目录

固有指令：

命名和使用语法

高级加密标准执行的固有指令

转换半 float 的指令

交叉编译器的固有指令

数据对齐，内存分配和内联汇编的固有指令

IA-64 架构的固有指令

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）

指令	操作	简要描述
_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	

指令	操作	简要描述
_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

指令	操作	简要描述
_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	Greater Than	如果 m1中的 16 位有符号大于 m2

指令	操作	简要描述
m1, __m64 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	R0 R1 i0 i1 置 2 个有符号 32 位整型
_mm_set_pi16(short s3, short s2, short s1, short s0)	set integer values	R0 R1 R2 R3 w0 w1 w2 w3 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	R0 R1 ... R7 b0 b1 ... b7 置 8 个有符号 8 位
_mm_set1_pi32(int i)	set integer values	R0 R1 i i 置 2 个有符号 32 位
_mm_set1_pi16(short s)	set integer values	R0 R1 R2 R3 w w w w 置 4 个有

指令	操作	简要描述
		符号 16 位
_mm_set1_pi8(char b)	set integer values	R0 R1 ... R7 b b ... b 置 8 个有符号 8 位
_mm_setr_pi32	set integer values	R0 R1 i1 i0 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	R0 R1 R2 R3 w3 w2 w1 w0 逆序置 4 个有符号 16 位
_mm_setr_pi8	set integer values	R0 R1 ... R7 b7 b6 ... b0 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

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_add_ss(__m128 a, __m128 b)</td><td>Addition</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 + b0</td><td>a1</td><td>a2</td><td>a3</td></tr></table>, 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</td></tr>
<tr><td>_mm_add_ps(__m128 a, __m128 b)</td><td>Addition</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 +b0</td><td>a1 + b1</td><td>a2 + b2</td><td>a3 + b3</td></tr></table></td></tr>
<tr><td>_mm_sub_ss(__m128 a, __m128 b)</td><td>Subtraction</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 - b0</td><td>a1</td><td>a2</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_sub_ps(__m128 a, __m128 b)</td><td>Subtraction</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 - b0</td><td>a1 - b1</td><td>a2 - b2</td><td>a3 - b3</td></tr></table></td></tr>
<tr><td>_mm_mul_ss(__m128 a, __m128 b)</td><td>Multiplication</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 * b0</td><td>a1</td><td>a2</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_mul_ps(__m128 a, __m128 b)</td><td>Multiplication</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 * b0</td><td>a1 * b1</td><td>a2 * b2</td><td>a3 * b3</td></tr></table></td></tr>
<tr><td>_mm_div_ss(__m128 a, __m128 b)</td><td>Division</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 / b0</td><td>a1</td><td>a2</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_div_ps(__m128 a, __m128 b)</td><td>Division</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 / b0</td><td>a1 / b1</td><td>a2 / b2</td><td>a3 / b3</td></tr></table></td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_sqrt_ss(__m128 a)</td><td>Squared Root</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>sqrt(a0)</td><td>a1</td><td>a2</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_sqrt_ps(__m128 a)</td><td>Squared Root</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>sqrt(a0)</td><td>sqrt(a1)</td><td>sqrt(a2)</td><td>sqrt(a3)</td></tr></table></td></tr>
<tr><td>_mm_rcp_ss(__m128 a)</td><td>Reciprocal</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>recip(a0)</td><td>a1</td><td>a2</td><td>a3</td></tr></table>计算倒数的近似值</td></tr>
<tr><td>_mm_rcp_ps(__m128 a)</td><td>Reciprocal</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>recip(a0)</td><td>recip(a1)</td><td>recip(a2)</td><td>recip(</td></tr></table></td></tr>
<tr><td>_mm_rsqrt_ss(__m128 a)</td><td>Reciprocal Squared Root</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>recip(sqrt(a0))</td><td>a1</td><td>a2</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_rsqrt_ps(__m128 a)</td><td>Reciprocal Squared Root</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th></tr><tr><td>recip(sqrt(a0))</td><td>recip(sqrt(a1))</td><td>recip(sq(a2))</td></tr></table></td></tr>
<tr><td>_mm_min_ss(__m128 a, __m128 b)</td><td>Computes Minimum</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>min(a0, b0)</td><td>a1</td><td>a2</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_min_ps(__m128 a, __m128 b)</td><td>Computes Minimum</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th></tr><tr><td>min(a0, b0)</td><td>min(a1, b1)</td><td>min(a2, b2)</td></tr></table></td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_max_ss(__m128 a, __m128 b)</td><td>Computes Maximum</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>max(a0, b0)</td><td>a1</td><td>a2</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_max_ps(__m128 a, __m128 b)</td><td>Computes Maximum</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th></tr><tr><td>max(a0, b0)</td><td>max(a1, b1)</td><td>max(a2, b2)</td></tr></table></td></tr>
</table>

SSE的逻辑操作

原型在 xmmintrin.h 头文件中

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_and_ps(__m128 a, __m128 b)</td><td>Bitwise AND</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 & b0</td><td>a1 & b1</td><td>a2 & b2</td><td>a3 & b3</td></tr></table>4 个 SPFP位与</td></tr>
<tr><td>_mm_andnot_ps(__m128 a, __m128 b)</td><td>Bitwise ANDNOT</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>~a0 & b0</td><td>~a1 & b1</td><td>~a2 & b2</td><td>~a3 & b3</td></tr></table>4 个 SPFP位与非</td></tr>
<tr><td>_mm_or_ps(__m128 a, __m128 b)</td><td>Bitwise OR</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 | b0</td><td>a1 | b1</td><td>a2 | b2</td><td>a3 | b3</td></tr></table>4 个 SPFP位或</td></tr>
<tr><td>_mm_xor_ps(__m128 a, __m128 b)</td><td>Bitwise Exclusive OR</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 ^ b0</td><td>a1 ^ b1</td><td>a2 ^ b2</td><td>a3 ^ b3</td></tr></table></td></tr>
</table>

SSE的比较操作

原型在 xmmintrin.h 头文件中

指令	操作	简要描述			
		R0	R1	R2	R3
_mm_cmpeq_ss(__m128 a, __m128 b)	Equal	(a0 == b0) ? 0xffffffff : 0x0	a1	a2	a3
_mm_cmpeq_ps	Equal	(a0 == b0) ? 0xffffffff : 0x0	(a1 == b1) ? 0xffffffff : 0x0	(a2 == b2) ? 0xffffffff : 0x0	(a3 == b3) ? 0xffffffff : 0x0
_mm_cmplt_ss	Less Than	(a0 < b0) ? 0xffffffff : 0x0	a1	a2	a3
_mm_cmplt_ps	Less Than	(a0 < b0) ? 0xffffffff : 0x0	(a1 < b1) ? 0xffffffff : 0x0	(a2 < b2) ? 0xffffffff : 0x0	(a3 < b3) ? 0xffffffff : 0x0
_mm_cmple_ss	Less Than or Equal	(a0 <= b0) ? 0xffffffff : 0x0	a1	a2	a3
_mm_cmple_ps	Less Than or Equal	(a0 <= b0) ? 0xffffffff : 0x0	(a1 <= b1) ? 0xffffffff : 0x0	(a2 <= b2) ? 0xffffffff : 0x0	(a3 <= b3) ? 0xffffffff : 0x0

指令	操作	简要描述			
		R0	R1	R2	R3
_mm_cmpgt_ss	Greater Than	(a0 > b0) ? 0xffffffff : 0x0	a1	a2	a3
_mm_cmpgt_ps	Greater Than	(a0 > b0) ? 0xffffffff : 0x0	(a1 > b1) ? 0xffffffff : 0x0	(a2 > b2) ? 0xffffffff : 0x0	(a3 > b3) ? 0xffffffff : 0x0
_mm_cmpge_ss	Greater Than or Equal	(a0 >= b0) ? 0xffffffff : 0x0	a1	a2	a3
_mm_cmpge_ps	Greater Than or Equal	(a0 >= b0) ? 0xffffffff : 0x0	(a1 >= b1) ? 0xffffffff : 0x0	(a2 >= b2) ? 0xffffffff : 0x0	(a3 >= b3) ? 0xffffffff : 0x0
_mm_cmpneq_ss	Not Equal	(a0 != b0) ? 0xffffffff : 0x0	a1	a2	a3
_mm_cmpneq_ps	Not Equal	(a0 != b0) ? 0xffffffff : 0x0	(a1 != b1) ? 0xffffffff : 0x0	(a2 != b2) ? 0xffffffff : 0x0	(a3 != b3) ? 0xffffffff : 0x0
_mm_cmpnlt_ss	Not Less Than	!(a0 < b0) ? 0xffffffff : 0x0	a1	a2	a3

指令	操作	简要描述			
		R0	R1	R2	R3
_mm_cmpnlt_ps	Not Less Than	!(a0 < b0) ? 0xffffffff : 0x0	!(a1 < b1) ? 0xffffffff : 0x0	!(a2 < b2) ? 0xffffffff : 0x0	!(a3 < b3) ? 0xffffffff : 0x0
_mm_cmpnle_ss	Not Less Than or Equal	!(a0 <= b0) ? 0xffffffff : 0x0	a1	a2	a3
_mm_cmpnle_ps	Not Less Than or Equal	!(a0 <= b0) ? 0xffffffff : 0x0	!(a1 <= b1) ? 0xffffffff : 0x0	!(a2 <= b2) ? 0xffffffff : 0x0	!(a3 <= b3) ? 0xffffffff : 0x0
_mm_cmpngt_ss	Not Greater Than	!(a0 > b0) ? 0xffffffff : 0x0	a1	a2	a3
_mm_cmpngt_ps	Not Greater Than	!(a0 > b0) ? 0xffffffff : 0x0	!(a1 > b1) ? 0xffffffff : 0x0	!(a2 > b2) ? 0xffffffff : 0x0	!(a3 > b3) ? 0xffffffff : 0x0
_mm_cmpnge_ss	Not Greater Than or Equal	!(a0 >= b0) ? 0xffffffff : 0x0	a1	a2	a3

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_cmpnge_ps</td><td>Not Greater Than or Equal</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>!(a0 >= b0) ? 0xffffffff : 0x0</td><td>!(a1 >= b1) ? 0xffffffff : 0x0</td><td>!(a2 >= b2) ? 0xffffffff : 0x0</td><td>!(a3 >= b3) ? 0xffffffff : 0x0</td></tr></table></td></tr>
<tr><td>_mm_cmpord_ss</td><td>Ordered</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(a0 ord? b0) ? 0xffffffff : 0x0</td><td>a1</td><td>a2</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_cmpord_ps</td><td>Ordered</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(a0 ord? b0) ? 0xffffffff : 0x0</td><td>(a1 ord? b1) ? 0xffffffff : 0x0</td><td>(a2 ord? b2) ? 0xffffffff : 0x0</td><td>(a3 ord? b3) ? 0xffffffff : 0x0</td></tr></table></td></tr>
<tr><td>_mm_cmpunord_ss</td><td>Unordered</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(a0 unord? b0) ? 0xffffffff : 0x0</td><td>a1</td><td>a2</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_cmpunord_ps</td><td>Unordered</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(a0 unord? b0) ? 0xffffffff : 0x0</td><td>(a1 unord? b1) ? 0xffffffff : 0x0</td><td>(a2 unord? b2) ? 0xffffffff : 0x0</td><td>(a3 unord? b3) ? 0xffffffff : 0x0</td></tr></table></td></tr>
<tr><td>_mm_comieq_ss</td><td>Equal</td><td><table><tr><th>R</th></tr><tr><td>(a0 == b0) ? 0x1 : 0x0</td></tr></table></td></tr>
<tr><td>_mm_comilt_ss</td><td>Less Than</td><td><table><tr><th>R</th></tr><tr><td>(a0 < b0) ? 0x1 : 0x0</td></tr></table></td></tr>
</table>

指令	操作	简要描述
_mm_comile_ss	Less Than or Equal	**R** (a0 <= b0) ? 0x1 : 0x0
_mm_comigt_ss	Greater Than	**R** (a0 > b0) ? 0x1 : 0x0
_mm_comige_ss	Greater Than or Equal	**R** (a0 >= b0) ? 0x1 : 0x0
_mm_comineq_ss	Not Equal	**R** (a0 != b0) ? 0x1 : 0x0
_mm_ucomieq_ss	Equal	**R** (a0 == b0) ? 0x1 : 0x0
_mm_ucomilt_ss	Less Than	**R** (a0 < b0) ? 0x1 : 0x0
_mm_ucomile_ss	Less Than or Equal	**R** (a0 <= b0) ? 0x1 : 0x0
_mm_ucomigt_ss	Greater Than	**R** (a0 > b0) ? 0x1 : 0x0
_mm_ucomige_ss	Greater Than or	**R** (a0 >= b0) ? 0x1 : 0x0

指令	操作	简要描述
	Equal	
_mm_ucomineq_ss	Not Equal	R r := (a0 != b0) ? 0x1 : 0x0

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	R (int)a0 将低位的 SPFP转成 32 位整型（舍入模式）
_mm_cvtss_si64	Convert to 64-bit integer	R (__int64)a0 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	R0 R1 (int)a0 (int)a1
_mm_cvttss_si32	Convert to 32-bit integer	R (int)a0 将低位的 SPFP转成 32 位整型（截尾模式）
_mm_cvttss_si64	Convert to 64-bit integer	R (__int64)a0 将低位的 SPFP转成 64 位有符号整型（截尾模式）

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_cvttps_pi32</td><td>Convert to two 32-bit integers</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>(int)a0</td><td>(int)a1</td></tr></table></td></tr>
<tr><td>_mm_cvtsi32_ss(__m128 a, int b)</td><td>Convert from 32-bit integer</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(float)b</td><td>a1</td><td>a2</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_cvtsi64_ss</td><td>Convert from 64-bit integer</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(float)b</td><td>a1</td><td>a2</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_cvtpi32_ps</td><td>Convert from two 32-bit integers</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(float)b0</td><td>(float)b1</td><td>a2</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_cvtpi16_ps(__m64 a)</td><td>Convert from four 16-bit integers</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(float)a0</td><td>(float)a1</td><td>(float)a2</td><td>(float)</td></tr></table>将 4 个 16 位有符号转为 SPFP</td></tr>
<tr><td>_mm_cvtpu16_ps</td><td>Convert from four 16-bit integers</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(float)a0</td><td>(float)a1</td><td>(float)a2</td><td>(float)</td></tr></table>将 4 个 16 位无符号转为 SPFP</td></tr>
<tr><td>_mm_cvtpi8_ps</td><td>Convert from four 8-bit integers</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(float)a0</td><td>(float)a1</td><td>(float)a2</td><td>(float)</td></tr></table>将低位的 4 个 8 位有符号转为 SPFP</td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_cvtpu8_ps</td><td>Convert from four 8-bit integers</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(float)a0</td><td>(float)a1</td><td>(float)a2</td><td>(float)</td></tr></table>将低位的 4 个 8 位无符号转为 SPFP</td></tr>
<tr><td>_mm_cvtpi32x2_ps(__m64 a, __m64 b)</td><td>Convert from four 32-bit integers</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(float)a0</td><td>(float)a1</td><td>(float)b0</td><td>(float)</td></tr></table>a 中的 2 个 32 位有符号和 b 中的 2 个 32 位转为 4 个 SPFP</td></tr>
<tr><td>_mm_cvtps_pi16(__m128 a)</td><td>Convert to four 16-bit integers</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(short)a0</td><td>(short)a1</td><td>(short)a2</td><td>(short)</td></tr></table>将 a 中的四个 SPFP转为 4 个有符号的 16 位整型</td></tr>
<tr><td>_mm_cvtps_pi8</td><td>Convert to four 8-bit integers</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(char)a0</td><td>(char)a1</td><td>(char)a2</td><td>(char)a3</td></tr></table>将 2 个 SPFP转到结果的低位的 4 个有符号 8 位</td></tr>
<tr><td>_mm_cvtss_f32</td><td>Extract</td><td>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。</td></tr>
</table>

SSE的加载操作

原型在 xxmintrin.h 头文件中

指令	操作	简要描述

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_loadh_pi(__m128 a, __m64 const *p)</td><td>Load high</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0</td><td>a1</td><td>*p0</td><td>*p1</td></tr></table>从地址 p 中加载的 64 位数据来置位 a 的高位的 SPFP</td></tr>
<tr><td>_mm_loadl_pi</td><td>Load low</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>*p0</td><td>*p1</td><td>a2</td><td>a3</td></tr></table>从地址 p 中加载的 64 位数据来置位 a 的低位的 SPFP</td></tr>
<tr><td>_mm_load_ss(float * p)</td><td>Load the low value and clear the three high values</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>*p</td><td>0.0</td><td>0.0</td><td>0.0</td></tr></table>加载 1 个 SPFP在低位，且高位清 0</td></tr>
<tr><td>_mm_load1_ps</td><td>Load one value into all four words</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>*p</td><td>*p</td><td>*p</td><td>*p</td></tr></table>加载 1 个 SPFP，且把它拷到所有的 4 个字节</td></tr>
<tr><td>_mm_load_ps</td><td>Load four values, address aligned</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>p[0]</td><td>p[1]</td><td>p[2]</td><td>p[3]</td></tr></table>加载 4 个 SPFP，地址必须是 16 字节对齐的</td></tr>
<tr><td>_mm_loadu_ps</td><td>Load four values, address unaligned</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>p[0]</td><td>p[1]</td><td>p[2]</td><td>p[3]</td></tr></table>加载 4 个 SPFP，地址不需要 16 字节对</td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td></td><td></td><td>齐</td></tr>
<tr><td>_mm_loadr_ps</td><td>Load four values in reverse</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>p[3]</td><td>p[2]</td><td>p[1]</td><td>p[0]</td></tr></table>逆序加载 4 个 SPFP，地址必须 16 字节对齐</td></tr>
</table>

SSE的置位操作

原型在 xmmintri.h 头文件中

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_set_ss(float w)</td><td>Set the low value and clear the three high values</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>w</td><td>0.0</td><td>0.0</td><td>0.0</td></tr></table>置低位的 1 个 SPFP，高位 3 字节清 0</td></tr>
<tr><td>_mm_set1_ps</td><td>Set all four words with the same value</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>w</td><td>w</td><td>w</td><td>w</td></tr></table>置 4 个 SPFP</td></tr>
<tr><td>_mm_set_ps(float z, float y, float x, float w)</td><td>Set four values, address aligned</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>w</td><td>x</td><td>y</td><td>z</td></tr></table>置 4 个 SPFP为 4 个数</td></tr>
<tr><td>_mm_setr_ps(float z, float y, float x, float w)</td><td>Set four values, in reverse order</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>z</td><td>y</td><td>x</td><td>w</td></tr></table>逆序置 4 个 SPFP为 4 个数</td></tr>
</table>

指令	操作	简要描述
_mm_setzero_ps(void)	Clear all four values	R0 R1 R2 R3 0.0 0.0 0.0 0.0 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	*p0 *p1 a2 a3 将高位的 2 个 SPFP存入地址 p 中
_mm_storel_pi	Store low	*p0 *p1 a0 a1 将低位的 2 个 SPFP存入地址 p 中
_mm_store_ss(float * p, __m128 a)	Store the low value	*p a0 存入最低的 SPFP
_mm_store1_ps(float * p, __m128 a)	Store the low value across all four words, address aligned	p[0] p[1] p[2] p[3] a0 a0 a0 a0 存储低位 SPFP并贯穿 4 个字节

指令	操作	简要描述
_mm_store_ps	Store four values, address aligned	p[0] p[1] p[2] p[3] a0 a1 a2 a3 存储 4 个 SPFP。地址必须是 16 字节对齐的
_mm_storeu_ps	Store four values, address unaligned	p[0] p[1] p[2] p[3] a0 a1 a2 a3 存储 4 个 SPFP。地址不需要 16 字节对齐
_mm_storer_ps	Store four values, in reverse order	p[0] p[1] p[2] p[3] a3 a2 a1 a0 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

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_extract_pi16（__m64 a, int n)</td><td>Extract one of four words</td><td><table><tr><th>R</th></tr><tr><td>(n==0) ? a0 : ((n==1) ? a1 : ((n==2) ? a</td></tr></table>从 a 中取 1 个。 n 必须是一个立即数</td></tr>
<tr><td>_mm_insert_pi16（__m64 a, int d, int n)</td><td>Insert word</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R</th></tr><tr><td>(n==0) ? d : a0;</td><td>(n==1) ? d : a1;</td><td>(n==2) ? d : a2;</td><td>d</td></tr></table>a 的 4 个数中插入一个 d。 n 必须是一个立即数</td></tr>
<tr><td>_mm_max_pi16(__m64 a, __m64 b)</td><td>Compute maximum</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>min(a0, b0)</td><td>min(a1, b1)</td><td>min(a2, b2)</td><td>mi</td></tr></table>得出 a、b 中对应位置的最大值</td></tr>
<tr><td>_mm_max_pu8</td><td>Compute maximum, unsigned</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>min(a0, b0)</td><td>min(a1, b1)</td><td>min(a2, b2)</td><td>mi</td></tr></table>得出 a、b 中的对应的无符号的最大值</td></tr>
<tr><td>_mm_min_pi16</td><td>Compute minimum</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>min(a0, b0)</td><td>min(a1, b1)</td><td>min(a2, b2)</td><td>min</td></tr></table>得出 a、b 中对应位置的最小值</td></tr>
<tr><td>_mm_min_pu8</td><td>Compute minimum, unsigned</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>min(a0, b0)</td><td>min(a1, b1)</td><td>min(a2, b2)</td><td>mi</td></tr></table>得出 a、b 中对应位置的最小值</td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_movemask_pi8(__m64 b)</td><td>Create eight-bit mask</td><td><table><tr><th>R</th></tr><tr><td>sign(a7)<<7 | sign(a6)<<6 |... | sign(a0)</td></tr></table>从 a 的最重要的比特位中创造出 1 个 8 位的掩码</td></tr>
<tr><td>_mm_mulhi_pu16(__m64 a, __m64 b)</td><td>Multiply, return high bits</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th></tr><tr><td>hiword(a0 * b0)</td><td>hiword(a1 * b1)</td><td>hiword(a2 * b2)</td></tr></table>a 和 b 的无符号相乘，返回 32 位中间结果的高 16 位</td></tr>
<tr><td>_mm_shuffle_pi16 (__m64 a, int n)</td><td>Return a combination of four words</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th></tr><tr><td>word (n&0x3) of a</td><td>word ((n>>2) &0x3) of a</td><td>word ((n>>4) &0x3) of a</td></tr></table>返回 a 的 4 个数的 1 个联合。 n 必须是一个立即数</td></tr>
<tr><td>_mm_maskmove_si64(__m64 d, __m64 n, char *p)</td><td>Conditional Store</td><td><table><tr><th>if (sign (n0))</th><th>if (sign (n1))</th><th>...</th></tr><tr><td>p[0] := d0</td><td>p[1] := d1</td><td>...</td></tr></table>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</td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_avg_pu8(__m64 a, __m64 b)</td><td>Compute rounded average</td><td><table><tr><th>R0</th><th>R1</th><th>...</th></tr><tr><td>(t >> 1) | (t & 0x01), where t = (unsigned char)a0 + (unsigned char)b0</td><td>(t >> 1) | (t & 0x01), where t = (unsigned char)a1 + (unsigned char)b1</td><td>...</td></tr></table>
计算 a 和 b 中无符号的平均值（ round 模式）</td></tr>
<tr><td>_mm_avg_pu16</td><td>Compute rounded average</td><td><table><tr><th>R0</th><th>R1</th><th>...</th></tr><tr><td>(t >> 1) | (t & 0x01), where t = (unsigned char)a0 + (unsigned char)b0</td><td>(t >> 1) | (t & 0x01), where t = (unsigned char)a1 + (unsigned char)b1</td><td>...</td></tr></table>
计算 a 和 b 中无符号的平均值（ round 模式）</td></tr>
<tr><td>_mm_sad_pu8</td><td>Compute sum of absolute differences</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th></tr><tr><td>abs(a0-b0) +... + abs(a7-b7)</td><td>0</td><td>0</td></tr></table>
计算 a 和 b 中无符号数的差的绝对值的总和， 且高位置 0</td></tr>
</table>

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

指令	操作	简要描述
_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

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_shuffle_ps (__m128 a, __m128 b, unsigned int imm8)</td><td>Shuffle</td><td>基于 imm8从 a 和 b 中选择 4 个指定的 SPFP，掩码必须是立即数</td></tr>
<tr><td>_mm_unpackhi_ps(__m128 a, __m128 b)</td><td>Unpack High</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a2</td><td>b2</td><td>a3</td><td>b3</td></tr></table>a 和 b 中的高位 2 个 SPFP进行交织</td></tr>
<tr><td>_mm_unpacklo_ps</td><td>Unpack Low</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0</td><td>b0</td><td>a1</td><td>b1</td></tr></table>a 和 b 中的低位 2 个 SPFP进行交织</td></tr>
<tr><td>_mm_move_ss</td><td>Set low word, pass in three high values</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>b0</td><td>a1</td><td>a2</td><td>a3</td></tr></table>将 a 的低位置成 b 的</td></tr>
<tr><td>_mm_movehl_ps</td><td>Move High to Low</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>b2</td><td>b3</td><td>a2</td><td>a3</td></tr></table></td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_movelh_ps</td><td>Move Low to High</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0</td><td>a1</td><td>b0</td><td>b1</td></tr></table></td></tr>
<tr><td>_mm_movemask_ps(_m128 a)</td><td>Create four-bit mask</td><td><table><tr><th>R</th></tr><tr><td>sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 | s</td></tr></table>从 4 个 SPFP的最重要比特位中创造 1 个 4 比特的掩码</td></tr>
</table>

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

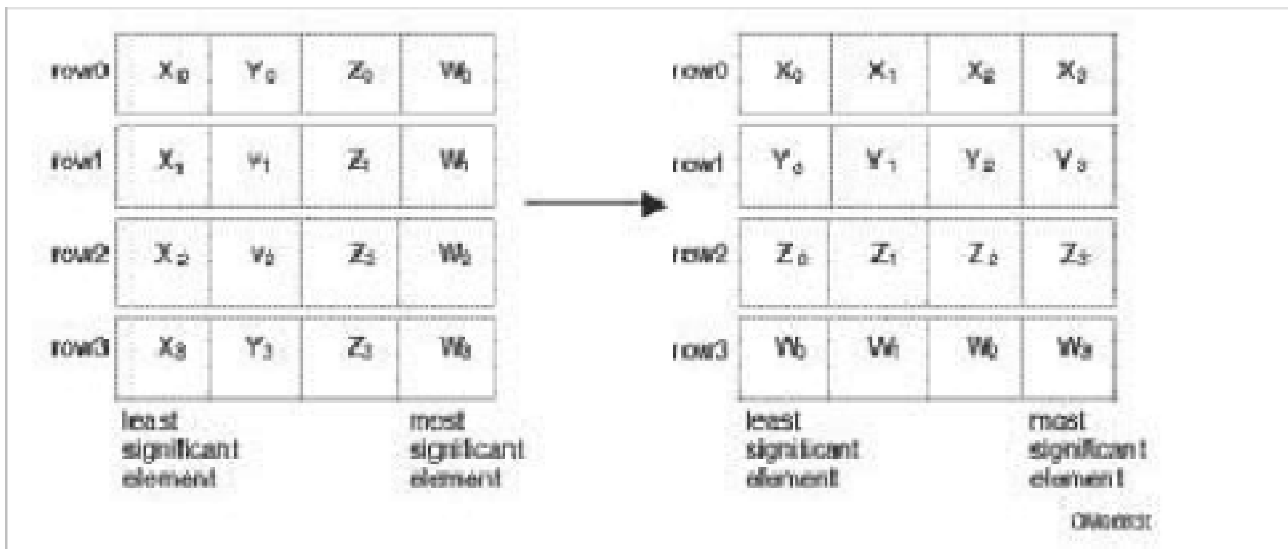
异常状态宏	宏参数
_MM_SET_EXCEPTION_STATE(x)	_MM_EXCEPT_INVALID
_MM_GET_EXCEPTION_STATE()	_MM_EXCEPT_DIV_ZERO
	_MM_EXCEPT_DENORM
Macro Definitions Write to and read from the six least significant control register bits, respectively.	_MM_EXCEPT_OVERFLOW

异常状态宏	宏参数
	_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

<table>
<tr><th>指令</th><th>操作</th><th>简要操作</th></tr>
<tr><td>_mm_add_sd(__m128d a, __m128d b)</td><td>Addition</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0 + b0</td><td>a1</td></tr></table></td></tr>
<tr><td>_mm_add_pd</td><td>Addition</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0 + b0</td><td>a1 + b1</td></tr></table></td></tr>
<tr><td>_mm_sub_sd</td><td>Subtraction</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0 - b0</td><td>a1</td></tr></table></td></tr>
<tr><td>_mm_sub_pd</td><td>Subtraction</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0 - b0</td><td>a1 - b1</td></tr></table></td></tr>
<tr><td>_mm_mul_sd</td><td>Multiplication</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0 * b0</td><td>a1</td></tr></table></td></tr>
<tr><td>_mm_mul_pd</td><td>Multiplication</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0 * b0</td><td>a1 * b1</td></tr></table></td></tr>
<tr><td>_mm_div_sd</td><td>Division</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0 / b0</td><td>a1</td></tr></table></td></tr>
<tr><td>_mm_div_pd</td><td>Division</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0 / b0</td><td>a1 / b1</td></tr></table></td></tr>
<tr><td>_mm_sqrt_sd</td><td>Computes Square Root</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>sqrt(b0)</td><td>a1</td></tr></table></td></tr>
</table>

指令	操作	简要操作
_mm_sqrt_pd(__m128d a)	Computes Square Root	R0: sqrt(a0) R1: sqrt(a1)
_mm_min_sd(__m128d a, __m128d b)	Computes Minimum	R0: min (a0, b0) R1: a1
_mm_min_pd	Computes Minimum	R0: min (a0, b0) R1: min(a1, b1)
_mm_max_sd	Computes Maximum	R0: max (a0, b0) R1: a1
_mm_max_pd	Computes Maximum	R0: max (a0, b0) R1: max (a1, b1)

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	R0: a0 & b0 R1: a1 & b1 计算 2 个 DPFP的位与
_mm_andnot_pd	Computes AND and NOT	R0: (~a0) & b0 R1: (~a1) & b1

指令	操作	简要描述
_mm_or_pd	Computes OR	R0: a0 | b0 R1: a1 | b1
_mm_xor_pd	Computes XOR	R0: a0 ^ b0 R1: a1 ^ b1

浮点比较操作

原型在 emmintrin.h 头文件中

指令	操作	简要描述
_mm_cmpeq_pd m128d a, __m128d b)	Equality	R0: (a0 == b0) ? 0xffffffffffffffff : 0x0 R1: (a1 == b1) ? 0xffffffffffffff
_mm_cmplt_pd	Less Than	R0: (a0 < b0) ? 0xffffffffffffffff : 0x0 R1: (a1 < b1) ? 0xffffffffffffffff
_mm_cmple_pd	Less Than or Equal	R0: (a0 <= b0) ? 0xffffffffffffffff : 0x0 R1: (a1 <= b1) ? 0xffffffffffffffff
_mm_cmpgt_pd	Greater Than	R0: (a0 > b0) ? 0xffffffffffffffff : 0x0 R1: (a1 > b1) ? 0xffffffffffffff
_mm_cmpge_pd	Greater Than or Equal	R0: (a0 >= b0) ? 0xffffffffffffffff : 0x0 R1: (a1 >= b1) ? 0xffffffffffffffff

指令	操作	简要描述
_mm_cmpord_pd	Ordered	R0: (a0 ord b0) ? 0xffffffffffffffff : 0x0 R1: (a1 ord b1) ? 0xffffffffffffffff 比较 a 和 b 是否有序
_mm_cmpunord_pd	Unordered	R0: (a0 unord b0) ? 0xffffffffffffffff : 0x0 R1: (a1 unord b1) ? 0xffffffffffffffff
_mm_cmpneq_pd	Inequality	R0: (a0 != b0) ? 0xffffffffffffffff : 0x0 R1: (a1 != b1) ? 0xffffffffffffffff
_mm_cmpnlt_pd	Not Less Than	R0: !(a0 < b0) ? 0xffffffffffffffff : 0x0 R1: !(a1 < b1) ? 0xffffffffffffffff
_mm_cmpnle_pd	Not Less Than or Equal	R0: !(a0 <= b0) ? 0xffffffffffffffff : 0x0 R1: !(a1 <= b1) ? 0xffffffffffffffff
_mm_cmpngt_pd	Not Greater Than	R0: !(a0 > b0) ? 0xffffffffffffffff : 0x0 R1: !(a1 > b1) ? 0xffffffffffffffff
_mm_cmpnge_pd	Not Greater Than or Equal	R0: !(a0 >= b0) ? 0xffffffffffffffff : 0x0 R1: !(a1 >= b1) ? 0xffffffffffffffff

指令	操作	简要描述
_mm_cmpeq_sd	Equality	**R0**: (a0 == b0) ? 0xffffffffffffffff : 0x0 **R1**: a1
_mm_cmplt_sd	Less Than	**R0**: (a0 < b0) ? 0xffffffffffffffff : 0x0 **R1**: a1
_mm_cmple_sd	Less Than or Equal	**R0**: (a0 <= b0) ? 0xffffffffffffffff : 0x0 **R1**: a1
_mm_cmpgt_sd	Greater Than	**R0**: (a0 > b0) ? 0xffffffffffffffff : 0x0 **R1**: a1
_mm_cmpge_sd	Greater Than or Equal	**R0**: (a0 >= b0) ? 0xffffffffffffffff : 0x0 **R1**: a1
_mm_cmpord_sd	Ordered	**R0**: (a0 ord b0) ? 0xffffffffffffffff : 0x0 **R1**: a1
_mm_cmpunord_sd	Unordered	**R0**: (a0 unord b0) ? 0xffffffffffffffff : 0x0 **R1**: a1

指令	操作	简要描述
_mm_cmpneq_sd	Inequality	**R0**: (a0 != b0) ? 0xffffffffffffffff : 0x0 **R1**: a1
_mm_cmpnlt_sd	Not Less Than	**R0**: !(a0 < b0) ? 0xffffffffffffffff : 0x0 **R1**: a1
_mm_cmpnle_sd	Not Less Than or Equal	**R0**: !(a0 <= b0) ? 0xffffffffffffffff : 0x0 **R1**: a1
_mm_cmpngt_sd	Not Greater Than	**R0**: !(a0 > b0) ? 0xffffffffffffffff : 0x0 **R1**: a1
_mm_cmpnge_sd	Not Greater Than or Equal	**R0**: !(a0 >= b0) ? 0xffffffffffffffff : 0x0 **R1**: a1
_mm_comieq_sd	Equality	**R**: (a0 == b0) ? 0x1 : 0x0
_mm_comilt_sd	Less Than	**R**: (a0 < b0) ? 0x1 : 0x0
_mm_comile_sd	Less Than or Equal	**R**: (a0 <= b0) ? 0x1 : 0x0

指令	操作	简要描述
_mm_comigt_sd	Greater Than	R (a0 > b0) ? 0x1 : 0x0
_mm_comige_sd	Greater Than or Equal	R (a0 >= b0) ? 0x1 : 0x0
_mm_comineq_sd	Not Equal	R (a0 != b0) ? 0x1 : 0x0
_mm_ucomieq_sd	Equality	R (a0 == b0) ? 0x1 : 0x0
_mm_ucomilt_sd	Less Than	R (a0 < b0) ? 0x1 : 0x0
_mm_ucomile_sd	Less Than or Equal	R (a0 <= b0) ? 0x1 : 0x0
_mm_ucomigt_sd	Greater Than	R (a0 > b0) ? 0x1 : 0x0
_mm_ucomige_sd	Greater Than or Equal	R (a0 >= b0) ? 0x1 : 0x0
_mm_ucomineq_sd	Not Equal	R (a0 != b0) ? 0x1 : 0x0

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。

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_cvtpd_ps(__m128d a)</td><td>Convert DP FP to SP FP</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(float) a0</td><td>(float) a1</td><td>0.0</td><td>0.0</td></tr></table></td></tr>
<tr><td>_mm_cvtps_pd(__m128 a)</td><td>Convert from SP FP to DP FP</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>(double) a0</td><td>(double) a1</td></tr></table></td></tr>
<tr><td>_mm_cvtepi32_pd(__m128i a)</td><td>Convert lower integer values to DP FP</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>(double) a0</td><td>(double) a1</td></tr></table>将低位的 2 个 32 位有符号转换为 DPFP</td></tr>
<tr><td>_mm_cvtpd_epi32(__m128d a)</td><td>Convert DP FP values to integer values</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(int) a0</td><td>(int) a1</td><td>0x0</td><td>0x0</td></tr></table>将 2 个 DPFP转为 32 位有符号整型</td></tr>
<tr><td>_mm_cvtsd_si32</td><td>Convert lower DP FP value to integer value</td><td><table><tr><th>R</th></tr><tr><td>(int) a0</td></tr></table>将低位的 DPFP转为 1 个 32 位有符号整型</td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_cvtsd_ss(__m128 a, __m128d b)</td><td>Convert lower DP FP value to SP FP</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(float) b0</td><td>a1</td><td>a2</td><td>a3</td></tr></table>
将低位的 DPFP转为 SPFP</td></tr>
<tr><td>_mm_cvtsi32_sd(__m128d a, int b)</td><td>Convert signed integer value to DP FP</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>(double) b</td><td>a1</td></tr></table>
将 b 的有符号整型转为 DPFP</td></tr>
<tr><td>_mm_cvtss_sd(__m128d a, __m128 b)</td><td>Convert lower SP FP value to DP FP</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>(double) b0</td><td>a1</td></tr></table></td></tr>
<tr><td>_mm_cvttpd_epi32(__m128d a)</td><td>Convert DP FP values to signed integers</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(int) a0</td><td>(int) a1</td><td>0x0</td><td>0x0</td></tr></table>
将 2 个 DPFP转为 32 位有符号整型（截尾模式）</td></tr>
<tr><td>_mm_cvttsd_si32 (__m128d a)</td><td>Convert lower DP FP to signed integer</td><td><table><tr><th>R</th></tr><tr><td>(int) a0</td></tr></table>
将低位的 DPFP转为 32 位有符号整型（截尾模式）</td></tr>
<tr><td>_mm_cvtpd_pi32(__m128d a)</td><td>Convert two DP FP values to signed</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>(int)a0</td><td>(int) a1</td></tr></table></td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td></td><td>integer values</td><td></td></tr>
<tr><td>_mm_cvttpd_pi32</td><td>Convert two DP FP values to signed integer values using truncate</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>(int)a0</td><td>(int) a1</td></tr></table>将 2 个 DPFP转为 32 位有符号整型（截尾模式）</td></tr>
<tr><td>_mm_cvtpi32_pd</td><td>Convert two signed integer values to DP FP</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>(double)a0</td><td>(double)a1</td></tr></table></td></tr>
<tr><td>_mm_cvtsd_f64</td><td>Extract DP FP value from first vector element</td><td></td></tr>
</table>

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

指令	操作	简要描述
_mm_load_pd(double const*dp)	Loads two DP FP values	R0 | R1 p[0] | p[1] 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	R0 | R1 *p | *p 地址不需要 16 字节对齐
_mm_loadr_pd	Loads two DP FP values in reverse order	R0 | R1 p[1] | p[0] 逆序，地址必须是 16 字节对齐
_mm_loadu_pd	Loads two DP FP values	R0 | R1 p[0] | p[1] 地址不需要 16 字节对齐
_mm_load_sd	Loads a DP FP value, sets upper DP FP to zero	R0 | R1 *p | 0.0 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	R0 | R1 a0 | *p 地址不需要 16 字节对齐

指令	操作	简要描述
_mm_loadl_pd	Loads a DP FP value as the lower DPFPvalue of the result	R0 | R1 *p | a1 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	R0 | R1 w | 0.0
_mm_set1_pd	Sets two DP FP valus to w	R0 | R1 w | w
_mm_set_pd(double w, double x)	Sets lower DP FP to x and upper to w	R0 | R1 x | w
_mm_setr_pd	Sets lower DP FP to w and upper to x	R0 | R1 w | x
_mm_setzero_pd	Sets two DPFPvalues to zero	R0 | R1 0.0 | 0.0
_mm_move_sd(__m128d a, __m128d b)	Sets lower DP FP value to the lower DPFPvalue of b	R0 | R1 b0 | a1

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	*dp a0 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	dp[0] dp[1] a0 a0 地址必须 16 字节对齐
_mm_store_pd	Stores two DPFP values	dp[0] dp[1] a0 a1 地址必须 16 字节对齐
_mm_storeu_pd	Stores two DPFP values	dp[0] dp[1] a0 a1 地址不需要 16 字节对齐
_mm_storer_pd	Stores two DPFP values in reverse order	dp[0] dp[1] a1 a0 地址必须是 16 字节对齐

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_storeh_pd(double *dp, __m128d a)</td><td>Stores upper DP FP value of a</td><td><table><tr><th>*dp</th></tr><tr><td>a1</td></tr></table></td></tr>
<tr><td>_mm_storel_pd(double *dp, __m128d a)</td><td>Stores lower DP FP value of a</td><td><table><tr><th>*dp</th></tr><tr><td>a0</td></tr></table></td></tr>
</table>

整型指令

整型算术操作

原型在 emmintrin.h 中

<table>
<tr><th>指令</th><th>操作</th><th>简要说明</th></tr>
<tr><td>_mm_add_epi8(__m128i a, __m128i b)</td><td>Addition</td><td><table><tr><th>R0</th><th>R1</th><th>...</th><th>R15</th></tr><tr><td>a0 + b0</td><td>a1 + b1;</td><td>...</td><td>a15 + b15</td></tr></table>a 和 b 中的 16 个无符号或有符号 8 位相加</td></tr>
<tr><td>_mm_add_epi16</td><td>Addition</td><td><table><tr><th>R0</th><th>R1</th><th>...</th><th>R7</th></tr><tr><td>a0 + b0</td><td>a1 + b1</td><td>...</td><td>a7 + b7</td></tr></table>a 和 b 中的 8 个有符号或无符号 16 位相加</td></tr>
<tr><td>_mm_add_epi32</td><td>Addition</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 + b0</td><td>a1 + b1</td><td>a2 + b2</td><td>a3 + b3</td></tr></table>a 和 b 中的 4 个有符号或无符号 32 位相加</td></tr>
</table>

指令	操作	简要说明
_mm_add_si64(__m64 a, __m64 b)	Addition	R0: a + b a 和 b 中的有符号或无符号 64 位相加
_mm_add_epi64(__m128i a, __m128i b)	Addition	R0: a0 + b0 ; R1: a1 + b1 2 个有符号或无符号 64 位相加
_mm_adds_epi8	Addition	R0: SignedSaturate (a0 + b0) ; R1: SignedSaturate (a1 + b1) ; ...: ... ; R: Si (a a 和 b 中的 16 个有符号 8 位相加（饱和算术）
_mm_adds_epi16	Addition	R0: SignedSaturate (a0 + b0) ; R1: SignedSaturate (a1 + b1) ; ...: ... ; R7: Si (a a 和 b 中的 8 个有符号 16 位相加（饱和算术）
_mm_adds_epu8	Addition	R0: UnsignedSaturate (a0 + b0) ; R1: UnsignedSaturate (a1 + b1) ; ...: ... ; R: Un (a a 和 b 中的 16 个无符号 8 位相加（饱和算术）
_mm_adds_epu16	Addition	R0: UnsignedSaturate (a0 + b0) ; R1: UnsignedSaturate (a1 + b1) ; ...: ... ; R7: Un (a a 和 b 中的 8 个无符号 16 位相加（饱和算术）

指令	操作	简要说明
_mm_avg_epu8	Computes Average	R0: (a0 + b0) / 2; R1: (a1 + b1) / 2; ...: ... a 和 b 中的 16 个无符号 8 位求均值（round 模式）
_mm_avg_epu16	Computes Average	R0: (a0 + b0) / 2; R1: (a1 + b1) / 2; ...: ... a 和 b 中的 8 个无符号 16 位求均值（round 模式）
_mm_madd_epi16	Multiplication and Addition	R0: (a0 * b0) + (a1 * b1); R1: (a2 * b2) + (a3 * b3); R2: (a4 * b4) + (a5 * b5) 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	R0: max(a0, b0); R1: max(a1, b1); ...: ...; R7: ma 逐对取最大值 a 和 b 中的 8 个有符号 16 位
_mm_max_epu8	Computes Maxima	R0: max(a0, b0); R1: max(a1, b1); ...: ... 逐对取最大值 a 和 b 中的 16 个无符号 8 位
_mm_min_epi16	Computes Minima	R0: min(a0, b0); R1: min(a1, b1); ...: ...; R7: mi

<table>
<tr><th>指令</th><th>操作</th><th>简要说明</th></tr>
<tr><td></td><td></td><td>逐对取最小值 a 和 b 中的 8 个有符号 16 位</td></tr>
<tr><td>_mm_min_epu8</td><td>Computes Minima</td><td><table><tr><th>R0</th><th>R1</th><th>...</th></tr><tr><td>min(a0, b0)</td><td>min(a1, b1)</td><td>...</td></tr></table>逐对取最小值 a 和 b 中的 16 个无符号 8 位</td></tr>
<tr><td>_mm_mulhi_epi16</td><td>Multiplication</td><td><table><tr><th>R0</th><th>R1</th><th>...</th></tr><tr><td>(a0 * b0)[31:16]</td><td>(a1 * b1)[31:16]</td><td>...</td></tr></table>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</td></tr>
<tr><td>_mm_mulhi_epu16</td><td>Multiplication</td><td><table><tr><th>R0</th><th>R1</th><th>...</th></tr><tr><td>(a0 * b0)[31:16]</td><td>(a1 * b1)[31:16]</td><td>...</td></tr></table>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</td></tr>
<tr><td>_mm_mullo_epi16</td><td>Multiplication</td><td><table><tr><th>R0</th><th>R1</th><th>...</th></tr><tr><td>(a0 * b0)[15:0]</td><td>(a1 * b1)[15:0]</td><td>...</td></tr></table>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</td></tr>
<tr><td>_mm_mul_su32(__m64 a, __m64 b)</td><td>Multiplication</td><td><table><tr><th>R0</th></tr><tr><td>a0 * b0</td></tr></table>将 a 和 b 中的低 32 位相乘</td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要说明</th></tr>
<tr><td>_mm_mul_epu32</td><td>Multiplication</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0 * b0</td><td>a2 * b2</td></tr></table>
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</td></tr>
<tr><td>_mm_sad_epu8</td><td>Computes Difference/Adds</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th></tr><tr><td>abs(a0 - b0) + abs (a1 - b1) +...+ abs (a7 - b7)</td><td>0x0</td><td>0x0</td></tr></table>
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</td></tr>
<tr><td>_mm_sub_epi8</td><td>Subtraction</td><td><table><tr><th>R0</th><th>R1</th><th>...</th><th>R15</th></tr><tr><td>a0 - b0</td><td>a1 - b1</td><td>...</td><td>a15 - b15</td></tr></table>
将 a 和 b 中的 16 个有符号或无符号 8 位求差</td></tr>
<tr><td>_mm_sub_epi16</td><td>Subtraction</td><td><table><tr><th>R0</th><th>R1</th><th>...</th><th>R7</th></tr><tr><td>a0 - b0</td><td>a1 - b1</td><td>...</td><td>a7 - b7</td></tr></table>
将 a 和 b 中的 8 个有符号或无符号 16 位求差</td></tr>
<tr><td>_mm_sub_epi32</td><td>Subtraction</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 - b0</td><td>a1 - b1</td><td>a2 - b2</td><td>a3 - b3</td></tr></table>
将 a 和 b 中的 4 个有符号或无符号 32 位求差</td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要说明</th></tr>
<tr><td>_mm_sub_si64(__m64 a, __m64 b)</td><td>Subtraction</td><td><table><tr><th>R</th></tr><tr><td>a - b</td></tr></table>
将 a 和 b 中的 1 个有符号或无符号 64 位求差</td></tr>
<tr><td>_mm_sub_epi64</td><td>Subtraction</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0 - b0</td><td>a1 - b1</td></tr></table>
将 a 和 b 中的 2 个有符号或无符号 64 位求差</td></tr>
<tr><td>_mm_subs_epi8</td><td>Subtraction</td><td><table><tr><th>R0</th><th>R1</th><th>...</th><th>R</th></tr><tr><td>SignedSaturate (a0 - b0)</td><td>SignedSaturate (a1 - b1)</td><td>...</td><td>Si (a</td></tr></table>
将 a 和 b 中的 16 个有符号 8 位求差（饱和模式）</td></tr>
<tr><td>_mm_subs_epi16</td><td>Subtraction</td><td><table><tr><th>R0</th><th>R1</th><th>...</th><th>R</th></tr><tr><td>SignedSaturate (a0 - b0)</td><td>SignedSaturate (a1 - b1)</td><td>...</td><td>Si (a</td></tr></table>
将 a 和 b 中的 8 个有符号 16 位求差（饱和模式）</td></tr>
<tr><td>_mm_subs_epu8</td><td>Subtraction</td><td><table><tr><th>R0</th><th>R1</th><th>...</th><th>R</th></tr><tr><td>UnsignedSaturate (a0 - b0)</td><td>UnsignedSaturate (a1 - b1)</td><td>...</td><td>Un (a</td></tr></table>
将 a 和 b 中的 16 个无符号 8 位求差（饱和模式）</td></tr>
<tr><td>_mm_subs_epu16</td><td>Subtraction</td><td><table><tr><th>R0</th><th>R1</th><th>...</th><th>R</th></tr><tr><td>UnsignedSaturate (a0 - b0)</td><td>UnsignedSaturate (a1 - b1)</td><td>...</td><td>Un (a</td></tr></table>
将 a 和 b 中的 8 个无符号 16 位求差（饱和模式）</td></tr>
</table>

整型逻辑操作

原型在 emmintrin.h 头文件中

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_and_si128(__m128i a, __m128i b)</td><td>Computes AND</td><td><table><tr><th>R0</th></tr><tr><td>a & b</td></tr></table></td></tr>
<tr><td>_mm_andnot_si128</td><td>Computes AND and NOT</td><td><table><tr><th>R0</th></tr><tr><td>(~a) & b</td></tr></table></td></tr>
<tr><td>_mm_or_si128</td><td>Computes OR</td><td><table><tr><th>R0</th></tr><tr><td>a | b</td></tr></table></td></tr>
<tr><td>_mm_xor_si128</td><td>Computes XOR</td><td><table><tr><th>R0</th></tr><tr><td>a ^ b</td></tr></table></td></tr>
</table>

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。

<table>
<tr><th>指令</th><th>操作</th><th>移位类型</th><th>简要描述</th></tr>
<tr><td>_mm_slli_si128(__m128i a, int imm)</td><td>Shift left</td><td>Logical</td><td><table><tr><th>R</th></tr><tr><td>a << (imm * 8)</td></tr></table>Imm必须是一个立即数，补 0</td></tr>
<tr><td>_mm_slli_epi16(__m128i a, int count)</td><td>Shift left</td><td>Logical</td><td><table><tr><th>R0</th><th>R1</th><th>...</th><th>R7</th></tr><tr><td>a0 << count</td><td>a1 << count</td><td>...</td><td>a7</td></tr></table></td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>移位类型</th><th>简要描述</th></tr>
<tr><td></td><td></td><td></td><td>8 个有符号或者无符号 16 位数左移，补 0</td></tr>
<tr><td>_mm_sll_epi16 (__m128i a, __m128i count)</td><td>Shift left</td><td>Logical</td><td><table><tr><th>R0</th><th>R1</th><th>...</th><th>R7</th></tr><tr><td>a0 << count</td><td>a1 << count</td><td>...</td><td>a7</td></tr></table></td></tr>
<tr><td>_mm_slli_epi32(__m128i a, int count)</td><td>Shift left</td><td>Logical</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 << count</td><td>a1 << count</td><td>a2 << count</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_sll_epi32 (__m128i a, __m128i count)</td><td>Shift left</td><td>Logical</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 << count</td><td>a1 << count</td><td>a2 << count</td><td>a3</td></tr></table></td></tr>
<tr><td>_mm_slli_epi64(__m128i a, int count)</td><td>Shift left</td><td>Logical</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0 << count</td><td>a1 << count</td></tr></table></td></tr>
<tr><td>_mm_sll_epi64 (__m128i a, __m128i count)</td><td>Shift left</td><td>Logical</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0 << count</td><td>a1 << count</td></tr></table></td></tr>
<tr><td>_mm_srai_epi16(__m128i a, int count)</td><td>Shift right</td><td>Arithmetic</td><td><table><tr><th>R0</th><th>R1</th><th>...</th><th>R7</th></tr><tr><td>a0 >> count</td><td>a1 >> count</td><td>...</td><td>a7</td></tr></table>右移 8 个有符号 16 位，补符号位</td></tr>
<tr><td>_mm_sra_epi16 (__m128i a, __m128i count)</td><td>Shift right</td><td>Arithmetic</td><td><table><tr><th>R0</th><th>R1</th><th>...</th><th>R7</th></tr><tr><td>a0 >> count</td><td>a1 >> count</td><td>...</td><td>a7</td></tr></table>右移 8 个有符号 16 位，补符号位</td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>移位类型</th><th>简要描述</th></tr>
<tr><td>_mm_srai_epi32(__m128i a, int count)</td><td>Shift right</td><td>Arithmetic</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 >> count</td><td>a1 >> count</td><td>a2 >> count</td><td>a3</td></tr></table>右移 4 个有符号 32 位，补符号位</td></tr>
<tr><td>_mm_sra_epi32(__m128i a, __m128i count)</td><td>Shift right</td><td>Arithmetic</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0 >> count</td><td>a1 >> count</td><td>a2 >> count</td><td>a3</td></tr></table>右移 4 个有符号 32 位，补符号位</td></tr>
<tr><td>_mm_srli_si128(__m128i a, int imm)</td><td>Shift right</td><td>Logical</td><td><table><tr><th>R</th></tr><tr><td>srl(a, imm*8)</td></tr></table></td></tr>
<tr><td>_mm_srli_epi16(__m128i a, int count)</td><td>Shift right</td><td>Logical</td><td><table><tr><th>R0</th><th>R1</th><th>...</th></tr><tr><td>srl(a0, count)</td><td>srl(a1, count)</td><td>...</td></tr></table>右移 8 个有符号或者无符号 16 位数，补 0</td></tr>
<tr><td>_mm_srl_epi16(__m128i a, __m128i count)</td><td>Shift right</td><td>Logical</td><td><table><tr><th>R0</th><th>R1</th><th>...</th></tr><tr><td>srl(a0, count)</td><td>srl(a1, count)</td><td>...</td></tr></table>右移 8 个有符号或者无符号 16 位数，补 0</td></tr>
<tr><td>_mm_srli_epi32(__m128i a, int count)</td><td>Shift right</td><td>Logical</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th></tr><tr><td>srl(a0, count)</td><td>srl(a1, count)</td><td>srl(a2, co</td></tr></table>右移 4 个有符号或无符号 32 位数，补 0</td></tr>
</table>

指令	操作	移位类型	简要描述
_mm_srl_epi32 (__m128i a, __m128i count)	Shift right	Logical	| R0 | R1 | ... | | srl(a0, count) | srl(a1, count) | ... | 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	| R0 | R1 | | srl(a0, count) | srl(a1, count) | 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	| R0 | R1 | | srl(a0, count) | srl(a1, count) | 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	| R0 | R1 | ... | | (a0 == b0) ? 0xff : 0x0 | (a1 == b1) ? 0xff : 0x0 | ... | 比较 16 个有符号或无符号的 8 位数
_mm_cmpeq_epi16	Equality	| R0 | R1 | ... | | (a0 == b0) ? 0xffff : 0x0 | (a1 == b1) ? 0xffff : 0x0 | ... |

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td></td><td></td><td>比较 8 个有符号或无符号的 16 位数</td></tr>
<tr><td>_mm_cmpeq_epi32</td><td>Equality</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th></tr><tr><td>(a0 == b0) ? 0xffffffff : 0x0</td><td>(a1 == b1) ? 0xffffffff : 0x0</td><td>(a2 == b2) ? 0xffffffff : 0x0</td></tr></table>比较 4 个有符号或无符号的 32 位数</td></tr>
<tr><td>_mm_cmpgt_epi8</td><td>Greater Than</td><td><table><tr><th>R0</th><th>R1</th><th>...</th></tr><tr><td>(a0 > b0) ? 0xff : 0x0</td><td>(a1 > b1) ? 0xff : 0x0</td><td>...</td></tr></table>比较 16 个有符号 8 位数</td></tr>
<tr><td>_mm_cmpgt_epi16</td><td>Greater Than</td><td><table><tr><th>R0</th><th>R1</th><th>...</th></tr><tr><td>(a0 > b0) ? 0xffff : 0x0</td><td>(a1 > b1) ? 0xffff : 0x0</td><td>...</td></tr></table>比较 8 个有符号 16 位</td></tr>
<tr><td>_mm_cmpgt_epi32</td><td>Greater Than</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th></tr><tr><td>(a0 > b0) ? 0xffffffff : 0x0</td><td>(a1 > b1) ? 0xffffffff : 0x0</td><td>(a2 > b2) ? 0xffffffff : 0x0</td></tr></table>比较 4 个有符号 32 位</td></tr>
<tr><td>_mm_cmplt_epi8</td><td>Less Than</td><td><table><tr><th>R0</th><th>R1</th><th>...</th></tr><tr><td>(a0 < b0) ? 0xff : 0x0</td><td>(a1 < b1) ? 0xff : 0x0</td><td>...</td></tr></table></td></tr>
</table>

指令	操作	简要描述
_mm_cmplt_epi16	Less Than	R0: (a0 < b0) ? 0xffff : 0x0 R1: (a1 < b1) ? 0xffff : 0x0 ...: ...
_mm_cmplt_epi32	Less Than	R0: (a0 < b0) ? 0xffffffff : 0x0 R1: (a1 < b1) ? 0xffffffff : 0x0 R2: (a2 < b2) ? 0xffffffff : 0x0

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	R0: (double)b R1: a1 转换 1 个有符号 64 位数到 DPFP
__int64 _mm_cvtsd_si64(__m128d a)	Convert according to rounding	R: (__int64) a0 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	R: (__int64) a0 转换低位的 DPFP到 1 个 64 位有符号整数, 截尾模式

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_cvtepi32_ps(__m128i a)</td><td>Convert to SP FP</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R</th></tr><tr><td>(float) a0</td><td>(float) a1</td><td>(float) a2</td><td>(</td></tr></table>转换 4 个有符号 32 位整数</td></tr>
<tr><td>_mm_cvtps_epi32</td><td>Convert from SP FP</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(int) a0</td><td>(int) a1</td><td>(int) a2</td><td>(int) a3</td></tr></table></td></tr>
<tr><td>_mm_cvttps_epi32</td><td>Convert from SP FP using truncate</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>(int) a0</td><td>(int) a1</td><td>(int) a2</td><td>(int) a3</td></tr></table>转换 4 个 SPFP为有符号的 32 位整数，截尾模式</td></tr>
</table>

整型移动操作

原型在 emmintrin.h 中

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_cvtsi32_si128 (int a)</td><td>Move and zero</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a</td><td>0x0</td><td>0x0</td><td>0x0</td></tr></table></td></tr>
<tr><td>_mm_cvtsi64_si128 (__int64 a)</td><td>Move and zero</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a</td><td>0x0</td></tr></table></td></tr>
<tr><td>_mm_cvtsi128_si32 (__m128i a)</td><td>Move lowest 32 bits</td><td><table><tr><th>R</th></tr><tr><td>a0</td></tr></table></td></tr>
</table>

指令	操作	简要描述
_mm_cvtsi128_si64	Move lowest 64 bits	R a0

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	R *p 地址 p 必须是 16 字节对齐
_mm_loadu_si128(__m128i const*p)	Load	R *p 地址 p 不需要 16 字节对齐
_mm_loadl_epi64	Load and zero	R0 R1 *p[63:0] 0x0

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	R0 R1 q0 q1

指令	操作	简要描述
		有符号
_mm_set_epi32(int i3, int i2, int i1, int i0)	Set four integer values	R0 R1 R2 R3 i0 i1 i2 i3 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	R0 R1 ... R15 b0 b1 ... b15 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	R0 R1 ... R15 b0 b1 ... b15 有符号
_mm_set1_epi64(__m64 q)	Set two integer values	R0 R1 q q
_mm_set1_epi32(int i)	Set four integer values	R0 R1 R2 R3 i i i i 有符号
_mm_set1_epi16(short w)	Set eight integer values	R0 R1 ... R7 w w w w
_mm_set1_epi8(char b)	Set sixteen integer values	R0 R1 ... R15 b b b b

指令	操作	简要描述
_mm_setr_epi64 (__m64 q0, __m64 q1)	Set two integer values in reverse order	R0 R1 q0 q1
_mm_setr_epi32 (int i0, int i1, int i2, int i3)	Set four integer values in reverse order	R0 R1 R2 R3 i0 i1 i2 i3
_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	R0 R1 ... R7 w0 w1 ... w7
_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	R0 R1 ... R15 b0 b1 ... b15
_mm_setzero_si128	Set to zero	R 0x0

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	*p a 不破坏 caches 将数据存入 p，p 必须是 16 字节对齐

指令	操作	简要描述
_mm_stream_si32(int *p, int a)	Store	*p a
_mm_store_si128(__m128i *p, __m128i b)	Store	*p a p 必须是 16 字节对齐
_mm_storeu_si128	Store	*p a p 不需要 16 字节对齐
_mm_maskmoveu_si128(__m128i d, __m128i n, char *p)	Conditional store	if (n0[7]) | if (n1[7]) | ... p[0] := d0 | p[1] := d1 | ...
_mm_storel_epi64	Store lowest	*p[63:0] a0

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

指令	操作	简要描述
_mm_stream_pd(double *p, __m128d a)	Store	p[0] p[1] a0 a1 p 必须是 16 字节对齐
_mm_stream_si128(__m128i *p, __m128i a)	Store	*p a p 必须 16 字节对齐
_mm_stream_si32(int *p, int a)	Store	*p a
_mm_stream_si64(__int64 *p, __int64 a)	Store	*p a
_mm_clflush (void const*p)	Flush	*p a
_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

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_packs_epi16(__m128i a, __m128i b)</td><td>Packed Saturation</td><td><table><tr><th>R0</th><th>...</th><th>R7</th><th>R8</th><th>...</th></tr><tr><td>Signed Saturate (a0)</td><td>...</td><td>Signed Saturate (a7)</td><td>Signed Saturate (b0)</td><td>..</td></tr></table>打包 16 个有符号 16 位数到有符号 8 位，饱和模式</td></tr>
<tr><td>_mm_packs_epi32</td><td>Packed Saturation</td><td><table><tr><th>R0</th><th>...</th><th>R3</th><th>R4</th><th>...</th></tr><tr><td>Signed Saturate (a0)</td><td>...</td><td>Signed Saturate (a3)</td><td>Signed Saturate (b0)</td><td>...</td></tr></table>打包 8 个有符号 32 位数到有符号 16 位，饱和模式</td></tr>
<tr><td>_mm_packus_epi16</td><td>Packed Saturation</td><td><table><tr><th>R0</th><th>...</th><th>R7</th><th>R8</th><th>...</th></tr><tr><td>Unsigned Saturate (a0)</td><td>...</td><td>Unsigned Saturate (a7)</td><td>Unsigned Saturate (b0)</td><td>...</td></tr></table>打包 16 个有符号 16 位数到无符号 8 位，饱和模式</td></tr>
<tr><td>_mm_extract_epi16 (__m128i a, int imm)</td><td>Extraction</td><td><table><tr><th>R0</th></tr><tr><td>(imm == 0) ? a0: ((imm == 1) ? a1: ... (i</td></tr></table>提取</td></tr>
<tr><td>_mm_insert_epi16 (__m128i a, int b, int imm)</td><td>Insertion</td><td><table><tr><th>R0</th><th>R1</th><th>...</th></tr><tr><td>(imm == 0) ? b : a0;</td><td>(imm == 1) ? b : a1;</td><td>...</td></tr></table>插入</td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_movemask_epi8(__m128i a)</td><td>Mask Creation</td><td><table><tr><th>R0</th></tr><tr><td>a15[7] << 15 | a14[7] << 14 | ... a1[7] <<</td></tr></table></td></tr>
<tr><td>_mm_shuffle_epi32（__m128i a, int imm)</td><td>Shuffle</td><td>按照 imm的说明拖拽 4 个有符号或无符号 32 位数</td></tr>
<tr><td>_mm_shufflehi_epi16</td><td>Shuffle</td><td>按照 imm的说明拖拽高位的 4 个有符号或无符号 16 位数</td></tr>
<tr><td>_mm_shufflelo_epi16</td><td>Shuffle</td><td>按照 imm的说明拖拽低位的 4 个有符号或无符号 16 位数</td></tr>
<tr><td>_mm_unpackhi_epi8(__m128i a, __m128i b)</td><td>Interleave</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th><th>...</th><th>R14</th><th>R15</th></tr><tr><td>a8</td><td>b8</td><td>a9</td><td>b9</td><td>...</td><td>a15</td><td>b15</td></tr></table></td></tr>
<tr><td>_mm_unpackhi_epi16</td><td>Interleave</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th><th>R4</th><th>R5</th></tr><tr><td>a4</td><td>b4</td><td>a5</td><td>b5</td><td>a6</td><td>b6</td></tr></table></td></tr>
<tr><td>_mm_unpackhi_epi32</td><td>Interleave</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a2</td><td>b2</td><td>a3</td><td>b3</td></tr></table></td></tr>
<tr><td>_mm_unpackhi_epi64</td><td>Interleave</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a1</td><td>b1</td></tr></table></td></tr>
<tr><td>_mm_unpacklo_epi8</td><td>Interleave</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th><th>...</th><th>R14</th><th>R15</th></tr><tr><td>a0</td><td>b0</td><td>a1</td><td>b1</td><td>...</td><td>a7</td><td>b7</td></tr></table></td></tr>
<tr><td>_mm_unpacklo_epi16</td><td>Interleave</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th><th>R4</th><th>R5</th><th>R6</th><th>R7</th></tr><tr><td>a0</td><td>b0</td><td>a1</td><td>b1</td><td>a2</td><td>b2</td><td>a3</td><td>b3</td></tr></table></td></tr>
</table>

<table>
<tr><th>指令</th><th>操作</th><th>简要描述</th></tr>
<tr><td>_mm_unpacklo_epi32</td><td>Interleave</td><td><table><tr><th>R0</th><th>R1</th><th>R2</th><th>R3</th></tr><tr><td>a0</td><td>b0</td><td>a1</td><td>b1</td></tr></table></td></tr>
<tr><td>_mm_unpacklo_epi64</td><td>Interleave</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0</td><td>b0</td></tr></table></td></tr>
<tr><td>__m64 _mm_movepi64_pi64(__m128i a)</td><td>Move</td><td><table><tr><th>R0</th></tr><tr><td>a0</td></tr></table>返回低 64 位</td></tr>
<tr><td>__m128i _mm_movpi64_epi64 (__m64 a)</td><td>Move</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0</td><td>0X0</td></tr></table></td></tr>
<tr><td>__m128i _mm_move_epi64(__m128i a)</td><td>Move</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0</td><td>0X0</td></tr></table></td></tr>
<tr><td>__m128d _mm_unpackhi_pd(__m128d a, __m128d b)</td><td>Interleave</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a1</td><td>b1</td></tr></table>交织 DPFP</td></tr>
<tr><td>_mm_unpacklo_pd</td><td>Interleave</td><td><table><tr><th>R0</th><th>R1</th></tr><tr><td>a0</td><td>b0</td></tr></table></td></tr>
<tr><td>int _mm_movemask_pd(__m128d a)</td><td>Create mask</td><td><table><tr><th>R</th></tr><tr><td>sign(a1) << 1 | sign(a0)</td></tr></table>从 a 的两个 DPFP的标志位创造 1 个两 bit 位的掩码</td></tr>
<tr><td>__m128d _mm_shuffle_pd(__m128d a, __m128d b,</td><td>Select values</td><td></td></tr>
</table>

指令	操作	简要描述
int i)		

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。

R
*p;

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	R0: a0 - b0; R1: a1 + b1; R2: a2 - b2; R3: a3 + b
_mm_hadd_ps	Add	R0: a0 + a1; R1: a2 + a3; R2: b0 + b1; R3: b2 + b
_mm_hsub_ps	Subtracts	R0: a0 - a1; R1: a2 - a3; R2: b0 - b1; R3: b2 - b
_mm_movehdup_ps	Duplicates	R0: a1; R1: a1; R2: a3; R3: a3;

指令	操作	简要描述
_mm_moveldup_ps	Duplicates	R0 | R1 | R2 | R3 a0; | a0; | a2; | a2;

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	R0 | R1 a0 - b0; | a1 + b1;
_mm_hadd_pd	Add	R0 | R1 a0 + a1; | b0 + b1;
_mm_hsub_pd	Subtract	R0 | R1 a0 - a1; | b0 - b1;
_mm_loaddup_pd(double const * dp)	Duplicate	R0 | R1 *dp; | *dp;
_mm_movedup_pd(__m128d a)	Duplicate	R0 | R1 a0; | a0;

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	Selects integer words from 2	

指令	操作	简要描述
_mm_blend_epi16(__m128i v1, __m128i v2, const int mask)	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	Sign extend 2 words into	

指令	操作	简要描述
_mm_cvtepi16_epi64(__m128i a)	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	

指令	操作	简要描述
__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,	Single float double precision rounding	

指令	操作	简要描述
__m128d s1) __m128d _mm_ceil_sd(__m128d dst, __m128d s1)		
__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	

指令	操作	简要描述
	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	

指令	操作	简要操作
_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 numberof set bits in a data operation	
int _mm_popcnt_u64(unsigned __int64 v)	Counts numberof 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

指令	描述
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 madeup 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 madeup 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.

指令	描述
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

指令	描述
	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.

指令	描述
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.

指令	描述
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 mayquery 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

指令	描述
	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.