SSE/AVX

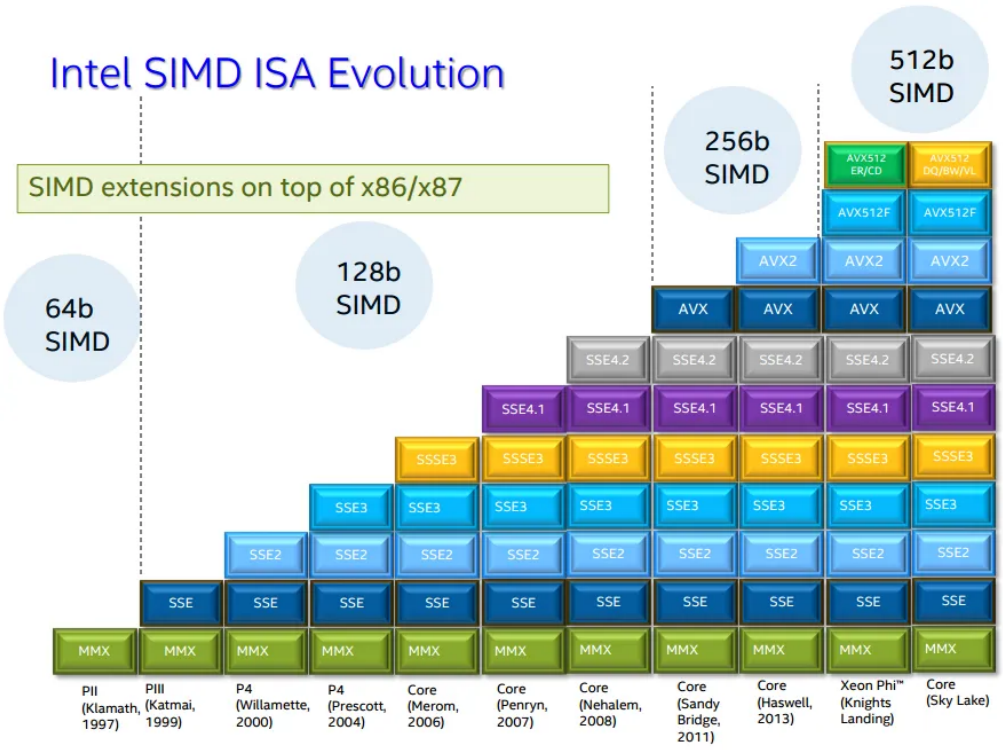
焦伟鹏，2023-08-06初稿

简介

`SSE`指令集是英特尔提供的基于`SIMD`（**单指令多数据，也就是说同一时间内，对多个不同的数据执行同一条命令**）的硬件加速指令，通过使用寄存器来进行并行加速。经过几代的迭代，最新的`AVX512`已经极大地扩展了指令集的功能。

历史发展

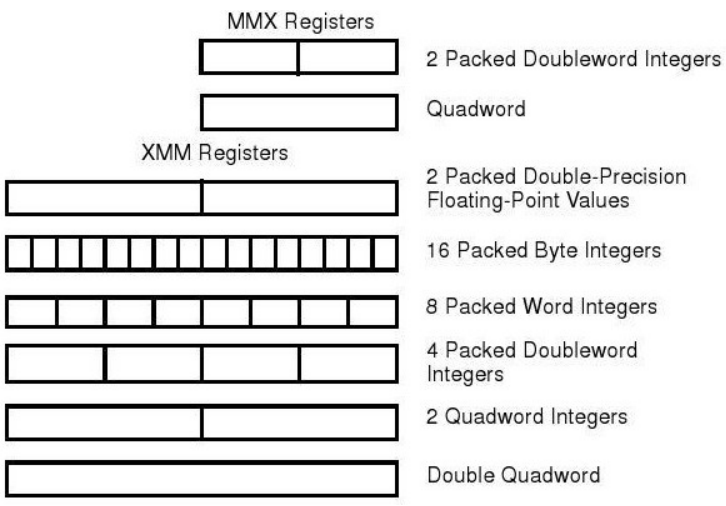
1993年，Intel公司推出了奔腾处理器，该类型处理器拥有两条执行流水线，和当时的处理器相比，可以同时执行两条指令，实现超标量性能。1996年，P6系列处理器中的奔腾II处理器引入了英特尔MMX技术，这是是最早的SIMD扩展指令。后续又相继推出了SSE、SSE2、SSE3、SSSE3和SSE4指令。2008年，Intel公司宣布将推出全新的Sandy Bridge微架构，并将引入AVX指令集。此后，Intel公司相继推出了AVX2和AVX512指令集扩展。



架构特性

Year Released	Name	Width (bits)	Width (FP words)
1996	MMX	64	2
1999	SSE	128	4
2011	AVX	256	8
2013	AVX-512	512	16

`MMX指令集（Multi Media eXtension）`率先在Pentium处理器中使用。MMX指令集只支持整型的向量操作，通过别名机制与*x87* FPU共用相同的位宽64bit数据寄存器组，不能与浮点数同时运行。在使用MMX指令时，需要将寄存器内的浮点数据保存。



`SSE指令集（Streaming SIMD Extension）`是对由MMX指令集引进的SIMD模型的扩展。SSE为了解决MMX指令集的限制，引进了8个专用的128bit位宽浮点寄存器XMM0~XMM7，与MMX使用的FPU寄存器组是相互独立的。后来Intel又陆续推出了SSE2、SSE3、SSE4，这使得SSE指令系列

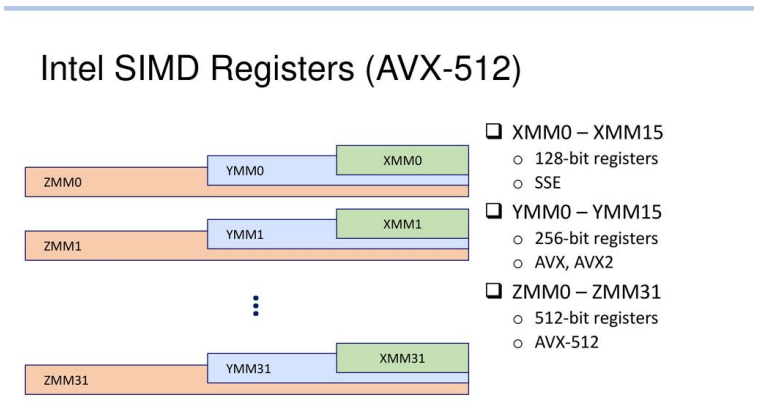
同时拥有了浮点数学运算功能和整数运算功能，XMM寄存器数量也由8个增加到了16个。

MMX与SSE关系：MMX和SSE的XMM寄存器组相互独立，理论上MMX与SSE指令可以并行执行。

	AMD	Intel
SSE 1	Athlon 64, 2003[3]	Pentium III, 1999[3]
SSE 2	Athlon 64, 2003	Pentium 4, 2000
SSE 3	Athlon 64 “Venice”, 2004	Pentium 4 “Prescott”, 2004
SSSE 3	Bobcat and Bulldozer, 2011	Penryn, 2007; Bonnell, 2008.
SSE 4.1	Jaguar, 2013; Bulldozer, 2011	Penryn, 2007; Silvermont, 2013.
FMA	Piledriver, 2012; not supported in Jaguar	Haswell, 2013
AVX 1	Jaguar, 2013; Bulldozer, 2011	Sandy Bridge, 2011
AVX 2	Excavator, 2015	Haswell, 2013 but only “Core i” and Xeon models, most Pentium and Celeron CPUs don’t support that.

AVX指令集合（Advanced Vector Extensions）是在Sandy Bridge架构下引入的新指令集，AVX在之前的SSE128bit位宽基础上扩展为256bit位宽，寄存器的个数并没有变化，还是16个。在汇编语言中，寄存器被新命名为 **ymm0** 到 **ymm15**，它们的低128位仍然可以以xmm0到xmm15的名字访问。AVX同时支持32位和64位浮点数，但是并不完全支持整型，在随后的 **AVX2** 中增加了对整型数据运算的支持。当前AVX最新的架构是 AVX512，寄存器个数32个，寄存器位宽扩展为512bit，寄存器被新命名为 **zmm0** 到 **zmm31**。

注：AVX指令集兼容旧的SSE指令集，在寄存器上共用一套寄存器组。SSE 和 AVX 各自有16个寄存器，SSE 的16个寄存器为 XMM0 - XMM15，AVX的16个寄存器为YMM0 - YMM15，XMM是128位寄存器，而YMM是256位寄存器。YMM寄存器是对XMM寄存器的扩展，在AVX中，YMM 低128 位等价于一个XMM寄存器，即在任意的AVX指令中，可以同时使用YMM寄存器和XMM寄存器。



	MMX	SSE(X86)	SSE(X64)	AVX	AVX2
寄存器	MM0-MM7	XMM0-XMM7	XMM0-XMM15	YMM0-YMM15	YMM0-YMM15
浮点		128bit	128bit	256bit	256bit
整型	64bit	128bit	128bit	128bit	256bit

硬件支持

查看硬件支持的SSE/AVX指令集种类可以通过以下3种方式：

windows查看指令支持

下载CPU-Z工具查看

linux查看指令支持

终端输入：`cat /proc/cpuinfo`

cpuid查看指令支持

x86芯片结构中，CPUID是处理器提供的一个指令操作码，能够让软件利用它分析出处理器的信息。可以在程序中使用CPUID查看处理器的类型和是否能够使用MMX/SSE指令集。

代码见附录：`cpuid查看硬件指令集`

指令索引

英特尔官方intrinsics索引：

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

编译环境

头文件

SSE和AVX指令集有多个不同版本，其函数也包含在对应版本的头文件里。

若不关心具体版本则可以使用 `<intrin.h>` 包含所有版本的头文件内容。

```
#include <mmintrin.h>   //MMX  4个64位寄存器
#include <xmmintrin.h>  //SSE(include mmintrin.h)  8个128位寄存器
#include <emmintrin.h>  //SSE2(include xmmintrin.h)  16个128位寄存器
#include <pmmintrin.h>  //SSE3(include emmintrin.h)  16个128位寄存器
#include <tmmintrin.h>  //SSSE3(include pmmintrin.h)  16个128位寄存器
#include <smmintrin.h>  //SSE4.1(include tmmintrin.h)  16个128位寄存器
#include <nmmintrin.h>  //SSE4.2(include smmintrin.h)  16个128位寄存器
#include <wmmintrin.h>  //AES(include nmmintrin.h)  加密算法专用
#include <immintrin.h>  //AVX(include wmmintrin.h)  16个256位寄存器
#include <intrin.h>     //(include immintrin.h)
#include <zmmintrin.h>  //AVX512
```

编译选项

除了头文件以外，我们还需要添加额外的编译选项，才能保证代码被编译成功。各版本的SSE和AVX都有单独的编译选项，比如-msseN，-mavxN(N表示版本编号)。经过简单测试后发现，此类编译选项支持向下兼容，比如-msse4可以编译SSE2的函数，-mavx也可以兼容各版本的SSE。

头文件	宏	编译器参数
avx2intrin.h	`__AVX2__`	-mavx2
avxintrin.h	`__AVX__`	-mavx

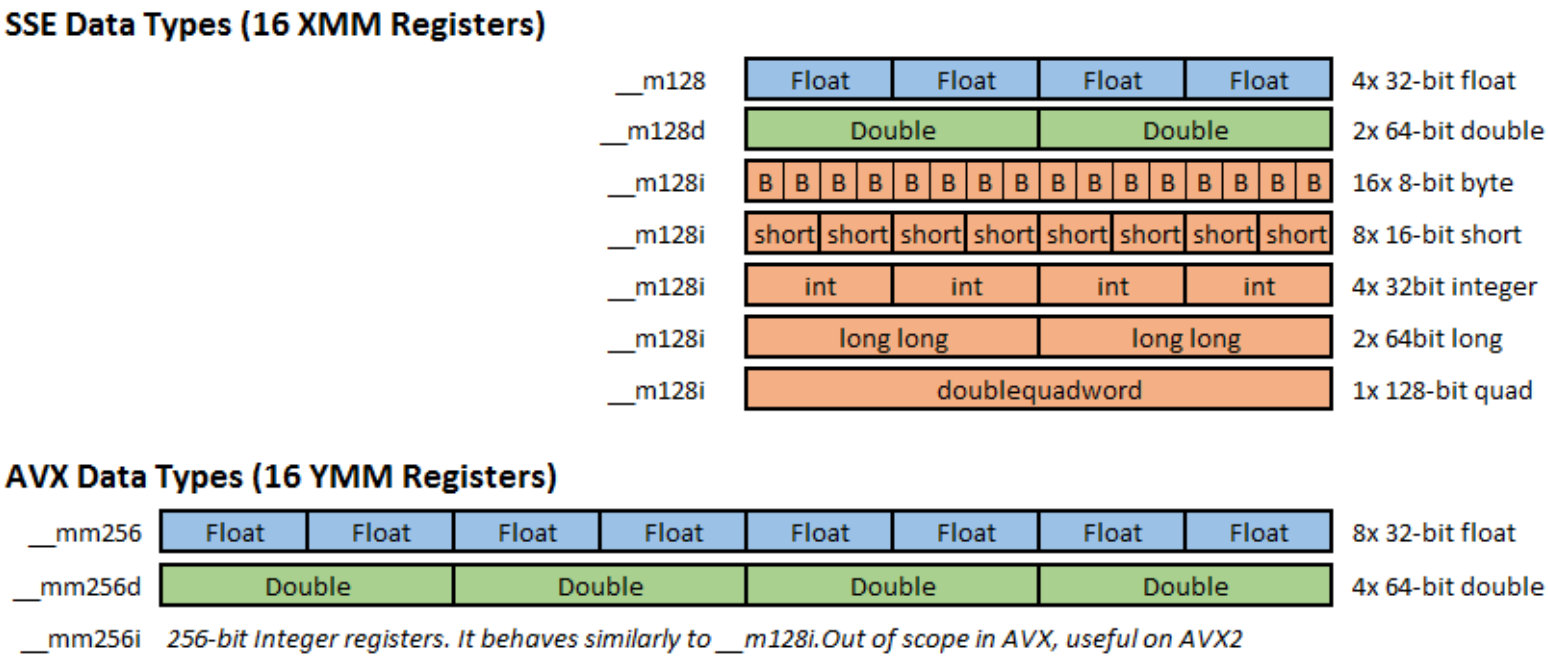
emmintrin.h	`__SSE2__`	-msse2
nmmintrin.h	`__SSE4_2__`	-msse4.2
xmmintrin.h	`__SSE__`	-msse
mmintrin.h	`__MMX__`	-mmmx

指令使用

数据格式

SSE 有三种类型定义 __m128，__m128d 和 __m128i，分别用以表示单精度浮点型、双精度浮点型和整型。
AVX 有三种类型定义 __m256，__m256d 和 __m256i，分别用以表示单精度浮点型、双精度浮点型和整型。

类型	SSE	AVX
单精度	`__m128`	`__m256`
双精度	`__m128d`	`__m256d`
整型	`__m128i`	`__m256i`



64位MM寄存器（MM0~MM7）： m64
128位SSE寄存器（XMM0~XMM15）：__m128、__m128d、__m128i
256位AVX寄存器（YMM0~YMM15）：__m256、__m256d、__m256i
512位AVX寄存器（ZMM0~ZMM31）：__m512、__m512d、__m512i

函数格式

SSE/AVX intrinsic functions 的命名习惯如下

`__<return_type>` `_<vector_size>` `_<intrin_op>` `_<suffix>`

`return_type`	如 m128、m256 和 m512 代表函数的返回值类型，m128 代表128位的向量，m256代表256位的向量，m512代表512位的向量
`vector_size`	如 mm、mm256 和 mm512 代表函数操作的数据向量的位长度，mm 代表 128 位的数据向量（SSE），mm256 代表256

	位的数据向量（AVX 和 AVX2），mm512 代表512位的数据向量
`intrin_op`	如 set、add 和 max 非常直观的解释函数功能。函数基础功能可以分为数值计算、数据传输、比较和转型四种
`suffix`	代表函数参数的数据类型，操作的数据粒度

一般情况下函数的命名格式如下

`_<mm/mm256/mm512>_<intrin_op>_<suffix>`

`suffix` 操作的数据类型

suffix is composed by a p (all the elements), ep (extended packed), s (single, the first element of the vector) and an additional string s (float), d (double), i128 (integer 128bit), i64 (integer 64bit), u64 (unsigned 64bit), i32 (integer 32bit), u32 (unsigned 32bit), i16 (integer 16bit), u16 (unsigned 16bit), i8 (integer 8bit), u8 (unsigned 8bit), valid combination are:

	s	d	i128	i64	u64	i32	u32	i16	u16	i8	u8
p	ps	pd				pi32	pu32	pi16	pu16	pi8	pu8
ep				epi64		epi32	epu32	epi16	epu16	epi8	epu8
s	ss		si128	si64							

pi和epi的区别在于MMX处理__mm64使用pi，SSE处理__mm128使用epi。

`<suffix>` 为操作数的类型，前一个或前两个字母代表这个操作数是packed（p），还是extended packed（ep），还是scaler（s），后面的字母代表数据类型。如下：

s：单精度浮点数

d：双精度浮点数

i128：有符号128bit整数

i64：有符号64bit整数

u64：无符号64bit整数

i32：有符号32bit整数

u32：无符号32bit整数

i16：有符号16bit整数

u16：无符号16bit整数

i8：有符号8bit整数

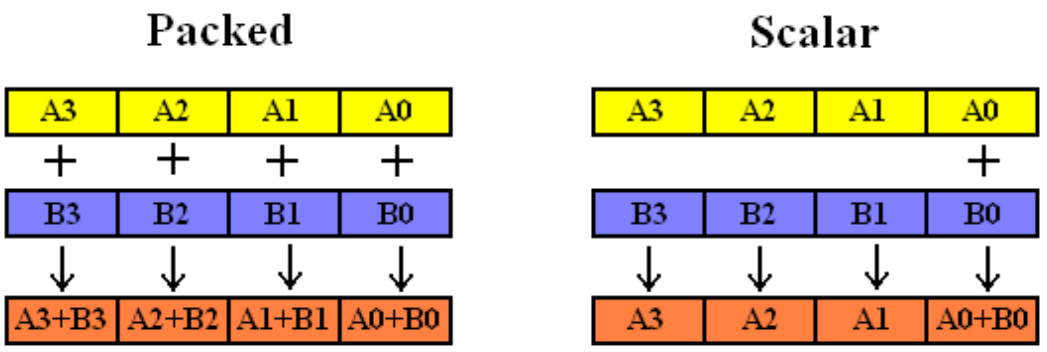
u8：无符号8bit整数

`intrin_op` 操作命令还配有一些可选的修饰符，表示一些特殊的作用，比如从内存对齐，逆序加载等

可选的修饰符	示例	描述
u	loadu	Unaligned memory: 对内存未对齐的数据进行操作
s	subs/adds	Saturate: 饱和计算将考虑内存能够存储的最小/最大值。
h	hsub/hadd	Horizontally: 在水平方向上做加减法
hi/lo	mulhi	高/低位
r	setr	Reverse order: 逆序初始化向量
fm	fmadd	Fused-Multiply-Add(FMA)运算，单一指令进行三元运算

packed和scaler：

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。



packed数据从右向左存储，也就是低位到高位存储，如

```
double a[2] = {1.0, 2.0};
__m128d t = _mm_load_pd(a);
```

那么在寄存器中，就是这样的

```
127----063----000
|  2.0 |  1.0 |
```

内存对齐

`u` 作为修饰表示着无需内存对齐，根据Intel® Intrinsics Guide介绍：

> _mm_store_ps : mem_addr must be aligned on a 16-byte boundary or a general-protection exception may be generated.

> _mm_storeu_ps : mem_addr does not need to be aligned on any particular boundary.

也就是说不加u的版本需要原数据有16字节内存对齐，否则在读取的时候就会触发边界保护产生异常。内存对齐要求的字节数就是指令需要处理的字节数，而要求内存对齐也是为了能够一次访问就完整地读到数据，从而提升效率。

`xx字节对齐的意思是要求数据的地址是xx字节的整数倍，128位宽的SSE要求` `16` `字节内存对齐，而256位宽的AVX函数则是要求` `32` `字节内存对齐。`

创建变量时设置N字节对齐可以用：

1. __declspec(align(N))，MSVC专用关键字
2. __attribute__((__aligned__(N)))，GCC专用关键字
3. alignas(N)，C++11关键字，不过我这里测试只能指定到16，否则就会warning并且无法生效。

```
alignas(16)                     float A[SIZE]; // C++11
__declspec(align(16))           float B[SIZE]; // MSVC
__attribute__((__aligned__(16))) float C[SIZE]; // GCC
```

对于new或malloc这种申请的内存也有相应的设置方法：

1. _aligned_malloc(size, N)，包含在<stdlib.h>头文件中，与malloc相比多了一个参数N用于指定内存对齐。注意！用此方法申请的内存需要用 _aligned_free() 进行释放。
2. new((std::align_val_t) N)，C++17新特性，需要在GCC7及以上版本使用 -std=c++17 编译选项开启。

```
float *A = new ((std::align_val_t)32) float[SIZE];             // C++17
delete[] A;
float *B = (float *)_aligned_malloc(sizeof(float) * SIZE, 32); // <stdlib.h> windows
_aligned_free(B);                                              // 用于释放_aligned_malloc申请的内存
```

结构体对齐:push按照用户定义的n字节对齐，n为1、2、4、8等，pop还原系统默认对齐方式

```
#pragma pack(push,n)
struct T
```

```
{
    //...
}
#pragma pack(pop)
```

函数类型

数据传输

数据读取

操 作	数 据	描 述
loadu	ps pd si128	不对齐读取
load	ss sd ps pd si128	对齐读取
load1	ps pd	读取首个数据
loadr	ps pd	逆向读取
lddqu	si128	读取不对齐的整数
loadl	pi pd	读取到低位
loadh	pi pd	读取到高位

AVX新增：`gather` `maskload`

gather 指令从地址不连续的内存中取出多个元素，组成一个向量

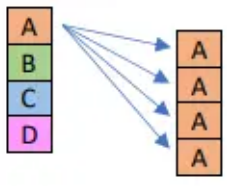
maskload根据掩码从内存加载元素，maskload可用来对付末尾不足256位的数据

数据设置

操 作	数 据	描 述
set	ps pd epi64 epi32 epi16 epi8	按输入设置
set1	ps pd epi64 epi32 epi16 epi8	全部设置为某值
setr	ps pd epi64 epi32 epi16 epi8	反向设置
setzero	ps pd si128	置零

AVX新增：`broadcast`

把标量拷贝到所有向量中，与set1类似



数据存储

操 作	数 据	描 述
storeu	ps pd si128	不对齐读取
store	ss sd ps pd si128	对齐读取
store1	ps pd	读取首个数据
storer	ps pd	逆向读取

storel	pi pd	读取到低位
storeh	pi pd	读取到高位
maskmove	si64 si128	根据掩码把数据存到内存中，指令非常耗时

AVX新增：`maskstore`

maskload根据掩码存储数据到内存

avx512才有scatter指令

缓存操作

操 作	数 据	描 述
prefetch	char*	预取数据到缓存层次中
stream	ps si128 si32 pi pd	不经过缓存的存储
stream_load	si128	不经过缓存的加载
sfence	void	写串行化
lfence	void	读串行化
mfence	void	读写都串行化
_mm_clflush	void*	使包含所有级别的缓存层次结构中的p的缓存行无效并刷新。

内存屏障

1. 阻止屏障两边的指令重排序

2. 强制把写缓冲区/高速缓存中的脏数据等写回主内存，让缓存中相应的数据失效（意思就是确保读到的数据都是内存中的最新的数据，确保数据的有效性）

- lfence，是一种Load Barrier 读屏障。在读指令前插入读屏障，可以让高速缓存中的数据失效，重新从主内存加载数据
- sfence，是一种Store Barrier 写屏障。在写指令之后插入写屏障，能让写入缓存的最新数据写回到主内存
- mfence，是一种全能型的屏障，具备ifence和sfence的能力

内存屏障使用例子

- LoadLoad屏障（读读屏障）：对于这样的语句Load1；LoadLoad；Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。
- StoreStore屏障（写写屏障）：对于这样的语句Store1；StoreStore；Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。
- LoadStore屏障（读写屏障）：对于这样的语句Load1；LoadStore；Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。
- StoreLoad屏障（写读屏障）：对于这样的语句Store1；StoreLoad；Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。

缓存一致性

CLFLUSH（Cache Line Flush，缓存行刷回）能够把指定缓存行（Cache Line）从所有级缓存中淘汰，若该缓存行中的数据被修改过，则将该数据写入主存。下一次从该缓存行进行读取时，必须再次从主内存中读取。

数据获取

操 作	数 据	描 述
extract	epi8 epi16 epi32 epi64	依据标签提取
movemask	ps pd epi8 pi8	判断负数位置

```
_mm_movemask_ps(__m128 a)
 a[0]为负数标记数为 1 otherwise 0
 a[1]为负数标记数为 2 otherwise 0
 a[2]为负数标记数为 4 otherwise 0
 a[3]为负数标记数为 8 otherwise 0

 __m128 a=_mm_setr_ps(-5.0f,10.0f,-325.0625f,81.125f);
int cnt=_mm_movemask_ps(a); // cnt=1+0+4+0=5
```

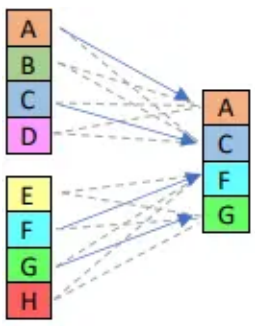
读写寄存器

内嵌原语	操作	对应SSE指令
unsigned int _mm_getcsr(void);	返回控制寄存器的内容	STMXCSR
void _mm_setcsr(unsigned int i);	设置控制寄存器的内容	LDMXCSR

排列组合

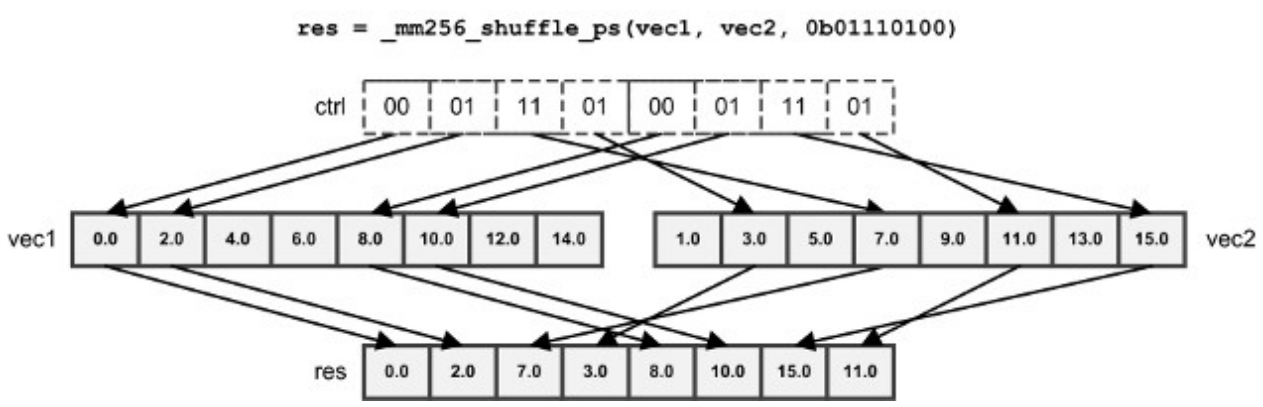
操 作	数 据	描 述
shuffle	ps pd epi64 epi32 epi16 epi8	按输入设置
unpackhi	ps pd epi64 epi32 epi16 epi8	全部设置为某值
unpacklo	ps pd epi64 epi32 epi16 epi8	反向设置
move	ss sd epi64	复制
movehl	ps	复制高位
movelh	ps	复制低位
insert	epi8 epi16 epi32 epi64 ps	依据掩码决定插入元素位置

_mm_shuffle_ps

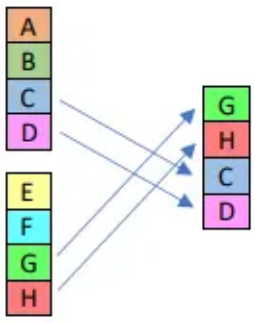


分别选择ab中的两个64bit数，放入结果中。如imm8为10(2)，将会选择a0放到最低位，b1放到最高位。

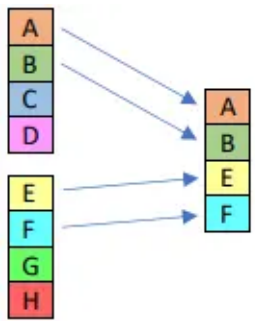
imm8 使用上与unsigned char类似，但类型一般是int，所以0-7bit才是有效数据位



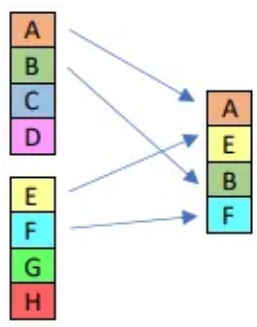
_mm_movehl_ps



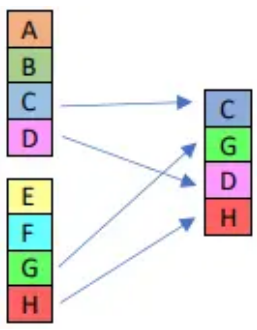
_mm_movelh_ps



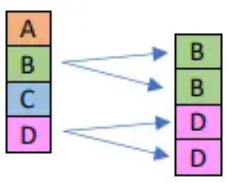
_mm_unpacklo_ps



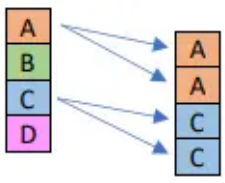
_mm_unpackhi_ps



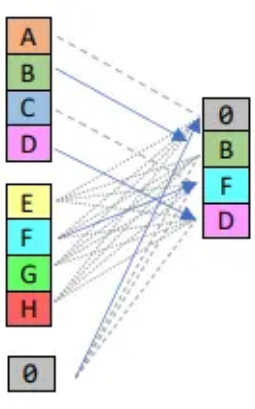
_mm_movehdup_ps



_mm_moveldup_ps



_mm_insert_ps

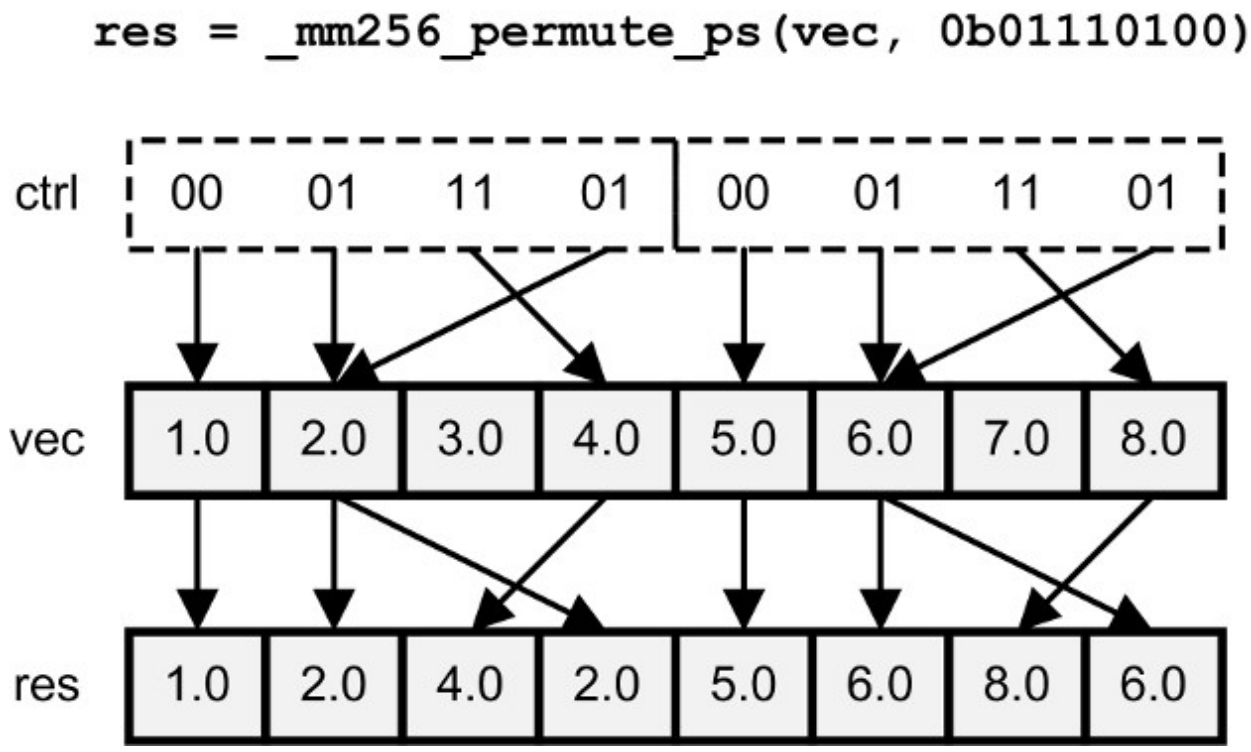



```
__m128 _mm_insert_ps (__m128 a, __m128 b, const int imm8)
```

```
int bid = value of bit 6-7 for id
float bv = b[bid];
bid = value of bit 4-5 for id
r[0] = (bid == 0) ? bv : a[0];
r[1] = (bid == 1) ? bv : a[1];
r[2] = (bid == 2) ? bv : a[2];
r[3] = (bid == 3) ? bv : a[3];
r[0] = (0x1 & id == 1) ? 0.0 : r[0];
r[1] = (0x2 & id == 1) ? 0.0 : r[1];
r[2] = (0x4 & id == 1) ? 0.0 : r[2];
r[3] = (0x8 & id == 1) ? 0.0 : r[3];
掩码的第 6、7 位决定了取 b 的哪个元素取代 a 中元素，而第 4、5 位决定了 a 的哪个元素被取代。
例子：
__m128 a=_mm_setr_ps(1.0f,-1.0f,1.5f,105.5f);
__m128 b=_mm_setr_ps(-5.0f,10.0f,-325.0625f,81.125f);
__m128 d=_mm_insert_ps(a,b,0xb0);
// 0xb0 =10 11 0000 , 10 = 2 表示选b[2]=-325.0625f, 11 = 3 表示a[3] , b[2]插入a[3]的位置
// d = 1.000000  -1.000000  1.500000  -325.062500
// 掩码的低4用来置0 1011 1100 这种情况 a[0]=a[0],a[1]=a[1],a[2]=a[2],a[3]=a[3]
// 1011 1110 这种情况 a[0]=a[0],a[1]=0,a[2]=0,a[3]=a[3]
// 1011 1111 这种情况 a[0]=0,a[1]=a[1],a[2]=0,a[3]=0
```

AVX新增：`permute`

数据类型	描述
_mm_permute_ps/pd _mm256_permute_ps/pd	根据8位控制值从输入向量中选择元素
(2)_mm256_permute4x64_pd/ (2)_mm256_permute4x64_epi64	根据8位控制值从输入向量中选择64位元素
_mm256_permute2f128_ps/pd	基于8位控制值从两个输入向量中选择128位块
_mm256_permute2f128_si256	基于8位控制值从两个输入向量中选择128位块
_mm_permutevar_ps/pd _mm256_permutevar_ps/pd	根据整数向量中的位从输入向量中选择元素
(2)_mm256_permutevar8x32_ps (2)_mm256_permutevar8x32_epi32	使用整数向量中的索引选择32位元素（浮点和整数）



类型转换

关键字 cvt （convert）

举例：

整数转浮点 __m128 _mm_cvtepi32_ps (__m128i a)

整数转整数 __m128i _mm_cvtepi32_epi64 (__m128i a)

浮点转整数 __m128i _mm_cvtps_epi32 (__m128 a)

浮点转浮点 __m128d _mm_cvtps_pd (__m128 a)

数值运算

算术型

操 作	数 据	描 述
add	ss ps epi8 epi16 epi32 epi64 sd pd	加
hadd	pd ps epi16 epi32	相邻数据相加
sub	ss ps epi8 epi16 epi32 epi64 sd pd	减
hsub	pd ps epi16 epi32	相邻数据相减
addsub	ps pd	偶数索引减，奇数索引加
mul	ss ps epi32 epu32 sd pd	乘
mulhi	epi16 epu16	取乘法结果的高位
mullo	epi16 epi32	取乘法结果的低位
div	ss ps sd pd	除
max	ss ps epi16 epu8 sd pd epi8 epi32 epu32 epu16	最大值
min	ss ps epi16 epu8 sd pd epi8 epi32 epu32 epu16	最小值
minpos	epu16	返回最小值及其索引
rsqrt	ss ps	开方的倒数
sqrt	ss ps sd pd	开方
ceil	pd ps sd ss	向上取整
floor	pd ps sd ss	向下取整
abs	epi8 epi16 epi32	求绝对值
avg	epu8 epu16	求均值
dp	pd ps	依据 mask 做乘法
blend	pd ps epi16	类似 C 中的三元运算符
blendv	pd ps epi8	类似 C 中的三元运算符
sign	epi8 epi16 epi32	依据参数改变参数符号
sad	epu8	计算差的绝对值
round	pd ps sd ss	舍入
rcp	ps ss	求倒数
popcnt	u32 u64	求二进制数中为 1 的位数

AVX新增：`FMA`指令

操 作	数 据	描 述
fmadd	ss sd ps pd	乘加（a*b+c）
fnmadd	ss sd ps pd	c-a*b
fmsub	ss sd ps pd	乘减 (a*b-c)
fnmsub	ss sd ps pd	-(a*b+c)

逻辑运算型

操 作	数 据	描 述
and	ps si128	与
andnot	ps si128	前一参数取反再与后一参数
xor	ps si128 pd	异或
or	ps si128	或
sra	epi16 epi32	算术右移
srl	epi16 epi32 epi64	逻辑右移
sll	epi16 epi32 epi64	左移

不直接支持 非，但可以通过 andnot 实现，以 si128 为例，展示 andnot 操作的定义如下：

```
__m128i _mm_andnot_si128(__m128i a, __m128i b);
r = (~a) & b;
如果 b 的位模式为全 1，那么 andnot 就实现了 not 运算。
```

比较

操 作	数 据	描 述
cmp	pd ps sd ss	比较
test_all_ones	/	测试是否所有位都是 1
test_all_zeros	/	测试和掩码的位与结果是否是 0
cmpeq	ss ps epi8 epi16 epi32 sd pd epi64	比较是否相等
cmplt	ss ps sd pd	测试是否小于
cmple	ss ps sd pd	小于等于
cmpgt	ss ps epi8 epi16 epi32 sd pd epi64	大于
cmpge	ss ps sd pd	大于等于
cmpneq	ss ps sd pd	不等于
cmpnlt	ss ps sd pd	不小于
cmpngt	ss ps sd pd	不大于
cmpnge	ss ps sd pd	不大于等于
cmpnle	ss ps sd pd	不小于等于

比较指令返回的结果是掩码，即如果比较成立的话，则对应位置的值为 1，否则为 0。
下面给出通用的 cmp 指令定义，其他的 cmp* 指令的语义都可归为 cmp 的某种特殊情况。

```
__m128 _mm_cmp_ps(__m128 a, __m128 b, const int mask)
r[0] = (a[0] op b[0]) ? oxffffffff:0;
r[1] = (a[1] op b[1]) ? oxffffffff:0;
r[2] = (a[2] op b[2]) ? oxffffffff:0;
r[3] = (a[3] op b[3]) ? oxffffffff:0;
其中 mask 的数值表示了执行的具体比较操作（op）
```

常用编译制导语句

- `#pragma ivdep` ：告诉编译器，下面的循环中没有变量依赖关系
- `#pragma vector aligned` ：提示编译器进行向量化，并且编译器将使用对齐的数据存取指令
- `#pragma vector temporal` ：提示编译器进行向量化，并且数据从内存中先取到 Cache 再读写
- `#pragma vector nontemporal` ：提示编译器进行向量化，并且数据不经过 Cache，直接从内存进行读写
  如果不指定是 `temporal` 还是 `nontemporal`，编译器会自动决定。
- `#pragma simd` ：要求编译器进行向量化
- `#pragma prefetch var:hint:distance` ：KNC only，指示编译器进行预取。 `var` 指定要预取的数组； `hint` 选 0 表示取到 L1 Cache，1 表示取到 L2 Cache； `distance` 表示提前多少个向量单元进行预取，比如其取值为 8 时，则提前 8 *16 个 float 或者 8* 8 个 double 进行预取。
  除了编译制导语句，也可以使用 Intrinsic 进行手动预取。函数是 `void _mm_prefetch(char const*address, int hint)`， `address` 是要预取回来的数据的头地址， `hint` 同制导语句含义。在手动预取时，需要设置 `-opt-prefetch=0` 或 `#pragma noprefetch` 以关闭编译器预取，避免发生不可知的冲突。
- `#pragma unroll(<UNROLL_NUM>)` ：提示编译器进行 `<UNROLL_NUM>` 路循环展开，因为太短的循环体不利于进行指令调度。

下面是一个等效 Demo：

```
for (int i = 0; i < N; i += 4) // 假设 N 是 4 的倍数
{
    y[i    ] += alpha * x[i];
    y[i + 1] += alpha * x[i + 1];
    y[i + 2] += alpha * x[i + 2];
    y[i + 3] += alpha * x[i + 3];
}
// 上面的代码等效于下面的代码
#pragma unroll(4)
for (int i = 0; i < N; i++)
    y[i] += alpha * x[i];
```

附录

指令集特性

指令集	特性
MMX	向量定点运算，与x87浮点共用寄存器组，使用时需要保存FPU数据
SSE	1. 添加8个128位寄存器XMM0~MMX7 2. 操作4个单精度浮点数 3. 支持打包(Packed)和标量(Scaler)操作
SSE	1. 8个128位XMM寄存器扩展为16个(2003年AMD增加) 2. 添加整数向量运算 3. 支持双精度浮点向量运算

	4. 添加缓存控制指令 5. 添加浮点数到整数的转换指令
SSE3	1. 寄存器内水平操作,例如打包在一个128位MMX寄存器内的2个64位整数,可以利用新指令对这两个整数进行算数运算。 2. 多线程优化指令,在超线程下提高性能
SSE4	1. STTNI(String and Text New Instructions)指令来帮助开发者处理字符搜索和比较,旨在加速对XML文件的解析 2. CRC32指令,帮助计算循环冗余校验值
AVX	16个128位XMM寄存器扩展为16个256位YMM寄存器
AVX2	256位整型数据的支持， 三运算指令（3-Operand Instructions）
AVX512	16个256位YMM寄存器扩展为32个512位ZMM寄存器

cpuid查看硬件指令集

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

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

class InstructionSet
{
    // forward declarations
    class InstructionSet_Internal;

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

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

    static bool MSR(void) { return CPU_Rep.f_1_EDX_[5]; }
    static bool CX8(void) { return CPU_Rep.f_1_EDX_[8]; }
    static bool SEP(void) { return CPU_Rep.f_1_EDX_[11]; }
    static bool CMOV(void) { return CPU_Rep.f_1_EDX_[15]; }
    static bool CLFSH(void) { return CPU_Rep.f_1_EDX_[19]; }
    static bool MMX(void) { return CPU_Rep.f_1_EDX_[23]; }
```

```
    static bool FXSR(void) { return CPU_Rep.f_1_EDX_[24]; }
    static bool SSE(void) { return CPU_Rep.f_1_EDX_[25]; }
    static bool SSE2(void) { return CPU_Rep.f_1_EDX_[26]; }

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

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

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

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

private:
    static const InstructionSet_Internal CPU_Rep;

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

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

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

            // Capture vendor string
```

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

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

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

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

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

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

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

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

        int nIds_;
        int nExIds_;
        std::string vendor_;
        std::string brand_;
        bool isIntel_;
        bool isAMD_;
        std::bitset<32> f_1_ECX_;
        std::bitset<32> f_1_EDX_;
        std::bitset<32> f_7_EBX_;
        std::bitset<32> f_7_ECX_;
```

```
        std::bitset<32> f_81_ECX_;
        std::bitset<32> f_81_EDX_;
        std::vector<std::array<int, 4>> data_;
        std::vector<std::array<int, 4>> extdata_;
    };
};

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

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

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

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

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

```
257     support_message("TBM",          InstructionSet::TBM());
258     support_message("XOP",          InstructionSet::XOP());
259     support_message("XSAVE",        InstructionSet::XSAVE());
260 }
```

内存对齐规则

a.结构体中的第一个成员在与结构体变量偏移量为0的地址处，**结构体首地址默认按自然对齐数对齐**

b.其它成员变量要对齐到一个名叫对齐数的整数倍的地址处.补充:对齐数=编译器默认的一个对齐数与该结构体中的成员变量大小中的较小值.vs默认对齐数为8. **32位系统自然对齐数为4字节，64位系统自然对齐数为8字节** 。

c.结构体总大小为最大对齐数(每个成员变量都有一个对齐数)的整数倍.

d.如果存在了嵌套结构体的情况,嵌套的结构体则对齐到自己的成员变量中最大对齐数的整数倍处,包含该嵌套结构体的结构体大小为其成员变量(包含嵌套结构体)最大对齐数的整数倍

```
typedef struct
{
    char a;
    short b;
    int c;
} TestStruct1;

typedef struct
{
    char a;
    int c;
    short b;
} TestStruct2;
```

（1）结构体1 a地址为结构体首地址，b为short类型，对齐字节数为2，应该偏离首地址2字节的倍数，int是4字节，应该偏离4字节地址的倍数

（2）结构体1最大的成员是int类型4字节，按照规则c，结构体1的大小为8字节

相对起始地址的偏移量	成员位置
0	char
1	
2	short
3	
4	int
5	
6	
7	

（3）结构体2 a地址为结构体首地址，int是4字节，应该偏离4字节地址的倍数，所以偏移量是4，b为short类型，对齐字节数为2，应该偏离首地址2字节的倍数，所以放在8

（4）结构体2最大的成员是int类型4字节，按照规则c，结构体2的大小为12字节

相对起始地址的偏移量	成员位置
0	char
1	
2	
3	
4	int
5	
6	
7	
8	short
9	
10	
11	

改变结构体对齐系数

相对起始地址的偏移量	对齐系数：8		对齐系数：2		对齐系数：1	
	成员位置	数值	成员位置	数值	成员位置	数值
0	char	0x11	char	0x11	char	0x11
1		0x55		0xdf	short	0x22
2	short	0x22	short	0x22		0x22
3		0x22		0x22	int	0x33
4	int	0x33	int	0x33		0x33
5		0x33		0x33		0x33
6		0x33		0x33		0x33
7		0x33		0x33		
TestStruct1 长度	8		8		7	

相对起始地址的偏移量	对齐系数：8		对齐系数：2		对齐系数：1	
	成员位置	数值	成员位置	数值	成员位置	数值
0	char	0x11	char	0x11	char	0x11
1		0x7f		0x00	int	0x33
2		0x00	int	0x33		0x33
3		0x00		0x33		0x33
4	int	0x33		0x33		0x33
5		0x33		0x33	short	0x22
6		0x33	short	0x22		0x22
7		0x33		0x22		
8	short	0x22				
9		0x22				
10		0x00				
11		0x00				
TestStruct2 长度	12		8		7	

此外结构体变量不是自然对齐的；

如果一个变量的内存地址正好是这个变量的长度的整数倍，那么这个变量就是自然对齐的。`char`，`short`，`int`，`double` 等基本类型变量是自然对齐的：

数组类型是依据其元素类型对齐：如果第一个元素可以对齐，那么后面的元素自然也可以对齐；如果第一个元素不可以对齐，那么后面的元素无法保证对齐。

如果数组元素是基本类型，那么这个数组后面的元素是可以对齐的；
如果数组元素是构造类型，那么这个数组的元素就不保证对齐了。

动态内存-字节对齐

C11

```
1 void * aligned_alloc (size_t alignment, size_t size)
2 void free(void *ptr)
```

POSIX

```
1 int posix_memalign (void **memptr, size_t alignment, size_t size)
2 void free(void *ptr)
```

window

```
1 void * _aligned_malloc(size_t size, size_t alignment);
2 void _aligned_free(void *ptr)
```

英特尔

```
1 void* _mm_malloc (int size, int align)
2 void _mm_free (void *p)
```

C++17

```cpp
int* p = new((std::align_val_t) 64) int[100];
delete[] p;
```

手写

```cpp
#include<iostream>

void* aligned_malloc(size_t size, int alignment)
{
    // 分配足够的内存, 这里的算法很经典, 早期的STL中使用的就是这个算法

    // 首先是维护FreeBlock指针占用的内存大小
    const int pointerSize = sizeof(void*);

    // alignment - 1 + pointerSize这个是FreeBlock内存对齐需要的内存大小
    // 前面的例子sizeof(T) = 20, __alignof(T) = 16,
    // g_MaxNumberOfObjectsInPool = 1000
    // 那么调用本函数就是alignedMalloc(1000 * 20, 16)
    // 那么alignment - 1 + pointSize = 19
    const int requestedSize = size + alignment - 1 + pointerSize;

    // 分配的实际大小就是20000 + 19 = 20019
    void* raw = malloc(requestedSize);

    // 这里实Pool真正为对象实例分配的内存地址
    uintptr_t start = (uintptr_t)raw + pointerSize;
    // 向上舍入操作
    // 解释一下, __ALIGN - 1指明的是实际内存对齐的粒度
    // 例如__ALIGN = 8时, 我们只需要7就可以实际表示8个数(0~7)
    // 那么~(__ALIGN - 1)就是进行舍入的粒度
    // 我们将(bytes) + __ALIGN-1)就是先进行进位, 然后截断
    // 这就保证了我是向上舍入的
    // 例如byte = 100, __ALIGN = 8的情况
    // ~(__ALIGN - 1) = (1 000)B
    // ((bytes) + __ALIGN-1) = (1 101 011)B
    // (((bytes) + __ALIGN-1) & ~(__ALIGN - 1)) = (1 101 000 )B = (104)D
    // 104 / 8 = 13, 这就实现了向上舍入
    // 对于byte刚好满足内存对齐的情况下, 结果保持byte大小不变
    // 记得《Hacker's Delight》上面有相关的计算
    // 这个表达式与下面给出的等价
    // ((((bytes) + _ALIGN - 1) * _ALIGN) / _ALIGN)
    // 但是SGI STL使用的方法效率非常高
    void* aligned = (void*)((start + alignment - 1) & ~(alignment - 1));

    // 这里维护一个指向malloc()真正分配的内存
    *(void**)((uintptr_t)aligned - pointerSize) = raw;

    // 返回实例对象真正的地址
    return aligned;
}


// 这里是内部维护的内存情况
//                    这里满足内存对齐要求
//                              |
// ------------------------------------------------------------------------
// | 内存对齐填充 | 维护的指针 | 对象1 | 对象2 | 对象3 | ..... | 对象n |
// ------------------------------------------------------------------------
// ^                     | 指向malloc()分配的地址起点
// |                     |
// -----------------------
template<typename T>
void aligned_free(T * aligned_ptr)
{
    if (aligned_ptr)
    {
        free(((T**)aligned_ptr)[-1]);
```

```
    }
}

bool isAligned(void* data, int alignment)
{
    // 又是一个经典算法, 参见<Hacker's Delight>
    return ((uintptr_t)data & (alignment - 1)) == 0;
}

void main() {
    int totalsize = 10;
    int* data = (int*)aligned_malloc(sizeof(int)*totalsize, 32);

    if (isAligned(data, 32)) {
        std::cout << "isAligned\n";
    }
    memset(data, 0, sizeof(int)*totalsize);
    data[5] = 1;

    for (int i = 0; i < totalsize; i++) {
        std::cout << data[i] << " ";
    }
    std::cout << "\n";
    aligned_free<int>(data);
    std::cout << "aligned_free\n";
    while(1){}
}
```

```
#include <memory>

void* aligned_malloc(size_t size, size_t alignment)
{
    size_t offset = alignment - 1 + sizeof(void*);
    void * originalP = malloc(size + offset);
    size_t originalLocation = reinterpret_cast<size_t>(originalP);
    size_t realLocation = (originalLocation + offset) & ~(alignment - 1);
    void * realP = reinterpret_cast<void*>(realLocation);
    size_t originalPStorage = realLocation - sizeof(void*);
    *reinterpret_cast<void**>(originalPStorage) = originalP;
    return realP;
}

void aligned_free(void* p)
{
    size_t originalPStorage = reinterpret_cast<size_t>(p) - sizeof(void*);
    free(*reinterpret_cast<void**>(originalPStorage));
}

int main()
{
    void * p = aligned_malloc(1024, 64);
    printf("0x%p\n", p);
    aligned_free(p);
    return 0;
}

```

获取__m128i的数据

获取8bit整数使用：

```
#define _mm_extract_epi8(x, imm) \
((((imm) & 0x1) == 0) ?   \
```

```
_mm_extract_epi16((x), (imm) >> 1) & 0xff : \
_mm_extract_epi16(_mm_srli_epi16((x), 8), (imm) >> 1))
```

获取16bit整数使用：

```
int _mm_extract_epi16(__m128i a, int imm)
```

获取32bit整数使用：

```
#define _mm_extract_epi32(x, imm) \
_mm_cvtsi128_si32(_mm_srli_si128((x), 4 * (imm)))
```

获取64bit整数使用：

```
#define _mm_extract_epi64(x, imm) \
_mm_cvtsi128_si64(_mm_srli_si128((x), 8 * (imm)))
```

其中的imm，是数字的下标

硬件内存屏障

防止乱序

sfence

store fence 在 sfence 指令前面的写操作必须在 sfence 指令后边的写操作前完成

如图：如果没有 sfence，是不能保证操作 1 在操作 2 执行前就执行完的，有了 sfence 才能保证操作 1 和操作 2 的顺序

lfence

load fence 指令前的读操作必须在 lfence 指令后边的读操作执行前执行

mfence

mix fence 是集合了 sfence 和 lfence 的所有作用于一身，mfence 前边的读写操作必须在 mfence 后边读写操作开始之前完成

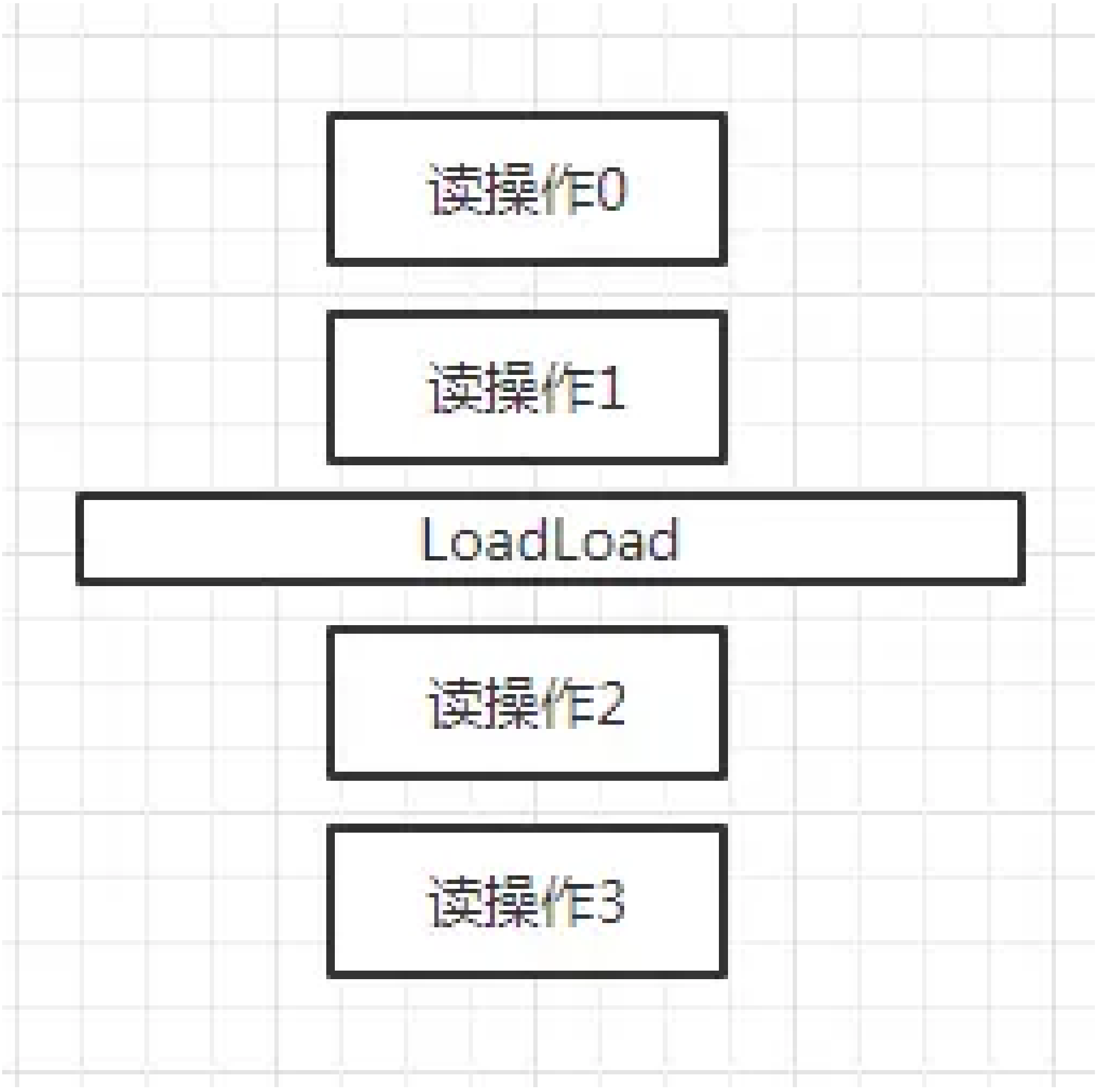
如图：mfence 可以保证 在执行读操作 2 和写操作 2 之前 必须执行完读操作 1 和写操作 1

内存屏障使用例子

LoadLoad 屏障

保证读操作的顺序，LoadLoad 前边的读操作必须在 LoadLoad 后边的读操作前完成

如图： LoadLoad 能保证 读操作 2 和者读操作 3 执行前 必须执行完读操作 0 和读操作 1



StoreStore 屏障

对比 LoadLoad 保证写操作有序

LoadStroe 屏障

对比 LoadLoad 保证读操作和写操作有序

StoreLoad 屏障

对比 LoadLoad 保证写操作和读操作有序

计算pi

```
#include<iostream>

#include <time.h>
#if _WIN32
#include <intrin.h>
#include <Windows.h>
#endif
#if __linux__
#include <immintrin.h>
#include <sys/time.h>
#endif


#include <random>
#include <time.h>

```

```
using std::default_random_engine;
using std::uniform_real_distribution;

using namespace std;

#if(1)
#define Print_TAG       "HPC_AVX2"
#define Debug_info(fmt,...)      {printf("%s info    -[fun:%-25.25s line:%05d:
]:",Print_TAG,__FUNCTION__,__LINE__);printf(fmt,##__VA_ARGS__);}

#if _WIN32

#define TINIT \
    LARGE_INTEGER cpu_freqence; \
    LARGE_INTEGER start; \
    LARGE_INTEGER end; \
    double run_time = 0.0; \
    QueryPerformanceFrequency(&cpu_freqence);

#define TIC \
  QueryPerformanceCounter(&start);

#define TOC(x) \
    QueryPerformanceCounter(&end); \
    run_time = (((end.QuadPart - start.QuadPart) * 1000.0f) / cpu_freqence.QuadPart); \
    Debug_info("%s taskes %f ms\r\n", x, run_time);

#elif __linux__
#define TINIT \
    struct timeval start, stop; \
    double elapsed_time;

#define TIC \
    gettimeofday(&start, NULL);

#define TOC(x) \
    gettimeofday(&stop, NULL); \
    elapsed_time = (stop.tv_sec - start.tv_sec) * 1000. + \
    (stop.tv_usec - start.tv_usec) / 1000.; \
    Debug_info("%s taskes %f ms\r\n", x, elapsed_time);

#else

#define TINIT
#define TIC
#define TOC(x)

#endif


/* 计算pi */

//正常的逐个累加运算
double compute_pi_naive(size_t dt) {

    double pi = 0.0;

    double delta = 1.0 / dt;

    for (size_t i = 0; i < dt; i++) {

        double x = (double)i / dt;

        pi += delta / (1 + x * x);
    }
    return pi * 4.0;
}


```

```cpp
double compute_pi_sim256(size_t dt) {

    alignas(32) double pi = 0.0;

    alignas(32) double delta = 1.0 / dt;

    __m256d ymm0, ymm1, ymm2, ymm3, ymm4;

    ymm0 = _mm256_set1_pd(1.0);

    ymm1 = _mm256_set1_pd(delta);

    ymm2 = _mm256_set_pd(0.0, delta, delta * 2, delta * 3);

    ymm4 = _mm256_setzero_pd();

    for (int i = 0; i < dt - 4; i += 4) {

        ymm3 = _mm256_set1_pd(i*delta);

        ymm3 = _mm256_add_pd(ymm3, ymm2);// 构造 x

        ymm3 = _mm256_mul_pd(ymm3, ymm3);// 构造 x^2

        ymm3 = _mm256_add_pd(ymm0, ymm3);// 构造 1 + x^2

        ymm3 = _mm256_div_pd(ymm1, ymm3);// 构造 delta / ( 1 + x^2 )

        ymm4 = _mm256_add_pd(ymm4, ymm3);// 叠加结果
    }

    alignas(32) double tmp[4];
    _mm256_store_pd(tmp, ymm4);

    pi += tmp[0] + tmp[1] + tmp[2] + tmp[3];

    return pi * 4.0;
}


int main() {//test_cal_pi

    TINIT;

    size_t dt = 134217728;

    double result1, result2;

    //普通函数计时
    TIC;
    result1 = compute_pi_naive(dt);
    TOC("naive")
    cout << "naive: " << result1 << endl;

    //simd256计时
    TIC;
    result2 = compute_pi_sim256(dt);
    TOC("simd256")
    cout << "simd256: " << result2 << endl;


    return 0;
}

```

指令测试

```cpp
#include <stdio.h>
#include<iostream>
#include <time.h>
#if _WIN32
#include <intrin.h>
#include <Windows.h>
#endif
#if __linux__
#include <immintrin.h>
#include <sys/time.h>
#endif

using namespace std;

bool isAligned(void* data, int alignment)
{
    // 又是一个经典算法, 参见<Hacker's Delight>
    return ((uintptr_t)data & (alignment - 1)) == 0;
}


int main()
{
    float* result=NULL;
    int *result_i=NULL;
    float A[16]={0.0f,0.1f,0.2f,0.3f,0.4f,0.5f,0.6f,0.7f,0.8f,0.9f,1.0f,1.1f,1.2f,1.3f,1.4f,1.5f};
    alignas(16) float B[16]={0.0f,0.1f,0.2f,0.3f,0.4f,0.5f,0.6f,0.7f,0.8f,0.9f,1.0f,1.1f,1.2f,1.3f,1.4f,1.5f};
    int C[16]={0,1,2,3,4,5,6,8,9,10,11,12,13,14,15};

    __m128 a,b,d;
    __m128i c,e;


    if (isAligned(&B[0], 16)) {
        std::cout << "isAligned\n";
    }

    // load
    result=(float*)&A;
    printf("[ref] input:        %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);

    a=_mm_loadu_ps(A);
    result=(float*)&a;
    printf("[sse] _mm_loadu_ps: %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);

    b=_mm_load_ps(B);
    result=(float*)&b;
    printf("[sse] _mm_load_ps:  %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);

    b=_mm_loadr_ps(B);
    result=(float*)&b;
    printf("[sse] _mm_loadr_ps: %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);

    b=_mm_load1_ps(B);
    result=(float*)&b;
    printf("[sse] _mm_load1_ps:  %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);

    b=_mm_load_ss(&B[1]);
    result=(float*)&b;
    printf("[sse] _mm_load_ss:  %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);

    b=_mm_loadh_pi(a,(__m64*)&A[0]);
    result=(float*)&b;
    printf("[sse] _mm_loadh_pi:  %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);

    b=_mm_loadl_pi(a,(__m64*)&A[0]);
    result=(float*)&b;
    printf("[sse] _mm_loadl_pi:  %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);

    c=_mm_loadu_si128((__m128i*)C);
```

```
    result_i=(int*)&c;
    printf("[sse] _mm_loadu_si128: %d  %d  %d  %d \r\n",result_i[0],result_i[1],result_i[2],result_i[3]);

    c=_mm_lddqu_si128((__m128i*)&C[4]);
    result_i=(int*)&c;
    printf("[sse] _mm_lddqu_si128: %d  %d  %d  %d \r\n",result_i[0],result_i[1],result_i[2],result_i[3]);

    // set
    b=_mm_setzero_ps();
    result=(float*)&b;
    printf("[sse] _mm_setzero_ps:  %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);

    b=_mm_set1_ps(A[1]);
    result=(float*)&b;
    printf("[sse] _mm_set1_ps:  %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);

    b=_mm_set_ps(A[0],A[1],A[2],A[3]); // high <- low
    result=(float*)&b;
    printf("[sse] _mm_set_ps:  %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);

    b=_mm_setr_ps(A[0],A[1],A[2],A[3]);
    result=(float*)&b;
    printf("[sse] _mm_setr_ps:  %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);


    a=_mm_setr_ps(1.0f,-1.0f,1.5f,105.5f);
    b=_mm_setr_ps(-5.0f,10.0f,-325.0625f,81.125f);
    d=_mm_insert_ps(a,b,0xb0); // 0xb0 =10 11 0000 , 10 = 2 表示选b[2]=-325.0625f, 11 = 3 表示a[3] , b[2]插入a[3]的位置
    // 1.000000  -1.000000  1.500000  -325.062500
    result=(float*)&d;
    printf("[sse] _mm_insert_ps:  %f  %f  %f  %f \r\n",result[0],result[1],result[2],result[3]);

    e=_mm_insert_epi32(c,0,2);
    result_i=(int*)&e;
    printf("[sse] _mm_insert_epi32: %d  %d  %d  %d \r\n",result_i[0],result_i[1],result_i[2],result_i[3]);

    __m128i v = _mm_set_epi32(3, 2, 1, 0); // initialise v to 4 x 32 bit int values
    int extract = _mm_extract_epi32(v, 3); // extract element 3 mast is 3 if imm8 set 4 is err
    printf("[sse] _mm_extract_epi32: %d \r\n",extract);

    int cnt=_mm_movemask_ps(b); // 1 2 4 8
    printf("[sse] _mm_movemask_ps: %d \r\n",cnt);

}
```