2021/03/23_x86逆向_第14课_浮点相关指令

笔记本： x86逆向-C
创建时间： 2021/3/23 星期二 11:46
作者： ileemi

定点

求 3.33 + 6.69 的结果：

```
int nNum = 333 + 669;
printf("%d.%d\r\n", nNum / 100, nNum % 100);
```

存在缺陷，精度越高，整数值的范围就越小。反之，精度越小，整数值的范围就越大。数据类型表示的数据范围空间是有限的。空间利用率不高。

浮点

> 能够有效的利用空间，提高数据类型范围空间的利用率。指明整数以及小数的位数。 采用 IEEE 浮点格式。

缺点：运算效率较低，使用硬件（浮点协处理器）去解决效率问题。

CPU遇到浮点指令就交给浮点协处理器去处理。

x87浮点指令

> Intel x87 FPU专门用于执行标量浮点计算，可以对单精度浮点（32位）、双精度浮点（64位）以及扩展双精度浮点（80位）进行计算，并顺从IEEE754标准。

寄存器：ST0~ST7（栈结构，一共只能存放8个数据，ST0为栈顶，ST7为栈底）。

常用浮点数指令表：

常用浮点数指令表

指令名称	使用格式	指 令 功 能
FLD	FLD IN	将浮点数 IN 压入 ST(0) 中。IN（mem 32/64/80）
FILD	FILD IN	将整数 IN 压入 ST(0) 中。IN（mem 32/64/80）
FLDZ	FLDZ	将 0.0 压入 ST(0) 中
FLD1	FLD1	将 1.0 压入 ST(0) 中
FST	FST OUT	ST(0) 中的数据以浮点形式存入 OUT 地址中。OUT(mem 32/64)
FSTP	FSTP OUT	和 FST 指令一样，但会执行一次出栈操作
FIST	FIST OUT	ST(0) 数据以整数形式存入 OUT 地址中。OUT(mem 32/64)
FISTP	FISTP OUT	和 FIST 指令一样，但会执行一次出栈操作
FCOM	FCOM IN	将 IN 地址数据与 ST(0) 进行实数比较，影响对应标记位
FTST	FTST	比较 ST(0) 是否为 0.0，影响对应标记位
FADD	FADD IN	将 IN 地址内的数据与 ST(0) 做加法运算，结果放入 ST(0) 中
FADDP	FADDP ST(N), ST	将 ST(N) 中的数据与 ST(0) 中的数据做加法运算，N 为 0 ～ 7 中的任意一个， 先执行一次出栈操作，然后将相加结果放入 ST(0) 中保存

代码示例：

```
int  main(int  argc,  char*  argv[])  {
    //  x87浮点指令
    float  f1  =  3.5f;
    float  f2  =  6.5f;
    float  ret  =  0.0f;
    __asm  {
        // 3.5入栈
        // 4.3入栈
        // 加法
        // 结果出栈
        fld  f1
        fld  f2

        //  指令后加  p后，  当前指令执行完后，对应寄存器上的值出栈
        fdivp  st(1),  st(0)
        //faddp  st(1),  st(0)
        //fsubp  st(1),  st(0)
        //fmulp  st(1),  st(0)
        fstp  ret

        //fadd  st(0),  st(1)
        //fst  ret
    }
    printf("%f\n",  ret);
    return  0;
}
```

> **在VS2019中对应的工程属性 --> "C/C++" --> "代码生成" --> "无增强指令(/arch:IA32)" 后编译器采用 "x87浮点指令进行算数运算"。**

代码示例：

```
float  f1  =  3.5f;
float  f2  =  6.5f;
printf("%f\n",  f1 + f2);

// 对应的汇编代码
fld  f1
fadd  f2 // 相加结果不用出栈，结果尽量放在栈顶，方便弹出
fstp  ret
```

多媒体扩展指令集 MMX

> MMX(Multi Media extension - 多媒体扩展指令集）指令集是 Intel公司于1996年推出的一项多媒体指令增强技术。MMX指令集中包括有57条多媒体指令，通过这些指令可以一次处理多个数据，在处理结果超过实际处理能力的时候也能进行正常处理，这样在软件的配合下，就可以得到更高的性能。

解决 "x87浮点指令" 不高效的问题。支持 **整型** 的并行运算。

MMX指令有8个64位寄存器（MM0~MM7），不过可惜都是借的FPU的，FPU原来有8个80位寄存器（ST(0)~ST(7)），现在用在了MMX上，所以用之后要加上一条EMMS指令，用以复位。

编译器在编译运算类的相关代码时依然采用 "x87浮点指令"，"double" 类型在读写内存时使用 "qword ptr" 进行解析，"float" 类型的变量在解析时使用 "dword ptr" 进行解析，如下图所示：

```
 float f1 = 3.5f;
000C1888 D9 05 38 7B 0C 00    fld       dword ptr [__real@40600000 (0C7B38h)]
000C188E D9 5D F8             fstp      dword ptr [f1]
 float f2 = 6.5f;
000C1891 D9 05 D8 7B 0C 00    fld       dword ptr [__real@40d00000 (0C7BD8h)]
000C1897 D9 5D EC             fstp      dword ptr [f2]
 float ret = 0.0f;
000C189A D9 EE                fldz
000C189C D9 5D E0             fstp      dword ptr [ret]
```

```
 double f1 = 3.5f;
00641888 DD 05 E8 7B 64 00    fld       qword ptr [__real@400c000000000000 (0647BE8h)]
0064188E DD 5D F4             fstp      qword ptr [f1]
 double f2 = 6.5f;
00641891 DD 05 F0 7B 64 00    fld       qword ptr [__real@401a000000000000 (0647BF0h)]
00641897 DD 5D E4             fstp      qword ptr [f2]
 double ret = 0.0f;
0064189A D9 EE                fldz
0064189C DD 5D D4             fstp      qword ptr [ret]
```

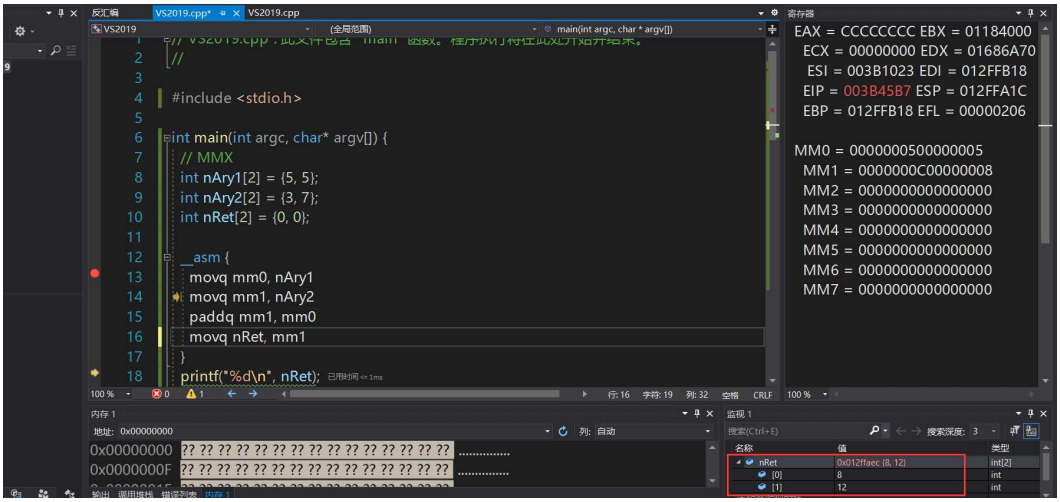
常用指令：

常用指令表

指令名称	使用格式	指 令 功 能
movd	*mmx, reg/mem32* *mmx, reg/mem32*	复制MMX寄存器中的低位双字到一个通用寄存器或内存中，也可以把通用寄存器或内存中的数据复制到MMX寄存器的低位双字中
movq	*mmx1, mmx2/mem64* *mmx1/mem64, mmx2*	把一个MMX寄存器的内容复制到另一个MMX寄存器中，这个指令也能被用来把一个内存区域中的内容复制到一个MMX寄存器中，或者把MMX寄存器中的内容复制到内存中
paddb	*mmx1, mmx2/mem64*	环绕方式,并行执行1个字节整型加法
paddd	*mmx1, mmx2/mem64*	环绕方式,并行执行4个字节整型加法
paddsb	*mmx1, mmx2/mem64*	饱和方式，并行执行有符号1个字节整型加法
paddsw	*mmx1, mmx2/mem64*	饱和方式，并行执行有符号2个字节整型加法
paddusb	*mmx1, mmx2/mem64*	饱和方式，并行执行无符号1个字节整型加法
paddusw	*mmx1, mmx2/mem64*	饱和方式，并行执行无符号2个字节整型加法
psubb	*mmx1, mmx2/mem64*	环绕方式,并行执行1个字节整型减法
psubw	*mmx1, mmx2/mem64*	环绕方式,并行执行2个字节整型减法
psubd	*mmx1, mmx2/mem64*	环绕方式,并行执行4个字节整型减法
psubsb	*mmx1, mmx2/mem64*	饱和方式，并行执行有符号1个字节整型加法

指令使用示例：

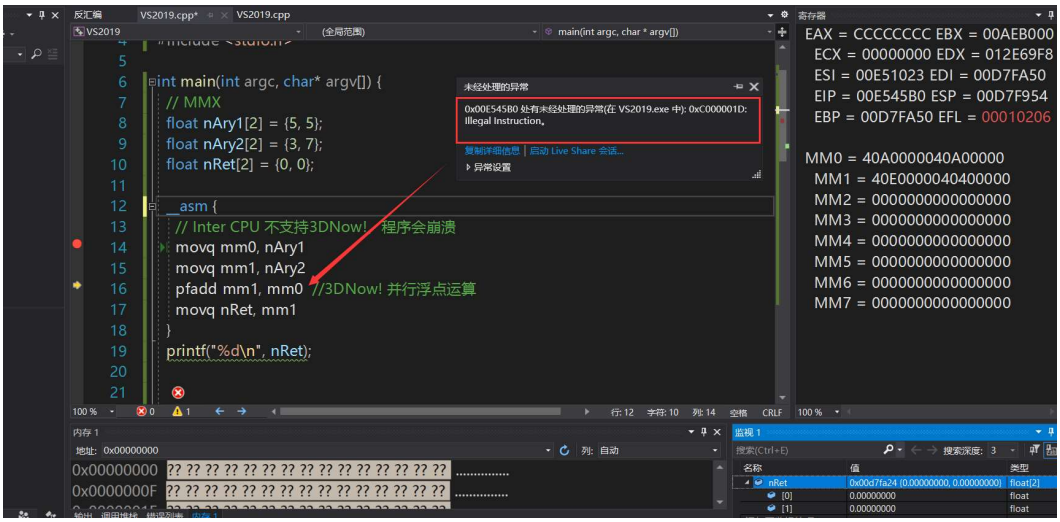
```
int  main(int  argc,  char*  argv[])  {
    int  nAry1[2]  =  {5,  5};
    int  nAry2[2]  =  {3,  7};
    int  nRet[2]  =  {0,  0};
  // MMX 多媒体扩展指令集 只扩展了整型的并行运算，浮点运算依然需要使用x87
    __asm  {
        movq  mm0,  nAry1
        movq  mm1,  nAry2
        paddq  mm1,  mm0
        movq  nRet,  mm1
    }
    printf("%d\n",  nRet);
    return  0;
}
```



AMD 3DNow! 指令集

> 3DNow!（据称是 "3D No Waiting!" 的缩写）是由AMD开发的一套SIMD多媒体指令集，支持单精度浮点数的矢量运算，用于增强x86架构的计算机在三维图像处理上的性能。

扩展了 Inter 的 MMX多媒体扩展指令集，支持单精度浮点数的矢量运算。**但后 Inter CPU 不支持 3DNow! 指令集，也就导致 使用 3DNow! 指令的程序在Inter CPU上运行会崩溃，如下图所示：**



3DNow! 部分指令：

Appendix A Recommended Substitutions for 3DNow!™ Instructions

Table A-1 lists the deprecated 3DNow!™ instructions and the recommended substitutions.

Table A-1. Substitutions for 3DNow!™ Instructions

64-Bit 3DNow!™ Instruction	128-Bit SSE Instruction	64-Bit MMX™ Instruction	Notes
FEMMS	N/A	EMMS (MMX)	
PAVGUSB	PAVGB	PAVGB	SSE and MMX™ instructions round according to the current rounding mode; 3DNow!™ instructions always round up.
PF2ID	CVTTPS2DQ		
PF2IW			CVTTPS2DQ may be used if 16-bit result is not necessary.
PFACC	HADDPS		
PFADD	ADDPS		
PFCMPEQ	CMPPS		
PFCMPGE	CMPPS		
PFCMPGT	CMPPS		
PFMAX	MAXPS		MAXPS may return -0.0.
PFMIN	MINPS		MINPS may return -0.0.
PFMUL	MULPS		
PFNACC	HSUBPS		
PFPNACC	ADDSUBPS		ADDSUBPS expects arguments in different positions from PFPNACC.
PFRCP			RCPSS may be used in conjunction with the Newton-Raphson algorithm.
PFRCPIT1			See PFRCP.
PFRCPIT2			See PFRCP.
PFRSQIT1			See PFRSQRT.
PFRSQRT			RSQRTSS may be used in conjunction with the Newton-Raphson algorithm.

SSE 指令

> SSE(Streaming SIMD Extensions)是英特尔在AMD的3DNow!发布一年之后，在其计算机芯片 Pentium III中引入的指令集，是继MMX的扩展指令集。

> SSE指令集提供了70条新指令。AMD后来在 Athlon XP中加入了对这个新指令集的支持。

在加快浮点运算的同时，改善了内存的使用效率，使内存速度更快。它对游戏性能的改善十分显著，按Intel的说法，SSE对下述几个领域的影响特别明显：3D几何运算及动画处理、图形处理（如 Photoshop)、视频编辑/压缩/解压（如MPEG和DVD)、语音识别以及声音压缩和合成等。

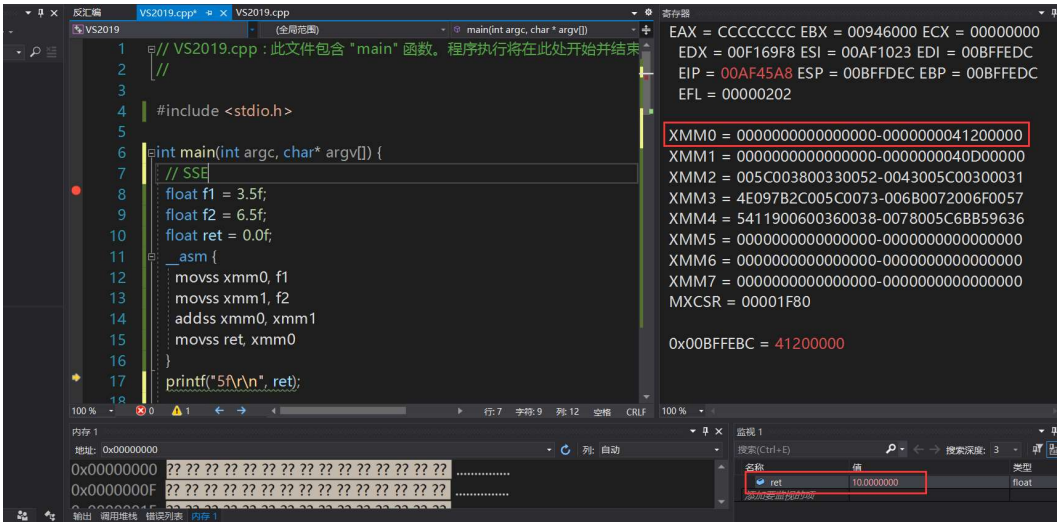
> **在VS2019中对应的工程属性 --> "C/C++" --> "代码生成" --> "流式处理 SIMD 扩展 (/arch:SSE) 或者 流式处理 SIMD 扩展 2 (/arch:SSE2)" 后编译器将采用 SSE 相关的指令参入运算并生成对应的汇编代码。**

常用指令表

指令名称	使用格式	指 令 功 能
MOVSS	xmm1,xmm2 xmm1,mem32 xmm2/mem32,xmm1	传送单精度数
MOVSD	xmm1,xmm2 xmm1,mem64 xmm2/mem64,xmm1	传送双精度数
MOVAPS	xmm1,xmm2/mem128 xmm1/mem128, xmm2	传送对齐的封装好的单精度数
MOVAPD	xmm1,xmm2/mem128 xmm1/mem128, xmm2	传送对齐的封装好的双精度数
ADDSS	xmm1, xmm2/mem32	单精度数加法
ADDSD	xmm1, xmm2/mem64	双精度数加法

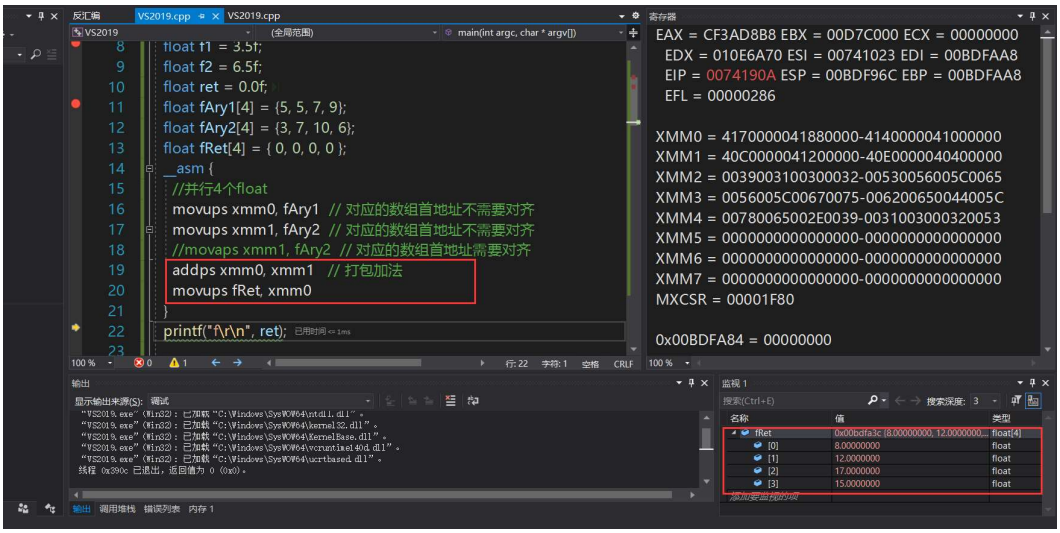
单运算，代码示例：

```
int  main(int  argc,  char*  argv[])  {
    float  f1  =  3.5f;
    float  f2  =  6.5f;
    float  ret  =  0.0f;
    __asm  {
        movss  xmm0,  f1
        movss  xmm1,  f2
        addss  xmm0,  xmm1
        movss  ret,  xmm0
    }
    printf("f\r\n",  ret);
    return  0;
}
```



并行运算，代码示例：

```
int main(int argc, char* argv[]) {
    float fAry1[4] = {5, 5, 7, 9};
    float fAry2[4] = {3, 7, 10, 6};
    float fRet[4] = { 0, 0, 0, 0 };
    __asm {
        movups xmm0, fAry1    // 对应的数组首地址不需要对齐
        movups xmm1, fAry2    // 对应的数组首地址不需要对齐
        //movaps xmm1, fAry2    // 对应的数组首地址需要对齐
        addps xmm0, xmm1      // 打包加法
        movups fRet, xmm0
    }
    return 0;
}
```



SSE1：

- SSE1主要是单精度浮点运算
- SSE有8个128位独立寄存器（XMM0~XMM7），16个字节（可同时做4个float运算）
- MM指64位MMX寄存器
- XMM指XMM寄存器

- m128指128位内存变量

SSE2：

- SSE2主要是双精度浮点运算
- SSE2与SEE1使用相同寄存器

AMD --> SSE5后，Inter 决定放弃使用 SSE 系列指令集。并推出 "AVX指令集"。

AVX 指令集

> AVX(Advanced Vector Extensio -- 高级向量扩展）。

AMX指令集是 Sandy Bridget和 Larrabee架构下的新指令集。AVX是在之前的128位扩展到和256位的单指令多数据流。而 Sandy Bridge 的单指令多数据流演算单元扩展到256位的同时数据传输也获得了提升，所以从理论上看CPU内核浮点运算性能提升到了2倍。

Intel AVX指令集，在单指令多数据流计算性能增强的同时也沿用了MMX/SSE指令集。不过和MMX/SSE的不同点在于增强的AVX指令，从指令的格式上就发生了很大的变化。x86(IA-32/ Intel64)架构的基础上增加了 prefix(Prefix)，所以实现了新的命令，也使更加复杂的指令得以实现，从而提升了x86CPU的性能。

寄存器：YMM0~YMM7（256位），一个寄存器8个float。**XMM 作为 YMM 的低128位。**

AVX常用指令

指令名称	使用格式	指 令 功 能
VMOVUPS	xmm1, xmm2/mem128 xmm1/mem128, xmm2 ymm1, ymm2/mem256 ymm1/mem256, ymm2	传送单精度数
VMOVUPD	xmm1, xmm2/mem128 xmm1/mem128, xmm2 ymm1, ymm2/mem256 ymm1/mem256, ymm2	传送双精度数
VADDPS	xmm1, xmm2, xmm3/mem128 ymm1, ymm2, ymm3/mem256	并行执行单精度数加法
VADDPD	xmm1, xmm2, xmm3/mem128 ymm1, ymm2, ymm3/mem256	并行执行双精度数加法
VSUBPS	xmm1, xmm2, xmm3/mem128 ymm1, ymm2, ymm3/mem256	并行执行单精度数减法
VSUBSD	xmm1, xmm2, xmm3/mem128 ymm1, ymm2, ymm3/mem256	并行执行双精度数减法

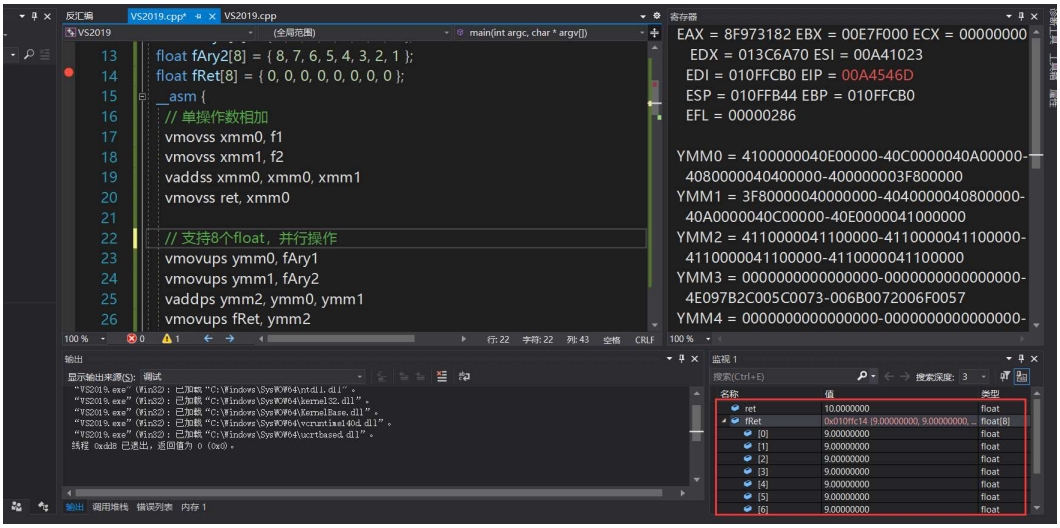
代码示例：

```
int main(int argc, char* argv[]) {
    // AVX
    float f1 = 3.5f;
    float f2 = 6.5f;
    float ret = 0.0f;
```

```
    float fAry1[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
    float fAry2[8] = { 8, 7, 6, 5, 4, 3, 2, 1 };
    float fRet[8] = { 0, 0, 0, 0, 0, 0, 0, 0 };
    __asm {
        // 单操作数相加
        vmovss xmm0, f1
        vmovss xmm1, f2
        vaddss xmm0, xmm0, xmm1
        vmovss ret, xmm0

        // 支持8个float，并行操作
        vmovups ymm0, fAry1
        vmovups ymm1, fAry2
        vaddps ymm2, ymm0, ymm1
        vmovups fRet, ymm2

        //转换类型 浮点转换整型
        vcvtss2si eax, xmm2 // AVX
        mov ebx, 99999999
        vmovdqa xmm4, xmm1
        vcvtsi2ss xmm3, xmm4, ebx // AVX
        //cvtsi2ss xmm3, ebx //SSE
    }
    printf("f\r\n", ret);
    return 0;
}
```



库函数完成并行运算

- MMX：mmintrin.h
- SSE1：xmmintrin.h
- SSE2：emmintrin.h
- SSE3：pmmintrin.h

- SSE4：smmintrin.h
- AVX：immintrin.h

MMX 使用示例：

```
__m128  m1 = { 0 };
__m128  m2 = { 0 };
__m128  m3 = { 0 };
m3 = _mm_add_ps(m1, m2);
```