintrin.h	<code>__m256i _mm256_alignr_epi8(__m256i, __m256i, const int);</code>
<a href="#">_mm256_and_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_and_pd(__m256d, __m256d);</code>
<a href="#">_mm256_and_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_and_ps(__m256, __m256);</code>
<a href="#">_mm256_and_si256</a>	AVX2	immintrin.h	<code>__m256i _mm256_and_si256(__m256i, __m256i);</code>
<a href="#">_mm256_andnot_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_andnot_pd(__m256d, __m256d);</code>
<a href="#">_mm256_andnot_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_andnot_ps(__m256, __m256);</code>
<a href="#">_mm256_andnot_si256</a>	AVX2	immintrin.h	<code>__m256i _mm256_andnot_si256(__m256i, __m256i);</code>
<a href="#">_mm256_avg_epu16</a>	AVX2	immintrin.h	<code>__m256i _mm256_avg_epu16(__m256i, __m256i);</code>
<a href="#">_mm256_avg_epu8</a>	AVX2	immintrin.h	<code>__m256i _mm256_avg_epu8(__m256i, __m256i);</code>
<a href="#">_mm256_blend_epi16</a>	AVX2	immintrin.h	<code>__m256i _mm256_blend_epi16(__m256i, __m256i, const int);</code>
<a href="#">_mm256_blend_epi32</a>	AVX2	immintrin.h	<code>__m256i _mm256_blend_epi32(__m256i, __m256i, const int);</code>
<a href="#">_mm256_blend_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_blend_pd(__m256d, __m256d, const int);</code>

内部函数名称	技术	标头	函数原型
_mm256_blend_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_blend_ps(__m256, __m256, const int);</code>
_mm256_blendv_epi8 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_blendv_epi8(__m256i, __m256i, __m256i);</code>
_mm256_blendv_pd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_blendv_pd(__m256d, __m256d, __m256d);</code>
_mm256_blendv_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_blendv_ps(__m256, __m256, __m256);</code>
_mm256_broadcast_pd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_broadcast_pd(__m128d const *);</code>
_mm256_broadcast_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_broadcast_ps(__m128 const *);</code>
_mm256_broadcast_sd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_broadcast_sd(double const *);</code>
_mm256_broadcast_ss <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_broadcast_ss(float const *);</code>
_mm256_broadcastb_epi8 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_broadcastb_epi8 (__m128i);</code>
_mm256_broadcastd_epi32 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_broadcastd_epi32(__m128i);</code>
_mm256_broadcastq_epi64 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_broadcastq_epi64(__m128i);</code>
_mm256_broadcastsd_pd <sup>2</sup>	AVX2	immintrin.h	<code>__m256d _mm256_broadcastsd_pd(__m128d);</code>
_mm256_broadcastsi128_si256 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_broadcastsi128_si256(__m128i);</code>
_mm256_broadcastss_ps <sup>2</sup>	AVX2	immintrin.h	<code>__m256 _mm256_broadcastss_ps(__m128);</code>
_mm256_broadcastw_epi16 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_broadcastw_epi16(__m128i);</code>
_mm256_castpd_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_castpd_ps(__m256d);</code>
_mm256_castpd_si256 <sup>2</sup>	AVX	immintrin.h	<code>__m256i _mm256_castpd_si256(__m256d);</code>
_mm256_castpd128_pd256 <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_castpd128_pd256(__m128d);</code>
_mm256_castpd256_pd128 <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_castpd256_pd128(__m256d);</code>
_mm256_castps_pd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_castps_pd(__m256);</code>
_mm256_castps_si256 <sup>2</sup>	AVX	immintrin.h	<code>__m256i _mm256_castps_si256(__m256);</code>
_mm256_castps128_ps256 <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_castps128_ps256(__m128);</code>
_mm256_castps256_ps128 <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_castps256_ps128(__m256);</code>
_mm256_castsi128_si256 <sup>2</sup>	AVX	immintrin.h	<code>__m256i _mm256_castsi128_si256(__m128i);</code>
_mm256_castsi256_pd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_castsi256_pd(__m256i);</code>
_mm256_castsi256_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_castsi256_ps(__m256i);</code>
_mm256_castsi256_si128 <sup>2</sup>	AVX	immintrin.h	<code>__m256i _mm256_castsi256_si128(__m256i);</code>
_mm256_cmov_si256	XOP	ammintrin.h	<code>__m256i _mm256_cmov_si256(__m256i, __m256i, __m256i);</code>
_mm256_cmp_pd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_cmp_pd(__m256d, __m256d, const int);</code>
_mm256_cmp_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_cmp_ps(__m256, __m256, const int);</code>
_mm256_cmpeq_epi16 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cmpeq_epi16(__m256i, __m256i);</code>

内部函数名称	技术	标头	函数原型
_mm256_cmpeq_epi32 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cmpeq_epi32(__m256i, __m256i);</code>
_mm256_cmpeq_epi64 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cmpeq_epi64(__m256i, __m256i);</code>
_mm256_cmpeq_epi8 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cmpeq_epi8(__m256i, __m256i);</code>
_mm256_cmpgt_epi16 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cmpgt_epi16(__m256i, __m256i);</code>
_mm256_cmpgt_epi32 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cmpgt_epi32(__m256i, __m256i);</code>
_mm256_cmpgt_epi64 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cmpgt_epi64(__m256i, __m256i);</code>
_mm256_cmpgt_epi8 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cmpgt_epi8(__m256i, __m256i);</code>
_mm256_cvtepi16_epi32 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepi16_epi32(__m128i);</code>
_mm256_cvtepi16_epi64 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepi16_epi64(__m128i);</code>
_mm256_cvtepi32_epi64 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepi32_epi64(__m128i);</code>
_mm256_cvtepi32_pd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_cvtepi32_pd(__m128i);</code>
_mm256_cvtepi32_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_cvtepi32_ps(__m256i);</code>
_mm256_cvtepi8_epi16 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepi8_epi16(__m128i);</code>
_mm256_cvtepi8_epi32 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepi8_epi32(__m128i);</code>
_mm256_cvtepi8_epi64 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepi8_epi64(__m128i);</code>
_mm256_cvtepu16_epi32 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepu16_epi32(__m128i);</code>
_mm256_cvtepu16_epi64 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepu16_epi64(__m128i);</code>
_mm256_cvtepu32_epi64 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepu32_epi64(__m128i);</code>
_mm256_cvtepu8_epi16 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepu8_epi16(__m128i);</code>
_mm256_cvtepu8_epi32 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepu8_epi32(__m128i);</code>
_mm256_cvtepu8_epi64 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepu8_epi64(__m128i);</code>
_mm256_cvtpd_epi32 <sup>2</sup>	AVX	immintrin.h	<code>__m128i _mm256_cvtpd_epi32(__m256d);</code>
_mm256_cvtpd_ps <sup>2</sup>	AVX	immintrin.h	<code>__m128 _mm256_cvtpd_ps(__m256d);</code>
_mm256_cvtph_ps <sup>2</sup>	F16C	immintrin.h	<code>__m256 _mm256_cvtph_ps(__m128i);</code>
_mm256_cvtps_epi32 <sup>2</sup>	AVX	immintrin.h	<code>__m256i _mm256_cvtps_epi32(__m256);</code>
_mm256_cvtps_pd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_cvtps_pd(__m128);</code>
_mm256_cvtps_ph <sup>2</sup>	F16C	immintrin.h	<code>__m128i _mm256_cvtps_ph(__m256, const int);</code>
_mm256_cvtpd_epi32 <sup>2</sup>	AVX	immintrin.h	<code>__m128i _mm256_cvtpd_epi32(__m256d);</code>
_mm256_cvtps_epi32 <sup>2</sup>	AVX	immintrin.h	<code>__m256i _mm256_cvtps_epi32(__m256);</code>
_mm256_div_pd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_div_pd(__m256d, __m256d);</code>
_mm256_div_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_div_ps(__m256, __m256);</code>
_mm256_dp_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_dp_ps(__m256, __m256, const int);</code>
_mm256_extractf128_pd <sup>2</sup>	AVX	immintrin.h	<code>__m128d _mm256_extractf128_pd(__m256d, const int);</code>

内部函数名称	技术	标头	函数原型
<a href="#">_mm256_extractf128_ps</a>	AVX	immintrin.h	<code>__m128 __mm256_extractf128_ps(__m256, const int);</code>
<a href="#">_mm256_extractf128_si256</a>	AVX	immintrin.h	<code>__m128i __mm256_extractf128_si256(__m256i, const int);</code>
<a href="#">_mm256_extracti128_si256</a>	AVX2	immintrin.h	<code>__m128i __mm256_extracti128_si256(__m256i, int);</code>
<a href="#">_mm256_fmadd_pd</a>	FMA	immintrin.h	<code>__m256d __mm256_fmadd_pd (__m256d, __m256d, __m256d);</code>
<a href="#">_mm256_fmadd_ps</a>	FMA	immintrin.h	<code>__m256 __mm256_fmadd_ps (__m256, __m256, __m256);</code>
<a href="#">_mm256_fmaddsub_pd</a>	FMA	immintrin.h	<code>__m256d __mm256_fmaddsub_pd (__m256d, __m256d, __m256d);</code>
<a href="#">_mm256_fmaddsub_ps</a>	FMA	immintrin.h	<code>__m256 __mm256_fmaddsub_ps (__m256, __m256, __m256);</code>
<a href="#">_mm256_fmsub_pd</a>	FMA	immintrin.h	<code>__m256d __mm256_fmsub_pd (__m256d, __m256d, __m256d);</code>
<a href="#">_mm256_fmsub_ps</a>	FMA	immintrin.h	<code>__m256 __mm256_fmsub_ps (__m256, __m256, __m256);</code>
<a href="#">_mm256_fmsubadd_pd</a>	FMA	immintrin.h	<code>__m256d __mm256_fmsubadd_pd (__m256d, __m256d, __m256d);</code>
<a href="#">_mm256_fmsubadd_ps</a>	FMA	immintrin.h	<code>__m256 __mm256_fmsubadd_ps (__m256, __m256, __m256);</code>
<a href="#">_mm256_fnmadd_pd</a>	FMA	immintrin.h	<code>__m256d __mm256_fnmadd_pd (__m256d, __m256d, __m256d);</code>
<a href="#">_mm256_fnmadd_ps</a>	FMA	immintrin.h	<code>__m256 __mm256_fnmadd_ps (__m256, __m256, __m256);</code>
<a href="#">_mm256_fnmsub_pd</a>	FMA	immintrin.h	<code>__m256d __mm256_fnmsub_pd (__m256d, __m256d, __m256d);</code>
<a href="#">_mm256_fnmsub_ps</a>	FMA	immintrin.h	<code>__m256 __mm256_fnmsub_ps (__m256, __m256, __m256);</code>
<a href="#">_mm256_frcz_pd</a>	XOP	ammintrin.h	<code>__m256d __mm256_frcz_pd(__m256d);</code>
<a href="#">_mm256_frcz_ps</a>	XOP	ammintrin.h	<code>__m256 __mm256_frcz_ps(__m256);</code>
<a href="#">_mm256_hadd_epi16</a>	AVX2	immintrin.h	<code>__m256i __mm256_hadd_epi16(__m256i, __m256i);</code>
<a href="#">_mm256_hadd_epi32</a>	AVX2	immintrin.h	<code>__m256i __mm256_hadd_epi32(__m256i, __m256i);</code>
<a href="#">_mm256_hadd_pd</a>	AVX	immintrin.h	<code>__m256d __mm256_hadd_pd(__m256d, __m256d);</code>
<a href="#">_mm256_hadd_ps</a>	AVX	immintrin.h	<code>__m256 __mm256_hadd_ps(__m256, __m256);</code>
<a href="#">_mm256_hadds_epi16</a>	AVX2	immintrin.h	<code>__m256i __mm256_hadds_epi16(__m256i, __m256i);</code>
<a href="#">_mm256_hsub_epi16</a>	AVX2	immintrin.h	<code>__m256i __mm256_hsub_epi16(__m256i, __m256i);</code>
<a href="#">_mm256_hsub_epi32</a>	AVX2	immintrin.h	<code>__m256i __mm256_hsub_epi32(__m256i, __m256i);</code>
<a href="#">_mm256_hsub_pd</a>	AVX	immintrin.h	<code>__m256d __mm256_hsub_pd(__m256d, __m256d);</code>
<a href="#">_mm256_hsub_ps</a>	AVX	immintrin.h	<code>__m256 __mm256_hsub_ps(__m256, __m256);</code>
<a href="#">_mm256_hsubs_epi16</a>	AVX2	immintrin.h	<code>__m256i __mm256_hsubs_epi16(__m256i, __m256i);</code>

内部函数名称	技术	标头	函数原型
_mm256_i32gather_епi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_i32gather_епi32(int const *, __m256i, const int);</code>
_mm256_i32gather_епi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_i32gather_епi64(__int64 const *, __m128i, const int);</code>
_mm256_i32gather_pd ↗	AVX2	immintrin.h	<code>__m256d _mm256_i32gather_pd(double const *, __m128i, const int);</code>
_mm256_i32gather_ps ↗	AVX2	immintrin.h	<code>__m256 _mm256_i32gather_ps(float const *, __m256i, const int);</code>
_mm256_i64gather_епi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_i64gather_епi32(int const *, __m256i, const int);</code>
_mm256_i64gather_епi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_i64gather_епi64(__int64 const *, __m256i, const int);</code>
_mm256_i64gather_pd ↗	AVX2	immintrin.h	<code>__m256d _mm256_i64gather_pd(double const *, __m256i, const int);</code>
_mm256_i64gather_ps ↗	AVX2	immintrin.h	<code>__m128 _mm256_i64gather_ps(float const *, __m256i, const int);</code>
_mm256_insertf128_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_insertf128_pd(__m256d, __m128d, int);</code>
_mm256_insertf128_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_insertf128_ps(__m256, __m128, int);</code>
_mm256_insertf128_si256 ↗	AVX	immintrin.h	<code>__m256i _mm256_insertf128_si256(__m256i, __m128i, int);</code>
_mm256_inserti128_si256 ↗	AVX2	immintrin.h	<code>__m256i _mm256_inserti128_si256(__m256i, __m128i, int);</code>
_mm256_lddqu_si256 ↗	AVX	immintrin.h	<code>__m256i _mm256_lddqu_si256(__m256i *);</code>
_mm256_load_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_load_pd(double const *);</code>
_mm256_load_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_load_ps(float const *);</code>
_mm256_load_si256 ↗	AVX	immintrin.h	<code>__m256i _mm256_load_si256(__m256i *);</code>
_mm256_loadu_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_loadu_pd(double const *);</code>
_mm256_loadu_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_loadu_ps(float const *);</code>
_mm256_loadu_si256 ↗	AVX	immintrin.h	<code>__m256i _mm256_loadu_si256(__m256i *);</code>
_mm256_macc_pd	FMA4	ammintrin.h	<code>__m256d _mm_macc_pd(__m256d, __m256d, __m256d);</code>
_mm256_macc_ps	FMA4	ammintrin.h	<code>__m256 _mm_macc_ps(__m256, __m256, __m256);</code>
_mm256_madd_епi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_madd_епi16(__m256i, __m256i);</code>
_mm256_maddsub_pd	FMA4	ammintrin.h	<code>__m256d _mm_maddsub_pd(__m256d, __m256d, __m256d);</code>
_mm256_maddsub_ps	FMA4	ammintrin.h	<code>__m256 _mm_maddsub_ps(__m256, __m256, __m256);</code>
_mm256_maddubs_епi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_maddubs_епi16(__m256i, __m256i);</code>
_mm256_mask_i32gather_епi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_mask_i32gather_епi32(__m256i, int const *, __m256i, __m256i, const int);</code>

内部函数名称	技术	标头	函数原型
_mm256_mask_i32gather_епi64 ↴	AVX2	immintrin.h	<code>__m256i _mm256_mask_i32gather_епi64(__m256i, __int64 const *, __m128i, __m256i, const int);</code>
_mm256_mask_i32gather_pd ↴	AVX2	immintrin.h	<code>__m256d _mm256_mask_i32gather_pd(__m256d, double const *, __m128i, __m256d, const int);</code>
_mm256_mask_i32gather_ps ↴	AVX2	immintrin.h	<code>__m256 _mm256_mask_i32gather_ps(__m256, float const *, __m256i, __m256, const int);</code>
_mm256_mask_i64gather_епi32 ↴	AVX2	immintrin.h	<code>__m128i _mm256_mask_i64gather_епi32(__m128i, int const *, __m256i, __m128i, const int);</code>
_mm256_mask_i64gather_епi64 ↴	AVX2	immintrin.h	<code>__m256i _mm256_mask_i64gather_епi64(__m256i, __int64 const *, __m256i, __m256i, const int);</code>
_mm256_mask_i64gather_pd ↴	AVX2	immintrin.h	<code>__m256d _mm256_mask_i64gather_pd(__m256d, double const *, __m256i, __m256d, const int);</code>
_mm256_mask_i64gather_ps ↴	AVX2	immintrin.h	<code>__m128 _mm256_mask_i64gather_ps(__m128, float const *, __m256i, __m128, const int);</code>
_mm256_maskload_епi32 ↴	AVX2	immintrin.h	<code>__m256i _mm256_maskload_епi32(int const *, __m256i);</code>
_mm256_maskload_епi64 ↴	AVX2	immintrin.h	<code>__m256i _mm256_maskload_епi64(__int64 const *, __m256i);</code>
_mm256_maskload_pd ↴	AVX	immintrin.h	<code>__m256d _mm256_maskload_pd(double const *, __m256i);</code>
_mm256_maskload_ps ↴	AVX	immintrin.h	<code>__m256 _mm256_maskload_ps(float const *, __m256i);</code>
_mm256_maskstore_епi32 ↴	AVX2	immintrin.h	<code>void _mm256_maskstore_епi32(int *, __m256i, __m256i);</code>
_mm256_maskstore_епi64 ↴	AVX2	immintrin.h	<code>void _mm256_maskstore_епi64(__int64 *, __m256i, __m256i);</code>
_mm256_maskstore_pd ↴	AVX	immintrin.h	<code>void _mm256_maskstore_pd(double *, __m256i, __m256d);</code>
_mm256_maskstore_ps ↴	AVX	immintrin.h	<code>void _mm256_maskstore_ps(float *, __m256i, __m256);</code>
_mm256_max_епi16 ↴	AVX2	immintrin.h	<code>__m256i _mm256_max_епi16(__m256i, __m256i);</code>
_mm256_max_епi32 ↴	AVX2	immintrin.h	<code>__m256i _mm256_max_епi32(__m256i, __m256i);</code>
_mm256_max_епi8 ↴	AVX2	immintrin.h	<code>__m256i _mm256_max_епi8(__m256i, __m256i);</code>
_mm256_max_епu16 ↴	AVX2	immintrin.h	<code>__m256i _mm256_max_епu16(__m256i, __m256i);</code>
_mm256_max_епu32 ↴	AVX2	immintrin.h	<code>__m256i _mm256_max_епu32(__m256i, __m256i);</code>
_mm256_max_епu8 ↴	AVX2	immintrin.h	<code>__m256i _mm256_max_епu8(__m256i, __m256i);</code>
_mm256_max_pd ↴	AVX	immintrin.h	<code>__m256d _mm256_max_pd(__m256d, __m256d);</code>
_mm256_max_ps ↴	AVX	immintrin.h	<code>__m256 _mm256_max_ps(__m256, __m256);</code>
_mm256_min_епi16 ↴	AVX2	immintrin.h	<code>__m256i _mm256_min_епi16(__m256i, __m256i);</code>
_mm256_min_епi32 ↴	AVX2	immintrin.h	<code>__m256i _mm256_min_епi32(__m256i, __m256i);</code>

内部函数名称	技术	标头	函数原型
_mm256_min_epi8 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_min_epi8(__m256i, __m256i);</code>
_mm256_min_epu16 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_min_epu16(__m256i, __m256i);</code>
_mm256_min_epu32 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_min_epu32(__m256i, __m256i);</code>
_mm256_min_epu8 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_min_epu8(__m256i, __m256i);</code>
_mm256_min_pd <sup>↗</sup>	AVX	immintrin.h	<code>__m256d _mm256_min_pd(__m256d, __m256d);</code>
_mm256_min_ps <sup>↗</sup>	AVX	immintrin.h	<code>__m256 _mm256_min_ps(__m256, __m256);</code>
_mm256_movedup_pd <sup>↗</sup>	AVX	immintrin.h	<code>__m256d _mm256_movedup_pd(__m256d);</code>
_mm256_movehdup_ps <sup>↗</sup>	AVX	immintrin.h	<code>__m256 _mm256_movehdup_ps(__m256);</code>
_mm256_moveldup_ps <sup>↗</sup>	AVX	immintrin.h	<code>__m256 _mm256_moveldup_ps(__m256);</code>
_mm256_movemask_epi8 <sup>↗</sup>	AVX2	immintrin.h	<code>int _mm256_movemask_epi8(__m256i);</code>
_mm256_movemask_pd <sup>↗</sup>	AVX	immintrin.h	<code>int _mm256_movemask_pd(__m256d);</code>
_mm256_movemask_ps <sup>↗</sup>	AVX	immintrin.h	<code>int _mm256_movemask_ps(__m256);</code>
_mm256_mpsadbw_epu8 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_mpsadbw_epu8(__m256i, __m256i, const int);</code>
_mm256_msub_pd	FMA4	ammintrin.h	<code>__m256d _mm_msub_pd(__m256d, __m256d, __m256d);</code>
_mm256_msub_ps	FMA4	ammintrin.h	<code>__m256 _mm_msub_ps(__m256, __m256, __m256);</code>
_mm256_msubadd_pd	FMA4	ammintrin.h	<code>__m256d _mm_msubadd_pd(__m256d, __m256d, __m256d);</code>
_mm256_msubadd_ps	FMA4	ammintrin.h	<code>__m256 _mm_msubadd_ps(__m256, __m256, __m256);</code>
_mm256_mul_epi32 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_mul_epi32(__m256i, __m256i);</code>
_mm256_mul_epu32 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_mul_epu32(__m256i, __m256i);</code>
_mm256_mul_pd <sup>↗</sup>	AVX	immintrin.h	<code>__m256d _mm256_mul_pd(__m256d, __m256d);</code>
_mm256_mul_ps <sup>↗</sup>	AVX	immintrin.h	<code>__m256 _mm256_mul_ps(__m256, __m256);</code>
_mm256_mulhi_epi16 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_mulhi_epi16(__m256i, __m256i);</code>
_mm256_mulhi_epu16 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_mulhi_epu16(__m256i, __m256i);</code>
_mm256_mulhrs_epi16 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_mulhrs_epi16(__m256i, __m256i);</code>
_mm256_mullo_epi16 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_mullo_epi16(__m256i, __m256i);</code>
_mm256_mullo_epi32 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_mullo_epi32(__m256i, __m256i);</code>
_mm256_nmacc_pd	FMA4	ammintrin.h	<code>__m256d _mm_nmacc_pd(__m256d, __m256d, __m256d);</code>
_mm256_nmacc_ps	FMA4	ammintrin.h	<code>__m256 _mm_nmacc_ps(__m256, __m256, __m256);</code>
_mm256_nmsub_pd	FMA4	ammintrin.h	<code>__m256d _mm_nmsub_pd(__m256d, __m256d, __m256d);</code>
_mm256_nmsub_ps	FMA4	ammintrin.h	<code>__m256 _mm_nmsub_ps(__m256, __m256, __m256);</code>
_mm256_or_pd <sup>↗</sup>	AVX	immintrin.h	<code>__m256d _mm256_or_pd(__m256d, __m256d);</code>
_mm256_or_ps <sup>↗</sup>	AVX	immintrin.h	<code>__m256 _mm256_or_ps(__m256, __m256);</code>

内部函数名称	技术	标头	函数原型
_mm256_or_si256 ↗	AVX2	immintrin.h	<code>__m256i _mm256_or_si256(__m256i, __m256i);</code>
_mm256_packs_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_packs_epi16(__m256i, __m256i);</code>
_mm256_packs_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_packs_epi32(__m256i, __m256i);</code>
_mm256_packus_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_packus_epi16(__m256i, __m256i);</code>
_mm256_packus_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_packus_epi32(__m256i, __m256i);</code>
_mm256_permute_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_permute_pd(__m256d, int);</code>
_mm256_permute_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_permute_ps(__m256, int);</code>
_mm256_permute2_pd	XOP	ammintrin.h	<code>__m256d _mm256_permute2_pd(__m256d, __m256d, __m256i, int);</code>
_mm256_permute2_ps	XOP	ammintrin.h	<code>__m256 _mm256_permute2_ps(__m256, __m256, __m256i, int);</code>
_mm256_permute2f128_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_permute2f128_pd(__m256d, __m256d, int);</code>
_mm256_permute2f128_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_permute2f128_ps(__m256, __m256, int);</code>
_mm256_permute2f128_si256 ↗	AVX	immintrin.h	<code>__m256i _mm256_permute2f128_si256(__m256i, __m256i, int);</code>
_mm256_permute2x128_si256 ↗	AVX2	immintrin.h	<code>__m256i _mm256_permute2x128_si256(__m256i, __m256i, const int);</code>
_mm256_permute4x64_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_permute4x64_epi64 (__m256i, const int);</code>
_mm256_permute4x64_pd ↗	AVX2	immintrin.h	<code>__m256d _mm256_permute4x64_pd(__m256d, const int);</code>
_mm256_permutevar_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_permutevar_pd(__m256d, __m256i);</code>
_mm256_permutevar_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_permutevar_ps(__m256, __m256i);</code>
_mm256_permutevar8x32_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_permutevar8x32_epi32(__m256i, __m256i);</code>
_mm256_permutevar8x32_ps ↗	AVX2	immintrin.h	<code>__m256 _mm256_permutevar8x32_ps (__m256, __m256i);</code>
_mm256_rcp_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_rcp_ps(__m256);</code>
_mm256_round_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_round_pd(__m256d, int);</code>
_mm256_round_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_round_ps(__m256, int);</code>
_mm256_rsqrt_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_rsqrt_ps(__m256);</code>
_mm256_sad_epu8 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sad_epu8(__m256i, __m256i);</code>
_mm256_set_epi16 ↗	AVX	immintrin.h	<code>(__m256i _mm256_set_epi16(short, short, short));</code>
_mm256_set_epi32 ↗	AVX	immintrin.h	<code>__m256i _mm256_set_epi32(int, int, int, int, int, int, int, int);</code>



内部函数名称	技术	标头	函数原型
_mm256_sign_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sign_epi16(__m256i, __m256i);</code>
_mm256_sign_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sign_epi32(__m256i, __m256i);</code>
_mm256_sign_epi8 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sign_epi8(__m256i, __m256i);</code>
_mm256_sll_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sll_epi16(__m256i, __m128i);</code>
_mm256_sll_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sll_epi32(__m256i, __m128i);</code>
_mm256_sll_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sll_epi64(__m256i, __m128i);</code>
_mm256_slli_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_slli_epi16(__m256i, int);</code>
_mm256_slli_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_slli_epi32(__m256i, int);</code>
_mm256_slli_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_slli_epi64(__m256i, int);</code>
_mm256_slli_si256 ↗	AVX2	immintrin.h	<code>__m256i _mm256_slli_si256(__m256i, int);</code>
_mm256_sllv_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sllv_epi32(__m256i, __m256i);</code>
_mm256_sllv_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sllv_epi64(__m256i, __m256i);</code>
_mm256_sqrt_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_sqrt_pd(__m256d);</code>
_mm256_sqrt_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_sqrt_ps(__m256);</code>
_mm256_sra_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sra_epi16(__m256i, __m128i);</code>
_mm256_sra_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sra_epi32(__m256i, __m128i);</code>
_mm256_srai_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srai_epi16(__m256i, int);</code>
_mm256_srai_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srai_epi32(__m256i, int);</code>
_mm256_srav_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srav_epi32(__m256i, __m256i);</code>
_mm256_srl_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srl_epi16(__m256i, __m128i);</code>
_mm256_srl_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srl_epi32(__m256i, __m128i);</code>
_mm256_srl_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srl_epi64(__m256i, __m128i);</code>
_mm256_srl_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srl_epi16(__m256i, int);</code>
_mm256_srl_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srl_epi32(__m256i, int);</code>
_mm256_srl_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srl_epi64(__m256i, int);</code>
_mm256_srl_si256 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srl_si256(__m256i, int);</code>
_mm256_srlv_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srlv_epi32(__m256i, __m256i);</code>
_mm256_srlv_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srlv_epi64(__m256i, __m256i);</code>
_mm256_store_pd ↗	AVX	immintrin.h	<code>void _mm256_store_pd(double *, __m256d);</code>
_mm256_store_ps ↗	AVX	immintrin.h	<code>void _mm256_store_ps(float *, __m256);</code>
_mm256_store_si256 ↗	AVX	immintrin.h	<code>void _mm256_store_si256(__m256i *, __m256i);</code>
_mm256_storeu_pd ↗	AVX	immintrin.h	<code>void _mm256_storeu_pd(double *, __m256d);</code>
_mm256_storeu_ps ↗	AVX	immintrin.h	<code>void _mm256_storeu_ps(float *, __m256);</code>

内部函数名称	技术	标头	函数原型
_mm256_storeu_si256 <sup>↗</sup>	AVX	immintrin.h	void _mm256_storeu_si256(__m256i *, __m256i);
_mm256_stream_load_si256 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_stream_load_si256(__m256i const *);
_mm256_stream_pd <sup>↗</sup>	AVX	immintrin.h	void _mm256_stream_pd(double *, __m256d);
_mm256_stream_ps <sup>↗</sup>	AVX	immintrin.h	void _mm256_stream_ps(float *, __m256);
_mm256_stream_si256 <sup>↗</sup>	AVX	immintrin.h	void _mm256_stream_si256(__m256i *, __m256i);
_mm256_sub_epi16 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_sub_epi16(__m256i, __m256i);
_mm256_sub_epi32 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_sub_epi32(__m256i, __m256i);
_mm256_sub_epi64 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_sub_epi64(__m256i, __m256i);
_mm256_sub_epi8 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_sub_epi8(__m256i, __m256i);
_mm256_sub_pd <sup>↗</sup>	AVX	immintrin.h	__m256d _mm256_sub_pd(__m256d, __m256d);
_mm256_sub_ps <sup>↗</sup>	AVX	immintrin.h	__m256 _mm256_sub_ps(__m256, __m256);
_mm256_subs_epi16 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_subs_epi16(__m256i, __m256i);
_mm256_subs_epi8 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_subs_epi8(__m256i, __m256i);
_mm256_subs_epu16 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_subs_epu16(__m256i, __m256i);
_mm256_subs_epu8 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_subs_epu8(__m256i, __m256i);
_mm256_testc_pd <sup>↗</sup>	AVX	immintrin.h	int _mm256_testc_pd(__m256d, __m256d);
_mm256_testc_ps <sup>↗</sup>	AVX	immintrin.h	int _mm256_testc_ps(__m256, __m256);
_mm256_testc_si256 <sup>↗</sup>	AVX	immintrin.h	int _mm256_testc_si256(__m256i, __m256i);
_mm256_testnzc_pd <sup>↗</sup>	AVX	immintrin.h	int _mm256_testnzc_pd(__m256d, __m256d);
_mm256_testnzc_ps <sup>↗</sup>	AVX	immintrin.h	int _mm256_testnzc_ps(__m256, __m256);
_mm256_testnzc_si256 <sup>↗</sup>	AVX	immintrin.h	int _mm256_testnzc_si256(__m256i, __m256i);
_mm256_testz_pd <sup>↗</sup>	AVX	immintrin.h	int _mm256_testz_pd(__m256d, __m256d);
_mm256_testz_ps <sup>↗</sup>	AVX	immintrin.h	int _mm256_testz_ps(__m256, __m256);
_mm256_testz_si256 <sup>↗</sup>	AVX	immintrin.h	int _mm256_testz_si256(__m256i, __m256i);
_mm256_unpackhi_epi16 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_unpackhi_epi16(__m256i, __m256i);
_mm256_unpackhi_epi32 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_unpackhi_epi32(__m256i, __m256i);
_mm256_unpackhi_epi64 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_unpackhi_epi64(__m256i, __m256i);
_mm256_unpackhi_epi8 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_unpackhi_epi8(__m256i, __m256i);
_mm256_unpackhi_pd <sup>↗</sup>	AVX	immintrin.h	__m256d _mm256_unpackhi_pd(__m256d, __m256d);
_mm256_unpackhi_ps <sup>↗</sup>	AVX	immintrin.h	__m256 _mm256_unpackhi_ps(__m256, __m256);
_mm256_unpacklo_epi16 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_unpacklo_epi16(__m256i, __m256i);
_mm256_unpacklo_epi32 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_unpacklo_epi32(__m256i, __m256i);
_mm256_unpacklo_epi64 <sup>↗</sup>	AVX2	immintrin.h	__m256i _mm256_unpacklo_epi64(__m256i, __m256i);

内部函数名称	技术	标头	函数原型
_mm256_unpacklo_epi8 ↗	AVX2	immintrin.h	<code>__m256i _mm256_unpacklo_epi8(__m256i, __m256i);</code>
_mm256_unpacklo_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_unpacklo_pd(__m256d, __m256d);</code>
_mm256_unpacklo_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_unpacklo_ps(__m256, __m256);</code>
_mm256_xor_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_xor_pd(__m256d, __m256d);</code>
_mm256_xor_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_xor_ps(__m256, __m256);</code>
_mm256_xor_si256 ↗	AVX2	immintrin.h	<code>__m256i _mm256_xor_si256(__m256i, __m256i);</code>
_mm256_zeroall ↗	AVX	immintrin.h	<code>void _mm256_zeroall(void);</code>
_mm256_zeroupper ↗	AVX	immintrin.h	<code>void _mm256_zeroupper(void);</code>
_movsb		intrin.h	<code>void __movsb(unsigned char *, unsigned char const *, size_t);</code>
_movsd		intrin.h	<code>void __movsd(unsigned long *, unsigned long const *, size_t);</code>
_movsw		intrin.h	<code>void __movsw(unsigned short *, unsigned short const *, size_t);</code>
_mulx_u32	BMI	immintrin.h	<code>unsigned int _mulx_u32(unsigned int, unsigned int, unsigned int*);</code>
_nop		intrin.h	<code>void __nop(void);</code>
_nvreg_restore_fence		intrin.h	<code>void __nvreg_restore_fence(void);</code>
_nvreg_save_fence		intrin.h	<code>void __nvreg_save_fence(void);</code>
_outbyte		intrin.h	<code>void __outbyte(unsigned short, unsigned char);</code>
_outbytestring		intrin.h	<code>void __outbytestring(unsigned short, unsigned char *, unsigned long);</code>
_outdword		intrin.h	<code>void __outdword(unsigned short, unsigned long);</code>
_outdwordstring		intrin.h	<code>void __outdwordstring(unsigned short, unsigned long *, unsigned long);</code>
_outword		intrin.h	<code>void __outword(unsigned short, unsigned short);</code>
_outwordstring		intrin.h	<code>void __outwordstring(unsigned short, unsigned short *, unsigned long);</code>
_pdep_u32 ↗	BMI	immintrin.h	<code>unsigned int _pdep_u32(unsigned int, unsigned int);</code>
_pext_u32 ↗	BMI	immintrin.h	<code>unsigned int _pext_u32(unsigned int, unsigned int);</code>
_popcnt	POPCNT	intrin.h	<code>unsigned int __popcnt(unsigned int);</code>
_popcnt16	POPCNT	intrin.h	<code>unsigned short __popcnt16(unsigned short);</code>
_rand16_step ↗	RDRAND	immintrin.h	<code>int _rand16_step(unsigned short *);</code>
_rand32_step ↗	RDRAND	immintrin.h	<code>int _rand32_step(unsigned int *);</code>
_rdseed16_step ↗	RDSEED	immintrin.h	<code>int _rdseed16_step(unsigned short *);</code>

内部函数名称	技术	标头	函数原型
<a href="#">_rdseed32_step</a>	RDSEED	immintrin.h	int _rdseed32_step(unsigned int *);
<a href="#">_rdtsc</a>		intrin.h	unsigned __int64 __rdtsc(void);
<a href="#">_rdtscp</a>	RDTSCP	intrin.h	unsigned __int64 __rdtscp(unsigned int*);
<a href="#">_ReadBarrier</a>		intrin.h	void _ReadBarrier(void);
<a href="#">_readcr0</a>		intrin.h	unsigned long __readcr0(void);
<a href="#">_readcr2</a>		intrin.h	unsigned long __readcr2(void);
<a href="#">_readcr3</a>		intrin.h	unsigned long __readcr3(void);
<a href="#">_readcr4</a>		intrin.h	unsigned long __readcr4(void);
<a href="#">_readcr8</a>		intrin.h	unsigned long __readcr8(void);
<a href="#">_readdr</a>		intrin.h	unsigned __readdr(unsigned);
<a href="#">_readeflags</a>		intrin.h	unsigned __readeflags(void);
<a href="#">_readfsbyte</a>		intrin.h	unsigned char __readfsbyte(unsigned long);
<a href="#">_readfsdword</a>		intrin.h	unsigned long __readfsdword(unsigned long);
<a href="#">_readfsword</a>		intrin.h	unsigned short __readfsword(unsigned long);
<a href="#">_readmsr</a>		intrin.h	unsigned __int64 __readmsr(unsigned long);
<a href="#">_readpmc</a>		intrin.h	unsigned __int64 __readpmc(unsigned long);
<a href="#">_ReadWriteBarrier</a>		intrin.h	void _ReadWriteBarrier(void);
<a href="#">_ReturnAddress</a>		intrin.h	void * _ReturnAddress(void);
<a href="#">_rorx_u32</a>	BMI	immintrin.h	unsigned int _rorx_u32(unsigned int, const unsigned int);
<a href="#">_rotl16</a>		intrin.h	unsigned short _rotl16(unsigned short, unsigned char);
<a href="#">_rotl8</a>		intrin.h	unsigned char _rotl8(unsigned char, unsigned char);
<a href="#">_rotr16</a>		intrin.h	unsigned short _rotr16(unsigned short, unsigned char);
<a href="#">_rotr8</a>		intrin.h	unsigned char _rotr8(unsigned char, unsigned char);
<a href="#">_rsm</a>		intrin.h	void _rsm(void);
<a href="#">_sarx_i32</a>	BMI	immintrin.h	int _sarx_i32(int, unsigned int);
<a href="#">_segmentlimit</a>		intrin.h	unsigned long __segmentlimit(unsigned long);
<a href="#">_sgdt</a>		intrin.h	void _sgdt(void*);
<a href="#">_shlx_u32</a>	BMI	immintrin.h	unsigned int _shlx_u32(unsigned int, unsigned int);
<a href="#">_shrx_u32</a>	BMI	immintrin.h	unsigned int _shrx_u32(unsigned int, unsigned int);

内部函数名称	技术	标头	函数原型
<a href="#">_sidt</a>		intrin.h	<code>void __sidt(void*);</code>
<a href="#">_slwpcb</a>	LWP	ammintrin.h	<code>void *_slwpcb(void);</code>
<a href="#">_stac</a>	SMAP	intrin.h	<code>void __stac(void);</code>
<a href="#">_storebe_i16</a>	MOVBE	immintrin.h	<code>void __storebe_i16(void *, short); [宏]</code>
<a href="#">_storebe_i32</a>	MOVBE	immintrin.h	<code>void __storebe_i32(void *, int); [宏]</code>
<a href="#">_store_be_u16</a>	MOVBE	immintrin.h	<code>void __store_be_u16(void *, unsigned short); [宏]</code>
<a href="#">_store_be_u32</a>	MOVBE	immintrin.h	<code>void __store_be_u32(void *, unsigned int); [宏]</code>
<a href="#">_Store_HLERelease</a>	HLE	immintrin.h	<code>void __Store_HLERelease(long volatile *, long);</code>
<a href="#">_StorePointer_HLERelease</a>	HLE	immintrin.h	<code>void __StorePointer_HLERelease(void * volatile *, void *);</code>
<a href="#">_stosb</a>		intrin.h	<code>void __stosb(unsigned char *, unsigned char, size_t);</code>
<a href="#">_stosd</a>		intrin.h	<code>void __stosd(unsigned long *, unsigned long, size_t);</code>
<a href="#">_stosw</a>		intrin.h	<code>void __stosw(unsigned short *, unsigned short, size_t);</code>
<a href="#">_subborrow_u16</a>		intrin.h	<code>unsigned char __subborrow_u16(unsigned char, unsigned short, unsigned short, unsigned short *);</code>
<a href="#">_subborrow_u32</a>		intrin.h	<code>unsigned char __subborrow_u32(unsigned char, unsigned int, unsigned int, unsigned int *);</code>
<a href="#">_subborrow_u8</a>		intrin.h	<code>unsigned char __subborrow_u8(unsigned char, unsigned char, unsigned char, unsigned char *);</code>
<a href="#">_svm_clgi</a>		intrin.h	<code>void __svm_clgi(void);</code>
<a href="#">_svm_invlpga</a>		intrin.h	<code>void __svm_invlpga(void*, int);</code>
<a href="#">_svm_skinit</a>		intrin.h	<code>void __svm_skinit(int);</code>
<a href="#">_svm_stgi</a>		intrin.h	<code>void __svm_stgi(void);</code>
<a href="#">_svm_vmload</a>		intrin.h	<code>void __svm_vmload(size_t);</code>
<a href="#">_svm_vmrn</a>		intrin.h	<code>void __svm_vmrn(size_t);</code>
<a href="#">_svm_vmsave</a>		intrin.h	<code>void __svm_vmsave(size_t);</code>
<a href="#">_t1mskc_u32</a>	ABM	ammintrin.h	<code>unsigned int __t1mskc_u32(unsigned int);</code>
<a href="#">_tzcnt_u32</a>	BMI	ammintrin.h, immintrin.h	<code>unsigned int __tzcnt_u32(unsigned int);</code>
<a href="#">_tzmsk_u32</a>	ABM	ammintrin.h	<code>unsigned int __tzmsk_u32(unsigned int);</code>
<a href="#">_ud2</a>		intrin.h	<code>void __ud2(void);</code>
<a href="#">_udiv64</a>		intrin.h	<code>unsigned int __udiv64(unsigned __int64, unsigned int, unsigned int *);</code>

内部函数名称	技术	标头	函数原型
<a href="#">_ull_rshift</a>		intrin.h	<code>unsigned __int64 [pascal/cdecl] __ull_rshift(unsigned __int64, int);</code>
<a href="#">_vmx_off</a>		intrin.h	<code>void __vmx_off(void);</code>
<a href="#">_vmx_vmptrst</a>		intrin.h	<code>void __vmx_vmptrst(unsigned __int64 *);</code>
<a href="#">_wbinvd</a>		intrin.h	<code>void __wbinvd(void);</code>
<a href="#">_WriteBarrier</a>		intrin.h	<code>void _WriteBarrier(void);</code>
<a href="#">_writecr0</a>		intrin.h	<code>void __writecr0(unsigned long);</code>
<a href="#">_writecr3</a>		intrin.h	<code>void __writecr3(unsigned long);</code>
<a href="#">_writecr4</a>		intrin.h	<code>void __writecr4(unsigned long);</code>
<a href="#">_writecr8</a>		intrin.h	<code>void __writecr8(unsigned long);</code>
<a href="#">_writedr</a>		intrin.h	<code>void __writedr(unsigned, unsigned);</code>
<a href="#">_writeeflags</a>		intrin.h	<code>void __writeeflags(unsigned);</code>
<a href="#">_writefsbyte</a>		intrin.h	<code>void __writefsbyte(unsigned long, unsigned char);</code>
<a href="#">_writefsdword</a>		intrin.h	<code>void __writefsdword(unsigned long, unsigned long);</code>
<a href="#">_writefsword</a>		intrin.h	<code>void __writefsword(unsigned long, unsigned short);</code>
<a href="#">_writemsr</a>		intrin.h	<code>void __writemsr(unsigned long, unsigned __int64);</code>
<a href="#">_xabort</a> ↗	RTM	immintrin.h	<code>void _xabort(unsigned int);</code>
<a href="#">_xbegin</a> ↗	RTM	immintrin.h	<code>unsigned _xbegin(void);</code>
<a href="#">_xend</a> ↗	RTM	immintrin.h	<code>void _xend(void);</code>
<a href="#">_xgetbv</a> ↗	XSAVE	immintrin.h	<code>unsigned __int64 _xgetbv(unsigned int);</code>
<a href="#">_xrstor</a> ↗	XSAVE	immintrin.h	<code>void _xrstor(void const*, unsigned __int64);</code>
<a href="#">_xsave</a> ↗	XSAVE	immintrin.h	<code>void _xsave(void*, unsigned __int64);</code>
<a href="#">_xsaveopt</a> ↗	XSAVEOPT	immintrin.h	<code>void _xsaveopt(void*, unsigned __int64);</code>
<a href="#">_xsetbv</a> ↗	XSAVE	immintrin.h	<code>void _xsetbv(unsigned int, unsigned __int64);</code>
<a href="#">_xtest</a> ↗	XTEST	immintrin.h	<code>unsigned char _xtest(void);</code>

## 另请参阅

[编译器内部函数](#)

[ARM 内部函数](#)

[ARM64 内部函数](#)

[x64 \(amd64\) 内部函数](#)

# x64 (amd64) 内部函数列表

项目 · 2023/06/16

本文档列出了 Microsoft C++ 编译器在面向 x64 (也称为 amd64) 时支持的内部函数。

有关各个内部函数的信息，请参见适用于你面向的处理器的资源：

- 头文件。 头文件的注释中记录了很多内部函数。
- [Intel 内部函数指南](#)。 使用搜索框查找特定的内部函数。
- [Intel 64 和 IA-32 体系结构软件开发人员手册](#)
- [Intel 体系结构指令集扩展编程参考](#)
- [Intel 高级矢量扩展](#)
- [AMD 开发人员指南、手册 & ISA 文档](#)

## x64 内部函数

下表列出了 x64 处理器上可用的内部函数。“技术”列列出了所需的指令集支持。使用 `_cpuid` 内部函数来确定运行时的指令集支持。如果两个条目均在同一行中，则它们表示同一内部函数的不同入口点。[宏] 指示该原型是一个宏。“标题”列中列出了函数原型所需的标题。为简单起见，`intrin.h` 标头同时包含 `immintrin.h` 和 `ammintrin.h`。

内部函数名称	技术	标头	函数原型
<a href="#">_addcarry_u16</a>		intrin.h	<code>unsigned char _addcarry_u16(unsigned char, unsigned short, unsigned short, unsigned short *);</code>
<a href="#">_addcarry_u32</a>		intrin.h	<code>unsigned char _addcarry_u32(unsigned char, unsigned int, unsigned int, unsigned int *);</code>
<a href="#">_addcarry_u64</a>		intrin.h	<code>unsigned char _addcarry_u64(unsigned char, unsigned __int64, unsigned __int64, unsigned __int64 *);</code>
<a href="#">_addcarry_u8</a>		intrin.h	<code>unsigned char _addcarry_u8(unsigned char, unsigned char, unsigned char, unsigned char *);</code>
<a href="#">_addcarryx_u32</a>	ADX	immintrin.h	<code>unsigned char _addcarryx_u32(unsigned char, unsigned int, unsigned int, unsigned int *);</code>
<a href="#">_addcarryx_u64</a>	ADX	immintrin.h	<code>unsigned char _addcarryx_u64(unsigned char, unsigned __int64, unsigned __int64, unsigned __int64 *);</code>
<a href="#">_addgsbyte</a>		intrin.h	<code>void __addgsbyte(unsigned long, unsigned char);</code>
<a href="#">_addgsdword</a>		intrin.h	<code>void __addgsdword(unsigned long, unsigned int);</code>
<a href="#">_addgsqword</a>		intrin.h	<code>void __addgsqword(unsigned long, unsigned __int64);</code>
<a href="#">_addgsword</a>		intrin.h	<code>void __addgsword(unsigned long, unsigned short);</code>
<a href="#">_AddressOfReturnAddress</a>		intrin.h	<code>void * __AddressOfReturnAddress(void);</code>
<a href="#">_andn_u32</a>	BMI	ammintrin.h	<code>unsigned int __andn_u32(unsigned int, unsigned int);</code>

内部函数名称	技术	标头	函数原型
_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);
_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);

内部函数名称	技术	标头	函数原型
_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);
_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);
_blsmask_u32 ↗	BMI	ammintrin.h, immintrin.h	unsigned int _blsmask_u32(unsigned int);
_blsmask_u64 ↗	BMI	ammintrin.h, immintrin.h	unsigned __int64 _blsmask_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);
_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 __fastcall(unsigned int);

内部函数名称	技术	标头	函数原型
_faststorefence		intrin.h	void __faststorefence(void);
_fxrstor <sup>2</sup>	FXSR	immintrin.h	void __fxrstor(void const*);
_fxrstor64 <sup>2</sup>	FXSR	immintrin.h	void __fxrstor64(void const*);
_fxsave <sup>2</sup>	FXSR	immintrin.h	void __fxsave(void*);
_fxsave64 <sup>2</sup>	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);
_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);

内部函数名称	技术	标头	函数原型
_interlockedbittestandreset		intrin.h	unsigned char _interlockedbittestandreset(long *, long);
_interlockedbittestandreset_HLEAcquire	HLE	immintrin.h	unsigned char _interlockedbittestandreset_HLEAcquire(long *, long);
_interlockedbittestandreset_HLERelease	HLE	immintrin.h	unsigned char _interlockedbittestandreset_HLERelease(long *, long);
_interlockedbittestandreset64		intrin.h	unsigned char _interlockedbittestandreset64(__int64 *, __int64);
_interlockedbittestandreset64_HLEAcquire	HLE	immintrin.h	unsigned char _interlockedbittestandreset64_HLEAcquire(__int64 *, __int64);
_interlockedbittestandreset64_HLERelease	HLE	immintrin.h	unsigned char _interlockedbittestandreset64_HLERelease(__int64 *, __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(__int64 *, __int64);
_interlockedbittestandset64_HLERelease	HLE	immintrin.h	unsigned char _interlockedbittestandset64_HLERelease(__int64 *, __int64);
_InterlockedCompareExchange		intrin.h	long _InterlockedCompareExchange (long volatile *, long, long);
_InterlockedCompareExchange_HLEAcquire	HLE	immintrin.h	long _InterlockedCompareExchange_HLEAcquire(long volatile *, long, long);
_InterlockedCompareExchange_HLERelease	HLE	immintrin.h	long _InterlockedCompareExchange_HLERelease(long volatile *, long, long);
_InterlockedCompareExchange_np		intrin.h	long _InterlockedCompareExchange_np (long *, long, long);
_InterlockedCompareExchange128		intrin.h	unsigned char _InterlockedCompareExchange128(__int64 volatile *, __int64, __int64, __int64*);

内部函数名称	技术	标头	函数原型
_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);
_InterlockedCompareExchange64_HLEAcquire	HLE	immintrin.h	__int64 _InterlockedCompareExchange64_HLEAcquire(__int64 volatile *, __int64, __int64);
_InterlockedCompareExchange64_HLERelease	HLE	immintrin.h	__int64 _InterlockedCompareExchange64_HLERelease(__int64 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_HLEAcquire(void *volatile *, void *, void *);
_InterlockedCompareExchangePointer_HLERelease	HLE	immintrin.h	void *_InterlockedCompareExchangePointer_HLERelease(void *volatile *, void *, void *);
_InterlockedCompareExchangePointer_np		intrin.h	void *_InterlockedCompareExchangePointer_np(void **, 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);

内部函数名称	技术	标头	函数原型
_InterlockedExchange64_HLEAcquire	HLE	immintrin.h	<code>__int64 _InterlockedExchange64_HLEAcquire(__int64 volatile *, __int64);</code>
_InterlockedExchange64_HLERelease	HLE	immintrin.h	<code>__int64 _InterlockedExchange64_HLERelease(__int64 volatile *, __int64);</code>
_InterlockedExchange8		intrin.h	<code>char _InterlockedExchange8(char volatile *, char);</code>
_InterlockedExchangeAdd		intrin.h	<code>long _InterlockedExchangeAdd(long volatile *, long);</code>
_InterlockedExchangeAdd_HLEAcquire	HLE	immintrin.h	<code>long _InterlockedExchangeAdd_HLEAcquire(long volatile *, long);</code>
_InterlockedExchangeAdd_HLERelease	HLE	immintrin.h	<code>long _InterlockedExchangeAdd_HLERelease(long volatile *, long);</code>
_InterlockedExchangeAdd16		intrin.h	<code>short _InterlockedExchangeAdd16(short volatile *, short);</code>
_InterlockedExchangeAdd64		intrin.h	<code>__int64 _InterlockedExchangeAdd64(__int64 volatile *, __int64);</code>
_InterlockedExchangeAdd64_HLEAcquire	HLE	immintrin.h	<code>__int64 _InterlockedExchangeAdd64_HLEAcquire(__int64 volatile *, __int64);</code>
_InterlockedExchangeAdd64_HLERelease	HLE	immintrin.h	<code>__int64 _InterlockedExchangeAdd64_HLERelease(__int64 volatile *, __int64);</code>
_InterlockedExchangeAdd8		intrin.h	<code>char _InterlockedExchangeAdd8(char volatile *, char);</code>
_InterlockedExchangePointer		intrin.h	<code>void * _InterlockedExchangePointer(void *volatile *, void *);</code>
_InterlockedExchangePointer_HLEAcquire	HLE	immintrin.h	<code>void * _InterlockedExchangePointer_HLEAcquire(void *volatile *, void *);</code>
_InterlockedExchangePointer_HLERelease	HLE	immintrin.h	<code>void * _InterlockedExchangePointer_HLERelease(void *volatile *, void *);</code>
_InterlockedIncrement		intrin.h	<code>long _InterlockedIncrement(long volatile *);</code>
_InterlockedIncrement16		intrin.h	<code>short _InterlockedIncrement16(short volatile *);</code>
_InterlockedIncrement64		intrin.h	<code>__int64 _InterlockedIncrement64(__int64 volatile *);</code>
_InterlockedOr		intrin.h	<code>long _InterlockedOr(long volatile *, long);</code>
_InterlockedOr_HLEAcquire	HLE	immintrin.h	<code>long _InterlockedOr_HLEAcquire(long volatile *, long);</code>
_InterlockedOr_HLERelease	HLE	immintrin.h	<code>long _InterlockedOr_HLERelease(long volatile *, long);</code>
_InterlockedOr_np		intrin.h	<code>long _InterlockedOr_np(long *, long);</code>
_InterlockedOr16		intrin.h	<code>short _InterlockedOr16(short volatile *, short);</code>
_InterlockedOr16_np		intrin.h	<code>short _InterlockedOr16_np(short *, short);</code>

内部函数名称	技术	标头	函数原型
_InterlockedOr64		intrin.h	<code>__int64 _InterlockedOr64(__int64 volatile *, __int64);</code>
_InterlockedOr64_HLEAcquire	HLE	immintrin.h	<code>__int64 _InterlockedOr64_HLEAcquire(__int64 volatile *, __int64);</code>
_InterlockedOr64_HLERelease	HLE	immintrin.h	<code>__int64 _InterlockedOr64_HLERelease(__int64 volatile *, __int64);</code>
_InterlockedOr64_np		intrin.h	<code>__int64 _InterlockedOr64_np(__int64 *, __int64);</code>
_InterlockedOr8		intrin.h	<code>char _InterlockedOr8(char volatile *, char);</code>
_InterlockedOr8_np		intrin.h	<code>char _InterlockedOr8_np(char *, char);</code>
_InterlockedXor		intrin.h	<code>long _InterlockedXor(long volatile *, long);</code>
_InterlockedXor_HLEAcquire	HLE	immintrin.h	<code>long _InterlockedXor_HLEAcquire(long volatile *, long);</code>
_InterlockedXor_HLERelease	HLE	immintrin.h	<code>long _InterlockedXor_HLERelease(long volatile *, long);</code>
_InterlockedXor_np		intrin.h	<code>long _InterlockedXor_np(long *, long);</code>
_InterlockedXor16		intrin.h	<code>short _InterlockedXor16(short volatile *, short);</code>
_InterlockedXor16_np		intrin.h	<code>short _InterlockedXor16_np(short *, short);</code>
_InterlockedXor64		intrin.h	<code>__int64 _InterlockedXor64(__int64 volatile *, __int64);</code>
_InterlockedXor64_HLEAcquire	HLE	immintrin.h	<code>__int64 _InterlockedXor64_HLEAcquire(__int64 volatile *, __int64);</code>
_InterlockedXor64_HLERelease	HLE	immintrin.h	<code>__int64 _InterlockedXor64_HLERelease(__int64 volatile *, __int64);</code>
_InterlockedXor64_np		intrin.h	<code>__int64 _InterlockedXor64_np(__int64 *, __int64);</code>
_InterlockedXor8		intrin.h	<code>char _InterlockedXor8(char volatile *, char);</code>
_InterlockedXor8_np		intrin.h	<code>char _InterlockedXor8_np(char *, char);</code>
_invlpg		intrin.h	<code>void __invlpg(void*);</code>
_invpcid	INVPCID	immintrin.h	<code>void _invpcid(unsigned int, void *);</code>
_inword		intrin.h	<code>unsigned short __inword(unsigned short);</code>
_inwordstring		intrin.h	<code>void __inwordstring(unsigned short, unsigned short *, unsigned long);</code>
_lgdt		intrin.h	<code>void _lgdt(void*);</code>
_lidt		intrin.h	<code>void __lidt(void*);</code>
_ll_lshift		intrin.h	<code>unsigned __int64 [pascal/cdecl] __ll_lshift(unsigned __int64, int);</code>
_ll_rshift		intrin.h	<code>__int64 [pascal/cdecl] __ll_rshift(__int64, int);</code>
_llwpcb	LWP	ammintrin.h	<code>void __llwpcb(void *);</code>

内部函数名称	技术	标头	函数原型
_loadbe_i16 ↗	MOVBE	immintrin.h	short _loadbe_i16(void const*); [宏]
_loadbe_i32 ↗	MOVBE	immintrin.h	int _loadbe_i32(void const*); [宏]
_loadbe_i64 ↗	MOVBE	immintrin.h	_int64 _loadbe_i64(void const*); [宏]
_load_be_u16	MOVBE	immintrin.h	unsigned short _load_be_u16(void const*); [宏]
_load_be_u32	MOVBE	immintrin.h	unsigned int _load_be_u32(void const*); [宏]
_load_be_u64	MOVBE	immintrin.h	unsigned _int64 _load_be_u64(void const*); [宏]
_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);
_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);

内部函数名称	技术	标头	函数原型
_mm_adds_epu16 ↗	SSE2	intrin.h	<code>__m128i _mm_adds_epu16(__m128i, __m128i);</code>
_mm_adds_epu8 ↗	SSE2	intrin.h	<code>__m128i _mm_adds_epu8(__m128i, __m128i);</code>
_mm_addsub_pd ↗	SSE3	intrin.h	<code>__m128d _mm_addsub_pd(__m128d, __m128d);</code>
_mm_addsub_ps ↗	SSE3	intrin.h	<code>__m128 _mm_addsub_ps(__m128, __m128);</code>
_mm_aesdec_si128 ↗	AESNI	immintrin.h	<code>__m128i _mm_aesdec_si128(__m128i, __m128i);</code>
_mm_aesdeclast_si128 ↗	AESNI	immintrin.h	<code>__m128i _mm_aesdeclast_si128(__m128i, __m128i);</code>
_mm_aesenc_si128 ↗	AESNI	immintrin.h	<code>__m128i _mm_aesenc_si128(__m128i, __m128i);</code>
_mm_aesenclast_si128 ↗	AESNI	immintrin.h	<code>__m128i _mm_aesenclast_si128(__m128i, __m128i);</code>
_mm_aesimc_si128 ↗	AESNI	immintrin.h	<code>__m128i _mm_aesimc_si128 (__m128i);</code>
_mm_aeskeygenassist_si128 ↗	AESNI	immintrin.h	<code>__m128i _mm_aeskeygenassist_si128 (__m128i, const int);</code>
_mm_alignr_epi8 ↗	SSSE3	intrin.h	<code>__m128i _mm_alignr_epi8(__m128i, __m128i, int);</code>
_mm_and_pd ↗	SSE2	intrin.h	<code>__m128d _mm_and_pd(__m128d, __m128d);</code>
_mm_and_ps ↗	SSE	intrin.h	<code>__m128 _mm_and_ps(__m128, __m128);</code>
_mm_and_si128 ↗	SSE2	intrin.h	<code>__m128i _mm_and_si128(__m128i, __m128i);</code>
_mm_andnot_pd ↗	SSE2	intrin.h	<code>__m128d _mm_andnot_pd(__m128d, __m128d);</code>
_mm_andnot_ps ↗	SSE	intrin.h	<code>__m128 _mm_andnot_ps(__m128, __m128);</code>
_mm_andnot_si128 ↗	SSE2	intrin.h	<code>__m128i _mm_andnot_si128(__m128i, __m128i);</code>
_mm_avg_epu16 ↗	SSE2	intrin.h	<code>__m128i _mm_avg_epu16(__m128i, __m128i);</code>
_mm_avg_epu8 ↗	SSE2	intrin.h	<code>__m128i _mm_avg_epu8(__m128i, __m128i);</code>
_mm_blend_epi16 ↗	SSE41	intrin.h	<code>__m128i _mm_blend_epi16 (__m128i, __m128i, const int);</code>
_mm_blend_epi32 ↗	AVX2	immintrin.h	<code>__m128i _mm_blend_epi32(__m128i, __m128i, const int);</code>
_mm_blend_pd ↗	SSE41	intrin.h	<code>__m128d _mm_blend_pd (__m128d, __m128d, const int);</code>
_mm_blend_ps ↗	SSE41	intrin.h	<code>__m128 _mm_blend_ps (__m128, __m128, const int);</code>
_mm_blendv_epi8 ↗	SSE41	intrin.h	<code>__m128i _mm_blendv_epi8 (__m128i, __m128i, __m128i);</code>
_mm_blendv_pd ↗	SSE41	intrin.h	<code>__m128d _mm_blendv_pd(__m128d, __m128d, __m128d);</code>
_mm_blendv_ps ↗	SSE41	intrin.h	<code>__m128 _mm_blendv_ps(__m128, __m128, __m128);</code>
_mm_broadcast_ss ↗	AVX	immintrin.h	<code>__m128 _mm_broadcast_ss(float const *);</code>
_mm_broadcastb_epi8 ↗	AVX2	immintrin.h	<code>__m128i _mm_broadcastb_epi8(__m128i);</code>
_mm_broadcastd_epi32 ↗	AVX2	immintrin.h	<code>__m128i _mm_broadcastd_epi32(__m128i);</code>
_mm_broadcastq_epi64 ↗	AVX2	immintrin.h	<code>__m128i _mm_broadcastq_epi64(__m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_broadcastsd_pd ↴	AVX2	immintrin.h	<code>__m128d _mm_broadcastsd_pd(__m128d);</code>
_mm_broadcastss_ps ↴	AVX2	immintrin.h	<code>__m128 _mm_broadcastss_ps(__m128);</code>
_mm_broadcastw_epi16 ↴	AVX2	immintrin.h	<code>__m128i _mm_broadcastw_epi16(__m128i);</code>
_mm_castpd_ps ↴	SSSE3	intrin.h	<code>__m128 _mm_castpd_ps(__m128d);</code>
_mm_castpd_si128 ↴	SSSE3	intrin.h	<code>__m128i _mm_castpd_si128(__m128d);</code>
_mm_castps_pd ↴	SSSE3	intrin.h	<code>__m128d _mm_castps_pd(__m128);</code>
_mm_castps_si128 ↴	SSSE3	intrin.h	<code>__m128i _mm_castps_si128(__m128);</code>
_mm_castsi128_pd ↴	SSSE3	intrin.h	<code>__m128d _mm_castsi128_pd(__m128i);</code>
_mm_castsi128_ps ↴	SSSE3	intrin.h	<code>__m128 _mm_castsi128_ps(__m128i);</code>
_mm_clflush ↴	SSE2	intrin.h	<code>void _mm_clflush(void const *);</code>
_mm_clmulepi64_si128 ↴	PCLMULQDQ	immintrin.h	<code>__m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int);</code>
_mm_cmov_si128	XOP	ammintrin.h	<code>__m128i _mm_cmov_si128(__m128i, __m128i, __m128i);</code>
_mm_cmp_pd ↴	AVX	immintrin.h	<code>__m128d _mm_cmp_pd(__m128d, __m128d, const int);</code>
_mm_cmp_ps ↴	AVX	immintrin.h	<code>__m128 _mm_cmp_ps(__m128, __m128, const int);</code>
_mm_cmp_sd ↴	AVX	immintrin.h	<code>__m128d _mm_cmp_sd(__m128d, __m128d, const int);</code>
_mm_cmp_ss ↴	AVX	immintrin.h	<code>__m128 _mm_cmp_ss(__m128, __m128, const int);</code>
_mm_cmpeq_epi16 ↴	SSE2	intrin.h	<code>__m128i _mm_cmpeq_epi16(__m128i, __m128i);</code>
_mm_cmpeq_epi32 ↴	SSE2	intrin.h	<code>__m128i _mm_cmpeq_epi32(__m128i, __m128i);</code>
_mm_cmpeq_epi64 ↴	SSE41	intrin.h	<code>__m128i _mm_cmpeq_epi64(__m128i, __m128i);</code>
_mm_cmpeq_epi8 ↴	SSE2	intrin.h	<code>__m128i _mm_cmpeq_epi8(__m128i, __m128i);</code>
_mm_cmpeq_pd ↴	SSE2	intrin.h	<code>__m128d _mm_cmpeq_pd(__m128d, __m128d);</code>
_mm_cmpeq_ps ↴	SSE	intrin.h	<code>__m128 _mm_cmpeq_ps(__m128, __m128);</code>
_mm_cmpeq_sd ↴	SSE2	intrin.h	<code>__m128d _mm_cmpeq_sd(__m128d, __m128d);</code>
_mm_cmpeq_ss ↴	SSE	intrin.h	<code>__m128 _mm_cmpeq_ss(__m128, __m128);</code>
_mm_cmpestra ↴	SSE42	intrin.h	<code>int _mm_cmpestra(__m128i, int, __m128i, int, const int);</code>
_mm_cmpestrc ↴	SSE42	intrin.h	<code>int _mm_cmpestrc(__m128i, int, __m128i, int, const int);</code>
_mm_cmpestri ↴	SSE42	intrin.h	<code>int _mm_cmpestri(__m128i, int, __m128i, int, const int);</code>
_mm_cmpestrm ↴	SSE42	intrin.h	<code>__m128i _mm_cmpestrm(__m128i, int, __m128i, int, const int);</code>
_mm_cmpestro ↴	SSE42	intrin.h	<code>int _mm_cmpestro(__m128i, int, __m128i, int, const int);</code>

内部函数名称	技术	标头	函数原型
_mm_cmpestrs <sup>↗</sup>	SSE42	intrin.h	int _mm_cmpestrs(_m128i, int, _m128i, int, const int);
_mm_cmpestrz <sup>↗</sup>	SSE42	intrin.h	int _mm_cmpestrz(_m128i, int, _m128i, int, const int);
_mm_cmpge_pd <sup>↗</sup>	SSE2	intrin.h	_m128d _mm_cmpge_pd(_m128d, _m128d);
_mm_cmpge_ps <sup>↗</sup>	SSE	intrin.h	_m128 _mm_cmpge_ps(_m128, _m128);
_mm_cmpge_sd <sup>↗</sup>	SSE2	intrin.h	_m128d _mm_cmpge_sd(_m128d, _m128d);
_mm_cmpge_ss <sup>↗</sup>	SSE	intrin.h	_m128 _mm_cmpge_ss(_m128, _m128);
_mm_cmppgt_epi16 <sup>↗</sup>	SSE2	intrin.h	_m128i _mm_cmppgt_epi16(_m128i, _m128i);
_mm_cmppgt_epi32 <sup>↗</sup>	SSE2	intrin.h	_m128i _mm_cmppgt_epi32(_m128i, _m128i);
_mm_cmppgt_epi64 <sup>↗</sup>	SSE42	intrin.h	_m128i _mm_cmppgt_epi64(_m128i, _m128i);
_mm_cmppgt_epi8 <sup>↗</sup>	SSE2	intrin.h	_m128i _mm_cmppgt_epi8(_m128i, _m128i);
_mm_cmppgt_pd <sup>↗</sup>	SSE2	intrin.h	_m128d _mm_cmppgt_pd(_m128d, _m128d);
_mm_cmppgt_ps <sup>↗</sup>	SSE	intrin.h	_m128 _mm_cmppgt_ps(_m128, _m128);
_mm_cmppgt_sd <sup>↗</sup>	SSE2	intrin.h	_m128d _mm_cmppgt_sd(_m128d, _m128d);
_mm_cmppgt_ss <sup>↗</sup>	SSE	intrin.h	_m128 _mm_cmppgt_ss(_m128, _m128);
_mm_cmpristra <sup>↗</sup>	SSE42	intrin.h	int _mm_cmpristra(_m128i, _m128i, const int);
_mm_cmpristrc <sup>↗</sup>	SSE42	intrin.h	int _mm_cmpristrc(_m128i, _m128i, const int);
_mm_cmpristri <sup>↗</sup>	SSE42	intrin.h	int _mm_cmpristri(_m128i, _m128i, const int);
_mm_cmpristrm <sup>↗</sup>	SSE42	intrin.h	_m128i _mm_cmpristrm(_m128i, _m128i, const int);
_mm_cmpristro <sup>↗</sup>	SSE42	intrin.h	int _mm_cmpristro(_m128i, _m128i, const int);
_mm_cmpristrs <sup>↗</sup>	SSE42	intrin.h	int _mm_cmpristrs(_m128i, _m128i, const int);
_mm_cmpristrz <sup>↗</sup>	SSE42	intrin.h	int _mm_cmpristrz(_m128i, _m128i, const int);
_mm_cmple_pd <sup>↗</sup>	SSE2	intrin.h	_m128d _mm_cmple_pd(_m128d, _m128d);
_mm_cmple_ps <sup>↗</sup>	SSE	intrin.h	_m128 _mm_cmple_ps(_m128, _m128);
_mm_cmple_sd <sup>↗</sup>	SSE2	intrin.h	_m128d _mm_cmple_sd(_m128d, _m128d);
_mm_cmple_ss <sup>↗</sup>	SSE	intrin.h	_m128 _mm_cmple_ss(_m128, _m128);
_mm_cmplt_epi16 <sup>↗</sup>	SSE2	intrin.h	_m128i _mm_cmplt_epi16(_m128i, _m128i);
_mm_cmplt_epi32 <sup>↗</sup>	SSE2	intrin.h	_m128i _mm_cmplt_epi32(_m128i, _m128i);
_mm_cmplt_epi8 <sup>↗</sup>	SSE2	intrin.h	_m128i _mm_cmplt_epi8(_m128i, _m128i);
_mm_cmplt_pd <sup>↗</sup>	SSE2	intrin.h	_m128d _mm_cmplt_pd(_m128d, _m128d);
_mm_cmplt_ps <sup>↗</sup>	SSE	intrin.h	_m128 _mm_cmplt_ps(_m128, _m128);
_mm_cmplt_sd <sup>↗</sup>	SSE2	intrin.h	_m128d _mm_cmplt_sd(_m128d, _m128d);
_mm_cmplt_ss <sup>↗</sup>	SSE	intrin.h	_m128 _mm_cmplt_ss(_m128, _m128);

内部函数名称	技术	标头	函数原型
<a href="#">_mm_cmpeq_pd</a>	SSE2	intrin.h	<code>_m128d _mm_cmpeq_pd(_m128d, _m128d);</code>
<a href="#">_mm_cmpeq_ps</a>	SSE	intrin.h	<code>_m128 _mm_cmpeq_ps(_m128, _m128);</code>
<a href="#">_mm_cmpeq_sd</a>	SSE2	intrin.h	<code>_m128d _mm_cmpeq_sd(_m128d, _m128d);</code>
<a href="#">_mm_cmpeq_ss</a>	SSE	intrin.h	<code>_m128 _mm_cmpeq_ss(_m128, _m128);</code>
<a href="#">_mm_cmpte_pd</a>	SSE2	intrin.h	<code>_m128d _mm_cmpte_pd(_m128d, _m128d);</code>
<a href="#">_mm_cmpte_ps</a>	SSE	intrin.h	<code>_m128 _mm_cmpte_ps(_m128, _m128);</code>
<a href="#">_mm_cmpte_sd</a>	SSE2	intrin.h	<code>_m128d _mm_cmpte_sd(_m128d, _m128d);</code>
<a href="#">_mm_cmpte_ss</a>	SSE	intrin.h	<code>_m128 _mm_cmpte_ss(_m128, _m128);</code>
<a href="#">_mm_cmptgt_pd</a>	SSE2	intrin.h	<code>_m128d _mm_cmptgt_pd(_m128d, _m128d);</code>
<a href="#">_mm_cmptgt_ps</a>	SSE	intrin.h	<code>_m128 _mm_cmptgt_ps(_m128, _m128);</code>
<a href="#">_mm_cmptgt_sd</a>	SSE2	intrin.h	<code>_m128d _mm_cmptgt_sd(_m128d, _m128d);</code>
<a href="#">_mm_cmptgt_ss</a>	SSE	intrin.h	<code>_m128 _mm_cmptgt_ss(_m128, _m128);</code>
<a href="#">_mm_cmple_pd</a>	SSE2	intrin.h	<code>_m128d _mm_cmple_pd(_m128d, _m128d);</code>
<a href="#">_mm_cmple_ps</a>	SSE	intrin.h	<code>_m128 _mm_cmple_ps(_m128, _m128);</code>
<a href="#">_mm_cmple_sd</a>	SSE2	intrin.h	<code>_m128d _mm_cmple_sd(_m128d, _m128d);</code>
<a href="#">_mm_cmple_ss</a>	SSE	intrin.h	<code>_m128 _mm_cmple_ss(_m128, _m128);</code>
<a href="#">_mm_cmplt_pd</a>	SSE2	intrin.h	<code>_m128d _mm_cmplt_pd(_m128d, _m128d);</code>
<a href="#">_mm_cmplt_ps</a>	SSE	intrin.h	<code>_m128 _mm_cmplt_ps(_m128, _m128);</code>
<a href="#">_mm_cmplt_sd</a>	SSE2	intrin.h	<code>_m128d _mm_cmplt_sd(_m128d, _m128d);</code>
<a href="#">_mm_cmplt_ss</a>	SSE	intrin.h	<code>_m128 _mm_cmplt_ss(_m128, _m128);</code>
<a href="#">_mm_cmponlt_pd</a>	SSE2	intrin.h	<code>_m128d _mm_cmponlt_pd(_m128d, _m128d);</code>
<a href="#">_mm_cmponlt_ps</a>	SSE	intrin.h	<code>_m128 _mm_cmponlt_ps(_m128, _m128);</code>
<a href="#">_mm_cmponlt_sd</a>	SSE2	intrin.h	<code>_m128d _mm_cmponlt_sd(_m128d, _m128d);</code>
<a href="#">_mm_cmponlt_ss</a>	SSE	intrin.h	<code>_m128 _mm_cmponlt_ss(_m128, _m128);</code>
<a href="#">_mm_cmport_pd</a>	SSE2	intrin.h	<code>_m128d _mm_cmport_pd(_m128d, _m128d);</code>
<a href="#">_mm_cmport_ps</a>	SSE	intrin.h	<code>_m128 _mm_cmport_ps(_m128, _m128);</code>
<a href="#">_mm_cmport_sd</a>	SSE2	intrin.h	<code>_m128d _mm_cmport_sd(_m128d, _m128d);</code>
<a href="#">_mm_cmport_ss</a>	SSE	intrin.h	<code>_m128 _mm_cmport_ss(_m128, _m128);</code>
<a href="#">_mm_cmponord_pd</a>	SSE2	intrin.h	<code>_m128d _mm_cmponord_pd(_m128d, _m128d);</code>
<a href="#">_mm_cmponord_ps</a>	SSE	intrin.h	<code>_m128 _mm_cmponord_ps(_m128, _m128);</code>
<a href="#">_mm_cmponord_sd</a>	SSE2	intrin.h	<code>_m128d _mm_cmponord_sd(_m128d, _m128d);</code>
<a href="#">_mm_cmponord_ss</a>	SSE	intrin.h	<code>_m128 _mm_cmponord_ss(_m128, _m128);</code>
<a href="#">_mm_com_epi16</a>	XOP	ammintrin.h	<code>_m128i _mm_com_epi16(_m128i, _m128i, int);</code>
<a href="#">_mm_com_epi32</a>	XOP	ammintrin.h	<code>_m128i _mm_com_epi32(_m128i, _m128i, int);</code>
<a href="#">_mm_com_epi64</a>	XOP	ammintrin.h	<code>_m128i _mm_com_epi64(_m128i, _m128i, int);</code>
<a href="#">_mm_com_epi8</a>	XOP	ammintrin.h	<code>_m128i _mm_com_epi8(_m128i, _m128i, int);</code>
<a href="#">_mm_com_epu16</a>	XOP	ammintrin.h	<code>_m128i _mm_com_epu16(_m128i, _m128i, int);</code>

内部函数名称	技术	标头	函数原型
_mm_com_epu32	XOP	ammintrin.h	<code>__m128i _mm_com_epu32(__m128i, __m128i, int);</code>
_mm_com_epu64	XOP	ammintrin.h	<code>__m128i _mm_com_epu32(__m128i, __m128i, int);</code>
_mm_com_epu8	XOP	ammintrin.h	<code>__m128i _mm_com_epu8(__m128i, __m128i, int);</code>
_mm_comieq_sd ↗	SSE2	intrin.h	<code>int _mm_comieq_sd(__m128d, __m128d);</code>
_mm_comieq_ss ↗	SSE	intrin.h	<code>int _mm_comieq_ss(__m128, __m128);</code>
_mm_comige_sd ↗	SSE2	intrin.h	<code>int _mm_comige_sd(__m128d, __m128d);</code>
_mm_comige_ss ↗	SSE	intrin.h	<code>int _mm_comige_ss(__m128, __m128);</code>
_mm_comigt_sd ↗	SSE2	intrin.h	<code>int _mm_comigt_sd(__m128d, __m128d);</code>
_mm_comigt_ss ↗	SSE	intrin.h	<code>int _mm_comigt_ss(__m128, __m128);</code>
_mm_comile_sd ↗	SSE2	intrin.h	<code>int _mm_comile_sd(__m128d, __m128d);</code>
_mm_comile_ss ↗	SSE	intrin.h	<code>int _mm_comile_ss(__m128, __m128);</code>
_mm_comilt_sd ↗	SSE2	intrin.h	<code>int _mm_comilt_sd(__m128d, __m128d);</code>
_mm_comilt_ss ↗	SSE	intrin.h	<code>int _mm_comilt_ss(__m128, __m128);</code>
_mm_comineq_sd ↗	SSE2	intrin.h	<code>int _mm_comineq_sd(__m128d, __m128d);</code>
_mm_comineq_ss ↗	SSE	intrin.h	<code>int _mm_comineq_ss(__m128, __m128);</code>
_mm_crc32_u16 ↗	SSE42	intrin.h	<code>unsigned int _mm_crc32_u16(unsigned int, unsigned short);</code>
_mm_crc32_u32 ↗	SSE42	intrin.h	<code>unsigned int _mm_crc32_u32(unsigned int, unsigned int);</code>
_mm_crc32_u64 ↗	SSE42	intrin.h	<code>unsigned __int64 _mm_crc32_u64(unsigned __int64, unsigned __int64);</code>
_mm_crc32_u8 ↗	SSE42	intrin.h	<code>unsigned int _mm_crc32_u8(unsigned int, unsigned char);</code>
_mm_cvtsi2ss ↗	SSE	intrin.h	<code>__m128 _mm_cvtsi2ss(__m128, int);</code>
_mm_cvts2si ↗	SSE	intrin.h	<code>int _mm_cvts2si(__m128);</code>
_mm_cvtepi16_epi32 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepi16_epi32(__m128i);</code>
_mm_cvtepi16_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepi16_epi64(__m128i);</code>
_mm_cvtepi32_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepi32_epi64(__m128i);</code>
_mm_cvtepi32_pd ↗	SSE2	intrin.h	<code>__m128d _mm_cvtepi32_pd(__m128i);</code>
_mm_cvtepi32_ps ↗	SSE2	intrin.h	<code>__m128 _mm_cvtepi32_ps(__m128i);</code>
_mm_cvtepi8_epi16 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepi8_epi16 (__m128i);</code>
_mm_cvtepi8_epi32 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepi8_epi32 (__m128i);</code>
_mm_cvtepi8_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepi8_epi64 (__m128i);</code>
_mm_cvtepu16_epi32 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepu16_epi32(__m128i);</code>
_mm_cvtepu16_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepu16_epi64(__m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_cvtepu32_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepu32_epi64(__m128i);</code>
_mm_cvtepu8_epi16 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepu8_epi16 (__m128i);</code>
_mm_cvtepu8_epi32 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepu8_epi32 (__m128i);</code>
_mm_cvtepu8_epi64 ↗	SSE41	intrin.h	<code>__m128i _mm_cvtepu8_epi64 (__m128i);</code>
_mm_cvtpd_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_cvtpd_epi32(__m128d);</code>
_mm_cvtpd_ps ↗	SSE2	intrin.h	<code>__m128 _mm_cvtpd_ps(__m128d);</code>
_mm_cvtph_ps ↗	F16C	immintrin.h	<code>__m128 _mm_cvtph_ps(__m128i);</code>
_mm_cvtps_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_cvtps_epi32(__m128);</code>
_mm_cvtps_pd ↗	SSE2	intrin.h	<code>__m128d _mm_cvtps_pd(__m128);</code>
_mm_cvtps_ph ↗	F16C	immintrin.h	<code>__m128i _mm_cvtps_ph(__m128, const int);</code>
_mm_cvtsd_f64 ↗	SSSE3	intrin.h	<code>double _mm_cvtsd_f64(__m128d);</code>
_mm_cvtsd_si32 ↗	SSE2	intrin.h	<code>int _mm_cvtsd_si32(__m128d);</code>
_mm_cvtsd_si64 ↗	SSE2	intrin.h	<code>__int64 _mm_cvtsd_si64(__m128d);</code>
_mm_cvtsd_si64x ↗	SSE2	intrin.h	<code>__int64 _mm_cvtsd_si64x(__m128d);</code>
_mm_cvtsd_ss ↗	SSE2	intrin.h	<code>__m128 _mm_cvtsd_ss(__m128, __m128d);</code>
_mm_cvtsi128_si32 ↗	SSE2	intrin.h	<code>int _mm_cvtsi128_si32(__m128i);</code>
_mm_cvtsi128_si64 ↗	SSE2	intrin.h	<code>__int64 _mm_cvtsi128_si64(__m128i);</code>
_mm_cvtsi128_si64x ↗	SSE2	intrin.h	<code>__int64 _mm_cvtsi128_si64x(__m128i);</code>
_mm_cvtsi32_sd ↗	SSE2	intrin.h	<code>__m128d _mm_cvtsi32_sd(__m128d, int);</code>
_mm_cvtsi32_si128 ↗	SSE2	intrin.h	<code>__m128i _mm_cvtsi32_si128(int);</code>
_mm_cvtsi64_sd ↗	SSE2	intrin.h	<code>__m128d _mm_cvtsi64_sd(__m128d, __int64);</code>
_mm_cvtsi64_si128 ↗	SSE2	intrin.h	<code>__m128i _mm_cvtsi64_si128(__int64);</code>
_mm_cvtsi64_ss ↗	SSE	intrin.h	<code>__m128 _mm_cvtsi64_ss(__m128, __int64);</code>
_mm_cvtsi64x_sd ↗	SSE2	intrin.h	<code>__m128d _mm_cvtsi64x_sd(__m128d, __int64);</code>
_mm_cvtsi64x_si128 ↗	SSE2	intrin.h	<code>__m128i _mm_cvtsi64x_si128(__int64);</code>
_mm_cvtsi64x_ss	SSE2	intrin.h	<code>__m128 _mm_cvtsi64x_ss(__m128, __int64);</code>
_mm_cvtss_f32 ↗	SSSE3	intrin.h	<code>float _mm_cvtss_f32(__m128);</code>
_mm_cvtss_sd ↗	SSE2	intrin.h	<code>__m128d _mm_cvtss_sd(__m128d, __m128);</code>
_mm_cvtss_si64 ↗	SSE	intrin.h	<code>__int64 _mm_cvtss_si64(__m128);</code>
_mm_cvtss_si64x	SSE2	intrin.h	<code>__int64 _mm_cvtss_si64x(__m128);</code>
_mm_cvtt_ss2si ↗	SSE	intrin.h	<code>int _mm_cvtt_ss2si(__m128);</code>
_mm_cvtpd_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_cvtpd_epi32(__m128d);</code>
_mm_cvtps_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_cvtps_epi32(__m128);</code>

内部函数名称	技术	标头	函数原型
_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_cvtss_si64	SSE2	intrin.h	_int64 _mm_cvtss_si64(_m128);
_mm_cvtss_si64x	SSE2	intrin.h	_int64 _mm_cvtss_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);
_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);
_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);

内部函数名称	技术	标头	函数原型
<a href="#">_mm_fnmadd_sd</a>	FMA	immintrin.h	<code>_m128d _mm_fnmadd_sd (_m128d, _m128d, _m128d);</code>
<a href="#">_mm_fnmadd_ss</a>	FMA	immintrin.h	<code>_m128 _mm_fnmadd_ss (_m128, _m128, _m128);</code>
<a href="#">_mm_fnmsub_pd</a>	FMA	immintrin.h	<code>_m128d _mm_fnmsub_pd (_m128d, _m128d, _m128d);</code>
<a href="#">_mm_fnmsub_ps</a>	FMA	immintrin.h	<code>_m128 _mm_fnmsub_ps (_m128, _m128, _m128);</code>
<a href="#">_mm_fnmsub_sd</a>	FMA	immintrin.h	<code>_m128d _mm_fnmsub_sd (_m128d, _m128d, _m128d);</code>
<a href="#">_mm_fnmsub_ss</a>	FMA	immintrin.h	<code>_m128 _mm_fnmsub_ss (_m128, _m128, _m128);</code>
<a href="#">_mm_frcz_pd</a>	XOP	ammintrin.h	<code>_m128d _mm_frcz_pd(_m128d);</code>
<a href="#">_mm_frcz_ps</a>	XOP	ammintrin.h	<code>_m128 _mm_frcz_ps(_m128);</code>
<a href="#">_mm_frcz_sd</a>	XOP	ammintrin.h	<code>_m128d _mm_frcz_sd(_m128d, _m128d);</code>
<a href="#">_mm_frcz_ss</a>	XOP	ammintrin.h	<code>_m128 _mm_frcz_ss(_m128, _m128);</code>
<a href="#">_mm_getcsr</a>	SSE	intrin.h	<code>unsigned int _mm_getcsr(void);</code>
<a href="#">_mm_hadd_epi16</a>	SSSE3	intrin.h	<code>_m128i _mm_hadd_epi16(_m128i, _m128i);</code>
<a href="#">_mm_hadd_epi32</a>	SSSE3	intrin.h	<code>_m128i _mm_hadd_epi32(_m128i, _m128i);</code>
<a href="#">_mm_hadd_pd</a>	SSE3	intrin.h	<code>_m128d _mm_hadd_pd(_m128d, _m128d);</code>
<a href="#">_mm_hadd_ps</a>	SSE3	intrin.h	<code>_m128 _mm_hadd_ps(_m128, _m128);</code>
<a href="#">_mm_haddd_epi16</a>	XOP	ammintrin.h	<code>_m128i _mm_haddd_epi16(_m128i);</code>
<a href="#">_mm_haddd_epi8</a>	XOP	ammintrin.h	<code>_m128i _mm_haddd_epi8(_m128i);</code>
<a href="#">_mm_haddd_epu16</a>	XOP	ammintrin.h	<code>_m128i _mm_haddd_epu16(_m128i);</code>
<a href="#">_mm_haddd_epu8</a>	XOP	ammintrin.h	<code>_m128i _mm_haddd_epu8(_m128i);</code>
<a href="#">_mm_haddq_epi16</a>	XOP	ammintrin.h	<code>_m128i _mm_haddq_epi16(_m128i);</code>
<a href="#">_mm_haddq_epi32</a>	XOP	ammintrin.h	<code>_m128i _mm_haddq_epi32(_m128i);</code>
<a href="#">_mm_haddq_epi8</a>	XOP	ammintrin.h	<code>_m128i _mm_haddq_epi8(_m128i);</code>
<a href="#">_mm_haddq_epu16</a>	XOP	ammintrin.h	<code>_m128i _mm_haddq_epu16(_m128i);</code>
<a href="#">_mm_haddq_epu32</a>	XOP	ammintrin.h	<code>_m128i _mm_haddq_epu32(_m128i);</code>
<a href="#">_mm_haddq_epu8</a>	XOP	ammintrin.h	<code>_m128i _mm_haddq_epu8(_m128i);</code>
<a href="#">_mm_hadds_epi16</a>	SSSE3	intrin.h	<code>_m128i _mm_hadds_epi16(_m128i, _m128i);</code>
<a href="#">_mm_haddw_epi8</a>	XOP	ammintrin.h	<code>_m128i _mm_haddw_epi8(_m128i);</code>
<a href="#">_mm_haddw_epu8</a>	XOP	ammintrin.h	<code>_m128i _mm_haddw_epu8(_m128i);</code>
<a href="#">_mm_hsub_epi16</a>	SSSE3	intrin.h	<code>_m128i _mm_hsub_epi16(_m128i, _m128i);</code>
<a href="#">_mm_hsub_epi32</a>	SSSE3	intrin.h	<code>_m128i _mm_hsub_epi32(_m128i, _m128i);</code>
<a href="#">_mm_hsub_pd</a>	SSE3	intrin.h	<code>_m128d _mm_hsub_pd(_m128d, _m128d);</code>
<a href="#">_mm_hsub_ps</a>	SSE3	intrin.h	<code>_m128 _mm_hsub_ps(_m128, _m128);</code>
<a href="#">_mm_hsubd_epi16</a>	XOP	ammintrin.h	<code>_m128i _mm_hsubd_epi16(_m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_hsubq_epi32	XOP	ammintrin.h	<code>__m128i _mm_hsubq_epi32(__m128i);</code>
_mm_hsubs_epi16	SSSE3	intrin.h	<code>__m128i _mm_hsubs_epi16(__m128i, __m128i);</code>
_mm_hsubw_epi8	XOP	ammintrin.h	<code>__m128i _mm_hsubw_epi8(__m128i);</code>
_mm_i32gather_epi32	AVX2	immintrin.h	<code>__m128i _mm_i32gather_epi32(int const *, __m128i, const int);</code>
_mm_i32gather_epi64	AVX2	immintrin.h	<code>__m128i _mm_i32gather_epi64(__int64 const *, __m128i, const int);</code>
_mm_i32gather_pd	AVX2	immintrin.h	<code>__m128d _mm_i32gather_pd(double const *, __m128i, const int);</code>
_mm_i32gather_ps	AVX2	immintrin.h	<code>__m128 _mm_i32gather_ps(float const *, __m128i, const int);</code>
_mm_i64gather_epi32	AVX2	immintrin.h	<code>__m128i _mm_i64gather_epi32(int const *, __m128i, const int);</code>
_mm_i64gather_epi64	AVX2	immintrin.h	<code>__m128i _mm_i64gather_epi64(__int64 const *, __m128i, const int);</code>
_mm_i64gather_pd	AVX2	immintrin.h	<code>__m128d _mm_i64gather_pd(double const *, __m128i, const int);</code>
_mm_i64gather_ps	AVX2	immintrin.h	<code>__m128 _mm_i64gather_ps(float const *, __m128i, const int);</code>
_mm_insert_epi16	SSE2	intrin.h	<code>__m128i _mm_insert_epi16(__m128i, int, int);</code>
_mm_insert_epi32	SSE41	intrin.h	<code>__m128i _mm_insert_epi32(__m128i, int, const int);</code>
_mm_insert_epi64	SSE41	intrin.h	<code>__m128i _mm_insert_epi64(__m128i, __int64, const int);</code>
_mm_insert_epi8	SSE41	intrin.h	<code>__m128i _mm_insert_epi8 (__m128i, int, const int);</code>
_mm_insert_ps	SSE41	intrin.h	<code>__m128 _mm_insert_ps(__m128, __m128, const int);</code>
_mm_insert_si64	SSE4a	intrin.h	<code>__m128i _mm_insert_si64(__m128i, __m128i);</code>
_mm_inserti_si64	SSE4a	intrin.h	<code>__m128i _mm_inserti_si64(__m128i, __m128i, int, int);</code>
_mm_lddqu_si128	SSE3	intrin.h	<code>__m128i _mm_lddqu_si128(__m128i const*);</code>
_mm_lfence	SSE2	intrin.h	<code>void _mm_lfence(void);</code>
_mm_load_pd	SSE2	intrin.h	<code>__m128d _mm_load_pd(double*);</code>
_mm_load_ps	SSE	intrin.h	<code>__m128 _mm_load_ps(float*);</code>
_mm_load_ps1	SSE	intrin.h	<code>__m128 _mm_load_ps1(float*);</code>
_mm_load_sd	SSE2	intrin.h	<code>__m128d _mm_load_sd(double*);</code>
_mm_load_si128	SSE2	intrin.h	<code>__m128i _mm_load_si128(__m128i*);</code>
_mm_load_ss	SSE	intrin.h	<code>__m128 _mm_load_ss(float*);</code>
_mm_load1_pd	SSE2	intrin.h	<code>__m128d _mm_load1_pd(double*);</code>

内部函数名称	技术	标头	函数原型
_mm_loaddup_pd ↗	SSE3	intrin.h	<code>__m128d _mm_loaddup_pd(double const*);</code>
_mm_loadh_pd ↗	SSE2	intrin.h	<code>__m128d _mm_loadh_pd(__m128d, double*);</code>
_mm_loadh_pi ↗	SSE	intrin.h	<code>__m128 _mm_loadh_pi(__m128, __m64*);</code>
_mm_loadl_epi64 ↗	SSE2	intrin.h	<code>__m128i _mm_loadl_epi64(__m128i*);</code>
_mm_loadl_pd ↗	SSE2	intrin.h	<code>__m128d _mm_loadl_pd(__m128d, double*);</code>
_mm_loadl_pi ↗	SSE	intrin.h	<code>__m128 _mm_loadl_pi(__m128, __m64*);</code>
_mm_loadr_pd ↗	SSE2	intrin.h	<code>__m128d _mm_loadr_pd(double*);</code>
_mm_loadr_ps ↗	SSE	intrin.h	<code>__m128 _mm_loadr_ps(float*);</code>
_mm_loadu_pd ↗	SSE2	intrin.h	<code>__m128d _mm_loadu_pd(double*);</code>
_mm_loadu_ps ↗	SSE	intrin.h	<code>__m128 _mm_loadu_ps(float*);</code>
_mm_loadu_si128 ↗	SSE2	intrin.h	<code>__m128i _mm_loadu_si128(__m128i*);</code>
_mm_macc_epi16	XOP	ammintrin.h	<code>__m128i _mm_macc_epi16(__m128i, __m128i, __m128i);</code>
_mm_macc_epi32	XOP	ammintrin.h	<code>__m128i _mm_macc_epi32(__m128i, __m128i, __m128i);</code>
_mm_macc_pd	FMA4	ammintrin.h	<code>__m128d _mm_macc_pd(__m128d, __m128d, __m128d);</code>
_mm_macc_ps	FMA4	ammintrin.h	<code>__m128 _mm_macc_ps(__m128, __m128, __m128);</code>
_mm_macc_sd	FMA4	ammintrin.h	<code>__m128d _mm_macc_sd(__m128d, __m128d, __m128d);</code>
_mm_macc_ss	FMA4	ammintrin.h	<code>__m128 _mm_macc_ss(__m128, __m128, __m128);</code>
_mm_maccd_epi16	XOP	ammintrin.h	<code>__m128i _mm_maccd_epi16(__m128i, __m128i, __m128i);</code>
_mm_macchi_epi32	XOP	ammintrin.h	<code>__m128i _mm_macchi_epi32(__m128i, __m128i, __m128i);</code>
_mm_macclo_epi32	XOP	ammintrin.h	<code>__m128i _mm_macclo_epi32(__m128i, __m128i, __m128i);</code>
_mm_maccs_epi16	XOP	ammintrin.h	<code>__m128i _mm_maccs_epi16(__m128i, __m128i, __m128i);</code>
_mm_maccs_epi32	XOP	ammintrin.h	<code>__m128i _mm_maccs_epi32(__m128i, __m128i, __m128i);</code>
_mm_maccsd_epi16	XOP	ammintrin.h	<code>__m128i _mm_maccsd_epi16(__m128i, __m128i, __m128i);</code>
_mm_maccshi_epi32	XOP	ammintrin.h	<code>__m128i _mm_maccshi_epi32(__m128i, __m128i, __m128i);</code>
_mm_maccslo_epi32	XOP	ammintrin.h	<code>__m128i _mm_maccslo_epi32(__m128i, __m128i, __m128i);</code>
_mm_madd_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_madd_epi16(__m128i, __m128i);</code>
_mm_maddd_epi16	XOP	ammintrin.h	<code>__m128i _mm_maddd_epi16(__m128i, __m128i, __m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_maddsd_epi16	XOP	ammintrin.h	<code>__m128i _mm_maddsd_epi16(__m128i, __m128i);</code>
_mm_maddsub_pd	FMA4	ammintrin.h	<code>__m128d _mm_maddsub_pd(__m128d, __m128d, __m128d);</code>
_mm_maddsub_ps	FMA4	ammintrin.h	<code>__m128 _mm_maddsub_ps(__m128, __m128, __m128);</code>
_mm_maddubs_epi16	SSSE3	intrin.h	<code>__m128i _mm_maddubs_epi16(__m128i, __m128i);</code>
_mm_mask_i32gather_epi32	AVX2	immintrin.h	<code>__m128i _mm_mask_i32gather_epi32(__m128i, int const *, __m128i, __m128i, const int);</code>
_mm_mask_i32gather_epi64	AVX2	immintrin.h	<code>__m128i _mm_mask_i32gather_epi64(__m128i, __int64 const *, __m128i, __m128i, const int);</code>
_mm_mask_i32gather_pd	AVX2	immintrin.h	<code>__m128d _mm_mask_i32gather_pd(__m128d, double const *, __m128i, __m128d, const int);</code>
_mm_mask_i32gather_ps	AVX2	immintrin.h	<code>__m128 _mm_mask_i32gather_ps(__m128, float const *, __m128i, __m128, const int);</code>
_mm_mask_i64gather_epi32	AVX2	immintrin.h	<code>__m128i _mm_mask_i64gather_epi32(__m128i, int const *, __m128i, __m128i, const int);</code>
_mm_mask_i64gather_epi64	AVX2	immintrin.h	<code>__m128i _mm_mask_i64gather_epi64(__m128i, __int64 const *, __m128i, __m128i, const int);</code>
_mm_mask_i64gather_pd	AVX2	immintrin.h	<code>__m128d _mm_mask_i64gather_pd(__m128d, double const *, __m128i, __m128d, const int);</code>
_mm_mask_i64gather_ps	AVX2	immintrin.h	<code>__m128 _mm_mask_i64gather_ps(__m128, float const *, __m128i, __m128, const int);</code>
_mm_maskload_epi32	AVX2	immintrin.h	<code>__m128i _mm_maskload_epi32(int const *, __m128i);</code>
_mm_maskload_epi64	AVX2	immintrin.h	<code>__m128i _mm_maskload_epi64(__int64 const *, __m128i);</code>
_mm_maskload_pd	AVX	immintrin.h	<code>__m128d _mm_maskload_pd(double const *, __m128i);</code>
_mm_maskload_ps	AVX	immintrin.h	<code>__m128 _mm_maskload_ps(float const *, __m128i);</code>
_mm_maskmoveu_si128	SSE2	intrin.h	<code>void _mm_maskmoveu_si128(__m128i, __m128i, char*);</code>
_mm_maskstore_epi32	AVX2	immintrin.h	<code>void _mm_maskstore_epi32(int *, __m128i, __m128i);</code>
_mm_maskstore_epi64	AVX2	immintrin.h	<code>void _mm_maskstore_epi64(__int64 *, __m128i, __m128i);</code>
_mm_maskstore_pd	AVX	immintrin.h	<code>void _mm_maskstore_pd(double *, __m128i, __m128d);</code>
_mm_maskstore_ps	AVX	immintrin.h	<code>void _mm_maskstore_ps(float *, __m128i, __m128);</code>
_mm_max_epi16	SSE2	intrin.h	<code>__m128i _mm_max_epi16(__m128i, __m128i);</code>
_mm_max_epi32	SSE41	intrin.h	<code>__m128i _mm_max_epi32(__m128i, __m128i);</code>
_mm_max_epi8	SSE41	intrin.h	<code>__m128i _mm_max_epi8 (__m128i, __m128i);</code>
_mm_max_epu16	SSE41	intrin.h	<code>__m128i _mm_max_epu16(__m128i, __m128i);</code>
_mm_max_epu32	SSE41	intrin.h	<code>__m128i _mm_max_epu32(__m128i, __m128i);</code>
_mm_max_epu8	SSE2	intrin.h	<code>__m128i _mm_max_epu8(__m128i, __m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_max_pd	SSE2	intrin.h	<code>__m128d _mm_max_pd(__m128d, __m128d);</code>
_mm_max_ps	SSE	intrin.h	<code>__m128 _mm_max_ps(__m128, __m128);</code>
_mm_max_sd	SSE2	intrin.h	<code>__m128d _mm_max_sd(__m128d, __m128d);</code>
_mm_max_ss	SSE	intrin.h	<code>__m128 _mm_max_ss(__m128, __m128);</code>
_mm_mfence	SSE2	intrin.h	<code>void _mm_mfence(void);</code>
_mm_min_epi16	SSE2	intrin.h	<code>__m128i _mm_min_epi16(__m128i, __m128i);</code>
_mm_min_epi32	SSE41	intrin.h	<code>__m128i _mm_min_epi32(__m128i, __m128i);</code>
_mm_min_epi8	SSE41	intrin.h	<code>__m128i _mm_min_epi8 (__m128i, __m128i);</code>
_mm_min_epu16	SSE41	intrin.h	<code>__m128i _mm_min_epu16(__m128i, __m128i);</code>
_mm_min_epu32	SSE41	intrin.h	<code>__m128i _mm_min_epu32(__m128i, __m128i);</code>
_mm_min_epu8	SSE2	intrin.h	<code>__m128i _mm_min_epu8(__m128i, __m128i);</code>
_mm_min_pd	SSE2	intrin.h	<code>__m128d _mm_min_pd(__m128d, __m128d);</code>
_mm_min_ps	SSE	intrin.h	<code>__m128 _mm_min_ps(__m128, __m128);</code>
_mm_min_sd	SSE2	intrin.h	<code>__m128d _mm_min_sd(__m128d, __m128d);</code>
_mm_min_ss	SSE	intrin.h	<code>__m128 _mm_min_ss(__m128, __m128);</code>
_mm_minpos_epu16	SSE41	intrin.h	<code>__m128i _mm_minpos_epu16(__m128i);</code>
_mm_monitor	SSE3	intrin.h	<code>void _mm_monitor(void const*, unsigned int, unsigned int);</code>
_mm_move_epi64	SSE2	intrin.h	<code>__m128i _mm_move_epi64(__m128i);</code>
_mm_move_sd	SSE2	intrin.h	<code>__m128d _mm_move_sd(__m128d, __m128d);</code>
_mm_move_ss	SSE	intrin.h	<code>__m128 _mm_move_ss(__m128, __m128);</code>
_mm_movedup_pd	SSE3	intrin.h	<code>__m128d _mm_movedup_pd(__m128d);</code>
_mm_movehdup_ps	SSE3	intrin.h	<code>__m128 _mm_movehdup_ps(__m128);</code>
_mm_movehl_ps	SSE	intrin.h	<code>__m128 _mm_movehl_ps(__m128, __m128);</code>
_mm_moveldup_ps	SSE3	intrin.h	<code>__m128 _mm_moveldup_ps(__m128);</code>
_mm_movelh_ps	SSE	intrin.h	<code>__m128 _mm_movelh_ps(__m128, __m128);</code>
_mm_movemask_epi8	SSE2	intrin.h	<code>int _mm_movemask_epi8(__m128i);</code>
_mm_movemask_pd	SSE2	intrin.h	<code>int _mm_movemask_pd(__m128d);</code>
_mm_movemask_ps	SSE	intrin.h	<code>int _mm_movemask_ps(__m128);</code>
_mm_mpsadbw_epu8	SSE41	intrin.h	<code>__m128i _mm_mpsadbw_epu8(__m128i, __m128i, const int);</code>
_mm_msub_pd	FMA4	ammintrin.h	<code>__m128d _mm_msub_pd(__m128d, __m128d, __m128d);</code>
_mm_msub_ps	FMA4	ammintrin.h	<code>__m128 _mm_msub_ps(__m128, __m128, __m128);</code>
_mm_msub_sd	FMA4	ammintrin.h	<code>__m128d _mm_msub_sd(__m128d, __m128d, __m128d);</code>

内部函数名称	技术	标头	函数原型
<code>_mm_msub_ss</code>	FMA4	ammintrin.h	<code>__m128 _mm_msub_ss(__m128, __m128, __m128);</code>
<code>_mm_msubadd_pd</code>	FMA4	ammintrin.h	<code>__m128d _mm_msubadd_pd(__m128d, __m128d, __m128d);</code>
<code>_mm_msubadd_ps</code>	FMA4	ammintrin.h	<code>__m128 _mm_msubadd_ps(__m128, __m128, __m128);</code>
<code>_mm_mul_epi32</code>	SSE41	intrin.h	<code>__m128i _mm_mul_epi32(__m128i, __m128i);</code>
<code>_mm_mul_epu32</code>	SSE2	intrin.h	<code>__m128i _mm_mul_epu32(__m128i, __m128i);</code>
<code>_mm_mul_pd</code>	SSE2	intrin.h	<code>__m128d _mm_mul_pd(__m128d, __m128d);</code>
<code>_mm_mul_ps</code>	SSE	intrin.h	<code>__m128 _mm_mul_ps(__m128, __m128);</code>
<code>_mm_mul_sd</code>	SSE2	intrin.h	<code>__m128d _mm_mul_sd(__m128d, __m128d);</code>
<code>_mm_mul_ss</code>	SSE	intrin.h	<code>__m128 _mm_mul_ss(__m128, __m128);</code>
<code>_mm_mulhi_epi16</code>	SSE2	intrin.h	<code>__m128i _mm_mulhi_epi16(__m128i, __m128i);</code>
<code>_mm_mulhi_epu16</code>	SSE2	intrin.h	<code>__m128i _mm_mulhi_epu16(__m128i, __m128i);</code>
<code>_mm_mulhrs_epi16</code>	SSSE3	intrin.h	<code>__m128i _mm_mulhrs_epi16(__m128i, __m128i);</code>
<code>_mm_mullo_epi16</code>	SSE2	intrin.h	<code>__m128i _mm_mullo_epi16(__m128i, __m128i);</code>
<code>_mm_mullo_epi32</code>	SSE41	intrin.h	<code>__m128i _mm_mullo_epi32(__m128i, __m128i);</code>
<code>_mm_mwait</code>	SSE3	intrin.h	<code>void _mm_mwait(unsigned int, unsigned int);</code>
<code>_mm_nmacc_pd</code>	FMA4	ammintrin.h	<code>__m128d _mm_nmacc_pd(__m128d, __m128d, __m128d);</code>
<code>_mm_nmacc_ps</code>	FMA4	ammintrin.h	<code>__m128 _mm_nmacc_ps(__m128, __m128, __m128);</code>
<code>_mm_nmacc_sd</code>	FMA4	ammintrin.h	<code>__m128d _mm_nmacc_sd(__m128d, __m128d, __m128d);</code>
<code>_mm_nmacc_ss</code>	FMA4	ammintrin.h	<code>__m128 _mm_nmacc_ss(__m128, __m128, __m128);</code>
<code>_mm_nmsub_pd</code>	FMA4	ammintrin.h	<code>__m128d _mm_nmsub_pd(__m128d, __m128d, __m128d);</code>
<code>_mm_nmsub_ps</code>	FMA4	ammintrin.h	<code>__m128 _mm_nmsub_ps(__m128, __m128, __m128);</code>
<code>_mm_nmsub_sd</code>	FMA4	ammintrin.h	<code>__m128d _mm_nmsub_sd(__m128d, __m128d, __m128d);</code>
<code>_mm_nmsub_ss</code>	FMA4	ammintrin.h	<code>__m128 _mm_nmsub_ss(__m128, __m128, __m128);</code>
<code>_mm_or_pd</code>	SSE2	intrin.h	<code>__m128d _mm_or_pd(__m128d, __m128d);</code>
<code>_mm_or_ps</code>	SSE	intrin.h	<code>__m128 _mm_or_ps(__m128, __m128);</code>
<code>_mm_or_si128</code>	SSE2	intrin.h	<code>__m128i _mm_or_si128(__m128i, __m128i);</code>
<code>_mm_packs_epi16</code>	SSE2	intrin.h	<code>__m128i _mm_packs_epi16(__m128i, __m128i);</code>
<code>_mm_packs_epi32</code>	SSE2	intrin.h	<code>__m128i _mm_packs_epi32(__m128i, __m128i);</code>
<code>_mm_packus_epi16</code>	SSE2	intrin.h	<code>__m128i _mm_packus_epi16(__m128i, __m128i);</code>
<code>_mm_packus_epi32</code>	SSE41	intrin.h	<code>__m128i _mm_packus_epi32(__m128i, __m128i);</code>
<code>_mm_pause</code>	SSE2	intrin.h	<code>void _mm_pause(void);</code>
<code>_mm_perm_epi8</code>	XOP	ammintrin.h	<code>__m128i _mm_perm_epi8(__m128i, __m128i, __m128i);</code>
<code>_mm_permute_pd</code>	AVX	immintrin.h	<code>__m128d _mm_permute_pd(__m128d, int);</code>

内部函数名称	技术	标头	函数原型
<a href="#">_mm_permute_ps</a>	AVX	immintrin.h	<code>__m128 _mm_permute_ps(__m128, int);</code>
<a href="#">_mm_permute2_pd</a>	XOP	ammintrin.h	<code>__m128d _mm_permute2_pd(__m128d, __m128d, __m128i, int);</code>
<a href="#">_mm_permute2_ps</a>	XOP	ammintrin.h	<code>__m128 _mm_permute2_ps(__m128, __m128, __m128i, int);</code>
<a href="#">_mm_permutevar_pd</a>	AVX	immintrin.h	<code>__m128d _mm_permutevar_pd(__m128d, __m128i);</code>
<a href="#">_mm_permutevar_ps</a>	AVX	immintrin.h	<code>__m128 _mm_permutevar_ps(__m128, __m128i);</code>
<a href="#">_mm_popcnt_u32</a>	POPCNT	intrin.h	<code>int _mm_popcnt_u32(unsigned int);</code>
<a href="#">_mm_popcnt_u64</a>	POPCNT	intrin.h	<code>_int64 _mm_popcnt_u64(unsigned _int64);</code>
<a href="#">_mm_prefetch</a>	SSE	intrin.h	<code>void _mm_prefetch(char*, int);</code>
<a href="#">_mm_rcp_ps</a>	SSE	intrin.h	<code>__m128 _mm_rcp_ps(__m128);</code>
<a href="#">_mm_rcp_ss</a>	SSE	intrin.h	<code>__m128 _mm_rcp_ss(__m128);</code>
<a href="#">_mm_rot_epi16</a>	XOP	ammintrin.h	<code>__m128i _mm_rot_epi16(__m128i, __m128i);</code>
<a href="#">_mm_rot_epi32</a>	XOP	ammintrin.h	<code>__m128i _mm_rot_epi32(__m128i, __m128i);</code>
<a href="#">_mm_rot_epi64</a>	XOP	ammintrin.h	<code>__m128i _mm_rot_epi64(__m128i, __m128i);</code>
<a href="#">_mm_rot_epi8</a>	XOP	ammintrin.h	<code>__m128i _mm_rot_epi8(__m128i, __m128i);</code>
<a href="#">_mm_roti_epi16</a>	XOP	ammintrin.h	<code>__m128i _mm_roti_epi16(__m128i, int);</code>
<a href="#">_mm_roti_epi32</a>	XOP	ammintrin.h	<code>__m128i _mm_roti_epi32(__m128i, int);</code>
<a href="#">_mm_roti_epi64</a>	XOP	ammintrin.h	<code>__m128i _mm_roti_epi64(__m128i, int);</code>
<a href="#">_mm_roti_epi8</a>	XOP	ammintrin.h	<code>__m128i _mm_roti_epi8(__m128i, int);</code>
<a href="#">_mm_round_pd</a>	SSE41	intrin.h	<code>__m128d _mm_round_pd(__m128d, const int);</code>
<a href="#">_mm_round_ps</a>	SSE41	intrin.h	<code>__m128 _mm_round_ps(__m128, const int);</code>
<a href="#">_mm_round_sd</a>	SSE41	intrin.h	<code>__m128d _mm_round_sd(__m128d, __m128d, const int);</code>
<a href="#">_mm_round_ss</a>	SSE41	intrin.h	<code>__m128 _mm_round_ss(__m128, __m128, const int);</code>
<a href="#">_mm_rsqrt_ps</a>	SSE	intrin.h	<code>__m128 _mm_rsqrt_ps(__m128);</code>
<a href="#">_mm_rsqrt_ss</a>	SSE	intrin.h	<code>__m128 _mm_rsqrt_ss(__m128);</code>
<a href="#">_mm_sad_epu8</a>	SSE2	intrin.h	<code>__m128i _mm_sad_epu8(__m128i, __m128i);</code>
<a href="#">_mm_set_epi16</a>	SSE2	intrin.h	<code>__m128i _mm_set_epi16(short, short, short, short, short, short, short, short);</code>
<a href="#">_mm_set_epi32</a>	SSE2	intrin.h	<code>__m128i _mm_set_epi32(int, int, int, int);</code>
<a href="#">_mm_set_epi64x</a>	SSE2	intrin.h	<code>__m128i _mm_set_epi64x(_int64, _int64);</code>
<a href="#">_mm_set_epi8</a>	SSE2	intrin.h	<code>__m128i _mm_set_epi8(char, char, char);</code>
<a href="#">_mm_set_pd</a>	SSE2	intrin.h	<code>__m128d _mm_set_pd(double, double);</code>

内部函数名称	技术	标头	函数原型
<a href="#">_mm_set_ps</a>	SSE	intrin.h	<code>__m128 _mm_set_ps(float, float, float, float);</code>
<a href="#">_mm_set_ps1</a>	SSE	intrin.h	<code>__m128 _mm_set_ps1(float);</code>
<a href="#">_mm_set_sd</a>	SSE2	intrin.h	<code>__m128d _mm_set_sd(double);</code>
<a href="#">_mm_set_ss</a>	SSE	intrin.h	<code>__m128 _mm_set_ss(float);</code>
<a href="#">_mm_set1_epi16</a>	SSE2	intrin.h	<code>__m128i _mm_set1_epi16(short);</code>
<a href="#">_mm_set1_epi32</a>	SSE2	intrin.h	<code>__m128i _mm_set1_epi32(int);</code>
<a href="#">_mm_set1_epi64x</a>	SSE2	intrin.h	<code>__m128i _mm_set1_epi64x(__int64);</code>
<a href="#">_mm_set1_epi8</a>	SSE2	intrin.h	<code>__m128i _mm_set1_epi8(char);</code>
<a href="#">_mm_set1_pd</a>	SSE2	intrin.h	<code>__m128d _mm_set1_pd(double);</code>
<a href="#">_mm_setcsr</a>	SSE	intrin.h	<code>void _mm_setcsr(unsigned int);</code>
<a href="#">_mm_setl_epi64</a>	SSE2	intrin.h	<code>__m128i _mm_setl_epi64(__m128i);</code>
<a href="#">_mm_setr_epi16</a>	SSE2	intrin.h	<code>__m128i _mm_setr_epi16(short, short, short, short, short, short, short, short);</code>
<a href="#">_mm_setr_epi32</a>	SSE2	intrin.h	<code>__m128i _mm_setr_epi32(int, int, int, int);</code>
<a href="#">_mm_setr_epi8</a>	SSE2	intrin.h	<code>__m128i _mm_setr_epi8(char, char, char);</code>
<a href="#">_mm_setr_pd</a>	SSE2	intrin.h	<code>__m128d _mm_setr_pd(double, double);</code>
<a href="#">_mm_setr_ps</a>	SSE	intrin.h	<code>__m128 _mm_setr_ps(float, float, float, float);</code>
<a href="#">_mm_setzero_pd</a>	SSE2	intrin.h	<code>__m128d _mm_setzero_pd(void);</code>
<a href="#">_mm_setzero_ps</a>	SSE	intrin.h	<code>__m128 _mm_setzero_ps(void);</code>
<a href="#">_mm_setzero_si128</a>	SSE2	intrin.h	<code>__m128i _mm_setzero_si128(void);</code>
<a href="#">_mm_sfence</a>	SSE	intrin.h	<code>void _mm_sfence(void);</code>
<a href="#">_mm_sha_epi16</a>	XOP	ammintrin.h	<code>__m128i _mm_sha_epi16(__m128i, __m128i);</code>
<a href="#">_mm_sha_epi32</a>	XOP	ammintrin.h	<code>__m128i _mm_sha_epi32(__m128i, __m128i);</code>
<a href="#">_mm_sha_epi64</a>	XOP	ammintrin.h	<code>__m128i _mm_sha_epi64(__m128i, __m128i);</code>
<a href="#">_mm_sha_epi8</a>	XOP	ammintrin.h	<code>__m128i _mm_sha_epi8(__m128i, __m128i);</code>
<a href="#">_mm_shl_epi16</a>	XOP	ammintrin.h	<code>__m128i _mm_shl_epi16(__m128i, __m128i);</code>
<a href="#">_mm_shl_epi32</a>	XOP	ammintrin.h	<code>__m128i _mm_shl_epi32(__m128i, __m128i);</code>
<a href="#">_mm_shl_epi64</a>	XOP	ammintrin.h	<code>__m128i _mm_shl_epi64(__m128i, __m128i);</code>
<a href="#">_mm_shl_epi8</a>	XOP	ammintrin.h	<code>__m128i _mm_shl_epi8(__m128i, __m128i);</code>
<a href="#">_mm_shuffle_epi32</a>	SSE2	intrin.h	<code>__m128i _mm_shuffle_epi32(__m128i, int);</code>
<a href="#">_mm_shuffle_epi8</a>	SSSE3	intrin.h	<code>__m128i _mm_shuffle_epi8(__m128i, __m128i);</code>
<a href="#">_mm_shuffle_pd</a>	SSE2	intrin.h	<code>__m128d _mm_shuffle_pd(__m128d, __m128d, int);</code>

内部函数名称	技术	标头	函数原型
<a href="#">_mm_shuffle_ps</a>	SSE	intrin.h	<code>__m128 _mm_shuffle_ps(__m128, __m128, unsigned int);</code>
<a href="#">_mm_shufflehi_epi16</a>	SSE2	intrin.h	<code>__m128i _mm_shufflehi_epi16(__m128i, int);</code>
<a href="#">_mm_shufflelo_epi16</a>	SSE2	intrin.h	<code>__m128i _mm_shufflelo_epi16(__m128i, int);</code>
<a href="#">_mm_sign_epi16</a>	SSSE3	intrin.h	<code>__m128i _mm_sign_epi16(__m128i, __m128i);</code>
<a href="#">_mm_sign_epi32</a>	SSSE3	intrin.h	<code>__m128i _mm_sign_epi32(__m128i, __m128i);</code>
<a href="#">_mm_sign_epi8</a>	SSSE3	intrin.h	<code>__m128i _mm_sign_epi8(__m128i, __m128i);</code>
<a href="#">_mm_sll_epi16</a>	SSE2	intrin.h	<code>__m128i _mm_sll_epi16(__m128i, __m128i);</code>
<a href="#">_mm_sll_epi32</a>	SSE2	intrin.h	<code>__m128i _mm_sll_epi32(__m128i, __m128i);</code>
<a href="#">_mm_sll_epi64</a>	SSE2	intrin.h	<code>__m128i _mm_sll_epi64(__m128i, __m128i);</code>
<a href="#">_mm_slli_epi16</a>	SSE2	intrin.h	<code>__m128i _mm_slli_epi16(__m128i, int);</code>
<a href="#">_mm_slli_epi32</a>	SSE2	intrin.h	<code>__m128i _mm_slli_epi32(__m128i, int);</code>
<a href="#">_mm_slli_epi64</a>	SSE2	intrin.h	<code>__m128i _mm_slli_epi64(__m128i, int);</code>
<a href="#">_mm_slli_si128</a>	SSE2	intrin.h	<code>__m128i _mm_slli_si128(__m128i, int);</code>
<a href="#">_mm_sllv_epi32</a>	AVX2	immintrin.h	<code>__m128i _mm_sllv_epi32(__m128i, __m128i);</code>
<a href="#">_mm_sllv_epi64</a>	AVX2	immintrin.h	<code>__m128i _mm_sllv_epi64(__m128i, __m128i);</code>
<a href="#">_mm_sqrt_pd</a>	SSE2	intrin.h	<code>__m128d _mm_sqrt_pd(__m128d);</code>
<a href="#">_mm_sqrt_ps</a>	SSE	intrin.h	<code>__m128 _mm_sqrt_ps(__m128);</code>
<a href="#">_mm_sqrt_sd</a>	SSE2	intrin.h	<code>__m128d _mm_sqrt_sd(__m128d, __m128d);</code>
<a href="#">_mm_sqrt_ss</a>	SSE	intrin.h	<code>__m128 _mm_sqrt_ss(__m128);</code>
<a href="#">_mm_sra_epi16</a>	SSE2	intrin.h	<code>__m128i _mm_sra_epi16(__m128i, __m128i);</code>
<a href="#">_mm_sra_epi32</a>	SSE2	intrin.h	<code>__m128i _mm_sra_epi32(__m128i, __m128i);</code>
<a href="#">_mm_srai_epi16</a>	SSE2	intrin.h	<code>__m128i _mm_srai_epi16(__m128i, int);</code>
<a href="#">_mm_srai_epi32</a>	SSE2	intrin.h	<code>__m128i _mm_srai_epi32(__m128i, int);</code>
<a href="#">_mm_srav_epi32</a>	AVX2	immintrin.h	<code>__m128i _mm_srav_epi32(__m128i, __m128i);</code>
<a href="#">_mm_srl_epi16</a>	SSE2	intrin.h	<code>__m128i _mm_srl_epi16(__m128i, __m128i);</code>
<a href="#">_mm_srl_epi32</a>	SSE2	intrin.h	<code>__m128i _mm_srl_epi32(__m128i, __m128i);</code>
<a href="#">_mm_srl_epi64</a>	SSE2	intrin.h	<code>__m128i _mm_srl_epi64(__m128i, __m128i);</code>
<a href="#">_mm_srli_epi16</a>	SSE2	intrin.h	<code>__m128i _mm_srli_epi16(__m128i, int);</code>
<a href="#">_mm_srli_epi32</a>	SSE2	intrin.h	<code>__m128i _mm_srli_epi32(__m128i, int);</code>
<a href="#">_mm_srli_epi64</a>	SSE2	intrin.h	<code>__m128i _mm_srli_epi64(__m128i, int);</code>
<a href="#">_mm_srli_si128</a>	SSE2	intrin.h	<code>__m128i _mm_srli_si128(__m128i, int);</code>
<a href="#">_mm_srlv_epi32</a>	AVX2	immintrin.h	<code>__m128i _mm_srlv_epi32(__m128i, __m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_srlv_epi64 <sup>2</sup>	AVX2	immintrin.h	<code>__m128i _mm_srlv_epi64(__m128i, __m128i);</code>
_mm_store_pd <sup>2</sup>	SSE2	intrin.h	<code>void _mm_store_pd(double*, __m128d);</code>
_mm_store_ps <sup>2</sup>	SSE	intrin.h	<code>void _mm_store_ps(float*, __m128);</code>
_mm_store_ps1 <sup>2</sup>	SSE	intrin.h	<code>void _mm_store_ps1(float*, __m128);</code>
_mm_store_sd <sup>2</sup>	SSE2	intrin.h	<code>void _mm_store_sd(double*, __m128d);</code>
_mm_store_si128 <sup>2</sup>	SSE2	intrin.h	<code>void _mm_store_si128(__m128i*, __m128i);</code>
_mm_store_ss <sup>2</sup>	SSE	intrin.h	<code>void _mm_store_ss(float*, __m128);</code>
_mm_store1_pd <sup>2</sup>	SSE2	intrin.h	<code>void _mm_store1_pd(double*, __m128d);</code>
_mm_storeh_pd <sup>2</sup>	SSE2	intrin.h	<code>void _mm_storeh_pd(double*, __m128d);</code>
_mm_storeh_pi <sup>2</sup>	SSE	intrin.h	<code>void _mm_storeh_pi(__m64*, __m128);</code>
_mm_storl_epi64 <sup>2</sup>	SSE2	intrin.h	<code>void _mm_storl_epi64(__m128i*, __m128i);</code>
_mm_storl_pd <sup>2</sup>	SSE2	intrin.h	<code>void _mm_storl_pd(double*, __m128d);</code>
_mm_storl_pi <sup>2</sup>	SSE	intrin.h	<code>void _mm_storl_pi(__m64*, __m128);</code>
_mm_storer_pd <sup>2</sup>	SSE2	intrin.h	<code>void _mm_storer_pd(double*, __m128d);</code>
_mm_storer_ps <sup>2</sup>	SSE	intrin.h	<code>void _mm_storer_ps(float*, __m128);</code>
_mm_storeu_pd <sup>2</sup>	SSE2	intrin.h	<code>void _mm_storeu_pd(double*, __m128d);</code>
_mm_storeu_ps <sup>2</sup>	SSE	intrin.h	<code>void _mm_storeu_ps(float*, __m128);</code>
_mm_storeu_si128 <sup>2</sup>	SSE2	intrin.h	<code>void _mm_storeu_si128(__m128i*, __m128i);</code>
_mm_stream_load_si128 <sup>2</sup>	SSE41	intrin.h	<code>__m128i _mm_stream_load_si128(__m128i*);</code>
_mm_stream_pd <sup>2</sup>	SSE2	intrin.h	<code>void _mm_stream_pd(double*, __m128d);</code>
_mm_stream_ps <sup>2</sup>	SSE	intrin.h	<code>void _mm_stream_ps(float*, __m128);</code>
_mm_stream_sd	SSE4a	intrin.h	<code>void _mm_stream_sd(double*, __m128d);</code>
_mm_stream_si128 <sup>2</sup>	SSE2	intrin.h	<code>void _mm_stream_si128(__m128i*, __m128i);</code>
_mm_stream_si32 <sup>2</sup>	SSE2	intrin.h	<code>void _mm_stream_si32(int*, int);</code>
_mm_stream_si64x	SSE2	intrin.h	<code>void _mm_stream_si64x(__int64 *, __int64);</code>
_mm_stream_ss	SSE4a	intrin.h	<code>void _mm_stream_ss(float*, __m128);</code>
_mm_sub_epi16 <sup>2</sup>	SSE2	intrin.h	<code>__m128i _mm_sub_epi16(__m128i, __m128i);</code>
_mm_sub_epi32 <sup>2</sup>	SSE2	intrin.h	<code>__m128i _mm_sub_epi32(__m128i, __m128i);</code>
_mm_sub_epi64 <sup>2</sup>	SSE2	intrin.h	<code>__m128i _mm_sub_epi64(__m128i, __m128i);</code>
_mm_sub_epi8 <sup>2</sup>	SSE2	intrin.h	<code>__m128i _mm_sub_epi8(__m128i, __m128i);</code>
_mm_sub_pd <sup>2</sup>	SSE2	intrin.h	<code>__m128d _mm_sub_pd(__m128d, __m128d);</code>
_mm_sub_ps <sup>2</sup>	SSE	intrin.h	<code>__m128 _mm_sub_ps(__m128, __m128);</code>
_mm_sub_sd <sup>2</sup>	SSE2	intrin.h	<code>__m128d _mm_sub_sd(__m128d, __m128d);</code>

内部函数名称	技术	标头	函数原型
_mm_sub_ss ↗	SSE	intrin.h	<code>__m128 _mm_sub_ss(__m128, __m128);</code>
_mm_subs_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_subs_epi16(__m128i, __m128i);</code>
_mm_subs_epi8 ↗	SSE2	intrin.h	<code>__m128i _mm_subs_epi8(__m128i, __m128i);</code>
_mm_subs_epu16 ↗	SSE2	intrin.h	<code>__m128i _mm_subs_epu16(__m128i, __m128i);</code>
_mm_subs_epu8 ↗	SSE2	intrin.h	<code>__m128i _mm_subs_epu8(__m128i, __m128i);</code>
_mm_testc_pd ↗	AVX	immintrin.h	<code>int _mm_testc_pd(__m128d, __m128d);</code>
_mm_testc_ps ↗	AVX	immintrin.h	<code>int _mm_testc_ps(__m128, __m128);</code>
_mm_testc_si128 ↗	SSE41	intrin.h	<code>int _mm_testc_si128(__m128i, __m128i);</code>
_mm_testnzc_pd ↗	AVX	immintrin.h	<code>int _mm_testnzc_pd(__m128d, __m128d);</code>
_mm_testnzc_ps ↗	AVX	immintrin.h	<code>int _mm_testnzc_ps(__m128, __m128);</code>
_mm_testnzc_si128 ↗	SSE41	intrin.h	<code>int _mm_testnzc_si128(__m128i, __m128i);</code>
_mm_testz_pd ↗	AVX	immintrin.h	<code>int _mm_testz_pd(__m128d, __m128d);</code>
_mm_testz_ps ↗	AVX	immintrin.h	<code>int _mm_testz_ps(__m128, __m128);</code>
_mm_testz_si128 ↗	SSE41	intrin.h	<code>int _mm_testz_si128(__m128i, __m128i);</code>
_mm_ucomieq_sd ↗	SSE2	intrin.h	<code>int _mm_ucomieq_sd(__m128d, __m128d);</code>
_mm_ucomieq_ss ↗	SSE	intrin.h	<code>int _mm_ucomieq_ss(__m128, __m128);</code>
_mm_ucomige_sd ↗	SSE2	intrin.h	<code>int _mm_ucomige_sd(__m128d, __m128d);</code>
_mm_ucomige_ss ↗	SSE	intrin.h	<code>int _mm_ucomige_ss(__m128, __m128);</code>
_mm_ucomigt_sd ↗	SSE2	intrin.h	<code>int _mm_ucomigt_sd(__m128d, __m128d);</code>
_mm_ucomigt_ss ↗	SSE	intrin.h	<code>int _mm_ucomigt_ss(__m128, __m128);</code>
_mm_ucomile_sd ↗	SSE2	intrin.h	<code>int _mm_ucomile_sd(__m128d, __m128d);</code>
_mm_ucomile_ss ↗	SSE	intrin.h	<code>int _mm_ucomile_ss(__m128, __m128);</code>
_mm_ucomilt_sd ↗	SSE2	intrin.h	<code>int _mm_ucomilt_sd(__m128d, __m128d);</code>
_mm_ucomilt_ss ↗	SSE	intrin.h	<code>int _mm_ucomilt_ss(__m128, __m128);</code>
_mm_ucomineq_sd ↗	SSE2	intrin.h	<code>int _mm_ucomineq_sd(__m128d, __m128d);</code>
_mm_ucomineq_ss ↗	SSE	intrin.h	<code>int _mm_ucomineq_ss(__m128, __m128);</code>
_mm_unpackhi_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_unpackhi_epi16(__m128i, __m128i);</code>
_mm_unpackhi_epi32 ↗	SSE2	intrin.h	<code>__m128i _mm_unpackhi_epi32(__m128i, __m128i);</code>
_mm_unpackhi_epi64 ↗	SSE2	intrin.h	<code>__m128i _mm_unpackhi_epi64(__m128i, __m128i);</code>
_mm_unpackhi_epi8 ↗	SSE2	intrin.h	<code>__m128i _mm_unpackhi_epi8(__m128i, __m128i);</code>
_mm_unpackhi_pd ↗	SSE2	intrin.h	<code>__m128d _mm_unpackhi_pd(__m128d, __m128d);</code>
_mm_unpackhi_ps ↗	SSE	intrin.h	<code>__m128 _mm_unpackhi_ps(__m128, __m128);</code>
_mm_unpacklo_epi16 ↗	SSE2	intrin.h	<code>__m128i _mm_unpacklo_epi16(__m128i, __m128i);</code>

内部函数名称	技术	标头	函数原型
_mm_unpacklo_epi32 <sup>2</sup>	SSE2	intrin.h	<code>__m128i _mm_unpacklo_epi32(__m128i, __m128i);</code>
_mm_unpacklo_epi64 <sup>2</sup>	SSE2	intrin.h	<code>__m128i _mm_unpacklo_epi64(__m128i, __m128i);</code>
_mm_unpacklo_epi8 <sup>2</sup>	SSE2	intrin.h	<code>__m128i _mm_unpacklo_epi8(__m128i, __m128i);</code>
_mm_unpacklo_pd <sup>2</sup>	SSE2	intrin.h	<code>__m128d _mm_unpacklo_pd(__m128d, __m128d);</code>
_mm_unpacklo_ps <sup>2</sup>	SSE	intrin.h	<code>__m128 _mm_unpacklo_ps(__m128, __m128);</code>
_mm_xor_pd <sup>2</sup>	SSE2	intrin.h	<code>__m128d _mm_xor_pd(__m128d, __m128d);</code>
_mm_xor_ps <sup>2</sup>	SSE	intrin.h	<code>__m128 _mm_xor_ps(__m128, __m128);</code>
_mm_xor_si128 <sup>2</sup>	SSE2	intrin.h	<code>__m128i _mm_xor_si128(__m128i, __m128i);</code>
_mm256_abs_epi16 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_abs_epi16(__m256i);</code>
_mm256_abs_epi32 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_abs_epi32(__m256i);</code>
_mm256_abs_epi8 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_abs_epi8(__m256i);</code>
_mm256_add_epi16 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_add_epi16(__m256i, __m256i);</code>
_mm256_add_epi32 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_add_epi32(__m256i, __m256i);</code>
_mm256_add_epi64 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_add_epi64(__m256i, __m256i);</code>
_mm256_add_epi8 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_add_epi8(__m256i, __m256i);</code>
_mm256_add_pd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_add_pd(__m256d, __m256d);</code>
_mm256_add_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_add_ps(__m256, __m256);</code>
_mm256_adds_epi16 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_adds_epi16(__m256i, __m256i);</code>
_mm256_adds_epi8 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_adds_epi8(__m256i, __m256i);</code>
_mm256_adds_epu16 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_adds_epu16(__m256i, __m256i);</code>
_mm256_adds_epu8 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_adds_epu8(__m256i, __m256i);</code>
_mm256_addsub_pd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_addsub_pd(__m256d, __m256d);</code>
_mm256_addsub_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_addsub_ps(__m256, __m256);</code>
_mm256_alignr_epi8 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_alignr_epi8(__m256i, __m256i, const int);</code>
_mm256_and_pd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_and_pd(__m256d, __m256d);</code>
_mm256_and_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_and_ps(__m256, __m256);</code>
_mm256_and_si256 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_and_si256(__m256i, __m256i);</code>
_mm256_andnot_pd <sup>2</sup>	AVX	immintrin.h	<code>__m256d _mm256_andnot_pd(__m256d, __m256d);</code>
_mm256_andnot_ps <sup>2</sup>	AVX	immintrin.h	<code>__m256 _mm256_andnot_ps(__m256, __m256);</code>
_mm256_andnot_si256 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_andnot_si256(__m256i, __m256i);</code>
_mm256_avg_epu16 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_avg_epu16(__m256i, __m256i);</code>
_mm256_avg_epu8 <sup>2</sup>	AVX2	immintrin.h	<code>__m256i _mm256_avg_epu8(__m256i, __m256i);</code>

内部函数名称	技术	标头	函数原型
<a href="#">_mm256_blend_epi16</a>	AVX2	immintrin.h	<code>__m256i _mm256_blend_epi16(__m256i, __m256i, const int);</code>
<a href="#">_mm256_blend_epi32</a>	AVX2	immintrin.h	<code>__m256i _mm256_blend_epi32(__m256i, __m256i, const int);</code>
<a href="#">_mm256_blend_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_blend_pd(__m256d, __m256d, const int);</code>
<a href="#">_mm256_blend_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_blend_ps(__m256, __m256, const int);</code>
<a href="#">_mm256_blendv_epi8</a>	AVX2	immintrin.h	<code>__m256i _mm256_blendv_epi8(__m256i, __m256i, __m256i);</code>
<a href="#">_mm256_blendv_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_blendv_pd(__m256d, __m256d, __m256d);</code>
<a href="#">_mm256_blendv_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_blendv_ps(__m256, __m256, __m256);</code>
<a href="#">_mm256_broadcast_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_broadcast_pd(__m128d const *);</code>
<a href="#">_mm256_broadcast_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_broadcast_ps(__m128 const *);</code>
<a href="#">_mm256_broadcast_sd</a>	AVX	immintrin.h	<code>__m256d _mm256_broadcast_sd(double const *);</code>
<a href="#">_mm256_broadcast_ss</a>	AVX	immintrin.h	<code>__m256 _mm256_broadcast_ss(float const *);</code>
<a href="#">_mm256_broadcastb_epi8</a>	AVX2	immintrin.h	<code>__m256i _mm256_broadcastb_epi8 (__m128i);</code>
<a href="#">_mm256_broadcastd_epi32</a>	AVX2	immintrin.h	<code>__m256i _mm256_broadcastd_epi32(__m128i);</code>
<a href="#">_mm256_broadcastq_epi64</a>	AVX2	immintrin.h	<code>__m256i _mm256_broadcastq_epi64(__m128i);</code>
<a href="#">_mm256_broadcastsd_pd</a>	AVX2	immintrin.h	<code>__m256d _mm256_broadcastsd_pd(__m128d);</code>
<a href="#">_mm256_broadcastsi128_si256</a>	AVX2	immintrin.h	<code>__m256i _mm256_broadcastsi128_si256(__m128i);</code>
<a href="#">_mm256_broadcastss_ps</a>	AVX2	immintrin.h	<code>__m256 _mm256_broadcastss_ps(__m128);</code>
<a href="#">_mm256_broadcastw_epi16</a>	AVX2	immintrin.h	<code>__m256i _mm256_broadcastw_epi16(__m128i);</code>
<a href="#">_mm256_castpd_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_castpd_ps(__m256d);</code>
<a href="#">_mm256_castpd_si256</a>	AVX	immintrin.h	<code>__m256i _mm256_castpd_si256(__m256d);</code>
<a href="#">_mm256_castpd128_pd256</a>	AVX	immintrin.h	<code>__m256d _mm256_castpd128_pd256(__m128d);</code>
<a href="#">_mm256_castpd256_pd128</a>	AVX	immintrin.h	<code>__m256d _mm256_castpd256_pd128(__m256d);</code>
<a href="#">_mm256_castps_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_castps_pd(__m256);</code>
<a href="#">_mm256_castps_si256</a>	AVX	immintrin.h	<code>__m256i _mm256_castps_si256(__m256);</code>
<a href="#">_mm256_castps128_ps256</a>	AVX	immintrin.h	<code>__m256 _mm256_castps128_ps256(__m128);</code>
<a href="#">_mm256_castps256_ps128</a>	AVX	immintrin.h	<code>__m256 _mm256_castps256_ps128(__m256);</code>
<a href="#">_mm256_castsi128_si256</a>	AVX	immintrin.h	<code>__m256i _mm256_castsi128_si256(__m128i);</code>
<a href="#">_mm256_castsi256_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_castsi256_pd(__m256i);</code>
<a href="#">_mm256_castsi256_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_castsi256_ps(__m256i);</code>
<a href="#">_mm256_castsi256_si128</a>	AVX	immintrin.h	<code>__m256i _mm256_castsi256_si128(__m256i);</code>

内部函数名称	技术	标头	函数原型
<code>_mm256_cmov_si256</code>	XOP	ammintrin.h	<code>__m256i _mm256_cmov_si256(__m256i, __m256i, __m256i);</code>
<code>_mm256_cmp_pd</code>	AVX	immintrin.h	<code>__m256d _mm256_cmp_pd(__m256d, __m256d, const int);</code>
<code>_mm256_cmp_ps</code>	AVX	immintrin.h	<code>__m256 _mm256_cmp_ps(__m256, __m256, const int);</code>
<code>_mm256_cmpeq_epi16</code>	AVX2	immintrin.h	<code>__m256i _mm256_cmpeq_epi16(__m256i, __m256i);</code>
<code>_mm256_cmpeq_epi32</code>	AVX2	immintrin.h	<code>__m256i _mm256_cmpeq_epi32(__m256i, __m256i);</code>
<code>_mm256_cmpeq_epi64</code>	AVX2	immintrin.h	<code>__m256i _mm256_cmpeq_epi64(__m256i, __m256i);</code>
<code>_mm256_cmpeq_epi8</code>	AVX2	immintrin.h	<code>__m256i _mm256_cmpeq_epi8(__m256i, __m256i);</code>
<code>_mm256_cmpgt_epi16</code>	AVX2	immintrin.h	<code>__m256i _mm256_cmpgt_epi16(__m256i, __m256i);</code>
<code>_mm256_cmpgt_epi32</code>	AVX2	immintrin.h	<code>__m256i _mm256_cmpgt_epi32(__m256i, __m256i);</code>
<code>_mm256_cmpgt_epi64</code>	AVX2	immintrin.h	<code>__m256i _mm256_cmpgt_epi64(__m256i, __m256i);</code>
<code>_mm256_cmpgt_epi8</code>	AVX2	immintrin.h	<code>__m256i _mm256_cmpgt_epi8(__m256i, __m256i);</code>
<code>_mm256_cvtepi16_epi32</code>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepi16_epi32(__m128i);</code>
<code>_mm256_cvtepi16_epi64</code>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepi16_epi64(__m128i);</code>
<code>_mm256_cvtepi32_epi64</code>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepi32_epi64(__m128i);</code>
<code>_mm256_cvtepi32_pd</code>	AVX	immintrin.h	<code>__m256d _mm256_cvtepi32_pd(__m128i);</code>
<code>_mm256_cvtepi32_ps</code>	AVX	immintrin.h	<code>__m256 _mm256_cvtepi32_ps(__m256i);</code>
<code>_mm256_cvtepi8_epi16</code>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepi8_epi16(__m128i);</code>
<code>_mm256_cvtepi8_epi32</code>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepi8_epi32(__m128i);</code>
<code>_mm256_cvtepi8_epi64</code>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepi8_epi64(__m128i);</code>
<code>_mm256_cvtepu16_epi32</code>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepu16_epi32(__m128i);</code>
<code>_mm256_cvtepu16_epi64</code>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepu16_epi64(__m128i);</code>
<code>_mm256_cvtepu32_epi64</code>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepu32_epi64(__m128i);</code>
<code>_mm256_cvtepu8_epi16</code>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepu8_epi16(__m128i);</code>
<code>_mm256_cvtepu8_epi32</code>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepu8_epi32(__m128i);</code>
<code>_mm256_cvtepu8_epi64</code>	AVX2	immintrin.h	<code>__m256i _mm256_cvtepu8_epi64(__m128i);</code>
<code>_mm256_cvtpd_epi32</code>	AVX	immintrin.h	<code>__m128i _mm256_cvtpd_epi32(__m256d);</code>
<code>_mm256_cvtpd_ps</code>	AVX	immintrin.h	<code>__m128 _mm256_cvtpd_ps(__m256d);</code>
<code>_mm256_cvtph_ps</code>	F16C	immintrin.h	<code>__m256 _mm256_cvtph_ps(__m128i);</code>
<code>_mm256_cvtps_epi32</code>	AVX	immintrin.h	<code>__m256i _mm256_cvtps_epi32(__m256);</code>
<code>_mm256_cvtps_pd</code>	AVX	immintrin.h	<code>__m256d _mm256_cvtps_pd(__m128);</code>
<code>_mm256_cvtps_ph</code>	F16C	immintrin.h	<code>__m128i _mm256_cvtps_ph(__m256, const int);</code>
<code>_mm256_cvtpd_epi32</code>	AVX	immintrin.h	<code>__m128i _mm256_cvtpd_epi32(__m256d);</code>

内部函数名称	技术	标头	函数原型
_mm256_cvttps_epi32 ↗	AVX	immintrin.h	<code>__m256i _mm256_cvttps_epi32(__m256);</code>
_mm256_div_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_div_pd(__m256d, __m256d);</code>
_mm256_div_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_div_ps(__m256, __m256);</code>
_mm256_dp_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_dp_ps(__m256, __m256, const int);</code>
_mm256_extractf128_pd ↗	AVX	immintrin.h	<code>__m128d _mm256_extractf128_pd(__m256d, const int);</code>
_mm256_extractf128_ps ↗	AVX	immintrin.h	<code>__m128 _mm256_extractf128_ps(__m256, const int);</code>
_mm256_extractf128_si256 ↗	AVX	immintrin.h	<code>__m128i _mm256_extractf128_si256(__m256i, const int);</code>
_mm256_extracti128_si256 ↗	AVX2	immintrin.h	<code>__m128i _mm256_extracti128_si256(__m256i, int);</code>
_mm256_fmadd_pd ↗	FMA	immintrin.h	<code>__m256d _mm256_fmadd_pd (__m256d, __m256d, __m256d);</code>
_mm256_fmadd_ps ↗	FMA	immintrin.h	<code>__m256 _mm256_fmadd_ps (__m256, __m256, __m256);</code>
_mm256_fmaddsub_pd ↗	FMA	immintrin.h	<code>__m256d _mm256_fmaddsub_pd (__m256d, __m256d, __m256d);</code>
_mm256_fmaddsub_ps ↗	FMA	immintrin.h	<code>__m256 _mm256_fmaddsub_ps (__m256, __m256, __m256);</code>
_mm256_fmsub_pd ↗	FMA	immintrin.h	<code>__m256d _mm256_fmsub_pd (__m256d, __m256d, __m256d);</code>
_mm256_fmsub_ps ↗	FMA	immintrin.h	<code>__m256 _mm256_fmsub_ps (__m256, __m256, __m256);</code>
_mm256_fmsubadd_pd ↗	FMA	immintrin.h	<code>__m256d _mm256_fmsubadd_pd (__m256d, __m256d, __m256d);</code>
_mm256_fmsubadd_ps ↗	FMA	immintrin.h	<code>__m256 _mm256_fmsubadd_ps (__m256, __m256, __m256);</code>
_mm256_fnmadd_pd ↗	FMA	immintrin.h	<code>__m256d _mm256_fnmadd_pd (__m256d, __m256d, __m256d);</code>
_mm256_fnmadd_ps ↗	FMA	immintrin.h	<code>__m256 _mm256_fnmadd_ps (__m256, __m256, __m256);</code>
_mm256_fnmsub_pd ↗	FMA	immintrin.h	<code>__m256d _mm256_fnmsub_pd (__m256d, __m256d, __m256d);</code>
_mm256_fnmsub_ps ↗	FMA	immintrin.h	<code>__m256 _mm256_fnmsub_ps (__m256, __m256, __m256);</code>
_mm256_frcz_pd	XOP	ammintrin.h	<code>__m256d _mm256_frcz_pd(__m256d);</code>
_mm256_frcz_ps	XOP	ammintrin.h	<code>__m256 _mm256_frcz_ps(__m256);</code>
_mm256_hadd_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_hadd_epi16(__m256i, __m256i);</code>
_mm256_hadd_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_hadd_epi32(__m256i, __m256i);</code>
_mm256_hadd_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_hadd_pd(__m256d, __m256d);</code>
_mm256_hadd_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_hadd_ps(__m256, __m256);</code>
_mm256_hadds_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_hadds_epi16(__m256i, __m256i);</code>
_mm256_hsub_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_hsub_epi16(__m256i, __m256i);</code>

内部函数名称	技术	标头	函数原型
_mm256_hsub_epi32 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_hsub_epi32(__m256i, __m256i);</code>
_mm256_hsub_pd <sup>↗</sup>	AVX	immintrin.h	<code>__m256d _mm256_hsub_pd(__m256d, __m256d);</code>
_mm256_hsub_ps <sup>↗</sup>	AVX	immintrin.h	<code>__m256 _mm256_hsub_ps(__m256, __m256);</code>
_mm256_hsubs_epi16 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_hsubs_epi16(__m256i, __m256i);</code>
_mm256_i32gather_epi32 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_i32gather_epi32(int const *, __m256i, const int);</code>
_mm256_i32gather_epi64 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_i32gather_epi64(__int64 const *, __m128i, const int);</code>
_mm256_i32gather_pd <sup>↗</sup>	AVX2	immintrin.h	<code>__m256d _mm256_i32gather_pd(double const *, __m128i, const int);</code>
_mm256_i32gather_ps <sup>↗</sup>	AVX2	immintrin.h	<code>__m256 _mm256_i32gather_ps(float const *, __m256i, const int);</code>
_mm256_i64gather_epi32 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_i64gather_epi32(int const *, __m256i, const int);</code>
_mm256_i64gather_epi64 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_i64gather_epi64(__int64 const *, __m256i, const int);</code>
_mm256_i64gather_pd <sup>↗</sup>	AVX2	immintrin.h	<code>__m256d _mm256_i64gather_pd(double const *, __m256i, const int);</code>
_mm256_i64gather_ps <sup>↗</sup>	AVX2	immintrin.h	<code>__m128 _mm256_i64gather_ps(float const *, __m256i, const int);</code>
_mm256_insertf128_pd <sup>↗</sup>	AVX	immintrin.h	<code>__m256d _mm256_insertf128_pd(__m256d, __m128d, int);</code>
_mm256_insertf128_ps <sup>↗</sup>	AVX	immintrin.h	<code>__m256 _mm256_insertf128_ps(__m256, __m128, int);</code>
_mm256_insertf128_si256 <sup>↗</sup>	AVX	immintrin.h	<code>__m256i _mm256_insertf128_si256(__m256i, __m128i, int);</code>
_mm256_inserti128_si256 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_inserti128_si256(__m256i, __m128i, int);</code>
_mm256_lddqu_si256 <sup>↗</sup>	AVX	immintrin.h	<code>__m256i _mm256_lddqu_si256(__m256i *);</code>
_mm256_load_pd <sup>↗</sup>	AVX	immintrin.h	<code>__m256d _mm256_load_pd(double const *);</code>
_mm256_load_ps <sup>↗</sup>	AVX	immintrin.h	<code>__m256 _mm256_load_ps(float const *);</code>
_mm256_load_si256 <sup>↗</sup>	AVX	immintrin.h	<code>__m256i _mm256_load_si256(__m256i *);</code>
_mm256_loadu_pd <sup>↗</sup>	AVX	immintrin.h	<code>__m256d _mm256_loadu_pd(double const *);</code>
_mm256_loadu_ps <sup>↗</sup>	AVX	immintrin.h	<code>__m256 _mm256_loadu_ps(float const *);</code>
_mm256_loadu_si256 <sup>↗</sup>	AVX	immintrin.h	<code>__m256i _mm256_loadu_si256(__m256i *);</code>
_mm256_macc_pd	FMA4	ammintrin.h	<code>__m256d _mm_macc_pd(__m256d, __m256d, __m256d);</code>
_mm256_macc_ps	FMA4	ammintrin.h	<code>__m256 _mm_macc_ps(__m256, __m256, __m256);</code>
_mm256_madd_epi16 <sup>↗</sup>	AVX2	immintrin.h	<code>__m256i _mm256_madd_epi16(__m256i, __m256i);</code>
_mm256_maddsub_pd	FMA4	ammintrin.h	<code>__m256d _mm_maddsub_pd(__m256d, __m256d, __m256d);</code>

内部函数名称	技术	标头	函数原型
_mm256_maddsub_ps	FMA4	ammintrin.h	<code>_m256 _mm_maddsub_ps(_m256, _m256, _m256);</code>
_mm256_maddubs_epi16	AVX2	immintrin.h	<code>_m256i _mm256_maddubs_epi16(_m256i, _m256i);</code>
_mm256_mask_i32gather_epi32	AVX2	immintrin.h	<code>_m256i _mm256_mask_i32gather_epi32(_m256i, int const *, _m256i, _m256i, const int);</code>
_mm256_mask_i32gather_epi64	AVX2	immintrin.h	<code>_m256i _mm256_mask_i32gather_epi64(_m256i, _int64 const *, _m128i, _m256i, const int);</code>
_mm256_mask_i32gather_pd	AVX2	immintrin.h	<code>_m256d _mm256_mask_i32gather_pd(_m256d, double const *, _m128i, _m256d, const int);</code>
_mm256_mask_i32gather_ps	AVX2	immintrin.h	<code>_m256 _mm256_mask_i32gather_ps(_m256, float const *, _m256i, _m256, const int);</code>
_mm256_mask_i64gather_epi32	AVX2	immintrin.h	<code>_m128i _mm256_mask_i64gather_epi32(_m128i, int const *, _m256i, _m128i, const int);</code>
_mm256_mask_i64gather_epi64	AVX2	immintrin.h	<code>_m256i _mm256_mask_i64gather_epi64(_m256i, _int64 const *, _m256i, _m256i, const int);</code>
_mm256_mask_i64gather_pd	AVX2	immintrin.h	<code>_m256d _mm256_mask_i64gather_pd(_m256d, double const *, _m256i, _m256d, const int);</code>
_mm256_mask_i64gather_ps	AVX2	immintrin.h	<code>_m128 _mm256_mask_i64gather_ps(_m128, float const *, _m256i, _m128, const int);</code>
_mm256_maskload_epi32	AVX2	immintrin.h	<code>_m256i _mm256_maskload_epi32(int const *, _m256i);</code>
_mm256_maskload_epi64	AVX2	immintrin.h	<code>_m256i _mm256_maskload_epi64(_int64 const *, _m256i);</code>
_mm256_maskload_pd	AVX	immintrin.h	<code>_m256d _mm256_maskload_pd(double const *, _m256i);</code>
_mm256_maskload_ps	AVX	immintrin.h	<code>_m256 _mm256_maskload_ps(float const *, _m256i);</code>
_mm256_maskstore_epi32	AVX2	immintrin.h	<code>void _mm256_maskstore_epi32(int *, _m256i, _m256i);</code>
_mm256_maskstore_epi64	AVX2	immintrin.h	<code>void _mm256_maskstore_epi64(_int64 *, _m256i, _m256i);</code>
_mm256_maskstore_pd	AVX	immintrin.h	<code>void _mm256_maskstore_pd(double *, _m256i, _m256d);</code>
_mm256_maskstore_ps	AVX	immintrin.h	<code>void _mm256_maskstore_ps(float *, _m256i, _m256);</code>
_mm256_max_epi16	AVX2	immintrin.h	<code>_m256i _mm256_max_epi16(_m256i, _m256i);</code>
_mm256_max_epi32	AVX2	immintrin.h	<code>_m256i _mm256_max_epi32(_m256i, _m256i);</code>
_mm256_max_epi8	AVX2	immintrin.h	<code>_m256i _mm256_max_epi8(_m256i, _m256i);</code>
_mm256_max_epu16	AVX2	immintrin.h	<code>_m256i _mm256_max_epu16(_m256i, _m256i);</code>
_mm256_max_epu32	AVX2	immintrin.h	<code>_m256i _mm256_max_epu32(_m256i, _m256i);</code>
_mm256_max_epu8	AVX2	immintrin.h	<code>_m256i _mm256_max_epu8(_m256i, _m256i);</code>

内部函数名称	技术	标头	函数原型
_mm256_max_pd ↗	AVX	immintrin.h	<code>__m256d __mm256_max_pd(__m256d, __m256d);</code>
_mm256_max_ps ↗	AVX	immintrin.h	<code>__m256 __mm256_max_ps(__m256, __m256);</code>
_mm256_min_ep16 ↗	AVX2	immintrin.h	<code>__m256i __mm256_min_ep16(__m256i, __m256i);</code>
_mm256_min_ep132 ↗	AVX2	immintrin.h	<code>__m256i __mm256_min_ep132(__m256i, __m256i);</code>
_mm256_min_ep8 ↗	AVX2	immintrin.h	<code>__m256i __mm256_min_ep8(__m256i, __m256i);</code>
_mm256_min_epu16 ↗	AVX2	immintrin.h	<code>__m256i __mm256_min_epu16(__m256i, __m256i);</code>
_mm256_min_epu32 ↗	AVX2	immintrin.h	<code>__m256i __mm256_min_epu32(__m256i, __m256i);</code>
_mm256_min_epu8 ↗	AVX2	immintrin.h	<code>__m256i __mm256_min_epu8(__m256i, __m256i);</code>
_mm256_min_pd ↗	AVX	immintrin.h	<code>__m256d __mm256_min_pd(__m256d, __m256d);</code>
_mm256_min_ps ↗	AVX	immintrin.h	<code>__m256 __mm256_min_ps(__m256, __m256);</code>
_mm256_movedup_pd ↗	AVX	immintrin.h	<code>__m256d __mm256_movedup_pd(__m256d);</code>
_mm256_movehdup_ps ↗	AVX	immintrin.h	<code>__m256 __mm256_movehdup_ps(__m256);</code>
_mm256_moveldup_ps ↗	AVX	immintrin.h	<code>__m256 __mm256_moveldup_ps(__m256);</code>
_mm256_movemask_ep18 ↗	AVX2	immintrin.h	<code>int __mm256_movemask_ep18(__m256i);</code>
_mm256_movemask_pd ↗	AVX	immintrin.h	<code>int __mm256_movemask_pd(__m256d);</code>
_mm256_movemask_ps ↗	AVX	immintrin.h	<code>int __mm256_movemask_ps(__m256);</code>
_mm256_mpsadbw_epu8 ↗	AVX2	immintrin.h	<code>__m256i __mm256_mpsadbw_epu8(__m256i, __m256i, const int);</code>
_mm256_msub_pd	FMA4	ammintrin.h	<code>__m256d __mm_msub_pd(__m256d, __m256d, __m256d);</code>
_mm256_msub_ps	FMA4	ammintrin.h	<code>__m256 __mm_msub_ps(__m256, __m256, __m256);</code>
_mm256_msubadd_pd	FMA4	ammintrin.h	<code>__m256d __mm_msubadd_pd(__m256d, __m256d, __m256d);</code>
_mm256_msubadd_ps	FMA4	ammintrin.h	<code>__m256 __mm_msubadd_ps(__m256, __m256, __m256);</code>
_mm256_mul_ep132 ↗	AVX2	immintrin.h	<code>__m256i __mm256_mul_ep132(__m256i, __m256i);</code>
_mm256_mul_epu32 ↗	AVX2	immintrin.h	<code>__m256i __mm256_mul_epu32(__m256i, __m256i);</code>
_mm256_mul_pd ↗	AVX	immintrin.h	<code>__m256d __mm256_mul_pd(__m256d, __m256d);</code>
_mm256_mul_ps ↗	AVX	immintrin.h	<code>__m256 __mm256_mul_ps(__m256, __m256);</code>
_mm256_mulhi_ep16 ↗	AVX2	immintrin.h	<code>__m256i __mm256_mulhi_ep16(__m256i, __m256i);</code>
_mm256_mulhi_epu16 ↗	AVX2	immintrin.h	<code>__m256i __mm256_mulhi_epu16(__m256i, __m256i);</code>
_mm256_mulhrs_ep16 ↗	AVX2	immintrin.h	<code>__m256i __mm256_mulhrs_ep16(__m256i, __m256i);</code>
_mm256_mullo_ep16 ↗	AVX2	immintrin.h	<code>__m256i __mm256_mullo_ep16(__m256i, __m256i);</code>
_mm256_mullo_ep132 ↗	AVX2	immintrin.h	<code>__m256i __mm256_mullo_ep132(__m256i, __m256i);</code>
_mm256_nmacc_pd	FMA4	ammintrin.h	<code>__m256d __mm_nmacc_pd(__m256d, __m256d, __m256d);</code>
_mm256_nmacc_ps	FMA4	ammintrin.h	<code>__m256 __mm_nmacc_ps(__m256, __m256, __m256);</code>

内部函数名称	技术	标头	函数原型
<code>_mm256_nmsub_pd</code>	FMA4	ammintrin.h	<code>__m256d _mm_nmsub_pd(__m256d, __m256d, __m256d);</code>
<code>_mm256_nmsub_ps</code>	FMA4	ammintrin.h	<code>__m256 _mm_nmsub_ps(__m256, __m256, __m256);</code>
<code>_mm256_or_pd</code>	AVX	immintrin.h	<code>__m256d _mm256_or_pd(__m256d, __m256d);</code>
<code>_mm256_or_ps</code>	AVX	immintrin.h	<code>__m256 _mm256_or_ps(__m256, __m256);</code>
<code>_mm256_or_si256</code>	AVX2	immintrin.h	<code>__m256i _mm256_or_si256(__m256i, __m256i);</code>
<code>_mm256_packs_epi16</code>	AVX2	immintrin.h	<code>__m256i _mm256_packs_epi16(__m256i, __m256i);</code>
<code>_mm256_packs_epi32</code>	AVX2	immintrin.h	<code>__m256i _mm256_packs_epi32(__m256i, __m256i);</code>
<code>_mm256_packus_epi16</code>	AVX2	immintrin.h	<code>__m256i _mm256_packus_epi16(__m256i, __m256i);</code>
<code>_mm256_packus_epi32</code>	AVX2	immintrin.h	<code>__m256i _mm256_packus_epi32(__m256i, __m256i);</code>
<code>_mm256_permute_pd</code>	AVX	immintrin.h	<code>__m256d _mm256_permute_pd(__m256d, int);</code>
<code>_mm256_permute_ps</code>	AVX	immintrin.h	<code>__m256 _mm256_permute_ps(__m256, int);</code>
<code>_mm256_permute2_pd</code>	XOP	ammintrin.h	<code>__m256d _mm256_permute2_pd(__m256d, __m256d, __m256i, int);</code>
<code>_mm256_permute2_ps</code>	XOP	ammintrin.h	<code>__m256 _mm256_permute2_ps(__m256, __m256, __m256i, int);</code>
<code>_mm256_permute2f128_pd</code>	AVX	immintrin.h	<code>__m256d _mm256_permute2f128_pd(__m256d, __m256d, int);</code>
<code>_mm256_permute2f128_ps</code>	AVX	immintrin.h	<code>__m256 _mm256_permute2f128_ps(__m256, __m256, int);</code>
<code>_mm256_permute2f128_si256</code>	AVX	immintrin.h	<code>__m256i _mm256_permute2f128_si256(__m256i, __m256i, int);</code>
<code>_mm256_permute2x128_si256</code>	AVX2	immintrin.h	<code>__m256i _mm256_permute2x128_si256(__m256i, __m256i, const int);</code>
<code>_mm256_permute4x64_epi64</code>	AVX2	immintrin.h	<code>__m256i _mm256_permute4x64_epi64 (__m256i, const int);</code>
<code>_mm256_permute4x64_pd</code>	AVX2	immintrin.h	<code>__m256d _mm256_permute4x64_pd(__m256d, const int);</code>
<code>_mm256_permutevar_pd</code>	AVX	immintrin.h	<code>__m256d _mm256_permutevar_pd(__m256d, __m256i);</code>
<code>_mm256_permutevar_ps</code>	AVX	immintrin.h	<code>__m256 _mm256_permutevar_ps(__m256, __m256i);</code>
<code>_mm256_permutevar8x32_epi32</code>	AVX2	immintrin.h	<code>__m256i _mm256_permutevar8x32_epi32(__m256i, __m256i);</code>
<code>_mm256_permutevar8x32_ps</code>	AVX2	immintrin.h	<code>__m256 _mm256_permutevar8x32_ps (__m256, __m256i);</code>
<code>_mm256_rcp_ps</code>	AVX	immintrin.h	<code>__m256 _mm256_rcp_ps(__m256);</code>
<code>_mm256_round_pd</code>	AVX	immintrin.h	<code>__m256d _mm256_round_pd(__m256d, int);</code>
<code>_mm256_round_ps</code>	AVX	immintrin.h	<code>__m256 _mm256_round_ps(__m256, int);</code>
<code>_mm256_sqrt_ps</code>	AVX	immintrin.h	<code>__m256 _mm256_sqrt_ps(__m256);</code>
<code>_mm256_sad_epu8</code>	AVX2	immintrin.h	<code>__m256i _mm256_sad_epu8(__m256i, __m256i);</code>

内部函数名称	技术	标头	函数原型
<a href="#">_mm256_set_epi16</a>	AVX	immintrin.h	<code>(__m256i _mm256_set_epi16(short, short, short);</code>
<a href="#">_mm256_set_epi32</a>	AVX	immintrin.h	<code>__m256i _mm256_set_epi32(int, int, int, int, int, int, int, int);</code>
<a href="#">_mm256_set_epi64x</a>	AVX	immintrin.h	<code>__m256i _mm256_set_epi64x(long long, long long, long long, long long);</code>
<a href="#">_mm256_set_epi8</a>	AVX	immintrin.h	<code>__m256i _mm256_set_epi8(char, char, char);</code>
<a href="#">_mm256_set_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_set_pd(double, double, double, double);</code>
<a href="#">_mm256_set_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_set_ps(float, float, float, float, float, float, float, float);</code>
<a href="#">_mm256_set1_epi16</a>	AVX	immintrin.h	<code>__m256i _mm256_set1_epi16(short);</code>
<a href="#">_mm256_set1_epi32</a>	AVX	immintrin.h	<code>__m256i _mm256_set1_epi32(int);</code>
<a href="#">_mm256_set1_epi64x</a>	AVX	immintrin.h	<code>__m256i _mm256_set1_epi64x(long long);</code>
<a href="#">_mm256_set1_epi8</a>	AVX	immintrin.h	<code>__m256i _mm256_set1_epi8(char);</code>
<a href="#">_mm256_set1_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_set1_pd(double);</code>
<a href="#">_mm256_set1_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_set1_ps(float);</code>
<a href="#">_mm256_setr_epi16</a>	AVX	immintrin.h	<code>(__m256i _mm256_setr_epi16(short, short, short);</code>
<a href="#">_mm256_setr_epi32</a>	AVX	immintrin.h	<code>__m256i _mm256_setr_epi32(int, int, int, int, int, int, int, int);</code>
<a href="#">_mm256_setr_epi64x</a>	AVX	immintrin.h	<code>__m256i _mm256_setr_epi64x(long long, long long, long long, long long);</code>
<a href="#">_mm256_setr_epi8</a>	AVX	immintrin.h	<code>(__m256i _mm256_setr_epi8(char, char, char);</code>
<a href="#">_mm256_setr_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_setr_pd(double, double, double, double);</code>
<a href="#">_mm256_setr_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_setr_ps(float, float, float, float, float, float, float, float);</code>
<a href="#">_mm256_setzero_pd</a>	AVX	immintrin.h	<code>__m256d _mm256_setzero_pd(void);</code>
<a href="#">_mm256_setzero_ps</a>	AVX	immintrin.h	<code>__m256 _mm256_setzero_ps(void);</code>
<a href="#">_mm256_setzero_si256</a>	AVX	immintrin.h	<code>__m256i _mm256_setzero_si256(void);</code>
<a href="#">_mm256_shuffle_epi32</a>	AVX2	immintrin.h	<code>__m256i _mm256_shuffle_epi32(__m256i, const int);</code>

内部函数名称	技术	标头	函数原型
_mm256_shuffle_epi8 ↗	AVX2	immintrin.h	<code>__m256i _mm256_shuffle_epi8(__m256i, __m256i);</code>
_mm256_shuffle_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_shuffle_pd(__m256d, __m256d, const int);</code>
_mm256_shuffle_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_shuffle_ps(__m256, __m256, const int);</code>
_mm256_shufflehi_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_shufflehi_epi16(__m256i, const int);</code>
_mm256_shufflelo_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_shufflelo_epi16(__m256i, const int);</code>
_mm256_sign_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sign_epi16(__m256i, __m256i);</code>
_mm256_sign_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sign_epi32(__m256i, __m256i);</code>
_mm256_sign_epi8 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sign_epi8(__m256i, __m256i);</code>
_mm256_sll_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sll_epi16(__m256i, __m128i);</code>
_mm256_sll_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sll_epi32(__m256i, __m128i);</code>
_mm256_sll_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sll_epi64(__m256i, __m128i);</code>
_mm256_slli_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_slli_epi16(__m256i, int);</code>
_mm256_slli_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_slli_epi32(__m256i, int);</code>
_mm256_slli_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_slli_epi64(__m256i, int);</code>
_mm256_slli_si256 ↗	AVX2	immintrin.h	<code>__m256i _mm256_slli_si256(__m256i, int);</code>
_mm256_sllv_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sllv_epi32(__m256i, __m256i);</code>
_mm256_sllv_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sllv_epi64(__m256i, __m256i);</code>
_mm256_sqrt_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_sqrt_pd(__m256d);</code>
_mm256_sqrt_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_sqrt_ps(__m256);</code>
_mm256_sra_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sra_epi16(__m256i, __m128i);</code>
_mm256_sra_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sra_epi32(__m256i, __m128i);</code>
_mm256_srai_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srai_epi16(__m256i, int);</code>
_mm256_srai_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srai_epi32(__m256i, int);</code>
_mm256_srav_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srav_epi32(__m256i, __m256i);</code>
_mm256_srl_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srl_epi16(__m256i, __m128i);</code>
_mm256_srl_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srl_epi32(__m256i, __m128i);</code>
_mm256_srl_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srl_epi64(__m256i, __m128i);</code>
_mm256_srli_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srli_epi16(__m256i, int);</code>
_mm256_srli_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srli_epi32(__m256i, int);</code>
_mm256_srli_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srli_epi64(__m256i, int);</code>
_mm256_srli_si256 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srli_si256(__m256i, int);</code>

内部函数名称	技术	标头	函数原型
_mm256_srlv_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srlv_epi32(__m256i, __m256i);</code>
_mm256_srlv_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_srlv_epi64(__m256i, __m256i);</code>
_mm256_store_pd ↗	AVX	immintrin.h	<code>void _mm256_store_pd(double *, __m256d);</code>
_mm256_store_ps ↗	AVX	immintrin.h	<code>void _mm256_store_ps(float *, __m256);</code>
_mm256_store_si256 ↗	AVX	immintrin.h	<code>void _mm256_store_si256(__m256i *, __m256i);</code>
_mm256_storeu_pd ↗	AVX	immintrin.h	<code>void _mm256_storeu_pd(double *, __m256d);</code>
_mm256_storeu_ps ↗	AVX	immintrin.h	<code>void _mm256_storeu_ps(float *, __m256);</code>
_mm256_storeu_si256 ↗	AVX	immintrin.h	<code>void _mm256_storeu_si256(__m256i *, __m256i);</code>
_mm256_stream_load_si256 ↗	AVX2	immintrin.h	<code>__m256i _mm256_stream_load_si256(__m256i const *);</code>
_mm256_stream_pd ↗	AVX	immintrin.h	<code>void _mm256_stream_pd(double *, __m256d);</code>
_mm256_stream_ps ↗	AVX	immintrin.h	<code>void _mm256_stream_ps(float *, __m256);</code>
_mm256_stream_si256 ↗	AVX	immintrin.h	<code>void _mm256_stream_si256(__m256i *, __m256i);</code>
_mm256_sub_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sub_epi16(__m256i, __m256i);</code>
_mm256_sub_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sub_epi32(__m256i, __m256i);</code>
_mm256_sub_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sub_epi64(__m256i, __m256i);</code>
_mm256_sub_epi8 ↗	AVX2	immintrin.h	<code>__m256i _mm256_sub_epi8(__m256i, __m256i);</code>
_mm256_sub_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_sub_pd(__m256d, __m256d);</code>
_mm256_sub_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_sub_ps(__m256, __m256);</code>
_mm256_subs_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_subs_epi16(__m256i, __m256i);</code>
_mm256_subs_epi8 ↗	AVX2	immintrin.h	<code>__m256i _mm256_subs_epi8(__m256i, __m256i);</code>
_mm256_subs_epu16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_subs_epu16(__m256i, __m256i);</code>
_mm256_subs_epu8 ↗	AVX2	immintrin.h	<code>__m256i _mm256_subs_epu8(__m256i, __m256i);</code>
_mm256_testc_pd ↗	AVX	immintrin.h	<code>int _mm256_testc_pd(__m256d, __m256d);</code>
_mm256_testc_ps ↗	AVX	immintrin.h	<code>int _mm256_testc_ps(__m256, __m256);</code>
_mm256_testc_si256 ↗	AVX	immintrin.h	<code>int _mm256_testc_si256(__m256i, __m256i);</code>
_mm256_testnzc_pd ↗	AVX	immintrin.h	<code>int _mm256_testnzc_pd(__m256d, __m256d);</code>
_mm256_testnzc_ps ↗	AVX	immintrin.h	<code>int _mm256_testnzc_ps(__m256, __m256);</code>
_mm256_testnzc_si256 ↗	AVX	immintrin.h	<code>int _mm256_testnzc_si256(__m256i, __m256i);</code>
_mm256_testz_pd ↗	AVX	immintrin.h	<code>int _mm256_testz_pd(__m256d, __m256d);</code>
_mm256_testz_ps ↗	AVX	immintrin.h	<code>int _mm256_testz_ps(__m256, __m256);</code>
_mm256_testz_si256 ↗	AVX	immintrin.h	<code>int _mm256_testz_si256(__m256i, __m256i);</code>
_mm256_unpackhi_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_unpackhi_epi16(__m256i, __m256i);</code>
_mm256_unpackhi_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_unpackhi_epi32(__m256i, __m256i);</code>

内部函数名称	技术	标头	函数原型
_mm256_unpackhi_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_unpackhi_epi64(__m256i, __m256i);</code>
_mm256_unpackhi_epi8 ↗	AVX2	immintrin.h	<code>__m256i _mm256_unpackhi_epi8(__m256i, __m256i);</code>
_mm256_unpackhi_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_unpackhi_pd(__m256d, __m256d);</code>
_mm256_unpackhi_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_unpackhi_ps(__m256, __m256);</code>
_mm256_unpacklo_epi16 ↗	AVX2	immintrin.h	<code>__m256i _mm256_unpacklo_epi16(__m256i, __m256i);</code>
_mm256_unpacklo_epi32 ↗	AVX2	immintrin.h	<code>__m256i _mm256_unpacklo_epi32(__m256i, __m256i);</code>
_mm256_unpacklo_epi64 ↗	AVX2	immintrin.h	<code>__m256i _mm256_unpacklo_epi64(__m256i, __m256i);</code>
_mm256_unpacklo_epi8 ↗	AVX2	immintrin.h	<code>__m256i _mm256_unpacklo_epi8(__m256i, __m256i);</code>
_mm256_unpacklo_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_unpacklo_pd(__m256d, __m256d);</code>
_mm256_unpacklo_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_unpacklo_ps(__m256, __m256);</code>
_mm256_xor_pd ↗	AVX	immintrin.h	<code>__m256d _mm256_xor_pd(__m256d, __m256d);</code>
_mm256_xor_ps ↗	AVX	immintrin.h	<code>__m256 _mm256_xor_ps(__m256, __m256);</code>
_mm256_xor_si256 ↗	AVX2	immintrin.h	<code>__m256i _mm256_xor_si256(__m256i, __m256i);</code>
_mm256_zeroall ↗	AVX	immintrin.h	<code>void _mm256_zeroall(void);</code>
_mm256_zeroupper ↗	AVX	immintrin.h	<code>void _mm256_zeroupper(void);</code>
_movsb	intrin.h	VOID	<code>_movsb(unsigned char *, unsigned char const *, size_t);</code>
_movsd	intrin.h	VOID	<code>_movsd(unsigned long *, unsigned long const *, size_t);</code>
_movsq	intrin.h	VOID	<code>_movsq(unsigned __int64 *, unsigned __int64 const *, size_t);</code>
_movsw	intrin.h	VOID	<code>_movsw(unsigned short *, unsigned short const *, size_t);</code>
_mul128	intrin.h	__int64	<code>_mul128(__int64, __int64, __int64 *);</code>
_mulh	intrin.h	__int64	<code>_mulh(__int64, __int64);</code>
_mulx_u32 ↗	BMI	immintrin.h	<code>unsigned int _mulx_u32(unsigned int, unsigned int, unsigned int*);</code>
_mulx_u64 ↗	BMI	immintrin.h	<code>unsigned __int64 _mulx_u64(unsigned __int64, unsigned __int64, unsigned __int64*);</code>
_nop	intrin.h	void	<code>_nop(void);</code>
_nvreg_restore_fence	intrin.h	void	<code>_nvreg_restore_fence(void);</code>
_nvreg_save_fence	intrin.h	void	<code>_nvreg_save_fence(void);</code>
_outbyte	intrin.h	void	<code>_outbyte(unsigned short, unsigned char);</code>
_outbytestring	intrin.h	void	<code>_outbytestring(unsigned short, unsigned char *, unsigned long);</code>
_outdword	intrin.h	void	<code>_outdword(unsigned short, unsigned long);</code>

内部函数名称	技术	标头	函数原型
<a href="#">_outdwordstring</a>		intrin.h	void __outdwordstring(unsigned short, unsigned long *, unsigned long);
<a href="#">_outword</a>		intrin.h	void __outword(unsigned short, unsigned short);
<a href="#">_outwordstring</a>		intrin.h	void __outwordstring(unsigned short, unsigned short *, unsigned long);
<a href="#">_pdep_u32</a>	BMI	immintrin.h	unsigned int _pdep_u32(unsigned int, unsigned int);
<a href="#">_pdep_u64</a>	BMI	immintrin.h	unsigned __int64 _pdep_u64(unsigned __int64, unsigned __int64);
<a href="#">_pext_u32</a>	BMI	immintrin.h	unsigned int _pext_u32(unsigned int, unsigned int);
<a href="#">_pext_u64</a>	BMI	immintrin.h	unsigned __int64 _pext_u64(unsigned __int64, unsigned __int64);
<a href="#">_popcnt</a>	POPCNT	intrin.h	unsigned int __popcnt(unsigned int);
<a href="#">_popcnt16</a>	POPCNT	intrin.h	unsigned short __popcnt16(unsigned short);
<a href="#">_popcnt64</a>	POPCNT	intrin.h	unsigned __int64 __popcnt64(unsigned __int64);
<a href="#">_rdrand16_step</a>	RDRAND	immintrin.h	int _rdrand16_step(unsigned short *);
<a href="#">_rdrand32_step</a>	RDRAND	immintrin.h	int _rdrand32_step(unsigned int *);
<a href="#">_rdrand64_step</a>	RDRAND	immintrin.h	int _rdrand64_step(unsigned __int64 *);
<a href="#">_rdseed16_step</a>	RDSEED	immintrin.h	int _rdseed16_step(unsigned short *);
<a href="#">_rdseed32_step</a>	RDSEED	immintrin.h	int _rdseed32_step(unsigned int *);
<a href="#">_rdseed64_step</a>	RDSEED	immintrin.h	int _rdseed64_step(unsigned __int64 *);
<a href="#">_rdtsc</a>		intrin.h	unsigned __int64 __rdtsc(void);
<a href="#">_rdtscp</a>	RDTSCP	intrin.h	unsigned __int64 __rdtscp(unsigned int*);
<a href="#">_ReadBarrier</a>		intrin.h	void _ReadBarrier(void);
<a href="#">_readcr0</a>		intrin.h	unsigned __int64 __readcr0(void);
<a href="#">_readcr2</a>		intrin.h	unsigned __int64 __readcr2(void);
<a href="#">_readcr3</a>		intrin.h	unsigned __int64 __readcr3(void);
<a href="#">_readcr4</a>		intrin.h	unsigned __int64 __readcr4(void);
<a href="#">_readcr8</a>		intrin.h	unsigned __int64 __readcr8(void);
<a href="#">_readdr</a>		intrin.h	unsigned __int64 __readdr(unsigned);
<a href="#">_readeflags</a>		intrin.h	unsigned __int64 __readeflags(void);
<a href="#">_readfsbase_u32</a>	FSGSBASE	immintrin.h	unsigned int _readfsbase_u32(void);
<a href="#">_readfsbase_u64</a>	FSGSBASE	immintrin.h	unsigned __int64 _readfsbase_u64(void);
<a href="#">_readgsbase_u32</a>	FSGSBASE	immintrin.h	unsigned int _readgsbase_u32(void);

内部函数名称	技术	标头	函数原型
_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);
_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);

内部函数名称	技术	标头	函数原型
<code>_sidt</code>		intrin.h	<code>void __sidt(void*);</code>
<code>_slwpcb</code>	LWP	ammintrin.h	<code>void *_slwpcb(void);</code>
<code>_stac</code>	SMAP	intrin.h	<code>void __stac(void);</code>
<code>_storebe_i16</code> ↗	MOVBE	immintrin.h	<code>void __storebe_i16(void *, short); [宏]</code>
<code>_storebe_i32</code> ↗	MOVBE	immintrin.h	<code>void __storebe_i32(void *, int); [宏]</code>
<code>_storebe_i64</code> ↗	MOVBE	immintrin.h	<code>void __storebe_i64(void *, __int64); [宏]</code>
<code>_store_be_u16</code>	MOVBE	immintrin.h	<code>void __store_be_u16(void *, unsigned short); [宏]</code>
<code>_store_be_u32</code>	MOVBE	immintrin.h	<code>void __store_be_u32(void *, unsigned int); [宏]</code>
<code>_store_be_u64</code>	MOVBE	immintrin.h	<code>void __store_be_u64(void *, unsigned __int64); [宏]</code>
<code>_Store_HLERelease</code>	HLE	immintrin.h	<code>void __Store_HLERelease(long volatile *, long);</code>
<code>_Store64_HLERelease</code>	HLE	immintrin.h	<code>void __Store64_HLERelease(__int64 volatile *, __int64);</code>
<code>_StorePointer_HLERelease</code>	HLE	immintrin.h	<code>void __StorePointer_HLERelease(void * volatile *, void *);</code>
<code>_stosb</code>		intrin.h	<code>void __stosb(unsigned char *, unsigned char, size_t);</code>
<code>_stosd</code>		intrin.h	<code>void __stosd(unsigned long *, unsigned long, size_t);</code>
<code>_stosq</code>		intrin.h	<code>void __stosq(unsigned __int64 *, unsigned __int64, size_t);</code>
<code>_stosw</code>		intrin.h	<code>void __stosw(unsigned short *, unsigned short, size_t);</code>
<code>_subborrow_u16</code>		intrin.h	<code>unsigned char __subborrow_u16(unsigned char, unsigned short, unsigned short, unsigned short *);</code>
<code>_subborrow_u32</code> ↗		intrin.h	<code>unsigned char __subborrow_u32(unsigned char, unsigned int, unsigned int, unsigned int *);</code>
<code>_subborrow_u64</code> ↗		intrin.h	<code>unsigned char __subborrow_u64(unsigned char, unsigned __int64, unsigned __int64, unsigned __int64 *);</code>
<code>_subborrow_u8</code>		intrin.h	<code>unsigned char __subborrow_u8(unsigned char, unsigned char, unsigned char, unsigned char *);</code>
<code>_svm_clgi</code>		intrin.h	<code>void __svm_clgi(void);</code>
<code>_svm_invlpga</code>		intrin.h	<code>void __svm_invlpga(void*, int);</code>
<code>_svm_skinit</code>		intrin.h	<code>void __svm_skinit(int);</code>
<code>_svm_stgi</code>		intrin.h	<code>void __svm_stgi(void);</code>
<code>_svm_vmlload</code>		intrin.h	<code>void __svm_vmlload(size_t);</code>
<code>_svm_vmrunt</code>		intrin.h	<code>void __svm_vmrunt(size_t);</code>
<code>_svm_vmsave</code>		intrin.h	<code>void __svm_vmsave(size_t);</code>

内部函数名称	技术	标头	函数原型
_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);
_umul128	intrin.h		unsigned __int64 _umul128(unsigned __int64, 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);

内部函数名称	技术	标头	函数原型
<a href="#">_writeeflags</a>		intrin.h	void _writeeflags(unsigned __int64);
<a href="#">_writefsbase_u32</a>	FSGSBASE	immintrin.h	void _writefsbase_u32(unsigned int);
<a href="#">_writefsbase_u64</a>	FSGSBASE	immintrin.h	void _writefsbase_u64(unsigned __int64);
<a href="#">_writegsbase_u32</a>	FSGSBASE	immintrin.h	void _writegsbase_u32(unsigned int);
<a href="#">_writegsbase_u64</a>	FSGSBASE	immintrin.h	void _writegsbase_u64(unsigned __int64);
<a href="#">_writegsbyte</a>		intrin.h	void _writegsbyte(unsigned long, unsigned char);
<a href="#">_writegsdword</a>		intrin.h	void _writegsdword(unsigned long, unsigned long);
<a href="#">_writegsqword</a>		intrin.h	void _writegsqword(unsigned long, unsigned __int64);
<a href="#">_writegsword</a>		intrin.h	void _writegsword(unsigned long, unsigned short);
<a href="#">_writemsr</a>		intrin.h	void _writemsr(unsigned long, unsigned __int64);
<a href="#">_xabort</a>	RTM	immintrin.h	void _xabort(unsigned int);
<a href="#">_xbegin</a>	RTM	immintrin.h	unsigned _xbegin(void);
<a href="#">_xend</a>	RTM	immintrin.h	void _xend(void);
<a href="#">_xgetbv</a>	XSAVE	immintrin.h	unsigned __int64 _xgetbv(unsigned int);
<a href="#">_xrstor</a>	XSAVE	immintrin.h	void _xrstor(void const*, unsigned __int64);
<a href="#">_xrstor64</a>	XSAVE	immintrin.h	void _xrstor64(void const*, unsigned __int64);
<a href="#">_xsave</a>	XSAVE	immintrin.h	void _xsave(void*, unsigned __int64);
<a href="#">_xsave64</a>	XSAVE	immintrin.h	void _xsave64(void*, unsigned __int64);
<a href="#">_xsaveopt</a>	XSAVEOPT	immintrin.h	void _xsaveopt(void*, unsigned __int64);
<a href="#">_xsaveopt64</a>	XSAVEOPT	immintrin.h	void _xsaveopt64(void*, unsigned __int64);
<a href="#">_xsetbv</a>	XSAVE	immintrin.h	void _xsetbv(unsigned int, unsigned __int64);
<a href="#">_xtest</a>	XTEST	immintrin.h	unsigned char _xtest(void);

## 另请参阅

[编译器内部函数](#)

[ARM 内部函数](#)

[ARM64 内部函数](#)

[x86 内部函数](#)

# 在所有体系结构上都可用的内部函数

项目 • 2023/06/16

Microsoft C/C++ 编译器和通用 C 运行时库 (UCRT) 使某些内部函数可用于所有体系结构。

## 编译器内部函数

以下内部函数可用于 x86、AMD64、ARM 和 ARM64 架构：

Intrinsic	标头
<a href="#">_AddressOfReturnAddress</a>	intrin.h
<a href="#">_BitScanForward</a>	intrin.h
<a href="#">_BitScanReverse</a>	intrin.h
<a href="#">_bittest</a>	intrin.h
<a href="#">_bittestandcomplement</a>	intrin.h
<a href="#">_bittestandreset</a>	intrin.h
<a href="#">_bittestandset</a>	intrin.h
<a href="#">__code_seg</a>	intrin.h
<a href="#">_debugbreak</a>	intrin.h
<a href="#">_disable</a>	intrin.h
<a href="#">_enable</a>	intrin.h
<a href="#">_fastfail</a>	intrin.h
<a href="#">_InterlockedAnd</a>	intrin.h
<a href="#">_InterlockedAnd16</a>	intrin.h
<a href="#">_InterlockedAnd8</a>	intrin.h
<a href="#">_interlockedbittestandreset</a>	intrin.h
<a href="#">_interlockedbittestandset</a>	intrin.h
<a href="#">_InterlockedCompareExchange</a>	intrin.h
<a href="#">_InterlockedCompareExchange16</a>	intrin.h

Intrinsic	标头
_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
_rotl16	intrin.h
_rotl8	intrin.h
_rotr16	intrin.h

Intrinsic	标头
<a href="#">_rotr8</a>	intrin.h
<a href="#">_WriteBarrier</a>	intrin.h

## UCRT 内部函数

以下 UCRT 函数的所有体系结构均具有内部函数形式：

Intrinsic	标头
<a href="#">abs</a>	stdlib.h
<a href="#">_abs64</a>	stdlib.h
<a href="#">acos</a>	math.h
<a href="#">acosf</a>	math.h
<a href="#">acosl</a>	math.h
<a href="#">_alloca</a>	malloc.h
<a href="#">asin</a>	math.h
<a href="#">asinf</a>	math.h
<a href="#">asinl</a>	math.h
<a href="#">atan</a>	math.h
<a href="#">atan2</a>	math.h
<a href="#">atan2f</a>	math.h
<a href="#">atan2l</a>	math.h
<a href="#">atanf</a>	math.h
<a href="#">atanl</a>	math.h
<a href="#">_byteswap_uint64</a>	stdlib.h
<a href="#">_byteswap_ulong</a>	stdlib.h
<a href="#">_byteswap_ushort</a>	stdlib.h
<a href="#">ceil</a>	math.h
<a href="#">ceilf</a>	math.h

Intrinsic	标头
ceil	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
log	math.h
log10	math.h
log10f	math.h
log10l	math.h
logf	math.h
logl	math.h

Intrinsic	标头
_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
sqrt	math.h
sqrtf	math.h
sqrtl	math.h
strcat	string.h
strcmp	string.h
strcpy	string.h
strlen	string.h
_strset	string.h

Intrinsic	标头
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
wcscopy	string.h
wcslen	string.h
_wcsset	string.h

在 Visual Studio 2022 版本 17.2 及更高版本中，这些函数在 x64 和 ARM64 平台上具有内在形式：

Intrinsic	标头
log2	math.h
log2f	math.h

## 另请参阅

[ARM 内部函数](#)

[ARM64 内部函数](#)

[x86 内部函数列表](#)

[x64 \(amd64\) 内部函数列表](#)

# 按字母顺序排序的内部函数列表

项目 • 2024/11/12

以下各部分介绍了部分或所有体系架构中可用的特定于 Microsoft 的内部函数。处理器制造商在头文件或其网站上记录其他受支持的内部函数。有关详细信息以及指向制造商文档的链接，请参阅以下文章：[ARM 内部函数](#)、[ARM64 内部函数](#)、[x86 内部函数](#)和[x64 内部函数](#)。此处未记录作为内部函数实现的 C 运行时库 (CRT) 函数。CRT 内部函数记录在[C 运行时库参考](#)中。

[\\_addfsbyte](#)、[\\_addfsword](#)、[\\_addfsdword](#)

[\\_addgsbyte](#)、[\\_addgsword](#)、[\\_addgsdword](#)、[\\_addgssqword](#)

[\\_AddressOfReturnAddress](#)

[\\_assume](#)

[\\_BitScanForward](#), [\\_BitScanForward64](#)

[\\_BitScanReverse](#), [\\_BitScanReverse64](#)

[\\_bittest](#), [\\_bittest64](#)

[\\_bittestandcomplement](#), [\\_bittestandcomplement64](#)

[\\_bittestandreset](#), [\\_bittestandreset64](#)

[\\_bittestandset](#), [\\_bittestandset64](#)

[\\_check\\_isa\\_support](#), [\\_check\\_arch\\_support](#)

[\\_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 内部函数  
\_InterlockedAddLargeStatistic  
\_InterlockedAnd 内部函数  
\_interlockedbittestandreset 内部函数  
\_interlockedbittestandset 内部函数  
\_InterlockedCompareExchange 内部函数  
\_InterlockedCompareExchange128  
\_InterlockedCompareExchangePointer 内部函数  
\_InterlockedDecrement 内部函数  
\_InterlockedExchange 内部函数  
\_InterlockedExchangeAdd 内部函数  
\_InterlockedExchangePointer 内部函数

[\\_InterlockedIncrement 内部函数](#)

[\\_InterlockedOr 内部函数](#)

[\\_InterlockedXor 内部函数](#)

[\\_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\_vmrund

\_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](#)

## 另请参阅

[编译器内部函数](#)

---

## 反馈

此页面是否有帮助？

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获取帮助

# **\_\_addfsbyte, \_\_addfword, \_\_addfsdword**

项目 • 2023/10/12

**Microsoft 专用**

将值添加到由相对于 **FS** 段开头的偏移量指定的内存位置。

## 语法

C

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

## 参数

*Offset*

[in] 与 **FS** 的起始位置间的偏移量。

*数据*

[in] 添加到内存位置的值。

## 要求

Intrinsic	体系结构
<b>__addfsbyte</b>	x86
<b>__addfword</b>	x86
<b>__addfsdword</b>	x86

头文件<intrin.h>

## 备注

这些例程只能用作内部函数。

结束 Microsoft 专用

## 另请参阅

[\\_incfsbyte, \\_incfsword, \\_incfsdword](#)

[\\_readfsbyte, \\_readfsdword, \\_readfsqword, \\_readfsword](#)

[\\_writefsbyte, \\_writefsdword, \\_writefsqword, \\_writefsword](#)

[编译器内部函数](#)

# **\_\_addgsbyte, \_\_addgsword, \_\_addgsdword, \_\_addgsqword**

项目 • 2023/10/12

**Microsoft 专用**

将值添加到由相对于 **GS** 段开头的偏移量指定的内存位置。

## 语法

C

```
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
);
```

## 参数

*Offset*

[in] 与 **GS** 的起始位置间的偏移量。

**数据**

[in] 添加到内存位置的值。

## 要求

Intrinsic	体系结构
<code>__addgsbyte</code>	X64

Intrinsic	体系结构
__addgsword	X64
__addgsdword	X64
__addgsqlword	X64

头文件<intrin.h>

## 备注

这些例程只能用作内部函数。

结束 Microsoft 专用

## 另请参阅

[\\_incgsbyte, \\_incgsword, \\_incgsdword, \\_incgsqlword](#)  
[\\_readgsbyte, \\_readgsdword, \\_readgsqlword, \\_readgsword](#)  
[\\_writegsbyte, \\_writegsdword, \\_writegsqlword, \\_writegsword](#)  
编译器内部函数

# \_AddressOfReturnAddress

项目 • 2023/10/12

## Microsoft 专用

提供保存当前函数返回地址的内存位置的地址。 此地址不能用于访问其他内存位置，(例如函数的参数)。

## 语法

C

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

## 要求

Intrinsic	体系结构
_AddressOfReturnAddress	x86、x64、ARM、ARM64

头文件<intrin.h>

## 备注

在使用 [/clr](#) 编译的程序中使用 `_AddressOfReturnAddress` 时，包含 `_AddressOfReturnAddress` 调用的函数将编译为本机函数。当按托管方式编译的函数调用包含 `_AddressOfReturnAddress` 的函数时，`_AddressOfReturnAddress` 可能无法按预期方式运行。

此例程仅可用作内部函数。

## 示例

C++

```
// 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();  
}
```

#### Output

```
0012FF78  
00401058  
00401058
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[关键字](#)

# `_assume`

项目 • 2023/06/16

Microsoft 专用

传递优化程序提示。

## 语法

```
C  
_assume(  
    expression  
)
```

## 参数

`expression`

对于可访问代码，假设评估为 `true` 的任何表达式。使用 `0` 向优化程序指示无法访问的代码。

## 备注

优化程序假设在关键字出现的地方由 `expression` 表示的条件为 `true` 并且直到修改 `expression`（例如，通过给变量赋值）前一直为 `true`。选择性使用通过 `_assume` 传递给优化程序的提示可以完善优化。

如果将 `_assume` 语句写成矛盾（始终评估为 `false` 的表达式），则其始终会被视为 `_assume(0)`。如果代码表现得不如预期，确保你定义的 `expression` 为有效和 `true`，如前所述。`_assume(0)` 语句是一个特例。使用 `_assume(0)` 来表示无法到达的代码路径。

### ⚠ 警告

程序的可达路径不得包含无效的 `_assume` 语句。如果编译器可以到达无效的 `_assume` 语句，程序可能会导致不可预测的潜在危险行为。

为了与以前的版本兼容，除非指定了编译器选项 [/Za \(禁用语言扩展\)](#)，否则 `_assume` 是 `_assume` 的同义词。

`_assume` 不是真正的内部函数。不必将其声明为函数，也不能在 `#pragma intrinsic` 指令中使用它。尽管没有生成任何代码，但还是会对由优化程序生成的代码产生影响。

只有在断言不可恢复时，才能在 `ASSERT` 中使用 `_assume`。请勿在其中有后续错误恢复代码的断言中使用 `_assume`，因为编译器可能优化掉错误处理的代码。

## 要求

Intrinsic	体系结构
<code>_assume</code>	x86、ARM、x64、ARM64、ARM64EC

## 示例

以下示例显示如何使用 `_assume(0)` 来指示无法获得 `switch` 语句的 `default` 大小写。

这是 `_assume(0)` 最典型的用法。此处，程序员了解 `p` 唯一可能的输入值是 1 或 2。如果传入 `p` 其他值，程序将变为无效并导致不可预测的行为。

C++

```
// 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.
    }
}
```

由于 `__assume(0)` 语句的结果，编译器不会生成代码来测试 `p` 是否具有不会在 `case` 语句中显示的值。

如果不确定表达式在运行时始终为 `true`，你可以使用 `assert` 函数保护该代码。此宏定义使用检查包装 `__assume` 语句：

C

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

要使 `default` 大小写优化生效，`__assume(0)` 语句必须是 `default` 大小写正文中的第一条语句。可惜的是，`ASSUME` 宏中的 `assert` 阻止编译器执行此优化。作为替代方案，你可以使用单独的宏，如下所示：

C

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

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[关键字](#)

# \_BitScanForward、\_BitScanForward64

项目 • 2023/10/17

**Microsoft 专用**

从设置位 (1) 的最低有效位 (LSB) 到最高有效位 (MSB) 搜索掩码数据。

## 语法

C

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

## 参数

*Index*

[out] 加载已找到的第一个设置位 (1) 的数位位置。

*掩码*

[in] 要搜索的 32 位或 64 位值。

## 返回值

如果掩码为零，则为 0；否则为非零值。

## 备注

如果找到集位，则第一个集位的位位置将写入第一个参数中指定的地址，并且函数返回 1。如果未找到位，该函数将返回 0，并且写入到第一个参数中的地址的值未定义。

## 要求

Intrinsic	体系结构
_BitScanForward	x86、ARM、x64、ARM64
_BitScanForward64	ARM64、x64

头文件<intrin.h>

## 示例

C++

```
// 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;
    }
}
```

Input

12

Output

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

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# \_BitScanReverse、\_BitScanReverse64

项目 · 2024/08/04

Microsoft 专用

从设置位 (1) 的最高有效位 (MSB) 到最低有效位 (LSB) 搜索掩码数据。

## 语法

```
C

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

## 参数

*Index*

[out] 加载已找到的第一个设置位 (1) 的数位位置。 其他情况下则不定义。

*掩码*

[in] 要搜索的 32 位或 64 位值。

## 返回值

如果 *Mask* 中设置了任何位，则为非零，如果未设置位，则为 0。

## 要求

[ ] 展开表

Intrinsic	体系结构	头文件
_BitScanReverse	x86、ARM、x64、ARM64	<intrin.h>
_BitScanReverse64	ARM64、x64	<intrin.h>

## 示例

C++

```
// 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;
    }
}
```

Input

12

Output

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

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_bittest、\_bittest64

项目 · 2024/08/04

Microsoft 专用

生成 `bt` 指令，从而检查地址 `b` 的位置 `a` 中的位，并返回该位的值。

## 语法

```
C

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

## 参数

`a`

[in] 指向要检查的内存的指针。

`b`

[in] 要测试的位位置。

## 返回值

指定位置的位。

## 要求

展开表

Intrinsic	体系结构	头文件
<code>_bittest</code>	x86、ARM、x64、ARM64	<intrin.h>
<code>_bittest64</code>	ARM64、x64	<intrin.h>

# 注解

此例程仅可用作内部函数。

## 示例

C++

```
// 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");
    }
}
```

Output

```
Number: 78002
Binary representation:
00000000000000010011000010110010
```

结束 Microsoft 专用

另请参阅

# 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | [在 Microsoft Q&A 获得帮助](#)

# **\_bittestandcomplement、 \_bittestandcomplement64**

项目 • 2023/06/16

**Microsoft 专用**

生成可以检查地址 `b` 的位 `a` 的指令，返回其当前值，然后将位设置为该值的补充值。

## 语法

C

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

## 参数

*a*

[in, out] 指向要检查的内存的指针。

*b*

[in] 要测试的位位置。

## 返回值

指定位置的位。

## 要求

Intrinsic	体系结构
<code>_bittestandcomplement</code>	x86、ARM、x64、ARM64
<code>_bittestandcomplement64</code>	x64、ARM64

## 注解

此例程仅可用作内部函数。

## 示例

C++

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

#pragma intrinsic(_bittestandcomplement)
#ifndef _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);
#ifndef _M_AMD64
    printf("Testing bit 0\n");
    result = _bittestandcomplement64(&i64, 0);
    printf("Value changed to %I64d, Result: %d\n", i64, result);
#endif
}
```

Output

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

# 另请参阅

[编译器内部函数](#)

# **\_bittestandreset、\_bittestandreset64**

项目 · 2024/08/04

**Microsoft 专用**

生成一个指令，该指令可以检查地址 `a` 的位 `b`，返回其当前值，然后将位设置为 0。

## 语法

```
C

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

## 参数

`a`

[in, out] 指向要检查的内存的指针。

`b`

[in] 要测试的位位置。

## 返回值

指定位置的位。

## 要求

[展开表](#)

Intrinsic	体系结构
<code>_bittestandreset</code>	x86、ARM、x64、ARM64
<code>_bittestandreset64</code>	x64、ARM64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

## 示例

C++

```
// 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");
}
```

Output

```
The number was negative.
```

结束 Microsoft 专用

另请参阅

# 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | [在 Microsoft Q&A 获得帮助](#)

# \_bittestandset、\_bittestandset64

项目 · 2024/08/04

Microsoft 专用

生成一个指令，该指令可以检查地址 `a` 的位 `b`，返回其当前值，然后将位设置为 1。

## 语法

```
C

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

## 参数

`a`

[in, out] 指向要检查的内存的指针。

`b`

[in] 要测试的位位置。

## 返回值

指定位置的位。

## 要求

展开表

Intrinsic	体系结构
<code>_bittestandset</code>	x86、ARM、x64、ARM64
<code>_bittestandset64</code>	x64、ARM64

## 注解

此例程仅可用作内部函数。

## 示例

C++

```
// 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>

#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);
}

```

### Output

```

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

```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

## 反馈

此页面是否有帮助？

是

否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# `__check_isa_support`, `__check_arch_support`

项目 • 2024/11/12

## Microsoft 专用

`__check_isa_support` - 检测处理器是否在运行时支持指定的 ISA 功能和 AVX10 版本。  
`__check_arch_support` - 检测拱门标志（请参阅 `(x86 /arch)`、`/arch (x64)`）是否在编译时支持指定的 ISA 功能和 AVX10 版本。

## 语法

C

```
_Bool __check_isa_support(
    unsigned feature,
    unsigned avx10_version
);

_Bool __check_arch_support(
    unsigned feature,
    unsigned avx10_version
);
```

C++

```
bool __check_isa_support(
    unsigned feature,
    unsigned avx10_version
);

bool __check_arch_support(
    unsigned feature,
    unsigned avx10_version
);
```

## 参数

*feature*

[in]要检查的 ISA 功能。

*avx10\_version*

[in]要检查的 AVX10 版本。如果不需要 AVX10 版本检查，则为 0。

# 返回值

`_check_isa_support` `true` 如果处理器支持 `feature` 并在 `avx10_version` 运行时返回，则返回; 否则返回 `false`。 `_check_arch_support` `true` 如果 `/arch` 标志支持 `feature` 并在 `avx10_version` 编译时返回，`false` 否则返回。

## 要求

 展开表

Intrinsic	体系结构
<code>_check_isa_support</code>	x86、x64
<code>_check_arch_support</code>	x86、x64

头文件 `<immintrin.h>`

## 注解

内部 `_check_isa_support` 函数为内部函数提供了更快的替代 `_cpuid` 方法，用于动态检查最常用的 CPU 功能。 内部 `_check_arch_support` 函数提供基于 ISA 扩展的编译时代码选择的替代方法 [predefined macros](#)。

这些内部函数中可以使用以下特征值。 这些值是在 `.ISA_AVAILABILITY.H` 中定义的。

 展开表

特征值名称	说明
<code>_IA_SUPPORT_VECTOR128</code>	长度高达 128 位的向量指令。 此功能已启用 SSE2 或更高版本扩展
<code>_IA_SUPPORT_VECTOR256</code>	长度高达 256 位的向量指令。 为 AVX2 或更高版本扩展启用此功能
<code>_IA_SUPPORT_VECTOR512</code>	长度高达 512 位的向量指令。 为 AVX-512 或更高版本扩展启用此功能
<code>_IA_SUPPORT_AVX10</code>	AVX10 支持。 为 AVX10.1 或更高版本扩展启用此功能
<code>_IA_SUPPORT_SSE42</code>	SSE4.2 支持
<code>_IA_SUPPORT_SV128X</code>	128 位标量标量 AVX-512 指令。 可用于指示某些有用的 AVX-512 指令 (如转换) 可用于标量代码
<code>_IA_SUPPORT_AVX10_2</code>	AVX10.2 支持

特征值名称	说明
<code>__IA_SUPPORT_APX</code>	APX 支持
<code>__IA_SUPPORT_FP16</code>	半精度浮点指令支持

可以使用 OR (|) 运算符组合多个特征值。

`__check_arch_support` 内部函数始终可以在编译时进行评估，因此在优化代码中使用内部函数不会添加额外的执行指令。Visual Studio 2022 版本 17.10 中添加了对这些内部函数的支持。

## 示例

此示例使用 256 位 AVX-512 指令将双精度值转换为 64 位有符号整数值。转换矢量代码未处理的任何源值的结尾循环也用于执行向量代码。在运行时支持之前检查编译时支持，以便尽可能避免运行时检查。

C++

```
// Compile this test with: /EHsc /O2
#include <iostream>
#include <vector>
#include <immintrin.h>
#include <isa_availability.h>
using namespace std;

#define CHECK_INSTRUCTION_SUPPORT(a,v) \
    (__check_arch_support((a),(v)) || __check_isa_support((a),(v)))

int main()
{
    vector<double> input = {0.3, 1.4, 2.5, 3.6, 4.7, 5.8, 6.9, 8.0, 9.1,
                           11.14};
    vector<__int64> output(10, 0);
    int i = 0;

    if (CHECK_INSTRUCTION_SUPPORT(__IA_SUPPORT_SV128X |
__IA_SUPPORT_VECTOR256, 0))
    {
        for ( ; i < input.size() - 4; i += 4)
        {
            __m256i values = _mm256_cvttpd_epi64(_mm256_load_pd(&input[i]));
            _mm256_storeu_epi64((void*)&output[i], values);
        }
        for ( ; i < input.size(); i++)
        {
            output[i] = input[i];
        }
    }
}
```

```
for (i = 0; i < output.size(); i++) {
    cout << "output[" << i << "] = " << output[i] << endl;
}
}
```

## Output

```
output[0] = 0
output[1] = 1
output[2] = 2
output[3] = 3
output[4] = 4
output[5] = 5
output[6] = 6
output[7] = 8
output[8] = 9
output[9] = 11
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

## 反馈

此页面是否有帮助?

是

否

[提供产品反馈](#) | 在 Microsoft Q&A 获取帮助

# **\_\_cpuid、 \_\_cpuidex**

项目 • 2023/06/16

## Microsoft 专用

生成可在 x86 和 x64 上使用的 `cpuid` 指令。本指令可查询处理器，以获取有关支持的功能和 CPU 类型的信息。

## 语法

C

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

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

## 参数

### *cpuInfo*

[out] 四个整数的数组，包含在 EAX、EBX、ECX 和 EDX 中返回的有关 CPU 支持的功能的信息。

### *function\_id*

[in] 在 EAX 中传递的指定要检索的信息的代码。

### *subfunction\_id*

[in] 在 ECX 中传递的指定要检索的信息的附加代码。

## 要求

Intrinsic	体系结构
<code>__cpuid</code>	x86、x64
<code>__cpuidex</code>	x86、x64

## 注解

此内部函数将存储由 `cpulInfo` 中的 `cpuid` 指令返回的支持功能和 CPU 信息，使用 EAX、EBX、ECX 和 EDX 寄存器值（按照这个顺序）填充的四个 32 位整数的数组。返回的信息具有不同含义，具体取决于作为 `function_id` 参数传递的值。随 `function_id` 的多个值返回的信息与处理器有关。

`_cpuid` 内部函数将在调用 `cpuid` 指令前清除 ECX 寄存器。`_cpuidex` 内部函数可在 ECX 寄存器生成 `cpuid` 指令之前，将其值设置为 `subfunction_id`。这使你能够收集有关该处理器的其他信息。

有关 Intel 处理器上要使用的特定参数和这些内部函数所返回的值的详细信息，请参阅“Intel 64 和 IA-32 体系结构软件开发人员手册第 2 卷：指令设置参考”[\[1\]](#) 和“Intel 体系结构指令集扩展编程参考”[\[2\]](#) 中的 `cpuid` 指令。对于在 EAX 和 ECX 中传递的 `function_id` 和 `subfunction_id` 参数，Intel 文档将使用术语“leaf”和“subleaf”。

有关 AMD 处理器上要使用的特定参数和这些内部函数所返回的值的详细信息，请参阅“AMD64 体系结构编程人员手册第 3 卷：通用和系统指令”中的 `cpuid` 指令和“修订指南”中的特定处理器系列。有关这些文档和其他信息的链接，请参阅 AMD [开发人员指南、手册 & ISA 文档](#)[\[3\]](#) 页。对于在 EAX 和 ECX 中传递的 `function_id` 和 `subfunction_id` 参数，AMD 文档将使用术语“function number”和“subfunction number”。

当 `function_id` 参数为 0，`cpulInfo[0]` 返回处理器支持的可用的最高非扩展 `function_id` 值。处理器制造商在 `cpulInfo[1]`、`cpulInfo[2]` 和 `cpulInfo[3]` 中进行编码。

支持特定的指令集扩展和 CPU 功能编码在更高 `function_id` 值返回的 `cpulInfo` 结果中。有关详细信息，请参阅上述链接的手册和以下示例代码。

某些处理器支持扩展函数 CPUID 信息。如果支持此操作，则 0x80000000 中的 `function_id` 值可用于返回信息。若要确定允许的有意义的最大值，请将 `function_id` 设置为 0x80000000。支持扩展功能的 `function_id` 的最大值将被写入 `cpulInfo[0]`。

## 示例

此示例显示了通过 `_cpuid` 和 `_cpuidex` 内部函数提供的一些信息。此应用列出了受当前处理器支持的指令集扩展。此输出显示了特定处理器的可能结果。

C++

```
// 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());
}

}

```

Output

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

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# `__debugbreak`

项目 • 2023/06/16

Microsoft 专用

将在代码中引起断点，并在其中提示用户运行调试程序。

## 语法

C

```
void __debugbreak();
```

## 要求

Intrinsic	体系结构	标头
<code>__debugbreak</code>	x86、x64、ARM、ARM64	<intrin.h>

## 注解

`__debugbreak` 编译器内部函数类似于 [DebugBreak](#)，是导致断点的可移植的 Win32 方式。

### ① 备注

使用 /clr 编译时，包含 `__debugbreak` 的函数将编译为 MSIL。`asm int 3` 可将函数编译为本机函数。有关详细信息，请参阅 [\\_asm](#)。

例如：

C

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

类似于：

C

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

在 x86 计算机上。

在 ARM64 上，`__debugbreak` 内部函数编译为指令 `brk #0xF000`。

此例程仅可用作内部函数。

**结束 Microsoft 专用**

## 另请参阅

[编译器内部函数](#)

[关键字](#)

# \_disable

项目 • 2023/10/12

Microsoft 专用

禁用中断。

## 语法

C

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

## 要求

Intrinsic	体系结构
_disable	x86、ARM、x64、ARM64

头文件<intrin.h>

## 备注

`_disable` 指示处理器去清除中断标记。在 x86 系统上，此函数会生成“清除中断标记”(`cli`)指令。

此函数只有在内核模式下才可用。如果在用户模式下使用，运行时会出现特权指令异常。

在 ARM 和 ARM64 平台上，此例程只能用作内部函数。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# \_div128

项目 • 2023/06/16

`_div128` 内部函数将 128 位整数除以 64 位整数。 返回值包含商，内部函数通过指针参数返回余数。 `_div128` 是 Microsoft 特定的。

## 语法

C

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

## 参数

*highDividend*

[in] 被除数的高 64 位。

*lowDividend*

[in] 被除数的低 64 位。

*divisor*

[in] 要除以的 64 位整数。

*remainder*

[out] 余数的 64 位整数位。

## 返回值

商的 64 位。

## 注解

传递 `highDividend` 中 128 位被除数的高 64 位，以及 `lowDividend` 中的低 64 位。 内部函数将此值除以除数。 它将余数存储在余数所指向的 64 位整数中，并返回 64 位的商。

`_div128` 内部函数从 Visual Studio 2019 RTM 开始可用。

# 要求

Intrinsic	体系结构	标头
_div128	X64	<immintrin.h>

## 另请参阅

[\\_udiv128](#)

[编译器内部函数](#)

# \_div64

项目 • 2023/06/16

`_div64` 内部函数将 64 位整数除以 32 位整数。 返回值包含商，内部函数通过指针参数返回余数。 `_div64` 是 Microsoft 特定的。

## 语法

C

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

## 参数

*dividend*

[in] 要除的 64 位整数。

*divisor*

[in] 要除以的 32 位整数。

*remainder*

[out] 余数的 32 位整数位。

## 返回值

商的 32 位。

## 注解

`_div64` 内部函数将被除数除以除数。 它将余数存储在余数所指向的 32 位整数中，并返回 32 位的商。

`_div64` 内部函数从 Visual Studio 2019 RTM 开始可用。

## 要求

Intrinsic	体系结构	标头
_div64	x86、x64	<immintrin.h>

## 另请参阅

[\\_udiv64](#)

[编译器内部函数](#)

# **\_\_emul, \_\_emulu**

项目 • 2023/10/12

**Microsoft 专用**

执行溢出 32 位整数可容纳范围的乘法运算。

## 语法

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

## 参数

*a*

[in] 乘法的第一个整数操作数。

*b*

[in] 乘法的第二个整数操作数。

## 返回值

乘法的结果。

## 要求

Intrinsic	体系结构
<code>__emul</code>	x86、x64
<code>__emulu</code>	x86、x64

头文件<intrin.h>

# 备注

`__emul` 将两个 32 位带符号值相乘并将结果以 64 位带符号整数值形式返回。

`__emulu` 将两个 32 位无符号整数值相乘并将结果以 64 位无符号整数值形式返回。

## 示例

C++

```
// 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 urestult = __emulu(ua, ub);

    cout << ua << " * " << ub << " = " << urestult << endl;
}
```

## 输出

Output

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

# 另请参阅

[编译器内部函数](#)

# \_enable

项目 • 2023/06/16

**Microsoft 专用**

启用中断。

## 语法

C

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

## 要求

Intrinsic	体系结构
_enable	x86、ARM、x64、ARM64

头文件<intrin.h>

## 注解

`_enable` 指示处理器设置中断标志。在 x86 系统上，此函数会生成“设置中断标志”(`sti`) 指令。

此函数只有在内核模式下才可用。如果在用户模式下使用，会引发特权指令异常。

**结束 Microsoft 专用**

## 另请参阅

[编译器内部函数](#)

# \_\_fastfail

项目 • 2023/10/12

Microsoft 专用

立即终止开销最少的调用过程。

## 语法

C

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

## 参数

*code*

[in] winnt.h 或 wdm.h 中的 `FAST_FAIL_<description>` 符号常量，指示进程终止的原因。

## 返回值

`__fastcall` 内部函数不会返回。

## 注解

`__fastcall` 内部函数为快速失败请求提供了一种机制 - 一种可能已损坏的进程请求立即终止进程的方法。无法使用常规异常处理设施处理可能已破坏程序状态和堆栈至无法恢复的严重故障。使用 `__fastcall` 终止开销最少的进程。

在内部，可以使用几个特定于体系结构的机制实现 `__fastcall`：

体系结构	说明	代码参数的位置
x86	int 0x29	ecx
x64	int 0x29	rcx
ARM	操作码 0xDEFB	r0
ARM64	操作码 0xF003	x0

快速失败请求是独立的请求，通常只需执行两个指令。执行快速失败请求后，内核就会采取相应的行动。在用户模式代码中，引发速快速失败事件时，除指令指针本身外不存在任何内存依赖项。即使在内存严重损坏的情况下，也能最大限度地提高其可靠性。

`code` 参数是 winnt.h 或 wdm.h 中的 `FAST_FAIL_<description>` 符号常量之一，描述了故障条件的类型。它以环境特定的方式合并到故障报告中。

用户模式快速失败请求显示为第二次机会不可持续异常，异常代码为 0xC0000409，并且具有至少一个异常参数。第一个异常参数为 `code` 值。此异常代码向 Windows 错误报告 (WER) 和调试基础结构表明进程已损坏，并且应采取最少的进程内操作来响应故障。内核模式快速失败请求可以通过使用专用检错代码 `KERNEL_SECURITY_CHECK_FAILURE` (0x139) 实现。在这两种情况下，都没有调用异常处理程序，因为程序预期处于损坏状态。如果存在调试程序，就有机会在终止进程之前检查程序的状态。

Windows 8 开始支持本机快速失败机制。原生不支持快速失败指令的 Windows 操作系统通常会将快速失败请求视为访问冲突，或视为 `UNEXPECTED_KERNEL_MODE_TRAP` bug 检查。在这些情况下，仍然会终止程序，但并不一定会快速终止。

`_fastfail` 只能用作内部函数。

## 要求

Intrinsic	体系结构
<code>_fastfail</code>	x86、x64、ARM、ARM64

头文件<intrin.h>

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# `__faststorefence`

项目 • 2023/06/16

## Microsoft 专用

保证每个以前的内存引用（包括加载和存储内存引用）在任何后续的内存引用之前全局可见。

## 语法

C

```
void __faststorefence();
```

## 要求

Intrinsic	体系结构
<code>__faststorefence</code>	X64

头文件<intrin.h>

## 注解

生成完整的内存屏障指令序列，以保证在此内部函数之前发出的加载和存储操作在执行继续之前全局可见。效果相当于所有 x64 平台上的 `_mm_mfence` 内部函数，但是速度更快。

在 AMD64 平台上，此例程会生成一个指令，该指令是比 `sfence` 指令更快的存储隔离。对于时间关键型代码，仅在 AMD64 平台上使用此内部函数而不是 `_mm_sfence`。在 Intel x64 平台上，`_mm_sfence` 指令速度更快。

此例程仅可用作内部函数。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# 快速浮点转换函数

项目 · 2023/06/16

Microsoft 专用

浮点类型和整型类型之间的快速转换函数。

## 语法

C

```
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);
```

## 参数

*value*

[in] 要转换的浮点值。

## 返回值

转换的整数类型结果。

## 要求

标头: <intrin.h>

体系结构: x86、x64

## 注解

这些内部函数是快速转换函数，可以尽快执行有效转换。与在标准 C++ 中一样，快速转换未完全定义。对于无效转换，它们可能会生成不同的值或异常。结果取决于目标平

台、编译器选项和上下文。这些函数可用于处理已检查范围的值。或者可用于处理以永远不会导致无效转换的方式生成的值。

快速转换内部函数从 Visual Studio 2022 起可用。

**结束 Microsoft 专用**

## 另请参阅

[编译器内部函数](#)

[饱和度浮动点转换函数](#)

[Sentinel 浮点转换函数](#)

# 饱和度浮点转换函数

项目 • 2023/06/16

## Microsoft 专用

浮点类型和使用 ARM 处理器兼容饱和策略的整数类型之间的转换函数。

## 语法

C

```
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);
```

## 参数

*value*

[in] 要转换的浮点值。

## 返回值

转换的整数类型结果。

## 要求

标头: <intrin.h>

体系结构: x86、x64

## 注解

这些内部函数是使用饱和度策略的浮点到整数类型转换函数：如果浮点值太高以至于无法适应目标类型，则它将映射到可能的最高目标值。如果值太低以至于无法适应，则将映射到可能的最低值。如果源值为 NaN，则结果中返回零。

饱和度转换内部函数从 Visual Studio 2019 版本 16.10 开始可用。

**结束 Microsoft 专用**

## 另请参阅

[编译器内部函数](#)

[快速浮点转换函数](#)

[Sentinel 浮点转换函数](#)

# Sentinel 浮点转换函数

项目 • 2023/10/21

## Microsoft 专用

使用与 Intel 体系结构 (IA) AVX-512 兼容的 sentinel 策略的浮点类型和整数类型之间的转换函数。

## 语法

C

```
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);
```

## 参数

*value*

[in] 要转换的浮点值。

## 返回值

转换的整数类型结果。

## 要求

标头: `<intrin.h>`

体系结构: x86、x64

## 注解

这些内在函数是使用 sentinel 策略的浮点到整型类型转换函数：它们返回距离零最远的结果值作为 `Nan` 的代理 sentinel 值。任何无效转换都将返回此 sentinel 值。返回的特定

sentinel 值取决于结果类型。

结果类型	Sentinel	<limits.h> 常数
int	-2147483648 (0x80000000)	INT_MIN
unsigned int	4294967295 (0xFFFFFFFF)	UINT_MAX
long long	-9223372036854775808 (0x8000000000000000)	LLONG_MIN
unsigned long long	18446744073709551615 (0xFFFFFFFFFFFFFFFF)	ULLONG_MAX

Sentinel 转换内部函数从 Visual Studio 2019 版本 16.10 开始可用。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[快速浮点转换函数](#)

[饱和度浮动点转换函数](#)

# `__getcallerseflags`

项目 • 2024/08/04

Microsoft 专用

从调用方上下文中返回 EFLAGS 值。

## 语法

C

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

## 返回值

调用方上下文中的 EFLAGS 值。

## 要求

[] 展开表

Intrinsic	体系结构
<code>__getcallerseflags</code>	x86、x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

## 示例

C++

```
// 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;
}
```

## Output

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

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

## 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获取帮助

# \_\_halt

项目 • 2023/10/12

## Microsoft 专用

停止微控制器，直至发生某个已启用的中断、无法屏蔽的中断 (NMI) 或重置。

## 语法

C

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

## 要求

Intrinsic	体系结构
<code>__halt</code>	x86、x64

头文件<intrin.h>

## 备注

`__halt` 函数等同于 `HLT` 计算机指令，并且仅在内核模式下可用。有关详细信息，请在 [Intel Corporation](#) 站点上搜索文档“体系结构软件开发人员手册第 2 卷：指令集参考”。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# \_InterlockedAdd 内部函数

项目 • 2023/06/16

## Microsoft 专用

当多线程拥有对共享变量的访问权限时，这些函数执行原子加法可确保成功完成操作。

## 语法

C

```
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
);
```

## 参数

## 加数

[in, out] 指向要相加的整数的指针；由加法结果代替。

## 值

[in] 要相加的值。

## 返回值

两个函数都将返回加法结果。

## 要求

Intrinsic	体系结构
_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

头文件 <intrin.h>

## 注解

带 `_acq` 或 `_rel` 后缀的这些版本的函数可在获取或发布语义后执行互锁加法。获取语义的意思是，在任何后续内存进行读取和写入之前，所有线程和处理器均能看见运算结果。进入临界区时，获取十分有用。发布语义的意思是，在运算结果自己显示出来之前，将强制所有内存的读取和写入对所有线程和处理器可见。离开临界区时，发布十分有用。带有 `_nf` (“no fence”) 后缀的内部函数不充当内存屏障。

这些例程只能用作内部函数。

## 示例： `_InterlockedAdd`

C++

```
// 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%lx 0x%lx 0x%lx", data1, data2, retval);
}
```

## 输出: \_InterlockedAdd

Output

```
0xffffffff00 0xff0000 0xffffffff00
```

## 示例: \_InterlockedAdd64

C++

```
// 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 = 0x00FF0000FFFFFFFFFF;
    __int64 retval;
    cout << hex << data1 << " + " << data2 << " = " ;
    retval = _InterlockedAdd64(&data1, data2);
    cout << data1 << endl;
    cout << "Return value: " << retval << endl;
}
```

## 输出: \_InterlockedAdd64

Output

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

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[与 x86 编译器冲突](#)

# \_InterlockedAddLargeStatistic

项目 • 2023/06/16

Microsoft 专用

执行互锁加法，其中第一个操作数为 64 位值。

## 语法

C

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

## 参数

### 加数

[in,out] 指向添加操作的第一个操作数的指针。指向的值由加法得出的结果代替。

### 值

[in] 第二个操作数；要添加到第一个操作数的值。

## 返回值

第二个操作数的值。

## 要求

Intrinsic	体系结构
_InterlockedAddLargeStatistic	x86

头文件<intrin.h>

## 注解

`_InterlockedAddLargeStatistic` 内部函数不是原子的，因为它作为两个单独的锁定指令实现。执行内部函数期间在另一个线程上发生的原子 64 位读取可能会导致读取不一致的值。

`_InterlockedAddLargeStatistic` 表现为读写屏障。有关详细信息，请参阅[\\_ReadWriteBarrier](#)。

**结束 Microsoft 专用**

## 另请参阅

[编译器内部函数  
与 x86 编译器冲突](#)

# \_InterlockedAnd 内部函数

项目 • 2024/08/04

## Microsoft 专用

用于在由多线程共享的变量上执行原子按位“与”操作。

## 语法

C

```
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  
);
```

## 参数

### *value*

[in, out] 指向第一个操作数的指针，将由结果替换。

### 掩码

[in] 第二个操作数。

## 返回值

第一个操作数的原始值。

## 要求

 展开表

Intrinsic	体系结构	Header
.-	x86、ARM、x64、ARM64	<intrin.h>
<code>_InterlockedAnd64</code>	ARM、x64、ARM64	<intrin.h>
	ARM、ARM64	<intrin.h>
	x64	<intrin.h>
	x86、x64	<immintrin.h>

## 注解

每个函数名称中的数字指定了参数的位大小。

在 ARM 和 ARM64 平台上，可以使用带 `_acq` 和 `_rel` 后缀的内部函数获取和发布语义，例如在临界区的起始位置。带 `_nf`（“无围墙”）后缀的内部函数不能充当内存屏障。

带 `_np`（“无预取”）后缀的函数可以阻止编译器插入可能的预取操作。

在支持硬件锁省略 (HLE) 指令的 Intel 平台，带 `_HLEAcquire` 和 `_HLERelease` 后缀的内部函数包括一个发送到处理器的提示，可以通过消除硬件中的锁写步骤来提升速度。如果在不支持 HLE 的平台上调用这些函数，则忽略此提示。

## 示例

C++

```
// 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%08x 0x%08x 0x%08x", data1, data2, retval);
}
```

Output

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

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[与 x86 编译器冲突](#)

## 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_interlockedbittestandreset 内部函数

项目 • 2024/08/04

Microsoft 专用

生成一个指令，该指令将地址 `a` 的位 `b` 设置为零并返回其原始值。

## 语法

C

```
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
);
```

## 参数

*a*

[in] 指向要检查的内存的指针。

*b*

[in] 要测试的位位置。

## 返回值

由 *b* 指定的位置上的位的原始值。

## 要求

[展开表](#)

Intrinsic	体系结构	头文件
<code>_interlockedbittestandreset</code>	x86、ARM、x64、ARM64	<code>&lt;intrin.h&gt;</code>
<code>.-.</code>	ARM、ARM64	<code>&lt;intrin.h&gt;</code>
<code>.-.</code>	ARM64	<code>&lt;intrin.h&gt;</code>
<code>%&gt;</code>	x86、x64	<code>&lt;immintrin.h&gt;</code>
<code>_interlockedbittestandreset64</code>	x64、ARM64	<code>&lt;intrin.h&gt;</code>
<code>%&gt;</code>	x64	<code>&lt;immintrin.h&gt;</code>

## 备注

在 x86 和 x64 处理器上，这些内部函数使用 `lock btr` 指令在原子操作中读取指定的位并将其设置为零。

在 ARM 处理器上，使用带 `_acq` 和 `_rel` 后缀的内部函数（例如在临界区的起点和结尾处）获取和发布语义。带 `_nf`（“无围墙”）后缀的 ARM 内部函数不能充当内存屏障。

在支持硬件锁省略 (HLE) 指令的 Intel 处理器上，带 `_HLEAcquire` 和 `_HLERelease` 后缀的内部函数包括一条发送到处理器的提示，可通过消除硬件中的锁写步骤加快速度。如果在不支持 HLE 的处理器上调用这些内部函数，则忽略此提示。

这些例程只能用作内部函数。

**结束 Microsoft 专用**

## 另请参阅

[编译器内部函数  
与 x86 编译器冲突](#)

---

## 反馈

此页面是否有帮助？



是



否

[提供产品反馈](#) | [在 Microsoft Q&A 获取帮助](#)

# \_interlockedbittestandset 内部函数

项目 • 2024/08/04

Microsoft 专用

生成可以检查地址 `a` 的位 `b` 的指令，返回其当前值，然后将位设置为 1。

## 语法

C

```
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
);
```

## 参数

*a*

[in] 指向要检查的内存的指针。

*b*

[in] 要测试的位位置。

## 返回值

在位置 *b* 上的位在其设置前的值。

## 要求

[展开表](#)

Intrinsic	体系结构	头文件
<code>_interlockedbittestandset</code>	x86、ARM、x64、ARM64	<code>&lt;intrin.h&gt;</code>
<code>.-.</code>	ARM、ARM64	<code>&lt;intrin.h&gt;</code>
<code>.-.</code>	ARM64	<code>&lt;intrin.h&gt;</code>
<code>%&gt;</code>	x86、x64	<code>&lt;immintrin.h&gt;</code>
<code>_interlockedbittestandset64</code>	x64、ARM64	<code>&lt;intrin.h&gt;</code>
<code>%&gt;</code>	x64	<code>&lt;immintrin.h&gt;</code>

## 注解

在 x86 和 x64 处理器上，这些内部函数使用 `lock bts` 指令读取指定的位并将其设置为 1。此操作为原子性操作。

在 ARM 和 ARM64 处理器上，使用带 `_acq` 和 `_rel` 后缀的内部函数（例如在临界区的起点和结尾处）获取和发布语义。带 `_nf`（“无围墙”）后缀的 ARM 内部函数不能充当内存屏障。

在支持硬件锁省略 (HLE) 指令的 Intel 处理器上，带 `_HLEAcquire` 和 `_HLERelease` 后缀的内部函数包括一条发送到处理器的提示，可通过消除硬件中的锁写步骤加快速度。如果在不支持 HLE 的处理器上调用这些内部函数，则忽略此提示。

这些例程只能用作内部函数。

## 结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[与 x86 编译器冲突](#)

---

## 反馈

此页面是否有帮助？

 是

 否

[提供产品反馈](#) | [在 Microsoft Q&A 获取帮助](#)

# \_InterlockedCompareExchange 内部函数

项目 • 2024/08/04

Microsoft 专用

执行联锁比较和交换。

## 语法

C

```
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
);
```

## 参数

### *Destination*

[in, out] 指向目标值的指针。忽略此标记。

### *Exchange*

[in] 交换值。忽略此标记。

### *Comparand*

[in] 与 *Destination* 指向的值进行比较的值。忽略此标记。

## 返回值

返回值是 *Destination* 指针指向的初始值。

## 要求

 展开表

Intrinsic	体系结构	标头
	x86、ARM、x64、ARM64	<intrin.h>
	ARM、ARM64	<intrin.h>

Intrinsic	体系结构	标头
.	x64	<intrin.h>
	x86、x64	<immintrin.h>

## 注解

`_InterlockedCompareExchange` 将 `Destination` 指向的值与 `Comparand` 值进行原子比较。如果 `Destination` 值等于 `Comparand` 值，`Exchange` 值将存储在由 `Destination` 指定的地址。否则，不执行任何操作。

`_InterlockedCompareExchange` 为 Win32 Windows SDK [InterlockedCompareExchange](#) 函数提供编译器内部支持。

`_InterlockedCompareExchange` 存几种变体，这些变体根据其涉及的数据类型和是否使用特定于处理器获取或发布语义而有所不同。

当 `_InterlockedCompareExchange` 函数对 32 位 `long` 整数值操作时，`_InterlockedCompareExchange8` 对 8 位整数值操作，`_InterlockedCompareExchange16` 对 16 位 `short` 整数值操作且 `_InterlockedCompareExchange64` 对 64 位整数值操作。若要详细了解 128 位值的类似内部函数，请参阅 [\\_InterlockedCompareExchange128](#)。

在所有 ARM 平台上，可以使用带 `_acq` 和 `_rel` 后缀的内部函数获取和发布语义，例如在关键部分的起始位置。带 `_nf`（“无围墙”）后缀的 ARM 内部函数不能充当内存屏障。

带 `_np`（“无预取”）后缀的函数可以阻止编译器插入可能的预取操作。

在支持硬件锁省略 (HLE) 指令的 Intel 平台，带 `_HLEAcquire` 和 `_HLERelease` 后缀的内部函数包括一个发送到处理器的提示，可以通过消除硬件中的锁写步骤来提升速度。如果在不支持 HLE 的平台上调用这些内部函数，则忽略此提示。

这些例程只能用作内部函数。

## 示例

在以下示例中，`_InterlockedCompareExchange` 用于简单的低等级线程同步。此方法作为多线程编程基础时有其自身的局限性；介绍此方法旨在说明联锁内部函数的典型用法。要得到最佳结果，请使用 Windows API。有关多线程编程的详细信息，请参阅[编写多线程 Win32 程序](#)。

```

// 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)
    {
#ifndef 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!
        }
#endif
    }
}

```

```

        // 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.
        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,

```

```
        &unused1,
        0,
        (LPDWORD)&unused2);

    WaitForMultipleObjects(4, threads, TRUE, timeoutTime);

    return 0;
}
```

### Output

```
<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
```

**结束 Microsoft 专用**

## 另请参阅

[\\_InterlockedCompareExchange128](#)  
[\\_InterlockedCompareExchangePointer 内部函数](#)  
[编译器内部函数](#)

## 反馈

此页面是否有帮助?

是

否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_InterlockedCompareExchange128 内部函数

项目 • 2024/08/04

Microsoft 专用

执行 128 位联锁比较和交换。

## 语法

```
C

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
);
```

## 参数

## 目标

[in,out] 指向目标的指针，它是两个 64 位整数的数组，被视为 128 位字段。目标数据必须对齐 16 字节，以避免出现常规保护错误。

### ExchangeHigh

[in] 可与目标高部分交换的 64 位整数。

### ExchangeLow

[in] 可与目标低部分交换的 64 位整数。

### ComparandResult

[in,out] 指向两个 64 位整数（被视为 128 位字段）的数组的指针，用于跟目标比较。在输出中，此数组被目标的原始值覆盖。

## 返回值

如果 128 位比较数与目标的原始值相等，则为 1。`ExchangeHigh` 和 `ExchangeLow` 覆盖 128 位目标。

如果比较数不等于目标的原始值，则为 0。目标的值不变，比较数的值将被目标的值覆盖。

## 要求

 展开表

Intrinsic	体系结构
<code>_InterlockedCompareExchange128</code>	x64、ARM64
<code>..</code>	ARM64
<code>_InterlockedCompareExchange128_np</code>	x64

头文件<intrin.h>

## 备注

`_InterlockedCompareExchange128` 内部函数生成前缀为 `lock` 的 `cmpxchg16b` 指令，以执行 128 位锁定比较和交换。AMD 64 位硬件的早期版本不支持此指令。若要检查 `cmpxchg16b` 指令的硬件支持，请使用 `InfoType=0x00000001 (standard function 1)` 调用 `_cpuid` 内部函数。如果支持指令，则 13 位 `CPUInfo[2]` (ECX) 为 1。

## ① 备注

值 `ComparandResult` 始终被覆盖。指令 `lock` 后，此内部函数会立即将初始值 `Destination` 复制到 `ComparandResult`。因此，`ComparandResult` 和 `Destination` 应指向单独的内存位置以避免意外行为。

尽管 `_InterlockedCompareExchange128` 可用于低级线程同步，但如果可以使用较小的同步函数（比如另一个 `_InterlockedCompareExchange` 内部函数），则无需同步超过 128 位。如果要在内存中对 128 位值进行原子访问，请使用 `_InterlockedCompareExchange128`。

如果在不支持 `cpxchg16b` 指令的硬件上运行使用内部函数的代码，则结果是不可预测的。

在 ARM 平台上，可以使用带 `_acq` 和 `_rel` 后缀的内部函数获取和发布语义，例如在临界区的起始位置。带 `_nf`（“无围墙”）后缀的 ARM 内部函数不能充当内存屏障。

带 `_np`（“无预取”）后缀的函数可以阻止编译器插入可能的预取操作。

此例程仅可用作内部函数。

## 示例

此示例使用 `_InterlockedCompareExchange128` 将两个 64 位整数数组的高字替换为其高字和低字的总和，并递增低字。对 `BigInt` 数组的访问是原子访问，但此示例使用单线程，并忽略锁定以简化操作。

C++

```
// cpxchg16b.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]);
}
```

## Output

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

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[\\_InterlockedCompareExchange 内部函数](#)

[与 x86 编译器冲突](#)

## 反馈

此页面是否有帮助?

是

否

[提供产品反馈](#) | 在 Microsoft Q&A 获取帮助

# \_InterlockedCompareExchangePointer 内部函数

项目 • 2024/08/04

## Microsoft 专用

如果 `Exchange` 和 `Destination` 地址相等，则执行原子操作，将 `Comparand` 地址存储在 `Destination` 地址中。

## 语法

```
C

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  
);
```

## 参数

### 目标

[in, out] 指向目标值的指针。忽略此标记。

### Exchange

[in] 交换指针。忽略此标记。

### Comparand

[in] 与目标值比较的指针。忽略此标记。

## 返回值

返回值是目标的初始值。

## 要求

 展开表

Intrinsic	体系结构	头文件
_InterlockedCompareExchangePointer	x86、ARM、x64、ARM64	<intrin.h>
.-.	ARM、ARM64	<iintrin.h>
%>	x86、x64	<immintrin.h>

## 备注

`_InterlockedCompareExchangePointer` 执行 `Destination` 地址与 `Comparand` 地址之间的原子比较。如果 `Destination` 地址与 `Comparand` 地址相等，则 `Exchange` 地址将存储在由 `Destination` 指定的地址中。否则，不会执行任何操作。

`_InterlockedCompareExchangePointer` 对 Win32 Windows SDK `InterlockedCompareExchangePointer` 函数提供了编译器内部函数支持。

有关如何使用 `_InterlockedCompareExchangePointer` 的示例，请参阅 [\\_InterlockedDecrement](#)。

ARM 平台上，如果需要（例如在临界区的起点和终点）获取和发布语义，可以使用带 `_acq` 和 `_rel` 后缀的函数。带 `_nf`（“无围墙”）后缀的 ARM 内部函数不能充当内存屏障。

带 `_np`（“无预取”）后缀的函数可以阻止编译器插入可能的预取操作。

在支持硬件锁省略 (HLE) 指令的 Intel 平台，带 `_HLEAcquire` 和 `_HLERelease` 后缀的内部函数包括一个发送到处理器的提示，可以通过消除硬件中的锁写步骤来提升速度。如果在不支持 HLE 的平台上调用这些内部函数，则忽略此提示。

这些例程只能用作内部函数。

## 结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)  
[关键字](#)

---

## 反馈

此页面是否有帮助？

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获取帮助

# \_InterlockedDecrement 内部函数

项目 • 2024/08/04

为 Win32 Windows SDK [InterlockedDecrement](#) 函数提供编译器内部支持。

\_InterlockedDecrement 内部函数是 Microsoft 特定函数。

## 语法

```
C

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
);
```

# 参数

`lpAddend`

[in, out] 指向要递减的变量的易失指针。

# 返回值

返回值是生成的递减值。

# 要求

 展开表

Intrinsic	体系结构
<code>%&gt;</code>	x86、ARM、x64、ARM64
<code>_InterlockedDecrement64</code>	ARM、x64、ARM64
	ARM、ARM64

头文件<intrin.h>

# 注解

`_InterlockedDecrement` 存在几种变体，这些变体根据其涉及的数据类型和是否使用特定于处理器获取或发布语义而有所不同。

当 `_InterlockedDecrement` 函数对 32 位整数值操作时，`_InterlockedDecrement16` 对 16 位整数值操作且 `_InterlockedDecrement64` 可以对 64 位整数值操作。

ARM 平台上，如果需要（例如在临界区的起点和终点）获取和发布语义，可以使用带 `_acq` 和 `_rel` 后缀的函数。带 `_nf` (“no fence”) 后缀的内部函数不充当内存屏障。

由 `lpAddend` 参数指向的变量必须与 32 位边界对齐；否则，此函数在多处理器 x86 系统和任何非 x86 系统上将失效。有关详细信息，请参阅 [align](#)。

这些例程只能用作内部函数。

# 示例

C++

```
// 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);
}
```

## 另请参阅

[编译器内部函数](#)

[关键字](#)

[与 x86 编译器冲突](#)

---

## 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_InterlockedExchange 内部函数

项目 • 2024/08/04

Microsoft 专用

生成原子指令以设置指定的值。

## 语法

C

```
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
);
```

## 参数

### Target

[in, out] 指向要交换的值。此函数会将此变量设置为 `Value` 并返回其之前的值。

### 值

[in] 与 `Target` 指向的值交换的值。

# 返回值

返回由 `Target` 指向的初始值。

## 要求

 展开表

Intrinsic	体系结构	Header
<code>.-.</code>	x86、ARM、x64、ARM64	<code>&lt;intrin.h&gt;</code>
<code>_InterlockedExchange64</code>	ARM、x64、ARM64	<code>&lt;intrin.h&gt;</code>
	ARM、ARM64	<code>&lt;intrin.h&gt;</code>
<code>%&gt;</code>	x86、x64	<code>&lt;immintrin.h&gt;</code>
<code>%&gt;</code>	x64	<code>&lt;immintrin.h&gt;</code>

## 备注

`_InterlockedExchange` 为 Win32 Windows SDK [InterlockedExchange](#) 函数提供编译器内部支持。

`_InterlockedExchange` 存在几种变体，这些变体根据其涉及的数据类型和是否使用特定于处理器获取或发布语义而有所不同。

当 `_InterlockedExchange` 函数对 32 位整数值操作时，`_InterlockedExchange8` 对 8 位整数值操作，`_InterlockedExchange16` 对 16 位整数值操作且 `_InterlockedExchange64` 对 64 位整数值操作。

在 ARM 平台上，可以使用带 `_acq` 和 `_rel` 后缀的内部函数获取和发布语义，例如在临界区的起始位置。带 `_nf` (“no fence”) 后缀的内部函数不充当内存屏障。

在支持硬件锁省略 (HLE) 指令的 Intel 平台，带 `_HLEAcquire` 和 `_HLERelease` 后缀的内部函数包括一个发送到处理器的提示，可以通过消除硬件中的锁写步骤来提升速度。如果在不支持 HLE 的平台上调用这些函数，则忽略此提示。

这些例程只能用作内部函数。

## 示例

有关如何使用 `_InterlockedExchange` 的示例，请参阅 [\\_InterlockedDecrement](#)。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[关键字](#)

[与 x86 编译器冲突](#)

---

## 反馈

此页面是否有帮助？

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_InterlockedExchangeAdd 内部函数

项目 • 2024/08/04

## Microsoft 专用

对 Win32 Windows SDK [\\_InterlockedExchangeAdd 内部函数](#)提供了编译器内部函数支持。

## 语法

```
C

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
);

```

## 参数

### 加数

[in, out] 要相加的值；由加法得出的结果代替。

### 值

[in] 要相加的值。

# 返回值

返回值是由 `Addend` 参数指向的变量的初始值。

## 要求

 展开表

Intrinsic	体系结构	Header
<code>.-.</code>	x86、ARM、x64、ARM64	<code>&lt;intrin.h&gt;</code>
<code>_InterlockedExchangeAdd64</code>	ARM、x64、ARM64	<code>&lt;intrin.h&gt;</code>
	ARM、ARM64	<code>&lt;intrin.h&gt;</code>
<code>%&gt;</code>	x86、x64	<code>&lt;immintrin.h&gt;</code>
<code>%&gt;</code>	x64	<code>&lt;immintrin.h&gt;</code>

## 备注

`_InterlockedExchangeAdd` 存在几种变体，这些变体根据其涉及的数据类型和是否使用特定于处理器获取或发布语义而有所不同。

当 `_InterlockedExchangeAdd` 函数对 32 位整数值操作时，`_InterlockedExchangeAdd8` 对 8 位整数值操作，`_InterlockedExchangeAdd16` 对 16 位整数值操作且 `_InterlockedExchangeAdd64` 对 64 位整数值操作。

ARM 平台上，如果需要（例如在临界区的起点和终点）获取和发布语义，可以使用带 `_acq` 和 `_rel` 后缀的函数。带 `_nf` (“no fence”) 后缀的内部函数不充当内存屏障。

在支持硬件锁省略 (HLE) 指令的 Intel 平台，带 `_HLEAcquire` 和 `_HLERelease` 后缀的内部函数包括一个发送到处理器的提示，可以通过消除硬件中的锁写步骤来提升速度。如果在不支持 HLE 的平台上调用这些内部函数，则忽略此提示。

这些例程只能用作内部函数。即使使用 `/Oi` 或 `#pragma intrinsic`，它们也是内部函数。对于这些内部函数，不能使用 `#pragma function`。

## 示例

有关如何使用 `_InterlockedExchangeAdd` 的示例，请参阅 [\\_InterlockedDecrement](#)。

## 另请参阅

[编译器内部函数](#)

[关键字](#)

[与 x86 编译器冲突](#)

---

## 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_InterlockedExchangePointer 内部函数

项目 • 2024/08/04

Microsoft 专用

执行原子交换操作，将作为第二个自变量传入的地址复制到第一个自变量并返回第一个自变量的原始地址。

## 语法

```
C

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
);
```

## 参数

*Target*

[in,out] 指向要交换值的指针的指针。 函数将值设置为值并返回之前的值。

*值*

[in] 要与由目标指向的值交换的值。

# 返回值

函数返回由目标指向的初始值。

## 要求

 展开表

Intrinsic	体系结构	头文件
<code>_InterlockedExchangePointer</code>	x86、ARM、x64、ARM64	<code>&lt;intrin.h&gt;</code>
<code>.-.</code>	ARM、ARM64	<code>&lt;intrin.h&gt;</code>
<code>%&gt;</code>	x64	<code>&lt;immintrin.h&gt;</code>

在 x86 体系结构上，`_InterlockedExchangePointer` 是调用 `_InterlockedExchange` 的宏。

## 备注

在 64 位系统上，这些参数都是 64 位并且必须在 64 位边界上对齐。否则，该函数将失败。在 32 位系统上，这些参数都是 32 位并且必须在 32 位边界上对齐。有关详细信息，请参阅 [align](#)。

ARM 平台上，如果需要（例如在临界区的起点和终点）获取和发布语义，可以使用带 `_acq` 和 `_rel` 后缀的函数。带 `_nf`（“无围墙”）后缀的内部函数不能充当内存屏障。

在支持硬件锁省略 (HLE) 指令的 Intel 平台，带 `_HLEAcquire` 和 `_HLERelease` 后缀的内部函数包括一个发送到处理器的提示，可以通过消除硬件中的锁写步骤来提升速度。如果在不支持 HLE 的平台上调用这些内部函数，则忽略此提示。

这些例程只能用作内部函数。

**结束 Microsoft 专用**

## 另请参阅

[编译器内部函数](#)  
[与 x86 编译器冲突](#)

---

## 反馈

此页面是否有帮助？

是

否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_InterlockedIncrement 内部函数

项目 • 2024/08/04

对 Win32 Windows SDK [InterlockedIncrement](#) 函数提供编译器内部函数支持。

\_InterlockedIncrement 内部函数是 Microsoft 特定函数。

## 语法

C

```
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
);
```

# 参数

`lpAddend`

[in, out] 指向要递增的变量的指针。

# 返回值

返回值是生成的递增值。

# 要求

 展开表

Intrinsic	体系结构	头文件
<code>%&gt;</code>	x86、ARM、x64、ARM64	<code>&lt;intrin.h&gt;</code>
<code>_InterlockedIncrement64</code>	ARM、x64、ARM64	<code>&lt;intrin.h&gt;</code>
	ARM、ARM64	<code>&lt;intrin.h&gt;</code>

# 备注

`_InterlockedIncrement` 存在几种变体，这些变体根据其涉及的数据类型和是否使用特定于处理器获取或发布语义而有所不同。

当 `_InterlockedIncrement` 函数对 32 位整数值操作时，`_InterlockedIncrement16` 对 16 位整数值操作且 `_InterlockedIncrement64` 可以对 64 位整数值操作。

ARM 平台上，如果需要（例如在临界区的起点和终点）获取和发布语义，可以使用带 `_acq` 和 `_rel` 后缀的函数。带 `_nf`（“无围墙”）后缀的内部函数不能充当内存屏障。

由 `lpAddend` 参数指向的变量必须与 32 位边界对齐；否则，此函数在多处理器 x86 系统和任何非 x86 系统上将失效。有关详细信息，请参阅 [align](#)。

Win32 函数在 `Wdm.h` 或 `Ntddk.h` 中声明。

这些例程只能用作内部函数。

# 示例

有关如何使用 `_InterlockedIncrement` 的示例，请参阅 [\\_InterlockedDecrement](#)。

# 另请参阅

[编译器内部函数](#)

[关键字](#)

[与 x86 编译器冲突](#)

---

## 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_InterlockedOr 内部函数

项目 • 2024/08/04

Microsoft 专用

对由多线程共享的变量执行原子位或操作。

## 语法

C

```
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  
);
```

## 参数

### 值

[in, out] 指向第一个操作数的指针，将由结果替换。

### 掩码

[in] 第二个操作数。

## 返回值

由第一个参数指向的原始值。

## 要求

 展开表

Intrinsic	体系结构	Header
.-.	x86、ARM、x64、ARM64	<intrin.h>
_InterlockedOr64	ARM、x64、ARM64	<intrin.h>
	ARM、ARM64	<intrin.h>
	x64	<intrin.h>
%>	x86、x64	<immintrin.h>
%>	x64	<immintrin.h>

## 注解

每个函数名称中的数字指定了参数的位大小。

ARM 平台上，如果需要（例如在临界区的起点和终点）获取和发布语义，可以使用带 `_acq` 和 `_rel` 后缀的函数。带 `_nf`（“无围墙”）后缀的 ARM 内部函数不能充当内存屏障。

带 `_np`（“无预取”）后缀的函数可以阻止编译器插入可能的预取操作。

在支持硬件锁省略 (HLE) 指令的 Intel 平台，带 `_HLEAcquire` 和 `_HLERelease` 后缀的内部函数包括一个发送到处理器的提示，可以通过消除硬件中的锁写步骤来提升速度。如果在不支持 HLE 的平台上调用这些内部函数，则忽略此提示。

## 示例

C++

```
// _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%lx 0x%lx 0x%lx", data1, data2, retval);
}
```

Output

```
0xffffffff00 0xfffff00 0xff00ff00
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数  
与 x86 编译器冲突](#)

## 反馈

此页面是否有帮助？

是

否

[提供产品反馈](#) | 在 Microsoft Q&A 获取帮助

# \_InterlockedXor 内部函数

项目 • 2024/08/04

## Microsoft 专用

在由多线程共享的变量上执行原子按位异或 (XOR) 操作。

## 语法

C

```
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  
);
```

## 参数

### 值

[in, out] 指向第一个操作数的指针，将由结果替换。

### 掩码

[in] 第二个操作数。

## 返回值

第一个操作数的原始值。

## 要求

 展开表

Intrinsic	体系结构	Header
.-.	x86、ARM、x64、ARM64	<intrin.h>
_InterlockedXor64	ARM、x64、ARM64	<intrin.h>
	ARM、ARM64	<intrin.h>
	x64	<intrin.h>
%>	x86、x64	<immintrin.h>
%>	x64	<immintrin.h>

## 注解

每个函数名称中的数字指定了参数的位大小。

ARM 平台上，如果需要（例如在临界区的起点和终点）获取和发布语义，可以使用带 `_acq` 和 `_rel` 后缀的函数。带 `_nf`（“无围墙”）后缀的 ARM 内部函数不能充当内存屏障。

带 `_np`（“无预取”）后缀的函数可以阻止编译器插入可能的预取操作。

在支持硬件锁省略 (HLE) 指令的 Intel 平台，带 `_HLEAcquire` 和 `_HLERelease` 后缀的内部函数包括一个发送到处理器的提示，可以通过消除硬件中的锁写步骤来提升速度。如果在不支持 HLE 的平台上调用这些内部函数，则忽略此提示。

## 示例

C++

```
// _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%lx 0x%lx 0x%lx", data1, data2, retval);
}
```

Output

```
0xfffff0000 0xfffff00 0xff00ff00
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数  
与 x86 编译器冲突](#)

## 反馈

此页面是否有帮助？

是

否

[提供产品反馈](#) | 在 Microsoft Q&A 获取帮助

# **\_\_inbyte**

项目 • 2023/10/12

**Microsoft 专用**

生成 `in` 指令，返回从 `Port` 指定的端口读取的单个字节。

## 语法

C

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

## 参数

端口

[in] 要从中读取数据的端口。

## 返回值

从指定端口读取的字节。

## 要求

Intrinsic	体系结构
<code>__inbyte</code>	x86、x64

头文件<intrin.h>

结束 Microsoft 专用

## 备注

此例程仅可用作内部函数。

## 另请参阅

## 编译器内部函数

# \_\_inbytestring

项目 • 2024/08/04

Microsoft 专用

使用 `rep insb` 指令从指定端口读取数据。

## 语法

C

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

## 参数

端口

[in] 要从中读取数据的端口。

Buffer

[out] 从该端口读取的数据在此处写入。

计数

[in] 要读取的字节和数据的数量。

## 要求

展开表

Intrinsic	体系结构
<code>__inbytestring</code>	x86、x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

## 另请参阅

[编译器内部函数](#)

---

## 反馈

此页面是否有帮助?



[提供产品反馈](#) | [在 Microsoft Q&A 获得帮助](#)

# **\_\_incfsbyte, \_\_incfsword, \_\_incfsdword**

项目 • 2023/10/12

**Microsoft 专用**

将 1 添加到由相对于 **FS** 段开头的偏移量指定的内存位置的值。

## 语法

```
C

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

## 参数

*Offset*

[in] 与 **FS** 的起始位置间的偏移量。

## 要求

Intrinsic	体系结构
<code>__incfsbyte</code>	x86
<code>__incfsword</code>	x86
<code>__incfsdword</code>	x86

头文件<intrin.h>

## 备注

这些内部函数仅在内核模式下可用，并且例程仅作为内部函数提供。

## 另请参阅

[\\_addfsbyte, \\_addfsword, \\_addfsdword](#)  
[\\_readfsbyte, \\_readfsdword, \\_readfsqword, \\_readfsword](#)  
[\\_writefsbyte, \\_writefsdword, \\_writefsqword, \\_writefsword](#)  
[编译器内部函数](#)

# **\_\_incgsbyte, \_\_incgsword, \_\_incgsdword, \_\_incgsqword**

项目 • 2024/08/04

## Microsoft 专用

将 1 添加到由相对于 **GS** 段开头的偏移量指定的内存位置的值。

## 语法

C

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

## 参数

*Offset*

[in] 与 **GS** 的起始位置间的偏移量。

## 要求

[+] 展开表

Intrinsic	体系结构
<b>__incgsbyte</b>	X64
<b>__incgsword</b>	X64
<b>__incgsdword</b>	X64

Intrinsic	体系结构
__incgsqword	X64

头文件<intrin.h>

## 备注

这些例程只能用作内部函数。

结束 Microsoft 专用

## 另请参阅

[\\_addgsbyte](#), [\\_addgsword](#), [\\_addgsdword](#), [\\_addgsqword](#)  
[\\_readgsbyte](#), [\\_readgsdword](#), [\\_readgsqword](#), [\\_readgsword](#)  
[\\_writegsbyte](#), [\\_writegsdword](#), [\\_writegsqword](#), [\\_writegsword](#)  
编译器内部函数

---

## 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_\_indword

项目 · 2024/08/04

Microsoft 专用

使用 `in` 指令从指定端口读取双字数据。

## 语法

C

```
unsigned long __indword(  
    unsigned short Port  
) ;
```

## 参数

端口

[in] 要从中读取数据的端口。

## 返回值

从端口读取的字。

## 要求

[] 展开表

Intrinsic	体系结构
<code>__indword</code>	x86、x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

结束 Microsoft 专用

# 另请参阅

[编译器内部函数](#)

---

## 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | [在 Microsoft Q&A 获取帮助](#)

# \_\_indwordstring

项目 • 2024/08/04

Microsoft 专用

使用 `rep insd` 指令从指定端口读取数据。

## 语法

C

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

## 参数

端口

[in] 要从中读取数据的端口。

Buffer

[out] 从该端口读取的数据在此处写入。

计数

[in] 要读取的字节和数据的数量。

## 要求

展开表

Intrinsic	体系结构
<code>__indwordstring</code>	x86、x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

## 另请参阅

[编译器内部函数](#)

---

## 反馈

此页面是否有帮助?



[提供产品反馈](#) | [在 Microsoft Q&A 获得帮助](#)

# \_\_int2c

项目 • 2023/06/16

## Microsoft 专用

生成 `int 2c` 指令，该指令会触发 `2c` 中断。

## 语法

C

```
void __int2c(void);
```

## 要求

Intrinsic	体系结构
<code>__int2c</code>	x86、x64

头文件<intrin.h>

## 结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# `__invlpg`

项目 • 2023/10/12

## Microsoft 专用

生成 x86 `invlpg` 指令，该指令针对与 Address 指向的内存相关联的页面使翻译外观缓冲区 (TLB) 失效。

## 语法

C

```
void __invlpg(  
    void* Address  
)
```

## 参数

*Address*

[in] 64 位地址。

## 要求

Intrinsic	体系结构
<code>__invlpg</code>	x86、x64

头文件<intrin.h>

## 备注

内部函数 `__invlpg` 发出特权指令，并且仅在内核模式下可用，其特权级别 (CPL) 为 0。

此例程仅可用作内部函数。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# \_\_inword

项目 · 2024/08/04

Microsoft 专用

使用 `in` 指令从指定端口读取数据。

## 语法

C

```
unsigned short __inword(  
    unsigned short Port  
>;
```

## 参数

端口

[in] 要从中读取数据的端口。

## 返回值

读取的数据字。

## 要求

展开表

Intrinsic	体系结构
<code>__inword</code>	x86、x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

结束 Microsoft 专用

# 另请参阅

[编译器内部函数](#)

---

## 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_\_inwordstring

项目 • 2023/10/12

Microsoft 专用

使用 `rep insw` 指令从指定端口读取数据。

## 语法

C

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

## 参数

端口

[in] 要从中读取数据的端口。

Buffer

[out] 从该端口读取的数据在此处写入。

计数

[in] 要读取的数据的字数。

## 要求

Intrinsic	体系结构
<code>__inwordstring</code>	x86、x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

结束 Microsoft 专用

# 另请参阅

[编译器内部函数](#)

# **\_\_lidt**

项目 • 2023/10/12

**Microsoft 专用**

在中断描述符表寄存器 (IDTR) 中加载指定内存位置中的值。

## 语法

C

```
void __lidt(void * Source);
```

## 参数

*Source*

[in] 指向要复制到 IDTR 的值的指针。

## 要求

Intrinsic	体系结构
<code>__lidt</code>	x86、x64

头文件<intrin.h>

## 备注

`__lidt` 函数等同于 `LIDT` 计算机指令，并且仅在内核模式下可用。有关详细信息，请在 [Intel Corporation](#) 站点上搜索文档“体系结构软件开发人员手册第 2 卷：指令集参考”。

**结束 Microsoft 专用**

## 另请参阅

[编译器内部函数](#)

[\\_sidt](#)

# \_\_ll\_lshift

项目 • 2023/10/12

**Microsoft 专用**

将提供的 64 位值向左移动指定的位数。

## 语法

C

```
unsigned __int64 __ll_lshift(
    unsigned __int64 Mask,
    int nBit
);
```

## 参数

*掩码*

[in] 要左移的 64 位整数值。

*nBit*

[in] 要位移的位数。

## 返回值

左移 *nBit* 位的掩码。

## 要求

Intrinsic	体系结构
<code>__ll_lshift</code>	x86、x64

头文件<intrin.h>

## 备注

如果为 64 位体系结构编译程序，并且 `nBit` 大于 63，则要移动的位数是 `nBit` 对 64 取模。如果为 32 位体系结构编译程序，并且 `nBit` 大于 31，则要移动的位数是 `nBit` 对 32 取模。

名称中的 `ll` 表示它是对 `long long` (`_int64`) 的操作。

## 示例

C++

```
// 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;
}
```

## 输出

Output

```
10000
```

### ① 备注

没有无符号版本的左移操作。这是因为 `__ll_lshift` 已使用无符号输入参数。与右移不同，左移没有符号依赖，因为无论值移动的符号如何，结果中的最低有效位始终设置为零。

结束 Microsoft 专用

## 另请参阅

`_ll_rshift`

`_ull_rshift`

编译器内部函数

# ll\_rshift

项目 • 2023/10/12

**Microsoft 专用**

将第一个参数指定的 64 位值右移第二个参数指定的位数。

## 语法

C

```
_int64 __ll_rshift(  
    _int64 Mask,  
    int nBit  
) ;
```

## 参数

*掩码*

[in] 要右移的 64 位整数值。

*nBit*

[in] 要移位的位数，在 x64 上按 64 取模，在 x86 上按 32 取模。

## 返回值

移位 *nBit* 位的掩码。

## 要求

Intrinsic	体系结构
<code>__ll_rshift</code>	x86、x64

头文件<intrin.h>

## 注解

如果第二个参数在 x64 上大于 64 (x86 上为 32) , 则将该参数按 64 取模 (x86 上为 32) 以确定要移位的位数。 `ll` 前缀表示它是对 `long long` (64 位有符号整数类型 `_int64` 的另一个名称) 的操作。

## 示例

C++

```
// 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;
}
```

## 输出

Output

```
fffffffffffff00
- 100
fffffffffffff00
- 10
```

### ① 备注

如果使用了 `_ull_rshift` , 则右移值的 MSB 将为零, 因此如果是负值, 则不会获得所需的结果。

结束 Microsoft 专用

# 另请参阅

[编译器内部函数](#)

[\\_ll\\_lshift](#)

[\\_ull\\_rshift](#)

# \_lzcnt16、\_lzcnt、\_lzcnt64

项目 • 2023/06/16

Microsoft 专用

计算 16 位、32 位或 64 位整数中的前导零数。

## 语法

```
C

unsigned short __lzcnt16(
    unsigned short value
);
unsigned int __lzcnt(
    unsigned int value
);
unsigned __int64 __lzcnt64(
    unsigned __int64 value
);
```

## 参数

*value*

[in] 要扫描前导零的 16 位、32 位或 64 位无符号整数。

## 返回值

*value* 参数中的前导零位数。如果 *value* 为零，则返回值是输入操作数的大小（16、32 或 64）。如果 *value* 的最高有效位为 1，则返回值为零。

## 要求

Intrinsic	体系结构
<code>__lzcnt16</code>	AMD: 高级位操作 (ABM) Intel: Haswell
<code>__lzcnt</code>	AMD: 高级位操作 (ABM) Intel: Haswell

Intrinsic	体系结构
<code>__lzcnt64</code>	AMD: 64 位模式下的高级位操作 (ABM)。
	Intel: Haswell

头文件<intrin.h>

## 注解

每个内部函数都会生成 `lzcnt` 指令。`lzcnt` 指令返回的值的大小与其参数的大小相同。在 32 位模式下，没有 64 位通用寄存器，因此不支持 64 位 `lzcnt`。

若要确定 `lzcnt` 指令的硬件支持，请调用具有 `InfoType=0x80000001` 的 `_cpuid` 内部函数，并检查 `CPUInfo[2]` (ECX) 的第 5 位。如果支持该指令，该位将为 1，否则为 0。如果在不支持 `lzcnt` 指令的硬件上运行使用内部函数的代码，则无法预测结果。

在不支持 `lzcnt` 指令的 Intel 处理器上，指令字节编码作为 `bsr` (逆向位扫描) 执行。如果代码可移植性是一个问题，请考虑改用 `_BitScanReverse` 内部函数。有关详细信息，请参阅 [\\_BitScanReverse\\_BitScanReverse64](#)。

## 示例

C++

```
// 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;
    }
}
```

```
}
```

### Output

```
_lzcnt16(0x0) = 16
_lzcnt16(0xff) = 8
_lzcnt16(0xffff) = 0
_lzcnt(0x0) = 32
_lzcnt(0xff) = 24
_lzcnt(0xffff) = 16
_lzcnt(0xffffffff) = 0
```

## 结束 Microsoft 专用

这一部分内容由超威半导体公司保留 2007 部分版权。保留所有权利。经超威半导体公司  
许可转载

## 另请参阅

[编译器内部函数](#)

# `_mm_cvtsi64x_ss`

项目 • 2023/10/12

## Microsoft 专用

生成将 64 位整数转换为标量单精度浮点值的 x64 扩展版本 (`cvtsi2ss`) 指令。

## 语法

C

```
_m128 _mm_cvtsi64x_ss(
    _m128 a,
    _int64 b
);
```

## 参数

*a*

[in] 包含四个单精度浮点值的 `_m128` 结构。

*b*

[in] 要转换为浮点值的 64 位整数。

## 返回值

一个 `_m128` 结构，其第一个浮点值是转换的结果。其他三个值都是从 *a* 复制而不加更改。

## 要求

Intrinsic	体系结构
<code>_mm_cvtsi64x_ss</code>	x64

头文件<intrin.h>

## 注解

该 `_m128` 结构表示 XMM 寄存器，因此内部函数允许将系统内存中的值 `b` 移动到 XMM 寄存器中。

此例程仅可用作内部函数。

## 示例

C++

```
// _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] );
}
```

Output

```
54.000000 0.000000 0.000000 0.000000
```

结束 Microsoft 专用

## 另请参阅

[\\_m128](#)

[编译器内部函数](#)

# \_mm\_cvtss\_si64x

项目 • 2023/06/16

## Microsoft 专用

生成将标量单精度浮点数转换为 64 位整数 (`cvtss2si`) 指令的 x64 扩展版本。

## 语法

C

```
_int64 _mm_cvtss_si64x(  
    __m128 value  
)
```

## 参数

*value*

[in] 包含浮点值的 `__m128` 结构。

## 返回值

64 位整数，第一个浮点值转换为整数的结果。

## 要求

Intrinsic	体系结构
<code>_mm_cvtss_si64x</code>	X64

头文件<intrin.h>

## 注解

结构值的第一个元素将转换为整数并返回。 MXCSR 中的舍入控制位用于确定舍入行为。如果小数部分为 0.5，则默认舍入模式为舍入到最接近的偶数。由于 `__m128` 结构表示 XMM 寄存器，因此内部函数从 XMM 寄存器获取一个值，并将其写入系统内存。

此例程仅可用作内部函数。

## 示例

C++

```
// _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);
}
```

Output

```
101
```

结束 Microsoft 专用

## 另请参阅

[\\_m128d](#)

[编译器内部函数](#)

# \_mm\_cvttss\_si64x

项目 • 2023/10/12

## Microsoft 专用

使用截断单精度浮点数将 x64 扩展版本的转换发出到 64 位整数 (`cvtss2si`) 指令。

## 语法

```
C  
__int64 _mm_cvttss_si64x(  
    __m128 value  
)
```

## 参数

*value*

[in] 包含单精度浮点值的 `__m128` 结构。

## 返回值

将第一个浮点值转换为 64 位整数的结果。

## 要求

Intrinsic	体系结构
<code>_mm_cvttss_si64x</code>	x64

头文件<intrin.h>

## 备注

仅当不精确的转换被截断为零时，内部函数与 `_mm_cvtss_si64x` 不同。由于结构 `__m128` 表示 XMM 寄存器，因此生成的指令将数据从 XMM 寄存器移动到系统内存中。

此例程仅可用作内部函数。

## 示例

C++

```
// _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);
}
```

Output

101

结束 Microsoft 专用

## 另请参阅

[\\_\\_m128](#)

[编译器内部函数](#)

# **\_mm\_extract\_si64、\_mm\_extracti\_si64**

项目 • 2023/06/16

**Microsoft 专用**

生成从第一个参数的低 64 位中提取指定位的 `extrq` 指令。

## 语法

```
C  
  
__m128i _mm_extract_si64(  
    __m128i Source,  
    __m128i Descriptor  
);  
__m128i _mm_extracti_si64(  
    __m128i Source,  
    int Length,  
    int Index  
);
```

## 参数

*Source*

[in] 一个 128 位字段，其输入数据位于较低的 64 位。

*描述符*

[in] 描述要提取的位字段的 128 位字段。

*Length*

[in] 一个整数，指定要提取的字段的长度。

*Index*

[in] 一个整数，指定要提取的索引的长度。

## 返回值

一个 128 位字段，其中提取的字段位于其最小有效位中。

## 要求

Intrinsic	体系结构
_mm_extract_si64	SSE4a
_mm_extracti_si64	SSE4a

头文件<intrin.h>

## 注解

这些内部函数生成从源中提取位的 `extrq` 指令。有两个版本：`_mm_extracti_si64` 是即时版本，`_mm_extract_si64` 是非即时版本。每个版本从源中提取一个位字段，该字段由其长度和最小有效位的索引定义。长度和索引的值取 mod 64，因此 -1 和 127 都解释为 63。如果（缩小的）索引和（减少的）字段长度之和大于 64，则结果未定义。字段长度的零值被解释为 64。如果字段长度和位索引均为零，则提取源的位 63:0。如果字段长度为零，但位索引不为零，则结果未定义。

在调用 `_mm_extract_si64` 中，描述符包含位为 13:8 的索引，以及要以位 5:0 提取的数据的字段长度。

如果使用编译器无法确定为整数常量的参数调用 `_mm_extracti_si64`，则编译器将生成代码以将这些值打包到 XMM 寄存器（描述符）中并调用 `_mm_extract_si64`。

若要确定 `extrq` 指令的硬件支持，请调用具有 `InfoType=0x80000001` 的 `__cpuid` 内部函数，并检查 `CPUInfo[2] (ECX)` 的第 6 位。如果支持指令，则此位为 1，否则为 0。如果在不支持 `extrq` 指令的硬件上运行使用内部函数的代码，则结果是不可预测的。

## 示例

C++

```
// 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] =      0xfedcba987654321011;
```

```
descriptor.ui64[0] = 0x0000000000000b1b1l;

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) & 0xffffffff;

cout << hex << "result1 = 0x" << result1.ui64[0] << endl;
cout << "result2 = 0x" << result2.ui64[0] << endl;
cout << "result3 = 0x" << result3.ui64[0] << endl;
}
```

#### Output

```
result1 = 0x30eca86
result2 = 0x30eca86
result3 = 0x30eca86
```

## 结束 Microsoft 专用

超威半导体公司保留 2007 部分版权。 经超威半导体公司许可转载

## 另请参阅

[\\_mm\\_insert\\_si64, \\_mm\\_inserti\\_si64](#)

[编译器内部函数](#)

# `_mm_insert_si64`, `_mm_inserti_si64`

项目 • 2023/06/16

## Microsoft 专用

生成 `insertq` 指令，以将位从其第二个操作数插入到其第一个操作数中。

## 语法

C

```
__m128i _mm_insert_si64(
    __m128i Source1,
    __m128i Source2
);
__m128i _mm_inserti_si64(
    __m128i Source1,
    __m128i Source2
    int Length,
    int Index
);
```

## 参数

`Source1`

[in] 一个 128 位的字段，输入数据在其低 64 位，其中将插入一个字段。

`Source2`

[in] 一个 128 位字段，其中包含要在其低位中插入的数据。对于 `_mm_insert_si64`，还包含其高位的字段描述符。

`Length`

[in] 一个整数常量，指定要插入的字段的长度。

`Index`

[in] 一个整数常量，指定要插入数据的字段的最低有效位的索引。

## 返回值

一个 128 位字段，其较低的 64 位包含原始低 64 位 `Source1`，指定的位字段替换为 `Source2` 的低位。返回值的高 64 位未定义。

# 要求

Intrinsic	体系结构
_mm_insert_si64	SSE4a
_mm_inserti_si64	SSE4a

头文件<intrin.h>

## 注解

这些内部函数生成 `insertq` 指令，用于将来自 Source2 的位插入到 Source1 中。有两个版本：`_mm_inserti_si64` 是即时版本，`_mm_insert_si64` 是非即时版本。每个版本从 Source2 中提取给定长度的位字段，并将其插入 Source1 中。提取的位是 Source2 的最低有效位。将插入这些位的字段 Source1 由其最低有效位的长度和索引定义。长度和索引的值取 mod 64，因此 -1 和 127 都解释为 63。如果（缩小的）位索引和（减少的）字段长度之和大于 64，则结果未定义。字段长度的零值被解释为 64。如果字段长度和位索引均为零，则将 Source2 的位 63:0 插入到 Source1 中。如果字段长度为零，但位索引不为零，则结果未定义。

在调用 `_mm_insert_si64` 时，字段长度包含在 Source2 的 77:72 位中，索引包含在位 69:64 中。

如果使用编译器无法确定为整数常量的参数调用 `_mm_inserti_si64`，则编译器将生成代码以将这些值打包到 XMM 寄存器中并调用 `_mm_insert_si64`。

若要确定 `insertq` 指令的硬件支持，请调用具有 `InfoType=0x80000001` 的 `__cpuid` 内部函数，并检查 `CPUInfo[2] (ECX)` 的第 6 位。如果支持该指令，则此位为 1，否则为 0。如果在不支持 `insertq` 指令的硬件上运行使用内部函数的代码，则结果是不可预测的。

## 示例

C++

```
// 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] = 0xfffffffffffff11;
    source2.ui64[0] = 0xfedcba987654321011;
    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;
}
```

#### Output

```
result1 = 0xffffffffffff3210fff
result2 = 0xffffffffffff3210fff
result3 = 0xffffffffffff3210fff
```

## 结束 Microsoft 专用

超威半导体公司保留 2007 部分版权。 经超威半导体公司许可转载

## 另请参阅

[\\_mm\\_extract\\_si64、\\_mm\\_extracti\\_si64](#)

[编译器内部函数](#)

# \_mm\_stream\_sd

项目 • 2023/10/12

**Microsoft 专用**

将 64 位数据写入内存位置，而不会污染缓存。

## 语法

C

```
void _mm_stream_sd(  
    double * Dest,  
    __m128d Source  
) ;
```

## 参数

*Dest*

[out] 指向将写入源数据的位置的指针。

*Source*

[in] 一个 128 位值，其中包含要写入其底部 64 位的 `double` 值。

## 返回值

无。

## 要求

Intrinsic	体系结构
<code>_mm_stream_sd</code>	SSE4a

头文件<intrin.h>

## 备注

内部函数生成 `movntsd` 指令。若要确定此指令的硬件支持，请调用 `InfoType=0x80000001` 的 `_cpuid` 内部函数，并检查 `CPUInfo[2] (ECX)` 的第 6 位。如果硬件支持该指令，则该位为 1，否则为 0。

如果在不支持 `movntsd` 指令的硬件上运行使用 `_mm_stream_sd` 内部函数的代码，则结果是不可预测的。

## 示例

C++

```
// 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;
}
```

Output

```
d[0] = -1, d[1] = 1
```

## 结束 Microsoft 专用

超威半导体公司保留 2007 部分版权。经超威半导体公司许可转载

## 另请参阅

[\\_mm\\_stream\\_ss](#)

[\\_mm\\_store\\_sd](#)

[\\_mm\\_sfence](#)

[编译器内部函数](#)

# \_mm\_stream\_si64x

项目 • 2023/06/16

Microsoft 专用

生成 MOVNTI 指令。 将源中的数据写入目标指定的内存位置，而不要污染缓存。

## 语法

C

```
void _mm_stream_si64x(
    __int64 * Destination,
    __int64 Source
);
```

## 参数

目标

[out] 指向源数据的写入位置的指针。

*Source*

[in] 要写入的数据。

## 要求

Intrinsic	体系结构
_mm_stream_si64x	X64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

## 示例

C

```
// _mm_stream_si64x.c
// processor: x64

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

#pragma intrinsic(_mm_stream_si64x)

int main()
{
    __int64 val = 0xFFFFFFFFFFFFFFI64;
    __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]);
}
```

Output

```
0 ffffffff 0 0
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# **\_mm\_stream\_ss**

项目 • 2023/10/12

**Microsoft 专用**

将 32 位数据写入内存位置，而不会污染缓存。

## 语法

C

```
void _mm_stream_ss(  
    float * Destination,  
    __m128 Source  
) ;
```

## 参数

### 目标

[out] 指向写入源数据的位置的指针。

### Source

[in] 一个 128 位数字，其中包含要在其最后 32 位中写入的 `float` 值。

## 返回值

无。

## 要求

Intrinsic	体系结构
<code>_mm_stream_ss</code>	SSE4a

头文件<intrin.h>

## 备注

内部函数生成 `movntss` 指令。若要确定此指令的硬件支持，请调用 `InfoType=0x80000001` 的 `_cpuid` 内部函数，并检查 `CPUInfo[2] (ECX)` 的第 6 位。如果支持该指令，则此位为 1，否则为 0。

如果在不支持 `movntss` 指令的硬件上运行使用 `_mm_stream_ss` 内部函数的代码，则结果是不可预测的。

## 示例

C++

```
// 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[2] = " << f[2] << ", f[3] = " << f[3] << endl;
}
```

Output

```
f[0] = -1, f[1] = -2
f[2] = -3, f[3] = 3
```

## 结束 Microsoft 专用

超威半导体公司保留 2007 部分版权。经超威半导体公司许可转载

## 另请参阅

[\\_mm\\_stream\\_sd](#)  
[\\_mm\\_stream\\_ps ↗](#)  
[\\_mm\\_store\\_ss ↗](#)  
[\\_mm\\_sfence ↗](#)  
编译器内部函数

# \_\_movsb

项目 • 2023/10/12

**Microsoft 专用**

生成移动字符串 (`rep movsb`) 指令。

## 语法

C

```
void __movsb(
    unsigned char* Destination,
    unsigned const char* Source,
    size_t Count
);
```

## 参数

**目标**

[out] 指向副本目标的指针。

*Source*

[in] 指向副本源的指针。

**计数**

[in] 要复制的字节数。

## 要求

Intrinsic	体系结构
<code>__movsb</code>	x86、x64

头文件<intrin.h>

## 备注

结果是，`Source` 指向的第一个 `Count` 字节被复制到 `Destination` 字符串中。

此例程仅可用作内部函数。

# 示例

C++

```
// 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);
}
```

Output

```
A big black dog. A big black dog.
```

结束 Microsoft 专用

**另请参阅**

[编译器内部函数](#)

# \_\_movsd

项目 • 2023/10/12

**Microsoft 专用**

生成移动字符串 (`rep movsd`) 指令。

## 语法

C

```
void __movsd(
    unsigned long* Destination,
    unsigned long* Source,
    size_t Count
);
```

## 参数

**目标**

[out] 操作的目标。

*Source*

[in] 操作的源。

**计数**

[in] 要复制的双字数。

## 要求

Intrinsic	体系结构
<code>__movsd</code>	x86、x64

头文件<intrin.h>

## 备注

结果是，`Source` 指向的第一个 `Count` 双字将复制到 `Destination` 字符串。

此例程仅可用作内部函数。

# 示例

C++

```
// 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");
}
```

Output

```
950 850 750 650 550 450 350 250 150 50
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# **\_\_movsq**

项目 • 2023/10/12

## Microsoft 专用

生成重复的移动字符串 (`rep movsq`) 指令。

## 语法

C

```
void __movsq(
    unsigned long long* Destination,
    unsigned long long const* Source,
    size_t Count
);
```

## 参数

### 目标

[out] 操作的目标。

### Source

[in] 操作的源。

### 计数

[in] 要复制的四字数。

## 要求

Intrinsic	体系结构
<code>__movsq</code>	x64

头文件<intrin.h>

## 备注

结果是“源”指向的第一个“计数”四字将复制到“目标”字符串。

此例程仅可用作内部函数。

# 示例

C++

```
// 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");
}
```

Output

```
950 850 750 650 550 450 350 250 150 50
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# \_\_movsw

项目 • 2023/10/12

**Microsoft 专用**

生成移动字符串 (`rep movsw`) 指令。

## 语法

C

```
void __movsw(
    unsigned short* Destination,
    unsigned short* Source,
    size_t Count
);
```

## 参数

**目标**

[out] 操作的目标。

*Source*

[in] 操作的源。

**计数**

[in] 要复制的字数。

## 要求

Intrinsic	体系结构
<code>__movsw</code>	x86、x64

头文件<intrin.h>

## 备注

结果是，源指向的第一个计数单词将复制到目标字符串。

此例程仅可用作内部函数。

# 示例

C++

```
// 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");
}
```

Output

```
0 1 2 3 4 5 6 7 8 9
```

结束 Microsoft 专用

另请参阅

[编译器内部函数](#)

# \_mul128

项目 • 2023/10/12

## Microsoft 专用

乘以作为前两个自变量传入的两个 64 位整数，将产品的高 64 位置于由 `HighProduct` 指向的 64 位整数，返回产品的低 64 位。

## 语法

C

```
__int64 __mul128(  
    __int64 Multiplier,  
    __int64 Multiplicand,  
    __int64 *HighProduct  
) ;
```

## 参数

“乘数”

[in] 要相乘的第一个 64 位整数。

Multiplicand

[in] 要相乘的第二个 64 位整数。

HighProduct

[out] 乘积的高 64 位。

## 返回值

产品的低 64 位。

## 要求

Intrinsic	体系结构
<code>_mul128</code>	x64

头文件<intrin.h>

## 示例

```
C

// mul128.c
// processor: x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_mul128)

int main()
{
    __int64 a = 0xffffffffffffffffI64;
    __int64 b = 0xf0000000I64;
    __int64 c, d;

    d = _mul128(a, b, &c);

    printf_s("%#I64x * %#I64x = %#I64x%I64x\n", a, b, c, d);
}
```

### Output

```
0xffffffffffffffff * 0xf0000000 = 0xeffffffffffffff10000000
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# \_\_mulh

项目 • 2024/08/04

Microsoft 专用

返回两个 64 位无符号整数的积的高 64 位。

## 语法

C

```
_int64 __mulh(  
    _int64 a,  
    _int64 b  
)
```

## 参数

a

[输入] 要相乘的第一个数字。

b

[输入] 要相乘的第二个数字。

## 返回值

128 位乘法运算结果的高 64 位。

## 要求

 展开表

Intrinsic	体系结构
<code>__mulh</code>	x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

## 示例

C++

```
// mulh.cpp
// processor: x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic (__mulh)

int main()
{
    __int64 a = 0xfffffffffffffI64;
    __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);
}
```

Output

```
0xfffffffffffff * 0xf0000000 = 0xeffffffff10000000
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

## 反馈

此页面是否有帮助?

是

否

[提供产品反馈](#) | 在 Microsoft Q&A 获取帮助

# **\_\_noop**

项目 • 2024/07/05

**特定于 Microsoft** `__noop` 的内部函数指定应忽略函数。分析了参数列表，但没有为参数生成任何代码。编译器认为引用参数是出于编译器警告 C4100 和类似分析的目的。

`__noop` 内在函数旨在用于采用数目可变的自变量的全局调试函数。

编译器在编译时将 `__noop` 内部函数转换为 0。

## 示例

下面的代码演示如何使用 `__noop`。

C++

```
// 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");
}
```

## 另请参阅

[编译器内部函数](#)

[关键字](#)

## 反馈

此页面是否有帮助?

是

否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# **\_\_nop**

项目 • 2023/06/16

## **Microsoft 专用**

生成不执行任何操作的特定于平台的计算机代码。

## **语法**

C

```
void __nop();
```

## **要求**

Intrinsic	体系结构
<code>__nop</code>	x86、ARM、x64、ARM64

头文件<intrin.h>

## **结束 Microsoft 专用**

## **注解**

`__nop` 函数等同于 `NOP` 计算机指令。有关 x86 和 x64 的详细信息，请在 [Intel Corporation](#) 站点上搜索文档“体系结构软件开发人员手册第 2 卷：指令集参考”。

## **另请参阅**

[编译器内部函数](#)

[\\_noop](#)

# **\_\_outbyte**

项目 • 2024/08/04

**Microsoft 专用**

生成 `out` 指令，将 `Data` 指定的 1 字节发送到 `Port` 指定的 I/O 端口。

## 语法

C

```
void __outbyte(
    unsigned short Port,
    unsigned char Data
);
```

## 参数

**端口**

[in] 要将数据发送到的端口。

**数据**

[in] 要从指定端口发送出去的字节。

## 要求

[] 展开表

Intrinsic	体系结构
<code>__outbyte</code>	x86、x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

**结束 Microsoft 专用**

# 另请参阅

[编译器内部函数](#)

---

## 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | [在 Microsoft Q&A 获取帮助](#)

# `__outbytestring`

项目 · 2024/08/04

## Microsoft 专用

生成 `rep outsb` 指令，该指令将由 `Buffer` 指向的数据的第一个 `Count` 字节发送到由 `Port` 指定的端口。

## 语法

C

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

## 参数

### 端口

[in] 要将数据发送到的端口。

### Buffer

[in] 要从指定端口发送出去的数据。

### 计数

[in] 要发送的字节和数据的数量。

## 要求

[+] 展开表

Intrinsic	体系结构
<code>__outbytestring</code>	x86、x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

## 反馈

此页面是否有帮助？

 是

 否

[提供产品反馈](#) | [在 Microsoft Q&A 获得帮助](#)

# **\_\_outdword**

项目 • 2023/06/16

**Microsoft 专用**

生成 `out` 指令以将双字 Data 发送到端口 Port。

## 语法

C

```
void __outdword(  
    unsigned short Port,  
    unsigned long Data  
)
```

## 参数

端口

[in] 要将数据发送到的端口。

数据

[in] 要发送的双字。

## 要求

Intrinsic	体系结构
<code>__outdword</code>	x86、x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

结束 Microsoft 专用

## 另请参阅

## 编译器内部函数

# `__outdwordstring`

项目 • 2024/08/04

## Microsoft 专用

生成 `rep outsd` 指令，该指令将从 `Buffer` 开始的 `Count` 个双字发送到 `Port` 指定的 I/O 端口。

## 语法

C

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

## 参数

### 端口

[in] 要将数据发送到的端口。

### Buffer

[in] 要从指定端口发送出去的数据的指针。

### 计数

[in] 要复制的双字数。

## 要求

 展开表

Intrinsic	体系结构
<code>__outdwordstring</code>	x86、x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

## 反馈

此页面是否有帮助？

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_\_outword

项目 • 2024/08/04

Microsoft 专用

生成 `out` 指令，该指令将“数据”一词发送到端口指定的 I/O 端口。

## 语法

C

```
void __outword(
    unsigned short Port,
    unsigned short Data
);
```

## 参数

端口

[in] 要将数据发送到的端口。

数据

[in] 要发送的数据。

## 要求

[] 展开表

Intrinsic	体系结构
<code>__outword</code>	x86、x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

结束 Microsoft 专用

# 另请参阅

[编译器内部函数](#)

---

## 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_\_outwordstring

项目 • 2024/08/04

## Microsoft 专用

生成 `rep outsw` 指令，该指令从 Port 指定的 I/O 端口发送出从 Buffer 开始的 Count 个单词。

## 语法

C

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

## 参数

### 端口

[in] 要将数据发送到的端口。

### Buffer

[in] 要从指定端口发送出去的数据的指针。

### 计数

[in] 要发送的字数。

## 要求

[] 展开表

Intrinsic	体系结构
<code>__outwordstring</code>	x86、x64

头文件<intrin.h>

## 注解

此例程仅可用作内部函数。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

## 反馈

此页面是否有帮助？

 是

 否

[提供产品反馈](#) | [在 Microsoft Q&A 获得帮助](#)

# `__popcnt16`, `__popcnt`, `__popcnt64`

项目 • 2024/08/04

## Microsoft 专用

计算 16、32 或 64 位无符号整数中 1 位的数量（填充计数）。

## 语法

```
C

unsigned short __popcnt16(
    unsigned short value
);
unsigned int __popcnt(
    unsigned int value
);
unsigned __int64 __popcnt64(
    unsigned __int64 value
);
```

## 参数

*value*

[in] 我们想要其填充计数的 16、32 或 64 位无符号整数。

## 返回值

*value* 参数中的 1 位的数量。

## 要求

  展开表

Intrinsic	体系结构
<code>__popcnt16</code>	高级位操作
<code>__popcnt</code>	高级位操作
<code>__popcnt64</code>	64 位模式下的高级位操作。

头文件<intrin.h>

## 备注

每个内部函数都会生成 `popcnt` 指令。在 32 位模式下，没有 64 位通用寄存器，因此不支持 64 位 `popcnt`。

若要确定 `popcnt` 指令的硬件支持，请调用具有 `InfoType=0x00000001` 的 `_cpuid` 内部函数，并检查 `CPUInfo[2] (ECX)` 的第 23 位。如果支持该指令，则此位为 1，否则为 0。如果在不支持 `popcnt` 指令的硬件上运行使用内部函数的代码，则结果是不可预测的。

## 示例

C++

```
#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;
    }
}
```

Output

```
__popcnt16(0x0) = 0
__popcnt16(0xff) = 8
__popcnt16(0xffff) = 16
__popcnt(0x0) = 0
__popcnt(0xff) = 8
__popcnt(0xffff) = 16
__popcnt(0xffffffff) = 32
```

结束 Microsoft 专用

超威半导体公司保留 2007 部分版权。 经超威半导体公司许可转载

## 另请参阅

[编译器内部函数](#)

---

## 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# `__rdtsc`

项目 • 2024/01/23

## Microsoft 专用

生成 `rdtsc` 指令，该指令将返回处理器时间戳。处理器时间戳记录自上次重置以来的时钟周期数。

## 语法

C

```
unsigned __int64 __rdtsc();
```

## 返回值

表示滴答计数的 64 位无符号整数。

## 要求

展开表

Intrinsic	体系结构
<code>__rdtsc</code>	x86、x64

头文件<intrin.h>

## 备注

此例程仅可用作内部函数。

在后代硬件中，对 TSC 值的解释与早期版本的 x64 不同。有关详细信息，请参阅硬件手册。

## 示例

C++

```
// 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);
}
```

Output

```
3363423610155519 ticks
```

结束 Microsoft 专用

## 另请参阅

[\\_rdtscp](#)

[编译器内部函数](#)

# \_\_rdtscp

项目 • 2023/10/18

## Microsoft 专用

生成 `rdtscp` 指令，将 `TSC_AUX[31:0]` 写入内存，并返回 64 位时间戳计数器 (`TSC`) 结果。

## 语法

C

```
unsigned __int64 __rdtscp(  
    unsigned int * AUX  
)
```

## 参数

AUX

[out] 指向包含计算机特定寄存器 `TSC_AUX[31:0]` 内容的位置的指针。

## 返回值

64 位无符号整数滴答计数。

## 要求

Intrinsic	体系结构
<code>__rdtscp</code>	x86、x64

头文件<intrin.h>

## 备注

`__rdtscp` 内部函数生成 `rdtscp` 指令。 若要确定此指令的硬件支持，请调用 `InfoType=0x80000001` 的 `_cpuid` 内部函数，并检查 `CPUInfo[3] (EDX)` 的第 27 位。 如果支持该指令，则此位为 1，否则为 0。 如果在不支持 `rdtscp` 指令的硬件上运行使用内部函数的代码，则结果是不可预测的。

此指令将等到所有以前的指令都已执行，并且所有以前的加载都全局可见。但是，它不是序列化指令。有关详细信息，请参阅 Intel 和 AMD 手册。

TSC\_AUX[31:0] 中值的含义取决于操作系统。

## 示例

C++

```
#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);
}
```

Output

```
3363423610155519 ticks
TSC_AUX was 0
```

结束 Microsoft 专用

## 另请参阅

[\\_rdtsc](#)

[编译器内部函数](#)

# \_ReadBarrier

项目 • 2023/06/16

## Microsoft 专用

限制可重新排列调用点上的内存访问操作的编译器优化。

### ⊗ 注意

已全部弃用且不应使用 `_ReadBarrier`、`_WriteBarrier` 和 `_ReadWriteBarrier` 编译器内部函数和 `MemoryBarrier` 宏。对于线程间的通信，请使用 C++ 标准库中定义的机制，例如 `atomic_thread_fence` 和 `std::atomic<T>`。对于硬件访问，请将 `/volatile:iso` 编译器选项与 `volatile` 关键字一起使用。

## 语法

C

```
void _ReadBarrier(void);
```

## 要求

Intrinsic	体系结构
<code>_ReadBarrier</code>	x86、x64

头文件<intrin.h>

## 注解

`_ReadBarrier` 内部函数将限制可删除或重新排列调用点上的内存访问操作的编译器优化。

结束 Microsoft 专用

另请参阅

编译器内部函数

关键字

# \_\_readcr0

项目 • 2024/08/04

Microsoft 专用

读取 CR0 寄存器并返回其值。

## 语法

C

```
unsigned long __readcr0(void); /* X86 */  
unsigned __int64 __readcr0(void); /* X64 */
```

## 返回值

CR0 寄存器中的值。

## 要求

 展开表

Intrinsic	体系结构
<code>__readcr0</code>	x86、x64

头文件<intrin.h>

## 备注

内部仅在内核模式下可用，且例程仅可用作内部。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_\_readcr2

项目 • 2024/08/04

Microsoft 专用

读取 CR2 寄存器并返回其值。

## 语法

C

```
unsigned __int64 __readcr2(void);
```

## 返回值

CR2 寄存器中的值。

## 要求

[] 展开表

Intrinsic	体系结构
<code>__readcr2</code>	x86、x64

头文件<intrin.h>

## 备注

内部仅在内核模式下可用，且例程仅可用作内部。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_\_readcr3

项目 • 2024/08/04

Microsoft 专用

读取 CR3 寄存器并返回其值。

## 语法

C

```
unsigned __int64 __readcr3(void);
```

## 返回值

CR3 寄存器中的值。

## 要求

[] 展开表

Intrinsic	体系结构
<code>__readcr3</code>	x86、x64

头文件<intrin.h>

## 备注

内部仅在内核模式下可用，且例程仅可用作内部。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_\_readcr4

项目 • 2024/08/04

Microsoft 专用

读取 CR4 寄存器并返回其值。

## 语法

C

```
unsigned __int64 __readcr4(void);
```

## 返回值

CR4 寄存器中的值。

## 要求

[] 展开表

Intrinsic	体系结构
<code>__readcr4</code>	x86、x64

头文件<intrin.h>

## 备注

内部仅在内核模式下可用，且例程仅可用作内部。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_\_readcr8

项目 • 2024/08/04

Microsoft 专用

读取 CR8 寄存器并返回其值。

## 语法

C

```
unsigned __int64 __readcr8(void);
```

## 返回值

CR8 寄存器中的值。

## 要求

展开表

Intrinsic	体系结构
<code>__readcr8</code>	x64

头文件<intrin.h>

## 备注

内部仅在内核模式下可用，且例程仅可用作内部。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# \_\_readdr

项目 • 2023/06/16

Microsoft 专用

读取指定调试寄存器的值。

## 语法

C

```
unsigned      __readdr(unsigned int DebugRegister); /* x86 */
unsigned __int64 __readdr(unsigned int DebugRegister); /* x64 */
```

## 参数

DebugRegister

[in] 一个从 0 到 7 的常数，用于标识调试寄存器。

## 返回值

指定调试寄存器的值。

## 注解

这些内部函数仅在内核模式下可用，例程只能用作内部函数。

## 要求

Intrinsic	体系结构
<code>__readdr</code>	x86、x64

头文件<intrin.h>

结束 Microsoft 专用

## 另请参阅

编译器内部函数

\_readeflags

# **\_\_readeflags**

项目 • 2023/06/16

**Microsoft 专用**

读取程序状态和控件 (EFLAGS) 寄存器。

## 语法

C

```
unsigned     int __readeflags(void); /* x86 */
unsigned __int64 __readeflags(void); /* x64 */
```

## 返回值

EFLAGS 寄存器的值。 返回值在 32 位平台上的长度为 32 位，在 64 位平台上的长度为 64 位。

## 注解

这些例程只能用作内部函数。

## 要求

Intrinsic	体系结构
<code>__readeflags</code>	x86、x64

头文件<intrin.h>

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[\\_\\_writeeflags](#)

# **\_\_readfsbyte、 \_\_readfsdword、 \_\_readfsqword、 \_\_readfsword**

项目 • 2024/08/04

Microsoft 专用

从相对于 FS 段开头的偏移量指定的位置读取内存。

## 语法

C

```
unsigned char __readfsbyte(  
    unsigned long Offset  
);  
unsigned short __readfsword(  
    unsigned long Offset  
);  
unsigned long __readfsdword(  
    unsigned long Offset  
);  
unsigned __int64 __readfsqword(  
    unsigned long Offset  
);
```

## 参数

*Offset*

[in] 从 FS 的开头开始读取的偏移量。

## 返回值

位置 FS:[*Offset*] 处的字节、字、双字或四字（由调用的函数名称指示）的内存内容。

## 要求

 展开表

Intrinsic	体系结构
__readfsbyte	x86

Intrinsic	体系结构
__readfsdword	x86
__readfsqword	x86
__readfsword	x86

头文件<intrin.h>

## 备注

这些例程只能用作内部函数。

结束 Microsoft 专用

## 另请参阅

[\\_writefsbyte, \\_writefsdword, \\_writefsqword, \\_writefsword](#)  
编译器内部函数

---

## 反馈

此页面是否有帮助?

 是  否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助

# **\_\_readgsbyte、 \_\_readgsdword、 \_\_readgsqword、 \_\_readgsword**

项目 • 2023/06/16

**Microsoft 专用**

从相对于 GS 段开头的偏移量指定的位置读取内存。

## 语法

```
C

unsigned char __readgsbyte(
    unsigned long Offset
);
unsigned short __readgsword(
    unsigned long Offset
);
unsigned long __readgsdword(
    unsigned long Offset
);
unsigned __int64 __readgsqword(
    unsigned long Offset
);
```

## 参数

*Offset*

[in] 从 `GS` 的开头开始读取的偏移量。

## 返回值

位置 `GS:[Offset]` 处的字节、字、双字或四字（由调用的函数名称指示）的内存内容。

## 要求

Intrinsic	体系结构
<code>__readgsbyte</code>	X64
<code>__readgsdword</code>	X64

Intrinsic	体系结构
<code>__readgsqword</code>	X64
<code>__readgsword</code>	X64

头文件<intrin.h>

## 注解

这些例程只能用作内部函数。

结束 Microsoft 专用

## 另请参阅

[\\_writegsbyte, \\_writegsdword, \\_writegsqword, \\_writegsword](#)  
编译器内部函数

# \_\_readmsr

项目 • 2023/10/12

Microsoft 专用

生成 `rdmsr` 指令来读取由 `register` 指定的特定于模型的寄存器并返回其值。

## 语法

C

```
__int64 __readmsr(  
    int register  
) ;
```

## 参数

*register*

[in] 要读取的特定于模型的寄存器。

## 返回值

指定寄存器中的值。

## 要求

Intrinsic	体系结构
<code>__readmsr</code>	x86、x64

头文件<intrin.h>

## 备注

此函数仅在内核模式下可用，且例程仅可用作内部函数。

有关详细信息，请参阅 AMD 文档。

结束 Microsoft 专用

# 另请参阅

[编译器内部函数](#)

# **\_\_readpmc**

项目 • 2023/10/17

**Microsoft 专用**

生成 `rdpmc` 指令，该指令读取计数器指定的性能监视计数器。

## 语法

C

```
unsigned __int64 __readpmc(  
    unsigned long counter  
) ;
```

## 参数

计数器

[in] 要读取的性能计数器。

## 返回值

指定的性能计数器值。

## 要求

Intrinsic	体系结构
<code>__readpmc</code>	x86、x64

头文件<intrin.h>

## 备注

此内部函数只在内核模式下可用，且例程只能用作内部函数。

结束 Microsoft 专用

## 另请参阅

## 编译器内部函数

# \_ReadWriteBarrier

项目 • 2023/10/12

## Microsoft 专用

限制可重新排列调用点上的内存访问的编译器优化。

### ⊗ 注意

已全部弃用且不应使用 `_ReadBarrier`、`_WriteBarrier` 和 `_ReadWriteBarrier` 编译器内部函数和 `MemoryBarrier` 宏。对于线程间的通信，请使用 C++ 标准库中定义的机制，例如 `atomic_thread_fence` 和 `std::atomic<T>`。对于硬件访问，请将 `/volatile:iso` 编译器选项与 `volatile` 关键字一起使用。

## 语法

C

```
void _ReadWriteBarrier(void);
```

## 要求

Intrinsic	体系结构
<code>_ReadWriteBarrier</code>	x86、x64

头文件<intrin.h>

## 注解

`_ReadWriteBarrier` 内部函数将限制可删除或重新排列调用点上的内存访问的编译器优化。

结束 Microsoft 专用

## 另请参阅

[\\_ReadBarrier](#)

[\\_WriteBarrier](#)

[编译器内部函数](#)

[关键字](#)

# \_ReturnAddress

项目 • 2023/06/16

## Microsoft 专用

`_ReturnAddress` 内部函数提供调用函数中指令的地址，该函数将在控制权返回给调用者后执行。

生成以下程序并在调试器中逐步完成。在逐步执行程序时，请注意从 `_ReturnAddress` 中返回的地址。然后，在从使用 `_ReturnAddress` 的函数返回后立即打开[如何：使用反汇编窗口](#)，并注意要执行的下一条指令的地址与从 `_ReturnAddress` 返回的地址匹配。

内联等优化可能会影响返回地址。例如，如果下面的示例程序使用 `/Ob1` 编译，`inline_func` 便会内联到调用函数 `main` 中。因此，从 `inline_func` 和 `main` 对 `_ReturnAddress` 的调用将各自产生相同的值。

在使用 `/clr` 编译的程序中使用 `_ReturnAddress` 时，包含 `_ReturnAddress` 调用的函数将编译为本机函数。当按托管方式编译的函数调用包含 `_ReturnAddress` 的函数时，`_ReturnAddress` 可能无法按预期方式运行。

## 要求

头文件<intrin.h>

## 示例

C++

```
// 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;
}
```

结束 Microsoft 专用

## 另请参阅

[\\_AddressOfReturnAddress](#)

[编译器内部函数](#)

[关键字](#)

# \_rotl8、\_rotl16

项目 • 2023/10/12

Microsoft 专用

通过指定的位位置数将输入值旋转到最高有效位 (MSB) 左侧。

## 语法

C

```
unsigned char _rotl8(
    unsigned char value,
    unsigned char shift
);
unsigned short _rotl16(
    unsigned short value,
    unsigned char shift
);
```

## 参数

*value*

[in] 要旋转的值。

*shift*

[in] 要旋转的位数。

## 返回值

旋转的值。

## 要求

Intrinsic	体系结构
_rotl8	x86、ARM、x64、ARM64
_rotl16	x86、ARM、x64、ARM64

头文件<intrin.h>

# 备注

不像左移操作，当执行向左旋转时，离开高端的高序位将移动到最低有效位位置。

## 示例

C++

```
// 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));
}
```

Output

```
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
```

结束 Microsoft 专用

另请参阅

`_rotr8, _rotr16`

编译器内部函数

# \_rotr8、\_rotr16

项目 • 2023/10/12

Microsoft 专用

通过指定的位位置数将输入值向右旋转到最高有效位 (MSB)。

## 语法

C

```
unsigned char _rotr8(
    unsigned char value,
    unsigned char shift
);
unsigned short _rotr16(
    unsigned short value,
    unsigned char shift
);
```

## 参数

*value*

[in] 要旋转的值。

*shift*

[in] 要旋转的位数。

## 返回值

旋转的值。

## 要求

Intrinsic	体系结构
_rotr8	x86、ARM、x64、ARM64
_rotr16	x86、ARM、x64、ARM64

头文件<intrin.h>

# 备注

不像右位移操作，当执行向右旋转时，离开低端的低顺序位将移动到高顺序位位置。

## 示例

C++

```
// 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));
}
```

Output

```
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
```

结束 Microsoft 专用

另请参阅

`_rotl8, _rotl16`

编译器内部函数

# **\_\_segmentlimit**

项目 • 2023/10/12

**Microsoft 专用**

生成 `lsl` (负载段限制) 指令。

## 语法

C

```
unsigned long __segmentlimit(  
    unsigned long a  
) ;
```

## 参数

*a*

[in] 一个指定段选择器的常量。

## 返回值

如果段选择器有效且在当前权限级别可见，则为由 *a* 指定的段选择器的段限制。

## 要求

Intrinsic	体系结构
<code>__segmentlimit</code>	x86、x64

头文件<intrin.h>

## 备注

如果无法检索到段限制，此指令就会失败。在失败时，此指令会清除 ZF 标志，并且返回值未定义。

此例程仅可用作内部函数。

# 示例

C++

```
#include <stdio.h>

#ifndef _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%lx eflags =0x%lx\n", sl, eflags);
    sl = __segmentlimit(KGDT_R3_DATA + RPL_MASK);

    eflags = __readeflags();

    printf("After: segment limit =0x%lx eflags =0x%lx 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;
}
```

Output

```
Before: segment limit =0xbaadbabe eflags =0x0
After: segment limit =0xffffffff eflags =0x256 eflags.zf = set
Success!
sl was changed
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# shiftleft128

项目 • 2023/10/12

## Microsoft 专用

将 128 位数量（表示为两个 64 位数量 `LowPart` 和 `HighPart`）按 `Shift` 指定的位数左移，返回高 64 位结果。

## 语法

C

```
unsigned __int64 __shiftleft128(  
    unsigned __int64 LowPart,  
    unsigned __int64 HighPart,  
    unsigned char Shift  
) ;
```

## 参数

`LowPart`

[in] 要位移的 128 位数量的低 64 位。

`HighPart`

[in] 要位移的 128 位数量的高 64 位。

`Shift`

[in] 要位移的位数。

## 返回值

高 64 位的结果。

## 要求

Intrinsic	体系结构
<code>__shiftleft128</code>	x64

头文件<intrin.h>

# 备注

Shift 值始终是模 64，如果调用 `__shiftleft128(1, 0, 64)`，该函数将向左位移低部分 0 位并返回高部分的 0 而不是另外预期的 1。

## 示例

C

```
// 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);
}
```

Output

```
0x1000000000000001 << 1 = 0x2000000000000002
0x1000000000000001 >> 1 = 0x0800000000000000
```

结束 Microsoft 专用

另请参阅

`_shiftright128`

编译器内部函数

# \_shiftright128

项目 • 2023/10/12

## Microsoft 专用

将 128 位数量（表示为两个 64 位数量 `LowPart` 和 `HighPart`）按 `Shift` 指定的位数右移并返回低 64 位结果。

## 语法

C

```
unsigned __int64 __shiftright128(  
    unsigned __int64 LowPart,  
    unsigned __int64 HighPart,  
    unsigned char Shift  
) ;
```

## 参数

`LowPart`

[in] 要位移的 128 位数量的低 64 位。

`HighPart`

[in] 要位移的 128 位数量的高 64 位。

`Shift`

[in] 要位移的位数。

## 返回值

结果的低 64 位。

## 要求

Intrinsic	体系结构
<code>_shiftright128</code>	x64

头文件<intrin.h>

# 备注

`Shift` 值始终是模 64，如果调用 `_shiftright128(0, 1, 64)`，该函数将向右位移高部分 0 位并返回低部分的 0 而不是另外预期的 1。

## 示例

有关示例，请参阅 [\\_shiftleft128](#)。

结束 Microsoft 专用

## 另请参阅

[\\_shiftleft128](#)

[编译器内部函数](#)

# \_\_sidt

项目 • 2023/10/12

## Microsoft 专用

将中断描述符表寄存器 (IDTR) 的值保存在指定内存位置。

## 语法

C

```
void __sidt(void * Destination);
```

## 参数

### 目标

[in] 指向要保存 IDTR 的内存位置的指针。

## 要求

Intrinsic	体系结构
<code>__sidt</code>	x86、x64

头文件<intrin.h>

## 备注

`__sidt` 函数等同于 `SIDT` 计算机指令。有关详细信息，请在 [Intel Corporation](#) 站点上搜索文档“体系结构软件开发人员手册第 2 卷：指令集参考”。

## 结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[\\_lidt](#)

# \_stosb

项目 • 2023/10/12

Microsoft 专用

生成存储字符串指令 (`rep stosb`)。

## 语法

C

```
void __stosb(
    unsigned char* Destination,
    unsigned char Data,
    size_t Count
);
```

## 参数

目标

[out] 操作的目标。

数据

[in] 要存储的数据。

计数

[in] 要写入的字节块的长度。

## 要求

Intrinsic	体系结构
<code>__stosb</code>	x86、x64

头文件<intrin.h>

## 备注

结果是将字符“Data”写入到“Destination”字符串中“Count”字节块。

此例程仅可用作内部函数。

# 示例

```
C

// 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);
}
```

## Output

```
*****
*@@@*@@@*****
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# \_\_stosd

项目 • 2023/06/16

Microsoft 专用

生成存储字符串指令 (rep stosd)。

## 语法

C

```
void __stosd(
    unsigned long* Destination,
    unsigned long Data,
    size_t Count
);
```

## 参数

目标

[out] 操作目标。

数据

[in] 要存储的数据。

计数

[in] 要写入的双字块的长度。

## 要求

Intrinsic	体系结构
__stosd	x86、x64

头文件<intrin.h>

## 注解

结果是双字“数据”写入“目标”指向的内存位置上的“计数”双字块。

此例程仅可用作内部函数。

## 示例

```
C

// 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]);
}
```

### Output

```
0 99999 99999 0
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# **\_\_stosq**

项目 • 2023/10/12

**Microsoft 专用**

生成存储字符串指令 (`rep stosq`)。

## 语法

C

```
void __stosq(
    unsigned __int64* Destination,
    unsigned __int64 Data,
    size_t Count
);
```

## 参数

**目标**

[out] 操作的目标。

**数据**

[in] 要存储的数据。

**计数**

[in] 要写入的四字块的长度。

## 要求

Intrinsic	体系结构
<code>__stosq</code>	AMD64

头文件<intrin.h>

## 备注

结果是将四字“Data”写入到“Destination”字符串中“Count”四字块。

此例程仅可用作内部函数。

## 示例

```
C

// stosq.c
// processor: x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__stosq)

int main()
{
    unsigned __int64 val = 0xFFFFFFFFFFFFFFI64;
    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]);
}
```

### Output

```
0 ffffffff ffffffff 0
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# \_\_stosw

项目 • 2023/10/12

Microsoft 专用

生成存储字符串指令 (`rep stosw`)。

## 语法

C

```
void __stosw(
    unsigned short* Destination,
    unsigned short Data,
    size_t Count
);
```

## 参数

目标

[out] 操作的目标。

数据

[in] 要存储的数据。

计数

[in] 要写入的字块的长度。

## 要求

Intrinsic	体系结构
<code>__stosw</code>	x86、x64

头文件<intrin.h>

## 备注

结果是将字“Data”写入到“Destination”字符串中“Count”字块。

此例程仅可用作内部函数。

# 示例

```
C

// 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]);
}
```

## Output

```
0 128 128 0
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# **\_\_svm\_clgi**

项目 • 2023/06/16

**Microsoft 专用**

清除全局中断标志。

## 语法

C

```
void __svm_clgi( void );
```

## 备注

`__svm_clgi` 函数等同于 `CLGI` 计算机指令。 全局中断标志确定微处理器是否因 I/O 完成、硬件温度警报或调试异常等事件而忽略、推迟或处理中断。

此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [AMD 公司](#) 网站搜索“AMD64 体系结构程序员手册第 2 卷：系统编程”。

## 要求

Intrinsic	体系结构
<code>__svm_clgi</code>	x86、x64

头文件<intrin.h>

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[\\_\\_svm\\_stgi](#)

# `__svm_invlpga`

项目 • 2023/10/12

## Microsoft 专用

使计算机的转换旁观缓冲区中的地址映射条目失效。 参数指定要使失效的页面的虚拟地址和地址空间标识符。

## 语法

C

```
void __svm_invlpga(void *Vaddr, int as_id);
```

## 参数

Vaddr

[in] 要失效的页面的虚拟地址。

as\_id

[in] 要失效的页面的地址空间标识符 (ASID)。

## 备注

`__svm_invlpga` 函数等同于 `INVLPGA` 计算机指令。此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [AMD 公司](#) 网站上搜索“AMD64 体系结构程序员手册第 2 卷：系统编程”文档，文档编号为 24593，修订版 3.11。

## 要求

Intrinsic	体系结构
<code>__svm_invlpga</code>	x86、x64

头文件<intrin.h>

结束 Microsoft 专用

# 另请参阅

[编译器内部函数](#)

# **\_\_svm\_skinit**

项目 • 2023/06/16

## Microsoft 专用

启动可验证安全软件的加载，如虚拟机监视器。

## 语法

C

```
void __svm_skinit(  
    int block_address  
)
```

## 参数

block\_address

64K 字节安全加载程序块的 32 位物理地址 (SLB)。

## 注解

`__svm_skinit` 函数等同于 `SKINIT` 计算机指令。此函数属于某个安全系统，该系统使用处理器和可信平台模块 (TPM) 来验证和加载受信任软件（称为安全内核 (SK)）。虚拟机监视器是安全内核的一个示例。安全系统验证初始化过程中加载的程序组件。如果计算机是多处理器，它可以防止组件被中断、设备访问或其他程序篡改。

`block_address` 参数指定名为安全加载程序块 (SLB) 的 64K 内存块的物理地址。SLB 包含一个名为安全加载程序的程序。它为计算机建立操作环境，然后加载安全内核。

此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [AMD 公司](#) 网站搜索“AMD64 体系结构程序员手册第 2 卷：系统编程”。

## 要求

Intrinsic	体系结构
<code>__svm_skinit</code>	x86、x64

头文件<intrin.h>

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# **\_\_svm\_stgi**

项目 • 2023/06/16

## Microsoft 专用

设置全局中断标志。

## 语法

C

```
void __svm_stgi(void);
```

## 备注

`__svm_stgi` 函数等同于 `STGI` 计算机指令。 全局中断标志确定微处理器是否因 I/O 完成、硬件温度警报或调试异常等事件而忽略、推迟或处理中断。

此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [AMD 公司](#) 网站搜索“AMD64 体系结构程序员手册第 2 卷：系统编程”。

## 要求

Intrinsic	体系结构
<code>__svm_stgi</code>	x86、x64

头文件<intrin.h>

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[\\_\\_svm\\_clgi](#)

# **\_\_svm\_vmlload**

项目 • 2023/06/16

**Microsoft 专用**

从指定的虚拟机控制块 (VMCB) 加载处理器状态的子集。

## 语法

C

```
void __svm_vmlload(
    size_t VmcbPhysicalAddress
);
```

## 参数

VmcbPhysicalAddress

[in] VMCB 的物理地址。

## 注解

`__svm_vmlload` 函数等同于 `VMLOAD` 计算机指令。此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [AMD 公司](#) 网站上搜索“AMD64 体系结构程序员手册第 2 卷：系统编程”文档，文档编号为 24593，修订版 3.11。

## 要求

Intrinsic	体系结构
<code>__svm_vmlload</code>	x86、x64

头文件<intrin.h>

结束 Microsoft 专用

## 另请参阅

## 编译器内部函数

[\\_\\_svm\\_vmrn](#)

[\\_\\_svm\\_vmsave](#)

# **\_\_svm\_vmrunt**

项目 • 2023/06/16

**Microsoft 专用**

开始执行与指定的虚拟机控制块 (VMCB) 对应的虚拟机来宾代码。

## 语法

C

```
void __svm_vmrunt(  
    size_t VmcPhysicalAddress  
)
```

## 参数

VmcPhysicalAddress

[in] VMCB 的物理地址。

## 注解

`__svm_vmrunt` 函数在 VMCB 中使用最少的信息开始执行虚拟机来宾代码。如果需要更多信息来处理复杂的中断或切换到另一个来宾，请使用 `__svm_vmsave` 或 `__svm_vmload` 函数。

`__svm_vmrunt` 函数等同于 `VMRUN` 计算机指令。此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [AMD 公司](#) 网站上搜索“AMD64 体系结构程序员手册第 2 卷：系统编程”文档，文档编号为 24593，修订版 3.11 及更高版本。

## 要求

Intrinsic	体系结构
<code>__svm_vmrunt</code>	x86、x64

头文件 <intrin.h>

结束 Microsoft 专用

# 另请参阅

[编译器内部函数](#)

[\\_svm\\_vmsave](#)

[\\_svm\\_vmload](#)

# `__svm_vmsave`

项目 • 2023/06/16

Microsoft 专用

在指定的虚拟机控制块 (VMCB) 存储处理器状态的子集。

## 语法

C

```
void __svm_vmsave(  
    size_t VmcbPhysicalAddress  
) ;
```

## 参数

VmcbPhysicalAddress

[in] VMCB 的物理地址。

## 注解

`__svm_vmsave` 函数等同于 `VMSAVE` 计算机指令。此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [AMD 公司](#) 网站上搜索“AMD64 体系结构程序员手册第 2 卷：系统编程”文档，文档编号为 24593，修订版 3.11 及更高版本。

## 要求

Intrinsic	体系结构
<code>__svm_vmsave</code>	x86、x64

头文件<intrin.h>

结束 Microsoft 专用

## 另请参阅

## 编译器内部函数

[\\_svm\\_vmrunt](#)

[\\_svm\\_vmload](#)

# ud2

项目 • 2023/06/16

## Microsoft 专用

生成未定义的指令。

## 语法

C

```
void __ud2();
```

## 备注

如果执行未定义的指令，处理器将引发操作码无效的异常。

`__ud2` 函数等同于 `UD2` 计算机指令。有关详细信息，请在 [Intel Corporation](#) 站点上搜索文档“体系结构软件开发人员手册第 2 卷：指令集参考”。

## 要求

Intrinsic	体系结构
<code>__ud2</code>	x86、x64

头文件<intrin.h>

## 结束 Microsoft 专用

## 示例

以下示例执行未定义的指令，这样会引发异常。然后，异常处理程序会将返回代码从 0 更改为 1。

C++

```
// __ud2_intrinsic.cpp
#include <stdio.h>
#include <intrin.h>
#include <excpt.h>
```

```
// compile with /EHs

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;
}
```

#### Output

```
Before __ud2(). Return code = 0.
In the exception handler.
After __ud2(). Return code = 1.
```

## 另请参阅

[编译器内部函数](#)

# \_udiv128

项目 • 2023/10/13

`_udiv128` 内部函数将 128 位无符号整数除以 64 位无符号整数。返回值包含商，内部函数通过指针参数返回余数。`_udiv128` 是 Microsoft 特定的。

## 语法

C

```
unsigned __int64 _udiv128(
    unsigned __int64 highDividend,
    unsigned __int64 lowDividend,
    unsigned __int64 divisor,
    unsigned __int64 *remainder
);
```

## 参数

*highDividend*

[in] 被除数的高 64 位。

*lowDividend*

[in] 被除数的低 64 位。

*divisor*

[in] 要除以的 64 位整数。

*remainder*

[out] 余数的 64 位整数位。

## 返回值

商的 64 位。

## 备注

传递 *highDividend* 中 128 位被除数的高 64 位，以及 *lowDividend* 中的低 64 位。内部函数将此值除以除数。它将余数存储在余数所指向的 64 位无符号整数中，并返回 64 位的商。

`_udiv128` 内部函数从 Visual Studio 2019 RTM 开始可用。

## 要求

Intrinsic	体系结构	标头
<code>_udiv128</code>	x64	<immintrin.h>

## 另请参阅

[\\_div128](#)

[编译器内部函数](#)

# \_udiv64

项目 • 2023/06/16

`_udiv64` 内部函数将 64 位无符号整数除以 32 位无符号整数。 返回值包含商，内部函数通过指针参数返回余数。 `_udiv64` 是 Microsoft 特定的。

## 语法

C

```
unsigned int _udiv64(
    unsigned __int64 dividend,
    unsigned int divisor,
    unsigned int* remainder
);
```

## 参数

*dividend*

[in] 要除以的 64 位无符号整数。

*divisor*

[in] 要除以的 32 位无符号整数。

*remainder*

[out] 32 位无符号整数余数。

## 返回值

商的 32 位。

## 注解

`_udiv64` 内部函数将被除数除以除数。 它将余数存储在余数指向的 32 位无符号整数中，并返回商的 32 位。

`_udiv64` 内部函数从 Visual Studio 2019 RTM 开始可用。

## 要求

Intrinsic	体系结构	标头
_udiv64	x86、x64	<immintrin.h>

## 另请参阅

[\\_div64](#)

[编译器内部函数](#)

# **\_\_ull\_rshift**

项目 • 2023/10/12

**Microsoft 专用**

在 x64 上，将第一个参数指定的 64 位值右移第二个参数指定的位数。

## 语法

C

```
unsigned __int64 __ull_rshift(  
    unsigned __int64 mask,  
    int nBit  
) ;
```

## 参数

**掩码**

[in] 要右移的 64 位整数值。

**nBit**

[in] 要移位的位数，在 x86 上按 32 取模，在 x64 上按 64 取模。

## 返回值

移位 **nBit** 位的掩码。

## 要求

Intrinsic	体系结构
<b>__ull_rshift</b>	x86、x64

头文件<intrin.h>

## 注解

如果第二个参数在 x86 上大于 31 (x64 上为 63) , 则将该参数按 32 取模 (x64 上为 64) 以确定要移位的位数。名称中的 `ull` 指示 `unsigned long long (unsigned __int64)`。

## 示例

C++

```
// 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;
}
```

Output

```
1
```

结束 Microsoft 专用

## 另请参阅

[\\_ll\\_lshift](#)

[\\_ll\\_rshift](#)

[编译器内部函数](#)

# \_umul128

项目 • 2023/10/13

## Microsoft 专用

乘以作为前两个自变量传入的两个 64 位无符号整数，将产品的高 64 位置于由 `HighProduct` 指向的 64 位无符号整数，并返回产品的低 64 位。

## 语法

C

```
unsigned __int64 _umul128(  
    unsigned __int64 Multiplier,  
    unsigned __int64 Multiplicand,  
    unsigned __int64 *HighProduct  
) ;
```

## 参数

“乘数”

[in] 要相乘的第一个 64 位整数。

Multiplicand

[in] 要相乘的第二个 64 位整数。

HighProduct

[out] 乘积的高 64 位。

## 返回值

产品的低 64 位。

## 要求

Intrinsic	体系结构	标头
<code>_umul128</code>	x64	<intrin.h>

# 示例

```
C

// umul128.c
// processor: x64

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

#pragma intrinsic(_umul128)

int main()
{
    unsigned __int64 a = 0xffffffffffffffffI64;
    unsigned __int64 b = 0xf0000000I64;
    unsigned __int64 c, d;

    d = _umul128(a, b, &c);

    printf_s("%#I64x * %#I64x = %#I64x%I64x\n", a, b, c, d);
}
```

## Output

```
0xffffffffffffffff * 0xf0000000 = 0xefffffff10000000
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# **\_\_umulh**

项目 • 2023/10/12

**Microsoft 专用**

返回两个 64 位无符号整数的产品的高 64 位。

## 语法

C

```
unsigned __int64 __umulh(  
    unsigned __int64 a,  
    unsigned __int64 b  
) ;
```

## 参数

a

[输入] 要相乘的第一个数字。

b

[输入] 要相乘的第二个数字。

## 返回值

128 位乘法运算结果的高 64 位。

## 要求

Intrinsic	体系结构
<code>__umulh</code>	x64、ARM64

头文件<intrin.h>

## 备注

这些例程只能用作内部函数。

## 示例

C++

```
// 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%016I64x * 0x%016I64x = 0x%016I64x_%016I64x \n", i, j,
            result, k);
    return 0;
}
```

Output

```
0x0000000000000010 * 0xfedcba9876543210 =
0x000000000000000f_edcba98765432100
```

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# **\_\_vmx\_off**

项目 • 2023/06/16

## **Microsoft 专用**

停用处理器中的虚拟机扩展 (VMX) 操作。

## **语法**

C

```
void __vmx_off();
```

## **备注**

`__vmx_off` 函数等同于 `VMXOFF` 计算机指令。此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [Intel Corporation](#) 网站点搜索“针对 IA-32 Intel 体系结构的 Intel 虚拟化技术规范”文档，文档编号为 C97063-002。

## **要求**

Intrinsic	体系结构
<code>__vmx_off</code>	x86、x64

头文件<intrin.h>

结束 Microsoft 专用

## **另请参阅**

[编译器内部函数](#)

# **\_\_vmx\_on**

项目 • 2023/10/13

## Microsoft 专用

激活处理器中的虚拟机扩展 (VMX) 操作。

## 语法

C

```
unsigned char __vmx_on(  
    unsigned __int64 *VmxonRegionPhysicalAddress  
) ;
```

## 参数

*VmxonRegionPhysicalAddress*

[in] 指向 64 位 4KB 对齐物理地址（指向 VMXON 区域）的指针。

## 返回值

值	含义
0	操作成功。
1	操作失败，当前 VMCS 的 <code>VM-instruction error field</code> 中提供了扩展状态。
2	操作失败，无可用状态。

## 备注

`__vmx_on` 函数对应 `VMXON` 计算机指令。此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请参阅 [Intel 64 和 IA-32 体系结构开发人员手册](#) 中的“Intel 64 和 IA-32 体系结构软件开发人员手册第 3C 卷：系统编程指南第 3 部分”。

## 要求

Intrinsic	体系结构
__vmx_on	x64

头文件<intrin.h>

**结束 Microsoft 专用**

## 另请参阅

[编译器内部函数](#)

# `__vmx_vmclear`

项目 • 2023/06/16

## Microsoft 专用

初始化指定的虚拟机控制结构 (VMCS)，并将其启动状态设置为 `Clear`。

## 语法

```
C

unsigned char __vmx_vmclear(
    unsigned __int64 *VmcsPhysicalAddress
);
```

## 参数

*VmcsPhysicalAddress*

[in] 指向包含要清除的 VMCS 的物理地址的 64 位内存位置的指针。

## 返回值

值	含义
0	操作成功。
1	操作失败，当前 VMCS 的 <code>VM-instruction error field</code> 中提供了扩展状态。
2	操作失败，无可用状态。

## 注解

应用程序可以通过使用 `_vmx_vmlaunch` 或 `_vmx_vmresume` 函数执行 VM 输入操作。

`_vmx_vmlaunch` 函数只能用于启动状态为 `Clear` 的 VMCS，而 `_vmx_vmresume` 函数只能用于启动状态为 `Launched` 的 VMCS。因此，使用 `_vmx_vmclear` 函数将 VMCS 的启动状态设置为 `Clear`。对第一个 VM 输入操作使用 `_vmx_vmlaunch` 函数，将 `_vmx_vmresume` 函数用于后续的 VM 输入操作。

`_vmx_vmclear` 函数等同于 `VMCLEAR` 计算机指令。此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [Intel Corporation](#) 网站点搜

索“针对 IA-32 Intel 体系结构的 Intel 虚拟化技术规范”文档，文档编号为 C97063-002。

## 要求

Intrinsic	体系结构
<code>__vmx_vmclear</code>	X64

头文件<intrin.h>

**结束 Microsoft 专用**

## 另请参阅

编译器内部函数

[\\_vmx\\_vmlaunch](#)

[\\_vmx\\_vmresume](#)

# `__vmx_vmlaunch`

项目 • 2023/10/12

## Microsoft 专用

使用当前虚拟机控制结构 (VMCS) 将调用应用程序置于 VMX 非 root 操作状态 (VM 输入)。

## 语法

C

```
unsigned char __vmx_vmlaunch(void);
```

## 返回值

值	含义
0	操作成功。
1	操作失败，当前 VMCS 的 <code>VM-instruction error field</code> 中提供了扩展状态。
2	操作失败，无可用状态。

## 备注

应用程序可以通过使用 `_vmx_vmlaunch` 或 `_vmx_vmresume` 函数执行 VM 输入操作。`_vmx_vmlaunch` 函数只能用于启动状态为 `Clear` 的 VMCS，而 `_vmx_vmresume` 函数只能用于启动状态为 `Launched` 的 VMCS。因此，使用 `_vmx_vmclear` 函数将 VMCS 的启动状态设置为 `Clear`，然后将 `_vmx_vmlaunch` 函数用于第一个 VM 输入操作，将 `_vmx_vmresume` 函数用于后续 VM 输入操作。

`_vmx_vmlaunch` 函数等同于 `VMLAUNCH` 计算机指令。此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [Intel Corporation](#) 网站点搜索“针对 IA-32 Intel 体系结构的 Intel 虚拟化技术规范”文档，文档编号为 C97063-002。

## 要求

Intrinsic	体系结构
<code>__vmx_vmlaunch</code>	x64

头文件<intrin.h>

**结束 Microsoft 专用**

## 另请参阅

[编译器内部函数](#)

[\\_vmx\\_vmresume](#)

[\\_vmx\\_vmclear](#)

# `__vmx_vmptrld`

项目 • 2023/10/12

## Microsoft 专用

从指定地址加载指向当前虚拟机控制结构 (VMCS) 的指针。

## 语法

C

```
int __vmx_vmptrld(  
    unsigned __int64 *VmcsPhysicalAddress  
>);
```

## 参数

VmcsPhysicalAddress

[in] 存储 VMCS 指针的地址。

## 返回值

0

操作成功。

1

操作失败，当前 VMCS 的 `VM-instruction error field` 中提供了扩展状态。

2

操作失败，无可用状态。

## 备注

VMCS 指针是一个 64 位物理地址。

`__vmx_vmptrld` 函数等同于 `VMPTRLD` 计算机指令。此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [Intel Corporation](#) 网站点搜索“针对 IA-32 Intel 体系结构的 Intel 虚拟化技术规范”文档，文档编号为 C97063-002。

# 要求

Intrinsic	体系结构
<code>__vmx_vmptrld</code>	x64

头文件<intrin.h>

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[\\_vmx\\_vmptrst](#)

# `__vmx_vmptrst`

项目 • 2023/06/16

**Microsoft 专用**

在指定地址存储当前虚拟机控制结构 (VMCS) 的指针。

## 语法

C

```
void __vmx_vmptrst(  
    unsigned __int64 *VmcsPhysicalAddress  
) ;
```

## 参数

VmcsPhysicalAddress

[in] 存储当前 VMCS 指针的地址。

## 注解

VMCS 指针是一个 64 位物理地址。

`__vmx_vmptrst` 函数等同于 `VMPTRST` 计算机指令。此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [Intel Corporation](#) 网站点搜索“针对 IA-32 Intel 体系结构的 Intel 虚拟化技术规范”文档，文档编号为 C97063-002。

## 要求

Intrinsic	体系结构
<code>__vmx_vmptrst</code>	x86、x64

头文件<intrin.h>

结束 Microsoft 专用

## 另请参阅

编译器内部函数

`_vmx_vmptrld`

# `__vmx_vmread`

项目 • 2023/10/12

## Microsoft 专用

从当前虚拟机控制结构 (VMCS) 读取指定字段，并将其置于指定位置。

## 语法

C

```
unsigned char __vmx_vmread(  
    size_t Field,  
    size_t *FieldValue  
) ;
```

## 参数

### 字段

[in] 要读取的 VMCS 字段。

### FieldValue

[in] 一个指针，它指向用于存储从 `Field` 参数指定的 VMCS 字段中读取的值的位置。

## 返回值

值	含义
0	操作成功。
1	操作失败，当前 VMCS 的 <code>VM-instruction error field</code> 中提供了扩展状态。
2	操作失败，无可用状态。

## 备注

`__vmx_vmread` 函数等同于 `VMREAD` 计算机指令。`Field` 参数的值是 Intel 文档中所述的编码字段索引。有关详细信息，请在 [Intel Corporation](#) 网站上搜索“针对 IA-32 Intel 体系结构的 Intel 虚拟化技术规范”的附录 C。

# 要求

Intrinsic	体系结构
<code>__vmx_vmread</code>	x64

头文件<intrin.h>

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[\\_vmx\\_vmwrit](#)

# `__vmx_vmresume`

项目 • 2023/06/16

## Microsoft 专用

通过使用当前虚拟机控件结构 (VMCS) 恢复 VMX 非根操作。

## 语法

C

```
unsigned char __vmx_vmresume(  
    void);
```

## 返回值

值	含义
0	操作成功。
1	操作失败，当前 VMCS 的 <code>VM-instruction error field</code> 中提供了扩展状态。
2	操作失败，无可用状态。

## 注解

应用程序可以通过使用 `__vmx_vmlaunch` 或 `__vmx_vmresume` 函数执行 VM 输入操作。

`__vmx_vmlaunch` 函数只可用于启动状态为 `clear` 的 VMCS，而 `__vmx_vmresume` 函数只可用于启动状态为 `Launched` 的 VMCS。因此，使用 `__vmx_vmclear` 函数将 VMCS 的启动状态设置为 `clear`，然后对第一个 VM 输入操作使用 `__vmx_vmlaunch` 函数，对后续 VM 输入操作使用 `__vmx_vmresume` 函数。

`__vmx_vmresume` 函数等同于 `VMRESUME` 计算机指令。此函数支持主机的虚拟机监视器与来宾操作系统及其应用程序进行交互。有关详细信息，请在 [Intel Corporation](#) 站点搜索 PDF 文档“适用于 IA-32 Intel 架构的 Intel 虚拟化技术规范”，文档编号 C97063-002。

## 要求

Intrinsic

体系结构

Intrinsic	体系结构
<code>__vmx_vmresume</code>	X64

头文件<intrin.h>

**结束 Microsoft 专用**

## 另请参阅

[编译器内部函数](#)

[\\_vmx\\_vmlaunch](#)

[\\_vmx\\_vmclear](#)

# `__vmx_vmwrit`e

项目 • 2023/10/12

**Microsoft 专用**

将指定值写入当前虚拟机控制结构 (VMCS) 中的指定字段。

## 语法

C

```
unsigned char __vmx_vmwrit(
    size_t Field,
    size_tFieldValue
);
```

## 参数

**字段**

[in] 要写入的 VMCS 字段。

**FieldValue**

[in] 要写入 VMCS 字段的值。

## 返回值

0

操作成功。

1

操作失败，当前 VMCS 的 `VM-instruction error field` 中提供了扩展状态。

2

操作失败，无可用状态。

## 备注

`__vmx_vmwrit`e 函数等同于 `VMWRITE` 计算机指令。`Field` 参数的值是 Intel 文档中所述的编码字段索引。有关详细信息，请在 [Intel Corporation](#) 网站上搜索“针对 IA-32 Intel 体系结构的 Intel 虚拟化技术规范”的附录 C。

# 要求

Intrinsic	体系结构
<code>__vmx_vmwriteln</code>	x64

头文件<intrin.h>

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

[\\_vmx\\_vmread](#)

# **\_\_wbinvd**

项目 • 2023/10/12

## Microsoft 专用

生成回写和无效缓存 (`wbinvd`) 指令。

## 语法

C

```
void __wbinvd(void);
```

## 要求

Intrinsic	体系结构
<code>__wbinvd</code>	x86、x64

头文件<intrin.h>

## 备注

此函数仅在内核模式下可用，其特权级别 (CPL) 为 0，并且例程仅可用作内部函数。

## 结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# \_WriteBarrier

项目 • 2023/10/12

## Microsoft 专用

限制可重新排列调用点上的内存访问操作的编译器优化。

### ⊗ 注意

已全部弃用且不应使用 `_ReadBarrier`、`_WriteBarrier` 和 `_ReadWriteBarrier` 编译器内部函数和 `MemoryBarrier` 宏。对于线程间的通信，请使用 C++ 标准库中定义的机制，例如 `atomic_thread_fence` 和 `std::atomic<T>`。对于硬件访问，请将 `/volatile:iso` 编译器选项与 `volatile` 关键字一起使用。

## 语法

C

```
void _WriteBarrier(void);
```

## 要求

Intrinsic	体系结构
<code>_WriteBarrier</code>	x86、x64

头文件<intrin.h>

## 备注

`_WriteBarrier` 内部函数将限制可删除或重新排列调用点上的内存访问操作的编译器优化。

结束 Microsoft 专用

## 另请参阅

[\\_ReadBarrier](#)

[\\_ReadWriteBarrier](#)

[编译器内部函数](#)

[关键字](#)

# \_\_writecr0

项目 • 2024/08/04

Microsoft 专用

将值 `Data` 写入 CR0 寄存器。

## 语法

C

```
void __writecr0(  
    unsigned __int64 Data  
>;
```

## 参数

数据

[in] 要写入 CR0 寄存器的值。

## 要求

 展开表

Intrinsic	体系结构
<code>__writecr0</code>	x86、x64

头文件<intrin.h>

## 备注

此内部函数只在内核模式下可用，例程只能用作内部函数。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

---

# 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | [在 Microsoft Q&A 获得帮助](#)

# \_\_writecr3

项目 • 2023/10/12

Microsoft 专用

将值 `Data` 写入 CR3 寄存器。

## 语法

C

```
void __writecr3(  
    unsigned __int64 Data  
>;
```

## 参数

数据

[in] 要写入 CR3 寄存器的值。

## 要求

Intrinsic	体系结构
<code>__writecr3</code>	x86、x64

头文件<intrin.h>

## 备注

此内部函数只在内核模式下可用，例程只能用作内部函数。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# \_\_writecr4

项目 • 2023/10/12

Microsoft 专用

将值 `Data` 写入 CR4 寄存器。

## 语法

C

```
void __writecr4(  
    unsigned __int64 Data  
>;
```

## 参数

数据

[in] 要写入 CR4 寄存器的值。

## 要求

Intrinsic	体系结构
<code>__writecr4</code>	x86、x64

头文件<intrin.h>

## 备注

此内部函数只在内核模式下可用，例程只能用作内部函数。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# \_\_writecr8

项目 • 2023/10/12

Microsoft 专用

将值 `Data` 写入 CR8 寄存器。

## 语法

C

```
void __writecr8(  
    unsigned __int64 Data  
>;
```

## 参数

数据

[in] 要写入 CR8 寄存器的值。

## 要求

Intrinsic	体系结构
<code>__writecr8</code>	x64

头文件<intrin.h>

## 备注

`__writecr8` 内部仅在内核模式下可用，且例程仅可用作内部。

结束 Microsoft 专用

## 另请参阅

[编译器内部函数](#)

# \_\_writedr

项目 • 2023/06/16

**Microsoft 专用**

将指定的值写入指定调试寄存器。

## 语法

C

```
void __writedr(unsigned DebugRegister, unsigned DebugValue); /* x86 */
void __writedr(unsigned DebugRegister, unsigned __int64 DebugValue); /* x64
 */
```

## 参数

DebugRegister

[in] 一个从 0 到 7 的数字，用于标识调试寄存器。

DebugValue

[in] 要写入调试寄存器的值。

## 注解

这些内部函数仅在内核模式下可用，例程只能用作内部函数。

## 要求

Intrinsic	体系结构
__writedr	x86、x64

头文件<intrin.h>

结束 Microsoft 专用

**另请参阅**

编译器内部函数

\_readaddr

# \_\_writeeflags

项目 • 2023/06/16

**Microsoft 专用**

将指定的值写入程序状态和控件 (EFLAGS) 寄存器。

## 语法

C

```
void __writeeflags(unsigned Value); /* x86 */
void __writeeflags(unsigned __int64 Value); /* x64 */
```

## 参数

**值**

[in] 要写入 EFLAGS 寄存器的值。对于 32 位平台，参数 `Value` 长度为 32 位，对于 64 位平台，其参数长度为 64 位。

## 注解

这些例程只能用作内部函数。

## 要求

Intrinsic	体系结构
<code>__writeeflags</code>	x86、x64

头文件<intrin.h>

**结束 Microsoft 专用**

## 另请参阅

[编译器内部函数](#)

[\\_readeflags](#)

# **\_\_writefsbyte, \_\_writefsdword, \_\_writefsqword, \_\_writefsword**

项目 • 2023/10/12

**Microsoft 专用**

将内存写入相对于 FS 段开头的偏移量指定的位置。

## 语法

```
C

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
);
```

## 参数

*Offset*

[in] 要写入的 FS 起始位置的偏移量。

*数据*

[in] 要写入的值。

## 要求

Intrinsic	体系结构
__writefsbyte	x86

Intrinsic	体系结构
<code>__writefsword</code>	x86
<code>__writefsdword</code>	x86
<code>__writefsqword</code>	x86

头文件<intrin.h>

## 备注

这些例程只能用作内部函数。

结束 Microsoft 专用

## 另请参阅

[\\_readfsbyte, \\_readfsdword, \\_readfsqword, \\_readfsword](#)

编译器内部函数

# **\_\_writegsbyte, \_\_writegsdword, \_\_writegsqword, \_\_writegsword**

项目 • 2023/10/12

**Microsoft 专用**

将内存写入由相对于 GS 段开头的偏移量指定的位置。

## 语法

```
C

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
);
```

## 参数

*Offset*

[in] 要写入的 GS 起始位置的偏移量。

*数据*

[in] 要写入的值。

## 要求

Intrinsic	体系结构
__writegsbyte	X64

Intrinsic	体系结构
<code>__writegsdword</code>	X64
<code>__writegsqlword</code>	X64
<code>__writegsword</code>	X64

头文件<intrin.h>

## 备注

这些例程只能用作内部函数。

结束 Microsoft 专用

## 另请参阅

[\\_readgsbyte, \\_readgsdword, \\_readgsqlword, \\_readgsword](#)

编译器内部函数

# \_\_writemsr

项目 • 2023/06/16

Microsoft 专用

生成写入模型特定寄存器 (`wrmsr`) 指令。

## 语法

C

```
void __writemsr(
    unsigned long Register,
    unsigned __int64 Value
);
```

## 参数

注册

[in] 模型特定寄存器。

值

[in] 要写入的值。

## 要求

Intrinsic	体系结构
<code>__writemsr</code>	x86、x64

头文件<intrin.h>

## 注解

此函数只能在内核模式下使用，并且此例程仅作为内部函数提供。

结束 Microsoft 专用

## 另请参阅

## 编译器内部函数

# 内联汇编程序

项目 · 2023/04/03

## Microsoft 专用

汇编语言用作多种用途，例如提高程序的速度，减少内存需求和控制硬件。使用内联汇编程序，在没有额外汇编程序和链接步骤的情况下，也可直接在您的 C 和 C++ 源程序中嵌入汇编语言指令。内联汇编程序生成到该编译器中，因此您不需要一个单独的汇编程序，例如 Microsoft Macro Assembler (MASM)。

### ① 备注

具有内联汇编代码的程序不能完全移植到其他硬件平台。如果要针对可移植性进行设计，请避免使用内联汇编程序。

ARM 和 x64 处理器不支持内联汇编程序。以下主题解释如何使用具有 x86 处理器的可视 Visual C/C++ 内联汇编：

- [内联汇编概述](#)
- [内联程序集的优点](#)
- [\\_asm](#)
- [在 \\_asm 块中使用汇编语言](#)
- [在 \\_asm 块中使用 C 或 C++](#)
- [在内联汇编程序中使用和保留寄存器](#)
- [跳转到内联程序集中的标签](#)
- [在内联程序集中调用 C 函数](#)
- [在内联程序集中调用 C++ 函数](#)
- [将 \\_asm 块定义为 C 宏](#)
- [优化内联程序集](#)

结束 Microsoft 专用

另请参阅

编译器内部函数和程序集语言

C++ 语言参考

# ARM 汇编程序参考

项目 • 2023/04/03

文档本部分中的文章提供了 Microsoft ARM 汇编程序 (armasm 或 armasm64) 和相关工具的参考资料。

## 相关文章

Title	说明
<a href="#">ARM 汇编程序命令行参考</a>	介绍 Microsoft armasm 和 armasm64 命令行选项。
<a href="#">ARM 汇编程序诊断消息</a>	介绍常见的 armasm 和 armasm64 警告和错误消息。
<a href="#">ARM 汇编程序指令</a>	介绍 Microsoft armasm 和 armasm64 中不同的 ARM 指令。
<a href="#">ARM 开发人员网站上的 ARM 体系结构参考手册</a> ↗	选择 ARM 体系结构的相关手册。每个手册都包含有关 ARM、Thumb、NEON 和 VFP 的参考部分，以及有关 ARM 程序集语言的其他信息。
<a href="#">ARM 开发人员网站上的 ARM 编译器 armasm 用户指南</a> ↗	选择最新版本以查找有关 ARM 程序集语言的最新信息。

### ① 重要

ARM 开发人员网站介绍的 armasm 汇编程序与 Visual Studio 中包含的 Microsoft armasm 汇编程序不同，并记录在本部分中。

## 另请参阅

[ARM 内部函数](#)

[ARM64 内部函数](#)

[编译器内部函数](#)

# Microsoft 宏汇编程序参考

项目 • 2024/10/31

Microsoft 宏汇编 (MASM) 提供了一些相对于内联程序集的优点。 MASM 包含具有循环、 算术和字符串处理等功能的宏语言。 MASM 可以更好地控制硬件。 使用 MASM， 还可以节省时间和减少内存开销。

## 本节内容

### [ML 和 ML64 命令行选项](#)

介绍 ML 和 ML64 命令行选项。

### [MASM for x64 \(ml64.exe\)](#)

有关如何创建适用于 x64 的输出文件的信息。

### [指令格式](#)

介绍 MASM 的基本指令格式和指令前缀。

### [指令参考](#)

提供指向讨论 MASM 中指令使用的文章的链接。

### [符号参考](#)

提供指向讨论 MASM 中符号使用的文章的链接。

### [运算符参考](#)

提供指向讨论 MASM 中运算符使用的文章的链接。

### [ML 错误消息](#)

介绍致命和非致命的错误消息和警告。

### [处理器制造商编程手册](#)

提供有关非 Microsoft 制造、 销售或支持的处理器的编程信息的链接。

### [MASM BNF 语法](#)

适用于 x64 的 MASM 的正式 BNF 说明。

## 相关章节

### [Visual Studio 中的 C++](#)

提供到 Visual Studio 和 Visual C++ 文档不同区域的链接。

# 另请参阅

[编译器内部函数](#)

[x86 内部函数](#)

[x64 \(amd64\) 内部函数](#)

---

## 反馈

此页面是否有帮助?

 是

 否

[提供产品反馈](#) | 在 Microsoft Q&A 获得帮助