目录

第 14 章 x87 FPU 程序设计

在解决实际问题时，常常会涉及到浮点数的处理。x87 浮点运算单元（Floating-Point Unit，FPU）是 80387、80487 的统称，它们是 Intel 公司为 80386SX、80486SX 设计的浮点数处理的协处理器。浮点数的存储表示与第三章中介绍的整数表示有所不同，它们的运算也有专门的指令，不同于前面介绍的通用机器指令。本章介绍浮点数的表示方法以及 x87 FPU 的部分浮点数运算指令。在学习浮点数表示和处理知识的过程中，可以编写相应的 C 语言程序，借助调试和观察反汇编代码等手段更好地掌握这些知识。

14．1 浮点数据

14.1.1 浮点数据在机内的表示形式

对于一个十进制数，不论是整数还是实数，都可以表示成如下形式：

$$\pm(a_n10^n + \cdots + a_110^1 + a_010^0 + a_{-1}10^{-1} + \cdots + a_{-m}10^{-m} + \cdots) = \pm\sum_{i=-m}^{n} a_i10^i$$

其中a_i (i=-m～n) 是 0 至 9 中的一个数码。

类似这种写法，任意一个十进制数也可以写成如下二进制的形式：

$$\pm(a_n2^n + a_{n-1}2^{n-1} + \cdots + a_12^1 + a_02^0 + a_{-1}2^{-1} + \cdots + a_{-m}2^{-m} + \cdots)$$

采用二进制表示法，上数可记为：$\pm\ a_na_{n-1}\cdots a_1a_0.a_{-1}\cdots a_{-m}\cdots$B

其中a_i (i=-m～n) 是 0、1 之一。

例如：123.45 = 1111011.01110011001100.....B

将十进制数转换为二进制数时，对于整数部分可以采用除 2 取余法，而对于小数部分用乘 2 取整法得到。

进一步规范化数的表示形式，使得整数部分为 1，可表示为：

123.45 = 1.11101101110011001100.....$*2^6$ B

同理：

0.012345 = 0.00000011001010010000101010111 1...B

= 1.10010100100001010101111...$*2^{-7}$ B

上面的表示法为科学表示法。对于一般的二进制数而言，小数点左边的一个二进制位恒为 1。是否存储该位，在 1985 年之前，各 CPU 厂商都有自己的浮点数表示规则，并不统一，使得不同计算机之间的程序移植困难。在 1985 年制定 IEEE 754 标准后，各 CPU 厂商都遵循了同一个标准，即不存储小数点左边的 1。

除了一般的规格化数据 ±1.XXXX*2^nB 外，还有一些数据是不能表示成这种形式的，称为非规格化数据（denormalized number，或 subnormal number）。例如，数值 0、数值-0，在小数点前无法出现整数 1，它的有效数字全部为 0。非常接近于 0 的数也有类似问题，如 $1.01 * 2^{-130}B$。表面上看，它写成了规范化形式，但是指数部分超出了（单精度浮点）表示

范围，若将指数调整到表示范围内，该数则要写成$0.000101 * 2^{-126}B$。

除规格化数据、非规格化数据外，还有一些特殊数值，如正无穷+∞、负无穷-∞、非数值（Not a Number，NaN）。非数值表示运算结果不是实数或者无穷，例如对-1 开平方，其结果就是非数值；非数值也可以用于表示未初始化的数据。

下面具体介绍 IEEE 754 中浮点数据的表示方法。它用三个组成部分——符号位、指数部分、有效数字部分来表示一个数。从十进制数 123.45 转换为二进制数的结果来看，在小数点后的位数是无穷的，计算机中只能存储有限的位数，因而只能用不同精度的数来近似。浮点数分为单精度符点数、双精度浮点数和高（扩展）精度浮点数，它们的组成形式类似，只是各部分的长度不同。

单精度浮点数（4 个字节）的组成：

31	30 23	22 0
符号	8 位指数	23 位尾数

双精度浮点数（8 个字节）的组成：

63	62 52	51 0
符号	11 位指数	52 位尾数

高精度浮点数（10 个字节）的组成：

79	78 64	63 0
符号	15 位指数	63 位尾数

符号位：浮点数据的最高二进制位，其值为 0 表示正数，1 表示负数。

指数（阶码）部分：以 2 为底数的指数。对于一个浮点数而言，指数可以为正也可以为负。为了表示正负，采用“移码”来表示，这与补码表示不同。对于单精度浮点数，指数部分占 8 个二进制位，偏移基数为 7FH（即 127）。对于某一个指数 A，它的编码为：127+A。当然，看到一个编码后，也很容易求出实际值是为“编码-127”。例如对于指数为 6，其编码为：10000101B；对于-7 的编码为 01111000B。

注意，指数编码中所有二进制位全 0 和全 1 两个编码有特殊含义。因此编码的范围为 00000001~11111110B， 对应的实际的指数值是从-126～+127。由此也可以猜测指数表示中用移码而不用补码的原因，主要是空出了两种特殊的编码。

双精度浮点数的指数部分占 11 个二进制位，其偏移基数为 3FFH。高精度浮点数的偏移基数为 3FFFH，其他规则等同单精度浮点数。

有效数字部分：对于规格化的数据 ±1.fff...ff*2nB ，只存储小数点后的部分。

非规格化数据：当指数和有效数字全为零时，其值为零（包括数值相等的+0 和-0）。

特殊值：指数全为 1，有效数字全为 0，表示+∞或-∞（有符号无穷大）；指数全 1 但有效数字不全为 0 表示非数 NaN（不是实数的一部分）。

【例 14.1】 将单精度浮点数 3FA00000H 转换成普通十进制实数的形式。

解：首先写成二进制形式：3FA00000H =0011 1111 1010 0000 0000 0000 0000 0000 B

将它分成符号、指数和有效数字三部分：

3FA00000H = 0 0111 1111 0100 0000 0000 0000 0000 000 B

符号位为 0，表明它是正数；

指数部分为 0111 1111，表示指数 = 127-127 = 0；

有效数字部分为 0100 0000 0000 0000 0000 000，表示的有效数=1.01=1+2^{-2}= 1.25；

这个浮点格式的单精度数转换成实数的最终结果为 1.25。

在数据存储时，仍遵循数据的最高字节放在地址最大的字节、数据的最低字节放在地址最小的字节中的原则（即小端存储的原则，Little Endian）。在内存中看到的存放结果为：00 00 A0 3F。

14.1.2 浮点类型变量的定义

在汇编语言程序设计中，可以用以下方式定义浮点数类型的变量。

```
f1    real4    1.25     ; 对应 C 语言的 float，单精度浮点数，4 个字节
f2    real8    1.25     ; 对应 C 语言的 double，双精度浮点数，8 个字节
f3    real10   1.25     ; 对应 C 语言的 long double，高精度浮点数，10 个字节
i1    dd       1.25     ; 4 个字节的变量
i2    dq       1.25     ; 8 个字节的变量
i3    dt       1.25     ; 10 个字节的变量
```

注意，变量 f1 和 i1 中的存储结果相同；变量 f2 和 i2 中的存储结果相同；变量 f3 和 i3 中的存储结果相同。但 f1、f2、f3 中的存储结果是不同的，它们分别对应单精度、双精度、高精度的浮点数。

与前面介绍的地址类型转换一样，给定一个地址，可以将该地址作为不同类型的变量的起始地址，即从指定的地址开始，将连续的字节当成指定的类型来解释。下面，给出了一个 C 语言程序的调试示例，可观察浮点数的存储形式，以及数据类型转换的结果。

```
float fx = 1.25;
int   iy,  iz;
iy = *(int *)&fx;
iz = (int) fx;
printf("%f %d %d\n", fx, iy,iz);
```

运行该程序，显示了三个数：1.250000　1067450368　1

其中 1067450368 对应的 16 进制为 3FA00000H。

在调试程序时，可以在监视窗口观察变量 fx、iy、iz 的地址（&fx,&iy,&iz），从内存窗口可以看到它们对应的内存单元的存储结果。

语句 iy＝*(int *)&fx; 是进行地址类型转换，即将一个浮点数的地址当成一个整型数的地址来看，保持内容不变。地址类型转换后，fx 和 iy 中存储的结果是一样的，皆为 00 00 A0　3F。语句 iz =(int)fx; 是进行数据类型转换，将一个浮点数转换为一个整型数，数的大小保持近似不变（有舍入误差）。

在汇编语言程序调试中，观察浮点数类型的变量的存储结果与 C 语言变量的储存结果是相同的。

14.2 x87 FPU 的寄存器

在早期的 CPU（8088/8086、80286、80386SX）中，需要浮点运算时，CPU 通过软件模拟来实现，浮点数运算很慢。为此，Intel 公司为 80386SX 设计了浮点运算协处理器 80387，为 80486SX 设计了协处理器 80487。它们统称为 x87 FPU(Floating-point Processing Unit)。浮点部件具有浮点数值运算的功能并提供相应指令系统，完成三角函数、指数函数、对数函数等运算，精度高、流量大、速度快。与此同时，Intel 公司也开发了集成协处理器的 CPU：80366DX、80486DX。到了奔腾时代，Intel 将协处理器全部集成在 CPU 内部，不再有带/不带协处理器 CPU 之说。不同时期的 CPU 中浮点运算部件中并不相同，下面简单介绍 x87 FPU 中的寄存器和 SSE 中的寄存器。

14.2.1 x87 FPU 数据寄存器

在 x87 FPU 中有 8 个 80 位的寄存器，用于存放扩展（高）精度的浮点数据。这些寄存器的名称为 r0、r1、……、r7。它们的用法与通用寄存器 eax、ebx 等的用法有很大的差别。eax 等寄存器是随机存取的，只要在指令中给出寄存器的名字，就可以访问到对应的寄存器，在机器指令中有对应寄存器的编码。FPU 中的 8 个数据寄存器组织成一个堆栈，按照后进先出的堆栈原则工作，不能在指令中直接使用这些寄存器的名字。装载一个数到 FPU 数据寄存器时，相当于执行一次入栈操作。当然在数据入栈时，首先将其转换为高精度浮点数据格式，然后将其入栈。当前栈顶的寄存器编号由 FPU 中的状态寄存器中的 TOP 字段指明。装载一个数到 FPU 的数据寄存器时（指令为 fld、fild、fbld），TOP 值减 1 且按 8 取模后，数据随后放在 R_{TOP} 指示的寄存器中。

设程序开始运行时，TOP=0。装载第一个数据到 FPU 的寄存器中时，TOP 减 1 模 8 后为 7，第一个数即放在 r7 中；若紧接着又装载第二个数据到 FPU 的寄存器中时，第二个数放在 r6 中，TOP=6。依次类推。当然，连续 8 次装载数据后，栈已装满，即所有的数据寄存器都被使用了，若此时再装载数据，则会出现栈溢出的错误，在 FPU 的状态寄存器中会有标志位来反映这一问题。从堆栈中弹出数据时（指令 fstp、fistp、fbstp），先将 R_{TOP} 指示的寄存器中的数据送到目的单元，然后 TOP=(TOP+1)模 8，同样，栈为空时，再弹出数据也会出现错误。

在使用 VS2019 等调试程序时，在寄存器窗口并不能直接看到 r0~r7，而是看到助记符 st0、st1、……、st7，但 st0~st7 并不是固定的与 r0~r7 对应，而是浮动的记号。助记符 st0 始终是指向栈顶的数据寄存器。在任意时刻，寄存器 R_{TOP} 都对应 st0，在该栈顶之下的寄存器由 st1 指示，对应的寄存器为 $R_{(TOP+1) \bmod 8}$，依次类推。在汇编语言程序中，可以的浮点数据寄存器符号为 st(i)，i=0~7，对于 st(0)可以写成 st。

假设程序一开始的指令是 “fld f1”和“fld f2”，TOP 的初始值为 0，执行后的结果

如图 14.1 所示。

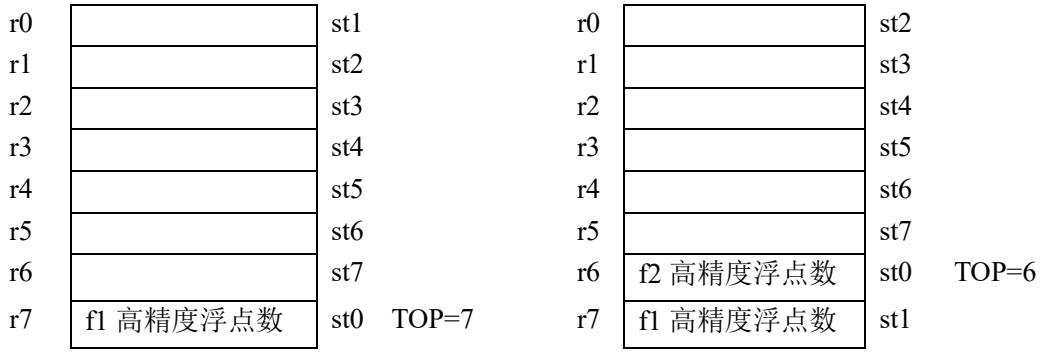

图 14.1 执行 2 条指令后 FPU 数据寄存器数据存放的示意图

14.2.2 x87 FPU 其他寄存器

FPU 中除了数据寄存器外，还有标记寄存器、状态寄存器、控制寄存器。

1、标记寄存器

为了表明每个浮点数据寄存器中数据的性质，用一个 16 位的寄存器来存放标记结果，称为标记寄存器（TAGS）。该寄存器被分为 8 个部分，每部分含 2 个二进制位，其中最高的 2 位（即第 15 位和第 14 位）是 r7 的标记结果；最低的 2 位是 r0 的标记结果，即由高到低分别对应 r7~r0 的标记结果。标记寄存器的结构如图 14.2 所示。

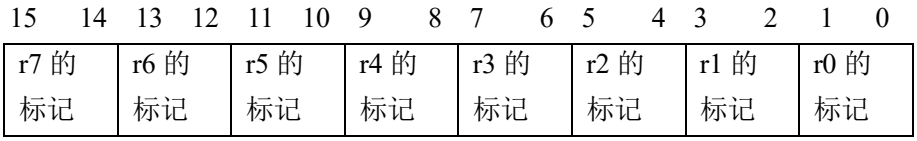

图 14.2 FPU 内的标记寄存器

各种标记值的含义如下：

00：对应的数据寄存器中存有有效的数据；

01：对应的数据寄存器中的数据为 0；

10：对应的数据寄存器中的数据是特殊数据（非数 NaN、无穷大或非规格化数）；

11：对应的数据寄存器内没有数据（空状态）。

在使用 VS2019 等调试程序时，在寄存器窗口可以看到 TAGS，其初值为 FFFF。执行“fld f1”后，TAGS 变为 3FFF；再执行“fld f2”，TAGS 变为 0FFF。

2. 状态寄存器

状态寄存器（STAT）表明 FPU 当前的各种操作状态以及每条浮点指令执行后所得结果的特征，其作用与 CPU 中的标志寄存器相当，各个标志位如图 14.3 所示。

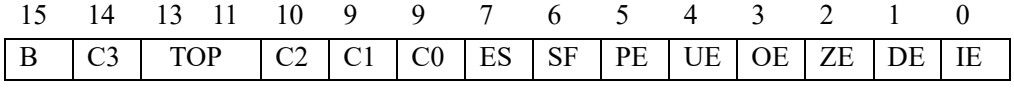

图 14.3 FPU 内的状态寄存器

状态寄存器含有 FPU 忙闲标志(Busy)、栈顶寄存器指针(TOP)、条件码标志位（Condition Code，C3~C0）、异常综合标志（Exception Summary，ES）、堆栈错误标志(Stack Fault，SF),

异常标志（Exception flags，PE、UE、OE、ZE、DE、IE）。

B=1： FPU 忙

TOP：指明了当前栈顶的浮点数据寄存器编号（0~7）

C0~C3：保存浮点运算结果的标志，与 CPU 中标志寄存器 EFLAGS 中的条件标志位 SF、ZF、OF、CF 等的作用类似

ES=1，至少存在一种错误，即第 5~0 位中至少有一位置 1；

SF=1：上溢（栈满时再入栈，SF=1，C1=1）；下溢（栈空再出栈，SF=1，C1=0）；

PE=1：精度异常，运算结果不能用二进制精确表示成目的操作数的格式；

UE=1：向下溢出异常，结果太小，小于目标操作数允许的最小值；

OE=1：向上溢出异常，结果太大，超过了目标操作数允许的最大值；

ZE=1：除 0 异常，除数为 0；

DE=1：不合规格操作数异常，至少有一个操作数是非规格化的；

IE=1： 无效操作异常，非法操作，如对负数开平方等。

在使用 VS2019 等调试程序时，在寄存器窗口可以看到状态寄存器 STAT。状态寄存器可以通过“fstsw ax”等指令传送到 ax 中，也可以通过“fstenv opd”等指令转存到主存中的变量缓冲区中，根据状态寄存器的值可做后续处理。置位后的错误标志必须用指令“fclex、fnclex、finit、fninit”清除，否则将保持不变。当 FPU 出现错误时，可利用控制寄存器中的相应屏蔽位对这六种错误进行屏蔽。

3. 控制寄存器

控制寄存器（CTRL）用于控制 FPU 的异常屏蔽、精度、舍入操作，各个控制位如图 14.4 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			IC	RC		PC				PM	UM	OM	ZM	DM	IM

图 14.4 FPU 的控制寄存器

控制寄存器中的最低 6 位分别与状态寄存器的最低 6 位对应，决定相应的错误是否被屏蔽。在状态寄存器中的某一错误标志位 1 的情况下，若控制寄存器中对应的异常屏蔽位为 1，则该异常被屏蔽，FPU 自动产生一个事先定义的结果，程序继续执行；若这个异常没有被屏蔽，则 FPU 将调用异常处理程序。

PC：用于控制浮点计算结果的精度，占 2 个二进制位。00 表示单精度，01 保留，10 表示双精度，11 表示扩展精度。

RC：舍入控制位，也占 2 个二进制位。00 表示四舍五入； 01 表示向下舍入，即取地板，得到小于等于源操作数的最大整数； 10 表示向上舍入，即取天花板，得到大于等于源操作数的最小整数； 11 表示截断舍入，舍入结果接近但绝对值不大于源操作数。注意，对于一个正数，向下舍入与截断舍入的结果相同，但对于负数，则不相同。

IC：无穷大控制，对 80387 以后的 FPU，该位必须置 1。

4、用于异常处理的三个寄存器

x87 FPU 中还有三个特殊的寄存器，“最后一条 x87 FPU 指令指针寄存器”、“最后一条数据指针寄存器”、“最后一条 x87 FPU 指令操作码寄存器”。在 VS2019 反汇编调试中，看到的最后一条指令指针寄存器为 EIP，它与前面介绍的 CPU 中的指令指针 EIP 不同。CPU 中的指令指针指向下一条要执行的指令的地址；而 FPU 中的 EIP 指示的是当前时刻之前、最后执行的一条 FPU 指令的地址。最后一条数据指针寄存器指示的是当前时刻之前、浮点运算指令中最后使用到内存单元的地址，在 VS209 反汇编调试中对应的是 EDO。最后一条 x87 FPU 指令操作码寄存器只有 11 位，所有的 FPU 指令的最高 5 位为 11011B,它不保留这 5 位，只保留后 11 位。这三个寄存器是由操作系统和异常处理程序使用，目的是让异常处理程序获得触发异常的指令的一些信息。注意，在异常产生后，状态寄存器上设置了异常标志，若控制寄存器上异常非屏蔽，则会自动调用异常处理程序。

14.3 x87 FPU 指令

x87 FPU（Float Point Unit，浮点运算单元）指令集，被 Intel 486、 Pentium、Pentium with MMX Technology、 Celeron、 Pentium II、Pentium III、Pentium 4、Intel Xeon、Intel Core Solo、Intel Core Duo、Intel Core 2 Duo、Intel Atom 等处理器所支持。

x87 FPU 具有自己的指令系统，它们的指令助记符全部以 f 开头，进一步地，以 fi 开头的指令表示整数操作；以 fb 开头指示 BCD 码数操作。浮点指令可分成传送、算术运算、超越函数、比较、常数、FPU 控制等类。

浮点指令一般需要 1 或 2 个操作数，操作数可以存放在 FPU 内的浮点数据寄存器中，也可以存放在主存中，但不能是立即数。当操作数存放在主存中时，可以采用在第 4 章中介绍的各种存储器寻址方式。在主存中的操作数的类型不必与浮点数据寄存器的类型一致，但必须类型明确。许多浮点指令将 st(0)作为隐含的目的操作数。

1. 数据传输的指令

fld ops ：将 ops 中的浮点数压入数据寄存器栈顶 st(0)，ops 为内存单元或 st(i)

fild ops ：将内存中的整数转换为高精度浮点数压入栈顶 st(0)

fbld ops ：将内存中的 bcd 码数压入栈顶 st(0)

fst opd ：将栈顶 st(0)的数据存放到 opd 中，opd 为内存单元或 st(i)
存放到内存单元中时，要进行浮点数据格式转换，栈顶不变

fist opd ：将栈顶 st(0)的数据按照整数格式存入到内存单元中，栈顶不变

fstp opd ：将栈顶 st(0)的数据按浮点格式存放到内存单元中，然后出栈

fistp opd ：将栈顶 st(0)的数据按照整数格式存入内存，然后出栈

fbstp opd ：将栈顶 st(0)的数据按 bcd 码数格式存入内存，然后出栈

fxch st(i)：st(0)与 st(i)的内容交换

fcmov**cc** st(0),st(i)：在条件 cc 成立时，(st(i))→st(0)，cc 指某种条件，例如：
fcmovb ：below ， cf=1； 与之相反的是 fcmovnb

fcmove ： equal, zf=1; 与之相反的是 fcmovne

类似的还有 fcmovbe (cf=1 或 zf=1), fcmovnbe , fcmovu (pf=1), fcmovnu (pf=0)。

注意：fstp 比 fst、fistp 比 fist 多一个弹出栈顶元素的操作，p 是 pop 的缩写。fld 与 fild 差别很大，前者是将(ops)当成一个浮点数看，后者将(ops)当成一个整数看。

2. 基本算术指令

fadd ops ：(st(0))+(ops)→st(0)，ops 为内存中的单/双精度数的地址

fadd st(i),st(j) ：(st(i))+ (st(j))→st(i)，i,j 中至少一个为 0

faddp st(i),st(0) ：(st(i))+ (st(j))→st(i),然后弹出 st(0)，i 不能为 0

fiadd ops ：(st(0))+(ops)→st(0)，ops 为内存中的字/双字整数的地址

fsub ops ：(st(0))-(ops)→st(0)，ops 为内存中的单/双精度数的地址

fsub st(i),st(j) ：(st(i))- (st(j))→st(i)，i,j 中至少一个为 0

fsubp st(i),st(0) ：(st(i))- (st(0))→st(i),然后弹出 st(0)，i 不能为 0

fsubr ops ：((ops)-st(0))-→st(0)，ops 为内存中的单/双精度数的地址

fsubr st(i),st(j) ：(st(j))- (st(i))→st(i)，i,j 中至少一个为 0

fsubrp st(i),st(0) ：(st(0))- (st(i))→st(i)，然后弹出 st(0)，i 不能为 0

fisub ops ：(st(0))-(ops)→st(0)，ops 为内存中的字/双字整数的地址

fmul ops ：(st(0))*(ops)→st(0)，ops 为内存中的单/双精度数的地址

fmul st(i),st(j) ：(st(i))*(st(j))→st(i)，i,j 中至少一个为 0

fmulp st(i),st(0) ：(st(i))*(st(0))→st(i),然后弹出 st(0)，i 不能为 0

fimul ops ：(st(0))*(ops)→st(0)，ops 为内存中的字/双字整数的地址

fdiv ops ：(st(0))/(ops)→st(0)，ops 为内存中的单/双精度数的地址

fdiv st(i),st(j) ：(st(i))/(st(j))→st(i)，i,j 中至少一个为 0

fdivp st(i),st(0) ：(st(i))/(st(0))→st(i),然后弹出 st(0)，i 不能为 0

fdivr st(i),st(j) ：(st(j))/(st(i))→st(i)，i,j 中至少一个为 0

fdivrp st(i),st(0) ：(st(0))/(st(i))→st(i),然后弹出 st(0)，i 不能为 0

fidiv ops ：(st(0))/(ops)→st(0)，ops 为内存中的字/双字整数的地址

fchs ： -(st(0)) →st(0)

fabs ： |(st(0))| →st(0)

fsqrt ：计算(st(0))的平方根→st(0)

fscale ： (st(0))*$2^{(st(1))}$->st(0)

frndint ： (st(0)) 舍入->st(0)，舍入方式由控制寄存器的 RC 决定。

fprem ：(st(0))%(st(1)) ->st(0) 求余数

从上面列出的部分指令来看，算术运算指令较多，但也有一些规律。对于加、减、乘、除浮点运算有以下形式：

① fadd/fsub/fmul/fdiv memreal，即 st(0)与一个内存单元浮点数据运算，结果存

放在 st(0)中。

② fadd/fsub/fmul/fdiv st(i),st(j)，要求两个寄存器中至少有一个是 st(0)

③ fsubr/fdivr st(i),st(j)，指令名带后缀 r，表示 reverse，即源操作数与目的操作数交换位置。同于加法、乘法，两个运算数交换位置后的结果是一样的，故不带 r。对于减法、除法，交换被减(被除)数与减(除)数的位置，结果是不同的。

④ faddp/fsubp/fmulp/fdivp st(i),st(0)，指令名带后缀 p，表示 pop，即有数要出栈，而栈顶为 st(0)。显然，st(0)不应做目的操作数地址，因为若做目的地址，操作结果存放在 st(0)中，然后又废弃该结果，运算就无实际意义了，因而 st(0)只做源操作数。

⑤ fsubrp/fdivrp st(i),st(0)，指令名带后缀 r 和 p，是 reverse 和 pop 的组合。

除此之外，还有 fiadd/fisub/fimul/fidiv meminteger，即 st(0)与一个内存单元中整数（字或者双字）运算。

3. 比较指令

fcom ops ：(st(0))为目的操作数，源操作数可以是内存单元或是其它的堆栈寄存器 st(i),根据比较结果设置状态寄存器中的条件标志位，设置规则表 14.1 所示。

fcomp ops ：在 fcom 功能的基础上，增加比较后弹出栈顶功能

fcompp ：st(0)与 st(1)比较，之后弹出数据寄存器栈中的两个数

表 14.1 浮点数比较的条件位设置方法

条件	C3	C2	C0
(ST(0))>源操作数	0	0	0
(ST(0))<源操作数	0	0	1
(ST(0))=源操作数	1	0	0
不可比较	1	1	1

类似的指令还有：fucom/fucomp/fucompp、ficom/ficomp、fcomi/fcomip、fucomi/fucomip、ftst。

在比较指令后，一般会有转移指令，根据比较结果执行不同的分支。fpu 的指令集中没有转移指令,因而要产生 fpu 条件跳转必须用间接方法。即用指令“fstsw ax”、“fstenv opd”等指令将状态寄存器的内容存入 ax 或一个内存单元里。之后，可以用指令 sahf 将标志信息存入 CPU 的标志寄存器里，最后使用 CPU 指令 jl/jg/je/ja/jb 来跳转至正确处执行。fcomi/fcomip 等指令直接设置了 CPU 中的 EFLAGS。

4. 超越函数指令

fsin ： 正弦函数， sin(st(0))→st(0) ，(st(0))为弧度

fcos ： 余弦函数， cos(st(0))→st(0) ，(st(0))为弧度

fptan ：正切函数，计算结果是以分数 y/x 的形式表示，其中 x=1。先将 y 压入堆栈，再把 1 压入堆栈，即 st(1)=tan(st(0))，st(0)=1

fpatan ：反正切函数，计算 arctan (st(1)/st(0)) →st(0)。先弹出一个数做分母，再弹出一个数做分子，计算后结果入栈。

5. 控制和常数指令

finit ：初始化浮点单元，每次开始浮点运算和运算完毕后应该执行该指令

在初始化之前，检查错误条件

fldcw mem16 ： 将字存储单元(mem16)中的内容存→控制寄存器

fstcw mem16 ：将(控制寄存器)→字存储单元(mem16)

fstsw ax / mem16 ：在检查错误条件后，将(状态寄存器)→ax 或字存储单元

fnstsw ax / mem16 ：不检查错误条件，将(状态寄存器)→ax 或字存储单元

fclex ：在检查错误条件后，清除状态寄存器中的异常标志

fnclex ：不检查错误条件，清除状态寄存器中的异常标志

fstenv/fnstenv ：将 FPU 状态寄存器、控制寄存器、标记寄存器、FPU 中三个特殊的寄存器中的内容都保存到内存中。

fsave/fnsave ：除了完成 fstenv 的功能外，还保存 FPU 的数据寄存器的内容到内存中。保存后初始化浮点单元，如同 finit/fninit。

fldpi ：将常数π送到 FPU 堆栈上

fld1 ：将常数 1.0 送到 FPU 堆栈上

fldz ：将 0.0 压入堆栈顶

fwait/wait ：使处理器处于等待状态，强制协处理器检查待处理且未屏蔽的异常，直到异常处理结束，才继续执行 fwait 之后的指令。(汇编程序可能在程序中自动插入该指令)

fnop ：空操作，等同于 CPU 的 nop 指令

注意，在编写汇编源程序时，应先通过“.387”、“.487”等伪指令说明使用哪种协处理器的指令集，在 80486 以后的机型只需说明 CPU 类型即可，如“.586”、“.686”等。

14.4 浮点数处理程序示例

随着 CPU 发展，浮点数运算的指令和相关的寄存器变化较大，增加的内容多，浮点数处理指令难于记忆。我们认为并不需要读者去记忆这些内容，只是学会一种方法，怎样去理解、应用相关的指令。要学习浮点数处理指令，可以先编写一个 C 语言程序，然后反汇编，看看有哪些浮点数处理指令。通过查阅相关的参考资料，如《Intel® 64 and IA-32 Architectures Software Developer’s Manual》、《Intel 奔腾系列 CPU 指令全集》等，了解相关指令的格式、功能和用法。

1、C 语言程序示例

下面给出了一个简单的 C 语言程序，实现两个浮点数相加，然后显示结果。

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    float x, y, z;
```

```
    x = 3.14;
    y = 5.701;
    z = x + y;
    printf("%f\n", z);
    return 0;
}
```

2、C 语言程序编译时生成汇编代码

为了更好的仿造 C 的编译结果来写汇编语言源程序，可以在对 C 程序进行编译时让编译器生成汇编语言程序。操作方法：设置“项目属性-> C/C++ ->输出文件->汇编程序输出”，选择“带源代码的程序集(/FAs)”。

3、C 语言程序编译时选用指令集

用 VS2019 的开发环境对上面的 C 语言程序进行编译后，看到的汇编代码并不是本章介绍的指令，而是 XMM 中的浮点数处理指令。但是，在编译时可以设置使用的指令集。操作方法：设置“项目属性-> C/C++ ->代码生成->启用增强指令集”中选择“无增强指令/arch:IA32”，则会生成使用 x87 FPU 指令的程序。

在反汇编窗口可以看到如下的程序片段。

```
x = 3.14;
00A51828  fld         dword ptr [__real@4048f5c3 (0A57B34h)]
00A5182E  fstp        dword ptr [x]
y = 5.701;
00A51831  fld         dword ptr [__real@40b66e98 (0A57B3Ch)]
00A51837  fstp        dword ptr [y]
u = 3.5;
00A5183A  fld         dword ptr [__real@40600000 (0A57B38h)]
00A51840  fstp        dword ptr [u]
z = x + y+u;
00A51843  fld         dword ptr [x]
00A51846  fadd        dword ptr [y]
00A51849  fadd        dword ptr [u]
00A5184C  fstp        qword ptr [z]
printf("%f\n", z);
00A5184F  sub         esp,8
00A51852  fld         qword ptr [z]
00A51855  fstp        qword ptr [esp]
00A51858  push        offset string "%f\n" (0A57B30h)
```

```
00A5185D  call        _printf (0A51046h)
00A51862  add         esp,0Ch
```

4、x87 FPU 程序示例

仿照 C 语言程序的编译结果，可以比较容易的写出汇编语言程序。下面给出了一个例子。

```
.686P  ;  686P 支持 80387 数学协处理器指令
       ;  将处理器选择伪指令换成 .387, .486 等均可
.model flat, stdcall
  ExitProcess proto stdcall :dword
  includelib  kernel32.lib
  printf      proto C :ptr sbyte, :vararg
  includelib  libcmt.lib
  includelib  legacy_stdio_definitions.lib
.data
  lpFmt db  "%f",0ah, 0dh, 0
  x  real4  3.14
  y  real4  5.701
  z  real4  0.0
.stack 200
.code
 main  proc  c
   fld    x   ; 将变量 x 中的内容压入 FPU 数据寄存器栈的栈顶 st(0)
   fadd   y   ; (st(0))+(y)->st(0)
   fst    z   ; (st(0))->z
   sub    esp,  8
   fstp  qword ptr [esp]  ; (st(0))转换成 4 字类型，送到内存堆栈中
   invoke printf,offset lpFmt
   invoke ExitProcess, 0
main  endp
end
```

该程序比用 C 语言编译后生成的汇编代码要简单。细心的读者可能会发现，程序中为什么没有直接使用 “invoke printf,offset lpFmt，z”，而使用“sub esp,8”和“fstp qword ptr [esp]”来传递 z 的值？

直接使用“invoke printf,offset lpFmt，z”，编译并没有错误，但运行结果不正确。原因在于 printf 显示浮点数时，要求一个 double 类型的浮点数，而 z 的定义为 real4。如果将 z 的定义改为 z real8 0.0，就可以直接使用 “invoke printf,offset

lpFmt，z”。另外，将 z 定义为 real4 和 real8 时，汇编语句“fst Z”对应的机器码不同，它分别是将 FPU 栈顶寄存器中的数据转换为单精度浮点数和双精度浮点数。

习题 14

14.1 将十进制浮点数 10.2 转换成二进制的科学表示法。

14.2 IEEE 754 标准中，规格化数据、非规格化数据、特殊值各是什么含义？

14.3 将十进制浮点数 10.2 用 IEEE 754 标准中单精度、双精度数来表示，结果各是什么？

14.4 x87 FPU 中有哪些寄存器？ 各有什么作用？

14.5 设数据段中定义了如下变量：

```
x dd 13
y real4 0.0
```

编写一个程序，完成 y=x 的功能，即将整型变量中的值送入一个浮点数变量中。反过来，编写程序完成 x=y 的功能。

上机实践 14

14.1 用 C 语言编写一个程序，从一组单精度浮点数中找出最大数并输出。要求编译生成代码时，在“启用增强指令集”中选择“无增强指令/arch:IA32”。用反汇编方法观察用到的 x87 FPU 指令。

14.2 编写一个汇编语言程序，从一组单精度浮点数中找出最大数并输出。