目录

[MMX（TM）技术的固有指令 4](#_Toc349205740)

[MMX（TM）技术普遍支持指令 4](#_Toc349205741)

[MMX（TM）技术包装的算法指令 5](#_Toc349205742)

[MMX（TM）技术移位指令 6](#_Toc349205743)

[MMX（TM）技术逻辑指令 8](#_Toc349205744)

[MMX（TM）技术比较指令 8](#_Toc349205745)

[MMX（TM）技术置位指令 9](#_Toc349205746)

[SSE的固有指令 10](#_Toc349205747)

[SSE的算术操作 10](#_Toc349205748)

[SSE的逻辑操作 13](#_Toc349205749)

[SSE的比较操作 14](#_Toc349205750)

[SSE的转换操作 18](#_Toc349205751)

[SSE的加载操作 20](#_Toc349205752)

[SSE的置位操作 22](#_Toc349205753)

[SSE的存储操作 23](#_Toc349205754)

[利用SSE进行缓存支持 24](#_Toc349205755)

[利用SSE指令的整型指令 24](#_Toc349205756)

[SSE的读写寄存器指令 27](#_Toc349205757)

[利用SSE的混杂指令 27](#_Toc349205758)

[SSE2的固有指令 29](#_Toc349205759)

[综述 29](#_Toc349205760)

[浮点指令 30](#_Toc349205761)

[浮点算术操作 30](#_Toc349205762)

[浮点逻辑操作 31](#_Toc349205763)

[浮点比较操作 32](#_Toc349205764)

[浮点转换操作 36](#_Toc349205765)

[浮点加载操作 38](#_Toc349205766)

[浮点置位操作 40](#_Toc349205767)

[浮点存储操作 41](#_Toc349205768)

[整型指令 42](#_Toc349205769)

[整型算术操作 42](#_Toc349205770)

[整型逻辑操作 47](#_Toc349205771)

[整型移位操作 48](#_Toc349205772)

[整型比较操作 50](#_Toc349205773)

[整型转换操作 52](#_Toc349205774)

[整型移动操作 53](#_Toc349205775)

[整型加载操作 53](#_Toc349205776)

[整型置位操作 54](#_Toc349205777)

[整型存储操作 56](#_Toc349205778)

[其它函数和指令 57](#_Toc349205779)

[缓存支持指令 57](#_Toc349205780)

[混杂指令 58](#_Toc349205781)

[类型转换指令 61](#_Toc349205782)

[暂停指令 62](#_Toc349205783)

[重排宏 62](#_Toc349205784)

[SSE3的固有指令 62](#_Toc349205785)

[综述 62](#_Toc349205786)

[整型向量指令 62](#_Toc349205787)

[单精度浮点型向量指令 62](#_Toc349205788)

[双精度浮点型向量指令 63](#_Toc349205789)

[宏函数 64](#_Toc349205790)

[混杂指令 64](#_Toc349205791)

[SSE3补充的固有指令 65](#_Toc349205792)

[综述 65](#_Toc349205793)

[加法指令 65](#_Toc349205794)

[减法指令 66](#_Toc349205795)

[乘法指令 67](#_Toc349205796)

[绝对值指令 68](#_Toc349205797)

[重排指令 68](#_Toc349205798)

[连接指令 69](#_Toc349205799)

[SSE4的固有指令 72](#_Toc349205800)

[综述 72](#_Toc349205801)

[向量化编译器和媒体加速器 72](#_Toc349205802)

[综述：SSE4向量化编译器和媒体加速器 72](#_Toc349205803)

[打包混合指令 72](#_Toc349205804)

[浮点型点积指令 73](#_Toc349205805)

[打包格式化转换指令 73](#_Toc349205806)

[打包整型min/max指令 74](#_Toc349205807)

[浮点型舍入指令 75](#_Toc349205808)

[DWORD乘法指令 76](#_Toc349205809)

[寄存器插入/提取指令 77](#_Toc349205810)

[测试指令 78](#_Toc349205811)

[打包DWORD到无符号WORD指令 78](#_Toc349205812)

[打包等于比较指令 78](#_Toc349205813)

[可缓存性支持指令 78](#_Toc349205814)

[高效加速的字符串和文本处理器 79](#_Toc349205815)

[综述 79](#_Toc349205816)

[打包比较指令 79](#_Toc349205817)

[应用定向加速器指令 80](#_Toc349205818)

[适用所有Intel架构的固有指令 81](#_Toc349205819)

[综述 81](#_Toc349205820)

[整型算术指令 81](#_Toc349205821)

[浮点型指令 82](#_Toc349205822)

[字符串和块拷贝指令 85](#_Toc349205823)

[混杂指令 86](#_Toc349205824)

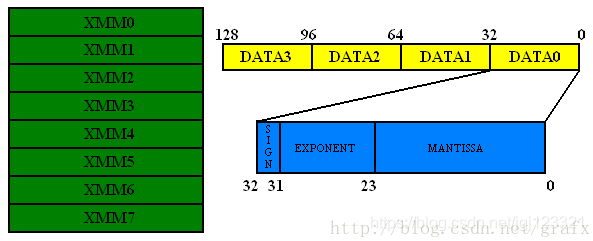
[Intrinsics for IA-32 and Intel® 64 Architectures Only 88](#_Toc349205825)

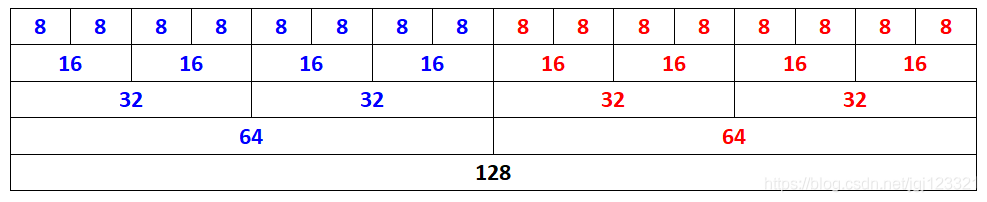
### SSE简介

SIMD（Single Instruction Multiple Data）是单指令多数据技术，目前Intel处理器支持的SIMD技术包括MMX，SSE，AVX。

SSE（Stream SIMD Extentions，数据流单指令多数据扩展）是英特尔继MMX（Multi Media eXtension，多媒体扩展指令集）之后推出的新一代CPU指令集。MMX提供了8个64bit的寄存器进行SIMD操作，SSE系列提供了8个128bit的寄存器进行SIMD操作。而最新的AVX指令则支持256bit的SIMD操作。

如下图所示，SSE新增的8个128位寄存器（xmm0 ~ xmm7），每个寄存器可以用来存放4个32位单精度浮点数，8个16位整型数。也就是说，SSE中的所有计算都是一次性针对4个浮点数来完成的，这种批处理会带来显著的效率提升。使用SSE优化之后，我们的代码不一定会得到4倍速的提升，因为编译器可能已经自动对某些代码进行SSE优化了。





### 如何使用SSE指令

       使用SSE指令有两种方式：一是直接在C/C++中嵌入（汇编）指令；二是使用Intel C++ Compiler或是Microsoft Visual C++中提供的支持SSE指令集的intrinsics内联函数。从代码可读和维护角度讲，推荐使用intrinsics内联函数的形式。intrinsics是对MMX、SSE等指令集的一种封装，以函数的形式提供，使得程序员更容易编写和使用这些高级指令，在编译的时候，这些函数会被内联为汇编，不会产生函数调用的开销。想要使用SSE指令，则需要包含对应的头文件：

1. #include <mmintrin.h> //mmx
2. #include <xmmintrin.h> //sse
3. #include <emmintrin.h> //sse2
4. #include <pmmintrin.h> //sse3

### SSE指令的格式

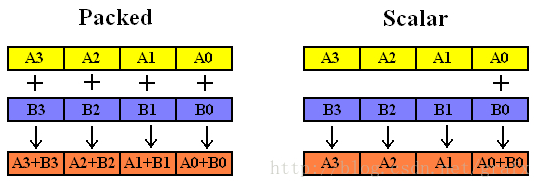
SSE指令通常由三部分构成：

第一部分为前缀\_mm（多媒体扩展指令集），表示该函数属于SSE指令集

第二部分为指令的操作类型，如\_add、\_mul等

第三部分通常由两个字母组成。第一个字母表示对结果变量的影响方式，为p或者s。

p(packed：包裹指令) ：该指令对xmm寄存器中的每个元素进行运算，即一次对四个浮点数(data0~data3)均进行计算；  
s(scalar：标量指令)：该指令对寄存器中的第一个元素进行运算，即一次只对xmm寄存器中的data0进行计算。



第二个字母表示参与运算的数据类型，s表示32位浮点数，d表示64位浮点数，i32表示带符号32位整型，i64表示带符号64位整型，u32表示无符号32位整型，以此类推。由于SSE只支持32位浮点数的运算，所以你可能会在这些指令封装函数中找不到包含非s修饰符的，但你可以在MMX和SSE2的指令集中去认识它们。

\_pixx（xx为长度，可以是8，16，32，64）packed操作所有的xx位有符号整数，使用的寄存器长度为64位；\_epixx（xx为长度）packed操作所有的xx位的有符号整数，使用的寄存器长度为128位；\_epuxx packed操作所有的xx位的无符号整数；

### SSE的数据类型

SSE指令中intrinsics函数的数据类型为：\_\_m128（单精度浮点数），如果使用sizeof(\_\_m128)计算该类型大小，结果为16，即等于四个浮点数长度。\_\_declspec(align(16))做为数组定义的修释符，表示该数组是以16字节为边界对齐的，因为SSE指令大部分支持这种格式的内存数据。他的定义如下：

1. typedef struct \_\_declspec(intrin\_type) \_\_declspec(align(16)) \_\_m128 {
2. float m128\_f32[4];
3. } \_\_m128;

除\_\_m128外、还包括\_\_m128d（双精度浮点数）和\_\_m128i（整型）。其中\_\_m128i是一个共用体类型，其定义如下 ：

1. typedef union \_\_declspec(intrin\_type)\_CRT\_ALIGN(16)\_\_m128i {
2. \_\_int8 m128i\_i8[16]; *//char*
3. \_\_int16 m128i\_i16[8]; *//short*
4. \_\_int32 m128i\_i32[4]; *//int*
5. \_\_int64 m128i\_i64[2]; *//long long*
6. unsigned \_\_int8 m128i\_u8[16]; *//uchar*
7. unsigned \_\_int16 m128i\_u16[8]; *//ushort*
8. unsigned \_\_int32 m128i\_u32[4]; *//uint*
9. unsigned \_\_int64 m128i\_u64[2]; *//ulonglong*
10. }\_\_m128i;

### AVX支持256宽度指令。

一、SIMD数据类型简介

\_\_m128：128位紧缩单精度（SSE）。

\_\_m128d：128位紧缩双精度（SSE2）。

\_\_m128i：128位紧缩整数（SSE2）。

\_\_m256：256位紧缩单精度（AVX）。

\_\_m256d：256位紧缩双精度（AVX）。

\_\_m256i：256位紧缩整数（AVX）。

注：紧缩整数包括了8位、16位、32位、64位的带符号和无符号整数。

这些数据类型与寄存器的对应关系为——

128位SSE寄存器（XMM0~XMM15）：\_\_m128、\_\_m128d、\_\_m128i。

256位AVX寄存器（YMM0~YMM15）：\_\_m256、\_\_m256d、\_\_m256i。

# MMX（TM）技术的固有指令

综述

关于MMX技术指令的细节

EMMS指令：为什么需要它

EMMS使用指南

## MMX（TM）技术普遍支持指令

原型在mmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_empty | Empty MM state | 清空多媒体状态 |
| \_mm\_cvtsi32\_si64 | Convert from int | int转\_\_m64，多余位补0 |
| \_mm\_cvtsi64\_si32 | Convert to int | \_\_m64低32位转int |
| \_mm\_cvtsi64\_m64 | Convert from \_\_int64 | 64位int转\_\_m64 |
| \_mm\_cvtm64\_si64 | Convert to \_\_int64 | \_\_m64转64位int |
| \_mm\_packs\_pi16(\_\_m64 m1, \_\_m64 m2) | Pack | m1的4个16位到低的4个8位，m2的4个16位到高的4个8位（有符号饱和的原则） |
| \_mm\_packs\_pi32(\_\_m64 m1, \_\_m64 m2) | Pack | m1的2个32位到低的2个16位，m2的2个32位到高的2个16位（有符号饱和原则） |
| \_mm\_packs\_pu16(\_\_m64 m1, \_\_m64 m2) | Pack | m1的4个16位到低的4个8位，m2的4个16位到高的4个8位（无符号饱和的原则） |
| \_mm\_unpackhi\_pi8(\_\_m64 m1, \_\_m64 m2) | Interleave | 交织m1的高一半的4个8位和m2的高一半的4个8位，以m1的数据开头 |
| \_mm\_unpackhi\_pi16(\_\_m64 m1, \_\_m64 m2) | Interleave | 交织m1的高一半的2个16位和m2的高一半的2个16位，以m1的数据开头 |
| \_mm\_unpackhi\_pi32(\_\_m64 m1, \_\_m64 m2) | Interleave | 交织m1的高一半的1个32位和m2的高一半的1个32位，以m1的数据开头 |
| \_mm\_unpacklo\_pi8(\_\_m64 m1, \_\_m64 m2) | Interleave | 交织m1的低一半的4个8位和m2的低一半的4个8位，以m1的数据开头 |
| \_mm\_unpacklo\_pi16(\_\_m64 m1, \_\_m64 m2) | Interleave | 交织m1的低一半的2个16位和m2的低一半的2个16位，以m1的数据开头 |
| \_mm\_unpacklo\_pi32(\_\_m64 m1, \_\_m64 m2) | Interleave | 交织m1的低一半的1个32位和m2的低一半的1个32位，以m1的数据开头 |

## MMX（TM）技术包装的算法指令

原型在mmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_add\_pi8(\_\_m64 m1, \_\_m64 m2) | Addition |  |
| \_mm\_add\_pi16(\_\_m64 m1, \_\_m64 m2) | Addition |  |
| \_mm\_add\_pi32(\_\_m64 m1, \_\_m64 m2) | Addition |  |
| \_mm\_adds\_pi8(\_\_m64 m1, \_\_m64 m2) | Addition | 饱和机制 |
| \_mm\_adds\_pi16(\_\_m64 m1, \_\_m64 m2) | Addition |  |
| \_mm\_adds\_pu8(\_\_m64 m1, \_\_m64 m2) | Addition | 无符号、饱和机制 |
| \_mm\_adds\_pu16(\_\_m64 m1, \_\_m64 m2) | Addition |  |
| \_mm\_sub\_pi8(\_\_m64 m1, \_\_m64 m2) | Subtraction |  |
| \_mm\_sub\_pi16(\_\_m64 m1, \_\_m64 m2) | Subtraction |  |
| \_mm\_sub\_pi32(\_\_m64 m1, \_\_m64 m2) | Subtraction |  |
| \_mm\_subs\_pi8(\_\_m64 m1, \_\_m64 m2) | Subtraction |  |
| \_mm\_subs\_pi16(\_\_m64 m1, \_\_m64 m2) | Subtraction |  |
| \_mm\_subs\_pu8(\_\_m64 m1, \_\_m64 m2) | Subtraction |  |
| \_mm\_subs\_pu16(\_\_m64 m1, \_\_m64 m2) | Subtraction |  |
| \_mm\_madd\_pi16(\_\_m64 m1, \_\_m64 m2) | Multiply and add | m1的4个16位×m2的4个16位得到4个32位的中间结果，然后分对相加得到2个32位结果 |
| \_mm\_mulhi\_pi16(\_\_m64 m1, \_\_m64 m2) | Multiplication | m1的4个有符号16位×m2的4个有符号16位，得到4个结果的高16位 |
| \_mm\_mullo\_pi16(\_\_m64 m1, \_\_m64 m2) | Multiplication | m1的4个有符号16位×m2的4个有符号16位，得到4个结果的低16位 |

## MMX（TM）技术移位指令

原型在mmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_sll\_pi16(\_\_m64 m, \_\_m64 count) | Logical shift left | m中的4个16位左移count位，补0 |
| \_mm\_slli\_pi16(\_\_m64 m, \_\_m64 count) | Logical shift left | m中的4个16位左移count位，补0，为了最好的性能，count是一个常量 |
| \_mm\_sll\_pi32(\_\_m64 m, \_\_m64 count) | Logical shift left | m中的2个32位左移count位，补0 |
| \_mm\_slli\_pi32(\_\_m64 m, \_\_m64 count) | Logical shift left | m中的2个32位左移count位，补0，为了最好的性能，count是一个常量 |
| \_mm\_sll\_pi64(\_\_m64 m, \_\_m64 count) | Logical shift left | m中的1个64位左移count位，补0 |
| \_mm\_slli\_pi64(\_\_m64 m, \_\_m64 count) | Logical shift left | m中的1个64位左移count位，补0，为了最好的性能，count是一个常量 |
| \_mm\_sra\_pi16(\_\_m64 m, \_\_m64 count) | Arithmetic shift right | m中的4个16位右移count位，保留符号位 |
| \_mm\_srai\_pi16(\_\_m64 m, \_\_m64 count) | Arithmetic shift right | m中的4个16位右移count位，保留符号位 |
| \_mm\_sra\_pi32(\_\_m64 m, \_\_m64 count) | Arithmetic shift right | m中的2个32位右移count位，保留符号位 |
| \_mm\_srai\_pi32(\_\_m64 m, \_\_m64 count) | Arithmetic shift right | m中的2个32位右移count位，保留符号位。Count是一个常量 |
| \_mm\_srl\_pi16(\_\_m64 m, \_\_m64 count) | Logical shift right | m中的4个16位右移count位，补0 |
| \_mm\_srli\_pi16(\_\_m64 m, \_\_m64 count) | Logical shift right | m中的4个16位右移count位，补0，count是一个常量 |
| \_mm\_srl\_pi32(\_\_m64 m, \_\_m64 count) | Logical shift right | m中的2个32位右移count位，补0 |
| \_mm\_srli\_pi32(\_\_m64 m, \_\_m64 count) | Logical shift right | m中的2个32位右移count位，补0，count是一个常量 |
| \_mm\_srl\_pi64(\_\_m64 m, \_\_m64 count) | Logical shift right | m中的1个64位右移count位，补0 |
| \_mm\_srli\_pi64(\_\_m64 m, \_\_m64 count) | Logical shift right | m中的1个64位右移count位，补0，count是一个常量 |

## MMX（TM）技术逻辑指令

原型在mmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_and\_si64(\_\_m64 m1, \_\_m64 m2) | Bitwise AND | m1的64位和m2中64位位与 |
| \_mm\_andnot\_si64(\_\_m64 m1, \_\_m64 m2) | Bitwise ANDNOT | m1的64位位非，然后和m2中64位位与 |
| \_mm\_or\_si64(\_\_m64 m1, \_\_m64 m2) | Bitwise OR | m1的64位和m2中64位位或 |
| \_mm\_xor\_si64(\_\_m64 m1, \_\_m64 m2) | Bitwise Exclusive OR | m1的64位和m2中64位位异或 |

## MMX（TM）技术比较指令

原型在mmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_cmpeq\_pi8(\_\_m64 m1, \_\_m64 m2) | Equal | 如果m1中的8位与m2中的8位相等，则结果全置1；否则全置0 |
| \_mm\_cmpeq\_pi16(\_\_m64 m1, \_\_m64 m2) | Equal | 如果m1中的16位与m2中的16位相等，则结果全置1；否则全置0 |
| \_mm\_cmpeq\_pi32(\_\_m64 m1, \_\_m64 m2) | Equal | 如果m1中的32位与m2中的32位相等，则结果全置1；否则全置0 |
| \_mm\_cmpgt\_pi8(\_\_m64 m1, \_\_m64 m2) | Greater Than | 如果m1中的8位有符号大于m2中的8位有符号，则结果全置1；否则全置0 |
| \_mm\_cmpgt\_pi16(\_\_m64 m1, \_\_m64 m2) | Greater Than | 如果m1中的16位有符号大于m2中的16位有符号，则结果全置1；否则全置0 |
| \_mm\_cmpgt\_pi32(\_\_m64 m1, \_\_m64 m2) | Greater Than | 如果m1中的32位有符号大于m2中的32位有符号，则结果全置1；否则全置0 |

## MMX（TM）技术置位指令

原型在mmintrin.h头文件中

Note：在摘要中关于mmx寄存器的比特位，第0位是最不重要的，第63是最重要的

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_setzero\_si64() | set to zero | 将64位置0 |
| \_mm\_set\_pi32(int i1,int i0) | set integer values | 置2个有符号32位整型 |
| \_mm\_set\_pi16 (short s3, short s2, short s1, short s0) | set integer values | 置4个有符号16位 |
| \_mm\_set\_pi8 (char b7, char b6, char b5, char b4, char b3, char b2, char b1, char b0) | set integer values | 置8个有符号8位 |
| \_mm\_set1\_pi32(int i) | set integer values | 置2个有符号32位 |
| \_mm\_set1\_pi16(short s) | set integer values | 置4个有符号16位 |
| \_mm\_set1\_pi8 (char b) | set integer values | 置8个有符号8位 |
| \_mm\_setr\_pi32 | set integer values | 逆序置2个有符号32位整型 |
| \_mm\_setr\_pi16 (short s3, short s2, short s1, short s0) | set integer values | 逆序置4个有符号16位 |
| \_mm\_setr\_pi8 | set integer values | 逆序置8个有符号8位 |

IA-64架构上的MMX（TM）技术指令

# SSE的固有指令

综述

SSE指令的细节

利用SSE指令编写程序

## SSE的算术操作

原型在xmmintrin.h头文件中

每个指令操作的结果存放在寄存器中。这些寄存器用R0-R3来描述，R1，R2，R3，R4分别表示结果寄存器中的4个32位。

\*single-precision, floating-point (SP FP):单精度浮点

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_add\_ss (\_\_m128 a, \_\_m128 b) | Addition | ，将a和b中的低位的单精度浮点的相加；高位的3个SPFP从a中取 |
| \_mm\_add\_ps (\_\_m128 a, \_\_m128 b) | Addition |  |
| \_mm\_sub\_ss (\_\_m128 a, \_\_m128 b) | Subtraction |  |
| \_mm\_sub\_ps (\_\_m128 a, \_\_m128 b) | Subtraction |  |
| \_mm\_mul\_ss (\_\_m128 a, \_\_m128 b) | Multiplication |  |
| \_mm\_mul\_ps (\_\_m128 a, \_\_m128 b) | Multiplication |  |
| \_mm\_div\_ss (\_\_m128 a, \_\_m128 b) | Division |  |
| \_mm\_div\_ps (\_\_m128 a, \_\_m128 b) | Division |  |
| \_mm\_sqrt\_ss (\_\_m128 a) | Squared Root |  |
| \_mm\_sqrt\_ps (\_\_m128 a) | Squared Root |  |
| \_mm\_rcp\_ss (\_\_m128 a) | Reciprocal | 计算倒数的近似值 |
| \_mm\_rcp\_ps (\_\_m128 a) | Reciprocal |  |
| \_mm\_rsqrt\_ss (\_\_m128 a) | Reciprocal Squared Root |  |
| \_mm\_rsqrt\_ps (\_\_m128 a) | Reciprocal Squared Root |  |
| \_mm\_min\_ss (\_\_m128 a, \_\_m128 b) | Computes Minimum |  |
| \_mm\_min\_ps (\_\_m128 a, \_\_m128 b) | Computes Minimum |  |
| \_mm\_max\_ss (\_\_m128 a, \_\_m128 b) | Computes Maximum |  |
| \_mm\_max\_ps (\_\_m128 a, \_\_m128 b) | Computes Maximum |  |

## SSE的逻辑操作

原型在xmmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_and\_ps(\_\_m128 a, \_\_m128 b) | Bitwise AND | 4个SPFP位与 |
| \_mm\_andnot\_ps(\_\_m128 a, \_\_m128 b) | Bitwise ANDNOT | 4个SPFP位与非 |
| \_mm\_or\_ps(\_\_m128 a, \_\_m128 b) | Bitwise OR | 4个SPFP位或 |
| \_mm\_xor\_ps(\_\_m128 a, \_\_m128 b) | Bitwise Exclusive OR |  |

## 

## SSE的比较操作

原型在xmmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_cmpeq\_ss(\_\_m128 a, \_\_m128 b) | Equal |  |
| \_mm\_cmpeq\_ps | Equal |  |
| \_mm\_cmplt\_ss | Less Than |  |
| \_mm\_cmplt\_ps | Less Than |  |
| \_mm\_cmple\_ss | Less Than or Equal |  |
| \_mm\_cmple\_ps | Less Than or Equal |  |
| \_mm\_cmpgt\_ss | Greater Than |  |
| \_mm\_cmpgt\_ps | Greater Than |  |
| \_mm\_cmpge\_ss | Greater Than or Equal |  |
| \_mm\_cmpge\_ps | Greater Than or Equal |  |
| \_mm\_cmpneq\_ss | Not Equal |  |
| \_mm\_cmpneq\_ps | Not Equal |  |
| \_mm\_cmpnlt\_ss | Not Less Than |  |
| \_mm\_cmpnlt\_ps | Not Less Than |  |
| \_mm\_cmpnle\_ss | Not Less Than or Equal |  |
| \_mm\_cmpnle\_ps | Not Less Than or Equal |  |
| \_mm\_cmpngt\_ss | Not Greater Than |  |
| \_mm\_cmpngt\_ps | Not Greater Than |  |
| \_mm\_cmpnge\_ss | Not Greater Than or Equal |  |
| \_mm\_cmpnge\_ps | Not Greater Than or Equal |  |
| \_mm\_cmpord\_ss | Ordered |  |
| \_mm\_cmpord\_ps | Ordered |  |
| \_mm\_cmpunord\_ss | Unordered |  |
| \_mm\_cmpunord\_ps | Unordered |  |
| \_mm\_comieq\_ss | Equal |  |
| \_mm\_comilt\_ss | Less Than |  |
| \_mm\_comile\_ss | Less Than or Equal |  |
| \_mm\_comigt\_ss | Greater Than |  |
| \_mm\_comige\_ss | Greater Than or Equal |  |
| \_mm\_comineq\_ss | Not Equal |  |
| \_mm\_ucomieq\_ss | Equal |  |
| \_mm\_ucomilt\_ss | Less Than |  |
| \_mm\_ucomile\_ss | Less Than or Equal |  |
| \_mm\_ucomigt\_ss | Greater Than |  |
| \_mm\_ucomige\_ss | Greater Than or Equal |  |
| \_mm\_ucomineq\_ss | Not Equal |  |

## SSE的转换操作

原型在xmmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_cvtss\_si32(\_\_m128 a) | Convert to 32-bit integer | 将低位的SPFP转成32位整型（舍入模式） |
| \_mm\_cvtss\_si64 | Convert to 64-bit integer | 将低位的SPFP转成64位有符号整型（舍入模式） |
| \_mm\_cvtps\_pi32 | Convert to two 32-bit integers |  |
| \_mm\_cvttss\_si32 | Convert to 32-bit integer | 将低位的SPFP转成32位整型（截尾模式） |
| \_mm\_cvttss\_si64 | Convert to 64-bit integer | 将低位的SPFP转成64位有符号整型（截尾模式） |
| \_mm\_cvttps\_pi32 | Convert to two 32-bit integers |  |
| \_mm\_cvtsi32\_ss(\_\_m128 a, int b) | Convert from 32-bit integer |  |
| \_mm\_cvtsi64\_ss | Convert from 64-bit integer |  |
| \_mm\_cvtpi32\_ps | Convert from two 32-bit integers |  |
| \_mm\_cvtpi16\_ps(\_\_m64 a) | Convert from four 16-bit integers | 将4个16位有符号转为SPFP |
| \_mm\_cvtpu16\_ps | Convert from four 16-bit integers | 将4个16位无符号转为SPFP |
| \_mm\_cvtpi8\_ps | Convert from four 8-bit integers | 将低位的4个8位有符号转为SPFP |
| \_mm\_cvtpu8\_ps | Convert from four 8-bit integers | 将低位的4个8位无符号转为SPFP |
| \_mm\_cvtpi32x2\_ps (\_\_m64 a, \_\_m64 b) | Convert from four 32-bit integers | a中的2个32位有符号和b中的2个32位转为4个SPFP |
| \_mm\_cvtps\_pi16 (\_\_m128 a) | Convert to four 16-bit integers | 将a中的四个SPFP转为4个有符号的16位整型 |
| \_mm\_cvtps\_pi8 | Convert to four 8-bit integers | 将2个SPFP转到结果的低位的4个有符号8位 |
| \_mm\_cvtss\_f32 | Extract | 从\_\_128的第一个向量元素摘取一个SPFP。在上下文应用中可能是最有效的方式。 |

## SSE的加载操作

原型在xxmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_loadh\_pi(\_\_m128 a, \_\_m64 const \*p) | Load high | 从地址p中加载的64位数据来置位a的高位的SPFP |
| \_mm\_loadl\_pi | Load low | 从地址p中加载的64位数据来置位a的低位的SPFP |
| \_mm\_load\_ss(float \* p ) | Load the low value and clear the three high values | 加载1个SPFP在低位，且高位清0 |
| \_mm\_load1\_ps | Load one value into all four words | 加载1个SPFP，且把它拷到所有的4个字节 |
| \_mm\_load\_ps | Load four values, address aligned | 加载4个SPFP。地址必须是16字节对齐的 |
| \_mm\_loadu\_ps | Load four values, address unaligned | 加载4个SPFP。地址不需要16字节对齐 |
| \_mm\_loadr\_ps | Load four values in reverse | 逆序加载4个SPFP。地址必须16字节对齐 |

## SSE的置位操作

原型在xmmintri.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_set\_ss(float w ) | Set the low value and clear the three high values | 置低位的1个SPFP，高位3字节清0 |
| \_mm\_set1\_ps | Set all four words with the same value | 置4个SPFP |
| \_mm\_set\_ps (float z, float y, float x, float w ) | Set four values, address aligned | 置4个SPFP为4个数 |
| \_mm\_setr\_ps(float z, float y, float x, float w ) | Set four values, in reverse order | 逆序置4个SPFP为4个数 |
| \_mm\_setzero\_ps(void) | Clear all four values | 4个SPFP清0 |

## SSE的存储操作

原型在xmmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_storeh\_pi(\_\_m64 \*p, \_\_m128 a) | Store high | 将高位的2个SPFP存入地址p中 |
| \_mm\_storel\_pi | Store low | 将低位的2个SPFP存入地址p中 |
| \_mm\_store\_ss (float \* p, \_\_m128 a) | Store the low value | 存入最低的SPFP |
| \_mm\_store1\_ps (float \* p, \_\_m128 a ) | Store the low value across all four words, address aligned | 存储低位SPFP并贯穿4个字节 |
| \_mm\_store\_ps | Store four values, address aligned | 存储4个SPFP。地址必须是16字节对齐的 |
| \_mm\_storeu\_ps | Store four values, address unaligned | 存储4个SPFP。地址不需要16字节对齐 |
| \_mm\_storer\_ps | Store four values, in reverse order | 逆序存储4个SPFP。地址必须是16字节对齐的 |

## 利用SSE进行缓存支持

原型在xmmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_prefetch (char const\*a, int sel) | Load |  |
| \_mm\_stream\_pi (\_\_m64 \*p, \_\_m64 a) | Store |  |
| \_mm\_stream\_ps (float \*p, \_\_m128 a) | Store |  |
| \_mm\_sfence (void) | Store fence |  |

## 利用SSE指令的整型指令

原型在xmmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_extract\_pi16 (\_\_m64 a, int n) | Extract one of four words | 从a中取1个。n必须是一个立即数 |
| \_mm\_insert\_pi16 (\_\_m64 a, int d, int n) | Insert word | a的4个数中插入一个d。n必须是一个立即数 |
| \_mm\_max\_pi16 (\_\_m64 a, \_\_m64 b) | Compute maximum | 得出a、b中对应位置的最大值 |
| \_mm\_max\_pu8 | Compute maximum, unsigned | 得出a、b中的对应的无符号的最大值 |
| \_mm\_min\_pi16 | Compute minimum | 得出a、b中对应位置的最小值 |
| \_mm\_min\_pu8 | Compute minimum, unsigned | 得出a、b中对应位置的最小值 |
| \_mm\_movemask\_pi8(\_\_m64 b) | Create eight-bit mask | 从a的最重要的比特位中创造出1个8位的掩码 |
| \_mm\_mulhi\_pu16(\_\_m64 a, \_\_m64 b) | Multiply, return high bits | a和b的无符号相乘，返回32位中间结果的高16位 |
| \_mm\_shuffle\_pi16(\_\_m64 a, int n) | Return a combination of four words | 返回a的4个数的1个联合。n必须是一个立即数 |
| \_mm\_maskmove\_si64(\_\_m64 d, \_\_m64 n, char \*p) | Conditional Store | 有条件的向地址p中存储d的元素。n中每个字节的高比特位决定了d中对应的字节是否存储 |
| \_mm\_avg\_pu8(\_\_m64 a, \_\_m64 b) | Compute rounded average | 计算a和b中无符号的平均值（round模式） |
| \_mm\_avg\_pu16 | Compute rounded average | 计算a和b中无符号的平均值（round模式） |
| \_mm\_sad\_pu8 | Compute sum of absolute differences | 计算a和b中无符号数的差的绝对值的总和，且高位置0 |

## SSE的读写寄存器指令

原型在xmmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_getcsr (void) | Return control register | 返回控制寄存器的内容 |
| \_mm\_setcsr (unsigned int i) | Set control register | 将控制寄存器置位指定的值 |

## 利用SSE的混杂指令

原型在xmmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_shuffle\_ps (\_\_m128 a, \_\_m128 b, unsigned int imm8) | Shuffle | 基于imm8从a和b中选择4个指定的SPFP，掩码必须是立即数 |
| \_mm\_unpackhi\_ps (\_\_m128 a, \_\_m128 b) | Unpack High | a和b中的高位2个SPFP进行交织 |
| \_mm\_unpacklo\_ps | Unpack Low | a和b中的低位2个SPFP进行交织 |
| \_mm\_move\_ss | Set low word, pass in three high values | 将a的低位置成b的 |
| \_mm\_movehl\_ps | Move High to Low |  |
| \_mm\_movelh\_ps | Move Low to High |  |
| \_mm\_movemask\_ps (\_\_m128 a) | Create four-bit mask | 从4个SPFP的最重要比特位中创造1个4比特的掩码 |

IA-64架构的SSE指令

## 宏函数

### 重排的宏函数

### 读写寄存器的宏函数

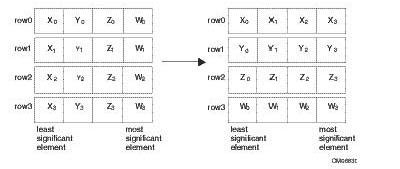
| **异常状态宏** | **宏参数** |
| --- | --- |
| \_MM\_SET\_EXCEPTION\_STATE(x) | \_MM\_EXCEPT\_INVALID |
| \_MM\_GET\_EXCEPTION\_STATE() | \_MM\_EXCEPT\_DIV\_ZERO |
|  | \_MM\_EXCEPT\_DENORM |
| **Macro Definitions** Write to and read from the six least significant control register bits, respectively. | \_MM\_EXCEPT\_OVERFLOW |
|  | \_MM\_EXCEPT\_UNDERFLOW |
|  | \_MM\_EXCEPT\_INEXACT |

### 矩阵变换的宏函数

\_MM\_TRANSPOSE4\_PS(row0, row1, row2, row3)

Matrix Transposition Using \_MM\_TRANSPOSE4\_PS Macro

类似于矩阵转置



# SSE2的固有指令

## 综述

## 浮点指令

### 浮点算术操作

函数原型在emmintrin.h头文件中

(double-precision, floating-point)DPFP，双精度浮点

| **指令** | **操作** | **简要操作** |
| --- | --- | --- |
| \_mm\_add\_sd(\_\_m128d a, \_\_m128d b) | Addition |  |
| \_mm\_add\_pd | Addition |  |
| \_mm\_sub\_sd | Subtraction |  |
| \_mm\_sub\_pd | Subtraction |  |
| \_mm\_mul\_sd | Multiplication |  |
| \_mm\_mul\_pd | Multiplication |  |
| \_mm\_div\_sd | Division |  |
| \_mm\_div\_pd | Division |  |
| \_mm\_sqrt\_sd | Computes Square Root |  |
| \_mm\_sqrt\_pd(\_\_m128d a) | Computes Square Root |  |
| \_mm\_min\_sd(\_\_m128d a, \_\_m128d b) | Computes Minimum |  |
| \_mm\_min\_pd | Computes Minimum |  |
| \_mm\_max\_sd | Computes Maximum |  |
| \_mm\_max\_pd | Computes Maximum |  |

### 浮点逻辑操作

原型在emmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_and\_pd(\_\_m128d a, \_\_m128d b) | Computes AND | 计算2个DPFP的位与 |
| \_mm\_andnot\_pd | Computes AND and NOT |  |
| \_mm\_or\_pd | Computes OR |  |
| \_mm\_xor\_pd | Computes XOR |  |

### 浮点比较操作

原型在emmintrin.h头文件中 功能看似相同的函数，有的是\_\_m128型，有的是int型

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_cmpeq\_pd(\_\_m128d a, \_\_m128d b) | Equality |  |
| \_mm\_cmplt\_pd | Less Than |  |
| \_mm\_cmple\_pd | Less Than or Equal |  |
| \_mm\_cmpgt\_pd | Greater Than |  |
| \_mm\_cmpge\_pd | Greater Than or Equal |  |
| \_mm\_cmpord\_pd | Ordered | 比较a和b是否有序 |
| \_mm\_cmpunord\_pd | Unordered |  |
| \_mm\_cmpneq\_pd | Inequality |  |
| \_mm\_cmpnlt\_pd | Not Less Than |  |
| \_mm\_cmpnle\_pd | Not Less Than or Equal |  |
| \_mm\_cmpngt\_pd | Not Greater Than |  |
| \_mm\_cmpnge\_pd | Not Greater Than or Equal |  |
| \_mm\_cmpeq\_sd | Equality |  |
| \_mm\_cmplt\_sd | Less Than |  |
| \_mm\_cmple\_sd | Less Than or Equal |  |
| \_mm\_cmpgt\_sd | Greater Than |  |
| \_mm\_cmpge\_sd | Greater Than or Equal |  |
| \_mm\_cmpord\_sd | Ordered |  |
| \_mm\_cmpunord\_sd | Unordered |  |
| \_mm\_cmpneq\_sd | Inequality |  |
| \_mm\_cmpnlt\_sd | Not Less Than |  |
| \_mm\_cmpnle\_sd | Not Less Than or Equal |  |
| \_mm\_cmpngt\_sd | Not Greater Than |  |
| \_mm\_cmpnge\_sd | Not Greater Than or Equal |  |
| \_mm\_comieq\_sd | Equality |  |
| Int \_mm\_comilt\_sd | Less Than |  |
| \_mm\_comile\_sd | Less Than or Equal |  |
| \_mm\_comigt\_sd | Greater Than |  |
| \_mm\_comige\_sd | Greater Than or Equal |  |
| \_mm\_comineq\_sd | Not Equal |  |
| \_mm\_ucomieq\_sd | Equality |  |
| Int \_mm\_ucomilt\_sd | Less Than |  |
| \_mm\_ucomile\_sd | Less Than or Equal |  |
| \_mm\_ucomigt\_sd | Greater Than |  |
| \_mm\_ucomige\_sd | Greater Than or Equal |  |
| \_mm\_ucomineq\_sd | Not Equal |  |

### 浮点转换操作

原型在emmintrin.h头文件中

进行类型转换，有些类型转换是会丢失精度的

有些情况下的rounding模式是由MXCSR寄存器中的值决定的。默认的rounding模式时趋于最近的值。

Note：c/c++中的rounding模式是截尾的。\_mm\_cvttpd\_epi32和\_mm\_cvttsd\_si32指令是用的截尾模式而不是MXCSR寄存器指定的模式。

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_cvtpd\_ps(\_\_m128d a) | Convert DP FP to SP FP |  |
| \_mm\_cvtps\_pd(\_\_m128 a) | Convert from SP FP to DP FP |  |
| \_mm\_cvtepi32\_pd(\_\_m128i a) | Convert lower integer values to DP FP | 将低位的2个32位有符号转换为DPFP |
| \_mm\_cvtpd\_epi32(\_\_m128d a) | Convert DP FP values to integer values | 将2个DPFP转为32位有符号整型 |
| \_mm\_cvtsd\_si32 | Convert lower DP FP value to integer value | 将低位的DPFP转为1个32位有符号整型 |
| \_mm\_cvtsd\_ss(\_\_m128 a, \_\_m128d b) | Convert lower DP FP value to SP FP | 将低位的DPFP转为SPFP |
| \_mm\_cvtsi32\_sd(\_\_m128d a, int b) | Convert signed integer value to DP FP | 将b的有符号整型转为DPFP |
| \_mm\_cvtss\_sd(\_\_m128d a, \_\_m128 b) | Convert lower SP FP value to DP FP |  |
| \_mm\_cvttpd\_epi32(\_\_m128d a) | Convert DP FP values to signed integers | 将2个DPFP转为32位有符号整型（截尾模式） |
| \_mm\_cvttsd\_si32(\_\_m128d a) | Convert lower DP FP to signed integer | 将低位的DPFP转为32位有符号整型（截尾模式） |
| \_mm\_cvtpd\_pi32(\_\_m128d a) | Convert two DP FP values to signed integer values |  |
| \_mm\_cvttpd\_pi32 | Convert two DP FP values to signed integer values using truncate | 将2个DPFP转为32位有符号整型（截尾模式） |
| \_mm\_cvtpi32\_pd | Convert two signed integer values to DP FP |  |
| \_mm\_cvtsd\_f64 | Extract DP FP value from first vector element |  |

### 浮点加载操作

原型在emmintrin.h头文件中

加载、置位操作和初始化\_\_m128d数据很类似。然而，置位操作有1个double类型的参数，预留给常量的初始化；而加载操作则有1个double类型的指针，用来模仿从内存加载数据的指令。

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_load\_pd(double const\*dp) | Loads two DP FP values | 地址必须是16字节对齐的 |
| \_mm\_load1\_pd(double const\*dp) | Loads a single DP FP value, copying to both elements | 地址不需要16字节对齐 |
| \_mm\_loadr\_pd | Loads two DP FP values in reverse order | 逆序，地址必须是16字节对齐 |
| \_mm\_loadu\_pd | Loads two DP FP values | 地址不需要16字节对齐 |
| \_mm\_load\_sd | Loads a DP FP value, sets upper DP FP to zero | 地址不需要16字节对齐 |
| \_mm\_loadh\_pd(\_\_m128d a, double const\*dp) | Loads a DP FP value as the upper DP FP value of the result | 地址不需要16字节对齐 |
| \_mm\_loadl\_pd | Loads a DP FP value as the lower DP FP value of the result | 地址不需要16字节对齐 |

### 浮点置位操作

原型在emmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_set\_sd(double w) | Sets lower DP FP value to w and upper to zero |  |
| \_mm\_set1\_pd | Sets two DP FP valus to w |  |
| \_mm\_set\_pd(double w, double x) | Sets lower DP FP to x and upper to w |  |
| \_mm\_setr\_pd | Sets lower DP FP to w and upper to x |  |
| \_mm\_setzero\_pd | Sets two DP FP values to zero |  |
| \_mm\_move\_sd( \_\_m128d a, \_\_m128d b) | Sets lower DP FP value to the lower DP FP value of b |  |

### 浮点存储操作

原型在emmintrin.h中

存储操作将数据对齐到地址

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_stream\_pd | Store |  |
| \_mm\_store\_sd(double \*dp, \_\_m128d a) | Stores lower DP FP value of a | 将低位的DPFP存储，地址不需要16字节对齐 |
| \_mm\_store1\_pd | Stores lower DP FP value of a twice | 地址必须16字节对齐 |
| \_mm\_store\_pd | Stores two DP FP values | 地址必须16字节对齐 |
| \_mm\_storeu\_pd | Stores two DP FP values | 地址不需要16字节对齐 |
| \_mm\_storer\_pd | Stores two DP FP values in reverse order | 地址必须是16字节对齐 |
| \_mm\_storeh\_pd(double \*dp, \_\_m128d a) | Stores upper DP FP value of a |  |
| \_mm\_storel\_pd(double \*dp, \_\_m128d a) | Stores lower DP FP value of a |  |

## 整型指令

### 整型算术操作

原型在emmintrin.h中

| **指令** | **操作** | **简要说明** |
| --- | --- | --- |
| \_mm\_add\_epi8(\_\_m128i a, \_\_m128i b) | Addition | a和b中的16个无符号或有符号8位相加 |
| \_mm\_add\_epi16 | Addition | a和b中的8个有符号或无符号16位相加 |
| \_mm\_add\_epi32 | Addition | a和b中的4个有符号或无符号32位相加 |
| \_mm\_add\_si64(\_\_m64 a, \_\_m64 b) | Addition | a和b中的有符号或无符号64位相加 |
| \_mm\_add\_epi64(\_\_m128i a, \_\_m128i b) | Addition | 2个有符号或无符号64位相加 |
| \_mm\_adds\_epi8 | Addition | a和b中的16个有符号8位相加（饱和算术） |
| \_mm\_adds\_epi16 | Addition | a和b中的8个有符号16位相加（饱和算术） |
| \_mm\_adds\_epu8 | Addition | a和b中的16个无符号8位相加（饱和算术） |
| \_mm\_adds\_epu16 | Addition | a和b中的8个无符号16位相加（饱和算术） |
| \_mm\_avg\_epu8 | Computes Average | a和b中的16个无符号8位求均值（round模式） |
| \_mm\_avg\_epu16 | Computes Average | a和b中的8个无符号16位求均值（round模式） |
| \_mm\_madd\_epi16 | Multiplication and Addition | a和b中的8个有符号16位相乘。逐对将有符号的32位相加且打包成4个有符号32位 |
| \_mm\_max\_epi16 | Computes Maxima | 逐对取最大值a和b中的8个有符号16位 |
| \_mm\_max\_epu8 | Computes Maxima | 逐对取最大值a和b中的16个无符号8位 |
| \_mm\_min\_epi16 | Computes Minima | 逐对取最小值a和b中的8个有符号16位 |
| \_mm\_min\_epu8 | Computes Minima | 逐对取最小值a和b中的16个无符号8位 |
| \_mm\_mulhi\_epi16 | Multiplication | 将a和b中的8个有符号16位相乘，且将8个有符号32位的高16比特打包 |
| \_mm\_mulhi\_epu16 | Multiplication | 将a和b中的8个无符号16位相乘，且将8个无符号32位的高16比特打包 |
| \_mm\_mullo\_epi16 | Multiplication | 将a和b中的8个有符号或无符号16位相乘，且将8个有符号或无符号32位的低16比特打包 |
| \_mm\_mul\_su32 (\_\_m64 a, \_\_m64 b) | Multiplication | 将a和b中的低32 位相乘 |
| \_mm\_mul\_epu32 | Multiplication | 将a和b中的2个无符号32位相乘，且将2个无符号64位结果打包 |
| \_mm\_sad\_epu8 | Computes Difference/Adds | 将a和b中的16个无符号8位求差的绝对值，再将低位的8个差和高位的8个差分别求和，然后再将2个无符号16位结果进行打包，放在低的64位和高的64位 |
| \_mm\_sub\_epi8 | Subtraction | 将a和b中的16个有符号或无符号8位求差 |
| \_mm\_sub\_epi16 | Subtraction | 将a和b中的8个有符号或无符号16位求差 |
| \_mm\_sub\_epi32 | Subtraction | 将a和b中的4个有符号或无符号32位求差 |
| \_mm\_sub\_si64 (\_\_m64 a, \_\_m64 b) | Subtraction | 将a和b中的1个有符号或无符号64位求差 |
| \_mm\_sub\_epi64 | Subtraction | 将a和b中的2个有符号或无符号64位求差 |
| \_mm\_subs\_epi8 | Subtraction | 将a和b中的16个有符号8位求差（饱和模式） |
| \_mm\_subs\_epi16 | Subtraction | 将a和b中的8个有符号16位求差（饱和模式） |
| \_mm\_subs\_epu8 | Subtraction | 将a和b中的16个无符号8位求差（饱和模式） |
| \_mm\_subs\_epu16 | Subtraction | 将a和b中的8个无符号16位求差（饱和模式） |

### 整型逻辑操作

原型在emmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_and\_si128(\_\_m128i a, \_\_m128i b) | Computes AND |  |
| \_mm\_andnot\_si128 | Computes AND and NOT |  |
| \_mm\_or\_si128 | Computes OR |  |
| \_mm\_xor\_si128 | Computes XOR |  |

### 整型移位操作

原型在emmintrin.h头文件中

Note:参数count是用于对象所有元素的移位数。

| **指令** | **操作** | **移位类型** | **简要描述** |
| --- | --- | --- | --- |
| \_mm\_slli\_si128 (\_\_m128i a, int imm) | Shift left | Logical | Imm必须是一个立即数，补0 |
| \_mm\_slli\_epi16 (\_\_m128i a, int count) | Shift left | Logical | 8个有符号或者无符号16位数左移，补0 |
| \_mm\_sll\_epi16 (\_\_m128i a, \_\_m128i count) | Shift left | Logical |  |
| \_mm\_slli\_epi32 (\_\_m128i a, int count) | Shift left | Logical |  |
| \_mm\_sll\_epi32 (\_\_m128i a, \_\_m128i count) | Shift left | Logical |  |
| \_mm\_slli\_epi64 (\_\_m128i a, int count) | Shift left | Logical |  |
| \_mm\_sll\_epi64 (\_\_m128i a, \_\_m128i count) | Shift left | Logical |  |
| \_mm\_srai\_epi16 (\_\_m128i a, int count) | Shift right | Arithmetic | 右移8个有符号16位，补符号位 |
| \_mm\_sra\_epi16 (\_\_m128i a, \_\_m128i count) | Shift right | Arithmetic | 右移8个有符号16位，补符号位 |
| \_mm\_srai\_epi32 (\_\_m128i a, int count) | Shift right | Arithmetic | 右移4个有符号32位，补符号位 |
| \_mm\_sra\_epi32 (\_\_m128i a, \_\_m128i count) | Shift right | Arithmetic | 右移4个有符号32位，补符号位 |
| \_mm\_srli\_si128 (\_\_m128i a, int imm) | Shift right | Logical |  |
| \_mm\_srli\_epi16 (\_\_m128i a, int count) | Shift right | Logical | 右移8个有符号或者无符号16位数，补0 |
| \_mm\_srl\_epi16 (\_\_m128i a, \_\_m128i count) | Shift right | Logical | 右移8个有符号或者无符号16位数，补0 |
| \_mm\_srli\_epi32 (\_\_m128i a, int count) | Shift right | Logical | 右移4个有符号或无符号32位数，补0 |
| \_mm\_srl\_epi32 (\_\_m128i a, \_\_m128i count) | Shift right | Logical | 右移4个有符号或无符号32位数，补0 |
| \_mm\_srli\_epi64 (\_\_m128i a, int count) | Shift right | Logical | 右移2个有符号或无符号64位数，补0 |
| \_mm\_srl\_epi64 (\_\_m128i a, \_\_m128i count) | Shift right | Logical | 右移2个有符号或无符号64位数，补0 |

### 整型比较操作

原型在emmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_cmpeq\_epi8 (\_\_m128i a, \_\_m128i b) | Equality | 比较16个有符号或无符号的8位数 |
| \_mm\_cmpeq\_epi16 | Equality | 比较8个有符号或无符号的16位数 |
| \_mm\_cmpeq\_epi32 | Equality | 比较4个有符号或无符号的32位数 |
| \_mm\_cmpgt\_epi8 | Greater Than | 比较16个有符号8位数 |
| \_mm\_cmpgt\_epi16 | Greater Than | 比较8个有符号16位 |
| \_mm\_cmpgt\_epi32 | Greater Than | 比较4个有符号32位 |
| \_mm\_cmplt\_epi8 | Less Than |  |
| \_mm\_cmplt\_epi16 | Less Than |  |
| \_mm\_cmplt\_epi32 | Less Than |  |

### 整型转换操作

原型在emmintrin.h头文件中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_\_m128d \_mm\_cvtsi64\_sd (\_\_m128d a, \_\_int64 b) | Convert and pass through | 转换1个有符号64位数到DPFP |
| \_\_int64 \_mm\_cvtsd\_si64 (\_\_m128d a) | Convert according to rounding | 转换低位的DPFP到1个64位有符号整数，round模式 |
| \_\_int64 \_mm\_cvttsd\_si64 (\_\_m128d a) | Convert using truncation | 转换低位的DPFP到1个64位有符号整数，截尾模式 |
| \_mm\_cvtepi32\_ps (\_\_m128i a) | Convert to SP FP | 转换4个有符号32位整数 |
| \_mm\_cvtps\_epi32 | Convert from SP FP |  |
| \_mm\_cvttps\_epi32 | Convert from SP FP using truncate | 转换4个SPFP为有符号的32位整数，截尾模式 |

### 整型移动操作

原型在emmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_cvtsi32\_si128 (int a) | Move and zero |  |
| \_mm\_cvtsi64\_si128(\_\_int64 a) | Move and zero |  |
| \_mm\_cvtsi128\_si32(\_\_m128i a) | Move lowest 32 bits |  |
| \_mm\_cvtsi128\_si64 | Move lowest 64 bits |  |

### 整型加载操作

原型在emmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_load\_si128(\_\_m128i const\*p) | Load | 地址p必须是16字节对齐 |
| \_mm\_loadu\_si128(\_\_m128i const\*p) | Load | 地址p不需要16字节对齐 |
| \_mm\_loadl\_epi64 | Load and zero |  |

### 整型置位操作

原型在emmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_set\_epi64(\_\_m64 q1, \_\_m64 q0) | Set two integer values | 有符号 |
| \_mm\_set\_epi32(int i3, int i2, int i1, int i0) | Set four integer values | 有符号 |
| \_mm\_set\_epi16(short w7, short w6, short w5, short w4, short w3, short w2, short w1, short w0) | Set eight integer values | 有符号 |
| \_mm\_set\_epi8(char b15, char b14, char b13, char b12, char b11, char b10, char b9, char b8, char b7, char b6, char b5, char b4, char b3, char b2, char b1, char b0) | Set sixteen integer values | 有符号 |
| \_mm\_set1\_epi64(\_\_m64 q) | Set two integer values |  |
| \_mm\_set1\_epi32(int i) | Set four integer values | 有符号 |
| \_mm\_set1\_epi16(short w) | Set eight integer values |  |
| \_mm\_set1\_epi8(char b) | Set sixteen integer values |  |
| \_mm\_setr\_epi64(\_\_m64 q0, \_\_m64 q1) | Set two integer values in reverse order |  |
| \_mm\_setr\_epi32(int i0, int i1, int i2, int i3) | Set four integer values in reverse order |  |
| \_mm\_setr\_epi16(short w0, short w1, short w2, short w3, short w4, short w5, short w6, short w7) | Set eight integer values in reverse order |  |
| \_mm\_setr\_epi8(char b15, char b14, char b13, char b12, char b11, char b10, char b9, char b8, char b7, char b6, char b5, char b4, char b3, char b2, char b1, char b0) | Set sixteen integer values in reverse order |  |
| \_mm\_setzero\_si128 | Set to zero |  |

### 整型存储操作

原型在emmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_stream\_si128 (\_\_m128i \*p, \_\_m128i a) | Store | 不破坏caches将数据存入p，p必须是16字节对齐 |
| \_mm\_stream\_si32 (int \*p, int a) | Store |  |
| \_mm\_store\_si128 (\_\_m128i \*p, \_\_m128i b) | Store | p必须是16字节对齐 |
| \_mm\_storeu\_si128 | Store | p不需要16字节对齐 |
| \_mm\_maskmoveu\_si128 (\_\_m128i d, \_\_m128i n, char \*p) | Conditional store |  |
| \_mm\_storel\_epi64 | Store lowest |  |

## 其它函数和指令

### 缓存支持指令

原型在emmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_stream\_pd(double \*p, \_\_m128d a) | Store | p必须是16字节对齐 |
| \_mm\_stream\_si128(\_\_m128i \*p, \_\_m128i a) | Store | p必须16字节对齐 |
| \_mm\_stream\_si32(int \*p, int a) | Store |  |
| \_mm\_stream\_si64(\_\_int64 \*p, \_\_int64 a) | Store |  |
| \_mm\_clflush(void const\*p) | Flush |  |
| \_mm\_lfence | Guarantee visibility |  |
| \_mm\_mfence | Guarantee visibility |  |

### 混杂指令

原型在emmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_packs\_epi16(\_\_m128i a, \_\_m128i b) | Packed Saturation | 打包16个有符号16位数到有符号8位，饱和模式 |
| \_mm\_packs\_epi32 | Packed Saturation | 打包8个有符号32位数到有符号16位，饱和模式 |
| \_mm\_packus\_epi16 | Packed Saturation | 打包16个有符号16位数到无符号8位，饱和模式 |
| \_mm\_extract\_epi16 (\_\_m128i a, int imm) | Extraction | 提取 |
| \_mm\_insert\_epi16 (\_\_m128i a, int b, int imm) | Insertion | 插入 |
| \_mm\_movemask\_epi8 (\_\_m128i a) | Mask Creation |  |
| \_mm\_shuffle\_epi32 (\_\_m128i a, int imm) | Shuffle | 按照imm的说明拖拽4个有符号或无符号32位数 |
| \_mm\_shufflehi\_epi16 | Shuffle | 按照imm的说明拖拽高位的4个有符号或无符号16位数 |
| \_mm\_shufflelo\_epi16 | Shuffle | 按照imm的说明拖拽低位的4个有符号或无符号16位数 |
| \_mm\_unpackhi\_epi8 (\_\_m128i a, \_\_m128i b) | Interleave |  |
| \_mm\_unpackhi\_epi16 | Interleave |  |
| \_mm\_unpackhi\_epi32 | Interleave |  |
| \_mm\_unpackhi\_epi64 | Interleave |  |
| \_mm\_unpacklo\_epi8 | Interleave |  |
| \_mm\_unpacklo\_epi16 | Interleave |  |
| \_mm\_unpacklo\_epi32 | Interleave |  |
| \_mm\_unpacklo\_epi64 | Interleave |  |
| \_\_m64 \_mm\_movepi64\_pi64 (\_\_m128i a) | Move | 返回低64位 |
| \_\_m128i \_mm\_movpi64\_epi64 (\_\_m64 a) | Move |  |
| \_\_m128i \_mm\_move\_epi64 (\_\_m128i a) | Move |  |
| \_\_m128d \_mm\_unpackhi\_pd (\_\_m128d a, \_\_m128d b) | Interleave | 交织DPFP |
| \_mm\_unpacklo\_pd | Interleave |  |
| int \_mm\_movemask\_pd (\_\_m128d a) | Create mask | 从a的两个DPFP的标志位创造1个两bit位的掩码 |
| \_\_m128d \_mm\_shuffle\_pd (\_\_m128d a, \_\_m128d b, int i) | Select values |  |

### 类型转换指令

支持单精度，双精度和整型向量的转换，这些指令不会改变值；在不改变值的情况下转换数据的类型。

\_\_m128 \_mm\_castpd\_ps(\_\_m128d in);

\_\_m128i \_mm\_castpd\_si128(\_\_m128d in);

\_\_m128d \_mm\_castps\_pd(\_\_m128 in);

\_\_m128i \_mm\_castps\_si128(\_\_m128 in);

\_\_m128 \_mm\_castsi128\_ps(\_\_m128i in);

\_\_m128d \_mm\_castsi128\_pd(\_\_m128i in);

### 暂停指令

原型在xmmintrin.h中

void \_mm\_pause(void)

### 重排宏

# SSE3的固有指令

## 综述

## 整型向量指令

原型在pmmintrin.h中

\_\_m128i \_mm\_lddqu\_si128(\_\_m128i const \*p)

加载一个不对齐的128位数据。和movdqu的区别就是在有些情况下它的性能更好。但是如果要读的内存刚刚被写过，它的性能将不如movdqu。



## 单精度浮点型向量指令

原型在pmmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_\_m128 \_mm\_addsub\_ps(\_\_m128 a, \_\_m128 b) | Subtract and add |  |
| \_mm\_hadd\_ps | Add |  |
| \_mm\_hsub\_ps | Subtracts |  |
| \_mm\_movehdup\_ps | Duplicates |  |
| \_mm\_moveldup\_ps | Duplicates |  |

## 双精度浮点型向量指令

原型在pmmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_\_m128d \_mm\_addsub\_pd (\_\_m128d a, \_\_m128d b) | Subtract and add |  |
| \_mm\_hadd\_pd | Add |  |
| \_mm\_hsub\_pd | Subtract |  |
| \_mm\_loaddup\_pd (double const \* dp) | Duplicate |  |
| \_mm\_movedup\_pd (\_\_m128d a) | Duplicate |  |

## 宏函数

原型在pmmintrin.h中

\_MM\_SET\_DENORMALS\_ZERO\_MODE(x)

Macro arguments: one of \_\_MM\_DENORMALS\_ZERO\_ON, \_MM\_DENORMALS\_ZERO\_OFF This causes "denormals are zero" mode to be turned on or off by setting the appropriate bit of the control register.

\_MM\_GET\_DENORMALS\_ZERO\_MODE()

No arguments. This returns the current value of the denormals are zero mode bit of the control register.

## 混杂指令

原型在pmmintrin.h中

extern void \_mm\_monitor(void const \*p, unsigned extensions, unsigned hints);

extern void \_mm\_mwait(unsigned extensions, unsigned hints);

# SSE3补充的固有指令

## 综述

函数原型在tmmintrin.h中。也可以用ia32intrin.h头文件。

## 加法指令

水平加法

extern \_\_m128i \_mm\_hadd\_epi16 (\_\_m128i a, \_\_m128i b);

水平打包有符号word相加。

Interpreting a, b, and r as arrays of 16-bit signed integers:

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

r[i] = a[2\*i] + a[2i+1];

r[i+4] = b[2\*i] + b[2\*i+1];

}

extern \_\_m128i \_mm\_hadd\_epi32 (\_\_m128i a, \_\_m128i b);

水平打包有符号dword相加。

Interpreting a, b, and r as arrays of 32-bit signed integers:

for (i = 0; i < 2; i++) {

r[i] = a[2\*i] + a[2i+1];

r[i+2] = b[2\*i] + b[2\*i+1];

}

extern \_\_m128i \_mm\_hadds\_epi16 (\_\_m128i a, \_\_m128i b);

水平打包有符号word相加，饱和模式。

Interpreting a, b, and r as arrays of 16-bit signed integers:

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

r[i] = signed\_saturate\_to\_word(a[2\*i] + a[2i+1]);

r[i+4] = signed\_saturate\_to\_word(b[2\*i] + b[2\*i+1]);

}

extern \_\_m64 \_mm\_hadd\_pi16 (\_\_m64 a, \_\_m64 b);

水平打包有符号word相加。

Interpreting a, b, and r as arrays of 16-bit signed integers:

for (i = 0; i < 2; i++) {

r[i] = a[2\*i] + a[2i+1];

r[i+2] = b[2\*i] + b[2\*i+1];

}

extern \_\_m64 \_mm\_hadd\_pi32 (\_\_m64 a, \_\_m64 b);

水平打包有符号dword相加

Interpreting a, b, and r as arrays of 32-bit signed integers:

r[0] = a[1] + a[0];

r[1] = b[1] + b[0];

extern \_\_m64 \_mm\_hadds\_pi16 (\_\_m64 a, \_\_m64 b);

水平打包有符号word相加，饱和模式。

Interpreting a, b, and r as arrays of 16-bit signed integers:

for (i = 0; i < 2; i++) {

r[i] = signed\_saturate\_to\_word(a[2\*i] + a[2i+1]);

r[i+2] = signed\_saturate\_to\_word(b[2\*i] + b[2\*i+1]);

## 减法指令

水平减法

extern \_\_m128i \_mm\_hsub\_epi16 (\_\_m128i a, \_\_m128i b);

水平打包有符号word相减。

Interpreting a, b, and r as arrays of 16-bit signed integers:

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

r[i] = a[2\*i] - a[2i+1];

r[i+4] = b[2\*i] - b[2\*i+1];

}

extern \_\_m128i \_mm\_hsub\_epi32 (\_\_m128i a, \_\_m128i b);

水平打包有符号dword相减。

Interpreting a, b, and r as arrays of 32-bit signed integers:

for (i = 0; i < 2; i++) {

r[i] = a[2\*i] - a[2i+1];

r[i+2] = b[2\*i] - b[2\*i+1];

}

extern \_\_m128i \_mm\_hsubs\_epi16 (\_\_m128i a, \_\_m128i b);

水平打包有符号word相减，饱和模式

Interpreting a, b, and r as arrays of 16-bit signed integers:

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

r[i] = signed\_saturate\_to\_word(a[2\*i] - a[2i+1]);

r[i+4] = signed\_saturate\_to\_word(b[2\*i] - b[2\*i+1]);

}

extern \_\_m64 \_mm\_hsub\_pi16 (\_\_m64 a, \_\_m64 b);

水平打包有符号word相减。

Interpreting a, b, and r as arrays of 16-bit signed integers:

for (i = 0; i < 2; i++) {

r[i] = a[2\*i] - a[2i+1];

r[i+2] = b[2\*i] - b[2\*i+1];

}

extern \_\_m64 \_mm\_hsub\_pi32 (\_\_m64 a, \_\_m64 b);

水平打包有符号dword相减。

Interpreting a, b, and r as arrays of 32-bit signed integers:

r[0] = a[0] - a[1];

r[1] = b[0] - b[1];

extern \_\_m64 \_mm\_hsubs\_pi16 (\_\_m64 a, \_\_m64 b);

水平打包有符号word相减，饱和模式。

Interpreting a, b, and r as arrays of 16-bit signed integers:

for (i = 0; i < 2; i++) {

r[i] = signed\_saturate\_to\_word(a[2\*i] - a[2i+1]);

r[i+2] = signed\_saturate\_to\_word(b[2\*i] - b[2\*i+1]);

}

## 乘法指令

extern \_\_m128i \_mm\_maddubs\_epi16 (\_\_m128i a, \_\_m128i b);

Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed words.

Interpreting a as array of unsigned 8-bit integers, b as arrays of signed 8-bit integers, and r as arrays of 16-bit signed integers:

for (i = 0; i < 8; i++) {

r[i] = signed\_saturate\_to\_word(a[2\*i+1] \* b[2\*i+1] + a[2\*i]\*b[2\*i]);

}

extern \_\_m64 \_mm\_maddubs\_pi16 (\_\_m64 a, \_\_m64 b);

Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed words.

Interpreting a as array of unsigned 8-bit integers, b as arrays of signed 8-bit integers, and r as arrays of 16-bit signed integers:

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

r[i] = signed\_saturate\_to\_word(a[2\*i+1] \* b[2\*i+1] + a[2\*i]\*b[2\*i]);

}

extern \_\_m128i \_mm\_mulhrs\_epi16 (\_\_m128i a, \_\_m128i b);

Multiply signed words, scale and round signed dwords, pack high 16-bits.

Interpreting a, b, and r as arrays of signed 16-bit integers:

for (i = 0; i < 8; i++) {

r[i] = (( (int32)((a[i] \* b[i]) >> 14) + 1) >> 1) & 0xFFFF;

}

extern \_\_m64 \_mm\_mulhrs\_pi16 (\_\_m64 a, \_\_m64 b);

Multiply signed words, scale and round signed dwords, pack high 16-bits.

Interpreting a, b, and r as arrays of signed 16-bit integers:

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

r[i] = (( (int32)((a[i] \* b[i]) >> 14) + 1) >> 1) & 0xFFFF;

}

## 绝对值指令

extern \_\_m128i \_mm\_abs\_epi8 (\_\_m128i a);

Compute absolute value of signed bytes.

Interpreting a and r as arrays of signed 8-bit integers:

for (i = 0; i < 16; i++) {

r[i] = abs(a[i]);

}

extern \_\_m128i \_mm\_abs\_epi16 (\_\_m128i a);

Compute absolute value of signed words.

Interpreting a and r as arrays of signed 16-bit integers:

for (i = 0; i < 8; i++) {

r[i] = abs(a[i]);

}

extern \_\_m128i \_mm\_abs\_epi32 (\_\_m128i a);

Compute absolute value of signed dwords.

Interpreting a and r as arrays of signed 32-bit integers:

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

r[i] = abs(a[i]);

}

extern \_\_m64 \_mm\_abs\_pi8 (\_\_m64 a);

Compute absolute value of signed bytes.

Interpreting a and r as arrays of signed 8-bit integers:

for (i = 0; i < 8; i++) {

r[i] = abs(a[i]);

}

extern \_\_m64 \_mm\_abs\_pi16 (\_\_m64 a);

Compute absolute value of signed words.

Interpreting a and r as arrays of signed 16-bit integers:

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

r[i] = abs(a[i]);

}

extern \_\_m64 \_mm\_abs\_pi32 (\_\_m64 a);

Compute absolute value of signed dwords.

Interpreting a and r as arrays of signed 32-bit integers:

for (i = 0; i < 2; i++) {

r[i] = abs(a[i]);

}

## 重排指令

extern \_\_m128i \_mm\_shuffle\_epi8 (\_\_m128i a, \_\_m128i b);

Shuffle bytes from a according to contents of b.

Interpreting a, b, and r as arrays of unsigned 8-bit integers:

for (i = 0; i < 16; i++){

 if (b[i] & 0x80){

  r[i] = 0;

 }

 else

 {

  r[i] = a[b[i] & 0x0F];

 }

}

extern \_\_m64 \_mm\_shuffle\_pi8 (\_\_m64 a, \_\_m64 b);

Shuffle bytes from a according to contents of b.

Interpreting a, b, and r as arrays of unsigned 8-bit integers:

for (i = 0; i < 8; i++){

 if (b[i] & 0x80){

  r[i] = 0;

 }

 else

 {

  r[i] = a[b[i] & 0x07];

 }

}

## 连接指令

extern \_\_m128i \_mm\_alignr\_epi8 (\_\_m128i a, \_\_m128i b, int n);

Concatenate a and b, extract byte-aligned result shifted to the right by n.

Interpreting t1 as 256-bit unsigned integer, a, b, and r as 128-bit unsigned integers:

t1[255:128] = a;

t1[127:0] = b;

t1[255:0] = t1[255:0] >> (8 \* n); // unsigned shift

r[127:0] = t1[127:0];

extern \_\_m64 \_mm\_alignr\_pi8 (\_\_m64 a, \_\_m64 b, int n);

Concatenate a and b, extract byte-aligned result shifted to the right by n.

Interpreting t1 as 127-bit unsigned integer, a, b and r as 64-bit unsigned integers:

t1[127:64] = a;

t1[63:0] = b;

t1[127:0] = t1[127:0] >> (8 \* n); // unsigned shift

r[63:0] = t1[63:0];

负指令

extern \_\_m128i \_mm\_sign\_epi8 (\_\_m128i a, \_\_m128i b);

Negate packed bytes in a if corresponding sign in b is less than zero.

Interpreting a, b, and r as arrays of signed 8-bit integers:

for (i = 0; i < 16; i++){

 if (b[i] < 0){

  r[i] = -a[i];

 }

 else

 if (b[i] == 0){

  r[i] = 0;

 }

 else

 {

  r[i] = a[i];

 }

}

extern \_\_m128i \_mm\_sign\_epi16 (\_\_m128i a, \_\_m128i b);

Negate packed words in a if corresponding sign in b is less than zero.

Interpreting a, b, and r as arrays of signed 16-bit integers:

for (i = 0; i < 8; i++){

 if (b[i] < 0){

  r[i] = -a[i];

 }

 else

 if (b[i] == 0){

  r[i] = 0;

 }

 else

 {

  r[i] = a[i];

 }

}

extern \_\_m128i \_mm\_sign\_epi32 (\_\_m128i a, \_\_m128i b);

Negate packed dwords in a if corresponding sign in b is less than zero.

Interpreting a, b, and r as arrays of signed 32-bit integers:

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

 if (b[i] < 0){

  r[i] = -a[i];

 }

 else

 if (b[i] == 0){

  r[i] = 0;

 }

 else

 {

  r[i] = a[i];

 }

}

extern \_\_m64 \_mm\_sign\_pi8 (\_\_m64 a, \_\_m64 b);

Negate packed bytes in a if corresponding sign in b is less than zero.

Interpreting a, b, and r as arrays of signed 8-bit integers:

for (i = 0; i < 16; i++){

 if (b[i] < 0){

  r[i] = -a[i];

 }

 else

 if (b[i] == 0){

  r[i] = 0;

 }

 else

 {

  r[i] = a[i];

 }

}

extern \_\_m64 \_mm\_sign\_pi16 (\_\_m64 a, \_\_m64 b);

Negate packed words in a if corresponding sign in b is less than zero.

Interpreting a, b, and r as arrays of signed 16-bit integers:

for (i = 0; i < 8; i++){

 if (b[i] < 0){

  r[i] = -a[i];

 }

 else

 if (b[i] == 0){

  r[i] = 0;

 }

 else

 {

  r[i] = a[i];

 }

}

extern \_\_m64 \_mm\_sign\_pi32 (\_\_m64 a, \_\_m64 b);

Negate packed dwords in a if corresponding sign in b is less than zero.

Interpreting a, b, and r as arrays of signed 32-bit integers:

for (i = 0; i < 2; i++){

 if (b[i] < 0){

  r[i] = -a[i];

 }

 else

 if (b[i] == 0){

  r[i] = 0;

 }

 else

 {

  r[i] = a[i];

 }

}

# SSE4的固有指令

## 综述

## 向量化编译器和媒体加速器

### 综述：SSE4向量化编译器和媒体加速器

### 打包混合指令

指令将多重操作放入一个指令。混合有条件的将源中的内容复制到对应的地方。

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_\_m128 \_mm\_blend\_ps(\_\_m128 v1, \_\_m128 v2, const int mask) | Selects float single precision data from 2 sources using constant mask |  |
| \_\_m128d \_mm\_blend\_pd(\_\_m128d v1, \_\_m128d v2, const int mask) | Selects float double precision data from 2 sources using constant mask |  |
| \_\_m128 \_mm\_blendv\_ps(\_\_m128 v1, \_\_m128 v2, \_\_m128 v3) | Selects float single precision data from 2 sources using variable mask |  |
| \_\_m128d \_mm\_blendv\_pd(\_\_m128d v1, \_\_m128d v2, \_\_m128d v3) | Selects float double precision data from 2 sources using variable mask |  |
| \_\_m128i \_mm\_blendv\_epi8(\_\_m128i v1, \_\_m128i v2, \_\_m128i mask) | Selects integer bytes from 2 sources using variable mask |  |
| \_\_m128i \_mm\_blend\_epi16(\_\_m128i v1, \_\_m128i v2, const int mask) | Selects integer words from 2 sources using constant mask |  |

### 浮点型点积指令

指令支持浮点型和double型的点积

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_mm\_dp\_pd ( \_\_m128d a, \_\_m128d b, const int mask) | Double precision dot product | 指令计算double型的点积 |
| \_mm\_dp\_ps ( \_\_m128 a, \_\_m128 b, const int mask) | Single precision dot product | 指令计算单精度型的点积 |

### 打包格式化转换指令

指令将打包的整型转为0扩展或者符号位扩展的整型

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_\_m128i \_mm\_cvtepi8\_epi32(\_\_m128i a) | Sign extend 4 bytes into 4 double words |  |
| \_\_m128i \_mm\_cvtepi8\_epi64 (\_\_m128i a) | Sign extend 2 bytes into 2 quad words |  |
| \_\_m128i \_mm\_cvtepi8\_epi16(\_\_m128i a) | Sign extend 8 bytes into 8 words |  |
| \_\_m128i \_mm\_cvtepi32\_epi64(\_\_m128i a) | Sign extend 2 double words into 2 quad words |  |
| \_\_m128i \_mm\_cvtepi16\_epi32(\_\_m128i a) | Sign extend 4 words into 4 double words |  |
| \_\_m128i \_mm\_cvtepi16\_epi64(\_\_m128i a) | Sign extend 2 words into 2 quad words |  |
| \_\_m128i \_mm\_cvtepu8\_epi32(\_\_m128i a) | Zero extend 4 bytes into 4 double words |  |
| \_\_m128i \_mm\_cvtepu8\_epi64(\_\_m128i a) | Zero extend 2 bytes into 2 quad words |  |
| \_\_m128i \_mm\_cvtepu8\_epi16(\_\_m128i a) | Zero extend 8 bytes into 8 word |  |
| \_\_m128i \_mm\_cvtepu32\_epi64(\_\_m128i a) | Zero extend 2 double words into 2 quad words |  |
| \_\_m128i \_mm\_cvtepu16\_epi32(\_\_m128i a) | Zero extend 4 words into 4 double words |  |
| \_\_m128i \_mm\_cvtepu16\_epi64(\_\_m128i a) | Zero extend 2 words into 2 quad words |  |

### 打包整型min/max指令

指令比较目标和源中的打包整型，返回最小值或者最大值

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_\_m128i \_mm\_max\_epi8( \_\_m128i a, \_\_m128i b) | Calculates maximum of signed packed integer bytes |  |
| \_\_m128i \_mm\_max\_epi32( \_\_m128i a, \_\_m128i b) | Calculates maximum of signed packed integer double words |  |
| \_\_m128i \_mm\_max\_epu32( \_\_m128i a, \_\_m128i b) | Calculates maximum of unsigned packed integer double words |  |
| \_\_m128i \_mm\_max\_epu16( \_\_m128i a, \_\_m128i b) | Calculates maximum of unsigned packed integer words |  |
| \_\_m128i \_mm\_min\_epi8( \_\_m128i a, \_\_m128i b) | Calculates minimum of signed packed integer bytes |  |
| \_\_m128i \_mm\_min\_epi32( \_\_m128i a, \_\_m128i b) | Calculates minimum of signed packed integer double words |  |
| \_\_m128i \_mm\_min\_epu32( \_\_m128i a, \_\_m128i b) | Calculates minimum of unsigned packed integer double words |  |
| \_\_m128i \_mm\_min\_epu16( \_\_m128i a, \_\_m128i b) | Calculates minimum of unsigned packed integer words |  |

### 浮点型舍入指令

指令覆盖了标量和打包的单精度及双精度浮点操作数

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_\_m128d \_mm\_round\_pd(\_\_m128d s1, int iRoundMode)  \_\_m128d \_mm\_floor\_pd(\_\_m128d s1)  \_\_m128d \_mm\_ceil\_pd(\_\_m128d s1) | Packed float double precision rounding |  |
| \_\_m128 \_mm\_round\_ps(\_\_m128 s1, int iRoundMode)  \_\_m128 \_mm\_floor\_ps(\_\_m128 s1)  \_\_m128 \_mm\_ceil\_ps(\_\_m128 s1) | Packed float single precision rounding |  |
| \_\_m128d \_mm\_round\_sd(\_\_m128d dst, \_\_m128d s1, int iRoundMode)  \_\_m128d \_mm\_floor\_sd(\_\_m128d dst, \_\_m128d s1)  \_\_m128d \_mm\_ceil\_sd(\_\_m128d dst, \_\_m128d s1) | Single float double precision rounding |  |
| \_\_m128 \_mm\_round\_ss(\_\_m128 dst, \_\_m128d s1, int iRoundMode)  \_\_m128 \_mm\_floor\_ss(\_\_m128d dst, \_\_m128 s1)  \_\_m128 \_mm\_ceil\_ss(\_\_m128d dst, \_\_m128 s1) | Single float single precision rounding |  |

### DWORD乘法指令

DWORD乘法指令是为了有助于标量。它允许4个32位乘32位同时进行。

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_\_m128i \_mm\_mul\_epi32( \_\_m128i a, \_\_m128i b) | Packed integer 32-bit multiplication of 2 low pairs of operands producing two 64-bit results |  |
| \_\_m128i \_mm\_mullo\_epi32( \_\_m128i a, \_\_m128i b) | Packed integer 32-bit multiplication with truncation of upper halves of results |  |

### 寄存器插入/提取指令

指令能够在通用寄存器和xmm寄存器之间插入和提取数据

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| \_\_m128 \_mm\_insert\_ps(\_\_m128 dst, \_\_m128 src, const int ndx) | Insert single precision float into packed single precision array element selected by index |  |
| int \_mm\_extract\_ps(\_\_m128 src, const int ndx) | Extract single precision float from packed single precision array element selected by index |  |
| int \_mm\_extract\_epi8(\_\_m128i src, const int ndx) | Extract integer byte from packed integer array element selected by index |  |
| int \_mm\_extract\_epi32(\_\_m128i src, const int ndx) | Extract integer double word from packed integer array element selected by index |  |
| \_\_int64 \_mm\_extract\_epi64(\_\_m128i src, const int ndx) | Extract integer quad word from packed integer array element selected by index |  |
| int \_mm\_extract\_epi16(\_\_m128i src, int ndx) | Extract integer word from packed integer array element selected by index |  |
| \_\_m128i \_mm\_insert\_epi8(\_\_m128i s1, int s2, const int ndx) | Insert integer byte into packed integer array element selected by index |  |
| \_\_m128i \_mm\_insert\_epi32(\_\_m128i s1, int s2, const int ndx) | Insert integer double word into packed integer array element selected by index |  |
| \_\_m128i \_mm\_insert\_epi64(\_\_m128i s2, int s, const int ndx) | Insert integer quad word into packed integer array element selected by index |  |

### 测试指令

打包的128位整型比较

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| Int \_mm\_testc\_si128(\_\_m128i s1, \_\_m128i s2) | Check for all ones in specified bits of a 128-bit value | 如果s1和s2比特位的与都是0就返回1，否则返回0 |
| \_mm\_testz\_si128 | Check for all zeros in specified bits of a 128-bit value | Returns 1 if the bitwise AND of s2 ANDNOT of s1 is all ones, else returns 0 |
| \_mm\_testnzc\_si128 | Check for at least one zero and at least one one in specified bits of a 128-bit value | (!\_mm\_testz) && (!\_mm\_testc) |

### 打包DWORD到无符号WORD指令

\_\_m128i \_mm\_packus\_epi32(\_\_m128i m1, \_\_m128i m2)

将8个打包有符号DWORD转为8个打包无符号WORD，当溢出时进行无符号饱和。

### 打包等于比较指令

\_\_m128i \_mm\_cmpeq\_epi64(\_\_m128i a, \_\_m128i b)

比较打包64位整型的相等性。

### 可缓存性支持指令

extern \_\_m128i \_mm\_stream\_load\_si128(\_\_m128i\* v1)

v1必须16字节对齐

## 高效加速的字符串和文本处理器

### 综述

### 打包比较指令

| **指令** | **操作** | **简要操作** |
| --- | --- | --- |
| Int \_mm\_cmpestri(\_\_m128i src1, int len1, \_\_m128i src2, int len2, const int mode) | Packed comparison, generates index | 指定长度的打包比较，生成一个索引并且存放在ECX内 |
| \_\_m128i \_mm\_cmpestrm(\_\_m128i src1, int len1, \_\_m128i src2, int len2, const int mode) | Packed comparison, generates mask | 指定长度的打包比较，生成一个掩码并且存放在XMM0内 |
| int \_mm\_cmpistri (\_\_m128i src1, \_\_m128i src2, const int mode) | Packed comparison, generates index | 含蓄长度的打包比较，生成一个索引并且存放在ECX内 |
| \_\_m128i \_mm\_cmpistrm (\_\_m128i src1, \_\_m128i src2, const int mode) | Packed comparison, generates mask | 含蓄长度的打包比较，生成一个掩码并且存放在XMM0内 |
| int \_mm\_cmpestrz (\_\_m128i src1, int len1, \_\_m128i src2, int len2, const int mode) | Packed comparison | 指定长度的打包比较，如果Zflag = 1，返回1，否则返回0 |
| int \_mm\_cmpestrc (\_\_m128i src1, int len1, \_\_m128i src2, int len2, const int mode) | Packed comparison | 指定长度的打包比较，如果Cflag = 1，返回1，否则返回0 |
| Int \_mm\_cmpestrs (\_\_m128i src1, int len1, \_\_m128i src2, int len2, const int mode) | Packed comparison | 指定长度的打包比较，如果Sflag = 1，返回1，否则返回0 |
| \_mm\_cmpestro | Packed comparison | 指定长度的打包比较，如果Oflag = 1，返回1，否则返回0 |
| \_mm\_cmpestra | Packed comparison |  |
| \_mm\_cmpistrz | Packed comparison |  |
| \_mm\_cmpistrc | Packed comparison |  |
| \_mm\_cmpistrs | Packed comparison |  |
| \_mm\_cmpistro | Packed comparison |  |
| \_mm\_cmpistra | Packed comparison |  |

### 应用定向加速器指令

原型在nmmintrin.h中

| **指令** | **操作** | **简要描述** |
| --- | --- | --- |
| int \_mm\_popcnt\_u32(unsigned int v) | Counts number of set bits in a data operation |  |
| int \_mm\_popcnt\_u64(unsigned \_\_int64 v) | Counts number of set bits in a data operation |  |
| unsigned int \_mm\_crc32\_u8(unsigned int crc, unsigned char v) | Accumulates cyclic redundancy check |  |
| unsigned int \_mm\_crc32\_u16(unsigned int crc, unsigned short v) | Performs cyclic redundancy check |  |
| unsigned int \_mm\_crc32\_u32(unsigned int crc, unsigned int v) | Performs cyclic redundancy check |  |
| unsigned \_\_int64 \_mm\_crc32\_u64(unsigned \_\_int64 crc, unsigned \_\_int64 v) | Performs cyclic redundancy check |  |

# 适用所有Intel架构的固有指令

### 综述

### 整型算术指令

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

Note:在旋转结果上通过一个常量的移位效率更高

### 浮点型指令

| **指令** | **描述** |
| --- | --- |
| double fabs(double) | Returns the absolute value of a floating-point value. |
| double log(double) | Returns the natural logarithm ln(x), x>0, with double precison. |
| float logf(float) | Returns the natural logarithm ln(x), x>0, with single precison. |
| double log10(double) | Returns the base 10 logarithm log10(x), x>0, with double precison. |
| float log10f(float) | Returns the base 10 logarithm log10(x), x>0, with single precison. |
| double exp(double) | Returns the exponential function with double precison. |
| float expf(float) | Returns the exponential function with single precison. |
| double pow(double, double) | Returns the value of x to the power y with double precison. |
| float powf(float, float) | Returns the value of x to the power y with single precison. |
| double sin(double) | Returns the sine of x with double precison. |
| float sinf(float) | Returns the sine of x with single precison. |
| double cos(double) | Returns the cosine of x with double precison. |
| float cosf(float) | Returns the cosine of x with single precison. |
| double tan(double) | Returns the tangent of x with double precison. |
| float tanf(float) | Returns the tangent of x with single precison. |
| double acos(double) | Returns the inverse cosine of x with double precison |
| float acosf(float) | Returns the inverse cosine of x with single precison |
| double acosh(double) | Compute the inverse hyperbolic cosine of the argument with double precison. |
| float acoshf(float) | Compute the inverse hyperbolic cosine of the argument with single precison. |
| double asin(double) | Compute inverse sine of the argument with double precison. |
| float asinf(float) | Compute inverse sine of the argument with single precison. |
| double asinh(double) | Compute inverse hyperbolic sine of the argument with double precison. |
| float asinhf(float) | Compute inverse hyperbolic sine of the argument with single precison. |
| double atan(double) | Compute inverse tangent of the argument with double precison. |
| float atanf(float) | Compute inverse tangent of the argument with single precison. |
| double atanh(double) | Compute inverse hyperbolic tangent of the argument with double precison. |
| float atanhf(float) | Compute inverse hyperbolic tangent of the argument with single precison. |
| double cabs(double complex z) | Computes absolute value of complex number. The intrinsic argument is a complex number made up of two double precison elements, one real and one imaginary. The input parameter z is made up of two values of double type passed together as a single argument. |
| float cabsf(float complex z) | Computes absolute value of complex number. The intrinsic argument is a complex number made up of two single precison elements, one real and one imaginary. The input parameter z is made up of two values of float type passed together as a single argument. |
| double ceil(double) | Computes smallest integral value of double precison argument not less than the argument. |
| float ceilf(float) | Computes smallest integral value of single precison argument not less than the argument. |
| double cosh(double) | Computes the hyperbolic cosine of double precison argument. |
| float coshf(float) | Computes the hyperbolic cosine of single precison argument. |
| float fabsf(float) | Computes absolute value of single precison argument. |
| double floor(double) | Computes the largest integral value of the double precison argument not greater than the argument. |
| float floorf(float) | Computes the largest integral value of the single precison argument not greater than the argument. |
| double fmod(double) | Computes the floating-point remainder of the division of the first argument by the second argument with double precison. |
| float fmodf(float) | Computes the floating-point remainder of the division of the first argument by the second argument with single precison. |
| double hypot(double, double) | Computes the length of the hypotenuse of a right angled triangle with double precison. |
| float hypotf(float, float) | Computes the length of the hypotenuse of a right angled triangle with single precison. |
| double rint(double) | Computes the integral value represented as double using the IEEE rounding mode. |
| float rintf(float) | Computes the integral value represented with single precison using the IEEE rounding mode. |
| double sinh(double) | Computes the hyperbolic sine of the double precison argument. |
| float sinhf(float) | Computes the hyperbolic sine of the single precison argument. |
| float sqrtf(float) | Computes the square root of the single precison argument. |
| double tanh(double) | Computes the hyperbolic tangent of the double precison argument. |
| float tanhf(float) | Computes the hyperbolic tangent of the single precison argument. |

### 字符串和块拷贝指令

Note：在IA-64的架构上，进行字符串和块拷贝时可以当做正规函数的调用来使用

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

### 混杂指令

| **指令** | **描述** |
| --- | --- |
| \_abnormal\_termination(void) | Can be invoked only by termination handlers. Returns TRUE if the termination handler is invoked as a result of a premature exit of the corresponding try-finally region. |
| \_\_cpuid | Queries the processor for information about processor type and supported features. The Intel(R) C++ Compiler supports the Microsoft\* implementation of this intrinsic. See the Microsoft documentation for details. |
| void \*\_alloca(int) | Allocates memory in the local stack frame. The memory is automatically freed upon return from the function. |
| int \_bit\_scan\_forward(int x) | Returns the bit index of the least significant set bit of x. If x is 0, the result is undefined. |
| int \_bit\_scan\_reverse(int) | Returns the bit index of the most significant set bit of x. If x is 0, the result is undefined. |
| int \_bswap(int) | Reverses the byte order of x. Bits 0-7 are swapped with bits 24-31, and bits 8-15 are swapped with bits 16-23. |
| int \_BitScanForward64(int x) | Returns the bit index of the least significant set bit of x. If x is 0, the result is undefined. |
| int \_BitScanReverse64(int x) | Returns the bit index of the most significant set bit of x. If x is 0, the result is undefined. |
| int \_bswap64(int x) | Reverses the byte order of x. |
| unsigned int \_\_cacheSize(unsigned int cacheLevel) | \_\_cacheSize(n) returns the size in bytes of the cache at level n. 1 represents the first-level cache. 0 is returned for a non-existent cache level. For example, an application may query the cache size and use it to select block sizes in algorithms that operate on matrices. |
| \_exception\_code(void) | Returns the exception code. |
| \_exception\_info(void) | Returns the exception information. |
| void \_enable(void) | Enables the interrupt. |
| void \_disable(void) | Disables the interrupt. |
| int \_in\_byte(int) | Intrinsic that maps to the IA-32 instruction IN. Transfer data byte from port specified by argument. |
| int \_in\_dword(int) | Intrinsic that maps to the IA-32 instruction IN. Transfer double word from port specified by argument. |
| int \_in\_word(int) | Intrinsic that maps to the IA-32 instruction IN. Transfer word from port specified by argument. |
| int \_inp(int) | Same as \_in\_byte |
| int \_inpd(int) | Same as \_in\_dword |
| int \_inpw(int) | Same as \_in\_word |
| int \_out\_byte(int, int) | Intrinsic that maps to the IA-32 instruction OUT. Transfer data byte in second argument to port specified by first argument. |
| int \_out\_dword(int, int) | Intrinsic that maps to the IA-32 instruction OUT. Transfer double word in second argument to port specified by first argument. |
| int \_out\_word(int, int) | Intrinsic that maps to the IA-32 instruction OUT. Transfer word in second argument to port specified by first argument. |
| int \_outp(int, int) | Same as \_out\_byte |
| int \_outpw(int, int) | Same as \_out\_word |
| int \_outpd(int, int) | Same as \_out\_dword |
| int \_popcnt32(int x) | Returns the number of set bits in x. |
| \_\_int64 \_rdpmc(int p) | Returns the current value of the 40-bit performance monitoring counter specified by p. |

## Intrinsics for IA-32 and Intel® 64 Architectures Only

| **指令** | **描述** |
| --- | --- |
| \_\_int64 \_rdtsc(void) | Returns the current value of the processor's 64-bit time stamp counter. This intrinsic is not implemented on systems based on IA-64 architecture. |
| int \_setjmp(jmp\_buf) | A fast version of setjmp(), which bypasses the termination handling. Saves the callee-save registers, stack pointer and return address. This intrinsic is not implemented on systems based on IA-64 architecture. |