目录

第 15 章 MMX 编程

多媒体扩展（Multi-Media Extension，MMX） 是为了提高视频、音频、图像等多媒体信号处理能力，Intel 公司设计的一套支持单指令多数据流（Single Instruction Multiple Data，SIMD）指令系统。从 Pentium Ⅱ开始引入到 IA-32 结构，为后续的 x86 SIMD 扩展，包括 x86-SSE、x86-AVX 等奠定了基础。本章介绍 MMX 技术的基本概念、运行环境、指令系统，给出了 MMX 编程示例，以及用 C 语言开发 MMX 程序的示例。

15．1 MMX 技术简介

1、单指令多数据流的基本概念

在多媒体信息处理中，涉及到用相同方法对大量的数据进行操作。这些数据通常用向量、矩阵来存储。例如，一幅图像的信息可以用一个矩阵来存储，图像的高度和宽度对应矩阵的行数和列数，每一个矩阵元素对应图像中一个像素点。一个视频可以表示为连续的多幅图像，可用一个多维矩阵来存储。图像处理即是对矩阵中的数据进行处理。例如，将某一个固定场景中两个不同时刻的图像相减，从差值图像中可用于发现两幅图发生了什么变化，进而判断有无物品丢失或者出现闯入者。图像相减的本质是两个对应的矩阵相减，也即矩阵中各个对应元素相减，涉及到对多个数据项进行相同的操作。采用传统的对矩阵元素逐个处理的方法效率无疑是比较低的。

从字面上看， Single Instruction Multiple Data （SIMD），就是要对多个数据项同时进行相同的操作。如图 15.1，给出了一种 4 个成对数据同时进行相同 OP 操作的示意。

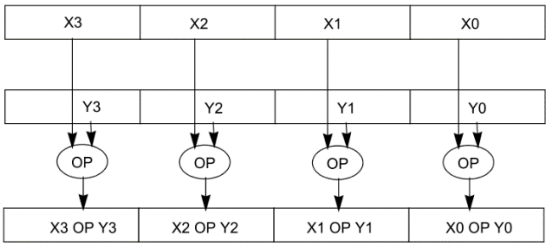

图 15.1 SIMD 运算示意

图 15.2 给出了一个具体的示例，它表示有 4 个字数据对进行加法运算。

	10 (000AH)	20 (0014H)	30 (001EH)	40 (0028H)
+	25 (0019H)	-1 (0FFFFH)	-35(0FFDDH)	35 (0023H)
	35 (0023H)	19 (0013H)	-5 (0FFFBH)	75 (004BH)

图 15.2 成组字数据加法运算

从图 15.2 可以看到各对数据是独立运算的，一对数据的运算与其他对的运算没有关系。虽然成组数据存放在 64 位寄存器中的表现与一个 64 位数据的表现形式相同，但是成组运算不同于一个 64 位数的运算。例如，若将图 15.2 中的数据视为一个 64 位的整型数，则有

如下结果，由低位向高位产生的进位会影响高位运算的结果。

000A0014001E0028H + 0019FFFFFFDD0023H = 00240013FFFB004BH

2、MMX 中的寄存器

支持 MMX 技术的处理器中有 8 个 64 位的寄存器，名为 mm0～mm7。每一个寄存器可以存放 8 个字节（byte）、或者 4 个字(word)、或者 2 个双字(double-word)数据。也就是说，可以将 8 个字节数据一个性存放到 MMX 寄存器中，形成一个打包整数（packed integers）。若两个这样的 MMX 寄存器相加，一次加法就可以实现 8 个字节数据的加法。4 字、2 个双字打包和运算是类似的。

MMX 寄存器与 CPU 中的 32 位寄存器 EAX 等一样，在指令中可以直接使用这些寄存器的名字，即采用寄存器寻址方式访问。但是这些寄存器不能用于寄存器间接寻址、变址寻址和基址加变址寻址，即不能用于寻址内存中的操作数。

MMX 寄存器的另一个用途是用于浮点运算，也即这些寄存器既可以用于整数运算，又可以用于浮点运算，它与 x87 FPU 中的数据寄存器是混叠使用的。调试程序时在寄存器窗口可以 MMX 寄存器和 FPU 的寄存器 ST 在同步变化。为了分清是在 MMX 指令中还是 FPU 指令中使用这些寄存器，在状态上需要做一些切换。

3、环绕和饱和运算

对于打包的整型数据，有环绕模式的运算（wrap around arithmetic）和饱和模式运算（saturation arithmetic）。饱和运算又分为有符号和无符号的饱和运算（signed/unsigned saturation arithmetic）。环绕运算和以前介绍过的运算是一样的，将数据看成是 0->7FH->80H->0FFH->0 的一个循环，例如对于一个字节的数据 70H 与另一个字节数据 0A0H 相加，结果为 10H。所谓饱和整数运算，是指计算的结果会被处理器自动修改，使其不会上溢或下溢。例如，对于无符号的字节数据，其上下边界为 255 和 0，若将数 70H 和 0A0H 用无符号饱和方法相加，结果为 0FFH；用无符号饱和法相减（70H-0A0H），结果为 0。用以前学过的知识，对于无符号数运算，判断是否溢出，使用标志位 CF，CF=1，则溢出，此时就对结果修正。

对于有符号的字节数据，其上下边界为 127（7FH）和-128（80H）。若将数 70H 和 0A0H 用有符号饱和方法相加，结果为 10H；用有符号饱和法相减（070H-0A0H），结果为 7FH；用有符号饱和法相减（0A0H-070H），结果为 80H。用以前学过的知识，对于有符号数运算，判断是否溢出，使用标志位 OF，OF=1，则溢出，此时就对结果修正。

对于字数据饱和运算的原则与字节运算相同，只是上下边界不同。对于无符号字数据，上下边界为 65535（0FFFFH）和 0；对于有符号字数据，上下边界为 32767（07FFFH）和-32768（8000H）。

设数据段中定义如下变量：

```
x       db    70H, 0A0H, 50H,  50H, 0F0H, 0F0H, 0F0H, 0F0H
y       db   0A0H,  70H, 30H, 0F0H,  01H,  20H,  81H, 0F0H
```

使用下列语句，可打包字节数据的加法运算：

```
movq  mm0, qword ptr x ; mm0 = F0 F0 F0 F0 50 50 A0 70
movq  mm1, qword ptr y ; mm1 = F0 81 20 01 F0 30 70 A0
paddb mm0, mm1         ; mm0 =  E0 71 10 F1 40 80 10 10
paddsb mm0,mm1         ; mm0 =  E0 80 10 F1 40 7F 10 10
paddusb mm0,mm1        ; mm0 =  FF FF FF F1 FF 80 FF FF
```

其中，paddb (ADD Packed Byte integers)为打包字节整数环绕加法指令；

paddsb（ADD Packed Signed Byte integers with signed saturation）为有符号饱和字节加法指令；

paddusb （ADD Packed UnSigned Byte integers with unsigned saturation）为无符号饱和字节加法指令。

15．2 MMX 指令简介

MMX(Multi Media eXtension,多媒体扩展)指令集,被 Pentium with MMX Technology、Celeron、Pentium Ⅱ、Pentium Ⅲ、Pentium 4、Intel Xeon、Intel Core Solo、Intel Core Duo、Intel Core2 Duo、Intel Atom 等处理器所支持。

MMX 指令集按功能可以分为八类：数据传送、算术运算、比较运算、逻辑运算、移位、转换、解组、状态控制。MMX 指令中常用字母后缀来标识需要处理的元素大小，b、w、d、q 分别对应字节（byte）、字（word）、双字（double word）、四字（quadword）。

1、数据传送

① movd opd, ops ; MOVe Doubleword

（ops）中的低位双字 ->opd。

opd、ops 中至少有一个 MMX 寄存器，ops 不能为立即数。

若 opd 为一个 MMX 寄存器，则该寄存器的左半部分为 0。

② movq opd, ops ; MOVe Quadword

（ops）四字 ->opd。其他要求同 movd。

2、算术运算

算术运算包括加、减、乘。对于打包整数各个对应元素独立运算，互不影响。根据运算方法的不同，加法运算细分为环绕加法（ADD Packed Byte/Word/Doubleword integers）、有符号饱和加法（ADD Packed Signed Byte/Word/Doubleword integers with signed saturation）和无符号饱和加法（ADD Packed Unsigned Byte/Word/Doubleword integers with unsigned saturation）。类似于加法运算，减法运算细分为环绕减法、有符号饱和减法和无符号饱和减法。乘法运算与加减法运算有所不同,两个字数据相乘,结果是一个双字。将字数据成组打包后，理论上存放结果的单元的长度要加倍，但是实际上目的操作数地址类型是不变的，它只能存放结果的一半。MMX 的乘法指令中采用的原则是保留乘积的高字或者

低字。另外，MMX 中提供了一种乘后相加的指令，先执行有符号打包整型字乘法，产生 4 个双字乘积后，对相邻双字相加得 2 个双字。

算术运算的指令如表 15.1 所示。

表 15.1 MMX 算术运算指令

指令类别	指令助记符	功能
加法运算（环绕）	paddb，paddw，paddd	字节、字、双字的环绕加法
加法运算（有符号饱和）	paddsb，paddsw	有符号打包整数饱和加法
加法运算（无符号饱和）	paddusb，paddusw	无符号打包整数饱和加法
减法运算（环绕）	psubb，psubw，psubd	字节、字、双字的环绕减法
减法运算（有符号饱和）	psubsb，psubsw	有符号打包整数饱和减法
减法运算（无符号饱和）	psubusb，psubusw	无符号打包整数饱和减法
乘法	pmullw，pmulhw	有符号打包整型的字乘法，每一对单字相乘的结果为双字，L/H 是保留双字的低/高字
乘后加法	pmaddwd	有符号打包整型字乘法，产生 4 个双字乘积，相邻双字相加得 2 个双字

注：pmullw 为 MULtiply Packed signed Word integers and store Low result

3、比较运算

① pcmpeqb/ pcmpeqw/ pcmpeqd

按字节/字/双字比较打包整型数中的各个对应元素是否相等（CoMPare Packed bytes /words/doublewords for EQual），若相等，则目的操作数中的对应元素被置为全 1（即 FF /FFFF/FFFFFFFF），否则被置为全 0。

② pcmpgtb/ pcmpgtw/ pcmpgtd

按字节/字/双字比较有符号打包整型数中的各对应元素，若目的操作数大于对应的源操作数，则目的操作数被置为全 1，否则被置为全 0。

4、逻辑运算

逻辑运算指令有 pand、por、pxor、pandn，它们分别对源操作数和目的操作数进行按位的逻辑与、逻辑或、逻辑异或、以及源操作数与求反的目的操作数按位逻辑与操作。

5、移位指令

移位指令中各个元素的处理规则与第 5 章中介绍的移位规则是相同的。移位指令包括逻辑左移（Shift Packed Data Left Logical）、逻辑左移（Shift Packed Data Right Logical）、算术右移（Shift Packed Data Right Arithmetic）。移位指令如表 15.2 所示。

表 15.2 MMX 移位指令

指令类别	指令助记符	功能
逻辑左移	PSLLW，PSLLD，PSLLQ	每个字、双字、四字分别逻辑左移
逻辑右移	PSRLW，PSRLD，PSRLQ	每个字、双字、四字分别逻辑右移

算术右移	PSRAW，PSRAD	每个字、双字分别算术右移

6、转换 （Conversion Instructions）

① packsswb （PACK Words into Bytes with Signed Saturation）

使用有符号饱和运算分别将源操作数和目的操作数中各个字整数分别压缩成有符号字节整数，压缩结果存放在目的寄存器中，源操作数的压缩结果放在高位。如果字的有符号值超出有符号字节的范围（即大于 7FH 或小于 80H），压缩后为 7FH 或 80H。

例如：设 mm0 = 0001 0234 5678 9ABC H

mm1 = 0045 6789 8ABC 0067 H

执行 packsswb mm0，mm1 后， mm0 = 45 7F 80 67 01 7F 7F 80H。

② packssdw （PACK Doublewords into Words with Signed Saturation）

将有符号双字数据压缩为字数据，压缩时使用饱和运算，方法类似 packsswb。

③ packuswb （PACK Words into Bytes with Unsigned Saturation）

使用无符号饱和运算将字数据压缩为字节数据。

7、解组（Unpack Instructions）

① punpckhbw

按字节取源操作数的高位字节（前 4 个字节）和目的操作数的高位（前 4 个字节）拼成 4 个字数据，放在目的操作数中。

例如，设 mm0 = 0001 0234 5678 9ABC H ，mm1 = 0045 6789 8ABC 0067 H

执行 punpckhbw mm0，mm1 后，mm0= 0000 4501 6702 8934H。

② punpckhwd (UNPack High-order Words)：数据的高位按字解组，拼接为双字

③ punpckhdq ：数据的高位按双字解组，拼接为四字

④ punpcklbw、punpcklwd、punpckldq：用数据的低位部分完成解组和拼接。

8、状态控制（EMMS Instruction）

emms 指令用来清除 MMX 的状态信息，换句话说，它通过设置 x87 FPU 中的标记寄存器 TAGS 为 FFFF，来表明所有 MMX 寄存器都未使用。从 MMX 指令转换为 FPU 指令之前，都应执行指令 emms。

15．3 MMX 编程示例

采用 MMX 指令编写程序与之前介绍的用 CPU 指令编写程序的方法差别不大，最主要的将数据打包运算，以减少相同指令执行的次数，提高程序运行效率。

【例 15.1】求两个数组之和，结果存入第三个数组中。

为了验证 MMX 指令能够加快运算速度，采用了在 C 语言程序中嵌入汇编语言的编写方法。比较用一般的 C 语句、非 MMX 指令、MMX 指令三种写法的执行效率。

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
```

```
#include <conio.h>
#define LEN 100000 // 数组大小
int main() {
	clock_t stTime, edTime;
	int	i,j;
	unsigned short	a[LEN];
	unsigned short	b[LEN];
	unsigned short	c[LEN];
	srand(time(NULL));
	for (i = 0; i < LEN; i++) {	// 生成随机数组
		a[i] = rand() ;
		b[i] = rand();
	}
	stTime = clock();
/*	// 为了比较性能，将运算 c = a + b 重复执行了 1000 次
	// 执行此处的语句，即可得到用传统方法写出的程序的效率
	 for (j = 0; j < 1000; j++) {
		for (i = 0; i < LEN; i++) {
			c[i] = a[i] + b[i];
		}
	}
*/
	__asm{
			mov	ecx,1000
		l1:
			lea	edi, a
			lea	esi, b
			lea	ebx, c
			mov	edx , LEN/4
		l2:
			movq	mm0, qword ptr [edi]
			movq	mm1, qword ptr [esi]
			paddw	mm0,mm1
			movq	qword ptr[ebx],mm0
			add	edi,8
			add	esi,8
			add	ebx,8
			dec	edx
			jnz	l2
			dec	ecx
			jnz	l1
			emms
	}
	edTime = clock();
```

```
        unsigned int spendtime = edTime - stTime;
        printf("time used: %d \n", spendtime);
        // 下面只是为了验证计算是否正确
        for (i = 0; i < 20; i++)
            printf("%d + %d = %d \n", a[i], b[i], c[i]);
        _getch();
        return 0;
    }
```

在本人的机器上，上述程序运行时间约为 70 毫秒。

如果运行注释中的 C 语言程序段，而不使用内嵌的汇编代码，用时约为 800 毫秒。

将内嵌的汇编段改为非 MMX 指令的实现方式，用时约为 200 毫秒。修改后的内嵌汇编代码段如下：

```
__asm{
    mov   ecx, 1000
    l1:
        lea   edi, a
        lea   esi, b
        lea   ebx, c
        mov   edx, LEN
    l2 :
        mov   ax, word ptr[edi]
        add   ax, word ptr[esi]
        mov   word ptr[ebx], AX
        add   edi, 2
        add   esi, 2
        add   ebx, 2
        dec   edx
        jnz   l2
        dec   ecx
        jnz   l1
}
```

从上面的例子可以看到，采用 MMX 指令后，可以显著提高程序的执行速度。

【例 15.2】 实现两个向量的内积

设有向量 a=(a1,a2,a3,a4)，向量 b=(b1,b2,b3,b4)

向量 a、b 的内积为 <a,b>=a1*b1+a2*b2+a3*b3+a4*b4。

```
.686P
.MMX
.model flat, c
  ExitProcess proto stdcall :DWORD
  printf      proto  :vararg
  includelib  libcmt.lib
  includelib  legacy_stdio_definitions.lib
.data
  buf1        sword   1, -2, 3, 400H
```

```
  buf2        sword   2, 3, 4, 500H
  buf3        sdword   0, 0
  lpFmt       db  "%d  %x(H)", 0dh,0ah,0
.stack 200
.code
main  proc
   movq     mm0,  qword ptr  buf1     ; mm0=04000003FFFE0001H
   movq     mm1,  qword ptr  buf2     ; mm1=0500000400030002H
   pmaddwd  mm0,  mm1                 ; mm0=0014000CFFFFFFFCH
   movq     qword ptr buf3, mm0
   mov      eax,  buf3
   add      eax,  buf3+4
   emms
   invoke printf, offset lpFmt, eax, eax
   invoke Exitprocess, 0
main  endp
end
```

执行该程序后，显示结果为 1310728 140008(H)。

注意：0400H*0500H = 00140000H，-2*3 = -6 = 0FFFFFFFAH

在编写程序的时候，注意处理器的选择伪指令要含有“.MMX”。

15．4 用 C 语言编写 MMX 应用程序

在了解用 MMX 指令可以提高大数据（矩阵、向量）运算速度后，除了用汇编语言编写 MMX 应用程序外，好奇的读者会问是否可以用 C 语言编写类似的程序以提高计算速度。答案是肯定的。

对于例 15.1 中的程序，只需做如下改动。

增加头文件 #include <mmintrin.h>

增加几个局部变量：

```
    __m64  *pa;  // 指向数组 a
    __m64  *pb;  // 指向数组 b
    __m64  *pc;  // 指向数组 b
    int  LEN4;    // 由于打包运算，一次运算 4 个数，总循环次数要减少。
```

对于原来的双种循环语句，该写成如下形式：

```
    for (j = 0; j < 1000; j++) {
        pa = (__m64 *)a;
        pb = (__m64 *)b;
        pc = (__m64 *)c;
        LEN4 = LEN / 4;
        for (i = 0; i < LEN4; i++) {
            *pc = _m_paddw(*pa, *pb);
            pa += 1;   // 反汇编后，地址是加 8
            pb += 1;
            pc += 1;
```

```
        }
    }
    _m_empty();  // 实际是 EMMS 指令
```

在本人的机器上，上段程序的执行时间约为 220 毫秒。

有兴趣的读者，可以打开 mmintrin.h。在该头文件中定义了 UNION 联合体__m64。该 64 位二进制信息可以看成是 8 个字节数据、4 个字数据、2 个双字数据或者 1 个 64 位数据。在头文件在还有很多函数，基本上就是在 MMX 指令上的封装，采用反汇编手段，可以看到其实现方法。

习题 15

15.1 MMX 中有哪 8 个 64 位的寄存器？

15.2 什么是环绕运算、有符号饱和运算、无符号饱和运算？

15.3 设有变量

```
x       db    70H, 0A0H, 50H,   50H, 0F0H, 0F0H, 0F0H, 0F0H
y       db   0A0H,   70H, 30H, 0F0H,   01H,   20H,   81H, 0F0H
z       db   8 DUP(0)
```

编写程序，完成无符号字节饱和加法 z=x+y，x、y 和 z 都看成为向量。要求分别用下面两种方式实现：

① 使用 MMX 指令；

② 使用 CPU 指令（第 5 章中介绍的），不准使用 MMX 指令。

15.4 设有变量

```
buf1    sword   1, -2, 3, 400H
buf2    sword   2, 3, 4, 500H
result  sdword  0, 0, 0, 0
```

编写程序，用 MMX 指令实现向量 buf1 和 buf2 的点乘，即各个对应元素相乘，结果存放在 RESULT 中。RESULT 中的结果为 00000002H，0FFFFFFFAH，0000000CH，00140000H。

提示：用到的指令有 movq、PMULLW，PMULHW、PUNPCKHWD、PUNPCKLWD、EMMS 等

上机实践 15

15.1 用 C 语言编写程序，实现两个矩阵的乘法。对矩阵相乘的部分进行计时。

假设矩阵中元素类型为 short，矩阵行列数都是 4 的倍数。

(1) 用传统方法（即逐个元素运算）实现

(2) 用 MMX 打包运算方法（参见 mmintrin.h 中的函数）

15.2 功能与上机实践 15.1 相同，即实现矩阵的乘法，但矩阵中的行、列数为任意正整数。