《UNIX 环境与 80X86》丛书

保护方式下的 80386 及其编程

周明德　主编

田云　何德书　宗耀堂　编著

M1011

清华大学出版社

目　录

绪论 …… 1
第1章　数据类型 …… 2
1.1　80386的数据类型…… 2
1.1.1　内存 …… 2
1.1.2　约定 …… 3
1.1.3　整数 …… 3
1.1.4　字符串 …… 6
1.1.5　ASCII …… 6
1.1.6　位 …… 6
1.1.7　BCD码…… 7
1.2　80387的浮点数据格式…… 8
1.2.1　浮点概述 …… 8
1.2.2　IEEE浮点标准 …… 8
1.2.3　如果没有80387怎么办? …… 9
1.2.4　数据格式 …… 9
1.2.5　整型数 …… 10
1.2.6　BCD码 …… 10
1.2.7　实数格式 …… 10
1.2.8　临时实型数 …… 12
1.2.9　特别的数 …… 13
1.2.10　异常 …… 16
第2章　机器状态和存储器寻址…… 17
2.1　寄存器 …… 17
2.1.1　通用寄存器 …… 17
2.1.2　处理器控制寄存器 …… 18
2.1.3　段寄存器 …… 22
2.2　存储器寻址的概念 …… 22
2.2.1　两部分的寻址 …… 22
2.2.2　表示法 …… 22
2.3　内存寻址机制 …… 23
2.3.1　段部分:段寄存器 …… 23
2.3.2　偏移部分:地址方式 …… 24
2.3.3　程序堆栈 …… 25
2.3.4　指针数据类型 …… 28
2.3.5　寻址方式和数据结构 …… 28
2.3.6　分段的策略 …… 29
2.4　指令的编码 …… 31
2.4.1　指令的字段 …… 31
2.4.2　立即常数 …… 32
2.4.3　寄存器操作数 …… 33
2.4.4　存储器操作数 …… 35
2.5　I/O空间 …… 42
2.6　浮点寄存器 …… 42
2.6.1　浮点累加器堆栈 …… 43
2.6.2　十六位的状态和控制寄存器 …… 44
第3章　指令集…… 50
3.1　指令格式描述 …… 50
3.1.1　指令格式 …… 50
3.1.2　指令说明 …… 51
3.1.3　指令操作 …… 52
3.1.4　指令举例 …… 52
3.1.5　指令的执行时间 …… 53
3.1.6　指令对标志位的影响 …… 53
3.1.7　指令异常 …… 53
3.1.8　指令的注意事项 …… 54
3.2　指令描述的语法 …… 54
3.2.1　符号 …… 54
3.2.2　函数 …… 57
3.3　指令的分类与摘要 …… 60
3.3.1　整数类指令的分类 …… 60
3.3.2　多段类指令摘要 …… 62
3.3.3　操作系统类指令摘要 … 63
3.3.4　浮点类指令摘要 …… 63
3.4　整数类指令集 …… 65
3.5　多段类指令 …… 172
3.5.1　函数定义 …… 172

3.5.2 异常程序 …… 172
3.5.3 多段类指令集 …… 173
3.6 操作系统类指令 …… 183
3.6.1 函数定义 …… 184
3.6.2 异常程序 …… 184
3.6.3 操作系统类指令集 …… 184
3.7 浮点类指令 …… 204
3.7.1 指令助记符 …… 205
3.7.2 累加器栈 …… 205
3.7.3 例子 …… 205
3.7.4 函数定义 …… 205
3.7.5 异常程序 …… 206
3.7.6 浮点类指令集 …… 207
第4章 保护虚拟地址方式 …… 250
4.1 存储管理机制 …… 251
4.1.1 地址转换 …… 251
4.1.2 段机制和分页机制 …… 251
4.1.3 虚拟存储器概念 …… 252
4.1.4 保护 …… 252
4.2 段机制 …… 257
4.2.1 段描述符表 …… 259
4.2.2 段选择子 …… 260
4.2.3 段描述符 …… 261
4.2.4 存储段描述符格式 …… 263
4.2.5 系统段及门 …… 265
4.2.6 描述符投影寄存器 …… 266
4.3 分页机制 …… 267
4.3.1 页表结构 …… 268
4.3.2 页表项格式 …… 271
4.3.3 虚拟存储页的驻留及修改 …… 273
4.3.4 页级保护 …… 273
4.3.5 修改页表项的软件问题 …… 274
4.4 处理器控制寄存器及系统段 …… 275
4.4.1 处理器控制寄存器 …… 275
4.4.2 段表基地址寄存器 …… 278
4.4.3 任务状态段格式 …… 279
4.5 对特权级敏感的指令 …… 282
4.5.1 被赋予特权的指令 …… 282
4.5.2 I/O 空间保护 …… 283
4.5.3 改变 EFLAGS 的指令 …… 288
4.6 控制转移方法 …… 288
4.6.1 同一特权级、同一任务的转移 …… 289
4.6.2 同一任务不同特权级的转移 …… 289
4.6.3 向外层返回 …… 293
4.7 段机制的详细说明 …… 293
4.7.1 异常概述 …… 294
4.7.2 存储器数据访问 …… 295
4.7.3 控制转移 …… 314
4.7.4 任务切换 …… 330
第5章 中断及异常 …… 340
5.1 中断 …… 341
5.1.1 INTR 中断 …… 341
5.1.2 NMI 中断 …… 341
5.2 异常 …… 342
5.2.1 指令的重新执行 …… 342
5.2.2 异常类型 …… 342
5.2.3 异常小结 …… 346
5.3 中断及异常的优先级 …… 347
5.4 屏蔽中断及异常 …… 347
5.5 中断/异常的转移方法 …… 348
5.5.1 中断及陷阱门 …… 349
5.5.2 NT=0 时的 IRET …… 351
5.5.3 通过任务门的转移 …… 352
5.5.4 任务门与中断/陷阱门的比较 …… 352
5.6 中断/异常的详细说明 …… 352
5.6.1 中断/异常的公共处理 …… 353
5.6.2 存储访问异常 …… 359
5.7 协处理器出错异常 …… 361
5.7.1 屏蔽与未屏蔽的异常 …… 362
5.7.2 协处理器出错类别 …… 362
5.7.3 协处理器出错异常的

优先级 …… 365
第6章 程序举列 …… 366
6.1 应用程序举例 …… 366
6.1.1 语法 …… 366
6.1.2 整数的例子 …… 368
6.1.3 浮点数的例子 …… 388
6.2 80386的编程…… 395
6.2.1 句法 …… 395
6.2.2 初始化实例概述 …… 396
6.2.3 初始化实例说明 …… 401
6.2.4 协处理异常处理程序实例 …… 416
第7章 80386的向下兼容性 …… 420
7.1 16位寄存器及寻址方式 …… 420
7.2 执行8086程序 …… 421
7.2.1 段及寻址 …… 422
7.2.2 8086中FLDENV、FSTENV及FNSTENV指令的格式 …… 425
7.2.3 虚拟8086方式的若干问题 …… 426
7.2.4 实地址方式的若干问题 …… 436
7.3 执行80286保护方式程序…… 440
附录A 排错 …… 442
A.1 术语 …… 442
A.2 调试断点 …… 442
A.3 其它调试功能 …… 447
附录B 80386,80286和8086之间的比较 …… 449
B.1 8086和80386的比较 …… 449
B.2 80286和80386的比较…… 451
附录C 80387,80287和8087之间的比较 …… 452
C.1 80287(和8087)与80387的比较 …… 452
C.2 8087和80387的比较 …… 454
附录D 二进制、十六进制、十进制数对照表 …… 456
附录E 以2为底的幂函数 …… 456
附录F ASCII码表 …… 457
附录G 80386操作码表…… 458
G.1 缩略语表 …… 458
G.2 80386操作码表…… 460
附录H 80386指令格式和时钟周期次数 …… 465
H.1 80386指令编码和时钟周期次数 …… 465
H.2 指令编码 …… 485
附录I 机器指令解码 …… 492
附录J 80387对于80386指令集的扩展 …… 497

绪　论

一、Intel 微处理器发展史

1971 年，Intel 公司的第一块微处理器诞生了，它叫做 4004，是一块 4 位的 CPU。4004 很快被增强为 8 位 CPU——8008。在今天看来，4004 和 8008 这兄弟俩可能很不起眼，但在当时，它们确是了不起的新事物。它们曾在教学、程控、计算等应用领域中发挥过作用，然而并没有将它们正式装过象样的计算机。我们将这两种芯片叫做 Intel 公司的第一代微处理器。

1974 年，Intel 推出了它的第二代微处理器 8080。这是第一个通用的微处理器，它的出现，在微处理器工业史上是个十分重要的里程碑。8080 是 8 位的 CPU 芯片。1978 年推出的第三代微处理器 8086，标志着微处理器作为"实际的"计算机的开始。86 系列从此诞生了，8086 是该系列的第一个成员。8086 是 16 位的处理器。8086 的小兄弟 8088(准 16 位的，数据总线仍为 8 位)，曾经引起了个人计算机的革命。自从 1981 年 IBM 公司推出风靡全球的 PC 机以来，8088 微处理器结构几乎出现在每一台早期的 PC 及其兼容机上。不仅如此，由于 PC 及其兼容机的普及，和 PC 拥有的软件之丰富，众多其它类型计算机设备的设计者们也都乐意使用 8086 或 8088。这样一来，8086 结构从它诞生之日起直到现在，一直是世界上最主要的微处理器结构。

1982 年，86 系列增加了新成员 80186。该芯片在结构上与 8086 一样，只是在同一片芯片上增加了一些公用的系统。同年，8086 体系结构的超集 80286 问世。80286 能做 8086 所做的一切，并增加了许多功能，特别是对多任务的支持，也就是说，可以同时执行一个以上的应用程序或任务。为了支持多任务，80286 解决了任务或任务之间的，以及任务的内存空间之间的保护问题。80186 和 80286 都是 16 位的。86 系列中最年轻的也是功能最强的成员是 80386，它是 1985 年发布面市的。80386 比它的兄长们有两个最主要的增强：一是 32 位的操作和数据类型；二是采用了页式管理技术。在 80386 以前，86 系列的成员在内存管理上都只有用段式管理，而 80386 不仅采用段式管理，还采用页式管理。这，我们在第 4、5、6 章中还将详细讨论。除此之外，80386 还拓展了 80286 的多任务管理能力，可以同时执行 8086、80286、80386 的应用程序、任务，甚至操作系统。

最后值得一提的是浮点协处理器。86 系列的每一代产品都有其相应的浮点运算协处理器，如 8087 对应 8086，80287 对应 80286，80387 对应 80386。这些协处理器和它们的处理器紧密耦合，形成了能够支持浮点运算及其数据结构的计算机结构。

二、和 8086、80286 的兼容

86 系列的每一代新产品都和老的产品保持完全兼容。80386 可以执行 80286、8086 的任何程序(第 7 章将详细讨论)。但本书并不在兼容性上多费笔墨，而集中讨论 80386 上的全部 32 位机的特性，以及如何通过软件使 80386 成为一台实际的 32 位机。

本书对象为具备 8086 和 80286 编程经验的、从事计算机开发和应用的教育、研究人员和工程技术人员。所以在介绍它们兼容性的时候，只须指出系列产品和 80386 之间的区别就够了(请阅读附录 B)。如果读者还想了解 8086 和 80286 的编程的话，那么可以利用的资料和手段是相当丰富的，但它们不在本书的范畴之中。

第1章 数据类型

1.1 80386的数据类型

从本质上说，计算机所能做的全部工作可归结为对数据的存储、检索和处理。所以，想要掌握某种计算机的编程，必须先了解该机支持的数据类型。常用的数据类型有：带符号整数，无符号整数，BCD(包括压缩的和非压缩的二-十进制码)，字符串，位，浮点数。上述数据类型，大多数可在常见的机器中找到。这里，我们着重介绍80386和其它机器，尤其是和8086、80286在数据类型上的差别。

1.1.1 内存

在详细讨论数据类型之前，先说一说内存的组织。和所有的传统的计算机一样，内存是一切信息最主要的源泉和归宿。简单说来，内存即是有唯一地址的字节的有序阵列。在正常情况下，地址从0开始，一直排到机器所能支持的最高地址。80386是32位机，一共有2^{32}个物理地址，或者说成"总共有4G(4×10^9)字节的物理内存"。注意这里是"物理"地址范围。在第2、4、6章中，你可以看到：在段式管理和页式管理中的虚拟内存的地址范围，可以大大超过2^{32}(4G)。

某些类型的数据，其二进制的表达形式超出8位(16位、32位、甚至更多)，于是不得不用多个相邻字节来存放一个数据值。常见的有两个相邻字节组成的字(Word)和4个相邻字节组成的双字(DWord或Double Word)。对字来说，这个16位的数据就有2^{16}个可能的值。对双字来说，这是个32位的数据，就会有2^{32}种可能的值。

存取多字节数据，首先要解决的问题是：数据的最低有效8位是放在相邻字节(地址号连续)的地址最低者(地址号最小的字节)之中，还是放在相邻字节的地址最高者(地址号最大的字节)之中，也就是首先要解决字和双字存取的基准的问题。关于基准的方案不外乎有两种，请看图1.1和1.2。

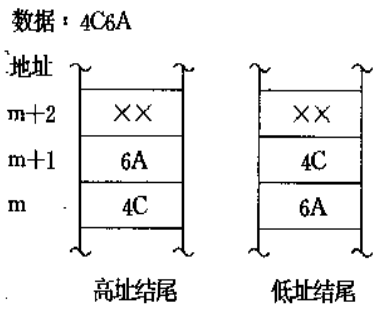


图1.1 高址结尾和低址结尾

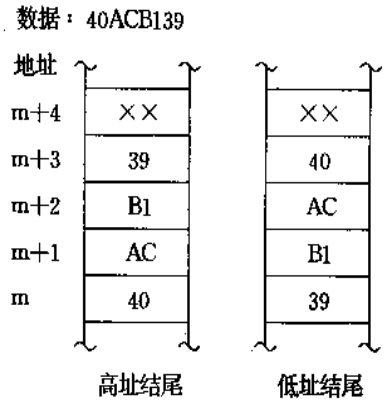


图1.2 双字的高址结尾和低址结尾

图 1.1 表示两个相邻字节存放一个字的方案。两图中左边的方案，将最低有效 8 位放在最高地址字节中，将最高有效 8 位放在最低地址字节中，这种方案被称之为“高址结尾(Big Endiun)”。两图中右边的方案，与左边的正好相反，将最低有效 8 位放在最低地址字节中，而将最高有效 8 位放在最高地址字节中，这种方案叫做“低址结尾(Little Endiun)”。

请读者记住，80386 采用的是低址结尾，即两图中右边的方案。若采用最低地址为(m)的两个相邻字节来存放一个字，该字的地址亦取(m)，该字的最低有效 8 位放在地址为(m)的字节中。

同理，若采用最低地址为(m)的 4 个相邻字节来存放一个双字的话，该双字的地址就是(m)，该双字的最低有效 8 位放在地址为(m)的字节中，而最高有效 8 位则放在地址为(m+3)的字节中。

1.1.2 约定

在这一节里，我们交代一下本书采用的有关数字表示方式的某些约定：

在数字后面加上“h”的，表示一个十六进制数。如 100h=256。在数字后面加上“b”的，表示一个二进制数。如 100b=4。在数字后面既没有“h”，也没有“b”，则表示一个十进制数。附录 D 给出了二进制、十六进制和十进制之间的转换表。

1.1.3 整数

和许多计算机一样，80386 也支持 8 位长度的字节、16 位长度的字、32 位长度的双字的整数。32 位的整数在 86 系列的其它机器 (8086、80186、80286)中是没有的。附录 E 给出了从 2^0 到 2^{32}的 2 的幂的表。

表 1.1 定义了全书通用的几种简略表示法和它们真正的值。

表 1.1 2 的次幂的缩简表

简 写	2 的次幂	十进制值
1K	2^{10}	1024
4K	2^{12}	4096
16K	2^{14}	16384
32K	2^{15}	32768
64K	2^{16}	65536
2G	2^{31}	2147483648
4G	2^{32}	4294967296

整数又分为无符号整数和带符号整数。80386 各种类型的整数和它们的有效数值范围如下页表：

长度	类　型	范　围	
		十六进制	十进制
8	字节的带符号整数 字节的无符号整数	80—00—7F 00—FF	−128 到 127 0 到 256
16	字的带符号整数 字的无符号整数	8000—0000—7FFF 0000—FFFF	−32K 到(32K−1) 0 到 64K
32	双字的带符号整数 双字的无符号整数	80000000—00000000—7FFFFFFF 00000000—FFFFFFFF	−2G 到(2G−1) 0 到 4G

1.1.3.1　无符号数整数

图 1.3 表示 80386 内存中无符号数据格式的位结构。

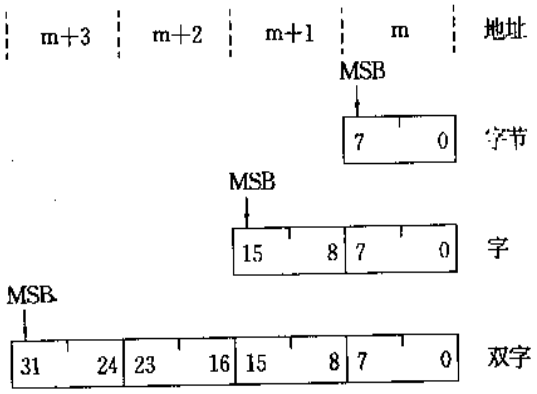


图 1.3　无符号整数

图中假定它们的起始地址均为 m，位 0 为最低有效位(LSB)，MSB 为最高有效位。

1.1.3.2　带符号整数

表示带符号整数的方法有以下四种：偏移值(Biased Numbers)，带符号的值(Sign Magnitude)，二进制反码(One's Complement)，二进制补码(Two's Complement)。

表 1.2 表示了 8 位长的带符号数的四种表示方法。80386 和 86 系列的所有成员都采用二进制补码作为带符号整数的表示法。

在描述二进制补码表示法之前，我们先简要介绍其余三种整数表示法，因为下面讨论浮点数据类型时，会用到这些表示法。

(1)　偏移值

由于偏移值表示法具有便于进行数据比较的优点，所以常用来表示浮点数的指数。在原来的正数或负数之上再加一个偏移量(Bias)，便得该数之偏移值。这个偏移量通常使表示法允许的最小负数变为 0，而使最大的正数变成表示法中的最大数。请看表 1.2 中的例子，其偏移量选择为 127，它使得最小负数(−127)的偏移值变成了 0，而最大的正数(128)则变成了 255。

(2)　带符号的值

所谓带符号的值即为数前冠以符号的值(或绝对值)。在带符号的数值的表示法中，有一

位用来表示符号(0 为正,1 为负),而其余的位则为值的二进制表示形式。浮点数的有效数字便是用这种方法表示的,其符号位给出了整个浮点数的符号。

表 1.2 带符号整数格式

十进制值	2 的补码	2 的反码	偏移值(=127)	带符号的值
128	无此值	无此值	11111111b	无此值
127	01111111b	01111111b	11111110b	01111111b
126	01111110b	01111110b	11111101b	01111110b
⋮	⋮	⋮	⋮	⋮
2	00000010b	00000010b	10000001b	00000010b
1	00000001b	00000001b	10000000b	00000001b
0	00000000b	00000000b	01111111b	00000000b
−0	无此值	11111111b	无此值	10000000b
−1	11111111b	11111110b	01111110b	10000001b
−2	11111110b	11111101b	01111101b	10000010b
⋮	⋮	⋮	⋮	⋮
−126	10000010b	10000001b	00000001b	11111110b
−127	10000001b	10000000b	00000000b	11111111b
−128	10000000b	无此值	无此值	无此值

(3) 二进制反码

对于正整数,其二进制反码与带符号的值具有相同的形式,即最高有效位为 0(表示正数),其余各位为该值(绝对值)的二进制表示。而负整数的二进制反码,由该数绝对值的二进制码逐位取反得到。由于此法简单,容易计算,所以在早期的计算机中用得很普遍,但现在用得少了。

(4) 二进制补码

所谓二进制补码,即是在二进制反码之上再加以 1。其最高有效位(MSB)也是符号位,而且也是 0 为正,1 为负。由于二进制补码易于实现,通过用于无符号数的简单的加法器就能实现从负数到二进制补码的转换,所以它得到了普遍的采用。这种优点对于既支持无符号数,又支持二进制补码数据的 80386 来说,也是颇为重要的。

图 1.4 给出了 80386 的二进制补码格式。

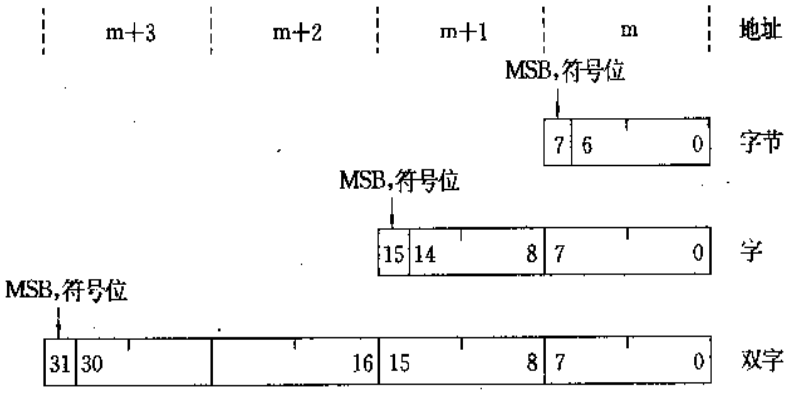


图 1.4 二进制补码字节、字、双字

80386 可对二进制补码进行各种各样的算术运算。第 3 章将讨论这些操作。和所有 86 系

列成员一样，80386 的二进制补码运算也能产生溢出。这些情况将在第 2 章中说明。

1.1.4 字符串

字符串就是字节、字、双字的相邻序列。对双字字串的支持只有 80386 才有，86 系列的其它成员是没有的。串的长度可从 1 到 2^{32}个元素。图 1.5 显示了这三种类型的字符串。

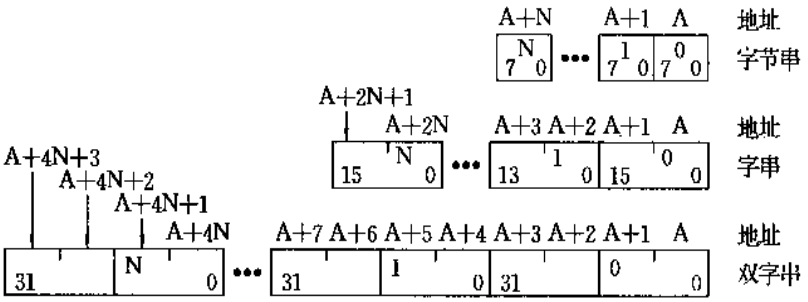


图 1.5 字节、字、双字字符串

80386 对字符串操作的指令有：将字符串从内存的一个区传送到另一个区、比较两个字符串、将一字符串用固定的值来填充、对 I/O 口的读写操作、按指定的数据值搜索字符串等。

1.1.5 ASCII

字符串的最常见的格式是 ASCII 码，因为大多数来自终端的数据都为 ASCII 码。附录 F 就是一个完整的 ASCII 码表。它包括数字、字母、特殊字符和控制字符。

在第 3 章中，我们将会看到 80386 支持对于 ASCII 码的诸如加和除的算术运算。

1.1.6 位

对位串操作的支持，在 86 系列中只有 80386 才有。

位支持是十分重要的，因为数据用单一的位来表示也是常见的。比如说，“位图”便是大家十分熟悉的例子。在位图显示中，每一个映射到内存中单一的位的“象素”，都用一个圆点(.)来表示。若某个圆点发光，表示该位为 1；若圆点不发光，表示该位为 0。还可以举些例子，如用位值来表示信号灯的忙(1)和闲(0)。当然，将一个完整的字节当作一个“象素”或“信号灯”不是不可以，但这样要浪费许多空间。仔细算一下，如果用一个字节，来表示上面例子中的一个数据元素，那么这个字节中有七位是无用的，也就是说，87.5%被浪费了。

80386 支持的位串，可以包含 2^{32}个位，并通过一个带符号的双字来检索。到了第 3 章我们再谈它们的实际操作。图 1.6 表示一个位串在内存中的布局。位的索引是个带符号数。在位串中不存在最低有效位的概念；即使是第 0 位，也不是最低有效位。

一个位在位串内部的地址用 32 位的带符号整数来表示，这整数叫做“位偏移量”，取值范围从(−2G)到(2G−1)。位偏移量由字节地址和位余数组成。位偏移量除以 8，便得到该位所在字节的地址，而位偏移量对于 8 的模指出了被检索的位在所在字节中的位置。图 1.7 举了两个例子，图中两个位的位偏移量分别为(23)和(−18)，它们都在地址为 N 的位串中。在第 6 章我们还要详细研究。

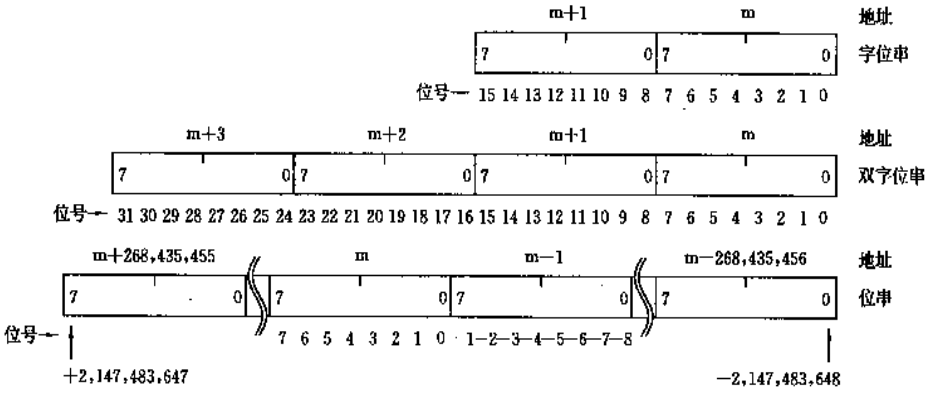


图 1.6 位数据类型

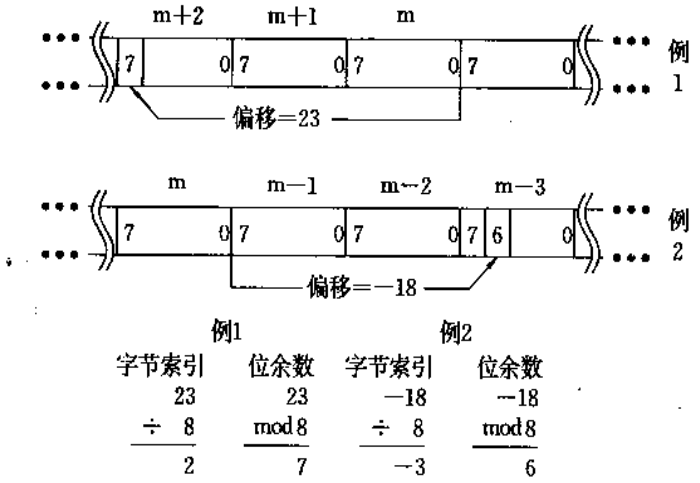


图 1.7 位串偏移量的例子

1.1.7 BCD 码

在 8086、80286 或其它机器上编过程序的读者，都已熟悉了 BCD 码，所以用不着多介绍了。表 1.3 给出了一张简单的 BCD 编码表。还告诉大家，在第 3 章中有 80386 对 BCD 进行加减操作的指令。

表 1.3 十进制数的二进制编码

二进制数	十进制数	二进制数	十进制数
0000 0000b	00	0000 1010b	0X
0000 0001b	01	0000 1011b	0X
0000 0010b	02	0000 1100b	0X
0000 0011b	03	0000 1101b	0X
0000 0100b	04	0000 1110b	0X
0000 0101b	05	0000 1111b	0X
0000 0110b	06	0001 0000b	10
0000 0111b	07	0001 0001b	11
0000 1000b	08	⋮	
0000 1001b	09		

注意：对于 BCD 表示法 X 表示非法值

80386 只会直接处理单字节的 BCD 码。多字节的 BCD 码如何处理在第 3 章里讲。每字节放两个 BCD 码位，0 到 3 位放的是低位，4 到 7 位放的是高位。非压缩的 BCD 码只在每个字节的 0 到 3 位放置一个码位(4 到 7 位空闲)。

1.2 80387 的浮点数据格式

80387 是 80386 的协处理器，它和 8087、80287 十分类似。

1.2.1 浮点概述

到目前为止，我们讨论过的众多数据类型，无论是带符号整数、无符号整数、字符串、位串、ASCII和BCD都无法表示带有分数或小数的数。尤其是一些重要的物理常数，象：3.1416，8.854×10^{-14}，6.02×10^{23}，9.81，9.11×10^{-31}，1.38×10^{-23}等等，它们既有整数部分又有小数部分，在数学上称之为实数或有理数。在本书里，为了简单起见，统称它们为实数，不管这样称呼在数学上究竟是否严格。其实前面讨论过的整数也是实数，只不过是实数集合中小数部分为 0 的子集而已。

在计算机里，用来表示数的位数是有限的，所以计算机不能精确地表示出所有的实数。而只能表示无穷个实数中的一个极小的子集。然而，就是这个子集，已经可以解决极大多数的实际问题，而且丢掉的精度也是微不足道的。所谓“浮点数”，就是实数类型数据的计算机表示法。这里向读者强调两个不同的概念：“实数”是描写一个数的集合的数学术语；而“浮点”则是计算机用来表示实数的一个子集的数据类型。

下面我们来比较几种方法，以说明浮点运算的必要。假如现在让计算机来处理一笔款项，¥23.41。当然你可将它乘上 100，变成了“2341”一个整数。用键盘输入，再将输入的 ASCII 码转换为 2341 的内部整数表示格式。于是在机内，对于整数格式的各种操作，似乎全能用上了。当需要输出之前，再将机内结果除以 100 得到实际的结果。但是想象之中的许多问题也随之发生了。其中之一，就是精度问题。如果你的 23.41 元钱，加上得到的 7%的利息，你是拿到¥25.04呢？还是拿到¥25.05？如果不讲足够的精度的话，你拿¥25.04，恐怕要吃亏一分钱呢。

还有一种方法，叫定点数。但须在一个字节中，用 3 个位作为小数部分。例如：

$$10110110b = 2^4 + 2^2 + 2^1 + 2^{-1} + 2^{-2} = 22.75$$

这种方法，所能表示的数的范围不大而且是固定不变的(从 0 到 2^5)，增量为 2^{-3}。这些问题的解决，一定要靠浮点表示法。浮点格式可以使二进制小数点浮动。它有专门用来规定二进制小数点的位置的部分，还有用来表示数据有效数字的部分。在保证精度的同时，还可以有效地拓展数的表示范围。

浮点数实际上是靠两个整数来表示的，一个保持着数据的有效数字，一个规定了二进制小数点的位置。同时浮点操作可分解为作用于这些整数对的整数指令(加，乘和位移)序列。但是浮点运算光靠软件来完成，其速度慢得无法忍受，所以人们设计出许多硬件来直接模拟浮点运算。80387 就是 Intel 公司为用户奉献的支持浮点运算的数字协处理器。

1.2.2 IEEE 浮点标准

在深入研究 80387 的数据类型之前，我们想对 IEEE 浮点标准作一些说明。

早在 1979 年之前，就有许多大型的和微型计算机采用了浮点数据类型。然而当时却没有

一个浮点表示法的统一标准，以致于同一个 FORTRAN 程序，在不同的机器上运行，得出的结果也不一样。

在开发 8087 同时，Intel 公司对浮点的标准产生了兴趣，尤其是想开发一个供 8087 来实现的浮点标准。在 Intel 公司公布了它提出的标准之后不久，IEEE 成立了一个关于开发浮点运算标准的委员会。经这个委员会许多成员几年的辛勤劳动，产生了 IEEE 任务 P754 草案 10.0"二进制浮点运算标准"(Draft 10.0 of IEEE Task P754，"A Standard for BinaryFloating-Point Arithmetic")。

草案中声明的该标准的部分宗旨如下：

(1) 促使已经存在于形形色色的计算机中的程序移植到遵循这个标准的机器上来。

(2) 增强编程者的才干和安全有效性，使那些虽不是谙熟数学方法的编程者也能产生高级的数学程序。

(3) 鼓励经验丰富者开发和奉献出实用而有效的计算程序，通过很少的编辑和再编译，移植到任何与该标准一致的机器上来。

这个 IEEE 标准被工业界广泛接受。它不但得到 Intel 芯片的支持，还得到许许多多微处理器、微型的和大型的计算机的支持。当然 80387 也不例外。

1.2.3 如果没有 80387 怎么办？

在一个 80386 系统中不含有 80387 是完全可能的，因为 80386 和 80387 是两个分离的硅集成芯片。若是你经常要处理大量的计算的话，可以用两种办法来解决：

(1) 在系统安装一块 80387。几乎所有的计算机制造商都会配备 80387 选件。

(2) 利用机内的数学模拟软件包。如果连软件包也没有，则设法配备或自编一个。

无论是用 80387 还是用模拟软件包，总是离不开为了 80387 设计的浮点数据类型和指令。也许你的系统上只有 80287，没有 80387。那么 80287 也可以运行，但要注意 80287 与 80387 的区别(请看附录 C)。

1.2.4 数据格式

表 1.4 列出了 80387 支持的七种数据类型。请注意它们表示的数的范围。80386 支持的最大整数为 2^{32}，或者 4.29×10^{9}(十进制)，这个数字很重要，要记住。

表 1.4 80387 支持的数据类型

数据类型	位	有效数字(十进制)	大致范围(十进制)
字的整数	16	4	$-32768\leqslant X\leqslant 32767$
短型整数	32	9	$-2\times10^{9}\leqslant X\leqslant +2\times10^{9}$
长型整数	64	18	$-9\times10^{18}\leqslant X\leqslant +9\times10^{18}$
压缩十进数(BCD)	80	18	$-99..99\leqslant X\leqslant +99..99$(18 位)
短型实数	32	6—7	$-3.39\times10^{-38}\leqslant X\leqslant 3.39\times10^{38}$
长型实数	64	15—16	$-1.80\times10^{-308}\leqslant X\leqslant 1.80\times10^{308}$
临时型实数	80	19	$-1.19\times10^{-1932}\leqslant X\leqslant 1.19\times10^{1932}$

80387 也支持整数数据类型。于是一个既有实型数据又有整型数据的复杂计算只用一个 80387 就能完成，省掉了 80387 和 80386 之间的数据传送。

1.2.5 整型数

下面一张表列出了 80386 和 80387 支持的整型数：

位	80386	80387
8	带符号字节	不支持
16	带符号字	字整数
32	带符号双字	短整数
64	不支持	长整数

所有整型数都用二进制补码来表示。16 位和 32 位的整型数，虽同时得到两种芯片的支持，但不同的芯片对它们的称呼不同。图 1.8 表示了 80387 支持的三种整型数。字整型数可以表示从(−32768)到(32767)范围内的整数；短整数可以表示从(-2.147×10^9)到(2.147×10^9)范围内的整数；长整数可以表示从(-9.233×10^{18})到(9.233×10^{18})范围内的整数。

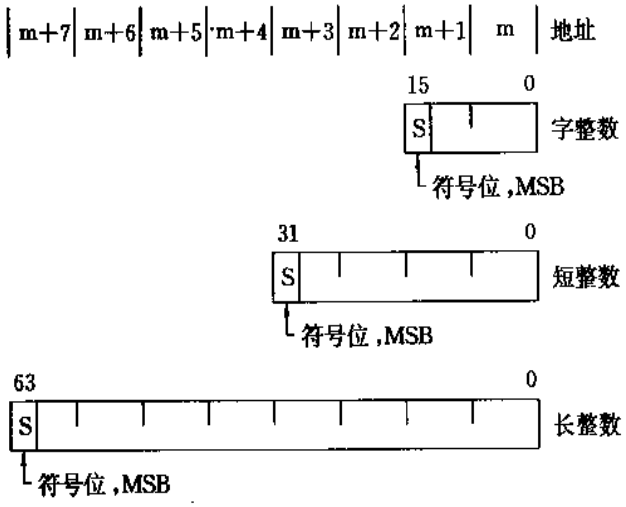


图 1.8 80387 的整型数(二进制补码)

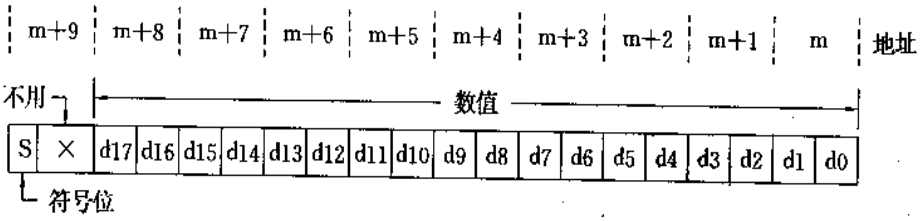
1.2.6 BCD 码

80387 支持的是 80 位的压缩的二－十进制数据类型，可以保持 18 位的十进制数和一个符号位。如图 1.9 所示。BCD 码的 18 位数字，正好和 COBOL 的标准符合。虽然 80 位的 BCD 码中，有 7 位没有被利用，但也没有必要搞到 19 位，反而使得用起来复杂了。

1.2.7 实数格式

实数格式就是浮点格式。图 1.10 给出了实型数的一般格式。这种格式是由有效数字、指数、符号三部分组成。下面给出在三种实型数中，上述每一部分所占的位数：

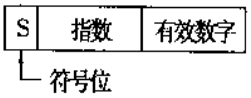
数据类型	总位数	符号位	指数字段	有效数字段
短实型数	32	1	8	23
长实型数	64	1	11	52
临时实型数	80	1	15	64



d_i

十进制	位
0	0000
1	0001
⋮	⋮
8	1000
9	1001
不用	1010
	⋮
	1111

图 1.9　80387 的 BCD 数据类型



这里,值=$(-1)^{符号}$*有效数字*$2^{指数}$

图 1.10　一般实型数格式

图 1.11 给出这三种数据类型的详细分布。若符号位为 1,表示该数为负,反之为正。

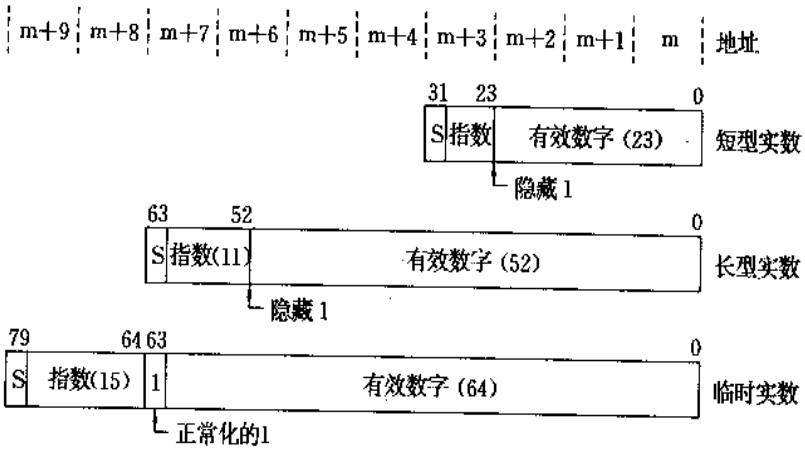


图 1.11　80387 的实数类型

有效数字有时被称之为尾数。下面我们就来说一说尾数的规则。在科学计数法中,同一个数可以有许多表示形式。例如:

$$1 = 100.0 \times 10^{-2} = 10.0 \times 10^{-1} = 1.0 \times 10^{0} = 0.1 \times 10^{1} = 0.01 \times 10^{2} = \cdots$$

在 80387 的实型数表示法中,虽然采用的是二进制,但上述问题是同样存在的。于是就需要做

一个规定。这个规定就是：任何实型数只能用下面的格式表示：

$$1.xxxxx \times 2^n \qquad (x\text{ 表示 1 或 0})$$

在二进制小数的左边，有且仅有一位，而且该位永远为 1。这样便产生了尾数的规则。

既然在二进制的有效数字中，小数点左边的一位永远是 1，那么我们只要承认这个事实，并将它永远"记在心中"就够了，不一定要把它"写"出来。省掉的这个"1"，还可以贡献出本应属于它的那个位空间，以提高表示近似数精度。80387 就在它支持的短实型数和长实型数的记数法中，压缩了小数点左边的这一位。在图 1.11 中看到的"隐藏 1"，就是这个被压缩了的位。于是，一个短的或长的实型数据，如果它的尾数为 0111…010b 的话，它的真正的有效数字应该是 1.0111…010b。指数字段是用偏移法来表示的带符号整数。在三种格式中，它们的偏移量分别为 127、1023 和 16383。例如，在短实型数中，指数 10000000b 对应为 2^1。偏移法和其它带符号整数表示法相比，有便于比较大小的优点。

图 1.12 可以和图 1.10 对照，它图解了我们刚刚解释过的浮点格式中的三个分量。

隐藏1

有效数字

$(-1)^{符号} * (1.S_{23}S_{22}S_{21}\cdots S_0) * 2^{(指数-127)}$ 短型实数

范围： $\pm 3.39 * 10^{38}$

精度： $\pm 1.18 * 10^{-38}$ (微小数:$1.40 * 10^{-45}$)

隐藏1

有效数字

$(-1)^{符号} * (1.S_{51}S_{50}S_{49}\cdots S_0) * 2^{(指数-1023)}$ 长型实数

范围： $\pm 1.80 * 10^{308}$

精度： $\pm 2.23 * 10^{-308}$ (微小数:$4.94 * 10^{-324}$)

$(-1)^{符号} * (1.S_{63}S_{62}S_{61}\cdots S_0) * 2^{(指数-16383)}$ 临时实数

范围： $\pm 1.19 * 10^{4932}$

精度： $\pm 3.36 * 10^{-4932}$ (微小数:$3.65 * 10^{-4951}$)

图 1.12 80387 支持的三种实型数的图解

图 1.12 中的"范围"给出了三种实型数所能表示的数值的范围(最大的绝对值)。图中的"精度"给出了三种实型数所能表示的最小正负数(最小指绝对值)。"微小数(Denormals)"留到以后再作解释。下面的例子都以短实型数的格式给出：

符号位	指数位	有效位	含义	值
0	10001110b	000100…0b	$(1+2^{-4})\times 2^{15}$	34816
1	01111100b	011000…0b	$-(1+2^{-2}+2^{-3})2^{-3}$	−0.17188
0	01111111b	000000…0b	$(1)\times 2^0$	1

1.2.8 临时实型数

我们上面讨论过的三种实型数格式，除了表示法的三个部分的位数不同以外，其余都十分

相似。但临时实型数格式和它们就不同了：

(1) 临时实型数格式是 80387 内部的格式。无论你给出什么样类型的数(短整数、BCD、短实型数等等)，在 80387 内部都将它们转换成临时实型数。

(2) 临时实型格式没有隐藏位。一个格式化的数的临时实型数格式的第 63 位永远为 1。80387 采用此临时实型数，最大限度地提高了运算精度，扩大了运算范围。假如你用短实型数做一个计算，往往会碰到中间结果大大地超出短实型数的表示范围的情况，尽管最后的结果仍然用短实型数来表示。

因为临时实型数在 80387 内部要直接参加运算。所以如果有"隐藏 1"的话该请它出来参加运算了。临时实型数的第 63 位，就是这位隐者出场的地方。当短型或长型实数格式在 80387 内部转换成临时实型格式时，第 63 位就自动插入"1"。

1.2.9 特别的数

80387 支持的三种格式(短实型数、长实型数和临时实型数)，在机内可能出现的数据表示，用表 1.5 到表 1.7 列出。

表 1.5 短的和长的实型数表示

符号	偏移指数	有效数字	种类
0/1	11…11	11…11	静的 NaN
0/1	11…11	⋮	
0/1	11…11	10…00	
0/1	11…11	01…11	信号的 NaN
0/1	11…11	⋮	
0/1	11…11	00…01	
0/1	11…11	00…00	无穷数
0/1	11…10	11…11	正常数
0/1	⋮	⋮	
0/1	00…01	00…00	
0/1	00…00	11…11	微小数
0/1	00…00	⋮	
0/1	00…00	00…01	
0/1	00…00	00…00	零

表 1.6 临时实型数表示

符号	偏移指数	有效数字	种类
0/1	11…11	111…11	静的 NaN
0/1	11…11	⋮	
0/1	11…11	110…00	
0/1	11…11	101…11	信号的 NaN
0/1	11…11	⋮	
0/1	11…11	100…01	
0/1	11…11	100…00	无穷数
0/1	11…10	111…11	正常数
0/1	⋮	⋮	

续表

符号	偏移指数	有效数字	种类
0/1	00…01	100…00	
0/1	00…00	111…11	伪微小数
0/1	00…01	⋮	
0/1	00…01	100…00	
0/1	00…01	011…11	微小数
0/1	00…01	⋮	
0/1	00…01	000…01	
0/1	00…01	000…00	零

表 1.7　不支持的临时实型数表示

符号	偏移指数	有效数字	种类
0/1	11…11	011…11	伪 NaN
0/1	11…11	⋮	
0/1	11…11	000…01	
0/1	11…11	000…00	伪无穷数
0/1	11…10	011…11	非正常数
0/1	⋮	⋮	
0/1	00…01	000…01	
0/1	11…10	000…00	伪零
0/1	⋮	000…00	
0/1	00…01	000…00	

在这些表中，除了在各种表示法的正常数据范围内的正常数(Normals)以外，我们还可以找到零(Zero)、微小数(Denormals)、无穷数(Infinities)、信号的非数字(SignalingNaN)、静的非数字(Quiet NaN)等，这些特别的数。下面我们就来逐一讨论这些数。

1. 零

零的偏移指数为 00…00b，有效数字段亦为 00…00b。零的偏移指数是保留的，也就是说，零的偏移数不能用来表示正常的实型数。并且注意，只在零作除数时，零才有正负之分，否则正零和负零没有区别。这一点，在第 3 章介绍 FDIV 指令时还会谈到。

2. 无穷数

无穷数的偏移指数 11…11b 也是保留的，不用来表示正常的实型数。无穷数的有效数字段全是为 0。无穷数分正无穷和负无穷。

3. 微小数

微小数用来表示非常小的数。在许多计算机中，从最小的可以规格化的数到零的表示法之间，没有任何形式的过渡。即，比最小的规格化的数再小一点的数，便是零的表示式。我们称这种从最小数到零的下溢为“陡然下溢”。80387 则不然，它为最小的正常数和零之间提供了平滑或渐变，它在两者之间插入了一系列从小到大的表示比零大，比最小正常数小的“微小数”表示法。80387 的这种下溢，我们称之为“逐渐下溢”。逐渐下溢使用的表示法，叫做微小数表示法，或非规格化的表示法。这种表示法会引起精度丢失，但它有效地拓展了能表示的非常小数的范围。很显然，尽管逐渐下溢要丢失一些精度，但还是比陡然下溢要优越得多。

在正常情况下，数字需要规格化(左移，直至有效数字的最高有效位为1)。然而，微小数的有效数字的最高位为0，它不符合规格化的要求。微小数的偏移指数为00…00b也是2^{-126}，2^{-1022}，和2^{-16382}的特殊表示形式。说它"特殊"，是因为2^{-126}，2^{-1022}和2^{-16382}的正常偏移指数为00…01b。微小数总会引起精度丢失；否则的话，便使用正常数。从图1.12上看出，利用微小数可以有效地表示非常小的小数。

4. 伪微小数(Pseudo Denormals)

80387支持伪微小数，但不会产生伪微小数。伪微小数和微小数的区别在于前者的有效数字的最高位为1，后者的有效数字的最高位为0。伪微小数的有效数字是符合规格化的要求的。伪微小数的偏移指数仍为00…00b。我们曾经说过，偏移指数00…00b和00…01b是等值的，都对应着2^{-126}，2^{-1022}，和2^{-16382}。所以说伪微小数可以用，但没有用规格化的格式表示。这就是它的特别之点。伪微小数是用微小数格式表示的正常数。

5. 非数字(NaN——Not a Number)

NaN——仅仅具有数字形式，但它并不是数字。NaN分为两大类：信号的SNaN和静的QNaN。当在操作中使用了SNaN，会引起非法操作异常；而在操作中使用了QNaN时，不会引起异常。这就是它们的区别，也是它们的命名的原因。

SNaN在短实数和长实数中，最高有效位为0，在临时实数中最高有效位是隐藏位为1，故第二位为0。

80387并不产生SNaN，而是编程器或编译器先将程序中的所有变量初始化为SNaN。于是若有变量在赋值前就被利用的话(此时该变量值仍是一个SNaN)，非法异常便产生了。用这种方法，就能检查出那些未赋值而使用的变量，这是应用SNaN的一个例子。NaN能在其有效数字段的小数部分存放关于产生这个NaN的场所和原因的各种信息。QNaN在短实数和长实数中，最高有效位为1，在临时实数中，最高有效位为1，且第二位也为1。QNaN产生于非法操作异常发生时；当这种异常发生，产生的结果是不定的(在下面给出)。在所有情况下(除了第3章要讨论的FCOM、FIST和FBSTP)，QNaN和SNaN被保留在操作的全过程中。在处理NaN时有下面几种特殊情况：

(1) 引用了NaN的无效操作异常，将产生不定的QNaN。整数、BCD和实数的不定在下面给出。

(2) 当操作含有SNaN和QNaN，其结果为QNaN。

(3) 两个SNaN之间的操作将产生一个更大的SNaN，然后再转换为QNaN。这种转变很简单，只要把有效字段的最高位置1就行了。

(4) 两个QNaN之间的操作将产生一个更大的QNaN。

(5) 一个SNaN和一个正常数之间的操作将产生一个SNaN，再转换成QNaN。

(6) 一个QNaN和一个正常数之间的操作将产生一个QNaN。

为了更清楚地说明上述结果，表1.8总结了有关NaN的操作。表中假定对操作数1和操作数2(op_1和op_2)发生了操作。

不确定值是上述静态NaN的特例。80387支持的每个数据类型(字，短的和长的整数，BCD和短长临时实型数)有其唯一的不确定值的表示法。当一个非法操作异常发生并且输入操作数中没有NaN时，80387就产生这种表示法。产生非法操作的指令在第3章中给出。非法异常如何产生在第5章中讨论。表1.9总结了每种数据类型的不确定值的编码。编码中最左边的一位是最高有效位。对于实数的表示法，从左到右分别是符号字段、指数字段以及有效

数字字段。

要注意，当我们讨论 BCD 数和整数类型时，我们没有说出全部情节。因为我们尚未提到不确定值。对整数数据类型，当产生不确定值时，就产生最负表示法。而对于 BCD 数则用未用过的编码来表示。

对于实数格式，可以装载和存储不确定值。因为实数的不确定值属于 NaN，当装载一个不确定值时是不会产生异常的。对于整数和 BCD 数，是不能装载不确定值的。对于整数，当装载不确定值时是被当作最大负数来处理的。对 BCD 数装载不确定值的结果没有定义。

表 1.7 给出的不被支持的临时值数据的表示法纯粹是因为 80387 不支持之故(有些是被 80287 所支持)。如果 80387 遇到这些数据格式将引起非法操作异常。

表 1.8　含有 NaN 的操作

		操作数 2		
		Q	S	R
操作数 1	Q	>Q	Q	Q
	S	Q	CQ(>S)	CQ(S)
	R	Q	CQ(S)	QI

关键字	
Q	静的 NaN
S	信号的 NaN
R	规则实数(非 NaN，合法数)
>Q	操作数 1 和操作数 2 的较大的 NaN
>S	操作数 1 和操作数 2 的较大的信号 NaN
CQ(X)	把 X 转换到 QNaN
QI	默认的静的 NaN—不定的实数

表 1.9　不确定值的编码

形　式	位	编　码
字整数	16	100…000b
短型整数	32	100…000b
长型整数	64	100…000b
BCD	80	1 1111111 1111 1111 XX…XXb
短型实数	32	1 11…11 10…00b
长型实数	64	1 11…11 10…00b
临时实数	80	1 11…11 100…00b

注意：X 表示不关心

1.2.10　异常

异常表示在当前操作期间已经检测到了某种类型的错误。例如，当你发动汽车时，如果车门还敞开或座位保险带没有系紧，就会响笛。两种情况都表示检测到了异常。

有许多可能的异常。例如上面所说的精度的丢失，取决于机器的状态，精度的丢失可能被当作异常来处理。我们也提到过信号的 NaN。对一信号的 NaN 的操作将产生别的异常。有许多其它可能的机器的异常。在第 5 章中将讨论异常条件的详细情况以及如何进行这些处理。

第 2 章　机器状态和存储器寻址

第 1 章叙述了 80386 和 80387 能识别的数据类型，以及在存储器中如何存储这些数据的。80386 和 80387 的机器指令直接对这些基本的数据类型进行操作。每条机器指令指定了要执行的操作，以及参与操作的输入或输出的数据的位置。输入或输出的数据叫做操作数。

本章叙述如何寻址由 80386 当作机器指令操作数的基本数据类型。操作数可以在 80386 处理器的寄存器中，或者在处理器外的主存储器中，或者在 I/O 存储器中，或者在指令中作为程序立即常数的数据。主存储器提供了一个非常大的存储器操作数的空间，但是访问主存储器比起对寄存器或立即常数的访问要慢得多。更进一步，有些操作要求某些操作数是在寄存器中，而不是在主存储器中。

本章首先叙述应用程序员可用的三种类型的寄存器：用于存储 32 位数的寄存器，用于控制处理器的执行的寄存器，用于寻址内存的段寄存器。接着我们叙述用来构成存储器地址的方法以及在指令组中操作数的二进制编码。然后介绍独立的 I/O 存储空间。最后叙述在 80387 浮点处理器中可以使用的存储浮点类型数据的寄存器，以及控制浮点指令操作的寄存器。

2.1　寄　存　器

应用程序员可用的寄存器组由 16 个寄存器组成，如图 2.1 所示。这些寄存器可以分成三类。以后再叙述的寄存器是系统程序员使用的，第 4 章叙述了支持存储器管理的寄存器，附录 A 叙述了支持程序调试的寄存器。

这三类寄存器叙述为：

(1) 用于算术和逻辑运算的 8 个 32 位的通用寄存器，它们也可用于基址和变址寻址时存放地址。

(2) 两个处理器控制寄存器。

(3) 6 个 16 位的段寄存器，它们用来寻址存储器的段。每个寄存器一次只提供直接访问一个段的内存。内存的段的含义在本章的后面加以说明，而在第 4 章中将加以详细说明。

80386 的寄存器组是先前的 86 系列处理器所用寄存器组的超集。先前的 16 位寄存器被扩展成 80386 上的 32 位寄存器。386 寄存器的名字是由老的 16 位寄存器的名字之前冠以 E 组成。例如，8086 中 16 位的 AX 寄存器扩展成 80386 中的 EAX 寄存器。16 位的 IP 寄存器扩展成 386 中的 EIP 寄存器等等。

2.1.1　通用寄存器

80386 有 8 个 32 位的通用寄存器，用于诸如加减乘除等的算术运算以及形成内存的地址。这在本章后面将会叙述。这 8 个寄存器定名为 EAX、ECX、EDX、EBX、ESP、EBP、ESI 和 EDI。见图 2.1 所示。

这些寄存器的低 16 位可以作为 16 位的寄存器独立访问。并且把它们命名为 AX、CX、DX、BX、SP、BP、SI 和 DI，这些就是 86 系列处理器的以前的成员的 8 个 16 位通用寄存器。这

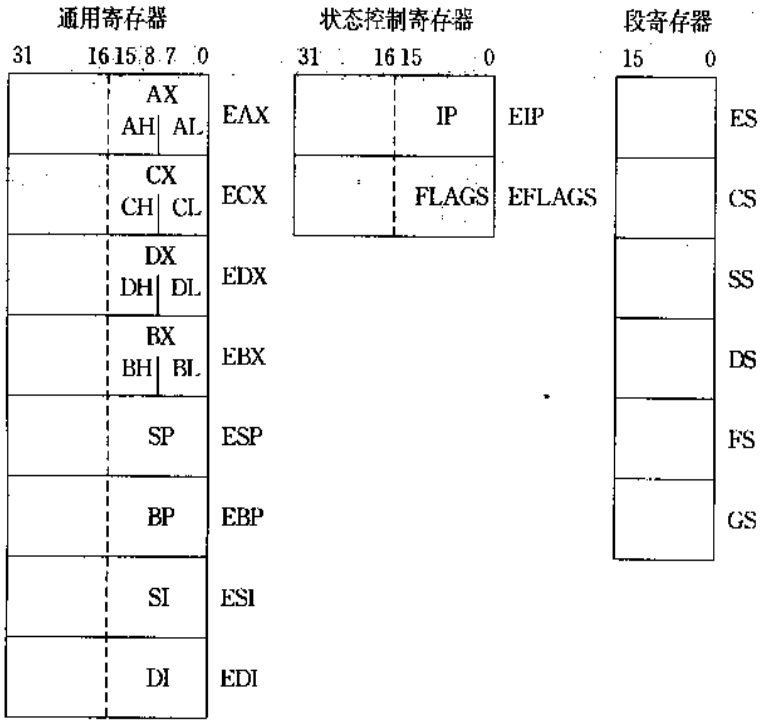


图 2.1 80386 的寄存器

为执行 8086 和 80286 代码提供了兼容的寄存器组。这些内容将在第 7 章中叙述。如果访问这些 16 位的寄存器，32 位通用寄存器的高 16 位不受影响。

AX、BX、CX、DX 四个带 X 的寄存器的高半和低半个寄存器可以作为 8 位的寄存器来独立访问。后缀 H 和 L 被加到 16 位寄存器名字第一个字母之后作为由高半个或低半个寄存器构成的 8 位寄存器的名字。例如 AL 寄存器是 AX 寄存器的低 8 位，并且也是 EAX 寄存器的最低 8 位；AH 寄存器是 AX 寄存器的高 8 位，也是 EAX 寄存器的 8 到 15 位。如果这些 8 位寄存器中的一个被访问，通用寄存器的其它所有位不受影响。在图 2.1 中可以看到 8 位、16 位以及 32 位寄存器交叠的示意。

通用寄存器的特征：可以指定任何通用寄存器的内容作为操作数的地址，以及在形成地址的过程中作如加法或减法这样简单的算术运算。但像串的运算和双精度的乘法和除法操作这类较复杂的操作，必须要从固定的寄存器中取一个或多个操作数。指令需要从专门的寄存器中读取一个或更多的操作数的技术称为特征。80386 用它来支持复杂的操作，这种复杂的操作要求 80386 指令格式中提供的两个以上的操作数。因为对寄存器的引用是隐含在指令中的，所以不必显式指定名字作为操作数。例如，ECX，ESI 和 EDI 寄存器就分别被串操作指令用来放串长度、源地址指针和目标地址指针。双精度乘法指令设计成用 EAX，EDX 寄存器存放双长度的结果。在第 3 章中与这些指令有关的地方将提供详细的例子。除了这些例外之外，一个操作数可以是任何寄存器或存储器地址。80386 操作数的选择是对称的。

2.1.2 处理器控制寄存器

两个寄存器控制 80386 的工作。它们是指令指针寄存器 EIP 和处理器状态和控制标志寄

存器 EFLAGS。

2.1.2.1 指令指针寄存器 EIP

32 位的 EIP 寄存器只起一个作用：指向处理器将要执行的下一条指令。当 80386 执行一个程序时，它取出指令并且执行程序中的每一条指令。即取出由 EIP 所指的指令，使 EIP 增加这条指令的长度，然后执行这条指令。增加的长度使 EIP 寄存器指向内存中指令序列的下一条指令。这样 EIP 总是指向相随的下一条要执行的指令。在第 3 章中叙述的控制转移指令可以改变这样顺序的指令流程。只要向 EIP 寄存器中装入一个新的值即可。16 位的 IP 寄存器包含在 EIP 寄存器的低 16 位中，IP 寄存器提供了用于执行 8086 和 80286 代码的 16 位的指令指针，这将在第 7 章叙述。

2.1.2.2 处理器的状态和控制标志寄存器 EFLAGS

32 位的 EFLAGS 寄存器包含有若干个状态标志和控制标志位。程序可置位控制标志位以控制 80386 的某些功能的运行。算术操作之后处理器自己置位适当的状态位，程序测试这些状态位，以检测特定的条件。图 2.2 展示了 EFLAGS 寄存器中的各个位字段。每个字段在下面会简要的叙述。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
														V	R		N	IO	P	O	D	I	T	S	Z		A		P		C
0	0	0	0	0	0	0	0	0	0	0	0	0	0	M	F	0	T	L		F	F	F	F	F	F	0	F	0	F	1	F

图 2.2 EFLAGS 寄存器详图

在图 2.2 中标为 0 或 1 的那些位是保留的位。保留的这些位必须按所示的用 0 或 1 来装载它，并且在检查 EFLAGS 时将它们忽略。这样能保证未来当 Intel 公司决定要用 EFLAGS 寄存器的这些位定义更多标志时的兼容性。

改变 EFLAGS 寄存器中的位的一个方法是把 EFLAGS 寄存器存储起来，然后再对其存储的映象修改想改的位，再用修改后的值装载 EFLAGS。用这个方法可以保证在将来的处理器上也行得通。这种方法只会修正想要修改的位，其余的位(包括在将来的处理器中定义的那些位)不会改变。

算术状态标志：CF、PF、AF、ZF、SF 和 OF 位是由大多数算术和逻辑指令来置位的。

CF　CF 是进位标志。如果一个算术运算超出最高有效位产生进位或借位，它就置位为 1(CF=1)；否则它被清为 0(CF=0)。此标志对无符号的算术运算提供了一个溢出条件；并支持多精度的算术运算。

PF　PF 是个奇偶标志位。它指示结果低 8 位的奇偶性。若结果的低 8 位中 1 的个数为偶数，PF 为 1；若结果中的 1 的个数为奇数则 PF 被置 0。这被称为奇的奇偶性，因为当数据和 PF 一起其中的 1 的个数始终为奇数。

AF　AF 是辅助进位位。如果位 3 向前有一个进位或借位，AF 被置为 1。它用在 BCD 算术运算中。

ZF　ZF 是零标志。如果结果是零，它被置位。

SF　SF 是符号标志。它被置成结果的最高有效位。在 2 的补码表示中，最高有效位是符号位。

OF　　OF 是 2 的补码的溢出标志。如果算术运算的结果超出了指定位数的 2 的补码所能表示的范围，OF 就被置位。大多数算术指令都会使其置位，例如，如果有一个进位进入到最高有效位，但它并没有产生进位，或者没有进位进入最高有效位，而它向前产生了进位都会使 OF 置位。

选择这一组状态使得只用一套算术指令可以对无符号二进制数、带符号的 2 的补码的二进整数以及 BCD 数进行运算。在对无符号的数进行操作之后由 CF 来检测出界的结果。对带符号的数操作之后可以用 OF 位。而对 BCD 操作之后用 AF 位可以检测是否超界。两数之间大小关系可以通过执行一个减法操作，然后观察

CF	对无符号数
SF XOR OF（SF 和 OF 之间的异或运算）	对带符号整数
AF	对 BCD 数

的状态来决定。

用对 SF 和 OF 作异或运算来判定带符号数的大小关系的方法甚至在减法操作结果发生溢出时也是正确的。

下面的例子展示了这些标志位是如何只用一套的指令来支持无符号数、带符号的 2 的补码整数和 BCD 数的运算的。

第 3 章叙述了每条机器指令是如何影响标志位的。这里我们看 CMP 指令。这条指令用从一个数中减去另一个数并置位相应的状态标志的方法来比较两个数。假定我们比较两个 8 位数。第一个数值是 85h，如果将其看作无符号数，其十进制值为 133。如果将其看作带符号的整数，其十进制值为－123，或者将其低四位看成 BCD 数字，则其十进制值为 5。第二个数值是 49h，当作无符号数为 73，带符号数为 73，BCD 数为 9。如图 2.3 中所示的细节，在 80386 上从 85h 中减去 49h 的结果是 3Ch。并且置位相应的标志位如下所示：

CF 是 0，因为在 b7 没有发生借位；

PF 是 1，因为结果的 8 个位中 1 的个数为偶数；

AF 是 1，因为在位 3 上发生了借位；

ZF 是 0，因为结果为非 0；

SF 是 0，因为最高有效位是 0；

OF 是 1，因为位 6 发生借位，但没有在位 7 上发生。

	386 表示		实际结果		
	十六进制	二进制	无符号	2 的补码	BCD
发生借位	1	1111	1		1
	85h	10000101b	133	－123	5
减	－49h	－01001001b	－73	－73	－9
结果	3Ch	00111100b	60	－196	6
溢出指示器			CF＝0	OF＝0	AF＝1
小于指示器			CF＝0	(SF XOR OF)＝1	AF＝1

图 2.3　状态标志支持多数据类型

在图 2.3 中，由 386 产生的二进制结果在左边示出。右边示出了二进制输入和结果是如何

作为无符号数、带符号2的补码整数或BCD数来解释的。

如果输入是无符号数，产生正确的结果，因为专为无符号数的溢出指示器CF是0。CF也是小于指示器，这样你可以看到，在本例中386正确地检测到133不小于73！

如果输入是带符号的整数，产生的值(−196)太大了，在8位中放不下。386置2的补码溢出指示器OF为1，指示结果溢出。不管其已经溢出，386仍可以正确地指示−123小于73。因为带符号小于指示器是1，即SF异或OF为1。

如果输入是BCD数，由二进制减法产生的值是一非法的十进制数字(C)。这是由BCD溢出指示器AF来指示的。AF还是BCD数的小于指示器，在此它指示5小于9。

CF，ZF，SF和OF的设置取决于在一个操作中所用数据的大小。32位数据的运算把SF设置成位31的结果。如果32位的结果为0，置位ZF，根据借位在位31之内或之外设置CF和OF。16位数据的运算把SF设置成位15的结果，如果16位的结果为0，置位ZF，根据借位在位15之内或之外设置CF和OF。8位数据的运算把SF设置成位7的结果，如果8位的结果为0，置位ZF，根据借位在位7之内或之外设置CF和OF。对于8位数据的标志设置在图2.3已经示出。

PF和AF的设置不取决于所执行的运算的大小。PF总是按照低8位的奇偶性而设置的。如果结果的位3上发生进位或借位，AF被置位为1。

处理器控制标志：TF、IF、DF、IOPL、NT、RF和VM位可以由程序来设置以便控制80386处理器的运行。大多数标志支持在本章后面叙述的80386的特征。本节总结了这些标志。是对包括有如何使用这些标志细述的章节的提前小结。大多数处理器控制标志可以由程序来访问。但有三个例外，它们是VM标志，IOPL字段和IF标志。第4章将叙述如何保护这三个控制标志。

TF　捕获允许标志控制产生单步中断来支持程序的调试(附录A)。当TF=1时，在每条指令执行结束处将发生单步中断。

IF　中断允许标志允许识别处理器引脚上发来的外部中断(第5章)。如果IF=1，接受外部中断，如果IF=0挂起外部中断。

DF　方向标志决定串操作指令在每步之后其变址寄存器是后加(DF=0)或者是后减(DF=1)。见第3章关于串操作指令的细述。

IOPL　特权级字段是个两位宽的字段。它支持第4章中叙述的保护方式。IOPL字段指定了要求执行I/O指令的特权级。如果当前的特权级别在数值上小于或等于IOPL，I/O指令可以执行，否则发生一个保护异常。

NT　嵌套任务位控制IRET指令的运行(第3章)。如果NT=0，用栈中保存的值恢复EFLAGS、CS和EIP执行常规的从中断返回的动作。如果NT=1，中断返回用一任务转换代替上述过程(第4章)。

RF　重启动标志控制着调试故障(附录A)是接受(RF=0)，或者是被忽略(RF=1)。在成功地完成每一条指令以后处理器会把RF清成0。而当接收到一个非调试故障的故障信号时，处理器把RF置成1。

VM　是虚拟8086方式位。如果该位置为1，处理器将在虚拟的8086方式下工作(第7章)，如果清零，处理器将工作在一般的保护方式下(第4章)。

运行在任何特权级下的程序都可以置位或清除RF，NT，DF和TF这些标志位。只有在特权级0下执行的程序才能改变VM及IOPL字段。IF位只能由具有I/O特权的程序所改变

(第4章)。

RF和VM位只能由IRET指令和任务切换来置位或清零。对其它的控制位则可由POPF指令来置位。

2.1.3 段寄存器

80386有六个16位的寻址内存的段的段寄存器。这些寄存器分别定名为ES、CS、SS、DS、FS和GS。如图2.1中所示。FS和GS是386上新增加的段寄存器。

要在一个给定的段中访问数据，一个程序必须用标识那个段的专门的值装载其中的一个段寄存器。这个专门的值叫做选择子。对于选择子，本章下面将有简要叙述。在第4章中将要详细叙述。六个段寄存器使一个程序可以同时访问多达六个段。当程序在一个段中完成并开始进入另外一个新段中去执行时，可以把新段的选择子装载到老段的段寄存器中。

2.2 存储器寻址的概念

80386使用叫做段的存储器寻址技术。它把存储器空间分成一个或多个叫做段的线性区域。一个存储器的地址由两个部分组成：一是段的部分，它标识所容纳的段；另一个是偏移部分，它指出在该段内的以字节计的偏移量。对于每个存储器的引用必须指定段和偏移两个部分。本章叙述了引用一个存储器操作数的每条指令中所用的基本的存储器寻址机制，并详细叙述指定段和偏移的各种选择方法。

分段寻址所提供的许多存储器结构的能力对于应用程序员来说是可见的。因此此时讨论这些方面是适当的。

关键的问题是如何把诸如代码的过程、数据区和程序的堆栈等的程序单元安排在一个或多个段中。

第1章介绍了物理内存是字节的一维(线性)数组，以及叙述了占据一个字节以上的数据类型在存储器中是如何放置的。在本章中我们看到，存储器被分成了段。从而对内存空间提供了第二个维。第4章讨论了以操作系统观点来看内存的寻址。叙述段保护的内存管理机制以及把它们重定位到一维的物理内存空间中去的问题。在第4章中你将看到，本章中所说的二维的虚地址如何转换成一维的物理内存地址的。

2.2.1 两部分的寻址

由于分段的原因，在80386上一个内存的地址有两个部分，一个段部分和一个偏移部分。具有内存操作数的指令必须指定这两个部分。段的部分是一个16位的段选择子，段选择子有一个14位的字段，用于识别16384种可能的段。32位的地址偏移部分给出了一个字节在所在段中的偏移。段内的连续的字节地址可以从0到该段的上限地址。因为地址偏移量为32位，可以指定最大的偏移量是4G字节，这就是一个段的最大长度。

2.2.2 表示法

本章中用来表示内存地址的表示法让段在左边的位置，偏移量在右边的位置，中间用冒号(:)把两部分分隔开。段的部分可能是段的名字或者段寄存器。如果给出段寄存器名，在那个寄存器中包含的段选择子标识了那个段。偏移量是以数或者方括号中的寄存器名给出。偏移

量也可以用表达式表示，它由涉及的寄存器名的表达式、比例因子和常数构成。这些反映了本章后面将讲到的寻址方式。

例如，在一个名叫 NEWDATA 的段中偏移为 1234 的地址将用表示法 NEWDATA：[1234]来表示。如果 ES 包含了 NEWDATA 段的选择子，EAX 通用寄存器包含了值 1234，用表示法 ES：[EAX]可以指定同一个地址。这种表示法是汇编程序对内存地址的表示法的一个子集。在第 6 章中我们定义并使用这种汇编程序的表示法。

图 2.4 说明了如何用两部分地址去访问内存。在图中有两个段，段 A 和段 B。段 A 包含有 11 个字节，因此在 A 段中的段偏移是从 0 到 10。段 B 包含 17 个字节，因此它的段偏移从 0 到 16。在地址 A：[5]上的字节即在 A 段内偏移 5 处，包含值 77。在地址 B：[12]上的字节，即在 B 段内偏移 12 处，包含值 88。

2.3 内存寻址机制

使用一个内存操作数的指令必须指定一个段的部分和一个偏移部分。大多数的程序倾向于同时只使用不多的几个段。但是在这些段中产生许多不同的偏移地址。80386 的寻址机制最适合这样的程序特性。

2.3.1 段部分：段寄存器

本章前面介绍的 6 个段寄存器特别用来放置段选择子作为地址的段部分。这些寄存器在一程序的任一点上可提供多到 6 个段的访问。

每个存储器的引用或是隐含地或是显式地或是默认地指定了段寄存器。段寄存器中包含有地址的段的部分的选择子。代码段的引用总是用 CS 段寄存器，堆栈段的引用总是用 SS 段寄存器。而某些字串处理指令总是使用 ES 段寄存器作为目标操作数的段寄存器。6 个段寄存器中的任何一个都可以被用来作其它所有的数据的引用。例如，图 2.4 中段 B 的选择子是在 FS 寄存器中，存储器引用 FS：[12]将访问存储在那个段中的偏移量为 12 的那个数据。即地址 B：[12]。如果 B 的选择子被从 FS 寄存器移到 DS 寄存器中，用地址 DS：[12]可以访问同一个数据。

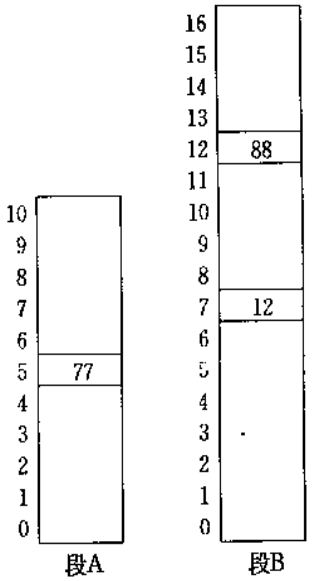


图 2.4 两部分的寻址

由 CS 寄存器在任何给定的时间上寻址的段叫做当前代码段。EIP 寄存器包含了由 CS 寄存器寻址的段中下一次要执行的指令的偏移地址，因此下一条要执行的指令的地址是 CS：[EIP]。如果当前代码段包含有指令和数据，数据就可以用 CS 寄存器作为地址的段的部分来引用。例如，在偏移量为 8 加上 EAX 寄存器中的值的一小块数据，可以从同一个段内的指令中用 CS：[EAX+8]来寻址。

所有代码段的引用使用 CS 寄存器，因此在任何时候只有一个代码段是可以寻址的。第 3 章中叙述的段间控制转移指令，可以被用来把一个新的值装入 CS 和 EIP 寄存器中，从而把执行的位置改变到别的代码段中。这就提供了一种有效的机制，使存储在不同的段中的过程之间传递控制。例如，应用程序的代码典型地是在与操作系统不同的代码段中的。这样，这些段间

控制传递的机制为应用程序调用操作系统服务提供了一个有效的方法。

由 SS 寻址的段叫做当前堆栈段。如 PUSH、POP、CALL 和 RETURN 这些堆栈的操作，使用由 SS 寻址的段内的一个程序的堆栈。该堆栈的顶是由包含在通用寄存器 ESP 中的偏移地址所指向。ESP 是专为这种用途而保留的。程序堆栈的顶的地址是 SS：[ESP]，程序堆栈的更详细的情节在本章后面要叙述。

DS、ES、FS 和 GS 寄存器对于寻址程序要求的一般的数据段是可以利用的。DS 寄存器是主要的数据段寄存器。因为它对于引用除堆栈外的数据段是一个默认的段寄存器。正如你将在本章后面看到的，引用由 CS、SS、ES、FS 和 GS 寻址的段中数据的指令比那些由 DS 寻址的指令要长一个字节。这使得引用非 DS 的段在程序存储量和执行时间上开销稍大一些，但也大不了多少。一般地说，按下述所做是一个好办法。即安排使 DS 寻址最经常引用的数据段，而用 ES、FS 和 GS 寄存器去寻址那些不太经常引用的数据段。

图 2.5 说明了段寄存器的使用情况。CS 指向当前代码段。EIP 给出在该段中下一条指令的偏移量。SS 指向堆栈段，而 ESP 标识了在该段中程序堆栈的顶。如下节将要说明的，堆栈的内存是由 ESP 向地址低的方向移动分配的。因此在 ESP 之上的堆栈段的部分已经分配给堆栈，而低于 ESP 部分的内存对于以后需要分配时是可用的空间。DS 指向第一个数据段，取名为 DATA1。ES，FS 和 GS 对于寻址其它的数据段是可以利用的。可能指向 DATA2，DATA3 或 DATA4 等，如虚线所示

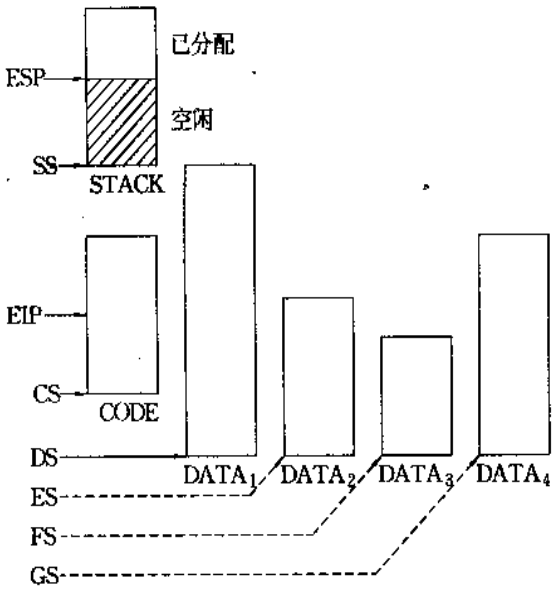


图 2.5 段寄存器的用法

2.3.2 偏移部分：地址方式

前节叙述了指定一个地址段部分的机制。因为每个内存的引用包含了一个段的部分和一偏移部分。我们现在叙述如何指定偏移部分。

80386 为构成一个地址的偏移部分提供了灵活的机制。引用一个存储器操作数的每条指令规定了计算偏移量的方法。这种规定法叫做指令的寻址方式。80386 上的寻址方式规定将其

相加构成偏移的成分多达三个：一个基地址寄存器，一个比例因子为1,2,4或8的变址寄存器，及一个常数位移量。可以由这三个成分相加得到偏移地址。

8个通用寄存器中的任意一个可以用作基地址寄存器，或基地址成分可以被忽略。除了堆栈指针寄存器外的7个通用寄存器中的任意一个可以被用作变址寄存器，或者变址成分可以被忽略。如果指定了一个变址寄存器，包含在该寄存器中的值可以按1,2,4或8的比例，在加到偏移量上之前换算。可以指定一个8位或32位的常数位移量，或者常数位移量也可以忽略。

这些**基地址**+(**变址**×**比例因子**)+**位移量**的寻址方式提供一强有力而又灵活的寻址机制。这种机制满足了由高级语言支持的数据结构的需要。下一节叙述诸如记录、数组甚至记录的数组是如何使用这些寻址方式的。

如果存储器操作数是对齐的，使用它就比较快。如果一操作数的偏移量是其大小的倍数，该操作数是对齐的。例如，如果一个字的偏移量是2的倍数，这个字是对齐的。不对齐的操作数也能访问，但时间上可能长一些。我们建议所有的操作数都应该对齐。

对于数据引用，其默认的段寄存器取决于所选择的基地址寄存器。如果基地址寄存器是ESP或者EBP，则默认的段寄存器从通常的DS改为SS。因为ESP和EBP寄存器是被设计来与堆栈一起来使用的。对于别的基地址寄存器的选择，包括无基地址寄存器，DS仍然是其默认的段寄存器。访问由非默认的段寄存器寻址的一个段内的数据要求使用一额外的指令字节来指定所要的段寄存器。用EBP作为一个变址寄存器(ESP不能成为变址寄存器)，不影响默认段寄存器的选择。默认段寄存器的选择只由所选的基地址寄存器所影响。

表2.1列出了种种可以利用的寻址方式。关于使用寻址方式的例子将在下一节中给出。

表2.1　80386寻址模式

基地址+(变址×比例因子)+位移量

$$\left\{\begin{array}{l}\text{无}\\ \text{EAX}\\ \text{ECX}\\ \text{EDX}\\ \text{EBX}\\ \text{ESP}\\ \text{EBP}\\ \text{ESI}\\ \text{EDI}\end{array}\right\}+\left\{\begin{array}{l}\text{无}\\ \text{EAX}\\ \text{ECX}\\ \text{EDX}\\ \text{EBX}\\ \text{---}\\ \text{EBP}\\ \text{ESI}\\ \text{EDI}\end{array}\right\}\times\left\{\begin{array}{l}1\\ 2\\ 4\\ 8\end{array}\right\}+\left\{\begin{array}{l}\text{无}\\ \text{8位}\\ \text{32位}\end{array}\right\}$$

注　(1) 如果ESP,EBP是基地址寄存器，则SS是默认的段寄存器。

(2) 虚线表示ESP不能被用作变址寄存器。

2.3.3　程序堆栈

程序堆栈支持LIFO(后进先出)的分配秩序。这种分配法很适合于表达式的结果和子程序所要求的嵌套的存储次序。两个基本的堆栈操作是PUSH和POP。PUSH操作把新的元素加到堆栈上去；而POP操作是把新近压入的元素移走。一个PUSH操作之后跟随一个POP操作使堆栈不变。

PUSH和POP操作支持使用堆栈存储临时值，例如，一个复杂计算的中间部分的值的存储。当这些中间部分的值被计算出来，结果可以压入堆栈。为了计算最后表达式的值，中间计算的结果从堆栈中弹出，并组合构成最后的结果。由堆栈提供的嵌套存储和子表达式计算的固

有的嵌套很相配。

由堆栈提供的嵌套存储也支持有效而灵活的子程序的调用和返回机制。这种机制很自然地处理了嵌套的过程。一条过程调用指令 CALL 把一个返回地址压入堆栈，在栈中保存着直到相配对的 RET 指令把它弹出为止。对一嵌套的子程序的调用把其返回的地址压入堆栈，这样隐藏一个外部过程的地址。从一嵌套的子程序的返回把返回地址从堆栈弹出，因为外部过程的返回地址被暴露在栈的顶部。下一节叙述使用程序堆栈支持完成子程序调用的机制。

堆栈在由 SS 寄存器寻址的存储器的段中。ESP 寄存器包含有在该段中程序堆栈的顶部的偏移量。两部分的地址 SS：[ESP]指向程序堆栈的当前的顶部。

2.3.3.1 在堆栈上的临时存储

在 80386 上，堆栈是往低地址方向增长的，并且每个堆栈元素为四字节宽(32 位)。为了把一个元素压入 80386 的堆栈，首先 ESP 寄存器要减去 4，然后把新的元素写到由 SS：ESP 寻址的存储位置上。为了把一个元素弹出堆栈，读取由 SS：[ESP]寻址的内存位置的内容恢复该元素，然后 ESP 寄存器加上 4。在任何时候当前栈顶是由 SS：[ESP]寻址的。而堆栈内的元素，由 SS：[ESP]开始的正的偏移量来寻址。第二个堆栈元素在地址 SS：[ESP]+4 处。第三个堆栈元素在 SS：[ESP]+8 处等等。如果堆栈是对齐的，堆栈上的操作数可以很快地被引用。因为堆栈元素的大小为 4 字节，要确保 ESP 寄存器总是包含 4 的倍数，这一点是很容易做到的。

图 2.6 说明 PUSH 和 POP 是如何对 80386 程序堆栈起作用的。左边的堆栈图展示了三个元素压入堆栈段中偏移地址为 FCH、F8H 和 F4H 之后的堆栈，该堆栈的大小为 100h 个字节。ESP 保持了 F4H 值，它是栈顶元素的偏移地址。图的底部是栈顶。因为堆栈是朝低地址方向增长的，图底部是低地址。右边的堆栈图展示了压入值 123456H 之后的堆栈。ESP 现在保持有值 F0H，它比原有值小 4。原栈顶元素现在成第二个栈顶元素。并且可以在 SS:[ESP+4]上寻到。一个 POP 操作将恢复堆栈到左边堆栈所示的状态。它只把 ESP 寄存器中的值调整了 4。

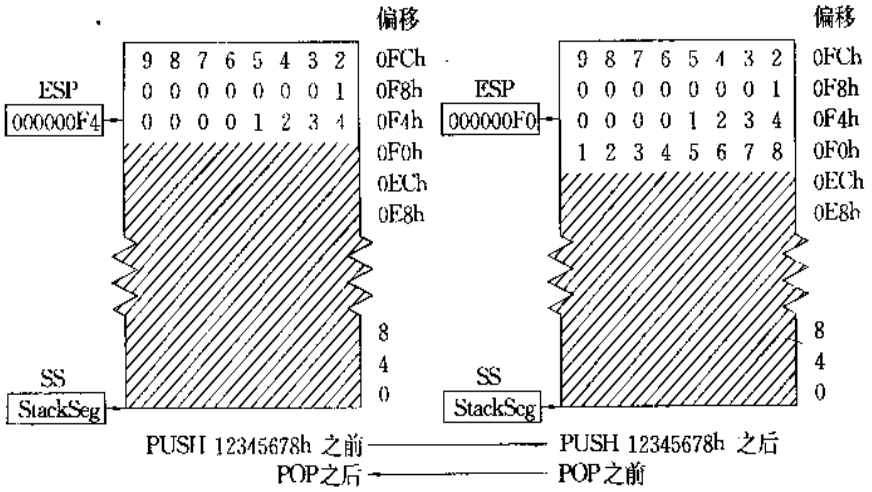


图 2.6 堆栈 PUSH 和 POP 的操作

2.3.3.2 子程序和堆栈

80386 程序堆栈支持一有效的子程序调用和返回的机制。CALL 指令把一个返回地址(跟在 CALL 指令之后的指令的地址)压入程序堆栈中去，并且跳转到子程序的开始地址去。RET 指令把返回地址从堆栈中弹出，然后跳转到与相配的 CALL 指令之后的那里重新开始执行。

程序堆栈也用来把参数传递给子程序，以及存储子程序的局部变量。由于使用堆栈，80386 有效地支持嵌套的子程序，包括递归的和可重入的子程序。

最适当的办法是用通用寄存器存储开头的少数几个参数和最经常使用的局部变量。因为寄存器比存储器访问得更快。这种办法能加快程序执行的速度。堆栈可以用来存储其余的参数和局部变量。

基本的子程序的连接：最简单的子程序的连接支持在程序堆栈上传递参数和分配局部变量的子程序的调用。一个子程序的调用分三步进行：

(1) 把参数压入程序堆栈。

(2) 用 CALL 指令把返回的地址压入堆栈并跳转到子程序去。

(3) 从 ESP 寄存器中减去一个常数，在堆栈上为局部变量保留出空间。

在子程序中，可以用相对于 ESP 寄存器的常数位移量来寻址参数和局部变量。当局部变量和参数的数目不超出 127 字节时，可以使用 ESP 作为基地址加一个带符号的 8 位位移量的寻址方式。否则必须使用 32 位的位移量。在子程序中，无论作为临时值压入或者弹出堆栈或者作为参数压栈传到别的子程序去，ESP 都要发生改变。在改变时要保持对上述的偏移量的正确跟踪。对于一个语言编译器来说这种跟踪是很容易的。但是，当书写一个汇编程序时可能会发生混淆！一个替换的办法是使用 EBP 来作为一个稳定的基地址寄存器，使其指向为活动的过程而分配的堆栈区域。这样就可使变量和参数按相对于 EBP 来说是固定的偏移来寻址。但要避免使用 EBP 当一般的通用寄存器来使用。

ESP 和 EBP 寄存器是被设置来有效地支持堆栈寻址的，因为无论是用 ESP 或 EBP 作为基地址寄存器都选择 SS 作为默认的段寄存器。这样允许 ESP 和 EBP 为当前过程寻址堆栈区域时不必再加一个字节来显式地指定堆栈段。

图 2.7 说明了调用一个带两个参数和 32 个字节局部变量的子程序后程序堆栈的状态。子程序和调用子程序的过程在同一个段中，因此用一个段内调用 CALL(第 3 章)把 4 字节的返回地址压入堆栈。压入堆栈的两个参数占 8 个字节。返回地址占 4 个字节。和 32 字节的局部变量总共 44 字节。

在子程序中，局部变量是由 SS：[ESP+n]来寻址的。这里 n 是 0～31。第一个参数在地址 SS：[ESP+40]处。第二个参数是在地址 SS：[ESP+36]处。

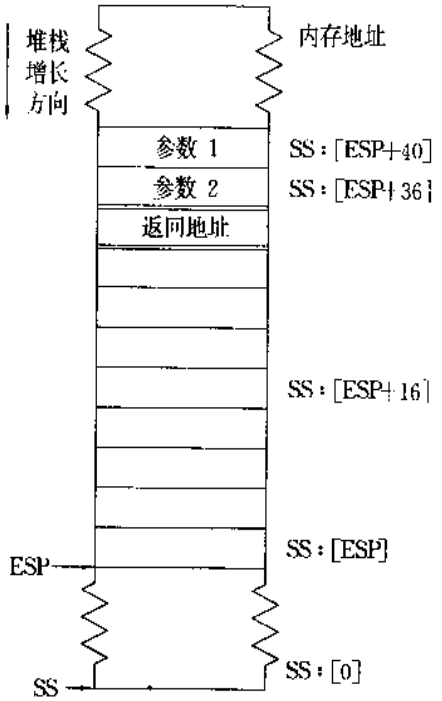


图 2.7 在程序堆栈中存储的参数、返回地址和局部变量

为了从子程序返回，要求与 CALL 过程的三个步骤相反。

(1) 加一个值到 ESP 上去把局部变量弹出程序堆栈。

(2) 把返回地址弹出堆栈，从子程序返回，并且跳转到弹出的地址上去。

(3) 加一个值到 ESP 上去以便把参数弹出堆栈。

在第 3 章中叙述的 RET 指令的形式兼有后两步骤。

2.3.4 指针数据类型

大多数数据类型都在第 1 章中讲过了。我们把指针数据类型推迟到现在介绍，是因为我们首先要介绍一些基本的寻址概念。指针数据类型即包含有各种数据的地址的一个值。指针对于组建象列表和树这样的复杂的数据结构是非常有用的。当一个程序执行时，这些数据结构在结构上有很大变化。在一个列表或树结构中的每个元素，包含有一个或更多的指向别的元素的指针，这样只要简单地在这些指针上存储别的元素的地址就可以把各元素连接起来或解开连接。其它如数组和记录这样的结构更适合于在结构上不变化的数据结构，因为它们在存储用法和执行时间上比由指针来连接的结构通常更有效。

80386 支持两种类型的指针：一是 48 位全指针，它包含全部两个部分的地址，当要用指针来连接不同段中的数据时就要用 48 位的指针。另一个是 32 位的指针，它只包含地址的偏移部分。只是偏移量的指针比 48 位的指针更为有效。但只当所有要连接的元素存储在同一个段中时才可以使用它。

在图 2.8 中所示的一个 48 位的指针包含了一个地址的两个部分。偏移部分在低 32 位，而段选择子是在高 16 位。为了用 48 位的指针寻址数据，必须把两部分装到寄存器中。在高 16 位的段选择子是装在某个段寄存器中的。低 32 位中的偏移量部分是装在某个通用寄存器中的，在此时它可以作为基地址寄存器。第 3 章将叙述 LDS、LES、LFS 和 LGS 指令。这些指令将在一条指令中把那些 48 位的指针装入段寄存器和通用寄存器中去。

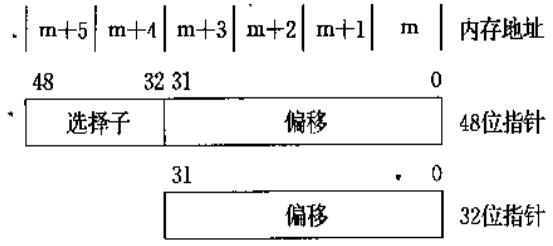


图 2.8 48 位和 32 的指针

如果用指针存储的所有的地址的段的部分都相同，一个最有用而恰当的办法是把这个段的选择子保持在专为这种目的而用的一个段寄存器中。只存储 32 位指针的偏移部分。这也在图 2.8 中示出了。为了用 32 位的指针寻址数据，这需把偏移部分装入作为基地址寄存器的某个通用寄存器中去。段的部分就用专门用于保持公共的段选择子的段寄存器。

在存储的用法和执行的时间上，32 位偏移指针比起 48 位指针更为有效。对于一个应用程序，如果所有的数据，包括程序堆栈是存储在同一个段中的，可以使用 32 位的指针。在这种方式中，SS、DS 和 ES 段寄存器装载公共段的选择子，这样公共段包含了所有的数据和程序堆栈。这样所有存储的引用可以用默认的数据段寄存器。注意，在这种方式中，虽然只有一个段，存储器地址仍应认为有一个段部分，只是隐含地用一个段包含了一切而已。

2.3.5 寻址方式和数据结构

80386 的基地址＋变址＋位移量的寻址方式直接支持高级语言的寻址需要。标量变量、记录数组甚至记录的数组和数组的记录这些都直接被 80386 的寻址方式所支持。如由 FOR-

TRAN 所要求的，这些数据结构可作为 Static 存储来分配。如由 Pascal 或者 C 所要求的，这些数据结构可以动态地分配在一个程序堆栈上(相对于 SS：ESP 来引用)。或者堆 Heap(通过指针直接地引用)。基地址和变址寄存器为寻址方式提供两个动态的成分，而位移量提供静态的成分。用在一数据段中的一个常数位移量可以简单地寻址静态分配的数据。用一相对于 ESP 或者 EBP 寄存器的常数位移量可以寻址分配在堆栈中的数据。表 2.2 表示了语言的需要和 80386 支持的寻址方式之间的关系，并且为每种寻址方式给出汇编语言表示的例子。

在语言翻译器中为了简化寄存器的分配，可以用 8 个通用寄存器中的一个作基地址寄存器，以及用除 ESP 外的一个作为比例变址寄存器。变址寄存器的值可以直接地被使用(比例因子为 1)或者按 2、4 或 8 的倍率用于 16 位(2 字节)、32 位(4 字节)、64 位(8 字节)变量变址，而不需要位移指令或使用一额外的寄存器。

表 2.2　数据结构和 80386 寻址方式

存储类型	结构类型	地址方式分类	例子
静态	标量的	位移	DS：[1000]
	数组	变址＋位移	DS：[ESI×4＋1004]
	记录	位移	DS：[1234h]
	记录的数组	变址＋位移	DS：[EDX×8＋1000h]
	数组的记录	变址＋位移	DS：[EBX×2＋1048h]
堆栈	标量的	基地址＋位移	SS：[ESP＋24]
	数组	基地址＋变址＋位移	SS：[ESP＋ESI×4＋120]
	记录	基地址＋位移	SS：[ESP＋28]
	记录的数组	基地址＋变址＋位移	SS：[EBP＋EDX×8－256]
	数组的记录	基地址＋变址＋位移	SS：[ESP＋EDI×2＋1000h]
堆	标量的	基地址	ES：[EBX]
	数组	基地址＋变址	FS：[EBX＋ESI×4]
	记录	基地址＋位移	DS：[EAX＋12]
	记录的数组	基地址＋变址＋位移	ES：[EBX＋EAX×8＋6]
	数组的记录	基地址＋变址＋位移	DS：[EAX＋EDI×2＋1800h]

2.3.6　分段的策略

80386 支持宽范围的分段策略，这种策略可以把程序单元分配存储在一个或多个存储器的段中。一个程序单元是一个独立的代码过程或者数据的区域，或者程序堆栈。每个程序单元应尽可能包含在一个段内，以求得在重定位、共享和保护各个单元方面有最大的灵活性。而在另一个极端上，整个应用程序的代码、数据和堆栈可以存储在单一的一个段中。

有两个关键特征使分段在寻址、保护和共享代码过程、数据区域和程序堆栈方面成为强有力的机制：

(1) 一个段可以是任意的大小。

(2) 第 4 章中将叙述的存储器管理机制，一个段是作为不可分的一个单元来重定位、共享和保护的。而段的大小是无所谓的。

由于段的大小可变，任何程序可以被接纳在单一的一个段中。在那里，它可由操作系统作为单一的单元来重定位和保护。一个 20 字节小的过程可以存储在一个段中，大到 100M 字节

的数据数组也可以存储在单一的一个段中。而一整个应用程序的所有代码和数据也可以存储在一个段中。

图 2.9 和 2.10 说明了把程序单元存储在一个段中的两种策略。用于存储数据的各种段可以按两种模式来组织。图 2.10 的模式只使用一个段来存储数据，因此被称为单段模式(尽管存在保存代码的第二个段)。图 2.9 的模式为在一个应用程序中把数据存储在多个段中。由此所要求的指针的大小也分成两种模式。单段存储模式可以用 32 位的指针，因为所有的数据和堆栈存储在同一个段中。多段存储必须使用 48 位的指针。因为数据存储在几个不同的段中，所以每个数据指针必须显式指定指向何段。

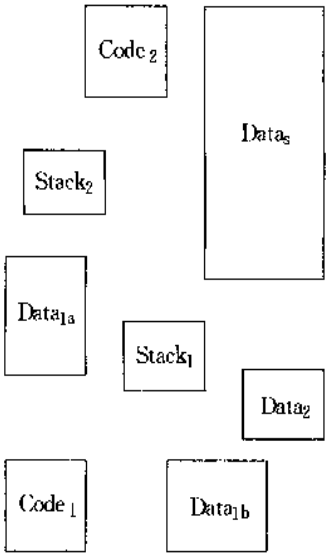


图 2.9　多段的分段策略

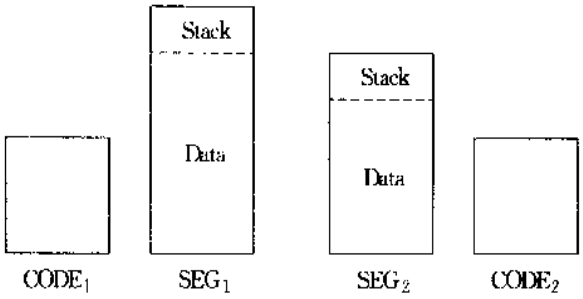


图 2.10　单段的分段策略

图 2.9 展示了对于两个不同的应用程序如何把代码、堆栈和数据存储在分开的段中。应用程序 1 的代码是包含在名字为 Code1 的段中。它的堆栈是在名字为 Stack1 的段中。而其数据在 Data1a 和 Data1b 中。它与应用程序 2 共享一个叫 Data 的数据段。应用程序 2 的代码、堆栈和数据分别为 Code2、Stack2 和 Data2。

在此多段策略中，两个应用程序存储在 8 个不同的段中。有一个段由两个应用程序共享。每个段可以有适当的访问限制。例如，两个代码段 Code1 和 Code2 可被限制成只可执行的访问权限。共享数据段 Data 对一个或两个应用程序可被限制成只读，以防止对其数据未授权的修改。因为有多个数据段，必须使用 48 位的指针。

可在下列两者之间进行权衡：把每个程序单元放到分开的段中去以求得在重定位和保护上的灵活性；或者把一个应用程序的所有的数据和堆栈存储在单一的一个段中，这样就可以只使用 32 位指针使工作更有效。单段策略也支持由大多数小型计算机和大型计算给出的应用模式(这些计算机没有分段机构)，从而支持了应用程序在这些环境之间的易移植性。

虽然在单个的应用中没有使用分段这种模式，操作系统软件还是可以使用分段去分隔和保护每个应用程序不被多任务环境中别的应用程序所破坏。在这种模式中每个应用程序以两个段来存储。一个包含应用程序的所有的代码。而另一个段包含有所有的数据和堆栈。操作系统存储在两个或更多的段中，并且把段安排得每个应用程序和所有别的应用程序互相隔离，

因此操作系统保护了所有的应用程序。

图 2.10 说明了单段分段策略。CODE1 保存应用程序 1 的代码，SEG1 保存了应用程序 1 的堆栈和数据，CODE2 保存应用程序 2 的代码，SEG2 保存了应用程序 2 的堆栈和数据。因为每个应用程序只有一个数据段，每个应用程序可以使用比较有效的偏移量仅有 32 位的指针。使用 32 位指针的一个结果是两个应用程序不再能象图 2.9 中所做的那样共享一个数据段了。而数据的公共段必须要允许读和写的访问操作，对于只读数据就不能如多段模式中那样具有保护作用。

上述的两种分段策略各有其优点和缺点。在许多情况下，操作系统软件决定使用一种模式或另一种模式。如果有一选择的话，我们建议对大多数应用程序应使用单段模式，因为它本来就有着简单性。单段模式由于它使用了简单的 32 位的指针，因而提供了很大的有效性。少数系统要求多段模式提供的共享和保护的灵活性，但是由简单的单段模式提供的服务足够大多数应用程序的使用。

2.4 指令的编码

本节叙述指令的编码，着重叙述对操作数的编码。首先在高级的水平上叙述通用指令的编码，然后给出立即数、寄存器和存储器操作数详细的位的编码形式。

在本节中，你不必要理解 80386 的编程，甚至在汇编语言级的也不必理解。前节资料通过用汇编语言助记符来叙述指令的操作数。本节进入更低的级别，面对的是 80386 机器指令中用于表示那些操作数助记符的实际的位的形式。如果你不需要了解这些详细信息，可以翻到本章后面 I/O 存储空间部分去看。

本章介绍的材料主要是针对如汇编程序、编译程序和调试程序这样的软件翻译器的开发者的。这些软件翻译器需要知道指令的编码。实际上，翻译器的主要功能是对程序员隐藏指令编码的细节。

2.4.1 指令的字段

图 2.11 示出了 80386 指令的一般格式。每条指令可以多到 5 个字段。左边的字段存储在较低的地址上。

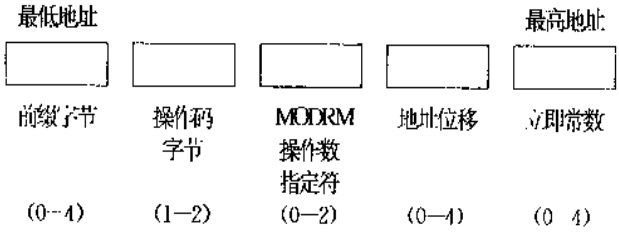


图 2.11 通用指令格式

前缀字节：通常在一条指令前没有前缀字节，但也可以指定多达 4 个字节的前缀字节。顾名思义，如果出现前缀，它们是一指令开头的几个字节。前缀只是被用来修饰和解释随后的指令，不会影响别的任何指令。前缀是用来指定不常用的指令参数，这样可以使常规情况下的指

令更为紧凑。指令前缀的一个例子是本章后面叙述的段寄存器跨越。它是被用来在默认的段寄存器不适合时为一存储器操作数指定段寄存器。其它的例子是第3章中要叙述的LOCK和REP前缀。

操作码字节：操作码字节直接跟随在任何前缀字节之后。每条指令中至少必须出现一个操作码字节。某些指令要两个操作码字节。特殊的一字节指令形式由一个操作码带三位的寄存器指定符组成(见图2.13)。

MODRM操作数指定符：如果存在MODRM操作数指定符则它是跟在操作码字节之后的。操作码决定了本指定符是否存在，并且本字段指定了两个操作数或者一个操作数加上额外的操作码的位。本字段可以指定一个寄存器操作数，一个存储器操作数、两个寄存器操作数，或一个寄存器操作数及一个存储器操作数。MODRM字段可以是一个字节或两个字节长。

地址位移量：如果有地址位移量，则它跟在MODRM字段之后，MODRM字段的子字段MOD表示有位移量及它的长度。只有存储器操作数才存在位移量。

立即常数：如果有立即常数，它是指令中最后一个字段。也就是说，立即常数存储在指令中地址最高的字节中。操作码决定立即常数是否存在，并定义了它的长度。本字段中最多可达4字节的立即数。

2.4.2 立即常数

立即常数这种操作数是最简单的操作数指定符类型。在这里操作数的值直接在指令中给出。在立即常数中的位数取决于指令中规定的操作数的大小以及操作码。如果存在立即常数，它总是指令中最后一个字段。在操作码字段和地址方式字段之后才出现。

图2.12说明了8位、16位或32位操作数各种形式的立即常数大小。常数可以是8位、16

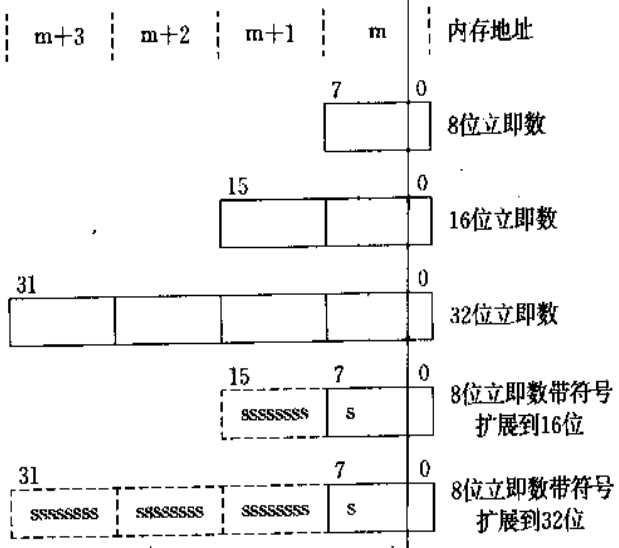


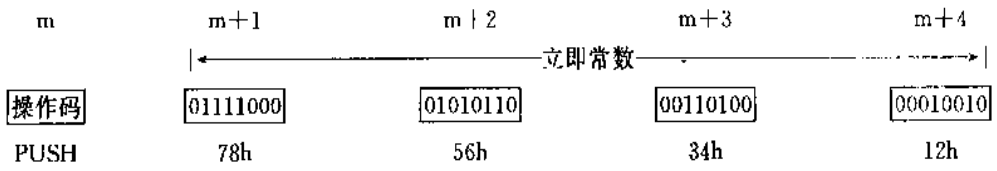
图2.12 立即操作数的编码

位或32位。16位和32位的立即常数的形式其低位字节存储在低地址，其高位字节存储在高地址。可以以三种形式来表示带符号或不带符号的字节、字和双字。另外对于16位和32位的操作数中一个小的值，可用一个带符号扩展的8位的值来表示，以节省空间。在这种情况下，直接在指令中给出一个8位的常数，用在高8位或24位上复制符号位位7的办法可以将其扩展

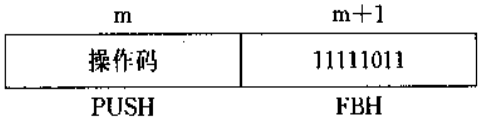
到 16 位或 32 位。这样做，对于具有值的范围为－128 至 127 的带符号的字和双字，对于值范围为 0 至 127 或 FF80 至 FFFFH 的无符号字或值范围为 0 至 127 或 FFFFFF80H 至 FFFFFFFFH 的无符号双字可以节省空间。

例子：把一个立即数压入堆栈的指令使用一种指令格式，它具有一个字节的操作码和 1 或 4 字节的立即数据。在此这条指令是一个很好的例子，因为它只有一个操作数，即一个立即常数。在本例中，存储器地址从顶开始表示为 m，m＋1，…，m＋4。在指令位之下给出其解释。注意，操作码是在最低地址的字节上。而常数存储用在低地址上存放低位字节的方式。

PUSH 12345678h 将被编码为：



PUSH －5 将被编码为：



注意，在最后一个例子中使用了一个短的一字节的立即常数，它允许在－128 到 127 范围中的数在指令中只用一个字节给出。当该指令被执行时，一个字节的常数 FBH 被扩展到 32 位。用带符号的扩展法得 FFFFFFFBH，2 的补码表示即为－5。

2.4.3 寄存器操作数

一条指令的操作数可以被分配在处理器的寄存器中。由指令中的寄存器操作数指定符选择 8 个通用寄存器中的一个，或 6 个段寄存器中的一个。指令中操作数的大小决定了 8 个通用寄存器中的一个是以 8 位、16 位或 32 位的操作数的寄存器来访问。6 个段寄存器总是以 16 位的寄存器来访问。

图 2.13 说明了在一条指令中如何编码寄存器操作数指定符。

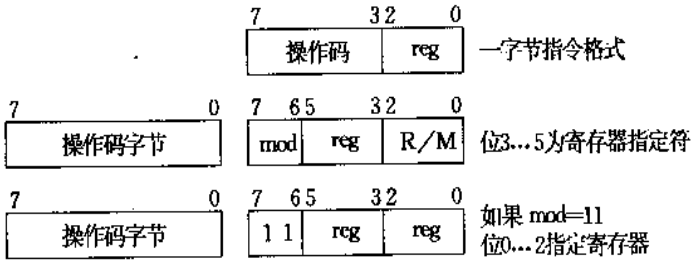


图 2.13　寄存器操作数的编码

一字节指令的形式使用指令的低三位编码 8 个通用寄存器中的一个。二、三字节的指令使用跟在一个或两个字节的操作码后的 MODRM 字节去编码一或二个寄存器操作数。MODRM

字节包含三个字段。R/M 字段在低三位上，取决于 mod 字段的设置；它指定是寄存器操作数，或者是存储器操作数的部分。mod 字段是一个二位的字段，它在 MODRM 字节的高二位上。它指定 R/M 字段应被解释成一个三位的寄存器名或作为一个存储器操作数。如果 MOD 字段的值为 11b(两位都是 1)，则 R/M 字段指定了一个寄存器操作数，否则 R/M 字段指定了一个存储器操作数。R/M 这个名字起源于本字段的双重解释，按照 MOD 字段的值，它作为一个寄存器 Register 或作为一个存储器 Memory 操作数。本字段取名为 MODRM 字节是取自第一个字段和最后一个字段的名字组合而成。这两个字段总是存在的。MODRM 字节的中间三位可以指定一个寄存器操作数或者提供作为 3 个附加的操作码位，这要取决于指令第一字节的操作码。

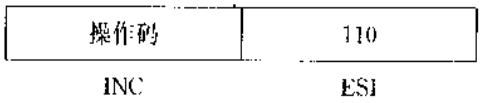
在所有这些情况中，寄存器操作数由指令中 3 个位的码来指定。表 2.3 按照指令操作数的大小给出了 8 位、16 位和 32 位通用寄存器的编码，及 3 位寄存器操作数指定符中的值。表2.4 给出了段寄存器操作数的 reg 字段的编码。

表 2.3　通用寄存器的编码

REG 码	32 位 寄存器	16 位 寄存器	8 位 寄存器
000b	EAX	AX	AL
001b	ECX	CX	CL
010b	EDX	DX	DL
011b	EBX	BX	BL
100b	ESP	SP	AH
101b	EBP	BP	CH
110b	ESI	SI	DH
111b	EDI	DI	BH

例子：最简单的寄存器操作数的编码使用一字节的指令格式，它把寄存器码放在一个字节的低 3 位而操作码放在其高 5 位。

INC ESI 将被编码成：



例子：用两个寄存器操作数的指令使用有 MODRM 字节的指令格式。在这种情况下，MOD 字段是 11b，表示 R/M 字段包含了一个第二寄存器操作数。

MOV　DH,DL 将被编码成：

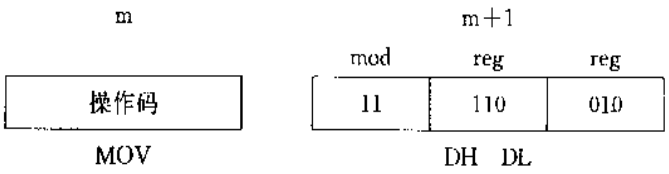
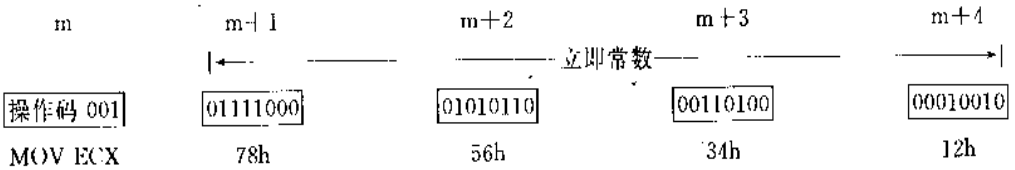


表 2.4 段寄存器的编码

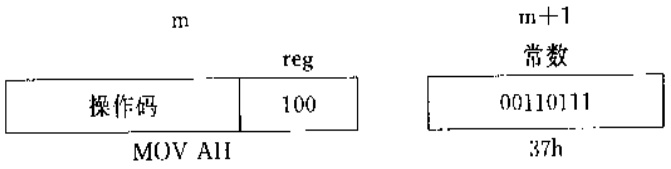
REG 码	段寄存器
000b	ES
001b	CS
010b	SS
011b	DS
100b	FS
101b	GS
110b	保留
111b	保留

例子:把一个立即常数移入寄存器的指令对于初始化循环计数是非常有用的。这样在整个循环中可以把计数变量分配给寄存器。这条指令使用了 1 字节的操作码的形式。在其低 3 位中有一个寄存器操作数指定符。在这些例子中存储器寻址顺着 m,m+1,…,m+4 来表示的。注意,操作码是在最低地址的那个字节中。而常数的存储使用低地址放低位字节。

MOV ECX,12345678h 将被编码成:



MOV AH,37h 将被编码成:



2.4.4 存储器操作数

一条 80386 指令只可以指定一个显式的存储器操作数。对于一个存储器操作数必须指定一个存储器地址的段部分和偏移地址部分两个部分。段的部分由给出段寄存器来指定,段寄存器中包含有操作数所在段的选择子。在大多数情况下,可以用默认的段寄存器。这样可以不要求别的指令的位。如果使用别的段寄存器,则需要用一个指令前缀字节。

偏移部分由地址模式来指定。地址模式指出基地址寄存器、变址寄存器、变址的比例因子以及常数位移量。对于指定存储器操作数的指令总是有地址模式的,并且包含在操作码字节之后一个或几个字节中。一个或两个字节的 MODRM 字段指定基地址寄存器变址寄存器和比例因子,并且指出了位移量的大小。如果指出,一、二或四字节的常数位移量跟在 MODRM 字节之后,则二或四字节位移量的低字节存储在低地址处,而高字节存储在高地址处。

2.4.4.1 一字节地址模式的编码

对于不给出变址寄存器,亦即当地址模式指定了至多一个基地址寄存器和一个位移量时,

这样一种重要的特殊情况，提供了一种特殊的短地址指定符。当使用只包含一个基地址寄存器或只包含一个位移量，或一基地址一位移量两个量这种简单的地址模式时可以节省一个字节。

图 2.14 示出这种一字节指定符的格式。这就是在上面的关于寄存器操作数的编码那一节中讨论的 MODRM 字节。其低三位 R/M 字段指定了基地址寄存器。本字段中 100b 码是一个专门的转换码，它指明将用 2 字节的形式，即在随后带有第二字节的地址模式格式。否则这三个位编码了 32 位的基地址寄存器，且用了如在表 2.3 中给出的用来指定寄存器操作数相同的编码。注意，在这种方式寻址中只可使用 32 位的寄存器。

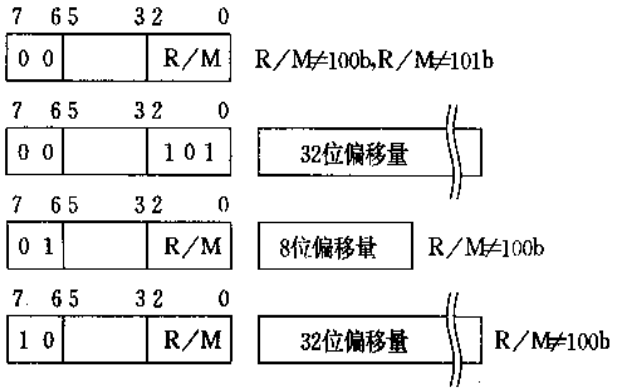
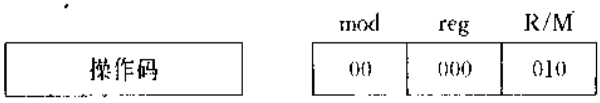


图 2.14 一字节地址模式格式

高二位(mod)指定了位移量的长度，00b 码指明了没有位移量。01b 码指明了之后跟有一个 8 位的位移量。在复制时把 8 位位移量的符号位 7 填充到高 24 位(位 8 到位 31)中去扩展成 32 位。10b 码指明了其后跟有全 32 位的偏移量。本字段中的 11 码指明 R/M 字段规定为寄存器操作数，而不是上述的存储器操作数。

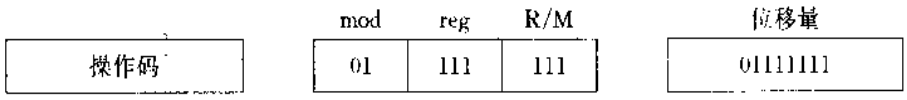
例子：如果 mod＝00b，则指定寄存器间接寻址模式。且基地址寄存器是在 R/M 字段中给出。

ADD EAX,DS:[EDX] 将被编码成：



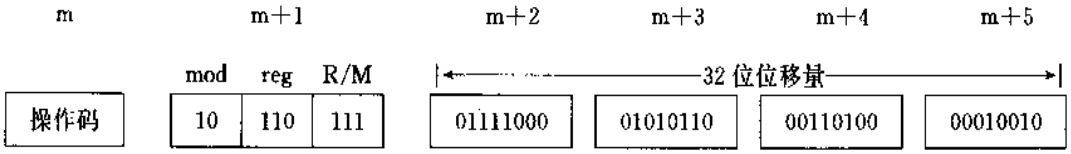
例子：如果 mod＝01b，则指定为寄存器间接加位移量寻址模式。并且基地址寄存器在 R/M字段中给出，其后还跟有一个 8 位的位移量。8 位位移量带符号扩展构成 32 位位移量加到基地址寄存器的值上。

SUB EDI,DS:[EDI＋127]将被编码成：



例子：如果 mod＝10b，则指定为寄存器间接加位移量寻址模式。并且基地址寄存器在 R/M字段中给出，其后还跟有一个 32 位位移量。在这里存储器地址以 m，m＋1，…，m＋5 来表示，以便指明 32 位位移量的低字节存储在低的存储器地址上。

ADD ESI,DS:[EDI+12345678h]将被编码成：

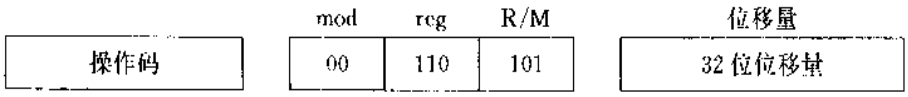


一个特殊情况需要更进一步说明。mod=00b 和 R/M=101b 的组合并不表示 EBP 作为一个基地址寄存器且不带位移量，而是只对位移量的情况的编码。就是说对既没有基地址寄存器，也没有变址寄存器，而只是带有一个全 32 位的位移量的情况的编码。

这种特殊情况是必需的，因为实际上需要十个基地址寄存器的编码数。8 个通用寄存器每个要一个。一个用于转换为二字节形式的编码以及一个用来指明没有基地址寄存器。因为对于基地址寄存器的编码只有三个位可以利用，于是 ESP 的编码被选作转换为二字节形式的 MODRM(mod≠11b 并且 R/M=100b)，而某些编码空间被从 mod 字段中窃取过来允许基地址字段去指定无基地址寄存器(mod=00b 和 R/M=101b)，这样编码意味着不能指定用 EBP 且无位移量的寻址模式，替之以用 EBP 作为基地址寄存器跟有一个 8 位的零位移量。

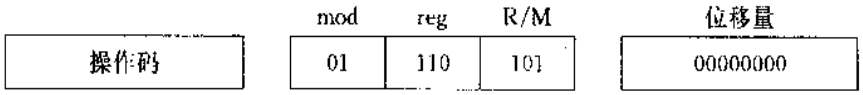
例子：一个静态分配的变量可以用只是位移量的寻址模式来寻址。其 mod=00b，R/M=101b。

MOV ESI,Scalar Var 将被编码成：



例子：用 EBP 作为基地址寄存器加一个 0 的位移量的存储器寻址可以用 mod=01b，R/M=101b 以及一个包含 0 的 8 位位移量来指定。

MOV ESI,SS:[EBP]将被编码成：



如果一条指令既要求一个位移量字段又要求一个立即常数，位移量首先出现。如在下面的例子中所示。

例子：一个静态分配的变量可以用 MOV 指令来初始化。这条 MOV 指令有仅仅是位移量的寻址模式带有 32 位立即常数字段。注意，立即数字段出现在位移量字段之后(在较高的存储器地址处)。

MOV ScalarVar,BigImm 将被编码成：

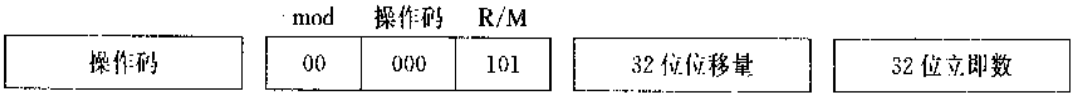


表 2.5 总结了一字节寻址模式形式的编码。mod 和 R/M 字段指定了基地址寄存器和位移量的大小，并且隐含指明了默认的段寄存器。右边一列给出了指定地址模式的汇编语言语法

的表示。因为汇编程序员通常不知道位移量的位数，Disp32 是要求 32 位位移量的变量的占有者。而 Dis8 是要求 8 位移量的变量的占有者。

表 2.5　一字节寻址模式的编码

MOD 字段	R/M 字段	寻址方式
00b	000b	DS:[EAX]
	001b	DS:[ECX]
	010b	DS:[EDX]
	011b	DS:[EBX]
	100b	DS:转换到 2 字节
	101b	DS:Disp 32
	110b	DS:[ESI]
	111b	DS:[EDI]
01b	000b	DS:Disp8[EAX]
	001b	DS:Disp8[ECX]
	010b	DS:Disp8[EDX]
	011b	DS:Disp8[EBX]
	100b	DS:转换到 2 字节
	101b	SS:Disp8[EBP]
	110b	DS:Disp8[ESI]
	111b	DS:Disp8[EDI]
10b	000b	DS:Disp32[EAX]
	001b	DS:Disp32[EBX]
	010b	DS:Disp32[EDX]
	011b	DS:Disp32[EBX]
	100b	DS:转换到 2 字节
	101b	SS:Disp32[EBP]
	110b	DS:Disp32[ESI]
	111b	DS:Disp32[EDI]

2.4.4.2　二字节寻址模式的编码

大多数通用的地址模式形式被编码成两字节。这样为指定变址寄存器和变址比例因子提供了附加的空间。这种两字节的形式也支持使用 ESP 作为基地址寄存器相对于程序堆栈的顶来寻址。

两字节的形式在图 2.15 中说明。如在一字节地址模式的形式中所提到的，两字节的形式是由在第一字节中的 R/M 字段中的 100b 转换码来指出的。第一字节的 mod 字段和一字节形式中的第一字节是完全一样的。

base 字段在第二字节中的位 0—2 中。它给出一个 32 位基地址寄存器的编码，其编法与在表 2.3 中给出的寄存器操作数的编码法相同。正如一字节指令的构成那样，mod＝00b 和 base＝101b 两字段的组合不表示 EBP 作为基地址寄存器且不带位移量。相反它是编码 变址＋位移量的情况的。就是说没有基地址寄存器但是有全 32 位位移量和一个变址寄存器的寻址方式。

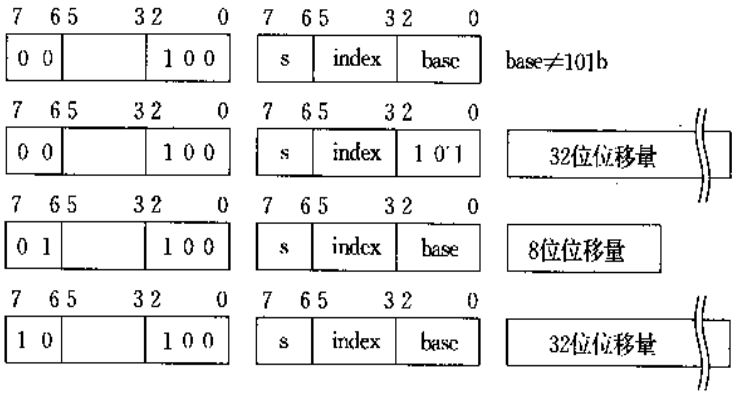


图 2.15 两字节地址模式格式

index 字段在位 3—5 中。它指定了 32 位变址寄存器的寄存器的编码。也和表 2.3 中的相同。在 index 字段中 100b 码不是指定 ESP 是一个变址寄存器，而是用来表示没有用变址寄存器。在位 6—7 的 s 字段中，它指定了变址寄存器的比例因子。如果不给出变址寄存器，比例因子必须为 00b。

表 2.6 总结了两字节地址模式形式的编码法。base 和 mod 字段指定了基地址寄存器及位移量的大小，并决定了默认的段寄存器。index 和 base 字段独立地修饰外加一个变址寄存器及带有一个比例因子的地址。和表 2.5 一样，地址模式是以汇编语言的方式给出的。变址寄存器修饰符与段/基地址/位移部分连在一起以得到完整的汇编语言的语法。例如，一个 EDX 基地址一个 8 位的位移量加一个 ECX 乘以 4 的比例因子的变址以汇编语言的语法表示将以 DS:Disp8[EDX][ECX * 4]给出。另一种替代的办法是变址部分可以变成加上去的形式如：DS:Disp[EDX + ECX]表示。

例子：一个基地址＋变址的寻址方式是用 mod＝00b 和 base(基地址)和 index(变址)字段分别给出基地址和变址寄存器来给出的。

MOV EAX,DS:[EBX＋ESI * 2]将被编码为：

	mod	reg	第 2 字节	s	idx	bas
操作码	00	000	100	01	110	011

例子：一个三部分齐全的基地址＋变址＋位移的寻址模式是由 mod－01b 指明 8 位的偏移量和 mod－10 来指明 32 位位移量的。

MOV EAX,SS:[ESP＋24][ESI * 8]将被编码为：

	mod	reg	第 2 字节	s	idx	bas	位移量
操作码	01	000	100	11	110	100	00011000

MOV EAX,DS:[9999999＋EDI＋EAX * 4]将被编码为：

	mod	reg	第 2 字节	s	idx	bas	
操作码	10	000	100	10	000	111	32 位位移量

表 2.6 两字节地址模式的编码

MOD 字段	R/M 字段	寻址方式
00	000	DS:[EAX]
	001	DS:[ECX]
	010	DS:[EDX]
	011	DS:[EBX]
	100	SS:[EBP]
	101	DS:Disp32
	110	DS:[ESI]
	111	DS:[EDI]
01	000	DS:Disp8[EAX]
	001	DS:Disp8[ECX]
	010	DS:Disp8[EDX]
	011	DS:Disp8[EBX]
	100	SS:Disp8[ESP]
	101	SS:Disp8[EBP]
	110	DS:Disp8[ESI]
	111	DS:Disp8[EDI]
10	000	DS:Disp32[EAX]
	001	DS:Disp32[EBX]
	010	DS:Disp32[EDX]
	011	DS:Disp32[EBX]
	100	SS:Disp32[ESP]
	101	SS:Disp32[EBP]
	110	DS:Disp32[ESI]
	111	DS:Disp32[EDI]

变址	寻址方式
000b	[EAX * s]
001b	[ECX * s]
010b	[EDX * s]
011b	[EBX * s]
100b	无变址
101b	[EBP * s]
110b	[ESI * s]
111b	[EDI * s]

s 值	变址比例因子
00b	1
01b	2
10b	4
11b	8

例子:在只带有一个位移量的组合中用指定一个带比例因子的变址,单个带比例的变址寻址可以处理静态分配的数组这样一种重要的情况。这是使用 mod=00b 和 base=101b 来指出无基地址,且带 32 位位移量,变址和比例字段是分别用来指定变址寄存器和比例因子的。

MOV EAX,TABLE[ESI * 4]将被编码为:

操作码	mod	reg	第2字节	s	idx	bas	32位位移量
	00	000	100	10	110	101	

例子：堆栈相对寻址是用二字节形式使用ESP作基地址寄存器不带变址码(100b)，并且选择0(栈顶)、8位或32位(内部堆栈元素引用)的位移量。

MOV EDI,0[ESP]；将被编码为：

操作码	mod	reg	第2字节	s	idx	bas
	00	111	100	00	100	100

MOV EDI,24[ESP]；将被编码为：

操作码	mod	reg	第2字节	s	idx	bas	00011000
	01	111	100	00	100	100	

2.4.4.3 跨段前缀字节

指令前缀字节可以在默认的段寄存器不适合的情况下指定寻址一个存储器操作数时使用的段寄存器。顾名思义，它是加在一条指令的前面的8位的字段。因为有6个段寄存器，所以有6个字节的编码专为跨段前缀字节。如表2.7所示。

例子：为了通过指针寻址存储器，首先把指针装载到段寄存器和通用寄存器对中。然后使用与指定段寄存器的跨段前缀相结合的通用寄存器作基地址的地址模式。提供了对由指针所指的数据的访问。

MOV EDX,ES:[ESI]将被编码为：

ES:前缀	操作码	mod	reg	base
00100110		00	010	110

表2.7 跨段前缀的编码

编码	段寄存器
26h	ES:
2Eh	CS:
36h	SS:
3Eh	DS:
64h	FS:
65h	GS:

例子：为了寻址存储在代码段中的数据，可以指定CS跨段前缀。例如，一张标号偏移地址表可以存储在代码段中，用来实现CASE或SWITCH语句。带比例的变址+位移与CS跨段前缀相结合构成了被引用该表的间接跳转指令JMP所用的必要的寻址模式。

JMP CS:24[ESI*4]将被编码为：

CS:前缀		mod	reg	第二字节	s	idx	bas	
00101110	操作码	00	100	100	10	110	101	32位位移量

2.5 I/O空间

前几节叙述了主存储空间的结构以及主存储器中的操作数在指令中是如何被引用的。还有个叫做I/O空间的第二个存储空间也是可以利用的。顾名思义，是用它来存储输入或输出(I/O)设备的控制通道是最适当的，设备则可以是如键盘、磁盘、CRT显示器、打印机等。为I/O通道单设一个独立的空间是适当的。因为I/O设备的寻址与保护和程序的代码和数据的寻址及保护是非常不同的。典型地说，一个I/O设备只有不多的几个控制通道，只需要少量的可编址存储，并且在系统中也只有少量的设备。

在I/O空间中I/O通道是不能重新分配的。并且要分别地保护它们，以便可以控制个别的设备的访问。另一方面，程序代码和数据需要成千上万字节的可编址存储，并且需要不同的保护机制。

在80386上提供了一个独立的I/O空间。I/O空间是不分段的。它是一个64K字节的一维地址空间。与主存储器一样，I/O空间是以字节来编址的。I/O通道可以是1、2或4字节的长度。如在主存储器中那样，用于寻址2或4字节通道的字节次序也用"低址结尾"的方法。较低地址的字节包含有最低有效位。较高地址的字节包含有最高有效位。

第3章中叙述的专门的指令被提供来在I/O空间和处理器寄存器及主存储器之间传输数据。

I/O地址规定如下：

(1) 如果I/O地址小于256，在指令中可以用8位的立即字段给出。

(2) 较大的I/O地址，或当程序执行时计算的地址从DX寄存器中取。

64K的I/O空间由第4章中所说的一种机制来保护。这种保护机制允许分别保护I/O空间中的每个字节。它与主存储器的保护机制是完全分开并全然不同的。

2.6 浮点寄存器

如在第1章中提到的，80387协处理器可以和80386一起使用来获得对浮点格式之一的数据的高性能的运算。即使你没有80387协处理器，你将必须用软件来仿真，以便能对浮点数据起作用。因此无论你用其芯片本身或者使用一个仿真，你都必须熟悉80387的结构。

当和80386一起工作时，80387增加三组寄存器来支持浮点运算。三组寄存器如图2.16所示。

(1) 一组8个80位的浮点累加器的堆栈能容纳多达8个浮点操作数。

(2) 三个16位的状态和控制寄存器：一个作为状态字，一个作为控制字，还有一个作为特征位字。

(3) 4个32位的错误指针寄存器(FIP、FCS、FOO和FOS)以识别发生异常的指令和存储器操作数。

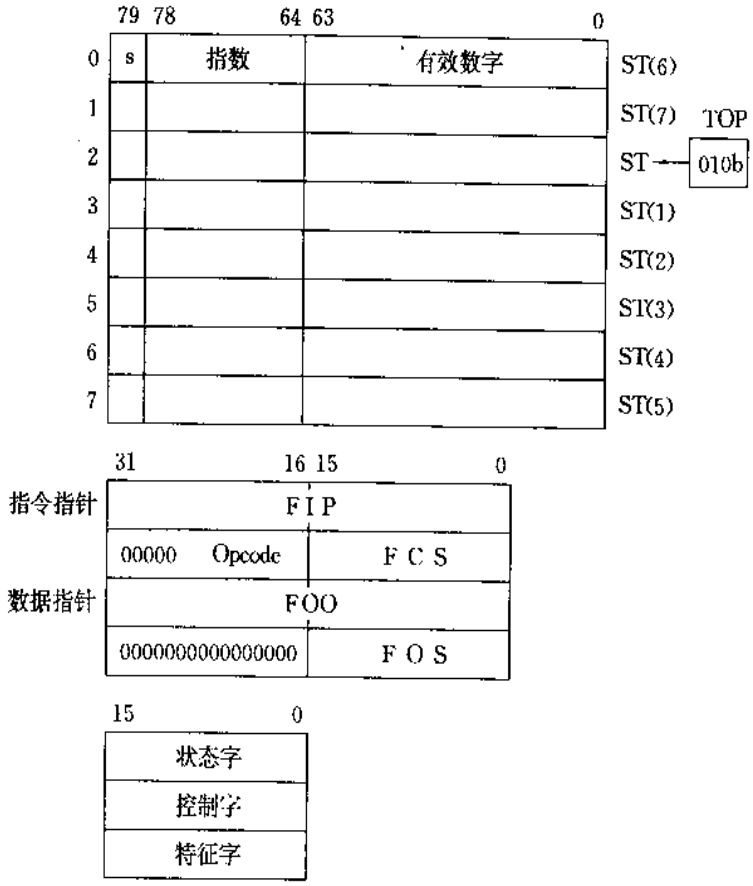


图 2.16

2.6.1 浮点累加器堆栈

浮点指令把80位的寄存器当作累加器堆栈来对待，而不是作为一个简单的8个寄存器的数组来处理。累加器堆栈的顶叫做ST。在ST以下的累加器的编址是相对于ST并以ST(i)来命名，i的范围是1到7。ST(0)可以被用作累加器堆栈顶的名字的代名字。

ST起一个累加器的作用，因为它隐含地被所有浮点指令当作一个操作数来使用。只从80386的操作码空间中的8个编码就可以提供64个浮点操作码。由于努力才达到了这样的密度。否则将是外加一个3位的寄存器字段的操作码位。为了避免大多数累加器模式的限制，与对内部的累加器的操作数的访问一起提供了8个累加器的堆栈。

虽然浮点运算必须用ST作为一个操作数，它们还是可以用名字来引用任何一个浮点累加器或一个存储器操作数作为第二个操作数。

累加器堆栈实际上是一个有8个物理寄存器的数组。用单独的一个三位的字段去识别当前的栈顶。小心使用术语寄存器和累加器是为了区分在这些浮点指令操作数的位置的两种解释。术语物理寄存器指的是堆栈的物理组织。术语累加器叙述一个浮点操作数的位置。这样我们就有了一个累加器堆栈和一个物理寄存器数组的说法。

在状态字中取名为TOP的一个三位的字段包含有当前累加器堆栈ST或ST(0)绝对的寄存器数。一个压入堆栈的操作将递减TOP并且将一个新值存储入新的ST中去。一个压入

(PUSH)堆栈操作之后，先前的 ST 成为 ST(1)，并且所有入栈的累加器使它们的 ST 相对名字加上 1。一个从堆栈的弹出(POP)将从当前的 ST 读一个值，然后增加 TOP。一个弹出(POP)之后，POP 之前取名为 ST(1)的累加器成为 ST，并且所有入栈的累加器使其 ST 相对名字减去 1。

3 位的 TOP 字段的增量和减量实现环绕。如果 TOP=000b，一压栈(PUSH)将递减 TOP 使之成为 111b，并把新的值存储到寄存器 7 中去。如果 TOP=111b，一个 POP 弹出操作将从寄存器 7 读取一个值。然后将 TOP 增加到 000b。

浮点累加器堆栈在图 2.16 中示出。假定 TOP 字段包含有 010b，在右面的 ST 相对累加器名对应于左边的物理寄存器数。当前累加器顶 ST 对应于物理寄存器 2。在堆栈中的每个 80 位的累加器提供存储一个第 1 章中所叙述的临时实数格式的实数。最高位是符号位，位 64—78 提供一个 15 位的指数字段，位 0—63 提供一个 64 位的有效数字字段。

第 3 章中叙述的浮点指令被设计来对这种累加器堆栈模式起作用的。装入指令从存储器读一个操作数，并且把它压入累加器堆栈。存储指令从当前栈顶取出值来并将其写到存储器中。如果对于另外的计算不马上需要该值，还可以有选择的弹出累加器堆栈。如加、乘的算术运算，从 ST 寄存器取出一个操作数，从另一个寄存器或存储器取出另一个操作数，并把结果存回到 ST 寄存器中。一个专门的运算并弹出的操作是可以利用的。这种操作对顶上的两个堆栈操作数运算即 ST 和 ST(1)，并且一次弹出堆栈。把结果存到新的 ST 中。这种运算并弹出的次序等价于分类堆栈机器的操作。这种操作从堆栈弹出两个源操作数 ST 和 ST(1)，执行一个运算。然后把结果压回到堆栈中去。

下面的例子用几个按指令执行的堆栈的片断说明累加器堆栈的操作。别的一些关于累加器堆栈的操作在第 6 章中给出。

假定你需要计算两个矢量的点积，它要求你去形成积的总和。你可以把累加的和放在 80387 的某个寄存器中，形成的每一个积放在另一个寄存器中。如果 ST 中保持部分积的总和，你可以用一个矢量的下一个元素装入到累加器堆栈中，再乘以另一个矢量的下一个元素，然后把结果加到部分积累加器中去的办法来执行一步矢量的点积。

图 2.17 示出的几个 80387 累加器堆栈的图，展示了矢量点积运算过程的这个步骤。第一个片断展示了存储在 ST 中具有部分积的和为 101.237 的累加器堆栈，其它的寄存器都是空的(用阴影表示)。左边示有物理寄存器数，累加器的名字示在右边。ST 在物理寄存器 5，即本例的起点上。这样，TOP 字段如所示的包含有 101b。

第一个矢量的下一个元素(其值为 1.21)用一条装载指令压入累加器堆栈，使堆栈有了两个元素。如下一个图所示。新的值被装入到物理寄存器 4 中，它就成为栈顶寄存器 ST。现在部分总和可以用 ST(1)来寻址，此时它当然还在物理寄存器 5 中。第三个图表示把 ST 乘以第二个矢量的下一个元素(其值为 5.0)后的堆栈。这把 ST 中的值改变为 6.05，但并不压入或弹出堆栈。最后一个图表示一个加法和弹出指令 FADDP 操作后的堆栈。这个操作就是把新的积加到部分总和寄存器中。这条指令把 ST(新的积)与 ST(1)(老的部分积)相加并把一个元素弹出堆栈。用结果 107.287 代换 ST。该值是现在的部分总和。这使堆栈处于本例开始时一样的状态。部分总和在 ST 中，这样你已经准备好了下一个迭代过程。

2.6.2 十六位的状态和控制寄存器

三个十六位的寄存器(特征字、控制字和状态字)控制着浮点指令的运算并提供状态信息。

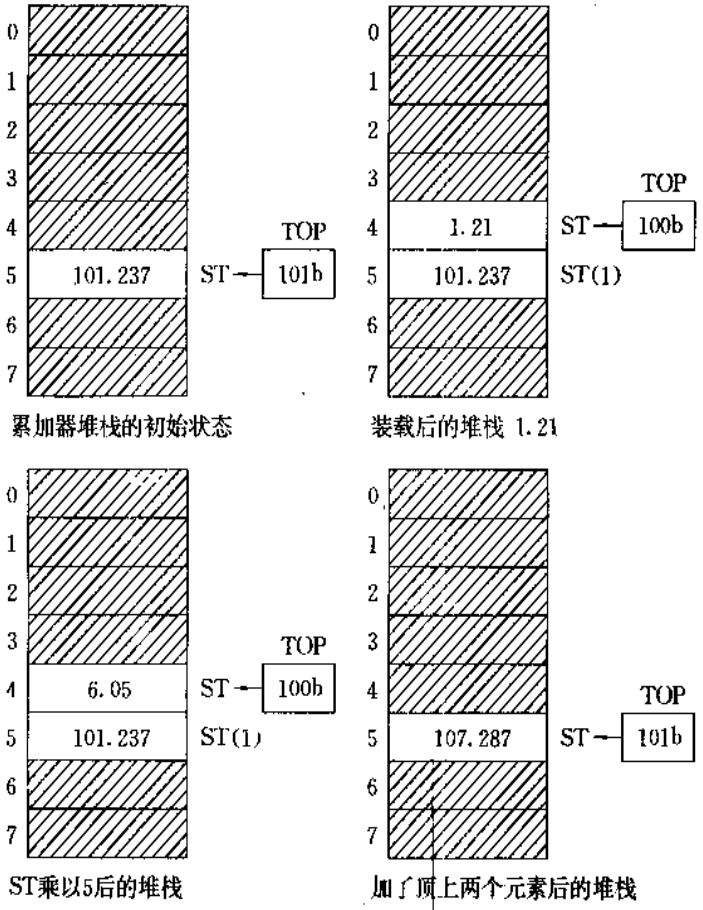


图 2.17 80387 累加器堆栈操作

2.6.2.1 特征字寄存器

图 2.18 中所示的 16 位的特征字寄存器包含有 8 个 2 位的字段，每个字段为对应的 8 个物理浮点寄存器所用。这些字段指示对应的物理寄存器容纳有一个有效的零或特殊的浮点数或是空的。特征字中的字段对应于物理寄存器，而不是相对于堆栈的顶。这样可以避免当累加器堆栈压入或弹出时特征字也跟着变动。

特征字段是用来检测累加器堆栈的上溢出和下溢出。如果一个压入操作递减 TOP 使其指向一个不空的寄存器，就发生堆栈的上溢出。如果企图去读或弹出一个空的寄存器，就发生堆栈的下溢出。堆栈的上溢出或下溢出将发生一个无效操作异常。

2.6.2.2 控制字寄存器

控制字寄存器可以由程序来进行设置以便控制 80387 的操作。在控制字中有三个字段。在图 2.19 中示出。

控制字中的每个字段简述如下：

(1) 位 0—5 包含有表 2.8 中所列的条件的异常屏蔽分离字。如果 80387 检测到一个异常，就测试对应那个异常的屏蔽位来决定应该把异常传递到软件错误处理程序(mask=0)还是传到 80387 中默认的错误处理程序去处理(mask=1)。如果一个异常的 mask 为 1，就说它被屏蔽了。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
tag7		tag6		tag5		tag4		tag3		tag2		tag1		tag0		TW(Tag)

特征字段编码	
Tag_i	含义
00b	有效
01b	零
10b	特指无穷,NaN,微小数
11b	空

图 2.18 特征字细节

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

精度控制	
PC	有效数字的精度
00b	24位(短型实数)
01b	(保留)
10b	53位(长型实数)
11b	64位(临时实数)

舍入控制	
RC	舍入方式
00b	舍入到最接近的
01b	舍入到负无穷
10b	舍入到正无穷
11b	截为0

图 2.19 控制字细节

表 2.8 异常状态和屏蔽位

状态	屏蔽	条件
IE	IM	非法操作
DE	DM	微小数
ZE	ZM	除以零
OE	OM	上溢出
UE	UM	下溢出
PE	PM	精度

(2) 位 8—9 包含有精度控制字段,PC。浮点加减乘除和开平方根的结果按照图 2.19 中给出的三种精度方式之一的规定舍入后被存储到目标中去。所有别的运算使用临时的实数精度或者使用由指令中指出的精度。

(3) 位 10- 11 包含有舍入控制字段,RC。图 2.19 中所列出的四种舍入模式可以用于舍入浮点运算的结果。

在控制字中其余的位由硬件接线使其成为图 2.19 中所示的值。当把一个值装载到这些寄存器中的时候,这些位应该为所示的值,或者应该从保存起来的控制字映象重新装入。这种映象不能被改变,以便支持和别的可能定义附加的控制位的数学处理器相兼容。

舍入:每当可能,80387 总是产生正确的结果。然而,在中间计算期间或者存储一个较小的数据类型期间,可能发生精度的丢失。当内部临时实数格式转成为一个短整数存储时,精度的丢失是明显的。当发生这些情况时,80387 将舍入结果,使它能与较小的操作数相符合。

80387 上有四种舍入模式,被规定在控制字寄存器的 RC 字段中:(1)舍到最接近的;(2)向下舍入到负无穷;(3)向上舍入到正无穷;(4)数量上截得较小。

当执行整数的算术运算时截小是有用的,因为它和整数除法指令对商进行舍入的方法相匹配。

舍到最接近的方式适合于大多数的应用。这是默认的舍入方式,因为当复位时 80387 初始化 RC 字段到 00b。如果你对数学方法不熟练,这个方法是足够的了。它产生最精确的结果形式并且决不把偏差引入到计算中去。

舍到正、负无穷方式被提供来为区间算术所用。当你认识到结果有错误,但你似乎应该对错误设置一个界限时这是很有用的。审慎的使用舍到无穷的方式,你可以正确地创立确定计算的错误的边界。

无穷大:先前的 86 系列数学处理器成员(8087 和 80287)能够以仿射和发射两种方式来处理无穷大数。并且用控制字的位 12 来指定其方式。80387 根据第 1 章中讨论的 IEEE 标准只能支持无穷大处理的仿射方式。

8087 和 80287 使用控制字寄存器中的位 12 的 1 来指定使用仿射的方式,0 表示使用发射的方式。这种老的协处理器在复位时或者经过执行 FINIT 指令之后,初始化该位到 0(发射方式的设置),在 80387 上该位也被初始化到 0。这样就把控制字初始化到所有协处理器都一样的值上。附录 C 叙述了这一点以及更详细的叙述了别的在 80386 和 8087/80287 协处理器之间的一些区别。

仿射方式辨别在计算中的正和负的无穷大。符号表示你从哪一个方向达到无穷大的,在有些情况下这可能是个有用的信息。然而一个算法可能产生正负零作为正确的结果。如果你然后执行 1/0 或 1/−0,你可以分别得到正无穷大或负无穷大。这样看起来有更多的信息,然而它未必有用。实际上,在有些情况下,可以把它看成不对的。这样,小心解释无穷大的符号是很重要的。

2.6.2.3 状态字寄存器

80387 设置状态寄存器中的某些位。这些位可以由程序来测试以便检验某些特殊情况。状态字寄存器如图 2.20 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
B	C3	TOP			C2	C1	C0	IR	SF	PE	UE	OE	ZE	DE	IE	SW(状态字)

图 2.20 状态字寄存器

状态字中每个字段简述如下:

(1) 当执行一个浮点指令时检测到一个异常，位 0—5 被置位。这些位是粘性的，任何时候 80387 在检测到条件时就置位这些位，但必须由程序显式地使它们复位。表 2.8 列出了这些位和异常之间对应的关系。

(2) 位 6(SF)是堆栈失效标志。如果一个非法操作异常是由于累加器堆栈的上溢或下溢引起的，该位就被置位。否则就复位。

(3) 位 8(C0)、9(C1)、10(C2)和 14(C3)包含有浮点状态码位。有一组浮点比较指令，这些指令设置这些位以指示比较的结果。这些条件码位位置的安排上是这样的，它们可以移到 80386 的 EFLAGS 寄存器的低 8 位去。这样同一组条件跳转指令可以被用来为浮点条件所使用，如同由 80386 检测到的条件码一样。第 6 章中有一个测试浮点条件码的例子。

(4) 位 11—13 保持有 TOP 字段，它表示哪一个物理寄存器对应于累加器堆栈的顶。

(5) 在控制字中的低 6 位中的异常屏蔽位设置屏蔽时，如果在状态字的低 6 位中的异常位指示了任一个未被屏蔽的异常，则位 7(IR)和位 15(B)是 1；否则 IR 和 B 为 0。

2.6.2.4 错误指针寄存器

在图 2.16 中所示的最后一组 80387 寄存器组是 4 个 32 位的寄存器，它们保存有指向最近执行的 80387 指令及数据的指针。其前两个寄存器 FIP 和 FCS 保存有指向最近执行的 80387 的指令的指针并带有两个操作码字节(忽略前缀)。FCS 保持有选择子和操作码而 FIP 保存有偏移量。第二对寄存器 FOO 和 FOS 指向该最近执行的指令的存储器操作数。FOS 保存有选择子而 FOO 保存有偏移地址。如果最近的协处理器指令没有存储器操作数，在 FOO 和 FOS 中的值是未定义的。FLDENV、FSTENV、FNSTENV、FRSTOR、FSAVE 和 FNSAVE 这些在第 3 章中叙述的指令是提供来装载和存储这些错误指针寄存器的。

操作码字段是由两部分组成。即由第一个指令字节(前缀后的)的低三位和第二指令字节组合而成。如图 2.21 所示。

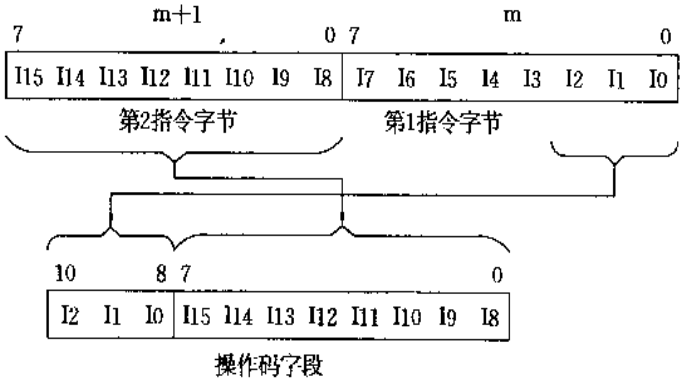


图 2.21 操作码字段的格式

所选的这两个字节来自任何指令前缀后的两个字节，因此包含有操作码字段和 MODRM 字段的第一个字节。在图 2.21 中，示出了在顶部的两个指令字节。如同它们是一 16 位的数一样，低位在低地址上。第一个指令字节示在右边的地址 m 处，其中包含有位 I0 到位 I7。第二个指令字节示在左边地址 m+1 处，其中含有位 I8 到 I15。第一指令字节的低三位(I0、I1 和 I2)是被放在操作码字段的位 8 到 10。第二指令字节的所有 8 位(I8 到 I15)是被放在操作码字段的位 0 到 7。因为第一指令字节的高 5 位(I3、I4、I5、I6 和 I7)与所有浮点指令的高 5 位都一样，

因此没有把它们存储在操作码字段中。操作码格式交换了操作码字节，以便所叙述的那些位在高位位置上，而不是在操作码字段的中间位置(注意早先的 386 的 B1 型并不存储操作码信息)。

错误指针对于分析和报告在浮点指令期间发生的异常是非常有用的。因为 80386 和 80387 可以并行执行，所以这些指针是必须的。一旦一条浮点指令被送到 80387，它就可以处理这条指令，而 80386 可以去执行别的指令。如果在一个 80387 的指令中发生一个错误，80386 上的 CS 和 EIP 寄存器可能并不在引起异常的指令附近。错误指针寄存器允许由 80386 和 80387 并行处理，又提供了诊断和恢复浮点异常的必要信息。

FLDENV、FSTENV、FNSTENV 操作数格式

在第 3 章中叙述的 FLDENV、FSTENV 和 FNSTENV 从图 2.22 中所示的一个 28 字节的存储器块中装入和存储控制字、状态字特征字和错误指针寄存器。在图示标为 0 的那些位是保留的。这些保留位在装载时必须以 0 来装载它，而在存储时可被忽略。

31 … 16	15 … 0	
0000000000000000	FOS	24
FOO		20
00000 操作码	FCS	16
FIP		12
0000000000000000	特征字	8
0000000000000000	状态字	4
0000000000000000	控制字	0

图 2.22 FLDENV、FSTENV、FNSTENV 的操作数格式

FRSTOR、FSAVE 和 FNSAVE 从一个 28 字节的存储器块中装入和存储所有的 80387 寄存器。该块中的低 28 个字节具有图 2.22 中所示的格式。装入或存储 ST(0)是对偏移 28 进行的，ST(1)是对偏移 38…等进行的，ST(7)的装入和存储是对偏移 98 进行的。当执行一个 FSAVE 或 FNSAVE 之后对应于标为空的累加器的存储器位没有定义。而 FRSTOR 指令的执行对此是忽略的。

第3章 指 令 集

80386 和 80387 的指令集相当丰富。本章涉及了 152 条 80386 指令和 74 条 80387 指令。从早期的 8086 到今天，指令集有了很大的发展，逐渐形成了象 80386 这样的支持多进程、多任务和虚拟存储器的 32 位处理器。在这个指令集中，既有大家熟悉的 8086 指令，也有在不同的处理器时代产生的支持更高级计算机结构性能（如虚拟内存）的指令。

本书将指令按用途分为下列四种类型：(1) 整数(Integer)，(2) 多段(Multiple Segment)，(3) 操作系统(Operating System)，(4) 浮点(Floating Point)。本章依次介绍这四类指令。

整数指令集占的篇幅最大，应用程序的作者可以在这里找到他们所需要的算术指令、逻辑指令、数据传送指令和简单的控制流指令。

第二类指令特为处理程序中的多段而设。应用程序一般只需处理单段的或平面的编程模式（在第 2 章中我们简单谈过），所以应用程序的作者不必涉猎这一部分。

第三类操作系统指令为操作系统的书写提供了很大的方便。同样，应用程序的作者也不需去看这一部分。

第四类为 80387 的浮点指令。

3.1 指令格式描述

任何事物只有从几个不同的角度加以描述，才能展示出它的全部性状。所以，在下面的指令集说明中，我们也用几种不同的方法来描述每一条指令。

首先给出与这条指令有关的所有汇编语言的语法及相应的机器码，然后用一段文字来描述该指令，这样往往比一个规格化的定义更好理解。当然，有关语义的规格化定义也是必不可少的。接下来我们举一两个例子，说明指令的使用方法和作用效果。最后给出这条指令的时钟期、执行时间、标志影响以及可能产生的异常。

3.1.1 指令格式

举例： **ADD** op_1 , op_2

（汇编格式） reg , reg

reg , mem

reg , imm

mem , reg

mem , imm

字节 0

0	0	1	1	0	1	1	1

（机器码）

这表示 ADD 指令允许有五种格式。上例中，reg 表示寄存器操作数，mem 表示存储单元操作数，imm 表示立即数操作数，下面还要详细介绍。汇编格式的右边，给出了这条指令的机器码。

斜体字 op_1 和 op_2 用在标准的（规格化的）操作说明中。这样，在操作部分我们只要直接用

$op_1=op_1+op_2$ 来表示 ADD 指令的操作过程，而不必对每种格式都分别用

reg=reg+reg

reg=reg+mem

……

来表示。

所有的操作数助记符列表如下：

reg：代表在第二章图 2.1 中描述的任何一种 8 位的、16 位的或 32 位的寄存器。斜体字 reg 代表这三种长度的寄存器中的随便哪一种，也就是说对寄存器的长度没有任何规定。

如果是特指某种长度的寄存器，则用 reg8、reg16、或 reg32，即在 reg 后面加上位长数字。某些指令对于使用的特殊寄存器作隐式的或显式的规定。而有些指令允许使用的寄存器种类较多，只要这些寄存器被规定为可以使用的。

mem：表示在第 2 章中描述过的任何一种内存寻址格式。内存格式可以是 8 位、16 位、32 位、48 位、64 位、80 位中的任何一种，甚至可以是 28 字节或 108 字节。在指令格式说明中，它们分别用 mem8、mem16、mem32……等来表示。当斜体字 mem 出现在指令格式中时，它仅代表 mem8，mem16 和 mem32。如果其它大小的内存允许被使用，当用显式标出。

imm：imm 表示任何长度的立即数(8 位，16 位或 32 位)。而 imm8，imm16 和 imm32 则表示特定长度的立即数。

Sreg：段寄存器。请详阅第 2 章。

ST：浮点累加器堆栈。许多浮点数据类型都用到这个堆栈，这在第二章中提到过。其栈顶用 ST 或 ST(0)表示，而栈顶下面的单元则用 ST(n)表示。请看第 2 章中的例子。

；：分号，用作汇编程序中的注释符号。

例如："ADD EAX，EBX；这是一个加法。"便是一条带注释的汇编语句。

注意：占据一行的一个格式化语句，要求它的操作数具有相同的长度，除非对操作数长度作了特殊规定。于是

ADD reg，reg

其操作数允许：reg8，reg8；reg16，reg16；或 reg32，reg32。而 reg8，reg32；reg16，reg8；…等等是不允许的。同样，

ADD reg，mem

只允许：reg8，mem8；reg16，mem16；或 reg32，mem32。还要注意的是，双操作数(二进制)操作的一般格式为

$op_1=op_1+op_2$

其中"+"号代表该指令相应的操作。在一般情况下，op_1 为目的操作数。例如，

ADD reg，mem

将 reg 与 mem 的和赋予 reg。

为方便读者查阅，我们在指令格式中还给出了指令的机器码。关于机器码，我们在篇后的附录 G 还将作详细的介绍。

3.1.2 指令说明

下面是一个指令说明的例子：

AAA 指令将在字节加法操作之后，将 AL 寄存器的内容进行 BCD 调整(非压缩的)。在一

般情况下，AAA 用在以 AL 的内容作为非压缩的 BCD 码执行字节加法之后，但也能够用于其它的 BCD 转换。

如果加法产生十进制进位，或者 AL 的低半字节大于 9，则 AL 增加 6，AH 增加 1，CF 和 AF 位置 1。如没有产生十进制进位，则 AH 不变，CF 和 AF 位清 0，AL 的低 4 位不变。无论是否产生十进制进位，AL 的高 4 位总为 0。

以上就是指令 AAA 的文字说明。在文字说明中，可能会再介绍一种有别于在操作部分描述过的算法。关于该指令的功能和用法也应当在这段文字里给出。

3.1.3 指令操作

```
实例：  AL=AL+(10*AH);
        AH=0;
        SF=B(7,AL);
        ZF=AL==0;
        PF=~(B(7,AL)^B(6,AL)^…^B(0,AL));
        AF=OF=CF=UNDEFINED;
```

以上便是 AAD 指令的算法描述。语句的顺序也是指令语义的一部分。也就是说，语句

```
ZF=AL==0;
```

在执行第一个语句

```
AL=AL+(10*AH);
```

之后再去访问 AL 的值。这个指令操作部分全面地描述了指令的功能。凡没有在这一部分出现的寄存器，标志，或操作数都不受该指令的影响。

用在指令操作部分的语法(也用于指令集说明的其它部分)将在下面描述。

3.1.4 指令举例

这些例子将有助于理解每条指令的操作。所有的例子都在 386 系统上做过实验，因此是有绝对把握的。下面举一个典型的例子：

```
ADD  EAX,EBX                                        OSZAPC
     执行前：EAX：01234567  EBX：76543210            ××××××
     执行后：EAX：77777777  EBX：76543210            000010
ADD  mem32,70000000h
     执行前：Mem：7FFFFFFF                           ××××××
     执行后：Mem：EFFFFFFF                           110010
```

这个根据 ADD 指令写的实例分为两部分：用 ADD 指令书写的汇编源程序，以及在指令执行前后的机器状态。寄存器状态一般表示为 16 进制，尽管它们并不带有后缀 H。而最右边的标志状态则用二进制表示，OSZAPC 对应着 OF，SF，ZF，AF，PF 和 CF 各位。无论是在寄存器状态或标志状态中，“×”均表示无关紧要或不知道。在上面的例中，ADD 指令执行前的标志状态就是不知道的，或是从了解指令的角度出发，指令执行之前的标志状态并不重要，它们不会影响指令的执行。

“U”表示不改变。在下面 JO 的例子中，指令执行前的标志也是不知道的，或是无关紧要的，而且指令的执行并不能改变它们。

JO near label1；label1 表示在程序中向前 68h 个字节

	OSZAPC
执行前:EIP：00000300	1 ××××××
执行后:EIP：00000368	U UUUUU

在指令执行之后，所有标志位保持不变。

3.1.5 指令的执行时间

以时钟脉冲个数(或时钟周期个数)来表示指令时间。并假设主频为 20MHz，给出指令执行所需的绝对时间。

3.1.6 指令对标志位的影响

在指令的说明和操作部分，已对标志位的变化做了说明，这里再次以醒目的方式提示读者将受到指令影响的标志位。

3.1.7 指令异常

在这一节中，我们预定义了若干级别的形简义明的函数调用，用于描述指令潜在的异常条件。例如，一条带有内存操作数的指令可能会产生由于内存操作数而引起的多种不同异常中的一个。于是调用下面给出的 Memory( )函数，应能检查出可能产生的出错条件。若指令说明中给出了下面定义过的一个或几个函数，表示该指令若是产生异常，必是这个或这些个给出的函数级别上的异常。若不给出异常，指令便不会产生异常。异常将在第 5 章讨论。

预定义的异常函数调用如下：

```
(1) Memory( ){
    /* 在访问内存期间的一切可能的异常统统包括在第 5 章中定义的 Access Virtua
        l( )程序之中。请详阅第 5 章。
    */
}
(2) Stack( ){
    /* 在访问堆栈期间的一切可能的异常统统包括在第 5 章中定义的 Access Virtua
        l( )程序之中，这里的 SReg 规定为 SS。请详阅第 5 章。
    */
}
(3) Code Reference( ){
    /* 检查控制转移指令的目标。如果目标的第一个字节超出了段界，则在控制转移指
       令之中报告一个段异常。当除目标指令的第一个字节之外的字节中产生段异常
       时，在指令取出之后，报告一个页异常。
    */
if(EIP>CS. Limit)
    /* 关于 SegmentException( )程序请阅第 6 章。
    */
    SegmentException($GP,0);
}
```

```
(4) AccessIO( ){
    /* 在第5章定义了此函数,以及所有可能的异常条件。请详阅第5章。
    */
}
(5) IOPLSensitive( ){
    /* SegmentException( )程序在第6章中定义。
       CPL 和 IOPL 将在第5章中详细讨论。
    */
if(CPL>IOPL){
    SegmentException($GP,0)
    }
}
(6)InvalidOpcode( ){
    /* 在第6章中定义 */
}
```

3.1.8 指令的注意事项

这一部分告诉读者应当特别注意的地方。例如,哪些指令是386或387才有,而86系列的其它成员没有;哪些指令执行起来和86系列的其它成员的指令不一样;指令的特殊边界效应或者不期望的结果等。并告诉读者可以参阅的章节,那里会有更多的例子和更详细的说明。

3.2 指令描述的语法

这一节里我们介绍在指令操作部分和指令异常部分所用的语法。这语法以C编程语言为基础。下面的一张表,给出了本书所用的C符号(包括扩充的)。

3.2.1 符号

逻辑操作符

&	逐位与	<<	左移
|	逐位或	>>	右移
^	逐位异或	~	求反(一元的),非

算术操作符

+	加	%	求模
−	减	++	增量
/	除	−−	减量

关系操作符和逻辑连接符

>	大于	>=	大于等于
<	小于	<=	小于等于

==	等于	||	或(逻辑连接)
! =	不等于	&&	与(逻辑连接)

其　它

=	赋值	&	操作数地址
/*	注释开始	[a]	地址为 a 的操作数
*/	注释结束	a is b	若 a 的类型为 b,即为真
{	语句块开始	c in(s)	若 c 属于 s 指定的集合,即为真
}	语句块结束		

常　数

DENORMAL	浮点不可规格化数	NORMAL	浮点规格化数
EMPTY	浮点寄存器空	NUMOPS	指定的操作数的数目
+INFINITY	浮点正无穷数	OPCODE	指令操作码
−INFINITY	浮点负无穷数	QUIETNaN	浮点静的非数字
MSB	最高有效位	UNDEFINED	未定义的结果
NaN	浮点非数字	UNSUPPORTED	不支持的浮点表示法

寄　存　器

EAX	32 位 EAX 寄存器	CH	8 位 CH 寄存器
ECX	32 位 ECX 寄存器	DH	8 位 DH 寄存器
EDX	32 位 EDX 寄存器	BH	8 位 BH 寄存器
EBX	32 位 EBX 寄存器	CS	16 位 CS(代码段)寄存器
ESP	32 位 ESP 寄存器	DS	16 位 DS(数据段)寄存器
EBP	32 位 EBP 寄存器	SS	16 位 SS(堆栈段)寄存器
ESI	32 位 ESI 寄存器	ES	16 位 ES(附加段)寄存器
EDI	32 位 EDI 寄存器	FS	16 位 FS(附加段)寄存器
AX	16 位 AX 寄存器	GS	16 位 GS(附加段)寄存器
CX	16 位 CX 寄存器	EIP	32 位指令指针寄存器
DX	16 位 DX 寄存器	CR0	32 位控制寄存器 0
BX	16 位 BX 寄存器	CR2	32 位控制寄存器 2
SP	16 位 SP 寄存器	CR3	32 位控制寄存器 3
BP	16 位 BP 寄存器	DR0	32 位调试寄存器 0
SI	16 位 SI 寄存器	DR1	32 位调试寄存器 1
DI	16 位 DI 寄存器	DR2	32 位调试寄存器 2
AL	8 位 AL 寄存器	DR3	32 位调试寄存器 3
CL	8 位 CL 寄存器	DR6	32 位调试寄存器 6
DL	8 位 DL 寄存器	DR7	32 位调试寄存器 7
BL	8 位 BL 寄存器	EFLAGS	32 位标志寄存器
AH	8 位 AH 寄存器	MSW	机器状态字

LDTR	局部描述符表寄存器	ST	浮点栈顶
IDTR	中断描述符表寄存器	ST(n)	浮点栈顶下面的第 n 个寄存器
GDTR	全局描述符表寄存器	TOP	栈顶指针
TR	任务寄存器	FIP	浮点指令指针
TW	浮点标记字	FCS	浮点代码段
SW	浮点状态字	FOO	浮点操作数偏移量
CW	浮点控制字	FOS	浮点操作数段

标　志

CF	进位标志	IOPL	I/O 特权级别
PF	奇偶标志	NT	嵌套任务标志
AF	辅助进位标志	RF	调试故障启用
ZF	零标志	VM	虚拟 8086 模式
SF	符号标志	C0	浮点条件码 0
TF	陷阱启用标志	C1	浮点条件码 1
IF	中断标志	C2	浮点条件码 2
DF	方向标志	C3	浮点条件码 3
OF	溢出标志		

其它标志/位/引脚

CPL	当前特权级	NMI	不可屏蔽中断引脚
DPL	描述符特权级	RPL	请求特权级
LOCK	总线锁定引脚	PE	保护启用

其它符号

imm8	8 位立即数	mem64	64 位内存指针
imm16	16 位立即数	mem80	80 位内存指针
imm32	32 位立即数	mem	8 位的，16 位的，或 32 位的内存指针
imm	8 位的，16 位的，或 32 位立即数	reg8	8 位寄存器
mem8	8 位内存指针	reg16	16 位寄存器
mem16	16 位内存指针	reg32	32 位寄存器
mem32	32 位内存指针	reg	8 位的，16 位的，或 32 位的寄存器
mem48	48 位内存指针	Sreg	段寄存器

全局变量

cnt	用于位计数的变量	disp	字节位移量
delta	在串说明中使用的变量	i	循环变量 tmp，

offset	位偏移量	tmp,temp	临时公用变量
quotient	临时存储商的变量		

数据类型

shortReal	浮点短实型数(32 位)	shortInt	80387 短整数(32 位)
longReal	浮点长实型数(64 位)	longInt	80387 长整数(64 位)
tempReal	浮点临时实型数(80 位)	int	整数
wordInt	80387 字整数(16 位)		

控制结构

if(e)b1　若表达式 e 为真,执行 b1

else b2　否则执行 b2

while(e)b3　当表达式 e 为真,执行 b3

for(e1;e2;e3)b4　执行 e1;当 e2 为真,执行 b4 和 e3

switch(e1){case－block}多路条件语句

case c1:　若常数 c1 匹配为真

break:　立即退出该程序级

以上大多数已为大家所熟识,或不用再解释了。只有少数还须再解释一下。

类型(type Casts)很象 C 语言的类型。它们之间可以互相转换。例如:

```
X=(shortInt)Y;
```

表示:在 Y 值转变为短整型数之后再赋予 X。"is"运算符可以用来检查数据类型。例如:

```
if(op1 is mem){
    whatever…
}
```

表示:若 op_1(第一个操作数)是一个内存操作数的话,便执行"whatever…"。

"in"运算符用于集合的概念。例如:

```
if(op1 in (mem 32,mem 64,mem 80)){
    Memory( );
}
```

表示:如果 op_1 在给定的集合(mem32,mem64,mem80)之中,便调用 Memory( )异常程序。

3.2.2 函数

在指令集说明中使用的一些预定义的过程如下:

(1) B(bitN,*op*){

```
    return((op>>bitN)&01b);
    }
```

说明:返回操作数 *op* 的第 N 位。

(2) Bits(bitN,num,*op*){

```
    for(mask=0,i=0;i<num;i++){
```

```
            mask=(mask<<1)| 1;
            }
            return((op>>bitN)& mask);
        }
```

说明：返回操作数 *op* 的从第 N 位开始的 num 个位。

(3) CarryO(bitN){

```
        return((Carrychain>>bit N)&01b);
        }
```

说明：返回最后一次 ALU 运算产生的进位码链中的第 N 位。

这里假定最后一次加法产生的进位码被保持在专用变量 Carry Chain 之中。

(4) BorrowO(bitN){

```
        return((BorrowChain>>bitN)&01b);
    }
```

说明：返回最后一次 ALU 运算产生的借位码中的第 N 位。

假定最后一次减法产生的借位码保持在专用变量 Borrow Chain 之中。

(5) Length(*op*){

```
        return(sizeof(op) * 8);
        }
```

说明：返回操作数长度(以位为单位)。C 函数 sizeof 返回给定数据类型的长度(以字节为单位)。

(6) SignEx(DesLen,OpLen,*op*){

```
        sign=B(OpLen,op);
        tmp=op;
        for(i=OpLen;i>DesLen;i++){
            tmp=(sign<<i)|tmp
            }
              return(tmp);
            }
```

说明：将原长为 OpLen 的操作数 *op*、符号扩展为 DesLen 规定的长度。

(7) ConCat(lenOfRes,op_2,lenOfop_1,op_1){

```
          mask=0;
          for(i=0;i<lenOfRes;i++){
              mask=(1<<i)|mask;
              }
              return(mask&((op2<<lenOfop1)|op1));
        }
```

说明：将 op_1 串接在 op_2 之后。lenOfRes 为串接后的长度，lenOfop_1 为 op_1 的长度。lenOfRes与 lenOfop_1 之差为 op_2 的长度。

(8) Push(*op*){

```
          len=Length(op)/8;
```

```
        SS：[ESP－len]=op；
        ESP=ESP－len；
    }
```

说明：ESP 为堆栈指针寄存器。该函数将操作数 *op* 压入栈内，作为新的栈顶元素。ESP 指向新的栈顶。

```
(9) Pop(op){
        len=Length(op)/8；
        tmp=SS：[ESP]；
        ESP=ESP+len；
        return(tmp)；
    }
```

说明：ESP 为堆栈指针寄存器。该函数使栈顶元素弹出，ESP 指向新的栈顶；*op* 仅用来决定操作数长度。

```
(10) Port(op){
            if(OPCODE in(OUT,OUTS)){
                AccessIO(op,Length(op)/8,&op2,1)；
      }          else {   /* 从端口读入 */
            AccessIO(op,Length(op)/8,&tmp,0)；
            return( * tmp)；
        }
    }
```

说明：此 Port 程序执行端口的输入输出。它只是 AccessIO( )程序的简单调用。

AccessIO( )的用法为：

AccessIO (端口号，操作之长度(以字节为单位，数据地址，1 (表示“写”)或0(表示“读”))；

AccessIO( )函数在第5章中定义。请详阅第5章。

```
(11) SegReg(op){
      if( op has Segment override){
          return(segment of op)
        }else {
          return(DS)；
        }
    }
```

说明：串操作指令根据 *op* 的段超越前缀确定哪个段用于引用 ESI 串元素。默认的 *op* 段超越前缀为 DS。

```
(12) StopExecution( ){
          while(1)；
    }
```

说明：执行一永无休止的空操作循环。

3.3 指令的分类与摘要

本节将指令按功能详细分类，并摘要列出各个指令的功能。

3.3.1 整数类指令的分类

3.3.1.1 整数类指令又按其功能分为下列10个子类：

(1) 算术运算类：符号整数，无符号整数，ASCII 和 BCD 数据类型的操作。

(2) 位操作：位数据类型的操作。

(3) 条件赋值：以标志状态为条件给字节赋值。

(4) 控制转移：条件的和无条件的控制转移。

(5) 数据传送：数据在内存，堆栈，端口和寄存器之间的传送操作。

(6) 标志控制：改变标志位，并使标志位成组地在 AH 寄存器和堆栈之间传送。

(7) 高级语言：实现某些高级语言操作。

(8) 逻辑。

(9) 串操作：串数据类型的操作。

(10) 其它。

3.3.1.2 整数类指令摘要

指令	摘　要
(1) 算术运算类指令	
AAA	加法之后的 ASCII 调整
AAD	除法之前的 ASCII 调整
AAM	乘法之后的 ASCII 调整
AAS	减法之后的 ASCII 调整
ADC	带进位的加法
ADD	整数加法
CBW/CWDE	将字节转换成字/将字转换成双字
CMP	比较
CWD/CDQ	将字转换成双字/将双字转换成四字
DAA	加法之后的十进制调整
DAS	减法之后的十进制调整
DEC	减量
DIV	无符号除
IDIV	符号除
IMUL	符号乘
INC	增量
MUL	无符号乘
NEG	二进制补码
SBB	带借位的减法
SUB	减

指令	摘　要
(2) 位操作类指令	
BSF	前向位扫描
BSR	后向位扫描
BT	位测试
BTC	位测试并求反
BTR	位测试并复位
BTS	位测试并置位
(3) 条件赋值类指令	
SETcc	根据条件码设置字节
(4) 控制转移类指令	
CALL	调用过程
Jcc	条件跳转
JMP	跳转
LOOP	用 ECX 计数器的循环
LOOPNZ/LOOPNE	用 ECX 计数器且不为0的循环/用 ECX 计数器且不等的循环
LOOPZ/LOOPE	用 ECX 计数器且为0的循环/用 ECX 计数器且相等的循环
RET	返回
(5) 数据传送类指令	
IN	从端口输入
LEA	装入有效地址
MOV	传送
MOVSX	传送和符号扩展
MOVZX	传送和零扩展
OUT	向端口输出
POP	从堆栈中弹出
POPA/POPAD	从栈弹出至所有通用寄存器
PUSH	压栈
PUSHA/PUSHAD	所有通用寄存器压栈
XCHG	交换
(6) 标志控制类指令	
CLC	清0进位标志
CLD	清0方向标志
CLI	清0中断标志
CMC	求进位标志的反码
LAHF	将标志装入 AH 寄存器
POPF/POPFD	从栈中弹出至标志位
PUSHF/PUSHFD	将标志压栈
SAHF	将 AF 寄存器存入标志寄存器
STC	置进位标志
STD	置方向标志
STI	置中断标志
(7) 高级语言类指令	
BOUND	检查数组索引是否违反边界

指令	摘　要
ENTER	建立堆栈框架
LEAVE	过程退出
(8) 逻辑类指令	
AND	与
NOT	非
OR	或
RCL	带 CF 的左旋
RCR	带 CF 的右旋
ROL	左旋
ROR	右旋
SAL/SHL	算术左移/逻辑左移
SAR	算术右移
SHLD	双精度左移
SHR	逻辑右移
SHRD	双精度右移
TEST	逻辑比较
XOR	异或
(9) 串操作类指令	
CMPS/CMPSB/CMPSW/CMPSD	比较串操作数
INS/INSB/INSW/INSD	输入串操作数
LODS/LODSB/LODSW/LODSD	装入串操作数
MOVS/MOVSB/MOVSW/MOVSD	传送串操作数
OUTS/OUTSB/OUTSW/OUTSD	输出串操作数
REP	重复
REPE/REPZ	相等时重复/为0时重复
REPNE/REPNZ	不等时重复/非0时重复
SCAS/SCASB/SCASW/SCASD	扫描串操作数
STOS/STOSB/STOSW/STOSD	存储串操作数
XLAT/XLATB	查表变换
(10) 其它类型指令	
LOCK	总线锁定
NOP	空操作

3.3.2 多段类指令摘要

指令	摘　要
CALL	过程调用
INT	中断过程的调用
INTO	溢出中断过程的调用
IRET	中断返回
JMP	跳转
LDS	将指针装入 DS
LES	将指针装入 ES
LFS	将指针装入 FS

指令	摘　要
LGS	将指针装入 GS
LSS	将指针装入 SS
MOV	段寄存器的传送
POP	从栈弹出至段寄存器
PUSH	压栈
RET	返回

3.3.3 操作系统类指令摘要

指令	摘　要
ARPL	调整请求特权级
CLTS	清任务切换标志
HLT	暂停
LAR	加载访问权
LGDT	加载全局描述符表
LIDT	加载中断描述符表
LLDT	加载局部描述符表
LMSW	加载机器状态字
LSL	加载段界限
LTR	加载任务寄存器
MOV	特殊寄存器的数据传送
SGDT	存储全局描述符表
SIDT	存储中断描述符表
SLDT	存储局部描述符表
SMSW	存储机器状态字
STR	存储任务寄存器
VERR	为读操作检验段
VERW	为写操作检验段

3.3.4 浮点类指令摘要

指令	摘　要
(1) 数据传送指令	
FBLP	BCD　装入
FBSTP	BCD　存储并出栈
FILD	整数装入
FIST	整数存储
FISTP	整数存储并出栈
FLD	实型数装入
FST	实型数存储
FSTP	实型数存储并出栈
FXCH	交换寄存器
(2) 算术运算指令	
FABS	绝对值

指令	摘　要
FADD/FIADD/FADDP	加
FCHS	改变符号
FDIV/FIDIV/FDIVP	除
FDIVR/FIDIVR/FDIVRP	反除(Division Reverse)
FMUL/FIMUL/FMULP	乘
FPREM	部分余数
FPREM1	部分余数—IEEE
FRNDINT	舍入为整数
FSCALE	2次幂扫描
FSUB/FISUB/FSUBP	减
FSUBR/FISUBR/FSUBRP	减反
FSQRT	平方根
FXTRACT	摘取指数和有效数字
(3)比较指令	
FCOM/FCOMP/FCOMPP	比较
FICOM/FICOMP	整数比较
FTST	测试
FUCOM/FUCOMP/FUCOMPP	无序比较
FXAM	检验
(4) 超越函数指令	
F2XM1	2^x-1
FCOS	余弦
FPATAN	部分反正切
FPTAN	部分正切
FSIN	正弦
FSINCOS	正弦和余弦
FYL2X	$y\times\log_2 x$
FYL2XP1	$y\times\log_2(x+1)$

注意：FCOS,FPTAN,FSIN,和 FSINCOS 的操作数和结果用弧度为单位。

(5) 常数	
FLD1	装入1
FLDL2E	装入 $\log_2 C$
FLDL2T	装入 $\log_2 10$
FLDLG2	装入 $\log_{10} 2$
FLDLN2	装入 $\log_e 2$
FLDPI	装入 π
FLDZ	装入0
(6) 控制指令	
FCLEX/FNCLEX	清异常
FDECSTP	堆栈指针减
FFREE	自由寄存器
FINCSTP	堆栈指针加一
FINIT/FNINIT	初始化处理器

指令	摘　要
FLDCW	装入控制字
FLDENV	装入环境
FNOP	空操作
FRSTOR	恢复状态
FSAVE/FNSAVE	保存状态
FSTCW/FNSTCW	存控制字
FSTENV/FNSTENV	存环境
FSTSW/FNSTSW	存状态字
FSTSWAX/FNSWAX	将状态字存入 AX
WAIT	等待协处理器

3.4 整数类指令集

本节给出了应用程序编程用的全部指令。每条指令都含有一到二个例子，以说明它们如何使用。第6章给出由这些指令组成的能完成特定任务的程序段的更多更复杂的例子。通过这些例子和程序段，读者应能较深地理解这些指令的目的和操作。

AAA

加法之后的 ASCII 调整

格式：

AAA　　0 0 1 1 0 1 1 1

说明：

AAA 指令将在字节加法之后，对 AL 寄存器的内容进行非压缩的 BCD 调整。通常，AAA 用在以 AL 寄存器的内容作为非压缩 BCD 码进行字节加法之后，但也能够用于其它 BCD 转换。

如果加法产生了十进制的进位，或者 AL 寄存器的低半字节的值大于9，则 AL 的值增加6，AH 的值增加1，CF 和 AF 置位。如果加法没产生十进制进位，那么 AH 中的值不变，AL 中低四位的值也不变，CF 和 AF 清0。无论发生或不发生进位，AL 中的高四位都被清0。

操作：

```
if((AL&0Fh)>9)||(AF==1){
    AL=AL+6;
    AH=AH+1;
    AF=1;
    CF=1;
} else {
    AF=0;
    CF=0;
}
```

AL=AL&0Fh;

OF=SF=ZF=PF=UNDEFINED;

例子：

	OSZAPC
ADD AL,BL	
执行前：AH:05 AL:07 BL:09	××××××
执行后：AH:05 AL:10 BL:09C	000100
AAA	
执行前：AH:05 AL:10 BL:09	000100
执行后：AH:06 AL:06 BL:09	×××1×1
ADD AL,BL	
执行前：AH:05 AL:07 BL:02	××××××
执行后：AH:05 AL:09 BL:02	000010
AAA	
执行前：AH:05 AL:09 BL:02	000010
执行后：AH:05 AL:09 BL:02	×××0×0

时钟：

4个时钟周期；在20MHz 情况下为0.2μs

标志影响：

AF 和 CF 如上所述；OF,SF,ZF 和 PF 为未定义。其它标志位不变。

AAD

除法之前的 ASCII 调整

格式：

AAD 11010101 00001010

说明：

在两个非压缩的 BCD 数进行除法之前，用 AAD 指令把在 AX 中的被除数调整至 AL 中。调整的结果，将使除法产生的商成为有效的 BCD 码。

AL 寄存器的值调整为 AL+(10*AH)，AH 寄存器的值调整为0。后续的除操作将有效 BCD 类型的商留在 AL 之中，而将余数留在 AH 之中。

操作：

AL=AL+(10*AH);

AH=0;

SF=B(7,AL);

ZF=AL==0;

PF=~(B(7,AL)^B(6,AL)^…^B(0,AL));

AF=OF=CF=UNDEFINED;

例子：

AAD　　OSZAPC

执行前：AH:05　AL:07　BL:09　　×××××××

执行后：AH:00　AL:39　BL:09　　×00×1×

DIV BL

执行前：AH:00　AL:39　BL:09　　×00×1×

执行后：AH:03　AL:06　BL:09　　×××1×1

时钟：

19个时钟周期；在20MHz 情况为0.95μs。

标志影响：

影响 SF，ZF 和 PF。OF，AF 和 CF 为未定义的。其它标志位不受影响。

AAM

乘法之后的 ASCII 调整

格式：

AAM

1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0

说明：

两个 BCD 数相乘(MUL)可能产生一无效的 BCD 结果。AAM 将此结果转成一对 BCD 数，并将它们分别留在 AH 寄存器和 AL 寄存器(AL 和 AH 合成一个 AX)中。

通常，AAM 指令紧随两个 BCD 数的乘法之后。AL 的值除以10的商放在 AH 中，AL 中原来的值除以10的模作为新的值被留在 AL 之中。

操作：

AH=AL/10;

AL=AL%10;

SF=B(7,AL);

ZF=AL==0;

PF=~(B(7,AL)^B(6,AL)^…^B(0,AL));

OF=AF=CF=UNDEFINED;

例子：

MUL BL　　OSZAPC

执行前：AH:00　AL:07　BL:09　　××××××

执行后：AH:00　AL:3F　BL:09　　0××××0

AAM

执行前：AH:00　AL:3F　BL:09　　0××××0

执行后：AH:06　AL:03　BL:09　　×00×1×

时钟：

17个时钟周期；在20MHz 情况下为0.85μs。

标志影响：

影响 SF，ZF 和 PF。OF，AF 和 CF 为未定义的。其它标志不变。

AAS

减法之后的 ASCII 调整

格式：

AAS

0	0	1	1	1	1	1	1

说明：

紧接着字节的减操作，AAS 将 AL 寄存器的内容进行一次非压缩的 BCD 调整。AAS 仅能用于 AL 作为非压缩的内容进行的字节减操作之后。

若减操作发生十进制借位，AL 的内容便减去6，AH 的内容被减掉1，CF 和 AF 置位。若不发生十进制借位，AH 的值不变，AL 的低四位也不变，CF 和 AF 清0。无论发生或不发生借位，AL 的高4位总被清0。

操作：

```
if((AL&0Fh)>9)||(AF==1){
    AL=AL-6;
    AH=AH-1;
    AF=1;
    CF=1;
 }else {
    AF=0;
    CF=0;
}
AL=AL & 0Fh;
OF, SF, ZF=PF=UNDEFINED;
```

例子：

	O S Z A P C
SUB AL,BL	
执行前：AH:05 AL:07 BL:09	× × × × × ×
执行后：AH:05 AL:FE BL:09	0 1 0 1 0 1
AAS	
执行前：AH:05 AL:FE BL:09	0 1 0 1 0 1
执行后：AH:04 AL:08 BL:09	× × × 1 × 1
SUB AL,BL	
执行前：AH:05 AL:09 BL:07	× × × × × ×
执行后：AH:05 AL:02 BL:07	0 0 0 0 0 0
AAS	
执行前：AH:05 AL:02 BL:07	0 0 0 0 0 0
执行后：AH:05 AL:02 BL:07	× × × 0 × 0

时钟：

4个时钟周期；在20MHz 情况下为0.2μs。

标志影响：

AF，CF 如上所述；OF，SF，ZF 和 PF 为未定义。其它标志位不变。

ADC

带进位加法

格式：

ADC op_1 , op_2	字节 0	字节 1	后续字节
reg , reg	0 0 0 1 0 0 d w	mod: reg : r/m	
mem , reg	0 0 0 1 0 0 0 w	mod: reg : r/m	
reg , mem	0 0 0 1 0 0 1 w	mod: reg : r/m	
reg , imm	1 0 0 0 0 0 s w	mod:0 1 0 : r/m	立即数字节
mem , imm	1 0 0 1 0 0 s w	mod:0 1 0 : r/m	立即数字节
AL/AX/EAX, imm (短编码)	0 0 0 1 0 1 0 w	立 即 数	字 节

说明：

op_1、op_2和 CF 整数相加，结果放入 op_1中。ADC 通常用作多字节、多字、或多双字加法的组成部分。

操作：

$op_1 = op_1 + op_2 + CF$;

OF=CarryO(MSB)^CarryO(MSB-1);

SF=B(MSB,op_1);

ZF=(op_1==0);

AF=CarryO(3);

PF=~(B(7,op_1)^B(6,op_1)^…^B(0,op_1));

CF=CarryO(MSB);

/*

下面的一张表定义了上述语法中的"+"操作。"+"操作必须从0位到 MSB 逐位进行。

a_i	b_i	c_{i-1}	r_i	c_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

表中 a_i 和 b_i 分别为 op_1和 op_2的第 i 位。c_{i-1}是第 i−1位的加法产生的进位，它参与了第 i 位的

运算。第 i 位的运算产生第 i 位的结果(r_i)和进位(c_i)。位下标 i 取值范围自0至 MSB。若 i=0，$c_{i-1}=c_{-1}$=CF，即 c_{-1}取执行 ADC 指令之前的 CF 值。

*/

例子：

ADC EAX,EBX	OSZAPC
执行前：EAX:01234567 EBX:76543210	×××××1
执行后：EAX:777777778 EBX:76543210	000010
ADC mem32,70000000h	OSZAPC
执行前：Mem:7FFFFFFF	×××××0
执行后：Mem:EFFFFFFF	110010
ADC AL,AH	OSZAPC
执行前：AL:F9 AH:65	×××××1
执行后：AL:5F AH:65	000011

时钟：(假设主频为20MHz)

ADC reg,reg：2个时钟周期；0.1μs。

ADC mem,reg：7个时钟周期；0.35μs。

ADC reg,mem：6个时钟周期；0.3μs。

ADC reg,imm：2个时钟周期；0.1μs。

ADC mem,imm：7个时钟周期；0.35μs。

标志影响：

影响 OF、CF、SF、ZF、AF 和 PF。其它标志位不变。

异常：

Memory( )；

实地址模式异常：

若内存操作数的任何部分是在 CS、DS、ES、FS、或 GS 段中大于 OFFFFh 的有效地址上，便产生中断13。若内存操作数的任何部分是在堆栈段 SS 中大于 OFFFFh 的有效地址上，则产生中断12。

保护模式异常：

对于在内存 CS、DS、ES、FS、或 GS 段中的操作数，若目标在一不可写的段内，或者源操作数是在一不可读的代码段中，或者操作数的任何部分是在超出了段界的有效地址上，则产生错误码为0000h 的通用保护异常(异常13)。对于在堆栈段内的操作数，若操作数的任何部分是在超出段界的有效地址上，则产生错误码为0000h 的堆栈异常(异常12)。若内存操作数是在标为“未出现”的 SS 段内，则产生错误码为段选择符的堆栈异常。若内存操作数是在标为“未出现”的 DS、ES、FS、或 GS 段内，则产生错误码为段选择符的“未出现”异常。页故障引起错误码为故障码的页故障线性地址在寄存器 CR2内的页故障异常。

虚拟8086模式异常：

同实地址模式异常。但异常的处理是在保护模式，且特权级为0的条件下进行的。页故障异常同保护模式下的页故障异常。

ADD

整数加法

格式：

ADD op_1 , op_2	字节 0	字节 1	后续字节
reg , reg	0 0 0 0 0 0 d w	mod reg r/m	
mem , reg	0 0 0 0 0 0 0 w	mod reg r/m	
reg , mem	0 0 0 0 0 0 1 w	mod reg r/m	
reg , imm	1 0 0 0 0 0 s w	mod 0 0 0 r/m	立即数字节
mem , imm	1 0 0 0 0 0 s w	mod 0 0 0 r/m	立即数字节
AL/AX/EAX , imm (短编码)	0 0 0 0 0 1 0 w	立 即 数	字 节

说明：

op_1与op_2整数相加，结果放入op_1中。

操作：

```
op1=op1+op2;
OF=CarryO(MSB)^CarryO(MSB-1);
SF=B(MSB,op1);
ZF=op1==0;
AF=CarryO(3);
PF=~(B(7,op1)^B(6,op1)^…^B(0,op1));
CF=CarryO(MSB)
/*
上述语法中的"+"操作同 ADC 指令所列表。"+"操作必须从0位到 MSB 逐位进行。表中 ai
和 bi 分别为 op1和 op2的第 i 位。ci-1是第 i-1位的加法产生的进位，它参与了第 i 位的运
算。第 i 位的运算产生第 i 位的结果(ri)和进位(ci)。位下标 i 取值范围从0到 MSB。若 i=
0，ci-1=c-1=0，即 c-1之值永远为0。
*/
```

例子：

ADD EAX,EBX	O S Z A P C
执行前：EAX:01234567 EBX:76543210	× × × × × ×
执行后：EAX:77777777 EBX:76543210	0 0 0 0 1 0
ADD mem32 ,70000000h	O S Z A P C
执行前：Mem:7FFFFFFF	× × × × × ×
执行后：Mem:EFFFFFFF	1 1 0 0 1 0
ADD AL,AH	O S Z A P C
执行前：AL:F9 AH:65	× × × × × ×
执行后：AL:5E AH:65	0 0 0 0 0 1

时钟：(假设主频为20MHz)

ADD　reg,reg:2个时钟周期;0.1μs。

ADD　mem,reg:7个时钟周期;0.35 μs。

ADD　reg,mem:6个时钟周期;0.3 μs

ADD　reg,imm:2个时钟周期;0.1 μs

ADD　mem,imm,:7个时钟周期;0.35 μs

标志影响:

影响OF,CF,SF,ZF,AF和PF。

其它标志位不变。

异常:

Memory( );

实地址模式异常:同ADC指令。

保护模式异常:同ADC指令。

虚拟8086模式:同ADC指令。

AND

与

格式

AND op_1 , op_2	字节0	字节1	后续字节
reg , reg	0 0 1 0 0 0 d w	mod: reg : r/m	
mem , reg	0 0 1 0 0 0 0 w	mod: reg : r/m	
reg , mem	0 0 1 0 0 0 1 w	mod: reg : r/m	
reg , imm	1 0 0 0 0 0 s w	mod:1 0 0: r/m	立即数字节
mem , imm	1 0 0 0 0 0 s w	mod:1 0 0: r/m	立即数字节
AL/AX/EAX , imm (短编码)	0 0 1 0 0 1 0 w	立 即 数	字 节

说明:

执行op_1和op_2之间的逻辑与。执行结果放入op_1。

操作:

$op_1=op_1$ & op_2;

OF=0;

SF=B(MSB,op_1);

ZF=(op_1==0);

AF=UNDEFINED;

PF=~(B(7,op_1)^B(6,op_1)^…^B(0,op_1));

CF=0;

/*

下面一张表定义了上述语法中的"&"操作。"&"操作必须从0到MSB位逐位进行。

a_i	b_i	r_i
0	0	0
0	1	0
1	0	0
1	1	1

表中 a_i 和 b_i 分别为 op_1 和 op_2 的第 i 位。r_i 为结果的第 i 位。位下标 i 的取值范围为从0到MSB。

*/

例子：

```
AND  EAX,EBX                                        OSZAPC
    执行前：EAX:01234567  EBX:76543210              ××××××
    执行后：EAX:00000000  EBX:76543210              001×10
AND mem8,70h                                        OSZAPC
    执行前：Mem:7F                                  ×××××××
    执行后：Mem:70                                  000×00
AND  AX,DI                                          OSZAPC
    执行前：AX:A5A5  DI:FFFF                        ×××××××
    执行后：AX:A5A5  DI:FFFF                        010×10
```

时钟：(假设主频为20MHz)

AND reg,reg：2个时钟周期；0.1μs。

AND mem,reg：7个时钟周期；0.35μs。

AND Reg,mem：6个时钟周期；0.3μs。

AND Reg,imm：2个时钟周期；0.1μs。

AND mem,imm：7个时钟周期；0.35μs。

标志影响：

CF=0；OF=0。SF，ZF 和 PF 的影响如上所述。

其它标志位不变。

异常：

Memory( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：同 ADC 指令。

BOUND

检查数组索引是否超越边界

格式：

BOUND op_1 , op_2

reg16 , mem32

reg32 , mem64

字节 0	字节 1
0 1 1 0 0 0 1 0	mod: reg : r/m

说明：

op_1必须是一个寄存器，其中存放的是一个符号数组索引。将op_1的内容与op_2中所给的低的和高的边界数据结构相比较。如果数组索引不在高低边界范围之内，便产生第五类异常。

这里假定边界数据结构存放在两个相邻的操作数(是字，或是双字)内。第一个操作数为边界下限(lower)，第二操作数为边界上限(upper)。

操作：

```
if((op1<op2)||(op1>[&.op2+sizeof(op1)])){
    Interrupt(5);
}
```

例子：

BOUND EAX,mem 64

执行前：EAX :000092FD mem:00000000

[&.mem+4] :00000064

执行后:Interrupt(5)

时钟：(假设主频为20MHz)

若未发生中断:10个时钟周期;0.5μs。

若产生中断5:44个时钟周期;2.2μs。

标志影响：

无。

异常：

Interrupt(5);

Memory( );

实地址模式异常:如果边界测试失败则产生中断5。其它中断同 ADC 指令。

保护模式异常:如果边界测试失败则产生中断5。其它异常同 ADC 指令。

虚拟8086模式异常:如果边界测试失败则产生中断5。其它异常同 ADC 指令。

BSF

前向位扫描

格式：

BSF op_1 , op_2

reg16 , reg16

reg16 , mem16

reg32 , reg32

reg32 , mem32

字节 0	字节 1	字节 2
0 0 0 0 1 1 1 1	1 0 1 1 1 1 0 0	mod: reg : r/m

说明：

自右至左地扫描由 op_2 指定的字或双字(从0位向15位或31位),直至遇到第一个内容为1的位。将该位的索引放入 op_1。

若整个的字或双字为全0,则 ZF 位置1,op_1 为未定义的。若找出一个内容为1的位,ZF 位即复位。

操作:

```
if(op2==0){
    ZF=1;
    op1=UNDEFINED;
} else {  /*  op2不为全0则扫描 */
    ZF=0;
    temp=0;
    while (B(temp, op2)==0){
        temp=temp+1;
    }
  op1=temp;
}
OF=SF=AF=PF=CF=UNDEFINED;
```

例子:

BSF EAX,EBX	O S Z A P C
执行前: EAX:xxxxxxxx EBX:76543210	x x x x x x
执行后: EAX:00000004 EBX:76543210	x x 0 x x x
BSF SI,mem16	O S Z A P C
执行前: SI:xxxx mem:0000	x x x x x x
执行后: SI:xxxx mem:0000	x x 1 x x x

时钟:(假设主频为20MHz)

(10+3n)个时钟周期:(0.5+0.15n)μs。

其中 n 为自右至左地扫描,直至遇到第一个内容为1的位时扫过的位数。

标志影响:

影响 ZF。

其它标志不变。

异常:

Memory( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟8086模式异常:同 ADC 指令。

注意:

该指令是80386所增的。

BSR

反 向 扫 描

格式:

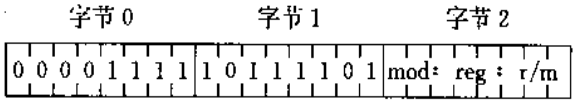
BSR op_1 , op_2

reg16 , reg16

reg16 , mem16

reg32 , reg32

reg32 , mem32



说明：

自左至右地扫描由 op_2 指定的字或双字(从15位或31位，向0位)，直至遇到第一个内容为1的位。将该位的索引放入 op_1。

若整个的字或双字为全0，则 ZF 置1，op_1 为未定义的。若找出一个内容为1的位，ZF 即复位。

操作：

```
if(op2==0) {
    ZF=1;
    op1=UNDEFINED;
} else {   /* op2不为全0则扫描 */
    ZF=0;
    temp=Length(op2)-1;
    while(B(temp,op2)==0) {
        temp=temp-1;
    }
    op1=temp;
}
OF=SF=AF=PF=CF=UNDEFINED;
```

例子：

BSR BX,CX　　OSZAPC

执行前：BX:××××　　CX:0030　　××××××

执行后：BX:0005　　CX:0030　　××0×××

BSR ESR,mem32　　OSZAPC

执行前：ESI:××××××××　mem:000F0000　　××××××

执行后：ESI:00000013　mem:000F0000　　××0×××

时钟：(假设主频为20MHz)

(10+3n)个时钟周期；(0.5+0.15n)μs。

其中 n 为找到第一个内容为1的位时已扫过的位数。

标志影响：

影响 ZF。

其它位不受影响。

异常：

Memory( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：同 ADC 指令。

注意：

该指令是80386新增的。

BT

位 测 试

格式：

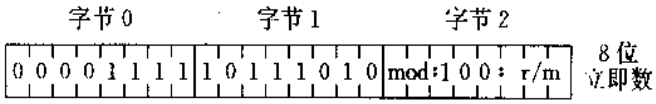
BT op_1 , op_2

reg16 , imm

mem16 , imm

reg32 , imm

mem32 , imm

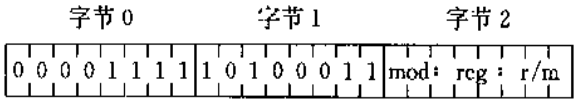


reg16 , reg16

mem16 , reg16

reg32 , reg32

mem32 , reg32



说明：

通过 op_2，在 op_1 中选择一位，将该位的值放入进位标志 CF 之中。

op_2 取符号整数值。于是 op_2 为16位可访问(−32K)到(32K−1)范围内的位串，op_2 为32位时可访问(−2G)到(2G−1)范围内的位串。

若 op_1 是个寄存器，待选位由 op_2 对于寄存器长度的模数 K 确定。若 op_1 是个内存位串，首先用 op_1 的内存地址，加上 op_2(位索引)除以8的商(变成3字节偏移量)，求出待选位所在的字或双字的地址；然后再由 op_2 对操作数长度(16或32)的模数来确定待选位在字或双字中的位置。

操作：

```
offset=op2% Length(op1);
disp=op2/8;
if(op1 in(reg16, reg32)) {
    CF=B(offset,op1);
} else {  /* 内存操作数 */
    CF=B(offset, [&op1+disp]);
}
OF=SF=ZF=AF=PF=UNDEFINED;
```

例子：

```
BT   BX,CX                                          OSZAPC
     执行前：BX:0A50        CX:4218                 ××××××
     执行后：BX:0A50        CX:4218                 ×××××0
BT   mem,EAX
```

执行前：EAX　　:004A6B48　　×××××××

[&.mem+04A6B48/8]:54BBD231

执行后：EAX　　:004A6B48　　×××××0

[&.mem+04A6B48/8]:54BBD231

时钟：(假设主频为20MHz)

BT　reg,imm:3个时钟周期;0.15μs。

BT　mem,imm:6个时钟周期;0.3μs。

BT　reg,reg:3个时钟周期;0.15μs。

BT　mem,reg:12个时钟周期;0.6μs。

标志影响：

影响CF。

其它标志不变。

异常：

Memory( );

实地址模式异常:同ADC指令。

保护模式异常:同ADC指令。

虚拟8086模式异常:同ADC指令。

注意：

这条指令是80386新增的。

如果op_1是一个内存操作数,处理器可以存取内存中的以

&mem+disp

为起始地址的两个字节或四个字节。

因此编程者必须十分小心,不要在非法的地址空间附近使用BT指令。特别是这条指令最好别用于直接引用存储映象的I/O寄存器。在这种情况下,我们建议使用MOV指令来实现存储映象设备寄存器之间的装入(Load)和存储(Store),并且建议使用BT的寄存器格式来操作已装入处理器寄存器的设备寄存器映象。

如果op_2是个立即数,BT指令只提供8位的立即数字段,而且这个立即数必须小于操作数长度(32或16位),否则将产生未定义的位偏移量。这个规则允许规定在一个寄存器内的任何位偏移量,而且将存储器位串中的立即数位偏移量限制在规定存储单元双字(或字)之内。内存位串中的更大的立即数位偏移量是通过下述方法实现的:汇编程序将该立即数位偏移量的低5位(对于32位操作数)或低4位(对于16位操作数)放入op_2的立即数字段;该立即数位偏移量被汇编程序除以8(右移三位),变成了字节偏移量,与op_1的地址相加,形成新的op_1字节地址。详见本指令的《操作》部分。

BTC

位测试并求反

格式：

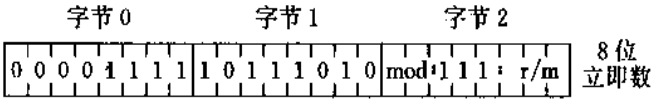
BTC op_1 , op_2

reg16 , imm

mem16 , imm

reg32 , imm

mem32 , imm

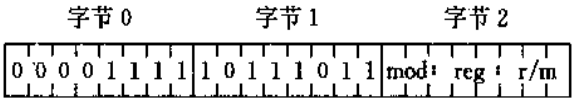


reg16 , reg16

mem16 , reg16

reg32 , reg32

mem32 , reg32



说明：

通过 op_2，在 op_1 中选择一位，将该位的值放入进位标志 CF 之中。在上述赋值操作完成后，op_1 中被选中的那位被求反。

op_2 取符号整数值。于是 op_2 为16位时可访问(−32K)到(32K−1)范围内的位串；op_2 为32位时可访问(−2G)到(2G−1)范围内的位串。

若 op_1 是个寄存器，待选位由 op_2 对于寄存器长度的模数来确定。若 op_1 是个内存位串，首先用 op_1 的内存地址，加上 op_2(位索引)除以8的商(变成字节偏移量)，求出待选位所在的字或双字的地址；然后再由 op_2 对操作数长度(16或32)的模数来确定待选位在字或双字中的位置。

操作：

```
offset=op2%Length(op1);
disp=op2/8;
if(op1 in(reg16, reg32)) {
    CF=B(offset, op1);
    B(offset,op1)=~B(offset,op1);
} else {  /*内存操作数  */
    CF=B(offset,[&.op1+disp]);
    B(offset,[&.op1+disp])=~B(offset,[&.op1+disp]);
}
OF=SF=ZF=AF=PF=UNDEFINED;
```

例子：

```
BTC  EAX,EDX                                   O S Z A P C
     执行前：EAX:0 A 5 0 FF 1 4   EDX:00A4CCD4   × × × × × ×
     执行后：EAX:0 A 4 0 FF 1 4   EDX:00A4CCD4   × × × × × 1
BTC  mem,CX                                    O S Z A P C
     执行前：CX                 :F 4 B 8        × × × × × ×
             [&.mem+F4B8/8]:D 2 3 1
     执行后：CX                 :F4B8           × × × × × 0
             [&.mem+F4B8/8]:D 3 3 1
```

时钟：(假设主频为20MHz)

BTC reg,imm：6个时钟周期；0.3μs。

BTC mem,imm：8个时钟周期；0.4μs。

BTC reg,reg：6个时钟周期；0.3μs。

BTC　mem,reg:13个时钟周期;0.65μs。

标志影响:

影响 CF。

其它标志不变。

异常:

Memory( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟8086模式异常:同 ADC 指令。

注意:

这条指令是80386新增的。

如果 op_1是一个内存操作数,处理器可以存取内存中的以

&mem+disp

为起始地址的两个字节或四个字节。

因此编程者必须十分小心,不要在非法的地址空间附近使用 BTC 指令。特别是这条指令最好别用于直接引用存储映象的 I/O 寄存器。在这种情况下,我们建议用 MOV 指令来实现存储映象设备寄存器之间的装入(Load)和存储(Store),并且建议使用 BTC 的寄存器格式来操作已装入处理器寄存器的设备寄存器映象。

如果 op_2是个立即数,BTC 指令只提供8位的立即数字段,而且这个立即数必须小于操作数长度(32或16位),否则将产生未定义的位偏移量。这个规则,允许规定在一个寄存器内的任何位偏移量,而且将存储器位串中的立即数位偏移量限制在规定存储单元双字(或字)之内。内存位串中的更大的立即数位偏移量是通过下述方法实现的:汇编程序将该立即数位偏移量的低5位(对于32位操作数)或低4位(对于16位操作数)作为 op_2的立即数字段;该立即数位偏移量被汇编程序除以8(右移三位),变成了字节偏移量,与 op_1的地址相加,形成新的 op_1字节地址。详见本指令的《操作》部分。

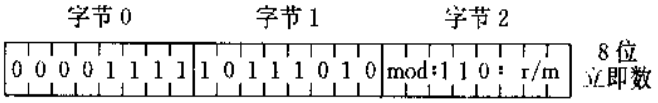
BTR

位测试并复位

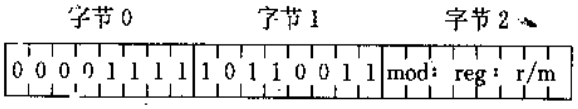
格式:

BTR　op_1　,　op_2

reg16　,　imm
mem16　,　imm
reg32　,　imm
mem32　,　imm



reg16　,　reg16
mem16　,　reg16
reg32　,　reg32
mem32　,　reg32



说明：

通过 op_2，在 op_1 中选择一位，将该位的值放入进位标志 CF 之中。在上述赋值操作完成后，op_1 中被选中的那位被复位为0。

op_2 取符号整数值。于是 op_1 为16位时可访问(−32K)到(32K−1)范围内的位串，op_2 为32位时可访问(−2G)到(2G−1)范围内的位串。

若 op_1 是个寄存器，待选位由 op_2 对于寄存器长度的模数来确定。若 op_1 是个内存位串，首先用 op_1 的内存地址，加上 op_2(位索引)除以8的商(变成了字节偏移量)，求出待选位所在的字或双字的地址；然后再由 op_2 对操作数长度(16或32)的模数来确定待选位在字或双字中的位置。

操作：

```
offset=op2%Length(op1)
disp=op2/8;
if(op1in(reg16,reg32)){
    CF=B(offset,op1);
    B(offset,op1)=0;
} else {   /*内存操作数*/
    CF=B(offset,[&op1+disp]);
    B(offset,[&op1+disp])=0;
OF=SF=ZF=AF=PF=UNDEFINED;
```

例子：

```
BTR  AX,23h                                    OSZAPC
    执行前：AX:53FE                            ××××××
    执行后：AX:53F6                            ×××××1
BTR  mem,ESI
    执行前：ESI                  :532011D4     ××××××
            [&mem+532011D4/8]    :FFFFFFFF
    执行后:ESI                   :532011D4     ××××××1
            [&mem+532011D4/8]    :FFEFFFFF
```

时钟：(假设主频为20MHz)

BTR reg,imm：6个时钟周期；0.3μs。

BTR mem,imm：8个时钟周期；0.4μs。

BTR reg,reg：6个时钟周期；0.3μs。

BTR mem,reg：13个时钟周期；0.65μs。

标志影响：

影响 CF。

其它标志不变。

异常：

Memory( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：同 ADC 指令。

注意：

这条指令是80386新增的。

如果 op_1 是一个内存操作数，处理器可以存取内存中的以

&mem+disp

为起始地址的两个字节或四个字节。

因此编程者必须十分小心，不要在非法的地址空间附近使用 BTR 指令。特别是这条指令最好别用于直接引用存储映象的 I/O 寄存器。在这种情况下，我们建议使用 MOV 指令来实现存储映象设备寄存器之间的装入(Load)和存储(Store)，并且建议使用 BTR 的寄存器格式来操作已装入处理器寄存器的设备寄存器映象。

如果 op_2 是个立即数，BTR 指令只提供8位的立即数字段，而且这个立即数必须小于操作数长度(32或16位)，否则将产生未定义的位偏移量。这个规则允许规定在一个寄存器内的任何位偏移量，而且将存储器位串中的立即数位偏移量限制在规定存储单元双字(或字)之内。内存位串中的更大的立即数位偏移量是通过下述方法实现的：汇编程序将该立即数位偏移量的低5位(对于32位操作数)或低4位(对于16位操作数)放入 op_2 的立即数字段；该立即数位偏移量被汇编程序除以8(右移三位)，变成了字节偏移量，与 op_1 的地址相加，形成了新的 op_1 字节地址。详见本指令的《操作》部分。

BTS

位测试并置位

格式：

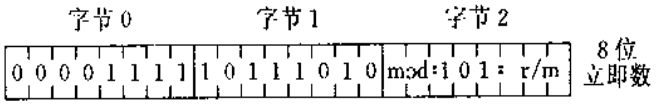
BTS op_1 , op_2

reg16 , imm

mem16 , imm

reg32 , imm

mem32 , imm

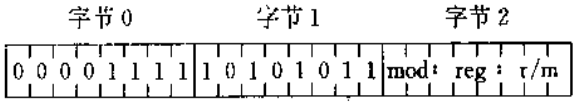


reg16 , reg16

mem16 , reg16

reg32 , reg32

mem32 , reg32



说明：

通过 op_2 在 op_1 中选择一位，将该位的值放入进位标志 CF 之中。在上述赋值操作完成之后，op_1 中被选中的那位被置位为1。

op_2 取符号整数值。于是 op_2 为16位时可访问(−32K)到(32K−1)范围内的位串，op_2 为32位时可访问(−2G)到(2G−1)范围内的位串。

若 op_1 是个寄存器，待选位由 op_2 对于寄存器长度的模数来确定。若 op_1 是个内存位串，首先用 op_1 的内存地址，加上 op_2(位索引)除以8的商(变成了字节偏移量)，求出待选位所在的字或

双字的地址；然后再由 op_2 对操作数长度(16或32)的模数来确定待选位在字或双字中的位置。

操作：

```
offset=op2%Length(op1);
disp=op2/8;
if(op1 in(reg16,reg32)){
    CF=B(offset, op1);
    B(offset, op1)=1;
} else {        /* 内存操作数 */
    CF=B(offset,[&op1+disp]);
    B(offset,[&op1+disp])=1;
}
OF=SF=ZF=AF=PF=UNDEFINED;
```

例子：

```
BTS  EAX,69h                                  OSZAPC
     执行前：EAX:0459B820                     ××××××
     执行后：EAX:0459BA20                     ×××××0
BTS  mem16,SI
     执行前：SI              :3014            ××××××
             [&mem+3014/8]   :0023
     执行后：SI              :3014            ×××××0
             [&mem+3014/8]   :0033
```

时钟：(假设主频为20MHz)

BTS reg,imm：6个时钟周期；0.3μs。

BTS mem,imm：8个时钟周期；0.4μs。

BTS reg,reg：6个时钟周期；0.3μs。

BTS mem,reg：13个时钟周期；0.65μs。

标志影响：

影响 CF。其它标志不变。

异常：

Memory( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：同 ADC 指令。

注意：

这条指令是80386新增的。

如果 op_1 是一个内存操作数，处理器可以存取内存中的以

&mem+disp

为起始地址的两个字节或四个字节。

因此编程者必须十分小心，不要在非法的地址空间附近使用 BTS 指令。特别是这条指令最好别用于直接引用存储映象的 I/O 寄存器。在这种情况下，我们建议使用 MOV 指令来实现和存储映象设备寄存器之间的装入(Load)和存储(Store)，并且建议使用 BTS 的寄存器格式

来操作已装入处理器寄存器的设备寄存器映象。

如果 op_2是个立即数，BTS 指令只提供8位立即数字段，而且这个立即数必须小于操作数长度(32或16位)，否则将产生未定义的位偏移量。这个规则允许规定在一个寄存器内的任何位偏移量，而且将存储器位串中的立即数位偏移量限制在规定存储单元双字(或字)之内。内存位串中的更大的立即数位偏移量是通过下述方法实现的：汇编程序将该立即数位偏移量的低5位(对于32位操作数)或低4位(对于16位操作数)放入 op_2的立即数字段；该立即数位偏移量被汇编程序除以8(右移三位)，变成了字节偏移量，与 op_1的地址相加，形成了新的 op_1字节地址。详见本指令的《操作》部分。

CALL

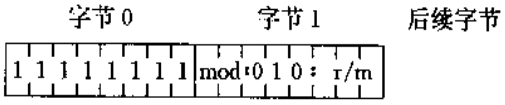
调用过程

格式：

CALL op_1

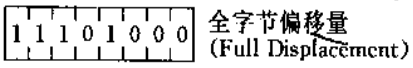
reg32

mem32



near-label

(近标号)



说明：

CALL 指令使得指令的执行在当前代码段的给定偏移量处继续下去。这里只讲在同一代码段内的调用。不同代码段之间的调用待在“多段”部分再作介绍。

段内调用的偏移量能由近(Near)标号、reg32和 mem32规定。mem32的含义为：偏移量由地址为 mem32的双字的内容规定。

当80386取出 CALL 指令后，指令指针已经指向了 CALL 指令的下一条指令，为了能在被调用的过程返回时进程能在这里继续，在控制转移发生之前先将此时的指令指针压入栈内。进程遇到 RET(返回)指令，将栈内信息弹出，这样便恢复到程序的原断点处继续执行。

在80386取出一条指令后，首先让指令指针 EIP 增长该条指令的长度，然后再执行这条指令。所以上面谈到在控制转移之前 EIP 可以直接入栈，而毋须再增长当前 CALL 指令的长度。

操作：

```
if(op1 is near-label) {
Push(EIP);
EIP=near-label; /* † */
} else if(op1 is reg32) {
    Push(EIP);
    EIP=reg32;
} else if(op1 is mem32) {
    Push(EIP);
    EIP=mem32;
}
/*
```

于邻近标号格式被汇编成相对于当前指令指针 EIP 的32位立即数位移量。于是在机内 EIP 修改为：
EIP=EIP+imm32
*/

例子：

CALL near-label1 ;在程序中，label1位于向前32字节处

执行前：EIP：300 ESP：200

执行后：EIP：332 ESP：1FC

SS：[ESP]：300

标志影响：

无。除非在保护状态下发生任务切换。

异常：

Memory( )；

Stack( )；

Code Reference( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：同 ADC 指令。

注意：

请阅第6章例3a。

CBW/CWDE

将字节转换成字/将字转换成双字

格式：

CBW

CWDE

说明：

把 AL 中的符号数按符号扩展为 AX。即把 AX 第7位的值赋予 AH 的每一位。或者，把 AX 中的符号数按符号扩展为 EAX。即把 EAX 第15位的值赋予 EAX 高16位的每一位。

操作：

```
if(OPCODE==CWDE) {
    EAX=Sign Ex(32,16,AX);
} else {    /*CBW   */
    AX=Sign Ex(16,8,AL);
}
```

例子：

CBW

执行前：AX：× × 7 F

执行后：AX：0 0 7 F

CWDE

执行前：EAX:××××FDF3

执行后：EAX:FFFFFDF3

时钟:(假设主频为20MHz)

3个时钟周期;0.15μs。

标志影响:

不影响任何标志位。

注意:

CWDE 和 CWD 的区别在于:CWD 使用 DX 和 AX 寄存器对,而 CWDE 用 EAX。

CLC

进位标志清0

格式:

CLC

1	1	1	1	1	0	0	0

说明:

进位标志清0。

操作:

CF=0;

例子:

CLC

执行前：CF:1

执行后：CF:0

时钟:(假设主频为20MHz)

2个时钟周期;0.1μs。

标志影响:

CF 被清0。其它标志不变。

CLD

方向标志清0

格式:

CLD

1	1	1	1	1	1	0	0

说明:

方向标志位(DF)清0。在方向标志位清0之后,串指令将增长它们的索引寄存器(ESI 和 EDI)。

操作:

DF=0;

例子：

CLD

执行前：DF:1

执行后：DF:0

时钟：(假设主频为20MHz)

2个时钟周期;0.1μs。

标志影响：

DF 清0。其它标志不变。

注意：

请阅第6章的例5。

CLI

中断标志清0

格式：

CLI

1	1	1	1	1	0	1	0

说明：

中断启用标志(IF)清0。从下一条指令执行完，一直到 IF 重新置位的期间，80386将不响应中断信号。

如果程序没有 I/O 特权，会引起异常(请阅第5章)。

操作：

IF=0;

例子：

CLI

执行前：IF:1

执行后：IF:0

时钟：(假设主频为20MHz)

3个时钟周期;0.15μs。

标志影响：

IF 清0。其它标志不变。

异常：

IOPLSensitive( );

实地址模式异常：无。

保护模式异常：

如果 CPL(当前特权级)在数值上大于 IOPL(I/O 特权级)则产生错误码为0000h 的通用保护异常(异常13)。

虚拟8086模式异常：

如果 IOPL 在数值上小于3(因为在虚拟8086模式下，CPL 总为3)，则产生错误码为0000h 的通用保护异常(异常13)。

在虚拟8086模式期间，如果 IOPL 的值小于3，捕获这条指令可以使保护模式的管理程序模拟这条指令。

CMC

进位标志位反码

格式：

CMC

1	1	1	1	0	1	0	1

说明：

进位标志位求反。

操作：

CF=~CF;

例子：

CMC

执行前：CF:1

执行后：CF:0

CMC

执行前：CF:0

执行后：CF:1

时钟：(假设主频为20MHz)

2个时钟周期；0.1μs。

标志影响：

CF 求反。其它标志不变。

CMP

比　较

格式：

CMP op_1 , op_2	字节 0	字节 1	后续字节
reg , reg	0 0 1 1 1 0 d w	mod: reg : r/m	
mem , reg	0 0 1 1 1 0 0 w	mod: reg : r/m	
reg , mem	0 0 1 1 1 0 1 w	mod: reg : r/m	
reg , imm	1 0 0 0 0 0 s w	mod:1 1 1 : r/m	立即数字节
mem , imm	1 0 0 0 0 0 s w	mod:1 1 1 : r/m	立即数字节
AL/AX/EAX,imm (短编码)	0 0 1 1 1 1 0 w	立　即　数	字　节

说明：

op_1减去op_2，但仅修改标志，而不保存相减的结果。常常在CMP指令后跟一个条件跳转指令或条件设置字节指令。

操作：

```
temp=op1 - op2;
/*"-"号视同SUB指令 */
OF=BorrowO(MSB)^BorrowO(MSB-1);
SF=B(MSB, temp);
ZF=temp==0;
AF=BorrowO(3);
PF=~(B(7,temp)^B(6,temp)^…^B(0,temp));
CF=BorrowO(MSB);
```

例子：

```
CMP   EAX,EBX                                   OSZAPC
      执行前：EAX:01234567   EBX:01234568        ××××××
      执行后：EAX:01234567   EBX:01234568        010111
JZ    EQUAL        ;没有跳转
CMP   AX,mem16                                  OSZAPC
      执行前：AX:0000   mem16:0000               ××××××
      执行后：AX:0000   mem16:0000               001010
JZ    EQUAL        ;跳转
```

时钟：(假设主频为20MHz)

```
CMP   reg,reg:2个时钟周期;0.1μs
CMP   reg,mem:5个时钟周期;0.25μs
CMP   mem,reg:6个时钟周期;0.3μs
CMP   reg,imm:2个时钟周期;0.1μs
CMP   mem,imm:6个时钟周期;0.3μs
```

标志影响：

影响OF、CF、SF、ZF、AF和PF。其它标志不变。

异常：

Memory( );

实地址模式异常：同ADC指令。

保护模式异常：同ADC指令。

虚拟8086模式异常：同ADC指令。

注意：

请看第6章中的例5。

CMPS/CMPSB/CMPSW/CMPSD

比较串操作数

格式：

CMPSB

CMPSW

CMPSD

CMPS mem1 , mem2

mem8 , mem8

mem16, mem16

mem32, mem32

说明：

CMPS 从[ESI]中减去 ES:[EDI]，但仅修改标志而不保留相减结果。mem_1及 mem_2，或者操作符 OPCODE 本身(CMPSB,CMPSW,CMPSD)，决定了操作数长度(8位，16位，或是32位)。上述减操作完成之后，ESI 和 EDI 根据操作数长度的不同，相应加减1，2，或4，以指向串中的下一个元素；若 DF 为0，则加，否则减。

CMPS 指令可与指令前缀 REPE/REPZ 或 REPNE/REPNZ 连用。若与 REPE/REPZ 连用，在 ECX 不为0且串元素相等(ZF==1)时，CMPS 指令重复执行。这样，CMPS 可用来找出两个串中存在的第一个不匹配元素。若与 REPNZ/REPNE 连用，当 ECX 不为0且串元素不等(ZF==0)时，CMPS 指令重复执行。用这种方法，CMPS 可以找出两个串中存在的第一个匹配元素。

操作数 mem_1和 mem_2，在汇编时仅被用来决定操作数长度。而被操作的串总是取自[ESI]和 ES:[EDI]。mem_2总是取自 ES:[EDI]。而 mem_1([ESI])可用不同的段超越前缀，而默认(缺省时)的段寄存器则为 DS。

操作：

```
if (OPCODE==CMPSB) {
  delta=1;
  temp=(byte)DS:[ESI]-(byte)ES:[EDI];
} else if (OPCODE==CMPSW) {
  delta=2;
  temp=(word)DS:[ESI]-(word)ES:[EDI];
} else if(OPCODE==CMPSD) {
  delta=4;
  temp=(dword)DS:[ESI]-(dword)ES:[EDI];
}else{                    /* CMPS */
/* 注意:默认的 SegReg(mem1)为 DS */
  delta=Length(mem1)/8;
  temp=Seg Reg(mem1):[ESI]-ES:[EDI];
}
/*"-"号视同 SUB 指令。*/
OF=BorrowO(MSB)^BorrowO(MSB-1);
SF=B(MSB,temp);
ZF=temp==0;
AF=BorrowO(3);
PF=~(B(7,temp)^B(6,temp)^…^B(0,temp));
CF=BorrowO(MSB);
if(DF==1) {
  delta=-delta;
}
ESI=ESI+delta;
```

EDI←EDI+delta;

例子:

```
CMPSB                                                    O S Z A P C
    执行前: ESI:0008  [ESI]:67  DF:0                      × × × × × ×
            EDI:0016  ES:[EDI]:65
    执行后: ESI:0009  [ESI-1]:67  DF:0                    0 0 0 0 1 0
            EDI:0017  ES:[EDI-1]:65
REPE  CMPS S1,S2  ;S1和 S2为16位串
                  ;[EDI]指向 S1,ES:[EDI]指向 S2           O S Z A P C
    执行前: ECX:4          DF:1
            ESI:0008       EDI:0016                       × × × × × ×
            [ESI]:0023     [EDI]:0023
            [ESI-2]:7923   [EDI-2]:7923
            [ESI-4]:7214   [EDI-4]:8215 /*第一个不匹配元素*/
            [ESI-6]:AA6D   [EDI-6]:AA6D
    执行后: ECX:1          DF:1                           0 0 1 0 1 0
            ESI:0002       EDI:0010
            [ESI+6]:0023   [EDI+6]:0023
            [ESI+4]:7923   [EDI+4]:7923
            [ESI+2]:7214   [EDI+2]:8215  /*第一个不匹配元素被找出。*/
            [ESI]:AA6D     [EDI]:AA6D
```

时钟:(假设主频为20MHz)

不重复: 10个时钟周期;0.5μs。

重复: (5+9n)个时钟周期;(0.25+0.45n)μs。

这里 n 为重复次数。

标志影响:

影响 OF,CF,SF,ZF,AF 和 PF。其它标志不变。

异常:

Memory( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟8086模式异常:同 ADC 指令。

注意:

请阅第6章中例5。

CWD/CDQ

将字转换成双字/将双字转换成四字

格式:

CWD

CDQ

1	0	0	1	1	0	0	1

1	0	1	0	0	1	1	w

说明：

CWD 把 AX 中的符号数按符号扩展为 DX:AX，即把 AX 的第15位的值赋予 DX 的每一位。CDQ 把 EAX 中的符号数按符号扩展为 EDX:EAX，即把 EAX 的第31位的值赋予 EDX 的每一位。

操作：

```
if(OPCODE==CDQ) {
    if(B(31,EAX)==1){
        EDX=FFFFFFFFh;
    } else {
        EDX=00000000h;
    }
} else {
    if(B(15,AX)    1){
        DX=FFFFh;
    } else {
        DX=0000h;
    }
}
```

例子：

CWD

执行前：AX:7615　DX:××××

执行后：AX:7615　DX:0000

CWD

执行前：AX:F103　DX:××××

执行后：AX:F103　DX:FFFF

时钟

2个时钟周期，若主频为20MHz，则为0.1μs。

标志影响：无。

注意：

CWD 和 CWDE 的区别在于：

CWD 使用 DX 和 AX 寄存器对，而 CWDE 则使用 EAX 寄存器。

DAA

加法之后的十进制调整

格式：

DAA

0	0	1	0	0	1	1	1

说明：

DAA 只能紧跟在操作两个压缩的 BCD 数(一个压缩字节)的，并将结果放在 AL 中的

ADD 指令后面使用。

DAA 将 AL 寄存器的内容转换成两位压缩的 BCD 数，结果仍存放在 AL 中。

若 AL 的低4位大于9，或者 AF(辅助进位标志)为1，则 AL 增长6，并非置 AF 为1。否则，AF 复位。若 AL 大于9Fh 或 CF 为1，则 AL 增加60h，并置 CF 为1。否则 CF 复位。

操作：

```
if(((AL&0Fh)>9)||AF) {
    AL=AL+6;
    CF  CF|Carry0(7);
    AF=1;
} else {
    AF=0;
}
if((AL  >9Fh)|| CF) {
    AL  AL+60h;
    CF=1;
} else {
    CF=0;
}
OF  UNDEFINED;
SF=B(7,AL);
ZF=(AL    0);
PF=~(B(7,AL)^B(6,AL)^…^B(0,AL));
```

例子：

	OSZAPC
ADD AL,BL	
执行前：AL:77 BL:13	×××××××
执行后：AL:8A BL:13	110000
DAA	
执行前：AL:8A BL:13	110000
执行后：AL:90 BL:13	×10110
ADD AL,BL	
执行前：AL:79 BL:35	×××××××
执行后：AL:AE BL:35	110000
DAA	
执行前：AL:AE BL:35	110000
执行后：AL:14 BL:35	×00111

时钟：(假设主频为20MHz)

4个时钟周期；0.2μs。

标志影响：

影响 CF 和 AF。OF，SF，ZF 和 PF 无定义。

其它标志不变。

DAS

减法之后的十进制调整

格式：

DAS

0	0	1	0	1	1	1	1

说明：

DAS 只能紧跟在操作两个压缩的 BCD 数(一个压缩字节)的，并将结果放入 AL 中的 SUB 指令后面使用。

DAS 将 AL 寄存器内容转换成两位有效的压缩 BCD 数，结果仍放在 AL 中。

若 AL 的低4位大于9，或者 AF(辅助进位标志)为1，则 AL 减去6，并将 AF 置位。否则 AF 复位。若 AL 大于9Fh 或 CF 为1，则 AL 减去60h 并 CF 置位。否则 CF 复位。

操作：

```
if (((AL & 0Fh)>9)|| AF) {
    AL=AL-6;
    CF=CF|BorrowO(7);
    AF=1;
} else {
    AF=0;
}
if((AL>9Fh)||CF) {
    AL=AL-60h;
    CF=1;
} else {
    CF=0;
}
OF=UNDEFINED;
SF=B(7,AL);
ZF=(AL==0);
PF=~(B(7,AL)^B(6,AL)^…^B(0,AL));
```

例子：

	OSZAPC
SUB AL,BL	
执行前：AL:71 BL:13	××××××
执行后：AL:5E BL:13	000100
DAS	
执行前：AL:5E BL:13	000100
执行后：AL:58 BL:13	×00100
SUB AL,BL	
执行前：AL:35 BL:47	××××××
执行后：AL:EE BL:47	010111
DAS	
执行前：AL:EE BL:47	010111

执行后：AL:88　BL:47　　×10111

时钟:(假设主频为20MHz)

4个时钟周期;0.2μs。

标志影响:

影响CF和AF。OF,SF,ZF和PF为无定义。

其它标志不变。

DEC

减　量

格式:

DEC　op_1

mem

reg

reg(字或双字)

(短编码)

	字节0	字节1
mem	1 1 1 1 1 1 1 w	mod:0 0 1: r/m
reg	1 1 1 1 1 1 1 w	mod:0 0 1: r/m
reg(短编码)	0 1 0 0 1: reg	

说明:

op_1减去1,其结果放回op_1。

操作:

```
op1=op1-1;
/*"-"号视同SUB指令*/
OF=CarryO(MSB)^CarryO(MSB-1);
SF=B(MSB,op1);
ZF=op1==0;
AF=CarryO(3);
PF=~(B(7,op1)^B(6,op1)^…^B(0,op1));
CF=CF;
```

例子:

DEC　EAX　　OSZAPC

执行前：EAX:01234567　　××××××

执行后：EAX:01234566　　00001U

DEC mem16

执行前：Mem:0000　　××××××

执行后：Mem:FFFF　　01011U

时钟:(假设主频为20MHz)

DEC　reg:2个时钟周期;0.1μs。

DEC　mem:6个时钟周期;0.3μs

标志影响:

影响OF,SF,ZF,AF和PF。

其它标志不变。

异常:

Memory( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟8086模式异常:同 ADC 指令。

注意

DEC 指令不会改变进位标志。如果希望进位标志更新的话应当用操作数为1的 SUB 指令。

DIV

无符号除

格式:

DIV op_1

reg

mem

字节 0　　字节 1

1	1	1	1	0	1	1	w	mod	1	1	0	r/m

说明:

无符号除操作以及余数操作,作用于给定的操作数 op_1 以及某些隐指的寄存器。隐指的是哪些寄存器取决于 op_1 的长度。请看下表:

op_1的长度	被除数	除数	商	余数
8	AX	op_1	AL	AH
16	DX:AX	op_1	AX	DX
32	EDX:EAX	op_1	EAX	EDX

商的小数部分被截去。余数总是小于除数。

如果商超出了商寄存器(AL,AX,EAX)所能表达的示数范围,则产生除数为0的错误,即中断0。若除数为0的错误发生,则商和余数都是无意义的。

操作:

```
if(op1==0)Interrupt(0);
if(Length(op1)==8){
    if((AX/op1)>FFh)Interrupt(0);
    temp=AX;
    AL=temp/op1;
    AH=temp%op1;
} else if(Length(op1)==16){
    if((ConCat(32,DX,16,AX)/op1)>FFFFh)Interrupt(0);
    temp=AX;
    AX=ConCat(32,DX,16,temp)/op1;
    DX=ConCat(32,DX,16,temp)%op1;
} else {/*Length(op1)==32 */
    if((ConCat(64,EDX,32,EAX)/op1)>FFFFFFFFh)Interrupt(0);
    temp=EAX;
```

```
        EAX  =ConCat(64,EDX,32,temp)/op1;
        EDX  =ConCat(64,EDX,32,temp)%op1;
    }
    OF=SF=ZF=AF=PF=CF=UNDEFINED;
    /*
    "/"号视同无符号除。AX,ConCat(DX,AX)(DX 和 AX 的串接),ConCat(EDX,EAX)(EDX 和 EAX 的
串接),及 op1,都被认为是无符号整型数。同样,"%"号视同无符号求模或求余数。
    */
```

例子:

```
DIV  EBX                                          OSZAPC
    执行前: EDX:76543210 EAX:01234567             ××××××
            EDX:89014573
    执行后: EDX:34DCEE8F EAX:DD1A57C8             ××××××
            EBX:89014573
DIV mem8                                          OSZAPC
    执行前: AH:78 AL:2 1   mem:5C                 ××××××
    执行后: Interrupt(0);
            AH:×× AL:××   mem:5C                  ××××××
```

时钟:(假设主频为20MHz)

op_1为8位:19个时钟周期;0.95μs。

op_1为16位:27个时钟周期;1.35μs

op_1为32位:43个时钟周期;2.15μs

标志影响:

OF,CF,SF,ZF,AF 和 PF 为未定义。

其它标志不变。

异常:

Memory( );

Interrupt(0);

实地址模式异常:如果商太大,超出寄存器的示数范围,则产生中断0。

其它异常同 ADC 指令。

保护模式异常:如果商太大,超出寄存器的示数范围,则产生"除异常"(异常0)。对于异常0,没有错误码压入堆栈。

其它异常同 ADC 指令。

虚拟8086模式异常:如果商太大,超出寄存器示数范围,则产生"除异常"(异常0)。对于异常0,没有错误码压入堆栈。

其它异常同 ADC 指令。

ENTER

建立堆栈框架

格式:

ENTER op_1 , op_2

imm16 , imm8 | 1 1 0 0 1 0 0 0 | 16 位偏移量,8 位嵌套深度

说明:

ENTER 建立堆栈框架。新的框架通常是由大多数高级语言在每次过程调用时建立的。

op_2规定了程序的嵌套深度。这个嵌套深度决定了从当前堆栈框架拷贝到由该 ENTER 指令建立的新的堆栈框架中的堆栈框架指针的数目。EBP 寄存器被用来将框架指针从当前堆栈框架拷贝到新的堆栈框架中。在 ENTER 指令结束时,EBP 指向新的堆栈框架指针。

op_1规定了本地变量的字节数,根据本地变量的字节数自动分配堆栈空间。

操作:

```
level=op2%32;
push(EBP);
frame=ESP;
if(level>0){
    while((--level)>0){
        EBP=EBP-4;
        push([EBP]);
    }
    push(frame);
}
EBP=frame;
ESP=ESP-ConCat(32,0000h,16,op1);
```

例子:

```
ENTER  0ch,  4
    执行前: EBP            :00000F38 SS:[EBP]:00000F4C
            SS:[EBP-4]     :0000100B SS:[EBP-8]:000010A4
            SS:[EBP-12]:00001102 ESP            :00000F04
    执行后: SS:[ESP+28]:00000F38 SS:[ESP+24]:0000100B
            SS:[ESP+20]:000010A4 SS:[ESP+16]:00001102
            SS:[ESP+12]:00000F00 EBP            :00000F00
            ESP            :00000EE4
```

时钟:(假设主频为20MHz)

5个时钟周期;0.25μs。

标志影响:无。

异常:

Stack( );

实地址模式异常:无

保护模式异常:

执行指令期间,如果 SP 或 ESP 在任一点上超过堆栈界限,则引起错误码为0000h 的堆栈故障异常。

虚拟8086模式异常:无

注意:

请阅第6章例3b。

IDIV

符号除

格式：

		字节0	字节1
IDIV	op_1		
	reg	1 1 1 1 0 1 1 w	mod:1 1 0: r/m
	mem		

说明：

符号除操作和余数操作用于给定的操作数op_1以及某些隐指的寄存器。隐指的是哪些寄存器取决于op_1的长度。请看下表：

op_1的长度	被除数	除数	商	余数
8	AX	op_1	AL	AH
16	DX:AX	op_1	AX	DX
32	EDX:EAX	op_1	EAX	EDX

商的小数部分被截去。余数与被除数符号相同，其量值总小于除数。下表是4除以3的被除数和除数的各种符号组合，以及得出的商和余数：

被除数	(op_1) (除数)	商	余数
+4	+3	+1	+1
+4	−3	−1	+1
−4	+3	−1	−1
−4	−3	+1	−1

除法必定遵守下面的恒等式：

被除数＝商×除数＋余数。

若 op 为0，或商超出了商寄存器(AL，AX 或 EAX)所能表达的示数范围，则产生除数为0的错误。若除数为0的错误发生，则商和余数没有意义。

操作：

```
if(op1==0)Interrupt(0);
if(Length(op1)==8){
  if(((AX>0)&&(AX/op1)>7Fh)||((AX<0)&&(AX/op1)<80h))
      Interrupt(0);
  temp=AX;
  AL=temp/op1;
  AH=temp%op1;
} else if(Length(op1)==16 {
  if (((ConCat(32,DX,16,AX)>0)&&(ConCat(32,DX,16,AX)/op1)>7FFFh)||
    ((ConCat(32,DX,16,AX)<0)&&(ConCat(32,DX,16,AX)/op1)<8000h))
```

```
    Interrupt(0);
  temp=AX;
  AX=ConCat(32,DX,16,temp)/op1;
  DX=ConCat(32,DX,16,temp)%op1;
} else  {/* Length(op1)==32 */
  if (((ConCat(64,EDX,32,EAX)>0)&&(ConCat(64,EDX,32,EAX)/op1)>7FFFFFFFh)||
    ((ConCat(64,EDX,32,EAX)<0)&&(ConCat(64,EDX,32,EAX)/op1)<80000000h))
      Interrupt(0);
  temp=EAX;
  EAX=ConCat(64,EDX,32,temp)/op1;
  EDX=ConCat(64,EDX,32,temp)%op1;
}
OF=SF=ZF=AF=PF=CF=UNDEFINED;
/*
"/"号视同符号除法。AX,ConCat(DX,AX)(DX 和 AX 的串接),ConCat(EDX,EAX)(EDX 和 EAX 的串接)及 op1,都被考虑为符号整型数。
同样,"%"号视同符号求模或求余数。
*/
```

例子:

IDIV BX	OSZAPC
执行前: DX:3210 AX:4567	×××××× BX:6773
执行后: DX:496E AX:7BE3	×××××× BX:6773
IDIV mem8	OSZAPC
执行前: AH:D8 AL:D7 mem:AD	××××××
执行后: AH:BF AL:78 mem:AD	××××××
IDIV BL	OSZAPC
执行前: AH:87 AL:B2 BL:19	××××××
执行后: Interrupt(0);	
AH:×× AL:×× BL:19	××××××

时钟:(假设主频为20MHz)

op_1为8位:19个时钟周期;0.95μs。

op_1为16位:27个时钟周期;1.35μs

op_1为32位:43个时钟周期;2.15μs

标志影响:

OF,CF,SF,ZF,AF 和 PF 为未定义。

其它标志位不变。

异常:

Memory( );

Interrupt( );

实地址模式异常:

若商太大,超出了寄存器的示数范围,则产生中断0。

其它异常同 ADC 指令。

保护模式异常:

若商太大，超出了寄存器的示数范围，则产生“除异常”(异常0)，没有错误码压入堆栈。

其它异常同 ADC 指令。

虚拟8086模式异常：

若商太大，超出了寄存器的示数范围，则产生“除异常”(异常0)，没有错误码压入堆栈。

其它异常同 ADC 指令。

注意：

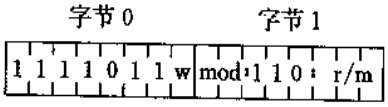
请看第6章第一节的例1及有关的讨论。

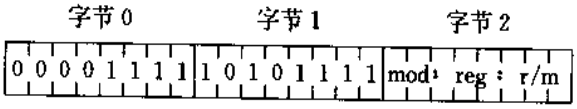
IMUL

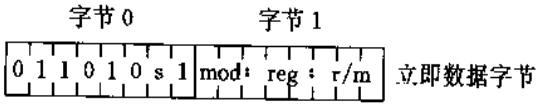
符号乘

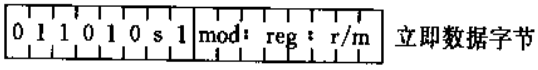
格式：

IMUL	op_1			
	reg			
	mem			
IMUL	op_1	,	op_2	
	reg16	,	reg16	
	reg16	,	mem16	
	reg32	,	reg32	
	reg32	,	mem32	
	reg16	,	imm	
	reg32	,	imm	
IMUL	op_1	,	op_2	, op_3
	reg16	,	reg16	, imm
	reg16	,	mem16	, imm
	reg32	,	reg32	, imm
	reg32	,	mem32	, imm









说明：

若 IMUL 只有一个操作数，其第二个操作数隐指 AL，AX，或 EAX。其16位的，32位的或64位的符号乘积相应地存入 AX，DX:AX，或 EDX:EAX。若 AH，DX，或 EDX 仅为 AL，AX，或 EAX 的符号扩展，则 CF 和 OF 被清0。否则 CF 和 OF 置位。

在两个操作数的情况下，$op_1 \times op_2$的积存入 op_1。若 $op_1 \times op_2$的积在 op_1可以表示的范围内，则 CF 和 OF 清0。否则 CF 和 OF 置位。

在三个操作数的情况下，$op_2 \times op_3$的积存入 op_1。若 $op_2 \times op_3$的积在 op_1可以表示的范围内，则 CF 和 OF 清0。否则 CF 和 OF 置位。

操作：

```
if(NUMOPS==1){
    if(Length(op1)==8){
```

```
        AX=AL * op1;
        if((AH==00h)||(AH==FFh)){
          CF=0; OF=0;
        } else {
          CF=1;OF=1;
        }
      } else if(Length(op1)==16{
        DX=(AX * op1)>>16;
        AX=(AX * op1)&0FFFFh;
        if((DX==0000h)||(DX==FFFFh)){
          CF=0;OF=0;
        } else {
          CF=1;OF=1;
        }
      } else {  /*Length(op1)==32*/
        EDX=(EAX * op1)>>32;
        EAX=(EAX * op1)&0FFFFFFFFh;
        if((EDX==00000000h)||(EDX==FFFFFFFFh)){
          CF=0;OF=0;
        } else {
          CF=1;OF=1;
        }
      }
    } else if(NUMOPS==2){
      doubleTemp=op1 * op2;
      op1=op1 * op2;
      if(doubleTemp!=op1){
        CF=1;OF=1;
      } else {
        CF=0;OF=0;
      }
    } else { /*NUMOPS==3*/
      op1=op2 * op3;
        doubleTemp=op2 * op3;
        if(doubleTemp!=op1){
          CF=1;OF=1;
        } else {
          CF=0;OF=0;
        }
    }
    SF=ZF=AF=PF=UNDEFINED;
    /*
        "*"号视同符号乘法。
    */
```

例子：

```
IMUL  BX                                          OSZAPC
    执行前：DX：××××  AX：1862                    ××××××
            BX：8536
    执行后：DX：F44E  AX：0EAC                    1××××1
```

BX :8536

IMUL ECX,mem32 — OSZAPC

执行前：ECX:00015792 mem32:00052692 — ××××××

执行后：ECX:E99D9D44 mem32:00052692 — 1××××1

IMUL CX,mem16,8 — OSZAPC

执行前：CX:×××× mem16:002A — ××××××

执行后：CX:0150 mem16:002A — 0××××0

时钟:(假设主频为20MHz)

IMUL reg :9～38个时钟周期;0.45～1.9μs。

IMUL mem :12～41个时钟周期;0.6～2.05μs。

IMUL reg,reg:9～38个时钟周期;0.45～1.9μs。

IMUL reg,mem :12～44个时钟周期;0.6～2.2μs。

IMUL reg,imm :9～38个时钟周期;0.45～1.9μs。

IMUL reg,reg,imm:9～38个时钟周期;0.45～1.9μs。

IMUL reg,mem,imm:12～41个时钟周期;0.48～2.05μs。

标志影响:

影响 OF,CF,SF,ZF,AF 和 PF 为未定义。

其它标志不变

异常:

Memory( )

实地址模式异常:同 ACD 指令。

保护模式异常:同 ADC 指令。

虚拟模式异常:同 ADC 指令。

注意:请看第6章例3a 和例3b。

IN

从端口输入

格式:

IN op_1 , op_1

AL , imm8 — 1 1 1 0 0 1 0 w 8位端口

AX , imm8

EAX , imm8

AL , DX — 1 1 1 0 1 1 0 w

AX , DX

EAX , DX

说明:

IN 从规定的端口将一数据字节传送到 AL,或将一数据字传送到 AX,或将一数据双字传送到 EAX。

I/O 口在0到64K－1的范围内。指令访问端口的格式有两种，一种是用立即数字节，另一种是间接地通过 DX 寄存器。

立即数字节格式允许存取0到255的端口。在这种情况下，端口地址的高位地址总为0。寄存器格式(DX)允许存取0到64K－1范围中的所有端口。

若当前的任务没有执行 I/O 的足够权限，则发生异常。请详阅第5章关于特权级别的详述。

操作：

op_1=port(op_2)；

例子：

```
IN  AL,20h
    执行前：AL:××   port(20):4 A
    执行后：AL:4 A  port(20):××
IN  EAX,DX
    执行前：EAX:        ××××××××  DX:4680
            port(4680):FFA78201
    执行后：EAX:        FFA78201  DX:4680
            port(4680):××××××××
```

时钟：(假设主频为20MHz)

实地址模式：

8位端口号：12个时钟周期；0.6μs。

DX 决定端口：13个时钟周期；0.65μs。

保护模式，CPL≤IOPL：

8位端口号：6个时钟周期；0.3μs。

DX 决定端口：7个时钟周期；0.35μs。

保护模式，CPL>IOPL：

8位端口号：26个时钟周期；1.3μs。

DX 决定端口：27个时钟周期；1.35μs。

虚拟8086模式：

8位端口号：26个时钟周期；1.3μs。

DX 决定端口：27个时钟周期；1.35μs。

标志影响：无。

异常：

AccessIO( )；

实地址模式异常：同 ADC 指令。

保护模式异常：

在任务状态段中有一个 I/O 位图，每一位对应于一个字节的 I/O 端口。若此位为0，则相应的端口可由任意特权的任务 I/O；若此位为1，则只有 CPL≤IOPL 的任务才能从此端口 I/O。其它异常同 ADC 指令。

虚拟8086异常：同保护模式。

INC

增　量

格式：

	字节0	字节1
INC op_1		
mem	1 1 1 1 1 1 1 w	mod:0 0 0 : r/m
reg		
reg		
（仅对字或双字）	0 1 0 0 0 : reg	
（短编码）		

说明：

op_1被加1，结果仍放入 op_1。除 CF 以外的所有标志位按运算结果被设置。

操作：

```
op1=op1+1;
/*"+"号视同 ADD 指令。*/
OF=CarryO(MSB)^CarryO(MSB-1);
SF=B(MSB,op1);
ZF=op1==0;
AF=Carry O(3);
PF=~(B(7,op1)^B(6,op1)^…^B(0,op1));
CF=CF;
```

例子：

	OSZAPC
INC　AX	
执行前：AX:22D9	××××××
执行后：AX:22DA	00000U
INC　mem32	
执行前：mem:0 5 6 2 1 3 4 0	××××××
执行后：mem:0 5 6 2 1 3 4 1	00001U

时钟：（假设主频为20MHz）

INC reg：2个时钟周期；0.1μs。

INC mem：6个时钟周期；0.3μs。

标志影响：

影响 OF，SF，ZF，AF 和 PF。其它标志不变。

异常：

Memory( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟模式异常：同 ADC 指令。

注意：

进位标志不受 INC 指令的影响。若希望更新进位标志的话，则使用操作数为1的 ADD 指令。

INS/INSB/INSW/INSD

输入串操作数

格式：

INSB

INSW

INSD

1	0	1	0	0	1	1	w

INS mem1 , reg2

mem8 , DX

mem16 , DX

mem32 , DX

说明：

从由 DX 指定的端口，将一字节(8位)、二字节(16位)，四字节(32位)的数据传送到 ES:[EDI]之中。在传送操作完成之后，根据不同的操作数长度，EDI 分别加减1，2，或4，以指向串中下一个元素。若方向标志 DF 为0则加，否则减。

使用的端口被规定在寄存器 DX 之中。不允许使用立即数端口格式。

注意，内存单元永远由 ES:[EDI]规定。INS 不允许使用其它的段超越前缀。mem1只用来表示操作数长度。

INS 可与前缀 REP 连用。在与 REP 连用的情况下，可以输入 ECX 个字节、字、或双字。请详阅 REP 指令。

若当前任务没有执行 I/O 的足够权限，则产生异常。请阅第5章关于特权级别的详述。

操作：

```
if(OPCODE==INSB){
    ES:[EDI]=(byte)port(DX);
    delta=1;
} else if(OPCODE==INSW) {
    ES:[EDI]=(word)port(DX);
    delta=2;
} else if(OPCODE==INSD) {
    ES:[EDI]=(dword)port(DX);
    delta=4;
} else{        /* INS */
    ES:[EDI]=(Length(mem1))port(DX);
    delta=Length(mem1)/8;
}
if(DF==1){
    delta=-delta;
}
EDI=EDI+delta;
```

例子：

INSD

执行前：EDI:00000052 ES:[EDI]:xxxxxxxx

DX:004C　DF:0　port(DX)=FFFFD832

执行后：EDI:00000056　ES:[EDI-4]=FFFFD832

DX:004C　DF:0　port(DX)=FFFFD832

REP INS S1,DX　;S1为一16位串

执行前：EDI:0374　DX:4220　DF:1　ECX:2

port(DX,1):6000　ES:[EDI]　:××××

port(DX,2):0455　ES:[EDI-2]　:××××

port(DX,3):6001　ES:[EDI-4]　:××××

执行后：EDI:0370　DX:4220　DF:1　ECX :0

port(DX,1):6000　ES:[EDI+4]　:6000

port(DX,2):0455　ES:[EDI+2]　:0455

port(DX,3):6001　ES:[EDI]　:××××

时钟:(假设主频为20MHz)

实地址模式：

不重复:15个时钟周期;0.75μs。

重复:(13+6n)个时钟周期;(0.65+0.3n)μs。

保护模式,CPL≤IOPL：

不重复:9个时钟周期;0.45μs。

重复:(7+6n)个时钟周期;(0.35+0.3n)μs。

保护模式,CPL>IOPL：

不重复:29个时钟周期;1.45μs。

重复:(27+6n)个时钟周期;(1.35+0.3n)μs。

虚拟8086模式：

不重复:29个时钟周期;1.45μs。

重复:(27+6n)个时钟周期;(1.35μs+0.3n)μs。

(其中 n 为迭代执行的次数)

标志影响:无。

异常:

Memory( );

AccessIO( )：

实地址模式异常:同 ADC 指令。

保护模式异常:同 IN 指令。

虚拟8086模式异常:同 IN 指令。

Jcc

条件跳转

格式:

Jcc near-label(近标号)

	字节 0	字节 1	后续字节
8位偏移量	0 1 1 1 c c c c	8位偏移量	
全偏移量	0 0 0 0 1 1 1 1	1 0 0 0 c c c c	全偏移量

说明：

字节中的CCCC为条件码，共有16种状态：

CCCC	助记符	相应的操作
0000	JO	当OF==1(溢出)时跳转
0001	JNO	当OF==0(无溢出)时跳转
0010	JB	当CF==1(低于/不高于等于/进位)时跳转
	JNAE	
	JC	
0011	JNB	当CF==0(不低于/高于等于/无进位)时跳转
	JAE	
	JNC	
0100	JE	当ZF==1(等于/0)时跳转
	JZ	
0101	JNE	当ZF==0(不等于/非0)时跳转
	JNZ	
0110	JBE	当CF==1||ZF==1(低于等于/不高于)时跳转
	JNA	
0111	JNBE	当CF==0 && ZF==0(不低于等于/高于)时跳转
	JA	
1000	JS	当SF==1(符号/负)时跳转
1001	JNS	当SF==0(无符号/正)时跳转
1010	JP	当PF==1(操作的结果中为"1"的位数为偶)时跳转
	JPE	
1011	JNP	当PF==0(操作的结果中为"1"的位数为奇)时跳转
	JPO	
1100	JL	当SF!=OF(小于/不大于等于)时跳转
	JNGE	
1101	JNL	当SF==OF(不小于/大于等于)时跳转
	JGE	
1110	JLE	当SF!=OF||ZF==1(小于等于/不大于)时跳转
	JNG	
1111	JNLE	当ZF==0 && SF==OF(不小于等于/大于)时跳转
	JG	

以上关于大于和小于请参考符号整数比较，关于高于和低于请参考无符号整数比较。

Jcc先进行标志位测试，如标志与条件相符，则产生向当前代码段内的规定标号的跳转(跳转的目标必须在同一代码段内)。否则继续执行下一条后续指令。

这里假定标志已被前面的指令以某种有效的方法设好(然而并不要求是紧挨着的前一条指令)。

若在同种状态下，汇编给出了多个助记符，(如CCCC为0111，助记符为JNBA和JA)这些助记符是同义的，它们之间没有区别，只是为使用方便而已。

操作：

```
if(Condition Code){
  EIP=Offset(near—label);/*† */
}
/*
    † 该近标号被汇编成一相对于当前指令指针的立即数偏移量。于是，机器将EIP修改成：
      EIP=EIP+imm;
```

该立即数偏移量可取8位到32位。
*/

例子：

```
JO  near—label1;  在程序中,label1  前移68h 字节          O S Z A P C
    执行前：EIP:00000300                                  1 ×××××
    执行后：EIP:00000368                                  U UUUUU
JAE near—label1;在程序中,label1后移537h 字节
    执行前：EIP:00000300                                  ××××× 1
    执行后：EIP:00000300                                  U UUUUU
```

因条件未满足(CF 为1)，于是该取下一条指令。当取出 JAE 指令时，EIP 已指向下一条指令，所以 EIP 不变。

标志影响：无。

异常：

Code Reference( )；

实地址模式异常：

如果目标指令的任何部分是在 CS 段中大于 FFFFh 的有效地址，则产生中断13。

保护模式异常：

对于在内存 CS 段中的目标，如果目标指令的任何部分是在超出了段界的有效地址，则产生错误码为0000h 的通用保护异常。若内存中的操作数是在标为"未出现"的 CS 段中，则产生错误码为段选择符的"未出现"异常(异常11)。

页故障引起错误码为故障码的，页故障线性地址在寄存器 CR2之中的页故障异常(异常14)。

虚拟8086模式异常：

同实地址模式异常，但这些异常的处理是在保护模式及特权级为0的情况下进行的。页故障异常同保护模式。

注意：

请阅第6章的例10。

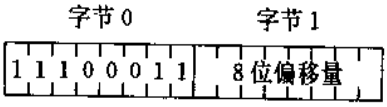
JCXZ/JECXZ

CX/ECX 为0时跳转

格式：

JCXZ short_label

JECXZ short_label



说明：

JCXZ/JECXZ 提供了在当前代码段范围内的条件分支。这种分支是以 CX/ECX 寄存器的内容为条件的。若寄存器为0，指令操作数所提供的偏移量便加到当前的指令指针上，指令的执行便在新的偏移量处继续。该偏移量为8位符号量，范围为(−128)到(+127)。

由于这条件跳转只能在同一段内，所以代码段寄存器不受影响。

操作：

```
if(ECX==0){
    EIP=shor_label; /* † */
}
/*
† short label 被汇编成相对于已指向下条后续指令的当前指令指针的8位立即数偏移量。
  EIP 被修改为：
      EIP = EIP+Sign Ex(32,8,imm8);
*/
```

例子：

JECXZ short_label1；在程序中，short_label1为前移42h

执行前：EIP:000DE300 ECX:0

执行后：EIP:000DE342 ECX:0

标志影响：无。

异常：

同 Jcc 指令。

JMP

跳　转

格式：

JMP op_1

reg32

mem32

near label(近标号)

短偏移量

全偏移量

	字节0	字节1
	1 1 1 1 1 1 1 1	mod:1 1 1: r/m

1 1 1 0 1 0 1 1	8位偏移量
1 1 1 0 1 0 0 1	全偏移量

说明：

跳转指令使指令执行在当前代码段中给定的偏移量处继续。段内跳转的偏移量能由近标号，reg32或 mem32规定。近标号格式规定了至给定标号的直接跳转。reg32和 mem32操作数格式为间接跳转，跳转的偏移量取自指定的寄存器或内存单元。

注意，JMP 指令和 CALL 指令不同，JMP 指令不保存返回信息。

操作：

```
if(op1 is near_label){
    EIP=Offset(near_label);/*† */
} else if(op1 is reg32){
    EIP=reg32;
} else if (op1 is mem32){
    EIP=mem32;
```

```
}
/*
近标号被汇编成相对于当前指令指针的立即数偏移量。于是 EIP 将变成
    EIP=EIP+imm;
立即数偏移量可以取8到32位。
*/
```

例子：

JMP near_label1 ;在程序中标号1(label1)为向前42h 字节

执行前：EIP:300

执行后：EIP:342

JMP CASETABLE[EAX * 4];CASETABLE 是一个跳转表，EAX * 4给出了跳转的目标在这张表中的索引

执行前：EIP:5022AC

CASETABLE[EAX * 4]:60AA40

执行后：EIP:60AA40

标志影响：

无。除非在保护模式下发生任务切换。

异常：

Memory( );

Code Reference( );

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：同 ADC 指令。

注意：

请再阅 JMP 指令的多段格式。

请看第6章例2。

LAHF

将标志装入 AH 寄存器

格式：

LAHF

1	0	0	1	1	1	1	1

说明：

标志字的低字节被装入 AH。这些标志按降序从高位到低位分别为符号、零、值0、辅助进位、值0、奇偶、值1、进位。下图表示 EFLAGS 寄存器的低8标志位。

7	6	5	4	3	2	1	0
S	Z		A		P		C
F	F	0	F	0	F	1	F

操作：

AH=(EFLAGS & 0FFh);

例子：

LAHF

执行前：AH:××××××××b	OSZAPC
执行后：AH:01010110b	×01110
	U01110

时钟：(假设主频为20MHz)

2个时钟周期：0.1μs。

标志影响：无。

LEA

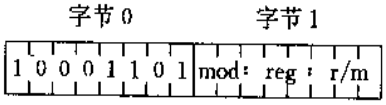
装入有效地址

格式：

LEA op_1 , op_2

reg32, mem

reg16, mem



说明：

计算 op_2的地址的偏移量部分(不是计算 op_2的值)，并将其装入 op_1。op_2必须是内存操作数。而 op_1一定是寄存器。如果 op_1是个16位的寄存器，则偏移量的低16位被存入 op_1。

内存地址的形成请看第2章。

操作：

op_1=&op_2;

例子：

LEA AX,[EBP+20h]

执行前：AX:××××	EBP :FFFA0246
执行后：AX:0266	EBP :FFFA0246

LEA ECX,[EAX*2+14ch]

执行前：EAX:0548901A	ECX :×××××××××
执行后：EAX:0548901A	ECX :0A91 2180

时钟：(假设主频为20MHz)

2个时钟周期：0.1μs。

若有效地址中含有两个寄存器，如：

LEA ECX,[EDX][4*ESI]DOLLAR　　LEA 要求一额外的时钟周期。

标志影响：无。

异常：

实地址模式异常：若 op_2为寄存器，则产生中断6。

保护模式异常：

若 op_2为寄存器，则产生无效操作码异常(异常6)。异常6不把任何错误码压入栈。

虚拟8086模式异常：同保护模式异常。

注意：

LEA 对地址计算非常有用，比如计算一多维数组的偏移址。它还是个多用途的算术指令。它执行通用的三地址寄存器加：

r1=r2+r3+常数

利用变址寻址中的定标系数方式(请阅第2章)可用来计算小的乘数。

LEAVE

过程退出

格式：

LEAVE

1	1	0	0	1	0	0	1

说明：

LEAVE 撤掉由相应的 ENTER 指令建立的堆栈框架。通过将 EBP(框架指针)的值赋还给 ESP，从而释放该堆栈空间。从栈中弹出的老框架指针(在 EBP 的值赋予 ESP 之后位于栈顶的)，被放入 EBP 中，从而为调用过程建立起框架指针。

通常，RET 指令用于将控制返回调用过程。RET 指令跟在反向链接之后，并为正在退出的过程删去已经压入堆栈的所有参数。

操作：

```
ESP=EBP;
EBP=Pop( );
```

例子：

```
LEAVE
    执行前：EBP:00000F00   SS:[EBP]:0000F38
            ESP:00000EB4
    执行后：EBP:00000F38
            ESP:00000F04
```

时钟：

4个时钟周期；若主频为20MHz，为0.2μs。

标志影响：无。

异常：

Stack( );

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：同 ADC 指令。

注意：

请阅第6章的例3a。

LOCK

总线锁定

格式：

LOCK instruction(指令)

1	1	1	1	0	0	0	0

说明：

为确保不可分割的读/修改/写操作的发生，在某些特殊指令的执行期间置起 LOCK # 信号。当总线上有多个处理器时，这个信号对保证总线以及所有附加在总线上的内存来说是很重要的，尤其是在一些敏感的操作期间。比如说，当一个"信号灯"正在被修改的时刻。LOCK 只能用于下列指令：

ADC,ADD,AND,BT	mem	, reg /imm
BTS,BTR,BTC, OR	mem	, reg /imm
SBB,SUB,XOR	mem	, reg /imm.
XCHG	reg	, mem
XCHG	mem	, reg
EDC,INC,NEG,NOT	mem	

若 LOCK 与不在此表中的指令连用，则发生非法操作码异常(请看第6章)。

操作：

```
LOCK # =0;
instruction;
LOCK # =1;
```

例子：

;当执行 BTS 时，LOCK # 信号被置起。

LOCK BTS dword ptr [ESI],ECX

异常：

InvalidOpcode( )；请看本指令《说明》。

被锁住的指令可能会产生 Memory( )异常。

注意：

80386的可锁定的指令表和86系列以前的成员的不一样。

请看附录 A。

请看第6章的例4。

LODS/LODSB/LODSW/LODSD

装入串操作数

格式：

LODSB

LODSW

1	0	1	0	1	1	0	w

LODSD

LODS

mem1

mem8

mem16

mem32

说明：

一字节、二字节、或四字节的数据（对于8位的、16位的、或32位的操作），从[ESI]，分别传送到AL、AX、或EAX寄存器。在传送之后，ESI改为指向串中下一个元素；根据不同的操作数长度，分别加减1、2或4；若DF为0则加，DF为1则减。

mem1规定了操作数长度，而实际的传送由ESI寄存器的内容所规定的地址完成。可以在mem1中规定用在[ESI]前面的段超越前缀。（默认的mem1段寄存器为DS。）

操作：

```
if(OPCODE==LODSB){
      AL = (byte)DS:[ESI];
      delta=1;
  } else if(OPCODE==LODSW){
        AX = (word)DS:[ESI];
        delta=2;
    } else if(OPCODE==LODSD){
        EAX=(dword)DS:[ESI];
         delta=4;
    } else {      /* LODS */
    /* 注意，默认的SegReg(mem1)==DS */
  if(Length(mem1)==8){
      AL=SegReg(mem1):[ESI];
      delta=1;
  } else if(Length(mem1)==16){
      AX=SegReg(mem1):[ESI];
        delta=2;
    } else if(Length(mem1)==32){
        EAX=SegReg(mem1):[ESI];
        delta=4;
    }
  }
  if(DF==1){
    delta=-delta;
  }
  ESI=ESI+delta;
```

例子：

LODSB

执行前：	ESI:00000052	DS:[ESI]:BF
	AL:xx	DF:0
执行后：	ESI:00000053	DS:[ESI-1]:BF
	AL:BF	DF:0

时钟：

5个时钟周期；在主频为20MHz 情况下，为0.25μs。

标志影响：无。

异常：

Memory( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：同 ADC 指令。

注意：

请阅第6章的例6a 和例6b。

LOOP

用 ECX 计数器的循环

格式：

LOOP short label(短标号，8位字节偏移量)

字节 0	字节 1
1 1 1 0 0 0 1 0	8 位偏移量

说明：

ECX 寄存器被减一，而不改变任何标志。若 ECX 被减一之后仍不为0，控制即被传送至由 short_label(短标号)规定的地方。若 ECX 为0，便执行紧跟在 LOOP 后面的指令。short_label 是在 LOOP 指令的+127字节和－128字节范围内(即相对于当前 EIP 的位移量为(－128)到(＋127)字节)。

操作：

```
ECX=ECX-1;/* 不改变任何标志 */
if(ECX!=0){
EIP=short_label;/* 于 */
}
/*
于短标号格式被汇编成简单的8位立即数偏移址。这偏移址是相对于当前指令指针的，它已指向了下一条后续指令。
        EIP 变为：
        EIP=EIP+SignEx(32,8,imm8)
*/
```

例子：

LOOP near_label1；在程序中，label1为后移4Eh 字节

执行前：EIP:000DE300 ECX:00000002

执行后：EIP:000DE2B2 ECX:00000001

LOOP near-label1；在程序中，label1为后移4Eh 字节

执行前：EIP:000DE300 ECX:00000001

执行后：EIP:000DE300 ECX:00000000

执行下一条后续指令

标志影响：无。

异常：

CodeReference( );

实地址模式异常:同 Jcc 指令。

保护模式异常:同 Jcc 指令。

虚拟8086模式异常:同 Jcc 指令。

注意:

LOOP 指令应当放在循环的底部,而 short_label 应当放在循环的顶部。

请看第6章的例2。

LOOPNZ/LOOPNE

用 ECX 计数器且不为0的循环/用 ECX 计数器且不相等的循环

格式:

LOOPNZ short label(短标号)

LOOPNE short_label(短标号)

字节0	字节1
1 1 1 0 0 0 0 0	8位偏移量

说明:

ECX 寄存器被减一,而不改变任何标志位。若 ECX 的值在减一后仍不为0,且 ZF 为0,控制被传送到由短标号规定的地方。若 ECX 为0,或者 ZF 为1,便执行紧跟在 LOOPNE 或 LOOPNZ 后面的指令。短标号是在 LOOPNZ/LOOPNE 指令的+127字节和-128字节的范围内(即相对于当前 EIP 的偏移量为-128到+127字节)。

操作:

```
ECX=ECX-1; /* 不改变任何标志 */
if((ZF==0)&&(ECX!=0)){
    EIP=short_label;/* 个 */
}
/*
个短标号格式被汇编成简单的8位立即数偏移量。该偏移量是相对于指向下一条后续指令的当前指令指针的。EIP 变为:
        EIP=EIP+SignEX(32,8,imm8);
*/
```

例子:

```
LOOPNE  near  label2;在程序中,label2后移42h 字节                O S Z A P C
    执行前: EIP:000DE300  ECX:00000008                          × × 1 × × ×
    执行后: EIP:000DE300  ECX:00000007                          U U 1 U U U
            下一条后续指令被执行(∵ZF=1)
LOOPNZ near_label3;在程序中,label3后移3Ch 字节
    执行前: EIP:000C8300  ECX:00000002                          × × 0 × × ×
    执行后: EIP:000C82C4  ECX:00000001                          U U 0 U U U
```

标志影响:无。

异常:

CodeReference( );

实地址模式异常:同 Jcc 指令。

保护模式异常:同 Jcc 指令。

虚拟8086模式异常:同 Jcc 指令。

注意:

LOOPNE 和 LOOPNZ 是同义的。

LOOPNE/LOOPNZ 应当放在循环的底部,而短标号则应放在循环的顶部。

请看第6章例11。

LOOPZ/LOOPE

用 ECX 计数器且为0的循环/用 ECX 计数器且相等的循环

格式:

LOOPZ short_label

LOOPE short_label

字节0	字节1
1 1 1 0 0 0 0 1	8位偏移量

说明:

ECX 寄存器被减一,而不影响任何标志。若 ECX 的值被减一后仍不为0且 ZF 为1,控制被传送到由短标号规定的地方。若 ECX 为0,或者 ZF 为0,便执行紧跟在 LOOPZ 或 LOOPE 后面的指令。短标号是在 LOOPZ/LOOPE 指令的+127字节和-128字节的范围内(即相对于当前 EIP 的偏移量为-128到+127字节)。

操作:

```
ECX=ECX-1;/* 不改变任何标志位 */
if((ZF==1)&&(ECX!=0)){
    EIP=short_label;/* †/*
}
/*
† 短标号格式被汇编成简单的8位立即数偏移量。该位移量是相对于指向下一条后续指令的当前指令指针的。
    EIP 变成:
      EIP=EIP+SignEx(32,8,imm8);
*/
```

例子:

```
LOOPE near label2;在程序中,label2后移4Eh 字节                    O S Z A P C
    执行前: EIP:000DE300   ECX:00000001                          × × 0 × × ×
    执行后: EIP:000DE300   ECX:00000000                          U U 0 U U U
            ;执行下一条后续指令。(∵ECX 为0)
LOOPZ near label3;在程序中,label3后移3Ch 字节
    执行前: EIP:000C8300   ECX:00000002                          × × 1 × × ×
    执行后: EIP:000C82C4   ECX:00000001                          U U 1 U U U
```

标志影响:无

异常:

CodeReference( );

实地址模式异常：同 Jcc 指令。

保护模式异常：同 Jcc 指令。

虚拟8086模式异常：同 Jcc 指令。

注意：

LOOPE 和 LOOPZ 是同义的。

LOOPE/LOOPZ 应当放在循环的底部，而短标号应当放在循环的顶部。

MOV

传　送

格式：

	字节 0	字节 1	后续字节
MOV op_1 , op_2	1 0 0 0 1 0 0 w	mod: reg : r/m	
reg , reg			
mem , reg			
mem ,AL/AX/EAX（短编码）	1 0 1 0 0 0 1 w	全字节偏移量	
reg , reg	1 0 0 0 1 0 1 w	mod: reg : r/m	
reg , mem			
AL/AX/EAX,mem（短编码）	1 0 1 0 0 0 0 w	全字节偏移量	
reg , imm	1 1 0 0 0 1 1 w	mod:0 0 0 : r/m	立即数据字节
mem , imm			
reg , imm（短编码）	1 0 1 1:w: reg	立即数据字节	

说明：

op_2的内容被复制到 op_1。op_1和 op_2可以是字节，字，或双字。

操作：

$op_1=op_2$；

例子：

MOV　AX,0A80h

　　执行前:AX:××××

　　执行后:AX:0 A8 0

MOV　EAX,mem32

　　执行前:EAX:×××××××× mem:0892ABDF

　　执行后:EAX:0 8 9 2 ABDF mem:0892ABDF

MOV　mem8,AL

　　执行前:AL:4A　　mem:× ×

执行后：AL：4A　　mem：4 A

时钟：(假设主频为20MHz)

MOV　reg，reg：2个时钟周期；0.1μs。

MOV　mem，reg：2个时钟周期；0.1μs。

MOV　reg，mem：4个时钟周期；0.2μs。

MOV　reg，imm：2个时钟周期；0.1μs。

MOV　mem，imm：2个时钟周期；0.1μs。

标志影响：无。

异常：Memory( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：同 ADC 指令。

MOVS/MOVSB/MOVSW/MOVSD

传送串操作数

格式：

MOVSB

MOVSW　　1 0 1 0 0 1 0 w

MOVSD

MOVS　mem1　，mem2

　　mem8　，mem8

　　mem16　，mem16

　　mem32　，mem32

说明：

MOVS 将由[ESI]指定的字节、字、或双字传送到 ES：[EDI]。传送操作完成后，ESI 和 EDI 寄存器的内容，根据操作数长度的不同，分别加减1，2，或4，以指向串中下一个元素。若 DF 为0则加，否则减。操作数长度由 mem1和 mem2决定，或由操作码(MOVSB，MOVSW，MOVSD)本身决定。

MOVS 指令可与 REP 前缀连用。若与 REP 连用，MOVS 指令重复执行 ECX 次。这样，长度为 ECX 的字节串，字串，或双字串，就从[ESI]传送至 ES：[EDI]。

mem1和 mem2格式被汇编程序用来决定操作数的长度以及 mem2的段超越前缀。被操作的字符串，总是从[ESI]取出，而传送到 ES：[EDI]。MOVS 可以包含一个段超越前缀，用来替代 mem2([ESI])的默认段寄存器 DS。而 mem1则唯一地取决于 ES：[EDI]。

操作：

```
if(OPCODE==MOVSB){
  delta=1;
  ES:[EDI]=(byte)DS:[ESI];
```

```
} else if(OPCODE==MOVSW){
    delta=2;
    ES:[EDI]=(word)DS:[ESI];
} else if(OPCODE==MOVSD){
    delta=4;
    ES:[EDI]=(dword)DS:[ESI];
} else {              /* MOVS */
  /*注意:默认的 SegReg(mem2)为 DS */
    delta=Length(mem1)/8;
    ES:[EDI]=SegReg(mem2):[ESI];
}
if(DF==1){
    delta=-delta;
}
ESI=ESI+delta;
EDI=EDI+delta;
```

例子:

MOVSD

执行前:ESI:06533A40　EDI:07822CD4

[ESI]:FFFFFFFF　ES:[EDI]:××××××××

DF:0

执行后:ESI:06533A44　EDI:07822CD8

[ESI-4]:FFFFFFFF　ES:[EDI-4]:FFFFFFFF

DF:0

REP　MOVSD

执行前:ECX:00000002　DF:1

EDI:00000010　ESI:00000008

ES:[EDI]:×××××××× [ESI]:01234567

ES:[EDI-4]×××××××× [ESI-4]:F421890A

执行后:ECX:00000000　DF:1

EDI:00000008　ESI:00000000

ES:[EDI+8]:01234567　[ESI+8]:01234567

ES:[EDI+4]:F421890A　[ESI+4]:F421890A

时钟:(假设主频为20MHz)

不重复:7个时钟周期;0.35μs。

重复:(7+4n)个时钟周期;(0.35+0.2n))μs。

(n 为迭代执行次数)

标志影响:无。

异常:

Memory( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟8086模式异常:同 ADC 指令。

注意：

请阅第6章的例1。

MOVSX

传送和符号扩展

格式：

MOVSX op_1 , op_2

reg16 , reg8

reg16 , mem8

reg16 , reg16

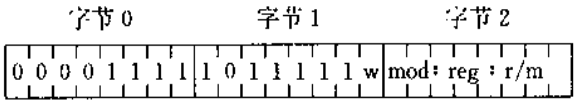
reg16 , mem16

reg32 , reg8

reg32 , mem8

reg32 , reg16

reg32 , mem16



说明：

字 op_1修改为字节 op_2的符号扩展；双字 op_1修改为字 op_2的符号扩展。若 op_1和 op_2均为字，则发生通常的传送。

操作：

```
if(Length(op1)==16){
    if(Length(op2)==8){
        op1=Sign Ex(16,8,op2);
    } else {
    op1=op2;      /*正常传送!*/
    }
} else {          /*Length(op1)==32*/
    if(Length(op2)==8){
        op1=SignEx(32,8,op2);
    } else {
        op1=SignEx(32,16,op2);
    }
}
```

例子：

MOVSX AX,BH

执行前：AX:×××× BH:9E

执行后：AX:FF9E BH:9E

MOVSX EDX,mem16

执行前：EDX:×××××××× mem:742D

执行后：EDX:0000742D mem:742D

时钟：(假设主频为20MHz)

MOVSX reg,reg：3个时钟周期；0.15μs。

MOVSX reg,mem:6个时钟周期;0.3μs。

标志影响:无。

异常:

Memory( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟8086模式异常:同 ADC 指令。

注意:

该指令是80386新增的。

MOVZX

传送和零扩展

格式:

MOVZX op_1 , op_2

reg16 , reg8

reg16 , mem8

reg16 , reg16

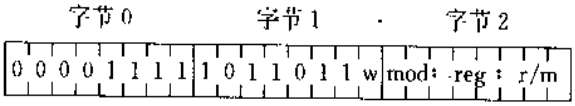
reg16 , mem16

reg32 , reg8

reg32 , mem8

reg32 , reg16

reg32 , mem16



说明:

字 op_1修改为字节 op_2的零扩展;双字 op_1修改为字 op_2的零扩展。若 op_1和 op_2均为字,则发生通常的传送。

操作:

```
if(Length(op1)==16{
    if(Length(op2)==8){
        op1=ConCat(16,00h,8,op2);
    } else {
        op1=op2;  /* 正常传送! */
    }
} else {          /*Length(op1)==32*/
    if(Length(op2)==8){
        op1=ConCat(32,000000h,8,op2);
    } else {
        op1=ConCat(32,0000h,16,op2);
    }
}
```

例子:

MOVZX AX,BH

执行前:AX:××××　BH:9E

执行后:AX:009E　BH:9E

MOVZX EDX,mem16

执行前:EDX:××××××××　mem:742D

执行后:EDX:0000742D　mem:742D

时钟:(假设主频为20MHz)

MOVZX reg,reg:3个时钟周期;0.15μs。

MOVZX reg,mem:6个时钟周期;0.3μs。

标志影响:无。

异常:

Memory( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟8086模式异常:同 ADC 指令。

注意:

该指令是80386新增的。

MUL

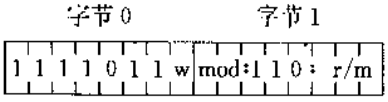
无符号乘

格式:

MUL op_1

reg

mem



说明:

MUL 指令只取一个操作数 op_1,作为被乘数。而乘数、积和标志位,请看下表:

被乘数 op_1 的长度	乘数	积	积的长度	若高位	CF	OF
8	AL	AX	16	AH 为全0	0	0
				AH 不为全0	1	1
16	AX	DX:AX	32	DX 为全0	0	0
				DX 不为全0	1	1
32	EAX	EDX:EAX	64	EDX 为全0	0	0
				EDX 不为全0	1	1

操作：

```
if(Length(op1)==8){
    AX=AL*op1;
    if(AH==00h){
        CF=0;OF=0;
    } else {
        CF=1;OF=1;
    }
} else if(Length(op1)==16){
  DX=(AX*op1)>>16;
  AX=FFFFh & (AX*op1);
  if(DX==0000h){
        CF=0; OF=0;
    } else {
        CF=1; OF=1;
    }
} else{  /* Length(op1)==32 */
    EDX=(EAX*op1)>>32;
    EAX=FFFFFFFFh & (EAX * op1);
    if(EDX==00000000h){
        CF=0;OF=0;
    } else {
        CF=1;OF=1
    }
}
SF=ZF=AF=PF=UNDEFINED;
/*
    "*"号视同无符号乘
*/
```

例子：

```
MUL  BX
    执行前:DX:×××× AX:229A BX:A431
    执行后:DX:1631 AX:477A BX:A431
MUL  mem8
    执行前:AH:×× AL:0A mem:12
    执行后:AH:00 AL:B4 mem:12
```

```
OSZAPC
××××××
1××××1

××××××
0××××0
```

时钟：(假设主频为20MHz)

MUL reg：9～41个时钟周期；0.45～2.05μs。

MUL mem：12～44个时钟周期；0.6～2.2μs。

标志影响：

影响 OF，CF，SF，ZF，AF 和 PF 为未定义。

其它标志不变。

异常：

Memory( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常:同 ADC 指令。

NEG

二进制补码

格式:

NEG op_1

mem

reg

字节0	字节1
1 1 1 1 0 1 1 w	mod 0 1 1 r/m

说明:

NEG 求给定操作数的二进制补码。0减 op_1,结果再放入 op_1。若操作数 op_1 的原值为0,则 CF 为0;否则 CF 为1。

操作:

```
CF=op1!=0;
op1=0-op1;
/* "-"号视同SUB指令 */
OF=CarryO(MSB)^CarryO(MSB-1);
SF=B(MSB,op1);
ZF=op1==0;
AF=CarryO(3);
PF=~(B(7,op1)^B(6,op1)^…^(B(0,op1)));
```

例子:

	OSZAPC
NEG EAX	
执行前:EAX:01234567	××××××
执行后:EAX:FEDCBA99	010111
NEG mem8	
执行前:Mem:00	××××××
执行后:Mem:00	001010

时钟:(假设主频为20MHz)

NEG reg:2个时钟周期;0.1μs。

NEG mem:7个时钟周期;0.35μs。

标志影响:

影响 OF,CF,SF,ZF 和 PF。CF 如上所述。

其它标志不变。

异常:

Memory( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟8086模式异常:同 ADC 指令。

NOP

空 操 作

格式：

NOP

1	0	0	1	0	0	0	0

(XCHG AX,AX 的简化)

说明：

NOP 什么也不做。该指令唯一的效果便是改变了下一条要执行的指令的地位。

操作：

;

例子：

NOP;什么也没发生。

时钟：(假设主频为20MHz)

3个时钟周期;0.15μs。

标志影响：无。

注意：

NOP 对于定时回路非常有用(它要用3个时钟脉冲)。NOP 指令还用来调整边界，尤其是对于执行次数最多的代码(如循环)，总是将它们对准于四字节的边界，以加快执行速度。

NOT

非

格式：

NOT op_1

reg

mem

字节 0								字节 1							
1	1	1	1	0	1	1	w	mod	0	1	0	r/m			

说明：

NOT 指令求操作数 op_1的逻辑非或反码。结果仍放入 op_1，不影响任何标志。

操作：

$op_1 = \sim op_1$;

/*

"～"号定义为逐位求反，请看下表：

a_i	r_i
0	1
1	0

其中 a_i 为 op_1的第 i 位，r_i 为结果的第 i 位。位下标 i 的取值范围为从0至 MSB。

*/

例子：

NOT EAX，

执行前：EAX：01234567

执行后：EAX：FEDCBA98

NOT mem8

执行前：mem：72

执行后：mem：8D

时钟：（假设主频为20MHz）

NOT reg：2个时钟周期；0.1μs。

NOT mem：7个时钟周期；0.35μs。

标志影响：无。

异常：

Memory( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：同 ADC 指令。

OR

或

格式：

OR op_1 , op_2	字节 0	字节 1	后续字节
reg , reg	0 0 0 0 1 0 d w	mod: reg : r/m	
mem , reg	0 0 0 0 1 0 0 w	mod: reg : r/m	
reg , mem	0 0 0 0 1 0 1 w	mod: reg : r/m	
reg , imm	1 0 0 0 0 0 s w	mod:0 0 1 : r/m	立即数字节
mem , imm	1 0 0 0 0 0 s w	mod:0 0 1 : r/m	立即数字节
AL/AX/EAX , imm（短编码）	0 0 0 0 1 1 0 w	立 即 数	字 节

说明：

执行 op_1 和 op_2 的逻辑或，结果放入 op_1。

操作：

$op_1=op_1|op_2$；

OF=0；

SF=B(MSB,op_1)；

ZF=$op_1==0$；

CF=0；

AF=UNDEFINED;
PF=~(B(7,op_1)^B(6,op_1)^…^B(0,op_1));
/*
"|"号定义为逐位"或",请看下表:

a_i	b_i	r_i
0	0	0
0	1	1
1	0	1
1	1	1

其中 a_i、b_i 为 op_1和 op_2的第 i 位,r_i 为结果的第 i 位。位下标 i 取值范围从0到 MSB。
*/

例子:

OR EAX,EBX　　OSZAPC

执行前:EAX:01234567　EBX:76543210　　××××××

执行后:EAX:77777777　EBX:76543210　　000×10

OR mem16,7878h

执行前:Mem:1F90

执行后:Mem:7FF8

OR AL,0FFh

执行前:AL:A2

执行后:AL:FF

时钟:(假设主频为20MHz)

OR reg,reg:2个时钟周期;0.1μs。

OR mem,reg:7个时钟周期;0.35μs。

OR reg,mem:6个时钟周期;0.3μs。

OR reg,imm:2个时钟周期;0.1μs。

OR mem,imm:7个时钟周期;0.35μs。

标志影响:

影响 OF(==0),CF(==0),SF,ZF 和 PF。AF 为未定义。

其他标志不变。

异常:

Memory( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟模式异常:同 ADC 指令。

OUT

向端口输出

格式:

OUT op_1 , op_2

imm8, AL

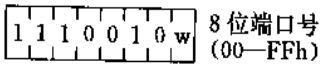
imm8, AX

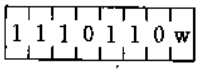
imm8, EAX

DX , AL

DX , AX

DX , EAX





DX 规定的端口可从0000到 FFFFh。

说明：

OUT 指令将字节，字或双字的数据，从 AL，AX 或 EAX 传送到指定的端口，注意从 00F8h 到00FFh 的 I/O 端口地址被 Intel 保留。

操作：

port(op_1)←op_2；

例子：

OUT 0C3h，AX

执行前：AX：1C9B port(00C3)：××××

执行后：AX：1C9B port(00C3)：1C9B

OUT DX，EAX

执行前：EAX：96238887 DX：4684

port(4684)：× × × × × × × ×

执行后：EAX：96238887 DX：4684

port(4684)：9 6 2 3 8 8 8 7

时钟：(假设主频为20MHz)

实地址模式

op_1为 imm8：10个时钟周期；0.5μs。

op_1为 DX：11个时钟周期；0.55μs。

保护模式，CPL≤IOPL

op_1为 imm8：4个时钟周期；0.2μs。

op_1为 DX：5个时钟周期；0.25μs。

保护模式，CPL>IOPL

op_1为 imm8：24个时钟周期；1.2μs。

op_1为 DX：25个时钟周期；1.25μs。

虚拟8086模式：

op_1为 imm8：24个时钟周期；1.2μs。

op_1为 DX：25个时钟周期；1.25μs。

标志影响：无。

异常：

Access( )；

实地址模式异常：同 IN 指令。

保护模式异常：同 IN 指令。

虚拟8086模式异常：同 IN 指令。

OUTS/OUTSB/OUTSW/OUTSD

输出串操作数

格式：

OUTSB
OUTSW
OUTSD

0	1	1	0	1	1	1	w

OUTS reg1 , mem2
DX , mem8
DX , mem16
DX , mem32

说明：

一字节，二字节，或四字节的数据，将从位于[ESI]的存储器传送到由 DX 指定的端口。操作数的长度是由操作码本身(OUTSB,OUTSW，或 OUTSD)或 mem2决定的。在上述传送完成之后，据操作数长度的不同，ESI 加减1，2，或4，以指向串中的下一个元素。若方向标志 DF 为0则加，否则减。

使用的端口被规定在寄存器 DX 之中。不允许使用立即数端口格式。

mem2规定了操作数长度。输出数据的内存地址总是存放在寄存器 ESI 中。可以在 mem2 中规定使用在[ESI]前面的段超越前缀。

OUTS 可与前缀 REP 连用。在与 REP 连用的情况下，可以输出 ECX 个字节、字、或双字。详阅 REP 指令。

操作：

```
if(OPCODE==OUTSB){
    port(DX)=(byte)DS:[ESI];
    delta=1;
  } else if(OPCODE==OUTSW){
      port(DX)=(word)DS:[ESI];
      delta=2;
  } else if(OPCODE==OUTSD){
      port(DX)=(dword)DS:[ESI];
      delta=4;
  } else {          /* OUTS */
        /*注意：默认的 SegReg(mem2)为 DS */
      if(Length(mem2)==8){
          port(DX)=(byte)SegReg(mem2):[ESI];
          delta=1;
      }else if(Length(mem2)==16){
          port(DX)=(word)SegReg(mem2):[ESI];
          delta=2;
      }else if(Length(mem2)==32){
          port(DX)=(dword)SegReg(mem2):[ESI];
          delta=4;
```

```
        }
    }
    if(DF==1){
        delta=-delta;
    }
    ESI=ESI+delta;
```

例子：

OUTSW

执行前：ESI:6240　[ESI]:FFF6

DX:004C　DF:0

port(DX):××××

执行后：ESI:6242　[ESI−2]:FFF6

DX:004C　DF:0

port(DX):FFF6

REP　OUTS DX,S1　　;S1是一字节串

执行前：ESI:00000040　　DX:20

[ESI]:F2　　port(DX):××

[ESI+1]:FF

[ESI+2]:00

DF:0　　CX:3

执行后：ESI:00000043　　DX:20

[ESI−3]:F2　　port(DX,1):F2

[ESI−2]:FF　　port(DX,2):FF

[ESI−1]:00　　port(DX,3):00

DF:0　　CX:0

时钟：(假设主频为20MHz)

实地址模式：

不重复：14个时钟周期；0.7μs

重复：(12+5n)个时钟周期：(0.6+0.25n)μs。

保护模式，CPL≤IOPL：

不重复：8个时钟周期；0.4μs。

重复：(6+5n)个时钟周期；(0.3+0.25n)μs。

保护模式，CPL>IOPL：

不重复：28个时钟周期；1.4μs。

重复：(26+5n)个时钟周期；(1.3+0.25n)μs。

虚拟8086模式：

不重复：28个时钟周期；1.4μs。

重复：(26+5n)个时钟周期；(1.3+0.25n)μs。

(以上 n 为迭代执行次数)

标志影响：无。

异常：

Memory( );

AccessIO( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 INS 指令。

虚拟8086模式异常:同 INS 指令。

POP

从堆栈中弹出

格式:

		字节 0	字节 1
POP	op_1	1 0 0 0 1 1 1 1	mod 0 0 0 r/m
	reg		
	mem		
	reg (短编码)	0 1 0 1 1 reg	

说明:

POP 将位于栈顶(SS:[ESP])的字或双字弹出,放入 op_1。然后栈指针 ESP 加2(对于16位操作数),或加4(对于32位操作数)。

操作:

```
if(Length(op1)==32){
    op1=SS:[ESP];
    ESP=ESP+4;
} else {
    op1=SS:[ESP];
    ESP=ESP+2;
}
```

例子:

POP AX

执行前: AX:× × × ×　SS:[ESP]　:F340　ESP :F4320530

执行后: AX:F 3 4 0　SS:[ESP-2]:F340　ESP :F4320532

时钟:(假设主频为20MHz)

4个时钟周期;0.2μs。

标志影响:无。

异常:

Memory( );

Stack( );

实地址模式异常:同 PUSH 指令。

保护模式异常:同 PUSH 指令。

虚拟8086模式异常:同 PUSH 指令。

注意：

当弹出16位操作数时，要当心栈对不准四字节边界。如不对准四字节边界，会影响执行速度。

POPA/POPAD

将栈顶的8个字或双字弹出，放入通用寄存器

格式：

POPA

POPAD

0	1	1	0	0	0	0	1

说明：

POPA/POPAD 将栈顶的8个字或双字弹出，放入8个通用寄存器。这样，POPA/POPAD 一下子代替了8个连续的 POP 指令。8个通用寄存器按下列顺序存放栈顶元素：

POPA：DI，SI，BP，SP，BX，DX，CX，AX（16位操作数）

POPAD：EDI，ESI，EBP，ESP，EBX，EDX，ECX，EAX（32位操作数）

注意，弹至 ESP 的值丢掉了。

操作：

```
if (OPCODE==POPAD){
  EDI = SS:[ESP+0];
  ESI = SS:[ESP+4];
  EBP = SS:[ESP+8];
  tmp = SS:[ESP+12];/* 弹至 ESP 的值丢掉了 */
  EBX = SS:[ESP+16];
  EDX = SS:[ESP+20];
  ECX = SS:[ESP+24];
  EAX = SS:[ESP+28];
  ESP = ESP+32;
} else {
  DI  = SS:[ESP+0];
  SI  = SS:[ESP+2];
  BP  = SS:[ESP+4];
  tmp = SS:[ESP+6];/* 弹至 SP 的值丢掉了 */
  BX  = SS:[ESP+8];
  DX  = SS:[ESP+10];
  CX  = SS:[ESP+12];
  AX  = SS:[ESP+14];
  ESP = ESP+16;
}
```

例子：

POPA

执行前：AX：××××　BX：××××　CX：××××　DX：××××

SP：7236　BP：××××　SI：××××　DI：××××

SS：[07236]：42FF　SS：[07238]：AA2C

SS：[0723A]：B290　SS：[0723C]：8861

O S Z A P C

× × × × × ×

SS:[0723E]:0000　SS:[07240]:0002

SS:[07242]:FFFF　SS:[07244]:2133

执行后：AX:2133　BX:FFFF　CX:0002　DX:0000　　UUUUUU

SP :7246　BP:B290　SI:AA2C　DI:42FF

SS:[07236]:42FF　SS:[07238]:AA2C

SS:[0723A]:B290　SS:[0723C]:8861

SS:[0723E]:0000　SS:[07240]:0002

SS:[07242]:FFFF　SS:[07244]:2133

时钟：

4个时钟周期；在主频为20MHz时，为0.2μs。

标志影响：无。

异常：

Stack( )；

实地址模式异常：同 POPF 指令。

保护模式异常：同 POPF 指令。

虚拟8086模式异常：同 POPF 指令。

POPF/POPFD

从栈中弹出至标志位

格式：

POPF

POPFD

1	0	0	1	1	1	0	1

说明：

将当前在栈顶的数据复制到32位的标志寄存器内。传送完成之后，栈指针加2(16位操作)或加4(32位操作)，使之指向新的栈顶。

32位标志双字如下图：

31	30 ……	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			V	R			N	IOP		O	D	I	T	S	Z		A		P		C
B	0 ……	0	0	M	F	0	T	L		F	F	F	F	F	F	F	0	F	0	F	1

POPF 只替换上图之低16位。

I/O 特权级标志只有在当前特权级为0时才能被替换。若当前特权级不为0，I/O 特权级标志不被替换。也不产生异常。

中断启用标志，只有在当前特权级与当前 I/O 特权级相等或更高时才可以改变。否则，中断启用标志不改变，也不发生异常。

操作：

```
RF=0;
NT=([ESP] &.00004000h)>>14;
if(CPL==00b){
```

```
    IOPL=([ESP] & 00003000h)>>12;
}
OF=([ESP]& 00000800h)>>11;
DF=([ESP]& 00000400h)>>10;
if(CPL<=IOPL){
    IF=([ESP]& 00000200h)>>9;
}
TF=([ESP]&00000100h)>>8;
SF=([ESP]&00000080h)>>7;
ZF=([ESP]&00000040h)>>6;
AF=([ESP]&00000010h)>>4;
PF=([ESP]&00000004h)>>2;
CF=([ESP]&00000001h)>>0;
if(OPCODE==POPFD){
    ESP=ESP+4;
} else {
    ESP=ESP+2;
}
```

例子：

POPFD ;特权级为0

		I I O O																
		V M	R F	N T	P L	P L	O F	D F	I F	T F		S F	Z F	A F	P F	1	C F
执行前:ESP:	20143110	0	×	×	×	×	×	×	×	×	×	×	0	×	0	×	1	×
[ESP]:	FFF601A5																	
执行后:ESP:	20143114	U	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	1

时钟：(假设主频为20MHz)

4个时钟周期；0.2μs。

标志影响：如上所述。

异常：

Stack( )；

实地址模式异常：

若堆栈操作数的任何部分是在堆栈段 SS 中大于 FFFFh 的有效地址，则产生中断12。

保护模式异常：

对于在 SS 段中的内存操作数，若操作数的任何部分是在超出了段界的有效地址，则产生错误码为0000h 的堆栈异常(异常12)。若内存操作数是在标为“不存在”的 SS 段内，则产生错误码为段选择符的堆栈异常。页故障引起错误码为故障码的、页故障线性地址在寄存器 CR2 中的页故障异常。

虚拟8086模式异常：

在8086模式，若 IOPL<3，该指令引起错误码为0000h 的通用保护异常(异常13)，以允许仿真。

对于在堆栈段内的操作数，若其任何部分是在超出段界的有效地址，则引起错误码为0000h 的堆栈异常。异常处理是在保护模式以及特权级为0的情况下进行的。页故障异常同保护模式。

PUSH

压　栈

格式：

		字节0	字节1
PUSH	op_1	1 1 1 1 1 1 1 1	mod:1 1 0: r/m
	reg		
	mem	0 1 1 0 1 0 s 0	立即数据字节
	imm		
	reg（短编码）	0 1 0 1 0: reg	

说明：

将 op_1 压入堆栈，并使栈指针 ESP 减2(16位操作)或减4(32位操作)，这样，刚压入的 op_1 便处在了新的栈顶位置。op_1 若为立即数，总被当作32位的操作数处理，尽管它也许只是个8位符号立即数。(请阅第2章)。

操作：

```
if(Length(op1)==32){
    SS:[ESP-4]=op1;
    ESP=ESP-4;
} else { /* Length (op1)==16 */
    SS:[ESP-2]=op1;
    ESP=ESP-2;
}
```

例子：

PUSH　24h

执行前：ESP:FFFFF320　SS:[FFFFF31C]:××××××××

执行后：ESP:FFFFF31C　SS:[FFFFF31C]:00000024

PUSH　AX

执行前:ESP:FFFFF31C　SS:[FFFFF31A]:××××

AX:F799

执行后:ESP:FFFFF31A　SS:[FFFFF31A]:F799

AX:F799

时钟：(假设主频为20MHz)

PUSH imm:2个时钟周期;0.1μs。

PUSH reg:2个时钟周期;0.1μs。

PUSH mem:5个时钟周期;0.25μs。

标志影响：无。

异常：

Stack( );

Memory( );

实地址模式异常：

无。但遇栈空间不足的情况(比如,在SP=1时想压入一个字,或在SP=1,2或3时想压入一个双字)会引起停机。因为这时连中断所必需(将CS,IS,及FLAGS压栈)的栈空间也无法满足。既然不能产生一个中断来处理这类异常,最终只得停机了事。

保护模式异常:

对于在CS,DS,ES,FS,或GS段内的内存操作数,若源操作数是在一不可读的代码段中,或操作数的任何部分是在超出了段界的有效地址,则产生错误码为0000h的通用保护异常。对于在SS段内的操作数,若其任何部分是在超出了段界的有效地址,则引起错误码为0000h的堆栈异常(异常12)。若内存操作数是在标为"不存在"的SS段内,则引起错误码为段选择符的堆栈异常。若内存操作数是在标为"不存在"的DS,ES,FS,或DS段内,则引起错误码为段选择符的"不存在"异常。页故障引起错误码为故障码的,页故障线性地址在寄存器CR2中的页故障异常。

虚拟8086模式异常:

对堆栈段中的操作数,若其任何部分是在超出了段界的有效地址,则引起错误码为0000h的堆栈异常。异常处理是在保护模式以及特权级为0的情况下进行的。页故障异常同保护模式。

注意:

在压入16位操作数时,要小心防止栈对不准四字节边界。如对不准四字节边界,会影响执行速度。

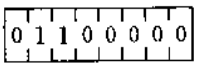
PUSHA/PUSHAD

所有通用寄存器压栈

格式:

PUSHA

PUSHAD



说明:

PUSHA/PUSHAD将8个字通用寄存器或8个双字寄存器的值一下子压入栈内,从而替代了8个连续的PUSH指令。这8个寄存器的顺序为:AX,CX,DX,BX,SP,BP,SI,DI;或者EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI。

注意:对SP或ESP,压栈的值是其原值。

被PUSHA压栈的寄存器的顺序与后续的POPA指令的寄存器顺序正好匹配。

操作:

```
if(OPCODE==PUSHAD){
    SS:[ESP-4] =EAX;
    SS:[ESP-8] =ECX;
    SS:[ESP-12]=EDX;
    SS:[ESP-16]=EBX;
    SS:[ESP-20]=ESP; /*注意:此为ESP的原值*/
    SS:[ESP-24]=EBP;
    SS:[ESP-28]=ESI;
    SS:[ESP-32]=EDI;
    ESP=ESP-32;
```

```
} else {
    SS:[ESP-2] =AX;
    SS:[ESP-4] =CX;
    SS:[ESP-6] =DX;
    SS:[ESP-8] =BX;
    SS:[ESP-10]=SP;/*注意:此为SP的原值*/
    SS:[ESP-12]=BP;
    SS:[ESP-14]=SI;
    SS:[ESP-16]=DI;
    ESP=ESP-16;
}
```

例子:

```
PUSHA                                                    O S Z A P C
    执行前:AX:0000   BX:1111   CX:2222   DX:3333          × × × × × ×
           SP:4444   BP:5555   SI:6666   DI:7777
           SS:[04442]:××××   SS:[04440]:××××
           SS:[0443E]:××××   SS:[0443C]:××××
           SS:[0443A]:××××   SS:[04438]:××××
           SS:[04436]:××××   SS:[04434]:××××
    执行后:AX:0000   BX:1111   CX:2222   DX:3333          U UUUUU
           SP:4434   BP:5555   SI:6666   DI:7777
           SS:[04442]:0000   SS:[04440]:1111
           SS:[0443E]:2222   SS:[0443C]:3333
           SS:[0443A]:4444   SS:[04438]:5555
           SS:[04436]:6666   SS:[04434]:7777
```

时钟:

18个时钟周期;在主频为20MHz 情况下,执行时间为0.9μs。

标志影响:无

异常:

Stack( );

实地址模式异常:同 PUSH 指令。

保护模式异常:

对于在 SS 段内的操作数,若其任何部分是在超出了段界的有效地址,则引起错误码为0000h 的堆栈异常(异常12)。若内存操作数是在标为"不存在"的 SS 段内,则引起错误码为段选择符的堆栈异常。页故障引起错误码为故障码的,页故障线性地址在寄存器 CR2中的页故障异常。

虚拟8086模式异常:同 PUSH 指令。

PUSHF/PUSHFD

将标志压栈

格式:

PUSHF
PUSHFD

1	0	0	1	1	1	0	0

说明：

栈指针减2(16位标志字)或减4(32位标志双字)，以形成新的栈顶。然后当前的标志字寄存器或标志双字寄存器被复制到新的栈顶。

EFLAGS 寄存器的图解：

31	30 ……	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			V	R			N	IOP		O	D	I	T	S	Z		A		P		C
0	0 ……	0	0	M	F	0	T	L		F	F	F	F	F	F	F	0	F	0	F	1

操作：

```
if(PUSHF){
    ESP=ESP-2;
    SS:[ESP]=(0<<15)|(NT<<14)|
        (IOPL<<12)|(OF<<11)|(DF<<10)|(IF<<9)|
        (SF<<7)|(ZF<<6)|(0<<5)|(AF<<4)|
        (0<<3)|(PF<<2)|(1<<1)|(CF<<0);
} else { /* PUSHFD */
    ESP=ESP-4;
    SS:[ESP]=(0000h<<16)|(0<<15)|(NT<<14)|
        (IOPL<<12)|(OF<<11)|(DF<<10)|(IF<<9)|
        (SF<<7)|(ZF<<6)|(0<<5)|(AF<<4)|
        (0<<3)|(PF<<2)|(1<<1)|(CF<<0)
}
```

例子：

```
                                    I I
                                    O O
                                  V R N P P O D I T S Z   A   P   C
                                  M F T L L F F F F F 0 F 0 F 1 F
PUSHFD
    执行前:[ESP-4]:xxxxxxxx ESP:FFF42104     0 0 1 0 0 0 1 1 0 1 1 0 0 0 0 1 1
    执行后:[ESP]   :00000046C3 ESP:FFF42100  0 0 1 0 0 0 1 1 0 1 1 0 0 0 0 1 1
```

时钟：

4个时钟周期；在主频为20MHz 情况下的执行时间为0.2μs。

标志影响：无。

异常：

Stack( )；

实地址模式异常：同 PUSH 指令。

保护模式异常：同 PUSHA/PUSHAD 指令。

虚拟8086模式异常：同 PUSH 指令。

注意：

请看第6章的例1。

RCL

带 CF 的左旋

格式：

		字节 0	字节 1	后续字节
RCL op_1 , op_2		1 1 0 1 0 0 0 w	mod:0 1 0: r/m	
reg , 1				
mem , 1				
reg , imm8		1 1 0 0 0 0 0 w	mod:0 1 0: r/m	8 位立即数
mem , imm8				
reg , CL		1 1 0 1 0 0 1 w	mod:0 1 0: r/m	
mem , CL				

说明：

op_1连带进位标志被多次左旋：每左旋一次，CF 移至 op_1的最低有效位，op_1的最高有效位移入 CF，op_1的其余位都左移一位。

左旋次数：op_2的低5位（高位无效）取（操作数长度+1）的模，便是 RCL 执行左旋的次数。

若左旋的次数为1，则溢出标志位作如下变化：如果在 RCL 指令执行后 CF 的值与结果的最高位相等，则溢出标志位清0；否则溢出标志位置1。若左旋次数不为1，则溢出标志位无意义。

操作：

```
if(Length(op1)==8){
    cnt=(op2&01Fh)%9;
      if(cnt>0){
          temp=(op1<<cnt)|(CF<<(cnt-1))|(op1>>(9-cnt));
          CF=(op1>>(8-cnt))&01;
      }
} else    if(Length(op1)==16){
    cnt=(op2&01Fh)%17;
      if (cnt>0){
        temp=(op1<<cnt)
              (CF<<(cnt-1));|(op1>>(17-cnt));
          CF=(op1>>(16-cnt))&01;
          op1= temp;
      }
} else { /* Length(op1)==32 */
    cnt=op2&01Fh;
    if(cnt>0){
      temp=(op1<<cnt)|(CF<<(cnt-1))|(op1>>(33-cnt));
      CF=(op1>>(32-cnt))&01;
      op1=temp;
    }
}
if (op2==1){
```

```
    OF=B(MSB,op1)^CF;
  } else {
      OF=UNDEFINED;
  }
```

例子：

RCL EAX,1	O S Z A P C
执行前：EAX:01234567	× × × × × 1
执行后：EAX:02468ACF	0 U U U U U 0
RCL mem8,CL	
执行前：mem:9B CL:11	× × × × × 0
执行后：mem:4D CL:11	× U U U U 1

时钟：(假设主频为20MHz)

op_1为 reg:9个时钟周期;0.45μs。

op_1为 mem:10个时钟周期;0.5μs。

标志影响：

影响 CF 和 OF。

其它标志不变。

异常：

Memory( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟8086模式异常:同 ADC 指令。

注意：

左旋次数为0的话,进位标志不变。

32位或33位的 RCL 不能执行,因为左旋的次数只在0～31范围内。

RCR

带 CF 的右旋

格式：

			字节 0	字节 1	
RCR	op_1	, op_2	1 1 0 1 0 0 0 w	mod:0 1 1 : r/m	
	reg	, 1			
	mem	, 1			
	reg	, imm8	1 1 0 0 0 0 0 w	mod:0 1 1 : r/m	8 位立即数
	mem	, imm8			
	reg	, CL	1 1 0 1 0 0 1 w	mod : 011 : r/m	
	mem	, CL			

说明：

op_1连带进位标志 CF 被多次右旋:每右旋一次,CF 进入 op_1的最高位,op_1的最低位进入

CF，op_1的其余各位都右移一位。

右旋的次数：op_2的低5位(高位无效)取(操作数长度+1)的模，便是 RCR 执行右旋的次数。

若右旋次数为1，则溢出标志作如下变化：若结果的两位最高位的值相等，则溢出标志清0，否则溢出标志为1。若右旋次数不为1，则溢出标志无意义。

操作：

```
if(op2==1){
  OF=B(MSB,op1)^CF;
} else {
  OF=UNDEFINED;
}
if(Length(op1)==8){
    cnt=(op2 & 01Fh)%9;
    if(cnt>0){
        temp=(op1>>cnt)|(CF<<(8-cnt))|(op1<<(9-cnt));
        CF= (op1>>(cnt-1)) & 01;
        op1= temp;
    }
} else if(Length(op1)==16){
    cnt=(op2&01Fh)%17;
    if(cnt>0){
        temp=(op1>>cnt)|(CF<<(16-cnt)) |(op1<<(17-cnt));
        CF= (op1>>(cnt-1)) & 01;
        op1= temp;
    }
} else { /* Length(op1)==32 */
    cnt=op2 & 01Fh;
    if(cnt>0){
        temp=(op1>>cnt)|(CF<<(32-cnt))|(op1<<(33-cnt));
        CF= (op1>>(cnt-1))& 01;
          op1=temp;
    }
}
```

例子：

RCR mem32,1	OSZAPC
执行前：mem:01234567	×××××0
执行后：mem:0091A2B3	1UUUUU1
RCR EDI,CL	
执行前：EDI:A3214551 CL:11	×××××0
执行后：EDI:45515190 CL:11	×UUUU1

时钟：

同 RCL 指令。

标志影响：

同 RCL 指令。

异常：

同 RCL 指令。

注意：

若右旋次数为0，进位标志 CF 不变。

32位或33位的 RCR 不能执行，因为右旋的次数只能在0到31范围内。

REP

重　复

格式：

REP String_Operation(串操作指令)

INS

MOVS

OUTS

STOS

1	1	1	1	0	0	1	0

说明：

REP 是一个串操作的前缀，它使其后的串操作重复执行 ECX 次。

操作：

```
while(ECX!=0){
    ECX=ECX-1;
    String_Operation; /* 串操作 */
}
OF,SF,ZF,AF,PF,CF--取决于执行的 String_Operation
```

例子：

```
REP MOVSD
    执行前：ESI          :00000008  EDI       :00000008
            ECX          :00000002  [ESI]     :01234567
            ES:[EDI]     :xxxxxxxx  [ESI-4]   :F421890A
            ES:[EDI-4]   :xxxxxxxx  DF        :1
    执行后：ESI          :00000000  EDI       :00000010
            ECX          :00000000  [ESI+8]   :01234567
            ES:[EDI+8]   :01234567  [ESI+4]   :F421890A
            ES:[EDI+4]   :F421890A  DF        :1
```

异常：

REP 指令本身不会产生异常，但后面的串操作可能产生异常。

REPE/REPZ

相等时重复/为0时重复

格式：

REPE/REPZ String_Operation(串操作)

CMPS

SCAS

1	1	1	1	0	0	1	1

说明：

REPE 是一个串操作的前缀，它使其后的串操作重复执行 ECX 次，或者执行到 ZF 为0时为止。REPZ 与 REPE 同义。

操作：

```
while(ECX!=0){
    ECX=ECX-1;
    String_Operation;
    if(ZF==0)
    break;
}
OF,SF,ZF,AF,PF,CF=取决于执行的String_Operation
```

例子：

REPE CMPSB

执行前：	ESI: 00000008	EDI	:16	ECX	:04
	[ESI] : 01	[ESI+1]	:23	[ESI+2]	:45
	[ESI+3] : 67	ES:[EDI]	:01	ES:[EDI+1]	:23
	ES:[EDI+2]:44	ES:[EDI+3]	:67	DF	:0
执行后：	ESI: 0000000B	EDI	:19	ECX	:01
	[ESI-3] : 01	[ESI-2]	:23	[ESI-1]	:45
	[ESI] : 67	ES:[EDI-3]	:01	ES:[EDI-2]	:23
	ES:[EDI-1]:44	ES:[EDI]	:67	DF	:0

异常：

REPE/REPZ 后面的串操作可能会引起异常。

注意：

请看第6章的例5。

REPNE/REPNZ

不等时重复/非0时重复

格式：

REPNE/REPNZ String_Operation(串操作)

CMPS

SCAS

1	1	1	1	0	0	1	0

说明：

REPNE 是一个串操作的前缀，它使其后的串操作重复执行 ECX 次或者执行到 ZF 为1时为止。REPNZ 与 REPNE 同义。

操作：

```
while(ECX!=0){
    ECX=ECX-1;
    String_Operation; */串操作*/
    if(ZF==1)break;
}
OF,SF,ZF,AF,PF,CF=取决于执行的String_Operation
```

例子：

REPNE CMPSB

执行前：	ESI：00000008	EDI	:16	ECX	:04
	[ESI] : 01	[ESI+1]	:23	[ESI+2]	:45
	[ESI+3] : 67	ES:[EDI]	:01	ES:[EDI+1]	:23
	ES:[EDI+2]:44	[EDI+3]	:67	DF	:0
执行后：	ESI：00000009	EDI	:17	ECX	:03
	[ESI-1] : 01	[ESI]	:23	[ESI+1]	:45
	[ESI+2] : 67	ES:[EDI-1]	:01	ES:[EDI]	:23
	ES:[EDI+1]:44	ES:[EDI+2]	:67	DF	:0

异常：

REPNE/REPNZ 后面的串操作可能会引起异常。

注意：

请看第6章的例5。

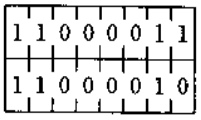
RET

返　回

格式：

NEAR RET

RET imm16



16位立即数字节偏移量

说明：

RET 使得指令执行回到相应的 CALL 指令后面的那条指令。RET 从栈顶找出它该返回的地址。也可以说，RET 执行了一次通过在栈顶的指针的间接跳转。即将使用的 EIP 值是在栈顶找出来的。

可选的 imm16的值加到 ESP 的值上面，相当于以栈顶削去了若干字节的参数。

操作：

```
EIP   =Pop( );
if(NUMOPS==1){
    ESP=ESP+imm16;
}
```

例子：

RET

执行前：EIP:804231AA　SS:[ESP]:032BBD4C

ESP:04260988

执行后：EIP:032BBD4C

ESP:0426098C

时钟：

NEAR RET：(10+m)个时钟周期。

RET imm16:(10+m)个时钟周期。

其中 m 为取指及开始执行目标指令所需的时间。

标志影响：

无。除非在保护模式下发生任务切换。

异常：

Stack( )；

CodeReference( )

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟8086模式异常:同 ADC 指令。

ROL

左　旋

格式：

			字节0	字节1	后续字节
ROL	op_1	, op_2			
	reg	, 1	1 1 0 1 0 0 0 w	mod:0 0 0: r/m	
	mem	, 1			
	reg	, imm8	1 1 0 0 0 0 0 w	mod:0 0 0: r/m	8 位立即数
	mem	, imm8			
	reg	, CL	1 1 0 1 0 0 1 w	mod:0 0 0: r/m	
	mem	, CL			

说明：

op_1被多次左旋:每左旋一次,op_1的最高位被移入最低有效位和进位标志,其余的位都左移一位,最后,最低有效位与进位标志相等。

左旋的次数:取决于 op_2的低5位。左旋次数在0到31范围内。

若左旋次数为1,则溢出标志作如下变化:若移位后的进位标志与 op_1的高位相等,则溢出标志清0,否则溢出标志为1。若左旋次数不为1,溢出标志无意义。

操作：

```
if(Length(op1)==8{
    cnt=(op2&01Fh)%8;
    if(cnt>0){
      op1=(op1<<cnt)|(op1>>(8-cnt));
      CF=B(0,op1);
    }
} else if(Length(op1)==16){
```

```
        cnt =(op2&01Fh)%16;
        if(cnt>0){
          op1=(op1<<cnt)|(op1>>(16-cnt));
          CF=B(0,op1);
        }
    } else { /* Length (op1)==32 */
        cnt=(op2&01Fh);
        if(cnt>0){
            op1=(op1<<cnt)|(op1>>(32-cnt));
            CF=B(0,op1);
        }
    }
    if(op2==1){
        OF=B(MSB,op1)^CF;
    } else {
        OF=UNDEFINED;
    }
```

例子：

ROL AL,1　　OSZAPC

执行前：AL:A6　　×××××

执行后：AL:4D　　1UUUU1

ROL BP,CL

执行前：BP:6279　CL:0A　　×××××

执行后：BP:E589　CL:0A　　×UUUU1

时钟：(假设主频为20MHz)

op_1为 reg:3个时钟周期;0.15μs。

op_1为 mem:7个时钟周期;0.35μs。

标志影响：

影响 CF,OF。

其它标志不变。

异常：

Memory( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟模式异常:同 ADC 指令。

注意：

若左旋次数为0,ROL 不改变进位标志或溢出标志。

ROR

右　旋

格式：

		字节0	字节1
ROR	op_1 , op_2	1 1 0 1 0 0 0 w	mod:0 0 1 : r/m
	reg , 1		
	mem , 1		
	reg , imm8	1 1 0 0 0 0 0 w	mod:0 0 1 : r/m　8位立即数
	mem , imm8		
	reg , CL	1 1 0 1 0 0 1 w	mod:0 0 1 : r/m
	mem , CL		

说明：

op_1被多次右旋：每右旋一次，op_1的最低有效位被移入最高有效位和进位标志，其余的位都右移一位。最后，最高有效位与进位标志相等。

右旋的次数：取决于op_2的低5位。右旋次数在0到31范围内。

若右旋次数为1，则溢出标志作如下变化：若移位后op_1的最高两位的值相等则溢出标志清0，否则溢出标志为1，。若右旋次数不为1，溢出标志无意义。

操作：

```
if(Length(op1)==8){
    cnt=op2%8;
    if(cnt>0){
        op1=(op1>>cnt)|(op1<<(8-cnt));
        CF=B(7,op1);
    }
} else if(Length(op1)==16){
    cnt=op2%16;
    if(cnt>0){
        op1=(op1>>cnt)|(op1<<(16-cnt));
        CF=B(15,op1);
    }
} else { /* Length(op1)==32 */
    cnt=op2%32;
    if(cnt>0){
        op1=(op1>>cnt)|(op1<<(32-cnt));
        CF=B(32,op1);
    }
}
if(op2==1){
    OF=B(MSB,op1)^B(MSB-1,op1);
} else {
    OF=UNDEFINED;
}
```

例子：

	OSZAPC
ROR mem16,1	
执行前：mem:E4A2	××××××
执行后：mem:7251	1 UUUU 0
ROR EDI,7	

执行前：EDI:07BDAF21　　× × × × × ×

执行后：EDI:42OF7B5E　　× UUUU 0

时钟：

同 ROL 指令。

标志影响：

同 ROL 指令。

异常：

同 ROL 指令。

注意：

若右旋次数为0，ROR 不影响 CF 或 OF。

SAHF

将 AH 寄存器存入标志寄存器

格式：

SAHF

1	0	0	1	1	1	1	0

说明：

AH 被传送到标志字的低字节。AH 的第7位被传送至符号标志位，AH 的第6位被传送至零标志位，AH 的第4位被传送至辅助进位标志位，AH 的第2位被传送至奇偶标志位，AH 的第0位被传送至进位标志位。

下图便是 EFLAGS 寄存器的低8位：

7	6	5	4	3	2	1	0
S	Z		A		P		C
F	F	0	F	0	F	1	F

操作：

SF=(AH &80h)>>7;
ZF=(AH &40h)>>6;
AF=(AH & 10h)>>4;
PF=(AH & 04h)>>2;
CF=(AH & 01h)>>0;

例子：

SAHF　　OSZAPC

执行前：AH:10×1×0×0b　　× × × × × ×

执行后：AH:10×1×0×0b　　U1 0 1 0 0

时钟：(假设主频为20MHz)

2个时钟周期；0.1μs。

标志影响：

影响 SF,ZF,AF,PF,CF。

异常:无

注意:

请阅第6章程序7。

SAL/SHL

算术左移/逻辑左移

格式:

			字节0	字节1	后续字节
SAL/SHL	op_1	, op_2	1 1 0 1 0 0 0 w	mod:1 0 0: r/m	
	reg	, 1			
	mem	, 1			
	reg	, imm8	1 1 0 0 0 0 0 w	mod:1 0 0: r/m	8 位立即数
	mem	, imm8			
	reg	, CL	1 1 0 1 0 0 1 w	mod:1 0 0: r/m	
	mem	, CL			

说明:

SAL 与 SHL 两者同义,没有区别。

每左移一位,最低有效位补0,以最高有效位移出的值赋予 CF。

op_2的低5位规定了左移的位数,所以左移的位数在0到31范围内。

若左移位数为1,溢出标志作如下变化:若移位后的 CF 与 op_1的高位相等,则溢出标志清0;否则溢出标志为1。若左移位数不为1,溢出标志无意义。

操作:

```
cnt=op2 & 01Fh;
if(cnt<Length(op1)){
    if(cnt>0){
        CF=B((Length(op1)-cnt),op1);
    }
    op1=op1<<cnt;
} else {
    CF=0;
    op1=0;
}
if(op2==1){
    OF=B(MSB,op1)^CF;
} else {
    OF=UNDEFINED;
}
SF=B(MSB,op1);
ZF=op1==0;
AF=UNDEFINED;
PF=~(B(7,op1)^B(6,op1)^…^B(0,op1));
```

例子：

SAL BX,1

执行前：BX:ADF3

执行后：BX:5BE6

SHL mem32,CL

执行前：mem:A6341209　CL:09

执行后：mem:68241200　CL:09

```
OSZAPC
××××××
100×01
OSZAPC
××××××
×00×10
```

时钟：(假设主频为20MHz)

op_1为 reg:3个时钟周期;0.15μs。

op_1为 mem:7个时钟周期;0.35μs。

标志影响：

影响 CF,OF。其它标志不变。

异常：

Memory( );

实地址模式异常:同 ADC 指令。

保护模式异常:同 ADC 指令。

虚拟8086模式异常:同 ADC 指令。

注意：

0次左移不改变任何标志。

SAR

算术右移

格式：

SAR	op_1	op_2	字节0	字节1	后续字节
	reg	1	1 1 0 1 0 0 0 w	mod:1 1 1: r/m	
	mem	1			
	reg	imm8	1 1 0 0 0 0 0 w	mod:1 1 1: r/m	8位立即数
	mem	imm8			
	reg	CL	1 1 0 1 0 0 1 w	mod:1 1 1: r/m	
	mem	CL			

说明：

该指令使op_1右移,执行的结果又存入op_1。右移的位数由op_2的低5位规定,所以op_1右移的次数在0到31范围内。

每右移一次(即右移一位),空出的高位被赋以op_1最高有效位(符号位)的值,而进位标志则赋以从op_1的最低有效位移出的值。

若右移次数为1,则溢出标志清0。否则溢出标志不变。

每右移一位,相当于被2除一次。但对于负数来说,用这种除法却越除越大,越趋向负无穷,

这是和 IDIV 不同的。

操作：

```
cnt=op2 & 01Fh;
if(cnt>0){
    CF=B(cnt-1,op1);
}
if(cnt<Length(op1)){
    if(Length(op1)==8){
        if(B(7,op1)==1){
            op1=(op1>>cnt)|(FFh<<(8-cnt));
        } else {
            op1=(op1>>cnt);
        }
    } else if(Length(op1)==16){
        if(B(15,op1)==1){
          op1=(op1>>cnt)|(FFFFh<<(16-cnt));
        } else {
          op1=(op1>>cnt);
          }
      } else { /* Length(op1)==32 */
        if(B(31,op1)==1){
          op1=(op1>>cnt)|(FFFFFFFFh<<(32-cnt));
        } else {
          op1=(op1>>cnt);
        }
    }
} else {
    if(B(Length(op1)-1,op1)==1){
      CF=1;
      if(Length(op1)==8){
          op1=FFh;
      } else {  /* must be16 */
          op1=FFFFh;
      }
    } else {
      CF=0;
      if(Length(op1)==8){
          op1==00h;
      } else {  /* must be 16 */
          op1=0000h;
      }
    }
}
if(op2==1){
    OF=0;
} else {
    OF=OF;
}
SF=B(MSB,op1);
ZF=op1==0;
AF=UNDEFINED;
PF=~(B(7,op1)^B(6,op1)^…^B(0,op1));
```

例子：

	OSZAPC
SAR mem8,1	
执行前：mem:84	××××××
执行后：mem:C2	010×00
SAR EDI,CL	
执行前：EDI:A6341209　CL:13	××××××
执行后：EDI:FFFFF4C6　CL:13	×10×11

时钟：

同 SAL 指令。

标志影响：

同 SAL 指令。

异常：

同 SAL 指令。

注意：

右移0次不改变任何标志位。

SAR 可用于快速执行无符号整数除以2的幂的除法。请看第6章程序1。

SBB

带借位减法

格式：

SBB op_1 , op_2	字节 0	字节 1	后续字节
reg , reg	0 0 0 1 1 0 d w	mod: reg : r/m	
mem , reg	0 0 0 1 1 0 0 w	mod: reg : r/m	
reg , mem	0 0 0 1 1 0 1 w	mod: reg : r/m	
reg , imm	1 0 0 0 0 0 s w	mod:0 1 1 : r/m	立即数字节
mem , imm	1 0 0 0 0 0 s w	mod:0 1 1 : r/m	立即数字节
AL/AX/EAX, imm (短编码)	0 0 0 1 1 1 0 w	立　即　数	字　节

说明：

op_1减去(op_2+CF)，结果放入 op_1。进位标志(借位标志)指出该减法有借位。这相当于：

$op_1 = op_1 - (op_2 + CF)$。

SBB 被用作多字节，多字，或多双字减法的一部分。

操作：

```
op1=op1-(op2+CF);
OF=BorrowO(MSB)^BorrowO(MSB-1);
SF=B(MSB,op1);
ZF=op1==0;
AF=BorrowO(3);
PF=~(B(7,op1)^B(6,op1)^…^B(0,op1));
CF=BorrowO(MSB);
```

/*

"-"号的定义请看下表，它的操作是从低到高逐位进行的。其中 x_i，y_i 为操作数 op_1 和 op_2 的第 i 位。b_{i-1} 是第 i－1 位操作时借的，也就是从第 i 位借出的。r_i 是结果的第 i 位，b_i 为第 i 位的借出。位下标 i 的取值范围从0到 MSB。若 i＝0，$b_{i-1}=b_{-1}=CF$。即 SBB 在0位操作时，减数中的 b_{-1} 取 CF(借位标志)之值。

*/

x_i	y_i	b_{i-1}	y_i	b_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

注意，减法也可定义为加法，将此表和前面的那个加法表比较之后就可以发现：

$x_i=a_i$

$y_i=\sim b_i$

$b_i=C_i$

80386的进位标志也可以当作借位标志。

例子：

SBB EAX，EBX　　OSZAPC

执行前：EAX：0743CE21　EBX：8D4BC956　　×××××1

执行后：EAX：79F804CA　EBX：8P4BC956　　010111

SBB mem8，1Ah

执行前：mem：72　　×××××0

执行后：mem：58　　000100

时钟：(假设主频为20MHz)

SBB reg，reg：2个时钟周期；0.1μs。

mem，reg：7个时钟周期；0.35μs。

reg，mem：6个时钟周期；0.3μs。

reg，imm：2个时钟周期；0.1μs。

mem，imm：7个时钟周期；0.35μs。

标志影响：

影响 OF，CF，SF，ZF，AF 和 PF。

其它标志不变。

异常：

Memory( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：同 ADC 指令。

SCAS/SCASB/SCASW/SCASD

扫描串操作数

格式：

SCASB
SCASW
SCASD
SCAS mem1
mem8
mem16
mem32

1	0	1	0	1	1	1	w

说明：

SCAS 从 AL，AX，或 EAX 中减去 ES:[EDI]（分别对应于字节操作，字操作，或双字操作）。不保留相减结果，只修改标志。EDI 也改指串中的下一个元素。

操作数 mem1，或者操作码本身（SCASB，SCASW，SCASD）可以决定操作数的长度。EDI 根据不同的操作数长度，相应地加减1，2，或4，以指向串中下一个元素。当方向标志 DF 为0时则加，否则减。

SCAS 操作可与前缀 REPE/REPZ 或 REPNE/REPNZ 连用。若加前缀 REPE/REPZ，当 ECX 不为0且串元素与AL，AX，或 EAX 相等（ZF==1）时，重复执行SCAS 指令。用这种方法，SCAS 可在串中找出第一个与 AL，AX 或 EAX 不匹配的元素。若加前缀 REPNE/REPNZ，当 ECX 不为0且串元素与 AL，AX，或 EAX 不等（ZF==0）时，重复执行 SCAS。用这种方法，SCAS 可在串中找出第一个与 AL，AX，或 EAX 匹配的元素。详阅 REPE 和 REPNE 的说明。

若出现 mem1的格式，它仅供汇编以决定操作数长度。而串操作数总是取自 ES:[EDI]，且不允许使用其它的段超越前缀。

操作：

```
if((Length(mem1)==8)||(OPCODE==SCASB)){
    delta= 1;
    temp=AL-(byte)ES:[EDI];
} else if((Length(mem1)==16)||(OPCODE==SCASW)){
    delta=2;
    temp=AX-(word)ES:[EDI];
} else { /*(Length(mem1)==32)||(OPCODE==SCASD)*/
    delta=4;
    temp=EAX-(dword)ES:[EDI];
}
/*"-"号视同于SUB指令。*/
OF=BorrowO(MSB)^BorrowO(MSB-1);
SF=B(MSB,temp);
ZF=temp==0;
AF=BorrowO(3);
PF=~(B(7,temp)^B(6,temp)^...^B(0,temp));
CF=BorrowO(MSB);
```

```
if(DF==1){
    delta=-delta;
}
EDI=EDI+delta;
```

例子：

```
SCASW                                                    O S Z A P C
    执行前：AX:0123          EDI:0016                    × × × × × ×
            ES:[EDI]:3210    D F:0
    执行后：AX:0123          EDI:0018                    0 1 0 0 0 1
            ES:[EDI-2]:3210  D F:0
REPNE  SCAS S1     ;注意 S1是一字节串                    O S Z A P C
    执行前：ECX:00000008  DF:0                           × × × × × ×
            AL:72  EDI:0008
            ES:[EDI]  :23  ES:[EDI-1]:23
            ES:[EDI-2]:72  ES:[EDI-3]:AA
    执行后：ECX:00000005  DF:0                           0 0 1 0 1 0
            AL:72  EDI:0005
            ES:[EDI+3]:23  ES:[EDI+2]:23
            ES:[EDI+1]:72  ES:[EDI]:AA
```

时钟：(假设主频为20MHz)

不重复执行：7个时钟周期；0.35μs。

重复执行：(5+8n)个时钟周期；(0.25+0.4n)μs。

(n 为迭代执行次数)

标志影响：无。

异常：

Memory( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：同 ADC 指令。

注意：请阅第6章程序5。

SETcc

根据条件码的字节置位

格式：

SETcc op_1

mem8

reg8

字节 0	字节 1	字节 2
0 0 0 0 1 1 1 1	1 0 0 1 c c c c	mod:0 0 0: r/m

说明：

字节1中的 cccc 为条件码，共有16种状态：

cccc	助记符	对应的操作
0000	SETO	若 OF==1(溢出)则 op_1置1，否则置0
0001	SETNO	若 OF==0(无溢出)则 op_1置1，否则置0
0010	SETB	若 CF==1(低于/不高于等于/进位)
	SETNAE	则 op_1置1，否则置0
	SETC	
0011	SETNB	若 CF==0(不低于/高于等于/无进位)
	SETAE	则 op_1置1，否则置0
	SETNC	
0100	SETE	若 ZF==1(等于/0)则 op_1置1，否则置0
	SETZ	
0101	SETNE	若 ZF==0(不等/非0)则 op_1置1
	SETNZ	否则置0
0110	SETBE	若 CF==1||ZF==1(低于等于/不高于)
	SETNA	则 op_1置1，否则置0
0111	SETNBE	若 CF==0&&ZF==0(不低于等于/高于)
	SETA	则 op_1置1，否则置0
1000	SETS	若 SF==1(符号/负)则 op_1置1，否则置0
1001	SETNS	若 SF==0(无符号/正)则 op_1置1，否则置0
1010	SETP	若 PF==1(操作结果中为"1"的位数为偶)则 op_1置1，
	SETPE	否则置0
1011	SETNP	若 PF==0(操作结果中为"1"的位数为奇)则 op_1置1，
	SETPO	否则置0
1100	SETL	若 SF!=OF(小于/不大于等于)则 op_1置1，
	SETNGE	否则置0
1101	SETNL	若 SF==OF(不小于/不大于等于)则 op_1置1，
	SETGE	否则置0
1110	SETLE	若 SF!==OF||ZF==1(小于等于/不大于)
	SETNG	则 op_1置1，否则置0
1111	SETNLE	若 ZF==0&&SF==OF(不小于等于/大于)
	SETG	则 op_1置1，否则置0

以上，关于大于和小于请参考符号整数比较，关于高于和低于请参考无符号整数比较。

若在同种状态下，汇编给出了多个助记符(如 cccc 为0111，助记符为 SETNBE 和 SETA)，这些助记符是同义的，它们之间没有区别，只是为使用方便而已，SETcc 只允许使用一字节的操作数。

操作：

```
if(ConditionCode){
    op1=1;
} else {
    op1=0;
}
```

例子：

```
SETNO   mem8                                  O S Z A P C
    执行前：mem8:× ×                          1 ×××××
    执行后：mem8:0 0                          1 UUUU U
SETNBE AL
    执行前：AL   :× ×                         ××1××0
    执行后：AL   :0 1                         UU 1 UU 0
```

时钟：(假设主频为20MHz)

SETcc reg8：4 个时钟周期；0.2μs。

SETcc mem8：5 个时钟周期；0.25μs。

标志影响：无。

异常：

Code Reference( )；

Memory( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ADC 指令。

注意：

该指令为 80386 新增的。

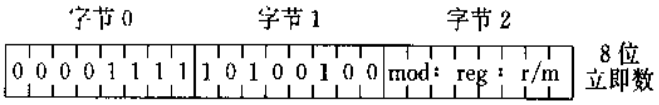
SHLD

双精度左移

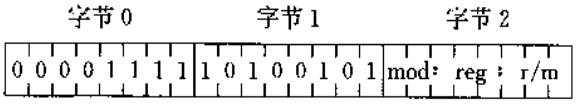
格式：

SHLD op_1, op_2, op_3

reg16, reg16, imm8
mem16, reg16, imm8
reg32, reg32, imm8
mem32, reg32, imm8



reg16, reg16, CL
mem16, reg16, CL
reg32, reg32, CL
mem32, reg32, CL



说明：

op_3 的低 5 位规定了左移的位数。也就是左移位数在 0 到 31 范围内。SHLD 指令操作 op_1 : op_2，结果放回 op_1。op_2 的高位移入 op_1 左移之后空出的低位。但 op_2 保持不变。进位标志 CF 变成了最后从 op_1 的最高有效位移出的值。

若 op_3 规定的左移位数为 0，SHLD 指令相当于一个空操作(Nop)。如果左移位数为 1，溢出标志发生如下变化：若结果的最高有效位与进位标志不一致，则溢出标志置位，否则清 0。如果规定的左移位数大于操作数长度，则标志位以及 op_1 中的结果为无意义。

操作：

```
cnt=op3 & 01Fh;
if (cnt==0)  {
    /* nop */
} else if (cnt>Length (op1)) {
    op1=OF=SF=ZF=PF=CF=AF=UNDEFINED;
} else {
```

```
    CF=B(Length (op1)-cnt, op1);
    op1=(op1<<cnt)|(op2>>(Length(op2)-cnt));
    OF=B(MSB,op1)^CF;
    SF=B(MSB,op1);
    ZF=op1==0;
    PF=(B(7,op1)^B(6,op1)^…^B(0,op1));
    AF=UNDEFINED;
}
```

例子：

SHLD mem16，AX，3　　O S Z A P C

　执行前：mem：0084　AX：00F3　　× × × × × ×

　执行后：mem：0420　AX：00F3　　0 0 0 × 0 0

SHLD DI，SI，CL

　执行前：DI：1209　SI：8552　CL：OB　　× × × × × ×

　执行后：DI：4C2A　SI：8552　CL：OB　　0 0 0 × 0 0

时钟：（假设主频为 20MHz）

op_1 为 reg：3 个时钟周期；0.15μs。

op_1 为 mem：7 个时钟周期；0.35μs。

标志影响：

影响 CF，PF，SF，ZF，OF。AF 为无定义。

其它标志不变。

异常：

Memory ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ADC 指令。

注意：

左移 0 位不改变任何标志！

该指令为 80386 新增的。

请看第 6 章例 6b。

SHR

逻辑右移

格式：

SHR op_1，	op_2	字节 0	字节 1	
reg，	1	1 1 0 1 0 0 0 w	mod 1 0 1 r/m	
mem，	1			
reg，	imm8	1 1 0 0 0 0 0 w	mod 1 0 1 r/m	8 位立即数
mem，	imm8			

reg, CL

mem, CL

1 1 0 1 0 0 1 w	mod 1 0 1 r/m

说明：

op_2 的低 5 位(高位无效)规定了右移位数，也就是说右移位数规定在 0 到 31 范围内。

op_1 被右移指定的位数，空出的高位变成 0。进位标志变成最后从最低有效位移出的值。结果放在 op_1 中。

若右移位数为 1，且结果的最高两位不等，溢出标志置位，否则清 0。若右移位数不为 1，则溢出标志无意义。

操作：

```
cnt←op2 & 01Fh;
if(cnt-<Length (op1)) {
    if (cnt>0){
        CF←B(cnt-1,op1);
    }
    op1←op1>>cnt;
}else {
    op1←0;
    CF←0;
}
if (op2==1) {
    OF=B(MSB, op1)^B(MSB-1, op1);
}else{
    OF=UNDEFINED;
}
    SF=B(MSB,op1);
    ZF=op1==0;
    AF=UNDEFINED;
    PF=~(B(7,op1)^B(6,op1)^…^B(0,op1));
```

例子：

	OSZAPC
SHR DX,1	
执行前：DX：84	××××××
执行后：DX：42	000×10
SHR mem 32, CL	
执行前：mem：F42BBD51 CL：0C	××××××
执行后：mem：000F42BB CL：0C	×00×11

时钟：

同 SHL 指令。

标志影响：

同 SHL 指令。

异常：

同 SHL 指令。

注意：

右移 0 位不改变任何标志。

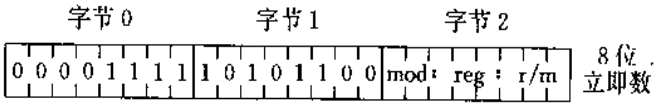
请看第 6 章程序 6b。

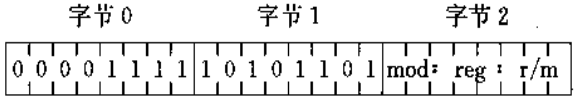
SHRD

双精度右移

格式：

SHRD op_1,	op_2,	op_3
reg16,	reg16	imm8
mem16,	reg16,	imm8
reg32,	reg32,	imm8
mem32,	reg32,	imm8
reg16,	reg16,	CL
mem16,	reg16,	CL
reg32,	reg32,	CL
mem32,	reg32,	CL





说明：

op_3 的低 5 位(高位无效)规定了右移的位数，即右移位数规定在 0 到 31 范围内。

SHRD 指令操作 op_1 : op_2，结果放回 op_1。op_1 右移之后空出的高位，用 op_2 的低位填充。进位标志 CF 变成最后一个从 op_1 的最低有效位移出的值。op_2 保持不变。

若 op_3 规定的右移位数为 0，SHRD 相当于一空操作(Nop)。若结果的最高两位不等，则溢出标志置位，否则清 0。

若在 op_3 中规定的右移位数大于操作数长度，标志位以及 op_1 中的结果均无意义。

操作：

```
cnt=op3 & 01Fh;
if(cnt==0) {
    /* nop */
}else if (cnt>Length (op1)) {
    op1=OF=SF=ZF=PF=CF=AF=UNDEFINED;
} else {
    CF=B((cnt-1),op1);
    op1=(op1>>cnt) | (op2<<(Length(op2)-cnt));
    OF=B(MSB,op1)^B(MSB-1,op1);
    SF=B(MSB,op1);
    ZF=op1==0;
    PF=~(B(7,op1)^B(6,op1)^...^B(0,op1));
    AF=UNDEFINED;
}
```

例子：

SHRD mem16, BP,2 　　OSZAPC

执行前：mem : 7ACO　　BP : F43P　　× × × × × ×

执行后：mem : 5EBO　　BP : F43D　　1 0 0 × 0 0

SHRD EAX, mem32, CL

执行前：EAX : 663B982F

mem：9963120A　CL：OA　　× × × × × ×

执行后：EAX：82998EE6

mem：9963120A　CL：OA　　1 1 0 × 1 0

时钟：

同 SHLD 指令。

标志影响：

同 SHLD 指令。

异常：

同 SHLD 指令。

注意：

右移 0 位不改变任何标志。

该指令是 80386 新增的。

请看第 6 章程序 6b。

STC

置 进 位

格式：

STC　　1 1 1 1 1 0 0 1

说明：

将进位标志置位。

操作：

CF=1；

例子：

STC

执行前：CF：0

执行后：CF：1

时钟：（假设主频为 20MHz）

2 个时钟周期；0.1μs。

标志影响：

影响 CF。其它标志不变。

STD

置方向标志

格式：

STD　　1 1 1 1 1 0 0 1

说明：

将方向标志置1。在方向标志置1之后，串操作指令将是减少它们的索引寄存器(ESI和EDI)的值。

操作：

DF←1；

例子：

STD

执行前：DF：0

执行后：DF：1

时钟：(假设主频为20MHz)

2个时钟周期；0.1μs。

标志影响：

影响DF。其它标志不变。

STI

置中断标志

格式：

STI 1 1 1 1 1 0 1 1

说明：

将中断启用标志置位。80386在下一条指令执行完后将会响应中断。

若当前的任务没有更改中断标志的权限，便产生一无定义操作码故障。关于特权级别的描述详见第5章。

操作：

IF←1；

例子：

STI

执行前：IF：0

执行后：IF：1

时钟：(假设主频为20MHz)

3个时钟周期；0.15μs。

标志影响：

影响IF。其它标志不变。

异常：

IOPL Scusitive ( )；

实地址模式异常：无。

保护模式异常：

若CPL>IOPL，则产生错误码为0000h的通用保护异常(异常13)。

虚拟8086模式异常：

若 IOPL<3,则产生错误码为 0000h 的通用保护异常(异常 13)。

在虚拟 8086 模式中,由于 IOPL<3 而浮获该条指令,可以允许保护模式的管理程序模拟由虚拟 8086 软件观察到的 IF 的状态。

STOS/STOSB/STOSW/STOSD

存储串操作数

格式:

STOSB

STOSW 1 0 1 0 1 0 1 w

STOSD

STOS mem1

mem8

mem16

mem32

说明:

STOS 将 AL 中的一个字节数据,或 AX 中的一个字数据,或 EAX 中的双字数据,传送至 ES:[EDI]。

在上述数据传送结束后,EDI 根据不同的操作数长度,相应地加减 1,2 或 4,以指向下一个存储串的单元。当方向标志 DF 为 0 时则加,否则减。操作数长度是由操作数 mem_1 或操作码(STOSB,STOSW,STOSD)本身决定的。

mem_1 仅用于规定操作数长度。ES:[EDI]作为数据传送的目的地是不会改变的,且不可能有其它的段超越前缀。AL,AX,EAX 作为被传送数据的源也是不会改变的。

STOS 指令可与 REP 前缀连用。这样可以实现利用寄存器 AL,AX,EAX 的内容使一个串初始化。

操作:

```
if ((Length(mem1)==8) || (OPCODE==STOSB)){
    delta=1;
    ES:[EDI]=AL;
} else if ((Length(mem1)==16) || (OPCODE==STOSW)){
    delta=2;
    ES:[EDI]=AX;
} else {  /* (Length(mem1)==32) || (OPCODE==STOSD) */
    delta=4;
    ES:[EDI]=EAX;
}
if (DF==1) {
    delta=-delta;
}
EDI=EDI+delta;
```

例子:

STOSW

执行前：EDI：0052　　ES：[EDI]：××××
　　AX：FFFF　　DF：0
执行后：EDI：0054　　ES：[EDI]：FFFF
　　AX：FFFF　　DF：0

时钟：

4 个时钟周期；若主频为 20MHz，则为 0.2μs。

标志影响：无。

异常：

Memory ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ADC 指令。

注意：

请阅第 6 章的程序 1。

SUB

减

格式：

SUB op_1, op_2	字节 0	字节 1	后续字节
reg, reg	0 0 1 0 1 0 d w	mod reg r/m	
mem, reg	0 0 1 0 1 0 0 w	mod reg r/m	
reg, mem	0 0 1 0 1 0 1 w	mod reg r/m	
reg, imm	1 0 0 0 0 0 s w	mod 1 0 1 r/m	立即数字节
mem, imm	1 0 0 0 0 0 s w	mod 1 0 1 r/m	立即数字节
AL/AX/EAX, imm（短编码）	0 0 1 0 1 1 0 w	立即数字节	

说明：

从 op_1 中减去 op_2，结果放入 op_1。

操作：

```
op1=op1-op2;
OF=BorrowO(MSB)^ BorrowO(MSB-1);
SF=B(MSB,op1);
ZF=op1==0;
AF=Borrow O(3);
PF=~(B(7,op1)^B(6,op1)^…^B(0,op1));
CF=BorrowO(MSB);
/*
        "-"是由低到高逐位计算的，请看下表：
```

x_i	y_i	b_{i-1}	r_i	b_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

其中 x_i 为 op_1 的第 i 位，y_i 为 op_2 的第 i 位。b_{i-1}是第 i－1 位操作时从第 i 位借的。r_i 是结果的第 i 位，b_i 为第 i 位的借出，位下标 i 的取值范围从 0 到 MSB。对于 SUB 指令，若 $i=0$，$b_{i-1}=b_{-1}=0$。

注意：减法也可以定义为加法。与前面的加法表作一比较，不难看出：

$x_i=a_i$

$y_i=\sim b_i$

$b_i=c_i$

80386 的进位标志也是借位标志。

*/

例子：

```
SUB EAX, EBX                                          OSZAPC
    执行前：EAX：0743CE21      EBX：8D4BC956          ××××××
    执行后：EAX：79F804CB      EBX：8D4BC956          000101
SUB mem8, 3A
    执行前：Mem：72                                   ××××××
    执行后：Mem：38                                   000100
```

时钟：（假设主频为 20MHz）

```
SUB reg,  reg  :  2 个时钟周期；0.1μs。
    mem,  reg  :  7 个时钟周期；0.35μs。
    reg,  mem  :  6 个时钟周期；0.3μs。
    reg,  imm  :  2 个时钟周期；0.1μs。
    mem,  imm  :  7 个时钟周期；0.35μs。
```

标志影响：

影响 OF，AF，CF，SF，ZF 和 PF。其余标志不变。

异常：

Memory ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ADC 指令。

TEST

逻辑比较

格式：

	字节0	字节1	后续字节
TEST op_1, op_2	1 0 0 0 0 1 0 w	mod: reg : r/m	
reg, reg			
reg, mem			
reg, imm	1 1 1 1 0 1 1 w	mod:0 0 0 : r/m	立即数据字节
mem, imm			
AL/AX/EAX,imm (短编码)	1 0 1 0 1 0 0 w		立即数据字节

说明：

执行 op_1 和 op_2 的逻辑与(AND)，但只修改标志，不保存结果。

操作：

```
temp=op1 & op2;
OF=0;
SF=B(MSB, temp);
ZF=temp==0;
AF=UNDEFINED;
PF=~(B(7,temp)^B(6,temp)^…^B(0,temp));
CF=0;
/*
    "&"(AND)是从低到高逐位操作的，其定义请看下表：
```

a_i	b_i	r_i
0	0	0
0	1	0
1	0	0
1	1	1

其中 a_i 为 op_1 的第 i 位，b_i 为 op_2 的第 i 位，r_i 为结果的第 i 位。位下标 i 的取值范围从 0 到 MSB。
*/

例子：

	OSZAPC
TEST EAX, EBX	
执行前：EAX：01234567 EBX：76543210	××××××
执行后：EAX：01234567 EBX：76543210	001×10
TEST mem 8, 75	
执行前：Mem：7F	××××××
执行后：Mem：7F	000×00

时钟：(假设主频为 MHz)

TEST reg, reg： 2 个时钟周期； 0.1μs。

TEST mem, reg： 7 个时钟周期； 0.35μs。

TEST reg, mem： 6 个时钟周期； 0.3μs。

TEST reg, imm：2个时钟周期；0.1μs。

TEST mem, imm：7个时钟周期；0.35μs。

标志影响：

影响OF(==0),CF(==0),SF,ZF和PF。

其它标志不变。

异常：

Memory ( )；

实地址模式异常：同ADC指令。

保护模式异常：同ADC指令。

虚拟8086模式异常：同ADC指令。

XCHG

交　换

格式：

XCHG op_1, op_2

reg, reg

mem, reg

reg, mem

字节0　字节1

1	0	0	0	0	1	1	w	mod:	reg:	r/m

AL/AX/EAX, reg

(字或双字)

(短编码)

1	0	0	1	0:	reg

说明：

op_2和op_1的内容交换。op_1和op_2可以是字节、字、或双字，但两者长度必须相等。

如果op_1或op_2中之一为mem，总线传送总是象带有LOCK前缀一样(LOCK#信号被置起)，即使没有指定LOCK亦是如此。XCHG对于信号灯操作是很有用的。

操作：

temp←op_1；

op_1←op_2；

op_2←temp；

例子：

XCHG AX, BX

执行前：AX：0A8F　BX：9042

执行后：AX：9042　BX：0A8F

SCHG mem8, AL

执行前：AL：4A　mem：0F

执行后：AL：0F　mem：4A

时钟：(假设主频为20MHz)

XCHG reg, reg　3个时钟周期；0.15μs。

XCHG reg，mem　5 个时钟周期；　0.25μs。

XCHG mem，reg　5 个时钟周期；　0.25μs。

标志影响：无

异常：

Memory ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ADC 指令。

XLAT/XLATB

查表变换

格式：

XLATB

XLAT mem 8

1	1	0	1	0	1	1	1

说明：

指令开头，AL 被认为一基址为[EBX]的表的无符号索引，从该表中读出一个表项。XLAT 将 AL 的内容置换成位于[EBX+AL]的表项。

AL 总是取无符号值。表的基址永远是[EBX]，与 mem8 无关。然而 mem8 允许规定一个段超越前缀，以替换默认的 DS：[EBX]。

操作：

```
if (OPCODE==XLATB) {
    AL←DS：[EBX+ConCat(000000h,AL)];
}else {
    AL←SegReg(mem8)：[EBX+ConCat(000000h,AL)];
}
```

例子：

XLATB

执行前：EBX　　：F000　　AL：30

　　[EBX+AL]　：7D

执行后：EBX　　：F000　　AL：7D

时钟：(假设主频为 20MHz)

5 个时钟周期；0.25μs。

标志影响：无。

异常：

Memory ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ADC 指令。

XOR

异　或

格式：

XOR op_1,	op_2	字节 0	字节 1	后续字节
reg,	reg	0 0 1 1 0 0 d w	mod reg r/m	
mem,	reg	0 0 1 1 0 0 0 w	mod reg r/m	
reg,	mem	0 0 1 1 0 0 1 w	mod reg r/m	
reg,	imm	1 0 0 0 0 0 s w	mod 1 1 0 r/m	立即数字节
mem,	imm	1 0 0 0 0 0 s w	mod 1 1 0 r/m	立即数字节
AL/AX/EAX,	imm	0 0 1 1 0 1 0 w	立　即　数	字　节

说明：

执行 op_1 和 op_2 之间的异或，结果放入 op_1。

操作：

```
op1=op1^op2;
OF=0;
SF=B(MSB,op1);
ZF=op1==0;
AF=UNDEFINED;
PF=~(B(7,op1)^B(6,op1)^...^B(0,op1));
CF=0;
/*
    "^"(异或)是从低位到高位逐位执行的，请看下表：
```

a_i	b_i	r_i
0	0	0
0	1	1
1	0	1
1	1	0

```
    ai 和 bi 为 op1 和 op2 的第 i 位，ri 为结果的第 i 位。位下标 i 的取值范围从 0 到 MSB。
*/
```

时钟：(假设主频为 20MHz)

XOR reg, reg：　2 个时钟周期；0.1μs。

XOR mem, reg：7 个时钟周期；0.35μs。

XOR reg, mem：6 个时钟周期；0.3μs。

XOR reg, imm：2 个时钟周期；0.1μs。

XOR mem, imm：7 个时钟周期；0.35μs。

标志影响：

OF(==0), CF(==0), SF, ZF 和 PF 受到影响。

其它标志不变。

异常：

Memory ( );

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ADC 指令。

3.5 多段类指令

关于单段编程模式(所有的段都映象到一个单一的段内)或多段编程模式，我们在第 2 章中已经讨论过了。这两种模式可以由操作系统或编程者来选择。但中断和异常处理却是例外，系统设计者总是用多段模式对它们进行处理。关于中断和异常详阅第 5 章。

本节描述支持多段编程模式的指令。如果你对此不感兴趣，可以跳过这一节。

第 5 章和第 6 章详细讨论这些多段类指令的使用，并且给出了许多实例。

本节中所有的指令都涉及到向段寄存器中装入选择符。在第 2 章中我们讨论了选择符和段寄存器，由于向段寄存器中装入了选择符，使得段成为一个可寻址的段。于是这个段内的数据可以直接通过偏移量进行访问(请看第 2 章)。常说的取指的操作，便是由 EIP 寄存器给出偏移量，对于代码段进行的一次访问。

3.5.1 函数定义

在本章的开头，我们定义过几种函数。使用这些函数大大简化了指令的说明和例子。在这里我们为同样的目的再引入一个程序。这个程序在本节内当作一个函数使用。

```
IntEXecTable (type, imm) {
  if (type==SEGMENT) {
    return ((word) IDTR.Base+4)
  } else {/* OFFSET */
    return ((dword) IDTR.Base +0)
  }
}
  /* 此 IntExecTable 程序返回被调用的中断服务程序的段或偏移量。而实际上，产生中断的语义(se-
     mantics)(门，任务开关，特权级别的变化)远比这里描述的要复杂得多，这待我们留到第 6 章去讨
     论。该程序被下面的中断所调用：
       Interrupt (imm,1)
     其中 imm 是中断表中的中断号，是个立即数。第二个操作数 1，表示必须进行特权级检查。
       */
```

3.5.2 异常程序

下面还有两个专门用于多段指令的异常程序：

```
(1) CodesegmentLoad ( ) {
  /* 在加载 CS 描述符时所有可能发生的段异常都包括在第 5 章中定义的 CSDescriptorLoad( )程序
     之中。请详阅第 5 章。 */
  }

(2) SegmentLoad ( ) {
  /* 在加载一个描述符时所有可能产生的段异常都包括在第 5 章中定义的 Descriptor.Load( )程序
     中。详阅第 5 章。  */
  }
```

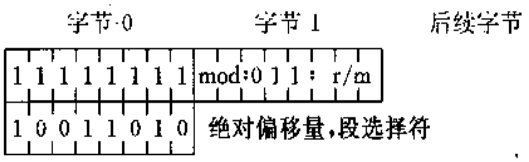
3.5.3 多段类指令集

CALL

调用过程

格式：

CALL op_1
mem48
far-label



说明：

在一新的代码段中的指定偏移量处继续执行指令。由16位段选择符和32位偏移量组成的48位指针(详见第2章)，指向一个不同段内的过程(段间调用)。

80386在取出CALL指令之后及执行CALL之前，已将EIP(指令指针寄存器)的值加上了CALL指令的长度，使它指向了紧接CALL的下一条指令。CALL指令先将此EIP值压入栈内，再进行控制转移。当遇着一RET指令时，栈内信息可使控制直接回到CALL的下条指令。

注意，当16位的CS进栈时，压入栈内的是一个32位字。这扩充的高16位是无定义的，为得是使栈保持与四字节边界对齐。这样改善了80386的性能。

详见第2章。

操作：

```
/* 远标号(far-label)被汇编成一个48位指针。其中的16位代码段选择符由下面的Segment程序返
   回;其中的32位指令偏移量由下面的Offset程序返回。
 */
if (op1 is far-label) {/* 于 */
  ESP  =ESP-4;
  [ESP]=CS;
  CS   =Segment(far_label);
  ESP  =ESP-4;
  [ESP]=EIP;
  EIP  =Offset(far_label);
} else if (op1 is mem48) {/* 于 */
  ESP  =ESP-4;
  [ESP]=CS;
  CS   =[&mem+4];
  ESP  =ESP-4;
  [ESP]=EIP;
  EIP  =[&mem];
}
/* 注于:
   由于远调用指令要加载CS寄存器,这将引起特权级的变化、门遍历和任务切换,所以其确切的效果
   可能和上面描述的不一样。这些更复杂的格式将在第5章中详细讨论。
 */
```

例子：

CALL FAR label; label=6320 : 0F03298A

执行前：EIP	：00000300	ESP	：	00000200	CS：5028
执行后：EIP	：0F03298A	ESP	：	000001F8	CS：6320
[ESP+4]：	××××5028	[ESP]：		00000300	

BET

执行前：EIP	：0F03298A	ESP	：	000001F8	CS：6320
[ESP+4]：	××××5028	[ESP]：		00000300	
执行后：EIP	：00000300	ESP	：	00000200	CS：5028

标志影响：

无，除非在保护模式下发生任务切换。

异常：

Memory ( )；

Stack ( )；

CodeSegmentLoad ( )；

CodeReference ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ADC 指令。

INT

调用中断过程

格式：

INT imm8

字节 0	字节 1
0 1 1 0 0 0 1 0	imm8（异常号）

说明：

INT 指令本质上是异常处理程序的软件调用。imm8 为异常号，由它决定哪个中断过程被调用。

扩展的标志寄存器、代码段寄存器及指令指针被压入栈内。控制被转移到由异常号规定的异常处理程序。在异常处理程序结束时，IRET 指令又把控制送回刚才执行被中断的地方。

注意当 16 位的 CS 进栈时，压入栈内的是一个 32 位字。这扩充的高 16 位是无定义的，为的是使栈保持与四字节边界对齐。这对 80386 的性能来说是很重要的。

这条指令可被用作对操作系统的一次“调用”。

操作：

```
ESP   =ESP-4;
[ESP] =EFLAGS;
ESP   =ESP-4;
[ESP] =CS;    /*↑*/
ESP   =ESP-4;
[ESP] =EIP;
CS    =IntExecTable (SEGMENT, imm8);
EIP   =IntExecTable (OFFSET, imm8);
IF    =0;
```

```
TF    =0;
NT    =0;
/*
  †由于INT指令要加载CS寄存器,这将引起特权级的变化,门遍历和任务切换,所以其确切结果可
  能和上面描述的不一样。这些更复杂的格式将在第5章和第6章中详细讨论。
*/
```

例子:

```
INT 5
    执行前: ESP        : FFFFF388      EIP        : 0000FA04
            EFLAGS     : 00004302      CS         :      F244
            IntExecTable(SEGMENT, 5)               :      0400
            IntExecTable(OFFSET, 5)                : 004AFDDC
    执行后: ESP        : FFFFF37C      EIP        : 004AFDDC
            EFLAGS     : 00000002      CS         :      0400
            [ESP+8]    : 00004302      [ESP+4]    : ××××F244
            [ESP]      : 0000FA04
```

标志影响:

无,除非在保护模式下发生任务切换。

异常:

Stack ( );

CodeSegmentLoad ( );

CodeReference ( );

实地址模式异常:同 ADC 指令。

保护模式异常:

对于在内存段 CS、DS、ES、FS 或 GS 中的操作数,如果源操作数是在一不可读的代码段中,或操作数的任何部分在超过了段界的有效地址,则产生错误码为 0000h 的通用保护异常(异常 13)。对于在堆栈段内的操作数,如果操作数的任何部分是在超出了段界的有效地址,则产生错误码为 0000h 的堆栈异常。如果内存操作数是在标为"不存在"的 SS 段内,则产生错误码为段选择符的堆栈异常。如果内存操作数是在标为"不存在"的 DS、ES、FS 或 GS 段内,则产生错误码为段选择符的"不存在"异常。页故障引起错误码为故障码的,页故障线性地址在寄存器 CR2 中的页故障异常(异常 14)。对于任务状态段的不相容性,如果执行任务切换,则会产生错误码为选择符的无效任务状态段异常(异常 10)。

虚拟 8086 模式异常:同 ADC 指令。

注意:

INT3 指令是一特殊的单字节指令格式(断点指令),常用于调试。详见附录 A。

INTO

调用溢出中断过程

格式:

INTO 1 1 0 0 1 1 1 0

说明：

在溢出标志为1时，INTO调用中断过程4(或异常处理4)。所以INTO指令实际上是中断过程4的条件调用(OF==1)。

扩展的标志寄存器、代码段寄存器以及指令指针被压入栈。控制被转移到相应中断级别的中断服务程序。在中断服务程序结束时，IRET指令又把控制送回刚才执行被中断的地方。

注意：当16位的CS进栈时，压入栈内的是一个32位字。这扩充的高16位是无定义的，为的是使栈保持与四字节边界对齐。这对80386的性能来说是重要的。

操作：

```
if (OF==1) {  /*  †  */
    ESP      =ESP-4;
    [ESP]    =EFLAGS;
    ESP      =ESP-4;
    [ESP]    =CS;
    ESP      =ESP-4;
    [ESP]    =EIP;
    CS       =IntExecTable (SEGMENT, 4);
    EIP      =IntExecTable (OFFSET, 4);
    IF       =0;
    TF       =0;
    NT       =0;
}
/*
  †由于INTO指令要加载CS寄存器，这将引起特权级的变化，门遍历和任务切换，所以确切的结果可
  能与上面描述的不一样。这些更复杂的格式将在第5章和第6章中详细讨论。
*/
```

例子：

```
INTO
    执行前: ESP        : FFFFF388  EIP        : OF822DDB        OSZAPC
            EFLAGS     : 00006FA7  CS         :      0008
            IntExecTable(SEGMENT,4)            :      4A00
            IntExecTable(OFFSET,4)             : 004AFDDC
    执行后: ESP        : FFFFF37C  EIP        : 004AFDDC        1UUUUU
            EFLAGS     : 00002CA7  CS         :      4A00
            [ESP+8]    : 00006FA7  [ESP+4]    : ××××0008
            [ESP]      : OF822DDB
```

时钟：(主频为20μHz)

若中断未发生：3个时钟周期；0.15μs。

若中断4发生：35个时钟周期；1.75μs。

异常：

Stack ( );

CodeSegmentLoad ( );

CodeReference( );

实地址模式异常：

若OF=1，中断4。

保护模式异常：

若 OF=1，中断 4。页故障引起错误码为故障码的，页故障线性地址在寄存器 CR2 中的页故障异常(异常 14)。

虚拟地址异常：

同保护模式。

IRET

中断返回

格式：

IRET

1	1	0	0	1	1	1	1

说明：

IRET 与中断调用的过程相反：它将 EIP，CS 和 EFLAGS 寄存器内容从栈中弹出，并将控制返回发生中断的地方。IRET 用在中断服务程序的结束处。

注意：当 CS 出栈时，一个 32 位字被弹出，其中低 16 位恢复 CS 的内容，高 16 位便遗弃了，其目的是为了保持栈和四字节边界的对齐。这对 80386 的性能是很重要的。

操作：

```
EIP     =[ESP];
ESP     =ESP+4;
CS      =[ESP]; /* † */
ESP     =ESP+4;
EFLAGS=[ESP];
ESP     =ESP+4;
/*
† 由于 IRET 指令要加载 CS 寄存器，这会引起特权级的变化，门遍历和任务切换，所以其精确的结果可
能与上面描述的不一样。这些更复杂的格式会在第 5 章和第 6 章中详细讨论。
*/
```

例子：

```
IRET
    执行前：ESP      : FFFFF37C  EIP      : 004B0245
            EFLAGS   : 00000002  CP       :      0400
            [ESP+8]  : 00004302  [ESP+4]  : ×××F244
            [ESP]    : 0000FA04
    执行后：ESP      : FFFFF388  EIP      : 0000FA04
            EFLAGS   : 00004302  CS       :      F244
```

标志影响：

无，除非在保护模式下发生任务切换。

异常：

Stack ( );

CodeSegmentLoad ( );

CodeReference ( );

实地址模式异常：同 ADC 指令。

保护模式异常：同 INT 指令。

虚拟模式异常：同 INT 指令。

JMP

跳　　转

格式：

JMP op_1

mem48

far-label

字节 0	字节 1	后续字节
1 1 1 1 1 1 1 1	mod:1 0 1 : r/m	
1 1 1 0 1 0 1 0	绝对偏移量,段选择符	

说明：

跳转到另一段的指定偏移量处继续执行指令。给定的 48 位远指针(由 16 位选择符和 32 位偏移量组成,详见第 2 章),指向不同段内的一个标号(段间跳转)。

这 48 位远指针可以直接通过一个远标号来规定,也可以间接地通过内存(mem48)来规定。

操作：

```
  /* 远标号被汇编成一48 位指针。其中 16 位代码段选择符由 Segment 程序返回,32 位偏移量由 Off-
  set 程序返回。
   */
if (op1 is far-label) {
  CS     =Segment(far-label);
  EIP    =Offset(far-label);
} else if (op1 is mem48) { /*† */
  CS     =[&mem+4];
  EIP    =[&mem];
}
  /* 由于远跳转指令加载 CS 寄存器,将会引起特权级的变化,门遍历和任务切换,所以精确的效果可
  能与上面描述的不一样。这些更复杂的格式将在第 5 章中详述。
   */
```

例子：

JMP FAR label1;　　label1=AC40 : 000000FB

执行前：EIP : 00000300　CS : 2DF0

执行后：EIP : 000000FB　CS : AC40

标志影响：

无,除非在保护模式下发生任务切换。

异常：

Memory ( );

CodeSegmentLoad ( );

CodeReference ( );

实地址模式异常：同 INT 指令。

保护模式异常：同 INT 指令。

虚拟模式异常：同 INT 指令。

Lsr

加载指针

格式：

				字节 0	字节 1	字节 2
Lsr	reg，	mem48				
LDS	reg，	mem48	；将指针装入到 DS	1 1 0 0 0 1 0 1	mod：reg：r/m	
LES	reg，	mem48	；将指针装入到 ES	1 1 0 0 0 1 0 0	mod：reg：r/m	
LFS	reg，	mem48	；将指针装入到 FS	0 0 0 0 1 1 1 1	1 0 1 1 0 1 0 0	mod：reg：r/m
LGS	reg，	mem48	；将指针装入到 GS	0 0 0 0 1 1 1 1	1 0 1 1 0 1 0 1	mod：reg：r/m
LSS	reg；	mem48	；将指针装入到 SS	0 0 0 0 1 1 1 1	1 0 1 1 0 0 1 0	mod：reg：r/m

说明：

Lsr 指令将一 48 位指针分成两个部分，装入段寄存器/通用寄存器对。这段寄存器就是指令助记符中规定的，DS、ES、FS、GS、SS 中的一个。通用寄存器由操作数 reg 指定。位于 mem48 的双字，作为偏移量，被装入通用寄存器。位于 mem48＋4 的字，作为段描述符被装入段寄存器。

在第 2 章中讲过，将选择符加载段寄存器可以使一个段变成可寻址的。关于段寄存器的加载，在第 5 章中还要详细讨论。

CS 不能通过 Lrs 指令加载（不能使用 LCS 指令）。CS 只能通过远控制流转移指令（CALL far label，IRET，JMPfar label，INT）加载。

操作：

```
reg=(dword) [&mem48];
srcg=(word) [&mem48+4];
```

例子：

```
LGS EAX, mem48
    执行前：EAX    :  ××××××××      GS：××××
            mem48  :  8034：OF4BC319
    执行后：EAX    :  OF4BC319      GS：8034
            mem48  :  8034：OF4BC319
```

时钟：（设主频为 20MHz）

实模式或虚拟 8086 模式下：7 个时钟周期；0.35μs。

保护模式下：22 个时钟周期；1.1μs。

异常：

Memory ( )；

SegmentLoad ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ADC 指令。

注意：

LSS、LFS 和 LGS 是 80386 新增的。用

LSS ESP，mem48

指令指向新的堆栈。由于 SS 寄存器和 ESP 寄存器用这同一条指令赋值，可以避免用分离的指令来分别修改它们所带来的任何潜在的问题。

MOV

段寄存器的传送

格式：

MOV	op_1	,	op_2
	sreg	,	reg16
	sreg	,	mem16
	reg16	,	sreg
	mem16	,	sreg

字节 0　　字节 1

1 0 0 0 1 1 1 0	mod：sreg：r/m

1 0 0 0 1 1 0 0	mod：sreg：r/m

说明：

op_2 的内容被复制到 op_1。

若 op_1 是一段寄存器，通过选择符的装入，使得一个新段变成了可寻址的。为了存取某一段中的数据、将标识该段的选择符装入一个段寄存器是必不可省的步骤。

若 op_1 是 SS，在下一条指令完成之后才可能产生中断。

段的处理与段寄存器的加载我们在第 5 章中还要讨论。

操作：

$op_1 = op_2$；

例子：

```
MOV AX, GS
    执行前：AX：××××  GS：F433
    执行后：AX：F433  GS：F433
MOV AX,Segment1        ;Segment1 的选择符被装入 AX
MOV GS,AX              ;该选择符再装入 GS
ADD GS：sum,DX         ;在 GS 段(Segment1)内'sum'单元中的数据成为可寻址的
                       ;了，并被用于本指令的相加操作中。
```

时钟：（假设主频为 20MHz）

MOV Sreg,reg16：　18 个时钟周期；　0.9μs

MOV Sreg,mem16：　19 个时钟周期；　0.95μs

MOV reg16,sreg：　2 个时钟周期；　0.1μs

MOV mem16,sreg：　2 个时钟周期；　0.1μs

异常：

SegmentLoad ( )；

Memory ( );

实地址模式异常：无。

保护模式异常：若 CPL 不为 0，则产生错误码为 0000h 的通用保护异常(异常 13)。

虚拟 8086 模式异常：

因在此模式下 CPL=3，所以要产生异常 13。

注意：

op_1 不能是 CS。

POP

从栈弹入段寄存器

格式：

POP op_1

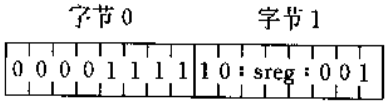
DS

ES

SS

FS

GS



说明：

POP 指令将字(选择符)从栈顶(SS：[ESP])弹出至指定的段寄存器(DS、ES、SS、FS 或 GS)，使得原来通过处于栈顶的选择符访问的段成为可寻址的。在完成将数据从栈顶传向 op_1 之后，栈指针 ESP 自动加 4。

要注意的是，段寄存器负载可能要求一个完整的段描述符负载。关于段寄存器负载的语义，将在第 5 章中讨论。

对 POP SS 指令，只有在下一条指令完成之后才会产生中断。

在 POP 指令中不能规定 CS 操作数。

操作：

op_1 =SS：[ESP];
ESP=ESP+4;

例子：

POP ES

执行前：ES：××××	SS：[ESP]	：F340	ESP：F4320530
执行后：ES：F340	SS：[ESP−4]	：F340	ESP：F4320534

时钟：

主频为 20MHz 时为 1.5μs，21 个时钟周期。

异常：

SegmentLoad ( );

Stack ( );

PUSH

压　栈

格式：

PUSH op_1

DS

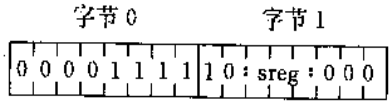
ES

CS

SS

FS

GS



说明：

op_1 被放入新的栈顶。栈指针 ESP 自动减 4。

操作：

SS : [ESP−2]=op_1;
ESP=ESP−4;

例子：

PUSH SS

执行前：SS : 8042　ESP : 0424　[SS : 420] : ××××

执行后：SS : 8042　ESP : 0420　[SS : 420] : 8042

时钟：

2 个时钟周期；20MHz 时为 0.1μs。

异常：

Stack ( );

实地址模式异常：同 3.4 节 PUSH 指令。

保护模式异常：同 3.4 节 PUSH 指令。

虚拟 8086 模式异常：同 3.4 节 PUSH 指令。

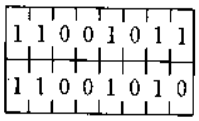
RET

返　回

格式：

RET

RET imm16



16 位立即数位移量

说明：

RET 使控制返回到紧跟相应的 CALL 指令的下一条指令。RET 指令从栈顶找出返回指

针(由CS和EIP组成的48位指针,请看第2章),这是个远指针。并将此远指针弹出,存入EIP寄存器和CS寄存器。也可以这样来理解RET指令:它是通过位于SS:[ESP]的内存指针的一次间接的远跳转,同时又修改了相应的堆栈。

RET指令可以和imm16选项连用。imm16的值被加到ESP上,于是从栈顶删掉若干字节的参数。

当16位的CS出栈时,弹出栈的是一个32位字,而其中高16位数据被抛弃。这样做的目的是为了保持栈与四字节边界的对齐。这对于80386的性能是很重要的。

操作:

```
EIP =[ESP];
ESP=ESP+4;
CS =[ESP]: /* † */
ESP=ESP+4;
if (NUMOPS==1) {
    ESP=ESP+imm16;
}
/*
  † 由于RET指令要加载CS寄存器,将会引起特权级的变化、门遍历、或任务切换,因此精确的效果可
  能与上述不一样。这些更复杂的格式将在第5章中详细讨论。
  */
```

例子:

FAR RET

执行前:	EIP	:0F03429A	ESP	:000001F8	CS:6320
	[ESP+4]	:××××5028	[ESP]	:00000300	
执行后:	EIP	:00000300	ESP	:00000200	CS:5028

时钟:

18+m个时钟周期

上述m为取指和开始执行目标指令的时间。

标志影响:

无,除非在保护模式下发生任务切换。

异常:

Stack ( );

CodeSegmentLoad ( );

CodeReference ( );

实地址模式异常:同ADC指令。

保护模式异常:同ADC指令。

虚拟8086模式异常:同ADC指令。

3.6 操作系统类指令

80386的硬件具有支持多种高级的操作系统特性的优势。它向操作系统的开发者提供了实现多任务、多用户以及虚拟存储的操作系统所必须的各种指令。

应用程序的开发者不必阅读本节。因为在应用程序中不允许使用本节的指令。

为了更好地理解本节的指令,你最好先看一下第5章。

除了标注 Level0( )异常的指令外，所有的指令都能在任何特权级别上使用。

3.6.1 函数定义

在本章开头，我们定义过几种函数。使用这些函数，大大地简化了指令的说明和例子。在这里我们为同样的目的再定义一个函数，这个函数在本节内使用。

```
/* 函数 FetchDescriptor( )的详细定义及其属性结构请看第 5 章。 */
Descr (num, op) {
  FetchDescriptor(op,&dword1,&dword2,&attributies)
  if(num==1)return(dword1);
  else          return(dword2);
}
```

3.6.2 异常程序

下面还有三个专门用于操作系统类指令的异常程序：

```
(1) SegmentLoad ( ) {
    /*  在装入一描述符时一切可能产生的段异常。它们被概括在第 5 章中定义的 DescriptorLoad( )
        程序之中，详见第 5 章。
    */
    }
(2) Level0( ) {
    /* 关于 CPL 详见第 5 章。SegmentException( )程序详见第 6 章。 */
    if (CPL! =0) {
      SegmentException ($GP,0);
    }
(3) 8086Mode ( ) {
    /* 若 80386 工作在实模式或虚拟 8086 模式，8086Mode( )报告一个 InvalidOpcode( )异常，详见第
        7 章。若 CR0 中的 PE 位为 0，处理器工作在实模式。若 CR0 中的 PE 为 1，且 EFLAGS 寄存器中
        的 VM 位为 1，处理器工作在虚拟 8086 模式。 */
    if ((CR0 & 1b)==0)   /* 实模式 */
      ||((CR0 & 1b)==1) &&
        (EFLAGS.VM==1))/* 虚拟 8086 模式 */)
      InvalidOpcode ( );
    }
```

3.6.3 操作系统类指令集

ARPL

调整申请特权级

格式：

ARPL	op_1	,	op_2
	reg16	,	reg16
	mem16	,	reg16

0	0	1	1	1	1	1	1

说明：

ARPL　用 op_2 的 RPL(申请特权级)去检查 op_1 的 RPL。op_1 和 op_2 的 RPL 分别是它们的

2个最低有限位规定的。若 op_1 的 RPL 小于 op_2 的 RPL，则零标志位置1，并将 op_2 的 RPL 值赋予 op_1 的 RPL(使它们的最低2位相等)。op_1 和 op_2 都可以是空的选择符。

ARPL 不用于应用程序，通常它只用于操作系统代码。常常，op_2 为调用者的代码段选择符，而 op_1 为调用者希望子程序去访问的段的选择符。这样的比较可以确保调用者不能够申请到超出许可范围的特权。

操作：

```
if ((op1 & 03h)<(op2 & 03h)) {
  ZF=1;
  op1=(op1 & FFFCh) | (op2 & 03h);
} else {
  ZF=0;
}
```

例子：

```
ARPL  mem16,AX                                   O S Z A P C
    执行前：mem16 : 6DD3   AX : 456A             × × × × × ×
    执行后：mem16 : 6DD3   AX : 456A             U U 0 U U U
ARPL  CX  ,AX
    执行前：CX    : FF44   AX : 62F1             × × × × × ×
    执行后：CX    : FF45   AX : 62F1             U U 1 U U U
```

时钟：(假设主频为20MHz)

ARPL reg16，reg16：20个时钟周期；1μs

ARPL mem16，reg16：21个时钟周期；1.05μs

标志影响：

影响ZF。其它标志不变。

异常：

Memory ( )；

8086Mode ( )；

实地址模式异常：

中断6。ARPL 是个保护模式的指令，在实地址模式中不认该指令。

保护模式异常：同 ADC 指令。

虚拟8086模式异常：

同实地址模式异常，但在保护模式下特权级为0时才处理。页故障引起错误码为故障码的，页故障线性地址在寄存器 CR2 内的页故障异常。

CLTS

清任务切换标志

格式：

CLTS

字节0	字节1
0 0 0 0 1 1 1 1	0 0 0 0 0 1 1 0

说明：

将机器状态字(MSW 或 CR0)中的任务切换标志清 0。每当任务切换时,该标志自动置 1(详阅第 5 章)。在系统中存在 80387 的情况下,TS(任务切换标志)用于将上下文切换时的系统开销减少到最低的程度。如果在任务切换之后(TS==1),新的任务试图使用 80387,将会产生一次失败。相应的失败处理程序首先将 80387 的上下文保存起来,然后再执行一次 CLTS 指令。于是新的任务可以使用 80387,并且不会破坏上一次任务的任何数据。只有使用 80387 的任务才会使系统为保存 80387 的先前状态而开销。

CLTS 只能在操作系统的代码中使用,并且只能在 0 级特权下执行。

操作:

TS=0;

例子:

CLTS

执行前: TS : 1

执行后: TS : 0

时钟:

5 个时钟周期;在主频为 20MHz 时为 0.25μs。

标志影响:

影响 CR0 寄存器中的 TS 位。其它标志都不变。

异常:

Level0 ( );

实地址模式异常: 无。

保护模式异常: 若 CPL 不为 0,则产生错误码为 0000h 的通用保护异常(异常 13)。

虚拟 8086 模式异常: 无。

HLT

暂　停

格式:

HLT　　1 1 1 1 0 1 0 0

说明

HLT 使 80386 停止执行。暂停(Halt)之后的系统,只有通过接受一个已经启用的中断(NMI 是一直启用着的),或者让系统复位,才能重新启动。中断和它们的启用,请阅第 5 章。

例子:

HLT　;　执行停止

时钟:

5 个时钟周期;0.25μs(主频为 20MHz)

标志影响:无。

异常:

Level0 ( );

实地址模式异常: 无。

保护模式异常：

若当前特权级不为0，则产生错误码为0000h的通用保护异常(异常13)。

虚拟8086模式异常：

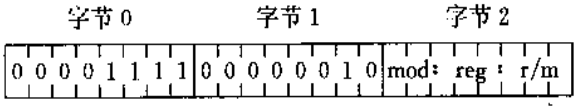
错误码为0000h的通用保护异常(异常13)。HLT只能在特权级为0时才能执行，而在虚拟8086模式下，80386工作的特权级为3。

LAR

加载存取权

格式：

LAR	op_1	,	op_2
	reg16	,	reg16
	reg16	,	mem16
	reg32	,	reg32
	reg32	,	mem32



说明：

op_2中的选择符指定了某段，将该段描述符中的属性字段(存取权)装入op_1。

该段的属性字段就是其描述符的高4字节和00FXFF00h相与的结果，其中X表示被LAR装入的值的第16位到第19位是无定义的。

如果选择符op_2指定的描述符，

- 在描述符表的范围内，
- 具有合法(或有效)的类型字段，
- CPL和op_2的RPL都不大于DPL，

则ZF置位，段的属性字段装入op_1。否则ZF清0，op_1不变。

内存段描述符(DType==1)的类型字段(Type)不管取什么值，对LAR指令均为合法或有效。于是只需比较CPL、RPL和DPL的值就行了。而系统段和门(DType==0)的类型对LAR指令是否有效，请看下表：

描述符类型	有效否
0 无定义的	无效
1 有效的286任务状态段(Available286TSS)	有效
2 局部描述符表(LDT)	有效
3 忙的286任务状态段(Busy286TSS)	有效
4 286调用门(286CallGate)	有效
5 任务门(TaskGate)	有效
6 286中断门(286InterruptGate)	无效
7 286自陷门(286TrapGate)	无效
8 无定义	无效
9 有效的386任务状态段(Available386TSS)	有效
A 无定义的	无效

B 忙的 386 任务状态段(Busy386TSS)	有效
C 386 调用门(386CallGate)	有效
D 无定义的	无效
E 386 中断门(386InterruptGate)	有效
F 386 自陷门(386TrapGate)	有效

操作：

```
if ((op2 & 0FFFCh) == 0)/* 空选择符 */
  goto Clear ZF;
/* 程序 FetchDescriptor 是在第 5 章中定义的。 */
/* 若描述符超出了表的范围,则返回 0。 */
if (FetchDescriptor(op2,&dword1,&dword2,&Attributes) == -0)
  goto ClearZF :
/* 描述符在表的范围内。 */
if(Attributes.DType==-1)/* 内存段 */
  switch (Attributes.Type) {
    case 0 : case 1 : /* 只读 */
    case 2 : case 3 : /* 读/写 */
    case 4 : case 5 : /* 只读,向下扩展 */
    case 6 : case 7 : /* 读/写,向下扩展 */
    case 8 : case 9 : /* 只执行 */
    case 10 : case 11 : /* 执行/读 */
      /* 只对不一致的段进行特权级检查。 */
      if ((Attsibutes.DPL<CPL) ||
          (Attsibutes.DPL<op2.RPL))
        goto CleasZF;
      break :
      /* 对于一致的段,不管 DPL。 */
    case 12 : case 13 : /* 只执行,一致 */
    case 14 : case 15 : /* 执行/读,一致 */
      break :
  }/* 开关结束  */
else /* DType==0,系统段或门  */
  switch (Attributes.Type) {
    case 1 : /* 有效的 286 任务状态段      */
    case 2 : /* 局部描述符表               */
    case 3 : /* 忙的 286 任务状态段        */
    case 4 : /* 286 调用门                 */
    case 5 : /* 任务门                     */
    case 9 : /* 有效的 386 任务状态段      */
    case 11 : /* 忙的 386 任务状态段       */
    case 12 : /* 386 任务门                */
      /* 类型有效,检查 DPL                */
      if (Attributes. DPL<CPL || Attributes.DPL<op2.RPL)
        goto ClearZF;
      break;
    Default: /* 其余类型都是无效的 */
              goto ClearZF;
    } /* 开关结束 */
/* 只有所有的检查都通过,才执行装入操作。 */
/* ZF 置 1,将属性装入 op1。 */
  ZF=1;
  op1=Attributes & 0F×FF00h;
```

```
  return;
/*若有一项检查没通过,便跳到此地。*/
/*ZF 清 0,op1 不变。*/
ClearZF:
  ZF=0;
  return;
```

例子:

OSZAPC

LAR EAX,mem32;假定描述符是可见的

执行前: Descr(0,mem32): 1F0FACFE　　× × × × × ×

EAX: ××××××××

执行后: Descr(0,mem32): 1FOFACFE　　× × 1 × × ×

EAX: 000XAC00

LAR EAX,mem32;假定描述符是不可见的。

执行前: Descr(0,mem32): 1FOFACFE　　× × × × × ×

EAX: ××××××××

执行后: Descr(0,mem32): 1FOFACFE　　× × 0 × × ×

EAX: UUUUUUUU

时钟:(假设主频为 20MHz)

从 reg: 15 个时钟周期;0.75μs

从 mem: 16 个时钟周期;0.8μs

标志影响:

影响 ZF。其它标志不变。

异常:

Memory ( );

8086Mode ( );

实地址模式异常:

中断 6。LAR 是条保护模式指令,在实地址模式中是不合法的。

保护模式异常:同 ADC 指令。

虚拟 8086 模式异常:

同实地址模式异常,但在保护模式的 0 特权级下处理。

页故障引起错误码为 0000h 的,页故障地址在 CR2 寄存器中的页故障异常(异常 14)。

LGDT

加载全局描述符表

格式:

LGDT op_1

mem48

字节 0	字节 1	字节 2
0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 1	mod:0 1 0: r/m

说明:

LGDT 将指令中给定的 48 位伪描述符(mem48)装入全局描述符表寄存器。伪描述符是

由两部分组成：在低字(低 16 位)中装的是界限，而在高双字(高 32 位)中装的是基址。

LGDT 只用在操作系统代码中；它不会出现在应用程序之中。

操作：

```
GDTR.Limit =[mem48];          /* 16 位 GDT 界限 */
GDTR.Base  =[&mem48+2];       /* 32 位 GDT 基址 */
```

例子：

```
LGDT mem48
    执行前：[mem 48]        : 4132     [&mem48+2] : AC405B10
            GDTR.Limit      : ××××     GDTR.Base  : ××××××××
    执行后：[mem 48]        : 4132     [&mem48+2] : AC405B10
            GDTR.Limit      : 4132     GDTR.Base  : AC405B10
```

时钟：

11 个时钟周期；0.55μs(主频为 20MHz)。

异常：

Memory ( )；

Level0 ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：

若 CPL 不为 0，则引起错误码为 0000h 的通用保护异常(异常 13)。其余同 ADC 指令。

虚拟 8086 模式异常：

因为 CPL 为 3(不等于 0)，所以要产生错误码为 0000h 的通用保护异常(异常 13)。

LIDT

加载中断描述符表

格式：

LIDT op_1

mem48

字节 0	字节 1	字节 2
0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 1	mod:0 1 1: r/m

说明：

LIDT 将指令中给定的 48 位伪描述符(mem48)装入中断描述符表寄存器。伪描述符和中断描述符表寄存器的结构相同，都是由两部分组成：在低字(低 16 位)中装的是界限，在高双字(高 32 位)中装的是基址。指令中的 mem48 给出了伪描述符的地址。

LIDT 只出现在操作系统代码中，它不会出现在应用程序中。

操作：

```
IDTR.Limit=[mem48];       /* 16 位 IDT 界限 */
IDTR.Base=[&mem48+2];     /* 32 位 IDT 界限 */
```

例子：

```
LIDT mem48
    执行前：[mem48]        : AAC3   [&mem48+2]   : 0BF4103D
            IDTR.Limit     : ××××   IDTR.Base    : ××××××××
```

执行后：[mem48]　　：AAC3　[&mem48+2]　：0BF4103D
　　IDTR.Limit　：AAC3　IDTR.Base　　：0BF4103D

时钟：

11 个时钟周期；0.55μs(主频为 20MHz)。

标志影响：无。

异常：

Memory ( )；

Level0 ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 LGDT 指令。

虚拟 8086 模式异常：同 LGDT 指令。

LLDT

加载局部描述符表

格式：

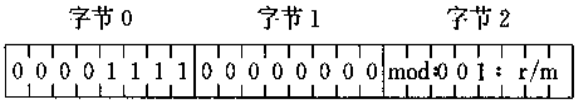
LLDT op_1

reg16

mem16

reg32

mem32



说明：

LLDT 将 op_1 中的选择符(访问一新的 LDT 段的)装入局部描述符表(LDT)寄存器。

给定的选择符必须指向全局描述符表(GDT)中的描述符类型为局部描述符表(LDT)的表项。在这种情况下，LDT 寄存器被加载。在第 5 章中我们再详细讨论这些检查。

op_1 可以是一个空的选择符。如果 op_1 是一个空选择符，将使得 LDT 被标为无效。如果 LDTR(局部描述符表寄存器)含有一空的选择符，欲装入一命名 LDT 段的选择符，则会引起段装入异常。

LLDT 仅出现在操作系统代码之中，它不会在应用程序中出现。

操作：

LDTR=op_1；

例子：

LLDT AX

　　执行前：AX：001C　LDTR：××××

　　执行后：AX：001C　LDTR：001C

时钟：

20 个时钟周期；1.0μs(主频为 20MHz)。

标志影响：无。

异常：

SegmentLoad ( );

Memory ( );

Level0 ( );

8086Mode ( );

实地址模式异常：同 LAR 指令。

保护模式异常：

若 CPL 不为 0，则产生错误码为 0000h 的通用保护异常(异常 13)。

如果选择符操作数不是去访问 GDT 和局部描述符表描述符的话，则产生错误码为出错选择符的通用保护异常。

其他异常同 ADC 指令。

虚拟 8086 模式异常：

非法操作码异常(异常 6)。LLDT 在虚拟 8086 模式下是条非法指令。

LMSW

加载机器状态字

格式：

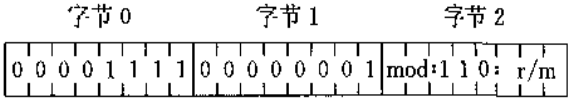
LMSW op_1

reg16

mem16

reg32

mem32



说明：

LMSW 将 op_1 装入 MSW(机器状态字，CR0 的低 16 位，详见第 2 章)。

将 MSW 中的 PE 位置为 1，便进入保护模式。这是 LMSW 指令的经常用途之一。如果的确是这样做了，在 LMSW 指令后面必须紧跟一条段内的跳转指令。

还要注意，为了和 80286 完全兼容，LMSW 指令不得改变 MSW 中的 ET 位。

注意：MSW 中的 PE 位和 LMSW 指令粘着。即 PE 位一旦被置为 1(已经进入保护模式)，便不能再用 LMSW 指令清 0 了。关于保护模式与实地址模式，请看第 5 章和第 7 章。

LMSW 仅出现在操作系统代码中，它不会出现在应用程序中。

操作：

MSW =(op_1 & FFEFh) | (MSW & 0011h);
/* CR0=(CR0 & FFFF0011h) | (0000FFEFh & op_1); */

例子：

LMSW AX

执行前：AX：0007　　MSW：0010

执行后：AX：0007　　MSW：0017

LMSW AX

执行前：AX：0010　　MSW：0007

执行后：AX：0010　　MSW：0001

时钟：(主频为 20MHz)

从 reg：10 个时钟周期；0.5μs。

从 mem：13 个时钟周期；0.65μs。

标志影响：无。

异常：

Memory ( )；

Level0 ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 LGDT 指令。

虚拟 8086 模式异常：同 LGDT 指令。

注意：

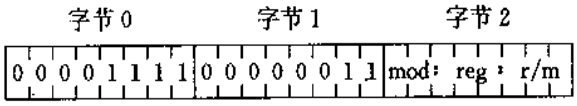
LMSW 指令和 80286 完全兼容。因此在 80386 的编程中，应当用 MOV CR0，而不用 LMSW 指令。

LSL

加载段界限

格式：

LSL	op_1	,	op_2
	reg16	,	reg16
	reg16		mem16
	reg32	,	reg32
	reg32	,	mem32



说明：

op_2 中的选择符指定了某段，将该段描述符中的段界限字段装入 op_1。

装入 op_1 的界限应当是字节界限。因此，如果指定的描述符中含有的是页面界限的话(G=1，这描述符中的 G 位将在第 5 章中讨论)，将此界限左移 12 位，空出的低 12 位全部充 1。

由 op_2 中的选择符指定的描述符必须满足：

- 在描述符表内，
- 其类型对 LSL 指令来说是合法的，
- CPL(当前特权级)和 op_2 的 RPL(申请特权级)不大于 DPL(描述符特权级)，

才能进行段界限的装入(op_1 修改并零标志位置 1)；否则 op_1 不变，零标志位清 0。

只要是内存段的描述符(其 DType 字段为 1)，在它的类型(Type)字段中可以有任何值，这些值对于 LSL 指令来说都是合法的。但对于系统段和门(DType=0)来说，在它的类型字段中哪些值是有效的，哪些值是无效的，请看下页表：

描述符类型	合法否
0 无定义	非法
1 有效的 286 任务状态段(Available286TSS)	合法
2 局部描述符表(LDT)	合法
3 忙的 286 任务状态段(Busy286TSS)	合法
4 286 调用门(286CallGate)	非法
5 任务门(TaskGate)	非法
6 286 中断门(286InterruptGate)	非法
7 286 自陷门(286TrapGate)	非法
8 无定义	非法
9 有效的 386 任务状态段(Available386TSS)	合法
A 无定义	非法
B 忙的 386 任务状态段(Busy386TSS)	合法
C 386 调用门(386CallGate)	非法
D 无定义	非法
E 386 中断门(386InterruptGate)	非法
F 386 自陷门(386TrapGate)	非法

操作：

```
if ((op2 & 0FFFCh)==0)/*空选择符 */
  goto ClearZF;
/*程序 FetchDescriptor 在第 5 章中定义的。*/
/*若描述符超出了表的范围,则返回 0。*/
if(FetchDescriptor(op2,&dword1,&dword2,&Attributes)==0)
  goto ClearZF;
/*描述符在表的范围内。*/
if (Attributes.DType==1)/*内存段*/
  switch (Attributes.Type){
    case 0:case 1:/*只读*/
    case 2:case 3:/*读/写*/
    case 4:case 5:/*只读,向下扩展*/
    case 6:case 7:/*读/写,向下扩展*/
    case 8:case 9:/*只执行*/
    case 10:case11:/*执行/读*/
    /*仅对不一致的段进行特权级检查。*/
    if ((Attributes.DPL<CPL)||
        (Attributes.DPL<op2.RPL))
      gotoClearZF;
      break;
    /*对于一致的段,忽略 DPL。*/
    case 12:case13:/*只执行,一致*/
    case 14:case15:/*执行/读,一致*/
      break;
}/*开关结束*/
else/* Dtype==0,系统段或门*/
  switch (Attributes.Type){
  case 1:/*Available 286TSS */
  case 2:/*LDT              */
  case 3:/*Busy      286TSS */
  case 9:/*Available 386TSS */
  case 11:/*Busy     386TSS */
      break;
  Default:/*其余类型皆非法*/
      goto Clear ZF;
```

```
  } /* 开关结束 */
/* 只有所有的检查都通过 */
/* ZF 置 1,段界限送入 op1。 */
  ZF=1;
  if (B(23,dword1)==0) {/* 字节界限 */
      op1=(dword2 & 000F0000h)|
           (dword1 & 0000FFFFh);
  }else{       /* 页面界限,须移位 */
      op1=(((dword2 & 000F0000h)|
           (dword1 & 0000FFFFh))<<12)|FFFh);
  }
  return;
/* 若有一项检查未通过,便跳到此地。 */
/* ZF 清 0,op1 不变。 */
ClearZF:
ZF=0;
return;
```

例子:

LSL EAX, mem32;假定描述符为可见的。

执行前:	Descr(0,mem32):	1 F0FACFE	O S Z A P C
	Descr(1,mem32):	0 0 8 F 8 1 0 0	× × × × × ×
	EAX:	××××××××	
执行后:	Descr(0,mem32):	1 F 0 F A C F E	U U 1 U U U
	Descr(1,mem32):	0 0 8 F 8 1 0 0	
	EAX:	FACFEFFF	

时钟:(假设主频为 20MHz)

字节界限来自 reg : 20 个时钟周期; 1.0μs

字节界限来自 mem : 21 个时钟周期; 1.05μs

页界限来自 reg : 25 个时钟周期; 1.25μs

页界限来自 mem : 26 个时钟周期; 1.3μs

标志影响:

影响 ZF。其它标志不变。

异常:

Memory ( );

8086Mode ( );

实地址模式异常:同 ARPL 指令。

保护模式异常:同 ADC 指令。

虚拟 8086 模式异常:同 ARPL 指令。

LTR

加载任务寄存器

格式:

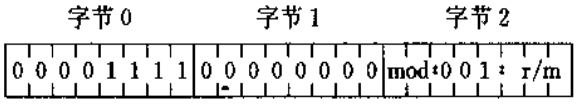
LTR op_1

reg16

mem16

reg32

mem32



说明：

LTR 将 op_1 中的选择符装入任务寄存器，使一个 TSS 段成为可寻址的。该加载的 TSS(任务状态段)被标为"忙"。

给定的选择子必须指向全局描述符表(GDT)中描述符类型为 TSS 的表项。若此条件满足，任务寄存器被加载。详见第 5 章。

LTR 只用于操作系统代码中。它不能在应用程序中出现。

操作：

TR=op_1;

例子：

LTR mem32

执行前：mem32：××××0014　TR：××××

执行后：mem32：UUUU0014　TR：0014

时钟：(主频为 20MHz)

op_1 为 reg：23 个时钟周期；1.15μs

op_1 为 mem：27 个时钟周期；1.35μs

标志影响：无。

异常：

SegmentLoad ( );

Memory ( );

8086Mode ( );

Level0 ( );

实地址模式异常：同 ARPL 指令。

保护模式异常：

若 CPL 不为 0，则产生错误码为 0000h 的通用保护异常(异常 13)。

若选择符操作数不是 GDT 和任务状态段描述符，则产生错误码为出错选择符的通用保护异常。其它异常同 ADC 指令。

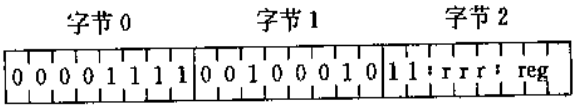
虚拟 8086 模式异常：同 LLDT 指令。

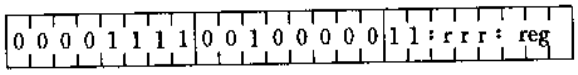
MOV

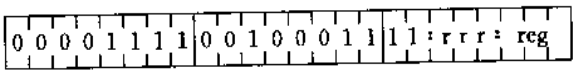
特殊寄存器的数据传送

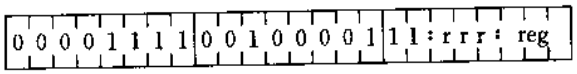
格式：

MOV	op_1	,	op_2
	reg32	,	CR0
	reg32	,	CR2
	reg32	,	CR3
	CR0	,	reg32
	CR2	,	reg32
	CR3	,	reg32
	reg32	,	DR0
	reg32	,	DR1
	reg32	,	DR2
	reg32	,	DR3
	reg32	,	DR6
	reg32	,	DR7
	DR0	,	reg32
	DR1	,	reg32
	DR2	,	reg32
	DR3	,	reg32
	DR6	,	reg32
	DR7	,	reg32









rrr	控制寄存器	调试寄存器
000	CR0	DR0
001		DR1
010	CR2	DR2
011	CR3	DR3
110		DR6
111		DR7

说明：

将 op_2 的内容复制到 op_1。

控制寄存器以及装入后的确切的效果将在第 5 章中描述。调试寄存器以及装入后的确切效果将在附录 A 中描述。

只有 32 位寄存器操作数才能用于 MOV 特殊寄存器指令。

特殊寄存器的传送只能在特权级别为 0 的操作中使用。

操作：

```
op1=op2; /* † */
/*
  †由于这些特殊寄存器的加载可能存在边界效应;所以这里的说明并不包括所有的情况。
 */
```

例子：

MOV CR0, EAX

执行前：EAX： 80000003　CR0： ××××××××

执行后：EAX： 80000003　CR0： 80000003

时钟：(假设主频为 20MHz)

reg to CR*：10 个时钟周期；0.5μs

reg to DR0-DR3：22 个时钟周期；1.1μs

reg to DR6,DR7：16 个时钟周期；0.8μs

CR* to reg：6 个时钟周期；0.3μs

DR0-DR3 to reg：22 个时钟周期；1.1μs

DR6,DR7 to reg：14 个时钟周期；0.7μs

标志影响：无。

异常：

Level0 ( )；

实地址模式异常：无

保护模式异常：

若 CPL 不为 0，则产生错误码为 0000h 的通用保护异常(异常 13)。

虚拟 8086 模式异常：

在此模式下 CPL=3，所以必产生错误码为 0000h 的通用保护异常。

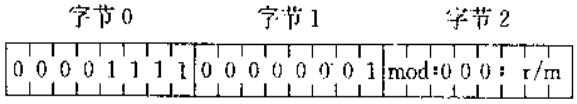
SGDT

存储全局描述符表

格式：

SGDT op_1

mem 48



说明：

全局描述符表寄存器的内容被存入位于 op_1 所指定的 6 个字节的伪描述符中。全局描述符表寄存器中的界限(16 位)存入 6 字节中的低字，而基址(32 位)则存入高双字。

SGDT 只用于操作系统代码中，它不出现在应用程序中。

操作：

```
mem 48          =GDTR.Limit;
[&mem48+2]      =GDTR.Base;
```

例子：

SGDT mem48

执行前：GDTR.Limit ：0032　　GDTR.Base　　：0BD542FF

mem48　　：××××　　[&mem48+2]：××××××××

执行后：GDTR.Limit ：0032　　GDTR.Base　　：0BD542FF

mem48　　：0032　　[&mem48+2]：0BD542FF

时钟：(主频为 20MHz)

9 个时钟周期；0.45μs

标志影响：无。

异常：

Memory ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ADC 指令。

SIDT

存储中断描述符表

格式：

SIDT op_1

mem48

字节 0	字节 1	字节 2
0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 1	mod:0 0 1 : r/m

说明：

中断描述符表寄存器内容被存入位于 op_1 所指定的 6 字节的伪描述符中。16 位界限被存入低字，32 位基址被存入高双字。

SIDT 只出现在操作系统代码中，它不会出现在应用程序中。

操作：

```
mem48          =IDTR.Limit;
[&mem48+2]     =IDTR.Base;
```

例子：

SIDT mem48

执行前：	IDTR.Limit	: 0032	IDTR.Base	: 0BD542FF
	mem48	: ××××	[&mem48+2]	: ××××××××
执行后：	IDTR.Limit	: 0032	IDTR.Base	: 0BD542FF
	mem48	: 0032	[&mem48+2]	: 0BD542FF

时钟：（主频为 20MHz）

9 个时钟周期；0.45μs

标志影响：无。

异常：

Memory ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ADC 指令。

SLDT

存储局部描述符表

格式：

SLDT op_1

reg16

mem16

reg32

mem32

字节 0	字节 1	字节 2
0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 0	mod:0 0 1: r/m

说明：

局部描述符表寄存器的内容存入 op_1。这个寄存器含有一指向当前 LDT 的选择符。

SLDT 只出现在操作系统代码中，它不会出现在应用程序中。

操作：

op_1=LDTR；

例子：

SLDT mem16

执行前：LDTR：5569　　[&mem16]：××××

执行后：LDTR：5569　　[&mem16]：5569

时钟：（主频为 20MHz）

2 个时钟周期；0.1μs。

标志影响：无。

异常：

Memory ( )；

8086Mode ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 LLDT 指令。

SMSW

存储机器状态字

格式：

SMSW op_1

reg16

mem16

reg32

mem32

字节 0	字节 1	字节 2
0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 1	mod:1 0 0: r/m

说明：

SMSW 将 MSW 寄存器的内容存入 op_1。MSW(CR0 的低 16 位)已在第 2 章中讨论过。

操作：

op_1=MSW；

例子：

SMSW AX

执行前：AX：××××　MSW：FF10

执行后：AX：FF10　MSW：FF10

时钟：主频为 20MHz

op_1 为 reg：2 个时钟周期；0.1μs。

op_1 为 mem：3 个时钟周期；0.15μs。

标志影响：无。

异常：

Memory ( )；

实地址模式异常：同 ADC 指令。

保护模式异常：同 ADC 指令。

虚拟模式异常：同 ADC 指令。

注意：

SMSW 与 80286 完全兼容。在 80386 编程时，应当用 MOV CR0 指令，而不用 SMSW。

STR

存储任务寄存器

格式：

STR op_1

reg16

mem16

字节 0	字节 1	字节 2
0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 0	mod:0 0 1 : r/m

说明：

将任务寄存器内容存入 op_1。

STR 指令仅在操作系统代码中出现，不会在应用程序中出现。

操作：

op_1=TR；

例子：

STR mem16

执行前：TR：FFD5　mem：××××

执行后：TR：FFD5　mem：FFD5

时钟：主频为 20MHz

2 个时钟周期；0.1μs。

标志影响：无。

异常：

Memory ( )；

8086Mode ( )；

实地址模式异常：同 ARPL 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 LLDT 指令。

VERR

读 检 验

格式：

VERR op_1

reg16

mem16

reg32

mem32

字节 0	字节 1	字节 2
0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 0	mod:1 0 0: r/m

说明：

VERR 指令判断由 op_1 中的选择符指定的段是否能读。

这选择符指向的描述符必须是一个合法的(在 LDT 或 GDT 之中)内存段描述符，并且在当前特权级上可读可见。如果 op_1 中选择符指定的段符合上述条件，则零标志 ZF 置位。否则 ZF 清 0。

指令执行的检验过程，连同留在零标志位中的结果，向我们报告了想要访问的描述符是否已经加载以及该段是否允许读操作。

操作：

```
if ((op1 & 0FFFCh)==0)/* 空的描述符 */
  goto Clear ZF;
/* 程序 FetchDescriptor ( )是在第 5 章中定义的 */
/* 若描述符超出了表的范围则返回 0。 */
if (FetchDescriptor(op1,& dword1,&dword2;&Attributes)==0)
  goto Clear ZF;
/* 描述符在表的范围内 */
if (Attributes.DType==1)/* 内存段 */
  switch (Attributes.Type) {
    case 0 : case1 : /* 只读 */
    case 2 : case3 : /* 读/写 */
    case 4 : case5 : /* 只读,向下扩展 */
    case 6 : case7 : /* 读/写,向下扩展 */
    case 10 : case11 : /* 执行/读 */
    /* 只对不一致的段进行特权级检查 */
      if ((Attributes. DPL<CPL) ||
        (Attributes.DPL<op1.RPL))
        goto ClearZF;
      break;
    /* 对于一致的段,不去管 DPL */
    case14 : case15 : /* 执行/读,一致 */
      break;
    Default : /* 其余的类型都是非法的 */
        goto ClearZF;
  } /* 开关结束 */
else /* DType==0,系统段或门 */
    goto ClearZF;/* 所有系统段都是非法的 */
```

```
/*所有的检查都通过了*/
/*零标志位该置1 */
  ZF=1;
  return;
/*未通过所有的检查*/
/*零标志位清0 */
ClearZF:
    ZF=0;
    return;
```

例子：

VERR mem16；假设段是可读的

	OSZAPC
执行前：	×××××××
执行后：	××1×××

时钟：（主频为 20MHz）

op_1 为 reg：10 个时钟周期；0.5μs

op_1 为 mem：11 个时钟周期；0.55μs

标志影响：

影响 ZF。其它标志不变。

异常：

Memory ( )；

8086Mode ( )；

实地址模式异常：同 ARPL 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ARPL 指令。

VERW

写 检 验

格式：

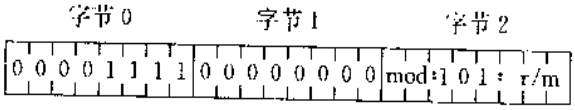
VERW op_1

reg16

mem16

reg32

mem32



说明：

VERW 指令判断由 op_1 中的选择符指定的段是否能写。

这选择符必须指向一个合法的（在 LDT 或 GDT 之中）内存段描述符，并在当前特权级上可写可见。若 op_1 中选择符指定的段满足上述条件，则零标志 ZF 置位，否则 ZF 清 0。

指令执行的检验过程，连同留在 ZF 中的结果，向我们报告了想要访问的描述符是否加载，以及该段是否允许写操作。

操作：

```
if ((op1 & 0FFFCh)==0) /* 空选择符 */
  goto ClearZF;
/* 程序 FetchDescreptor ( )是在第 5 章中定义的 */
/* 若描述符超出了表的界限,则返回 0。 */
if (FetchDescriptor(op1,&dword1,&dword2,&Attributes)==0)
  goto ClearZF;
/* 描述符在表中 */
if (Attributes.DType==1)/* 内存段 */
  switch (Attributes.Type) {
    case 2: case 3: /* 读/写 */
    case 6: case 7: /* 读/写,向下扩展 */
      /* 只对不一致的段进行特权级检查 */
      if ((Attributes.DPL<CPL) ||
        (Attributes.DPL<op1.RPL))
      goto ClearZF;
    break;
    Default; /* 其余类型都是非法的 */
      goto clearZF;
  } /* 开关结束 */
else /* DType==0,系统段或门  */
  goto ClearZF; /* 所有系统段都是非法的 */
/* 所有的检查全都通过 */
/* 零标志位置 1 */
  ZF=1;
  return;
/* 未能通过全部检查 */
/* 零标志位清 0 */
ClearZF:
  ZF=0;
  return;
```

例子：

VERW mem16；假定该段是可读的

	OSZAPC
执行前：	×××××××
执行后果：	××1×××

时钟：

同 VERR 指令。

标志影响：

同 VERR 指令。

异常：

Memory ( )；

8086Mode ( )；

实地址模式异常：同 ARPL 指令。

保护模式异常：同 ADC 指令。

虚拟 8086 模式异常：同 ARPL 指令。

3.7 浮点类指令

本节介绍 80386/80387 系统的浮点类指令，即 80387 支持的所有算术指令、数据传送指

令、比较指令、超越函数指令和控制指令等等。

对每条指令，都举一至两个例子。到了第 4 章的下半部分，我们再举些与本章指令有关的应用实例。这些例子都是由许多指令组成，能实现一定功能的程序段。通过这些实例，可对每一条浮点指令有深刻的理解。

有些浮点指令的语法和描写 80386 指令的语法有较大的不同。现在我们先来介绍本节(浮点类指令集)要用到的语法。

3.7.1 指令助记符

所有浮点指令的汇编助记符都冠一标准前缀，或附一标准后缀，以标明指令的类属：

F　所有浮点指令的第一个字母都是 F，比如 FADD、FLD 等等。这和 86 系列的其它成员不一样。

FI　所有操作整型数的指令都以 FI 打头。如，FIADD、FILD 等。

FB　所有和 BCD 数据类型有关的指令都以 FB 开头。如 FBLD、FBST 等等。

F××P　所有使得栈弹出一次的指令都以 P 结尾，如 FSTP、FADDP 等等。

F××PP　所有能使堆栈弹出两次的指令都以 PP 结尾，如 FCOMPP、FUCOMPP 等等。

FN××　FN××指令不检查数字异常，如 FNINIT、FNSAVE 等。而其它所有的指令在执行之前都要先检查未屏蔽的数字异常。

3.7.2 累加器栈

在第 2 章中我们说过，80387 使用一累加器栈，而且所有的操作都以栈顶作为操作数之一。栈顶用符号 ST 表示。而栈中其它的累加器，相对于栈顶，记作 ST(n)，表示它是 ST 之下的第 n 个栈元素。比如，ST(3)表示栈顶之下的第 3 个累加器。

许多浮点操作都含有栈操作(压入或弹出)。为了告诉人们关于栈顶的情况，我们在例子中使用 TOP 记号。如，TOP=3，表示当前的栈顶寄存器是第 3 个物理浮点寄存器。

3.7.3 例子

下面是一个浮点指令的实例。这是个比较和出栈指令。

ST 与 ST(3)进行数值比较。然后栈顶被弹出，TOP 由 4 变成 5。

FCOMP ST(3)

执行前：ST　：$2.4560 * 10^6$

ST(3)　：$9.4102 * 10^4$

TOP　：4 C3：×C2：× C0：×

执行后：ST(2)　：9.4102×10^4

TOP　：5 C3：0 C2：0 C0：0

注意：压栈使栈顶指针的值下降(在栈中存入一个新项)，而出栈使栈顶指针的值增加(从栈中削掉一项)。请看第 2 章，以及图 2.24。

3.7.4 函数定义

下面再介绍几个本节专用的函数：

(1) FPop ( ) {

```
      TOP=TOP+1;
      if(TOP>7) {
        TOP=0;
      }
    }
    /* 出栈,栈顶指针 TOP 加一。 */
(2) FPush ( ) {
      TOP=TOP-1;
      if (TOP<0) {
        TOP=7;
      }
    }
    /* 压栈,栈顶指针 TOP 减一。 */
(3) LdErrorPointer (memp) {
      FIP=[memp+12];
      FCS=[memp+16];
      FOO=[memp+20];
      FOS=[memp+24];
    }
(4) StError Pointer (mem.p) {
      [memp+12]=FIP;
      [memp+16]=FCS;
      [memp+20]=FOO;
      [memp+24]=FOS;
    }
    /* FIP(Floating-point Instraction Pointer,浮点指令指针), */
    /* FCS(Floating-point Code Segmet,浮点代码段) */
    /* FOO(Floating point Operand Offset,浮点操作数偏移量) */
    /* FOS(Floating-point Operand Segment,浮点操作数段) */
    /* 上述四个变量存放在起始地址为[memp+12]的内存单元中。 */
    /* FIP 和 FCS 是最新的浮点指令的 EIP 和 CS 的值的拷贝。 */
    /* FOO 和 FOS 是最新的浮点指令(如果它存在的话)的"mem"的偏移量和段 */
(5) abs(op); /* 返回 op 的绝对值 */
(6) arctan (op); /* 返回 op 的反正切值(返回值以弧度为单位) */
(7) cos(op); /* 返回 op 的余弦值(op 以弧度为单位) */
(8) log2(op); /* 返回 op 的以 2 为底的对数 */
(9) sin(op); /* 返回 op 的正弦值(op 以弧度为单位) */
(10) sqrt(op); /* 返回 op 的平方根 */
(11) tan(op); /* 返回 op 的正切值(op 以弧度为单位) */
```

3.7.5 异常程序

下面介绍一个本节(浮点类指令集)专用的异常程序:

```
Float (Src) {
  if (异常 Src 未被屏蔽) {
      Interrupt (16);
  }
/* Src 包括:
  IS: 由于栈溢出或下溢造成的无效操作
  I: 任何原因的无效操作
  D: 不可规格化数操作
  Z: 被 0 除
  O: 溢出
  U: 下溢
  P: 精度
```

```
    */
}
```

3.7.6 浮点类指令集

F2XM1

2^x-1

格式：

F2XM1

说明：

计算 $2^{ST}-1$，其中 ST 为栈顶的内容。计算完成后，结果放入栈顶。

注意：操作数 ST 必须在 −0.5 到 0.5 的范围内。若 ST 超出此范围，结果将无定义。

操作：

```
if (-0.5 <= ST <= 0.5) {
    ST=2^ST-1;
} else {
    UNDEFINED;
}
```

例子：

F2XM1

执行前：ST：$3.49921 * 10^{-2}$

执行后：ST：$2.45512 * 10^{-2}$

异常：

Float (IS, I, D, U, P);

注意：

80287 允许输入范围为 0.0 到 0.5；而 80387 允许输入范围为 −0.5 到 0.5。

FABS

绝 对 值

格式：

FABS

说明：

栈顶元素改变成它的绝对值。FABS 之后，栈顶元素总为正。

操作：

```
if (ST<0) {
   ST=-ST;
}
```

例子：

FABS

执行前：ST：$-7.324 * 10^{-402}$

执行后：ST：$7.324 * 10^{-402}$

FADD/FIADD/FADDP

加

格式：

FADD			;实型数加 ST(1),ST
FADD	op_1		;实型数加
	ST(n)		
	mem32		;短实型数
	mem64		;长实型数
FIADD	op_1		;整型数加
	mem16		;字整数
	mem32		;短整数
FADDP	op_1	, op_2	;实型数加并出栈
	ST	, ST(n)	
	ST(n)	, ST	
FADD	op_1	, op_2	;实型数加
	ST	, ST(n)	
	ST(n)	, ST	

说明：

浮点操作数相加。如果不规定操作数，则隐指 ST(1)+ST，结果放入栈顶。若只给出一个操作数 op_1，则隐指 ST+op_1，结果放入栈顶。若给出了两个操作数，便执行 op_1+op_2，结果放入 op_1(ST 或 ST(n))。

如果 op_1 为内存操作数(字整数或短整数，短实数或长实数)，在它参与任何操作之前，自动转换成内部的临时实型数格式。

无任何操作数的 FADD，或 FADDP，可令栈顶弹出。所以，无操作数的 FADD，与 FADDP ST(1),ST 同义。

操作：

```
if (NUMOPS==0) {
    ST(1)=ST(1)+ST;
    FPop( );  /*栈顶弹出,结果保存在 ST*/
} else if (NUMOPS==1) {
    ST=ST+(tempReal)op1;
} else {     /*NUMOPS==2 */
    op1=op1+op2;
    if (OPCODE==FADDP) {
        FPop( );    /*栈顶弹出*/
    }
```

}

例子：

FADD mem32

执行前：ST：$4.51200*10^5$ mem：$3.664*10^3$

执行后：ST：$4.54864*10^5$ mem：$3.664*10^3$

异常：

Memory ( )；

Float (IS,I,D,U,O,P)；

注意：

下表概括了各种对无穷数的加操作：

操作数		结果 (op_1)	
op_1	op_2	符号	量值
$+\infty$	$+\infty$	+	∞
$-\infty$	$-\infty$	−	∞
$-\infty$	$+\infty$	无效操作	
$+\infty$	$-\infty$	无效操作	
$\pm z$	$\pm\infty$	取无穷数的符号	∞
$\pm\infty$	$\pm z$	取无穷数的符号	∞

其中 $0\leqslant z<\infty$

FBLD

BCD 码装入

格式：

FBLD mem80

说明：

mem80 指向的 10 个字节的压缩的 BCD 码内存操作数被转换成临时实型数，并压入栈顶。这种转换非常精确。mem80 的 BCD 的每一位十进数字的范围为从 0h 到 9h。

操作：

FPush ( )；

ST=(tempReal)mem80；

例子：

FBLD mem80

执行前：mem80 ： −000000000000005698 TOP：4

执行后：mem80 ： −000000000000005698 TOP：3

ST ： $-5.698*10^3$

异常：

Float (18)；

Memory ( );

注意：

非法的 BCD 码的装入，将使无定义的临时实型数压入栈顶。

FBSTP

BCD 码存储并出栈

格式：

FBSTP mem80

说明：

栈顶被转换为 18 位的 BCD 码，存储到 mem80 指向的 10 个字节的内存单元中，然后该栈顶被弹出。该 ST 用 RC 规定的舍入方式舍入为一个整型数。所以，在 FBSTP 指令前毋须 FRNDINT 指令。

操作：

```
tmp=(BCD)ST;
FPop ( );
mem80=tmp;
```

例子：

```
FBLD mem80                                  ;Round Down
执行前：ST      : 8.99033 * 10^3      TOP : 7  RC : RD
        mem 80  : ××××××××××××××××××
执行后：ST      : 8.99033 * 10^3      TOP : 0  RC : RD
        mem80   : 000000000000008990
```

异常：

```
Float (IS,I);
Memory ( );
if (ST is QUIETNaN) {   /* 栈顶为一静的 NaN */
   Float (I);
}
```

注意：

对于 FBSTP，ST 可以是一个非规格化数，但在 80287 里是不可以的。

这条指令速度很慢，而且可能引起中断等待的问题。

对静的 NaN 的操作通常不会产生异常。但在这里，如果 ST 是一个静的 NaN(非数字)，将产生无效操作异常。

FCHS

改 变 符 号

格式：

FCHS

说明：

栈顶元素符号求反。若栈顶为+0.0,则被 FCHS 指令变为−0.0。

操作：

ST=−ST;

例子：

FCHS

执行前：ST： $6.221 * 10^{37}$

执行后：ST： $-6.221 * 10^{37}$

异常：

Float (IS);

FCLEX/FNCLEX

清 除 异 常

格式：

FCLEX

FNCLEX

说明：

FCLEX/FNCLEX 使 80387 清除状态字寄存器中的所有的异常和"忙"位(Busy Bit)。

FCLEX 检查未屏蔽的数字异常;而 FNCLEX 不检查。

操作：

SW=SW & 7F00h;

例子：

FCLEX

执行前：SW：F450

执行后：SW：7400

FCOM/FCOMP/FCOMPP

比　　较

格式：

FCOM	op_1	;比较
	mem32	
	mem64	
	ST(i)	
FCOMP	op_1	;比较并弹出

mem32

mem64

ST(i)

FCOMPP ;比较并弹出两个

说明：

给定的操作数与栈顶进行数值比较。按下表设置条件码。如操作码为FCOMPP，与栈顶进行比较的是ST(1)。

	C3	C2	C0
ST>op_1	0	0	0
ST<op_1	0	0	1
ST==op_1	1	0	0
无序	1	1	1

如操作码为FCOMP，在比较之后栈被弹出一次。如操作码为FCOMPP，在比较之后栈顶被弹出两次。

如两个操作数中无论哪一个为NaN的话，便为无序比较。

在比较中不管零的符号。

操作：

```
if (OPCODE==FCOMPP) {
  op1=ST(1);
}
if ((ST is NaN) || (op1 is NaN)) {
  C3=1; C2=1; C0=1;
} else if (ST>op1) {
  C3=0; C2=0; C0=0;
} else if (ST<op1) {
  C3=0; C2=0; C0=1;
} else if (ST==op1) {
  C3=1; C2=0; C0=0;
}
if (OPCODE==FCOMP) {
  FPop ();
} else if (OPCODE==FCOMPP) {
  FPop ();
  FPop ();
}
```

例子：

FCOMP ST(3)

执行前：ST : $2.4560 * 10^6$

ST(3) : $9.4102 * 10^4$

TOP : 4 C3 : × C2 : × C0 : ×

执行后：TOP : 5 C3 : 0 C2 : 0 C0 : 0

异常：

```
Float (IS, I, D);
if ((ST is QUIETNaN) || (op1 is QUIETNaN)) {
```

```
    Float (I);
}
```

注意：

一般情况下，对于静的 NaNs 的操作不产生操作异常。但在这里，若操作数之一为静的 NaN，FCOM 产生无效操作异常。

第 4 章的程序 7，给出了根据 80387 条件码标志跳转的例子。

FCOS

余　弦

格式：

FCOS

说明：

计算 ST 的余弦值。计算完成后，栈顶被设置为 ST 的余弦值，并再将 1.0 压入堆栈。于是

ST(1)/ST=COS(ST′)

其中 ST′为计算余弦值之前的栈顶。FCOS 指令完成后，ST 一定为 1.0，而 ST(1)为 ST′的余弦值。

FCOS 的操作数必须在 0 到 $\pi * 2^{62}$的范围内。如果源操作数满足上述条件，C2 置为 0。否则 C2 置为 1，ST 和栈顶指针(TOP)都保持不变。

操作：

```
if (ST<π*2^62) {
  ST=COS(ST);
  FPush ( );
  ST=1.0;
  C2=0;
} else {
  /*注意：ST 不变*/
  C2=1;
}
```

例子：

FCOS

执行前：ST	$: 4.51200 * 10^{5}$	TOP：4	C2：×
执行后：ST	$: 1.00000 * 10^{0}$	TOP：3	C2：0
ST(1)	$: 4.53137 * 10^{-1}$		

异常：

Float (IS,I,D,U,P);

注意：

该指令是 80387 新增的。

FDECSTP

栈指针减一

格式：

FDECSTP

说明：

状态字中的栈顶指针 TOP 被减一。标记(Tag word)或栈顶本身不变。若执行 FDECSTP 时 TOP 为 0，则 TOP 变成 7。如果向栈中压入一个新的元素，栈顶指针如同执行 FDECSTP 那样被减一。

操作：

```
TOP=TOP-1;
if (TOP<0) {
   TOP=7;
}
```

例子：

FDECSTP

执行前：TOP：4

执行后：TOP：3

注意：

该指令可以直接控制栈指针。当程序需要一个比 80387 提供的 8 个硬件累加器更大的"虚拟"累加器栈时，这条指令非常有用。这些"虚拟"栈寄存器是驻留在内存的，当检测到栈无效操作异常时，这个虚拟的堆栈将由软件来管理。支持虚拟累加器栈的指令有 FDECSTP，FINCSTP 和 FFREE。

FDIV/FIDIV/FDIVP

除

格式：

```
FDIV                    ;实数除，并弹 ST 出栈
FDIV  op1               ;实数除
      ST(n)
      mem32             ;短实数
      mem64             ;长实数
FIDIV op1               ;整数除
      mem16             ;字整数
      mem32             ;短整数
FDIVP op1     ,op2      ;实数除并弹出 ST
      ST      ,ST(n)
```

```
        ST(n) ,ST
FDIV    op1   ,op2     ;实数除
        ST    ST(n)
        ST(n) ST
```

说明：

如不规定操作数，隐指 ST(1)/ST，结果放入栈顶。如只规定一个操作数，隐指 ST/op_1，结果放入栈顶。如果规定了两个操作数，则执行 op_1/op_2，结果放入 op_1(ST 或 ST(n))。

如果 op_1 为内存操作数(字整数或短整数，短实数或长实数)，在它参与任何操作之前，自动转换为内部的临时实型数格式。

不带任何操作数的 FDIV，可令栈顶弹出。于是，不带操作数的 FDIV 指令，与 FDIVP ST(1),ST 同义。

操作：

```
if (NUMOPS==0) {
    ST(1)=ST(1)/ST;
    FPop( );     /*栈顶弹出,结果放入 ST*/
} else if (NUMOPS==1) {
    ST=ST/(TempReal) op1;
} else {            /*NUMOPS==2*/
    op1=op1/op2;
    if (OPCODE=FDIVP) {
        FPop ( );          /*栈顶弹出*/
    }
}
```

例子：

FDIV ST,ST(3)

执行前：ST　　：$2.240 * 10^{-2}$

　　ST(3)　：$3.664 * 10^{3}$

执行后：ST　　：$6.113 * 10^{-6}$

　　ST(3)　：$3.664 * 10^{3}$

异常：

Memory ( );

Float (IS,I,Z,U,P,D,O);

注意：

下表概括了各种对于无穷数的除操作：

操作数		结果(op_1)	
op_1	op_2	符号	量值
±∞	±∞	非法操作	
±∞	±z	操作数符号之异或	∞
±z	±∞	操作数符号之异或	0
±∞	±0	操作数符号之异或	∞
±z	±0	操作数符号之异或	∞

其中 $0 \leqslant z < +\infty$

请看第 4 章的程序 11。

FDIVR/FIDIVR/FDIVRP

反　除

格式：

```
FDIVR                   ;实数反除,ST/ST(1)
FDIVR  op1              ;实数反除
       ST(n)
       mem32            ;短实数
       mem64            ;长实数
FIDIVR op1              ;整数反除
       mem16            ;字整数
       mem32            ;短整数
FDIVRP op1     ,op2     ;实数反除并出栈
       ST      ,ST(n)
       ST(n)   ,ST
FDIVR  op1     ,op2     ;实数反除
       ST      ,ST(n)
       ST(n)   ,ST
```

说明：

如不规定操作数，隐指 ST/ST(1)，结果放入栈顶。如只规定一个操作数，隐指 op_1/ST，结果放入栈顶。如果规定了两个操作数，则执行 op_2/op_1，结果放入 op_1(ST 或 ST(n))。

FDIVR/FIDIVR/FDIVRP 和 FDIV/FIDIV/FDIVP 相比较，只是除数和被除数换了位而已，其它没有什么差别。例如，对于 FDIVR，被除数/除数为 op_2/op_1，而对于 FDIV 则为 op_1/op_2。

如果 op_1 为内存操作数(字整数或短整数，短实数或长实数)，在它参与任何操作之前，自动转换为内部的临时实型数格式。

不带任何操作数的 FDIVR 指令会使栈顶弹出。于是，不带操作数的 FDIVR 指令，与 FDIVRP ST(1),ST 同义。

操作：

```
if (NUMOPS==0) {
    ST(1)=ST/ST(1);
    FPop( );     /*栈顶弹出,结果放在ST中*/
} else if (NUMOPS==1) {
    ST=(tempReal) op1/ST;
} else {         /*NUMOPS==2 */
    op1=op2/op1;
    if (OPCODE=FDIVRP) {
      FPop( );   /*栈顶弹出*/
    }
```

}

例子：

FDIVR ST,ST(3)

执行前：ST　　：$2.240 * 10^{-2}$

ST(3)　：$3.664 * 10^{3}$

执行后：ST　　：$1.636 * 10^{5}$

ST(3)　：$3.664 * 10^{3}$

异常：

Memory ( )；

Float (IS,I,Z,U,P,D,O)；

注意：

下表概括了各种对于无穷数的反除操作：

操作数		结果(op_1)	
op_1	op_2	符号	量值
±∞	±∞	无效操作	
±∞	±z	操作数符号之异或	0
±z	±∞	操作数符号之异或	∞
±0	±∞	操作数符号之异或	∞
±0	±z	操作数符号之异或	∞

其中 $0 \leqslant z < +\infty$

FFREE

释放寄存器

格式：

FFREE ST(n)

说明：

与指定的寄存器相关的标记字位被置为11b。这表示该指定的栈单元变为空。但浮点栈和浮点栈指针都不变。

操作：

```
tmp=((TOP+n) % 8);  /*物理寄存器号*/
TW=TW|(11b<<(tmp * 2));
```

例子：

FFREE ST(3)

执行前：TOP：3　TW：0FC0

执行后：TOP：3　TW：3FC0

注意：

标记字描述了物理的栈寄存器，而FFREE提供了相对于栈顶的累加器栈访问。

FICOM/FICOMP

整数比较

格式：

FICOM op_1 ； 整数比较
mem16 ； 字整数
mem32 ； 短整数
FICOMP op_1 ； 整数比较和出栈
mem16 ； 字整数
mem32 ； 短整数

说明：

字整数或短整数格式的操作数，自动转换为临时实型数，然后与栈顶比较。并按下表设置条件码：

	C3	C2	C0
ST>op_1	0	0	0
ST<op_1	0	0	1
ST==op_1	1	0	0
无序	1	1	1

若操作码为 FICOMP，栈被弹出。

若栈顶为 NaN，则发生无序比较。在比较时，不管零的符号。

操作：

```
tmp=(tempReal) op1;
if (ST is NaN) {
    C3=1; C2=1; C0=1;
} else if (ST>tmp) {
    C3=0; C2=0; C0=0;
} else if (ST<tm   {
    C3=0; C2=0, C0=1;
} else if (ST==tmp) {
    C3=1; C2=0; C0=0;
}
if (OPCODE==FICOMP) {
    FPop( );
}
```

例子：

FICOMP mem16

执行前：mem16：94320 ST：$6.0059 * 10^3$
TOP：4 C3：× C2：× C0：×
执行后：mem16：94320
TOP：5 C3：0 C2：0 C0：1

异常：

```
Float (IS,I,D);
if (ST is QUIETNaN); {/* 栈顶为静的 NaN */
   Float (I);
}
```

注意：

第 4 章程序 7 给出了一个 80387 按条件码跳转的例子。

FILD

整数装入

格式：

FILD op_1

mem16 ;字整数

mem32 ;短整数

mem64 ;长整数

说明：

将 op_1(内存字整数，内存短整数，或内存长整数)从内存读出，转换成临时实型数格式，再将其压入栈顶。

操作：

```
FPush ( );
ST -(tempReal) op1;
```

例子：

FILD mem32

执行前：mem32 ： F234D9A1 TOP：6

执行后：mem32 ： F234D9A1 TOP：5

ST ： $-2.31417439*10^8$

异常：

Float (IS);

Memory ( );

FINCSTP

栈指针加一

格式：

FINCSTP

说明：

状态字中的栈指针被加一。标记字(TW)和浮点栈的内容不变。如果执行 FINCSTP 指令时栈顶指针(TOP)为 7，TOP 便变成 0。弹出一个栈元素，栈指针便加一。

操作：

TOP=(TOP+1) % 8;

例子：

FINCSTP

执行前：TOP：7

执行后：TOP：0

注意：

该指针可以直接控制栈指针。当需要一个比 80387 提供的 8 个硬件累加器更大的“虚拟”累加器栈时，这条指令非常有用。这些“虚拟”栈寄存器是驻留内存的，当检测到栈无效操作异常时，这个虚拟的堆栈应由软件来管理。支持虚拟累加器栈的指令有 FDECSTP，FINCSTP，和 FFREE。

FINIT/FNINIT

初始化处理器

格式：

FINIT

FNINIT

说明：

FINIT/FNINIT 指令将所有的控制字，状态字和标记字寄存器，设置为它们的默认值。该指令执行的结果：使机器的舍入控制设置为最近舍入；所有的异常都被屏蔽；精度定为 64 位；状态字中除了四个条件码位为无定义之外，其余的位都置为 0；所有的浮点栈寄存器(TW)都置为空。

FINIT 检查未屏蔽的数字异常；FNINIT 不检查。

操作：

```
CW=037Fh; /*最近舍入,屏蔽所有异常*/
          /*64 位精度*/
SW=SW & 4700h;
TW=FFFFh;
```

例子：

FINIT

执行前：CW：1D7E　SW：D401　TW：3FFC

执行后：CW：037F　SW：4400　TW：FFFF

FIST

整数存储

格式：

FIST op_1

mem16 ;字整数

mem32 ;短整数

说明：

栈顶被舍入为一整数，其长度与 op_1 一致，为字整数或短整数。舍入的方式取决于控制字(CW)中的舍入控制(RC)位。然后，该整数被存入 op_1。在存储前，负 0 转换为正 0 之补码。

操作：

```
if (op1 is mem16) {
    op1=(WordInt) ST;
} else {   /* mem32 */
    op1=(shorInt) ST;
}
```

例子：

FIST mem16 ;上舍入

执行前：ST : $8.90133 * 10^1$ TOP : 5 RC : RU

执行后：ST : $8.90133 * 10^1$ TOP : 5 RC : RU

mem16 : 005A

异常：

```
Float (IS,I,P);
Memory ( );
if (ST is QUIETNaN) {
    Float (I);
}
```

注意：

FIST 不能存储长整型数。

一般情况下，对于静的 NaN 的操作不引起操作异常。但在这里，如果 ST 是个静的 NaN，产生一无效操作异常。

FISTP

整数存储并弹出

格式：

FISTP op_1

mem16 ; 字整数

mem32 ; 短整数

mem64 ; 长整数

说明：

栈顶被舍入为一整数，其长度与 op_1 一致，为字整数，短整数或长整数。舍入的方式取决于控制字(CW)中的舍入控制(RC)位。然后该整数被存入 op_1。负 0 转换成正 0 之补码。栈被弹出。

操作：

```
if (op1 is mem16) {
  op1=(WordInt) ST;
} else if (op1 is mem32) {
  op1=(shortInt) ST;
} else { /* mem64 */
  op1=(LongInt)ST;
}
FPop( );
```

例子：

FISTP mem64 ;最近舍入

执行前：ST : $-4.32990*10^3$ TOP：5 RC：RN

执行后：mem64 : FFFFFFFFFFFFEF16 TOP：6 RC：RN

异常：

Float (IS,I,P);

Memory ( );

注意：

FISTP 可以存储长整数，FIST 不能。

FLD

实 数 装 入

格式：

FLD op_1

mem32 ; 短实型数

mem64 ; 长实型数

mem80 ; 临时实型数

ST(i)

说明：

op_1 压栈，成为新的栈顶。若操作数为长短实型数，在装入新的栈顶前要转换成临时实型数。

操作：

```
FPush ( );
ST=(tempReal) op1;
```

例子：

FLD mem32；mem32 为短实型数

执行前：mem32 : $3.2611*10^2$ TOP：2

执行后：mem32 : $3.2611*10^2$ TOP：1

ST : $3.2611*10^2$

异常：

Memory ( );

```
if (op1 in (mem32, mem64)) {
Float (I,D);
}
Float (IS);
```

注意：

在所有含有扩展的格式中，80287 将标志 Float(D)异常，而 80387 仅对短实数或长实数标志 Float(D)异常。

FLDcon

装 入 常 数

格式：

FLD1	；	装入 1.0
FLDL2E	；	装入 $\log_2 e$
FLDL2T	；	装入 $\log_2 10$
FLDLG2	；	装入 $\log_{10} 2$
FLDLN2	；	装入 $\log_e 2$
FLDPI	；	装入 π
FLDZ	；	装入 0.0

说明：

将指令本身规定的常数压入栈内，成为新的栈顶。

操作：

```
FPush ( );
switch (OPCODE) {
  case FLD1      : ST=+1.0      ;break;
  case FLDL2E    : ST=log2e     ;break;
  case FLDL2T    : ST=log210    ;break;
  case FLDLG2    : ST=log102    ;break;
  case FLDLN2    : ST=loge2     ;break;
  case FLDPI     : ST=π         ;break;
  case FLDZ      : ST=+0.0      ;break;
}
```

例子：

FLD1

执行前：TOP ：6

执行后：TOP ： 5　ST：$1.6 * 10^0$

FLDPI

执行前：TOP ： 0

执行后：TOP ： 7　ST：$3.1416 * 10^0$

异常：

Float (IS)；

注意：请看第 6 章程序 9。

FLDCW

装入控制字

格式：

FLDCW mem16

说明：

FLDCW 将在 mem16 中的值装入控制字。如果 SW 中的所有异常标志都未被新的 CW 屏蔽的话，将出现一个异常。

操作：

CW=mem16;

例子：

FLDCW mem16

执行前：CW：xxxx mem16：1F4F

执行后：CW：1F4F mem16：1F4F

异常：

Float (I,IS,D,Z,O,U,P,S)

FLDENV

装 入 环 境

格式：

FLDENV mem

说明：

FLDENV 将给定的内存区里内容装至机器的环境。机器环境由控制字、状态字、标记字和最新执行的浮点指令的出错指针寄存器组成。关于出错指针寄存器(Error-Pointer Register)的格式详见第 2 章。出错指针寄存器中含有关于最新操作码和被访数据的信息。

操作：

CW=mem;
SW=[&mem+4];
TW=[&mem+8];
LdErrorPoint (&mem);

例子：

FLDENV mem

执行前：CW：xxxx SW：xxxx TW：xxxx EIP：xxxxxxxx

FCS：xxxx：FDP：xxxxxxxx FDS：xxxx

[&mem+0]：xxxx137F [mem+4]：xxxx0400

[&mem+8]：××××A0FE　　[mem+12]：0000FFD0
[&mem+16]：××××0241　　[mem+20]：005F3DD0
[&mem+24]：××××3015

执行后：CW：137F　SW：0400　　TW：AOFE　FIP：0000FFD0
FCS：0241　FDP：005F3DD0　FOS：3015
[&mem+0]：××××137F　　[mem+4]：××××0400
[&mem+8]：××××AOFE　　[mem+12]：0000FFD0
[&mem+16]：××××0241　　[mem+20]：005F3DD0
[&mem+24]：××××3015

异常：

Float (I,IS,D,Z,O,U,P,S)

FMUL/FIMUL/FMULP

乘

格式：

```
FMUL                    ;  实数乘,ST(1),ST
FMUL  op1               ;  实数乘
      ST(n)
      mem32             ;  短实数
      mem64             ;  长实数
FIMUL op1               ;  整乘数
      mem16             ;  字整数
      mem32             ;  短整数
FMULP op1    ,  op2     ;  整数乘并弹出
      ST     ,  ST(n)
      ST(n)  ,  ST
FMUL  op1    ,  op2     ;  实数乘
      ST     ,  ST(n)
      ST(n)  ,  ST
```

说明：

如不规定操作数，隐指 ST(1) * ST，结果存入栈顶。若只规定一个操作数，隐指 ST * op_1，结果存入栈顶。若规定了两个操作数，则执行 op_1 * op_2，结果放入 op_1(ST 或 ST(n))。

如果 op_1 为内存操作数(字整数，短整数，短实数，长实数)，在它参与任何操作之前，自动转换为内部的临时实型数格式。

不带任何操作数的 FMUL，可令堆栈弹出。所以，不带操作数的 FMUL，与 FMULP ST(1),ST 同义。

操作：

```
if (NUMOPS==0) {
```

```
        ST(1)=ST(1) * ST;
        FPop( );     /*栈弹出,结果放在ST中*/
    } else if (NUMOPS==1) {
        ST=ST * (tempReal) op1;
    } else {          /*NUMOPS==2*/
        op1=op1 * op2;
        if (OPCODE==FMULP) {
            FPop( );          /*栈弹出*/
        }
    }
```

例子:

FMULP ST, ST(3)

执行前: ST : $8.410 * 10^{-12}$ TOP : 4

ST(3) : $9.934 * 10^{9}$

执行后: ST : $8.354 * 10^{-2}$ TOP : 5

ST(2) : $9.934 * 10^{9}$

异常:

Memory ( );

Float (IS,I,O,P,D,U);

注意:

下表概括了各种对于无穷数的乘操作:

		结　果(op_1)	
op_1	op_2	符　号	量值
±∞	±∞	操作数符号之异或	∞
±∞	±×	操作数符号之异或	∞
±0	±∞	无效操作	
±∞	±0	无效操作	

其中 0<x<∞

请看第 6 章程序 10。

FNOP

空 操 作

格式:

FNOP

说明:

FNOP 将栈顶存入栈顶,等效于 NOP 指令。

操作:

ST=ST;

例子:

FNOP

FPATAN

部分反正切

格式：

FPATAN

说明：

计算 ST(1)/ST 的反正切。计算完成之后，栈顶被弹出，计算结果放入新的栈顶。

80387 对于 FPATAN 操作数范围不加限制。然而其结果的范围为：

ST(1)的符号	ST 的符号	|ST(1)|<|ST|	结　果
+	+	是	ATAN(ST(1)/ST)
+	+	否	π/2−ATAN(ST(1)/ST)
+	−	是	π−ATAN(ST(1)/−ST)
+	−	否	π/2−ATAN(−ST(1)/ST)
−	+	是	−ATAN(ST(1)/ST)
−	+	否	−π/2+ATAN(ST(1)/−ST)
−	−	是	−π+ATAN(−ST(1)/−ST)
−	−	否	−π/2−ATAN(−ST(1)/−ST)

操作：

```
ST(1)=arctan(ST(1)/ST);
FPop( );  /*FPop 之后结果放入 ST*/
```

例子：

FPATAN

执行前：ST　　：4.51200　$*10^5$　TOP：4

　　ST(1)　：3.664　　$*10^3$

执行后：ST　　：8.12039　$*10^{-3}$　TOP：5

异常：

Float (IS,I,D,U,P);

注意：

80287 操作数范围为：

$0 \leqslant |ST(1)| < |ST(0)| < +\infty$

而 80387 无此限制。

FPREM

部分余数

格式：

FPREM

说明：

FPREM 计算 ST/ST(1)的部分余数。产生的余数是精确的(没有精度误差是可能的)。这余数是通过一系列成功的比例减产生的。当这两个操作数的量值相差甚远时，这一系列的减要花费很长时间。为防止中断等待的严重降低，指令只部分地计算余数，而采用软件循环完成余下的处理。所以，“部分余数”由此而得名。

FPREM 每执行一次，可以使被除数 ST 的绝对值减少 ST(1)绝对值的 2^{64}倍。若 ST 与 ST(1)在数值上相差不甚远($|ST|<|ST(1)|*2^{64}$)，除法 ST/ST(1)可一次完成，得出：

商=ST/ST(1)，

余数=ST%ST(1)，

条件码C2=0

C0,C3,C1 等于商的最低有效 3 位。

若$|ST|\geqslant|ST(1)|*2^{64}$，则得部分余数：

$ST=ST\mp ST(1)*(2^{64}-1)$

(ST 与 ST(1)同号取“－”号，ST 与 ST(1)异号取“＋”号)

条件码 C2=1，

$\pm(2^{64}-1)$可视为暂时的商。

不管怎样，下面的恒等式总归成立：

(部分余数)=ST－ST(1)＊(暂时的)商

而且余数或部分余数与 ST 原来的符号一致。

FPREM 对于 80287 是重要的，因为大多数周期超越函数的自变量被限制在 0 到 $\pi/4$ 的范围内。FPREM 便将它们的自变量改变到合适的大小。但由于这样的参数限制在 80387 内全部被取消(真是太棒了!)，所以 FPREM 指令就没有那么重要了。

操作：

```
if (|ST|<|ST(1)|*2^64) {
  C2=0
  quotient=ST/ST(1);
  ST=ST%ST(1);
  /*余数的符号与ST原来的符号一致   */
  if ((quotient % 8)>=4) C0=1;
  if ((quotient % 4)>=2) C3=1;
  if ((quotient % 2)  =1) C1=1;
} else {  /*以下假定了ST(1)>0   */
  if (ST>0) {
     ST=ST-ST(1)*(2^64-1);
  } else {ST=ST+ST(1) * (2^64-1);}
  C2=1;
}
```

例子：

FPREM			CCCC
执行前：	ST	：$-4.317*10^2$	3 2 1 0
	ST(1)	：$6.283*10^0$	××××
执行后：	ST	：$-4.456*10^0$	
	ST(1)	：$6.283*10^0$	0 0 0 1

/* 商为 10111100,(−68) */

异常:

Float (IS,I,U,D);

注意:

条件码 C0,C3,C1 在 80287 内,可能不会真正等于商的最低有效三位。

请阅 FPREM1。

若 ST(1)为∞,则 ST 不变,商为 0。

请看第 4 章程序 8。

FPREM1

部分余数——IEEE

格式:

FPREM1

说明:

FPREM1 是为了和 IEEE 兼容而设。

FPREM1 与 FPREM 的区别在于:在 ST/ST(1)的除法运算一次完成之后,FPREM1 所得的余数总在−abs(ST(1)/2)到 abs(ST(1)/2)之间。(abs 为求绝对值的函数。)而 FPREM,当 ST>0,余数在 0 到 abs(ST(1))之间;若 ST<0,余数在−abs(ST(1))到 0 之间。因而 FPREM1 和 FPREM,对于同一对 ST 和 ST(1),所得的商也不尽相同。

FPREM1 所得的商是最接近 ST/ST(1)的整数,而 FPREM 所得商之绝对值总是小于等于 ST/ST(1)的绝对值。

其余请看 FPREM 指令。

操作:

```
if (|ST|<|ST(1)| * 2^64) {
    C2=0;
    quotient=ST/ST(1);
    ST=ST % ST(1);
    /* 除法和求模的结果导致位于栈顶的余数在−abs(ST(1)/2)到 abs(ST(1)/2)范围内 */
    if ((quotient % 8)>=4) C0=1;
    if ((quotient % 4)>=2) C3=1;
    if ((quotient % 2)==1) C1=1;
} else {  /* 以下假定 ST(1)>0  */
    if (ST>0) {
        ST=ST−ST(1) * (2^64−1);
    } else {ST=ST+ST(1) * (2^64−1);}
    C2=1;
}
```

例子:

FPREM1			C C C C
执行前:	ST	: $-4.317 * 10^2$	3 2 1 0
	ST(1)	: $6.283 * 10^0$	× × × ×
执行后:	ST	: $1.827 * 10^0$	
	ST(1)	: $6.283 * 10^0$	1 0 1 0

异常：

Float (IS,I,U,D)；

FPTAN

部分正切

格式：

FPTAN

说明：

计算 ST 的正切函数。在计算完成之后，栈顶变为原 ST 的正切值，再将常数 1.0 压入栈内。即

ST(1)/ST=tan(ST′)

其中 ST′为 FPTAN 指令执行前的栈顶。ST 为 1.0，ST(1)为 ST′的正切值。

FPTAN 的操作数必须在 0 到 $\pi * 2^{62}$范围内。若源操作数在此范围内，C2 被清 0。否则，C2 置 1，ST 以及栈顶指针(TOP)不变。

操作：

```
if (ST<π*2^62) {
    ST=tan (ST);
    FPush ( );
    ST=1.0;
    C2=0;
} else {
    /*注意 ST 和 TOP 均无改变*/
    C2=1;
}
```

例子：

FPTAN

执行前：ST　　：$9.47103 * 10^2$　TOP：4　C2：×

执行后：ST　　：$1.00000 * 10^0$　TOP：3　C2：0

　　ST(1)　：$1.14408 * 10^1$

异常：

Float (IS,I,D,U,P)；

FRNDINT

舍入为整数

格式：

FRNDINT

说明：

栈顶元素被舍入为一整数。控制字中的 RC(舍入控制)字段的设置决定了舍入方式。

RC 字段	舍入方式
00	舍入为最接近的整数(四舍五入)
01	朝负无穷的方向舍入
10	朝正无穷的方向舍入
11	截短(chop),即朝 0 的方向舍入

操作

ST=(int) ST;

例子:

FRNDINT ;最近舍入

执行前: ST : $-4.317 * 10^0$ RC : 00

执行后: ST : $-4.000 * 10^0$ RC : 00

FRNDINT 朝负无穷的方向舍入

执行前: ST : $-4.317 * 10^0$ RC : 01

执行后: ST : $-5.000 * 10^0$ RC : 01

异常:

Float (IS,I,D,P);

FRSTOR

恢 复 状 态

格式:

FRSTOR mem

说明:

从由 mem 指定的 108 字节单元中重新恢复 80387 的全部状态。其中包括 80387 的环境和 8 个 80b 浮点栈寄存器。

装入内存的格式和 FSAVE 指令完全一样。

若状态字和控制字的结合是在一异常结合之中,那么任何异常都是可能发生的。

操作:

```
CW=mem;
SW=[&mem+4];
TW=[&mem+8];
LdErrorPoint (&mem);
for (i=0; i<8; i++) {
    ST(i)=[&mem+28+i * 10];
}
```

例子:

FRSTOR mem; mem 是个 108 字节的数据单元

异常:

Float (I,IS,D,Z,O,U,P,S)

Memory ( );

FSAVE/FNSAVE

保存状态

格式：

FSAVE mem

FNSAVE mem

说明：

FSAVE/FNSAVE 将 80387 的完整的状态存入内存单元 mem。保存的状态包括：控制字、状态字和标记字寄存器；出错指针寄存器；以及完整的浮点寄存器堆栈(8 个 80b 寄存器)。关于出错指针的详细格式以及保存的区域请看第 5 章，出错指针含有关于最新执行的指令和所使用的数据地址的信息。

在 80387 状态存完之后，FSAVE 会象 FINIT 指令一样重新将 80387 初始化。

FSAVE 检查未屏蔽的数字异常，而 FNSAVE 不检查。

操作：

```
mem=CW;
[&mem+4]=SW;
[&mem+8]=TW;
StErrorPointer (&mem);
for (i=0; i<8; i++) {
    [&mem+28+i*10]=ST(i);
}
FINIT;
```

例子：

FSAVE mem ;mem 是个 108 字节的数据单元。

异常：

Memory ( );

FSCALE

乘以 2 的幂

格式：

FSCALE

说明：

ST 乘以 2 的 ST(1)次方。也就是将 ST(1)的值(先用截短的舍入法变成了整数)加到 ST 的(2 的)指数中去。这是一个乘以 2 的幂的快速算法，很有用。

FSCALE 用一个数值上小于 ST(1)的，最接近 ST(1)的整数来代替 ST(1)参加运算。也就是说，参加运算的实际上是采用截短的方式由 ST(1)舍入的整数。但指令执行前后，ST(1)的内容并没有改变。

操作：

$ST = ST * 2^{ST(1)}$

例子：

FSCALE

执行前：ST　　：$-1.0111b * 2^6$

　　ST(1)　：$-1.0110b * 2^4$

执行后：ST　　：$-1.0111b * 2^{16}$

　　ST(1)　：$-1.0110b * 2^4$

(注：$ST(1) = -1.0110b * 2^4 = -1.375 \times 16 = -22$)

异常：

Float (IS,I,D,O,U,P)；

注意：

对于 80287，若操作数 ST(1)

$0 < |ST(1)| < 1$

则视操作数 ST(1)为 0，FSCALE 指令的结果无定义。而 80387 没有上述限制。

如果操作数引起上溢或下溢，则产生异常。

下表概括了带有无穷数的 FSCALE 操作：

操作数		结果	
ST	ST(1)	符号	值
$\pm\infty$	$-\infty$	无效操作	
$\pm\infty$	$+\infty$	同 ST	∞
$\pm\infty$	$\pm z$	同 ST	∞
± 0	$\pm\infty$	同 ST	0
$\pm x$	$+\infty$	上溢	
$\pm x$	$-\infty$	下溢	

其中$0 < x < \infty$

$0 \leqslant z < \infty$

FSIN

正　弦

格式：

FSIN

说明：

计算 ST 的正弦函数。计算完成后，栈顶被置换成 ST 的正弦值，再将常数 1.0 压栈。于是

$ST(1)/ST = \sin(ST')$

其中 ST′为指令执行前的栈顶。ST 为 1.0，ST(1)为 ST′的正弦。

FSIN 指令的操作数必须在 0 到 $\pi * 2^{62}$范围内。如果此条件满足，C2 被清 0。否则 C2 置 1，并且 ST 及 TOP 均不变。

操作：

```
if (ST<π*2^62) {
    ST=sin(ST);
    FPush( );
    ST=1.0;
    C2=0;
} else {
    /*ST 及 TOP 均无改变*/
    C2=1;
}
```

例子：

FSIN

执行前：ST　　：　$8.41229 * 10^{3}$　TOP：4　C2：×

执行后：ST　　：　$1.00000 * 10^{0}$　TOP：3　C2：0

ST(1)　：$-7.80288 * 10^{-1}$

异常：

Float (IS,I,D,U,P);

注意：

该指令为 80387 新增的。

FSINCOS

正弦和余弦

格式：

FSINCOS

说明：

计算 ST 的正弦函数余弦函数。计算完成之后，栈顶被置换成 ST 的正弦值，再将原 ST 的余弦值压入栈内。于是执行的结果成为：

ST(1)=sin(ST′)

ST　　=cos(ST′)

ST′为指令 FSINCOS 执行前的栈顶。

FSINCOS 指令的操作数必须在 0 到 $\pi * 2^{62}$范围内。若该条件满足，C2=0。否则 C2=1，ST 和 TOP 不变。FSINCOS 指令结果的精度可能不及 FSIN 和 FCOS 指令。

操作：

```
if (ST<π*2^62) {
    tmp=cos(ST);
    ST=sin(ST);
    FPush ( );
    ST=tmp;
    C2=0;
} else {
    /* ST 及 TOP 均无改变*/
    C2=1;
```

}

例子：

FSINCOS

执行前：ST　：2.97421　* 10^6　TOP：4　C2：×

执行后：ST(1)：9.86024　* 10^{-1}　TOP：3　C2：0

ST　：1.66602　* 10^{-1}

异常：

Float (IS,I,D,U,P);

注意：

该指令为 80387 新增的。

FSQRT

平 方 根

格式：

FSQRT

说明：

栈顶被栈顶的平方根置换。

注意：

FSQRT(−0)=−0

FSQRT(+∞)=+∞

FSQRT(−∞)为无效操作。

操作：

FSQRT

执行前：ST：8.410 * 10^4

执行后：ST：2.900 * 10^2

异常：

Float (IS,I,U,P,D);

注意：

请看第 6 章程序 11。

FST

实 数 存 储

格式：

FST op_1

mem32　；短实数

mem64　；长实数

ST(i)

说明：

栈顶被存入 op_1。若 op_1 为短实数或长实数，在存储之前，栈顶先按照 RC(舍入控制)进行类型转换。若栈顶为 NaN 或无穷数，栈顶的指数及有效数字被截短，而不是舍入为吻合目标尺寸。

操作：

```
if (op1 is mem32) {
    op1=(shortReal) ST;
} else if (op1 is mem64) {
    op1=(longReal) ST;
} else {
    op1=ST;
}
```

例子：

FST mem64；mem64 是长实数

执行前：ST	：$2.9921 * 10^{-9}$	TOP：2
mem64	：$\times.\times\times\times\times * 10^{x}$	
执行后：ST	：$2.9921 * 10^{-9}$	TOP：2
mem64	：$2.9921 * 10^{-9}$	

异常：

Memory ( )；

Float (IS,I,O,U,P)；

注意：

为保持与 80287 兼容，在 FST 指令期间，D 异常不被标志。

FST 不能将临时实型数存入内存，而 FSTP 可以。

FSTCW/FNSTCW

存储控制字

格式：

FSTCW mem16

FNSTCW mem16

说明：

FSTCW/FNSTCW 将控制字存入 mem16。FSTCW 检查未屏蔽的数字异常，而 FNSTCW 不检查。

操作：

```
mem16=CW;
```

例子：

FSTCW mem16

执行前：CW：1F4D　mem16：××××

执行后：CW：1F4D　mem16：1F4D

FSTENV/FNSTENV

存储环境变量

格式：

FSTENV　mem

FNSTENV　mem

说明：

FSTENV/FNSTENV 将机器环境存入指定的内存单元。所谓"环境"是指控制字，状态字，标记字寄存器，以及最新执行的浮点指令的出错指针。关于出错指针详见第 2 章。出错指针含有关于最新执行的指令和该指令使用的数据地址的信息。

FSTENV 检查未屏蔽的数字异常；而 FNSTENV 不检查。

操作：

```
mem=CW;
[&mem+4]=SW;
[&mem+8]=TW;
StErrorPointer (&mem);
```

例子：

FSTENV mem

执行前：CW ：137F SW： 0400　　TW：AOFE

FCS：0241 FOO：005F3DD0　FOS：3015

FIP：0000FFD0

[&mem+0]：××××××××　[mem+4]：××××××××

[&mem+8]：××××××××　[mem+12]：××××××××

[&mem+16]：××××××××　[mem+20]：××××××××

[&mem+24]：××××××××

执行后：CW：137F SW：0400　　TW：AOFE

FCS：0241 FOO：005F3DD0　FOS：3015

FIP：0000FFD0

[&mem+0]：××××137F　[mem+4]：××××0400

[&mem+8]：××××A0FE　[mem+12]：0000FFD0

[&mem+16]：××××0241　[mem+20]：005F3DD0

[&mem+24]：××××3015

异常：

Memory ( )；

FSTP

实数存储并弹出

格式：

FST op_1

mem32 ; 短实型数

mem64 ; 长实型数

mem80 ; 临时实型数

ST(i)

说明：

栈顶被存入 op_1。若 op_1 为短实型数或长实型数，在存储前，栈顶先按照 RC(舍入控制)进行类型转换。若栈顶为 NaN 或无穷数，则栈顶的指数和有效数字将被截短，而不是按目的存储器的大小舍入。存储完成后栈顶弹出。

FSTP 可以将临时实型数存入内存，而 FST 不能。

操作：

```
if (op1 is mem32) {
  op1=(shortReal) ST;
} else if (op1 is mem64) {
  op1=(longReal) ST;
} else {
  op1=ST;
}
FPop ( );
```

例子：

FSTP mem64

执行前：	ST	：8.9901	$*10^{-102}$	TOP：7
	mem64	：x.xxxx	$*10^{x}$	
执行后：	ST	：应为原 ST(1)之值		TOP：0
	mem64	：8.9901	$*10^{-102}$	

异常：

Memory ( );

Float (IS,I,O,U,P);

注意：

为保持与 80287 的兼容，在 FSTP 指令期间不标志 D 异常。

请阅第 6 章程序 10。

FSTSW/FNSTSW

储存状态字

格式：

FSTSW mem16

FNSTSW mem16

说明：

FSTSW/FNSTSW 将当前状态字存入 mem16。FSTSW 检查未屏蔽的数字异常，FNSTSW 不检查。

操作：

mem16=SW；

例子：

FSTSW mem16

执行前：SW：FF81 mem16：××××

执行后：SW：FF81 mem16：FF81

异常：

Memory ( )；

FSTSW AX/FNSTSW AX

将状态字存入 AX

格式：

FSTSW AX

FNSTSW AX

说明：

FSTSW AX/FNSTSW AX 将当前状态字存入 80386 的 AX 寄存器。该指令用于根据 80387 的条件标志进行控制流转移。

FSTSW 检查未屏蔽的数字异常，而 FNSTSW 不检查。

操作：

AX=SW；

例子：

FSTSW AX

执行前：SW：0900 AX：××××

执行后：SW：0900 AX：0900

注意：

请阅第 6 章程序 7。

FSUB/FISUB/FSUBP

减

格式：

```
FSUB                    ;  实数减,ST(1)-ST
FSUB  op1               ;  实数减
      ST(n)
      mem32             ;  短实数
      mem64             ;  长实数
FISUB op1               ;  整数减
      mem16             ;  字整数
      mem32             ;  短整数
FSUBP op1     ,  op2    ;  实数减并弹出
      ST      ,  ST(n)
      ST(n)   ,  ST
FSUB  op1     ,  op2    ;  实数减
      ST      ,  ST(n)
      ST(n)   ,  ST
```

说明：

若不规定操作数，执行 ST(1)-ST，结果放入栈顶。

若规定一个操作数 op_1，执行 ST-op_1，结果放入栈顶。

若规定两个操作数，执行 op_1-op_2，结果放入 op_1(ST 或 ST(n))。

若 op_1 为内存操作数(字整数或短整数，短实数或长实数)，在用它执行任何操作之前，它必先自动转换成内部的临时实型数格式。

无任何操作数的 FSUB 可使栈被弹出。因此，没有操作数的 FSUB 与 FSUBP ST(1),ST 同义。

操作：

```
if (NUMOPS==0) {
    ST(1)=ST(1)-ST;
    FPop( );        /*弹出栈,结果在ST中*/
} else if (NUMOPS==1) {
    ST=ST-(tempReal) op1;
} else {            /*NUMOPS==2*/
    op1=op1-op2;
    if (OPCODE==FSUBP) {
        FPop( );        /*栈弹出*/
    }
}
```

例子：

FSUB ST, ST(3); ST=ST-ST(3);

执行前：ST　　：$8.410 * 10^{12}$　TOP：7

　　ST(3)　：$9.934 * 10^{11}$

执行后：ST　　：$7.419 * 10^{12}$　TOP：7

　　ST(3)　：$9.934 * 10^{11}$

异常：

Memory ( );

Float (IS,I,O,P,D,U);

注意：

下表摘要带有无穷数的减操作：

操作数		结果 (op_1)	
op_1	op_2	符号	值
$+\infty$	$-\infty$	+	∞
$-\infty$	$+\infty$	−	∞
$+\infty$	$+\infty$	无效操作	
$-\infty$	$-\infty$	无效操作	
$\pm\infty$	$\pm z$	同∞	∞
$\pm z$	$\pm\infty$	与∞符号相反	∞

其中 $0 \leqslant z < \infty$

FSUBR/FISUBR/FSUBRP

反　减

格式：

```
FSUBR                  ; 实数反减,ST-ST(1)
FSUBR  op1             ; 实数反减
       ST(n)
       mem32           ; 短实数
       mem64           ; 长实数
FISUBR op1             ; 整数减
       mem16           ; 字整数
       mem32           ; 短整数
FSUBRP op1   , op2     ; 实数反减
       ST    , ST(n)
       ST(n) , ST
FSUBR  op1   , op2     ; 实数反减
       ST    , ST(n)
       ST(n) , ST
```

说明：

FSUBR/FISUBR/FSUBRP 与 FSUB/FISUB/FSUBP 相比，只是减数和被减数相互颠倒而已，其余皆同。

操作：

```
if (NUMOPS==0) {
    ST(1)=ST-ST(1);
    FPop( );  /*栈顶弹出,结果在栈顶中  */
} else if (NUMOPS==1)  {
    ST=(tempReal) op1-ST;
```

```
} else {        /* NUMOPS==2 */
    op1=op2-op1;
    if (OPCODE==FSUBRP) {
        FPop( );        /* 栈顶弹出 */
    }
}
```

例子：

FSUBR mem ; ST=mem−ST

执行前：mem : 9934

ST : $8.410 * 10^4$

执行后：mem : 9934

ST : $-7.4166 * 10^4$

异常：

Memory ( );

Float (IS,I,O,P,D,U);

注意：下表概述了带有无穷数的反减：

操作数		结果	
op_1	op_2	符号	值
$+\infty$	$-\infty$	−	∞
$-\infty$	$+\infty$	+	∞
$+\infty$	$+\infty$	无效操作	
$-\infty$	$-\infty$	无效操作	
$\pm\infty$	$\pm z$	与∞符号相反	∞
$\pm z$	$\pm\infty$	同∞	∞

其中 $0 \leqslant z < \infty$

FTST

测　试

格式：

FTST

说明：

将栈顶与 0 作数字比较。然后根据比较结果设置状态字中的条件码：

	C3	G2	C0
ST>0.0	0	0	0
ST<0.0	0	0	1
ST==0.0	1	0	0
无序	1	1	1

若 ST 为 NaN，则为无序比较。在比较中不管 0 的符号。

操作：

```
if (ST is NaN) {
    C3=1; C2=1; C0=1;
} else if (ST>0.0) {
    C3=0; C2=0; C0=0;
} else if (ST<0.0) {
    C3=0; C2=0; C0=1;
} else if (ST==0.0) {
    C3=1; C2=0; C0=0;
}
```

例子：

FTST

执行前：ST：$2.4560 * 10^6$　　TOP：4

C3：×　　C2：×　　C0：×

执行后：ST：$2.4560 * 10^6$　　TOP：4

C3：0　　C2：0　　C0：0

异常：

```
Float (IS,I,D);
if (ST is QUIETNaN) {
    Float (I);
}
```

FUCOM/FUCOMP/FUCOMPP

无序比较

格式：

FUCOM	op_1	；无序比较
	mem32	；短实数
	mem64	；长实数
	ST(i)	
FUCOMP	op_1	；无序比较并弹出
	mem32	；短实数
	mem64	；长实数
	ST(i)	
FUCOMPP		；无序比较并弹出两次

说明：

指定的操作数与栈顶进行数值比较，并按下表设置条件码。若操作码为 FUCOMPP，与栈顶进行比较的操作数为 ST(1)。

	C3	C2	C0
ST>op_1	0	0	0
ST<op_1	0	0	1
ST==op_1	1	0	0
无序	1	1	1

若操作码为FUCOMP,栈被弹出一次。若操作码为FUCOMPP,栈被弹出两次。若操作数中有NaNs,则为无序。比较时不考虑0的符号。

操作:

```
if (OPCODE==FUCOMPP) {
  op1=ST(1);
}
if ((ST is NaN) || (op1 is NaN)) {
  C3=1; C2=1; C0=1;
} else if (ST>op1) {
  C3=0; C2=0; C0=0;
} else if (ST<op1) {
  C3=0; C2=0; C0=1;
} else if (ST==op1) {
  C3=1; C2=0; C0=0;
}
if (OPCODE==FUCOMP) {
  FPop ( );
} else if (OPCODE==FUCOMPP) {
  FPop ( );
  FPop ( );
}
```

例子:

FUCOMPP

执行前: ST : $6.1224 * 10^{-4}$

ST(1) : $5.3210 * 10^{-11}$

TOP : 4 C3 : × C2 : × C0 : ×

执行后: TOP : 6 C3 0 C2 : 0 C0 : 0

异常:

Float (IS,I,D);

注意:

除了FUCOM不会因操作数是静的NaN而引起异常外(FCOM会因此引起异常),FUCOM(以及所有的不同的弹出形式)等同于FCOM(包括它的各种变形)。

FXAM

检　验

格式:

FXAM

说明：

检验栈顶，并按下表设置状态字中的条件码：

C3	C2	C1	C0	ST 的值
0	0	0	0	+不支持的数
0	0	0	1	+NaN
0	0	1	0	−不支持的数
0	0	1	1	−NaN
0	1	0	0	+规格数
0	1	0	1	+∞
0	1	1	0	−规格数
0	1	1	1	−∞
1	0	0	0	+0
1	0	0	1	+空
1	0	1	0	−0
1	0	1	1	−空
1	1	0	0	+非规格数
1	1	0	1	+空
1	1	1	0	−非规格数
1	1	1	1	−空

所谓不支持数包括：伪 NaN，伪无穷数，未规格数，伪 0。详见第 1 章。

操作：

```
if (ST<0.0) {
    C1=1;
}
if      (ST is UNSUPPORTED )  {C3=0;C2=0;C0=0;}
else if (ST is NaN         )  {C3=0;C2=0;C0=1;}
else if (ST is NORMAL      )  {C3=0;C2=1;C0=0;}
else if (ST is INFINITY    )  {C3=0;C2=1;C0=1;}
else if (ST==0.0           )  {C3=1;C2=0;C0=0;}
else if (ST is DENORMAL    )  {C3=1;C2=1;C0=0;}
else if (ST is EMPTY       )  {C3=1;C2=0;C0=1;}
```

例子：

FXAM

执行前：ST：3.21950 * 10^{-131} TOP：2

C3：× C2：× C1：× C0：×

执行后：ST：3.21950 * 10^{-131} TOP：2

C3：0 C2：1 C1：0 C0：0

注意：

以上数据类型详见第 1 章。

FXCH

交换寄存器

格式：

FXCH

FXCH op_1

ST(n)

说明：

op_1 的内容与栈顶内容交换。许多 80387 指令仅对栈顶进行操作(如 FSQRT,FSIN,FPATAN),而 FXCH 便于这些指令对其它栈元素进行操作。

操作：

```
if (NUMOPS==1) {
    tmp=ST;
    ST=ST(n);
    ST(n)=tmp;
} else {
    tmp=ST;
    ST=ST(1);
    ST(1)=tmp;
}
```

例子：

FXCH ST(3)

执行前：	ST	: 9.99321	$*10^{-92}$	TOP : 3
	ST(3)	: −7.22964	$*10^{75}$	
执行后：	ST	: −7.22964	$*10^{75}$	TOP : 3
	ST(3)	: 9.99321	$*10^{-92}$	

异常：

Float (IS);

注意：

请看第 6 章程序 11。

FXTRACT

摘出指数和有效数字

格式：

FXTRACT

说明：

原栈顶之非偏移指数,以实数形式置换了栈顶的内容。而原栈顶的有效数字及符号再压入栈,成为新的栈顶。于是原栈顶分解为指数(ST(1))—非偏移指数和有效数字(ST)两个部分。有效数字部分的指数为 0。

操作：

```
tmp=ST;
if (tmp==0) {
    ST=-INFINITY;
} else {
    /* 注意偏移 16383 被减去 */
    ST=(tempReal) ((tmp>>64) & 7FFFh) -16383;
}
FPush ( );
                /*  64 位 */
ST=(tempReal) (tmp & FFFFFFFFFFFFFFFFh/2^63);
if (tmp<0.0) {
    ST=-ST;
}
```

例子：

FXTRACT

执行前：ST　　：　$1.1001101b * 2^{39}$　TOP=3

执行后：ST　　：　$1.1001101b * 2^{0}$　TOP=2

　　ST(1)　：　$1.00111b * 2^{5}$

异常：

Float (IS,I,Z,D);

注意：

若操作数为 0,ST(1)(指数)置为$-\infty$,ST(有效数字)被置为 0,而符号与原栈顶符号相同。这与 80287 不同。若操作数为$+\infty$,ST(1)为$+\infty$,ST 为∞,符号同原栈顶符号。

FYL2X

$$y * \log_2 x$$

格式：

FYL2X

说明：

利用栈顶部的两个元素来计算函数：

$ST(1) * \log_2 ST$。

栈顶被弹出,新栈顶被置换为上式计算结果。

ST 必须在 0 到$+\infty$范围内,ST(1)可从$-\infty$到$+\infty$。若操作数超出上述范围,指令将产生一无定义的结果,但检测不到异常。

EYL2X 可用来计算任意底的对数。例如计算 $\log_n X$,利用对数公式,可得

$$\log_n X=(1/\log_2 n) * \log_2 X$$

操作：

```
if ((0≤ST<+INFINITY) &&
    (-INFINITY<ST(1)<+INFINITY)) {
    tmp=ST(1) * log2(ST);
    FPop ( );
```

```
        ST=tmp;
    } else {
        UNDEFINED;
    }
```

例子：

FYL2X

执行前：ST : 9.87311 * 10^{9}

ST(1) : −2.79022 * 10^{-3}

执行后：ST : −9.26377 * 10^{-2}

异常：

Float (IS,I,D,U,P);

注意：

请阅第 6 章程序 9。

FYL2XP1

$$y * \log_2(x+1)$$

格式：

FYL2XP1

说明：

利用栈顶部的两个元素来计算函数

$$ST(1) * \log_2(ST+1.0)$$

栈顶被弹出，新的栈顶被置换成上式计算结果。

ST 取值范围为

$-(1-\sqrt{2}/2)$到$+(1-\sqrt{2}/2)$。

ST(1)取值范围为$-\infty$到$+\infty$。若操作数超出上述范围，将产生无定义的结果，但检测不到异常。

操作：

```
if ((−(1−sqrt(2)/2)<ST     <(1−sqrt(2)/2 )) &&
    (−INFINITY   <ST(1) <+INFINITY  )) {
    tmp=ST(1) * log2(ST+1.0);
    FPop ( );
    ST=tmp;
} else {
    UNDEFINED;
}
```

例子：

FYL2XP1

执行前：ST : 0.17230 * 10^{0}

ST(1) : 2.99013 * 10^{13}

执行后：ST : 6.85762 * 10^{12}

异常：

Float (IS,I,D,U,P);

WAIT

等待协处理器

格式：

WAIT

说明：

WAIT 将暂停 80386 的操作，直至数字协处理器变成不忙。

这条指令用于使主处理器与它的数字协处理器同步。

操作：

```
while (80387 is busy)
{
}
```

例子：

WAIT；不动作

注意：

只有在浮点数存入内存(如：FST(P)，FIST(P)和 FBSTP)之后和该存入的数据被 80386 某指令所用之前才需要 WAIT 指令。该指令可确保在 80386 访问这个数据之前，这个数据就已由 80387 存入了内存。若在浮点数存入，到 80386 访问这个存入的数据之间，要执行另一条 80387 指令的话，那么 WAIT 指令就不需要了。8086/8087 和 80286/80287 要求使用 WAIT 指令的情况比 80386/80387 多。

第4章　保护虚拟地址方式

前面各章的内容，对于编写80386应用程序所需了解的各种功能，已经做了详细描述。这些内容对于编程人员具有直接的指导作用。本章及第5章将描述80386硬件直接支持的操作系统的各种特性(包括虚存管理及中断/异常处理等)，并假定读者已经熟悉多任务操作系统中有关虚拟存储管理的基本概念。本章最初的目标是帮助系统软件的开发人员熟悉386系统。然而应用程序设计人员在阅读本章内容后会发现，本章前半部分有助于理解80386存储管理、系统保护及任务支持的概念。对于使用多段模型(已在第2章做过详细讨论)编程的应用程序设计人员，这些知识是特别有用的。

操作系统，即使是单用户个人计算机，必须能同时处理多件事情，才能充分利用计算机硬件资源及用户时间。386CPU芯片使计算机具有足够的计算能力，耗时较多的命令或操作(例如大文件的排序、多目录的字符串查找等)，可以放在后台并行进行处理。在等待后台任务完成之前，空出来的前台可以用来完成其它的任务，例如执行其它的操作系统命令。一个任务中各程序之间的交互作用，根据使用者的意图和要求，可以来回进行切换。

在多用户分时操作系统支持下，系统中可以运行多个程序，这些程序分别构成不同用户的任务，这些任务可以是用户执行的操作系统命令、用户的应用程序，也可以是后台并行处理的命令或任务以及连续交互响应的字处理程序等。任务本身以某种方式共享处理器的时间及资源，使得不同的任务似乎是彼此异步地同时在执行。一个任务在需要等待某种事件(例如等待磁盘访问的完成)时，可以挂起其执行，让出处理机由另外的任务使用。此后，如某任务等待的事件已经发生(例如磁盘操作已经完成)，并由设备给出相应的中断信号，则处理机被中断，正执行的任务被挂起，转去处理被响应的中断要求。中断处理完成之后，有可能使等待事件的任务恢复执行。

操作系统使用本章及第5章描述的机制，对计算机的资源(主存储器空间、执行时间及外围设备)进行分配和保护。通过把这些资源分配给系统中的各个任务，并对这些资源进行保护，使所有任务得以有效运行直至完成。操作系统的资源分配及保护策略，由80386硬件存储管理及保护机制实现。80386的存储管理及保护机制，保护(系统中的)每一个任务不被另外的任务破坏。例如，操作系统通过使用存储管理机制，保证分配给不同任务的存储区域互不重叠(共享存储区域除外)；通过使用保护机制，保证系统中任何一个用户任务都不能访问分配给操作系统的存储区域。

用于分配及保护资源的设施分为3个组，每一组反映如上定义的一种资源类别：即主存储空间，处理器时间及外围设备。存储管理设施，负责存储空间的分配及保护；这些设施与第5章描述的中断及异常机制一起，进行处理器时间的分配；I/O管理设施进行外围设备的分配和保护。存储管理及I/O管理设施是本章以下部分要研究的主题。

异常是一种重要的操作系统设施，第5章将对此进行详细描述。为方便介绍其它的操作系统设施，本章也将提及异常的某些重要方面。系统中的异常，就象一条不可见的CALL指令。当不正常的或不合法的操作在系统中出现时，系统就强制执行这一指令。执行该指令的过程或者终止产生异常的程序，或者在排除异常的原因后返回原来的程序，并在产生异常的断点处恢

复执行。异常用于向操作系统软件报告不正常的或不合法的各种事件。读者将会发现异常是一种允许操作系统保持对各种资源进行控制的强制机制。如果一个程序试图要使用或获取不属于该程序的资源，则系统将产生异常，并向操作系统报告该程序的这一企图。

4.1 存储管理机制

任何完整的存储管理系统，都包含有两个关键的部分：保护及地址转换。提供保护的目的，是要避免系统中的一个任务访问属于另一任务的或属于操作系统的存储区域。地址转换，不仅使操作系统可以灵活地把存储区域分配给各个任务，而且也是一种重要的保护机制。

4.1.1 地址转换

正如在第1章所定义的，计算机中的物理存储器，是一个存储字节的线性数组。每一字节占用一个唯一的地址，称为该字节的物理地址。第2章定义了由程序访问存储器使用的地址。这些地址分两个部分去寻址80386使用的二维分段存储器。这个分为两个部分的地址，并不直接用于寻址物理存储器，而是由地址转换机制，把程序地址转换或映射为物理存储器地址。这种转换机制支持虚拟地址的概念，该虚拟地址就是由程序中的两个部分组成的地址。使用虚拟地址这一术语，是因为这样的地址，并不直接对应于物理存储器中的某一位置，而是通过虚拟——物理地址映射函数间接地进行对应。

注意：由程序产生的每一个地址，都是含有两个部分的虚拟地址，无论是在最关键的操作系统内部，还是在各种应用程序中都是这样。无论哪一种类型的程序在系统中运行，都由存储管理机制把虚拟地址转换为物理存储器地址。设计人员可以安排某些物理地址不被任何虚拟地址所映射，所以虚拟——物理地址的转换也为存储器保护创造了条件。除这种基本的保护以外，我们还可对地址转换函数加以扩展，即把某些虚拟地址视为无效地址。当出现无效虚拟地址时，转换机制将不建立物理地址，并向系统报告一个异常，以便操作系统软件可以采取适当的动作。

4.1.2 段机制和分页机制

为了减少规定地址转换函数所需的信息量，需将较大的连续存储块映射成单个单元。这时只需对整个存储块的映射做一次规定，而不要求对每个字节进行单独的说明。段机制和分页机制是两种广泛使用的地址转换技术。这两种机制之间的区别在于，如何将虚拟地址组织成块以便进行映射，对转换信息如何进行规定以及编程人员如何进行操作。

段机制和分页机制，都是使用驻留在存储器的各种表格，规定各自的转换函数。这些存储在存储器的表格，只允许由操作系统进行访问，应用程序不能对其进行修改。这一点很重要，因为修改转换表格的内容，也就改变了转换函数，进而也就改变了虚拟地址空间。实际上，操作系统为每一任务维护一个不同的转换表格集，使每个任务可使用的地址，跨越整个虚拟地址空间，其结果是每一任务具有不同的虚拟地址空间，并使任务之间彼此隔离开来。

80386以两级虚拟——物理地址转换使用段和分页两种机制（见图4.1）。第一级使用段机制，把包含两个部分虚拟地址空间地址，转换为一个中间地址空间的地址，这里的中间地址空间称为线性地址空间。线性空间的地址也就称为线性地址。第二级使用分页机制，把线性地址转换为物理地址。在地址转换过程中，段机制总是要启用的；分页机制则根据需要而被启用或

被禁用。如分页机制被禁用，则由段机制转换产生的线性地址，直接作为物理地址使用，相当于分页机制转换函数，把线性地址与物理地址进行等同映射。在图 4.1 中把这种情况作为绕过分页机制转换函数的旁路示出。所以，如分页机制被禁用，则线性地址就是物理地址，否则，要把线性地址用页表进行转换，才能得到物理地址。

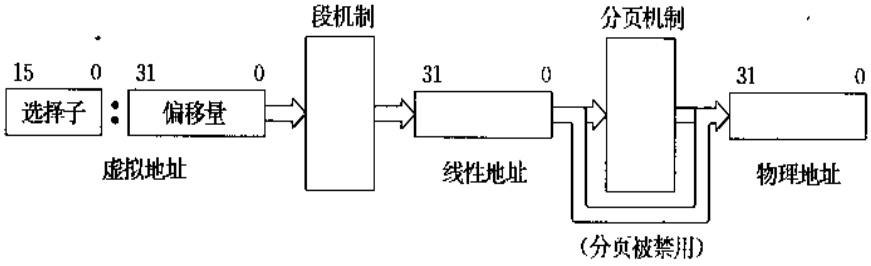


图 4.1 虚拟—物理地址转换

线性地址空间具有与物理地址空间同样的结构。和二维虚拟地址空间相对比，线性地址空间与物理地址空间都是一维地址空间。虚拟地址空间包含的段数，最大可到 16K 个，每个段可以有 4G 字节的容量，从而构成 64 兆兆(2^{46})字节容量的虚拟地址空间。而线性地址空间及物理地址空间的容量都是 4G(2^{32})字节。事实上，如分页机制被禁用，这时，线性地址等于物理地址，线性地址空间就是物理地址空间。

段及分页这两种机制是两种不同的转换机制，是整个地址转换函数的不同的转换级。虽然两种机制都利用存储在主存储器中的转换表，但这些表具有彼此独立的表结构。事实上，段表存储在线性地址空间，而页表存储在物理地址空间。因此，段转换表可由分页机制重新进行定位而不需段机制的参与。段转换机制把虚拟地址转换为线性地址，并在线性地址中访问段转换机制的表格，而不会觉察分页机制已把线性地址转换为物理地址。

类似地，分页机制对于程序产生的地址所使用的虚拟地址空间一无所知。分页机制只是直接地把线性地址转换为物理地址，并且在物理地址中访问转换表格，并不知道虚拟地址空间的存在，甚至不知道段转换机制的存在。

下面一节我们先对什么是虚拟存储进行简单的说明，并对段和分页机制的有关准则做一简要讨论，再引入某些基本的保护概念，然后再分别对段和分页机制进行描述。

4.1.3 虚拟存储器概念

虚拟存储器是一种设计技术，用于提供比在计算机系统中实际可以使用的物理主存储器大得多的存储器空间。使用者会产生一种错觉，好象在程序中可以使用非常大的物理存储空间，从而使得编程人员在写程序时，不用考虑计算机中物理存储器的实际容量，而在实际的计算机中并不具有那样大的空间。使用虚拟存储器的好处是：一个程序可以很容易地在物理存储容量大不一样的、配置范围很广的计算机上运行；编程人员使用虚拟存储器可以写出比任何实际配置的物理存储器都大得多的程序。

虚拟存储器由存储管理机制及一个大容量的快速硬盘存储器支持。在程序运行的任何时刻，只把虚拟地址空间的一小部分映射到主存储器，其余部分则存储在磁盘上。因为只有存储在主存储器中的部分虚拟存储器可由处理器使用，这种虚拟存储技术将依赖程序内部访问存储器的局部化特性，在程序执行中只需整个虚拟存储器中的少量存储内容在主存储中驻留。而

存储在一个单独的任务中。然而，我们即将看到，在一个任务内操作的保护机制，更适合于保护操作系统，使其不被应用程序破坏。这种机制，使操作系统由所有任务共享，并且可在每一任务对其进行访问，而且仍然保护了操作系统，使其不被应用程序破坏。这种保护操作系统的方法，是把操作系统存储在虚拟地址空间的一个公共区域，然后，再使每一任务按此区域分配一个同样的虚拟地址空间，并进行同样的虚拟—物理地址映射。各个任务公用的这部分虚拟地址空间，被称为全局地址空间。

仅由一个任务占有的虚拟地址空间部分，即不被任何其它任务共享的虚拟地址部分，称为局部地址空间。局部地址空间包含的代码和数据，是任务私有的，需要与系统中的其它任务相隔离。

在每个任务中有不同的局部地址空间。因此，两个不同任务中，对同一虚拟地址的访问，实际上转换为不同的物理地址。这就使操作系统对每个任务的存储器，可以赋予相同的虚拟地址，仍然保证任务的隔离。另一方面，对全局地址空间中同一虚拟地址的访问，在所有任务中都转换为同样的物理地址，从而支持公共的代码及数据的共享，例如对操作系统的共享。在本章详细描述段和分页机制的有关小节中，将通过实例对全局的及局部的地址空间进行更详细的讨论。

第 6 章中给出的初始化实例，展示出全局和局部地址空间，如何像假定的那样进行定义。在该实例中，操作系统在全局地址空间，并且被所有任务共享，每个任务的应用程序代码和数据，则存放在局部地址空间。

4.1.4.2　同一任务内的保护

在一个任务之内，定义有四种执行特权级别，用来限制对任务中的段进行访问。这些任务，按照包含在段中的数据的敏感性及任务中不同部分的程序，对可信任的程度进行区分。最敏感的数据，分配最高的特权级别。在最高特权级别中，数据只能由任务中最受信任的部分进行访问。不太敏感的数据，分配较低的特权级别。在较低的特权级别中，任务中不太被信任的部分也可访问这些数据。

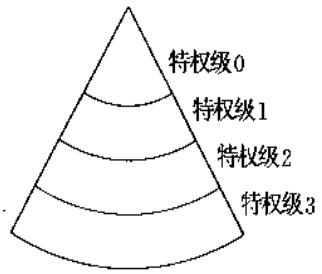


图 4.2　四种特权级

特权级别用数字 0～3 表示，数字 0 表示最高特权级，而数字 3 表示最低特权级。要弄清楚用数字赋值的级别 0～3 的意义，只需记住：数字较大的级别具有较低的特权；为避免模糊和混淆，在比较特权级别时，我们不使用大于或小于这样的术语，而使用术语“里面”或“内层”，来表示较高特权级，级别的数字较小；使用“外面”或“外层”来表示较低特权级，级别的数字较大。0 级为最内层的特权级，3 级为最外层的特权级，按这样的表示方法，四种特权级的层次关系可图示为图 4.2 的四个同心圆。

每个存储器段都与一个特权级别相联系。特权级别限制是指，只有足够级别的程序，才可对相应的段进行访问。在第 2 章曾说明：处理器是从 CS 寄存器为段地址寻址的段中，获取并执行指令的。当前特权级或 CPL，就是当前活动代码段的特权级，这个特权级也定义了当前正在执行的程序的特权级。CPL 确定哪些段可由程序进行访问。

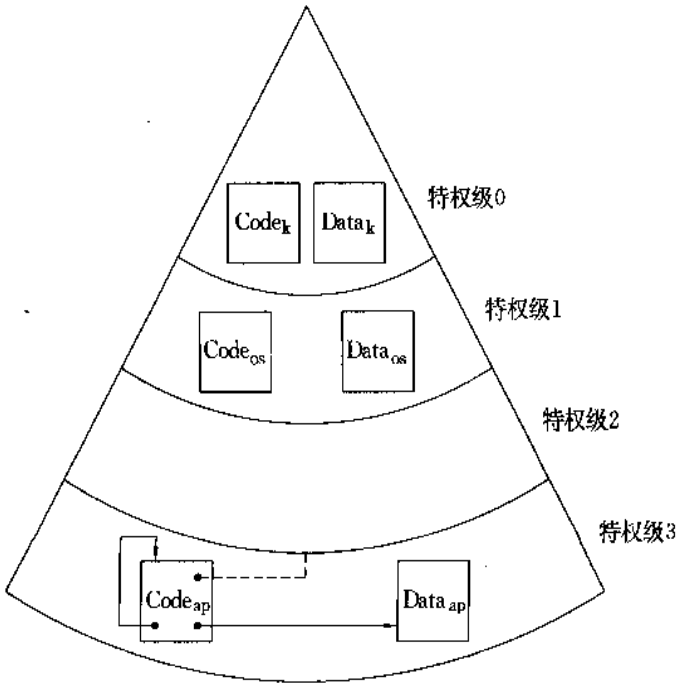
每当一个程序试图访问一个段时，就把当前特权级与要访问的段的特权级进行比较，以决

定是否允许这一访问。对给定CPL执行的程序，允许访问同一级别或外层级别的数据段。如试图访问内层级别的数据段则是非法的，并产生一个异常，向操作系统报告这一违反特权规则的操作。

第2章已说明，每一特权级有各自独立的程序堆栈，以避免与共享栈区有关的保护问题。当一个程序从一个特权级切换到另一个特权级执行时，程序使用的堆栈，从原特权级的栈段改变为新特权级的栈段。从一个特权级切换到另一特权级的方法，将在4.6控制转移方法一节中描述。

特权级的典型用法，是把操作系统的核心放在0级，操作系统的其余部分放在1级，而应用程序放在3级，留下2级供中间软件级使用。对特权级进行这样的安排，使得在0级的操作系统核心，有权访问任务中的所有存储段；在1级的、除操作系统核心外的操作系统其余部分，有权访问除0级以外的所有存储段；而3级的应用程序，只能访问程序本身的存储段，这些存储段也是在3级。这样的安排使得操作系统的核心得以保护而不被操作系统的其余部分所访问，也保护了整个操作系统，不允许应用程序对操作系统的存储段进行访问。

图4.3、4.4及图4.5图示出四个特权级、特权级的段的分配、以及基于当前正执行特权极CPL的段的访问规则。属于0级的操作系统核心，存储在称为$Code_k$的0级代码段，及称为$Data_k$的0级数据段。属于1级的操作系统其余各段，存储在$Code_{os}$段及$Data_{os}$段。属于3级的应用程序，存储在$Code_{ap}$段及$Data_{ap}$段。在这几个图中没有使用2级。当执行段$Code_{ap}$中的应用程序代码时，CPL为3，这时仅仅应用程序段$Code_{ap}$及$Data_{ap}$可被访问，如图4.3所示。

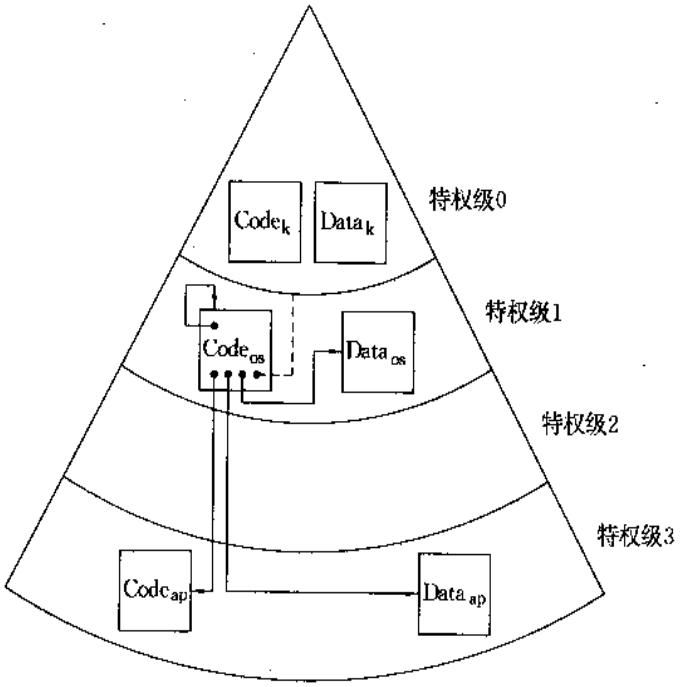


实线箭头指向的段可被访问，虚线表示不可访问

图4.3 特权级3的代码可访问的段

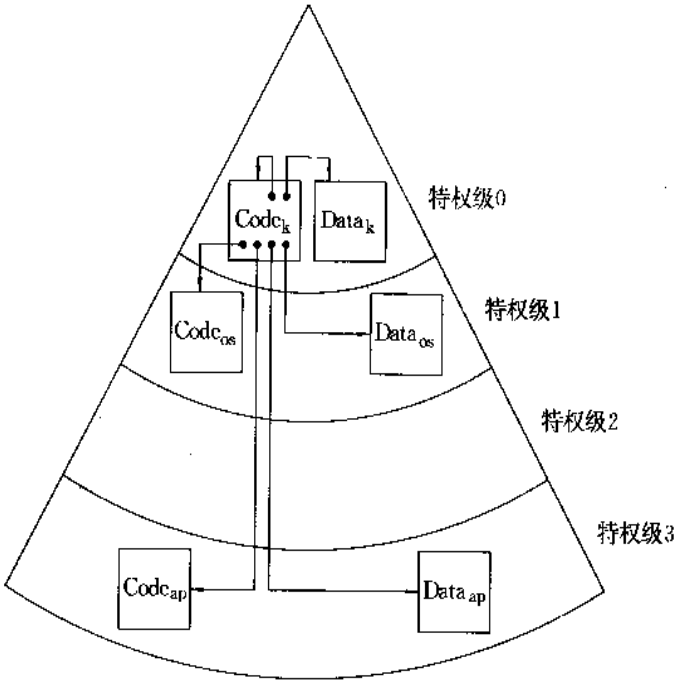
图4.4示出当程序在操作系统非核心部分执行时，CPL为1，可以访问自己拥有的段及应用程序的段。

图4.5示出当程序在操作系统核心执行时，CPL为0，可以访问所有级别的6种存储段。



实线箭头指向的段可被访问，虚线表示不可访问

图 4.4　特权级 1 的代码可访问的段



0级代码段 Code$_k$可以访问系统中所有的存储段

图 4.5　特权级 0 的代码可访问的段

4.1.4.3 组合保护级

前面两节描述了80386提供的保护的两个方面:任务之间的保护及任务之内的保护。本节描述这两个方面如何进行配合,以完成应用程序之间的保护,使所有应用程序共享操作系统,并保护操作系统使其不被任何应用程序所破坏。

每个应用程序被安排为一个独立的任务。与应用程序有关的代码及数据,存储在局部于该任务的3级的段中。操作系统存储在全局于所有任务的1级段中。其核心存储在全局的0级段中。

应用程序及操作系统的这种布局示于图4.6。图中,同心圆表示保护级别,而3级环中的径向虚线表示任务的边界。这样的安排使得操作系统及核心,由所有任务共享。这是因为操作系统及核心安排在全局地址空间中,在0级及1级中没有径向线而对所有任务起作用。因为应用程序在3级而操作系统在0级和1级,所以操作系统得到保护而不被应用程序破坏。虽然所有的应用程序都在3级,然而各个不同的应用程序,存储在不同任务中的不同虚拟地址空间中,因而提供了对应用程序之间的保护。

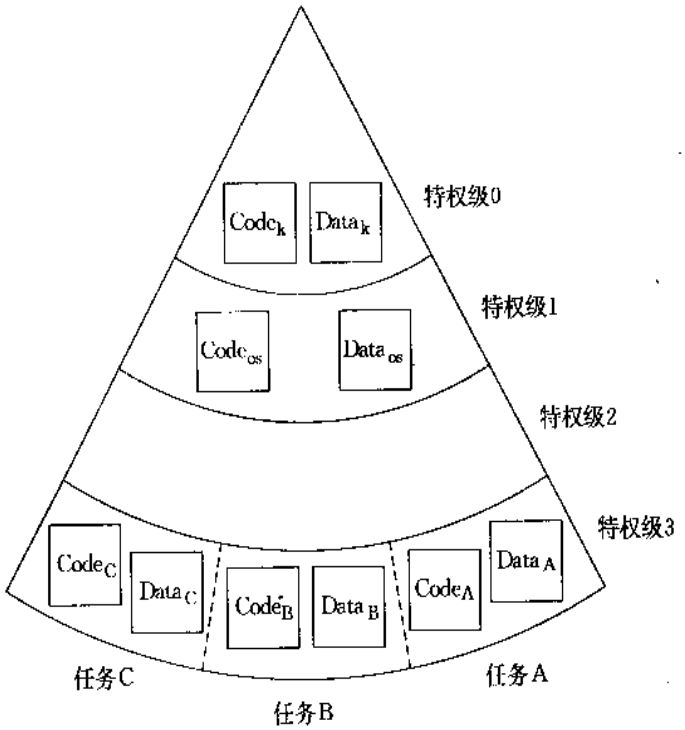


图4.6 任务间及任务内的保护

4.2 段 机 制

段机制把虚拟存储器组织成大小可变的容量单位的集合,称之为段。80386的段模式,使用具有两个部分的虚拟地址,即段部分及偏移量部分。第2章对这种分两个部分的虚拟地址已做过介绍,描述了所提供的段的寄存器,用以保存地址的段部分,并详细说明了可用来产生地址偏移量部分的寻址方式。本章描述段定义,及如何将虚拟地址转换为线性地址,而对虚拟地

址的段部分，进行详细的描述。在第3章描述了可把选择子装入到段寄存器中，使段成为可寻址的指令。本章将详细描述当段寄存器被装入时，如何使一个段成为可寻址的详细过程。

段是形成虚拟—线性地址转换机制的基础。每个段由三个参数进行定义，其中两个参数建立了由段基地址及段内偏移量形成的虚拟地址到线性地址之间的联系。这三个参数是：

(1) 段的基地址(Base Address)，规定线性地址空间中段的开始地址。基地址是线性地址对应于段内偏移量为0的虚拟地址。

(2) 段的界限(Limit)，表示在虚拟地址中，段内可以使用的最大偏移量，界限定义段的容量。

(3) 段的属性(Attributes)，表示段的特性。例如该段是否可被读出或写入，或者该段作为一个程序来执行，以及段的特权级等等。

段的界限定义虚拟地址空间中段的大小，基地址及界限定义段所映射的线性地址的范围。段内在偏移量从0到limit范围内的虚拟地址，对应于从Base到Base+Limit范围内的线性地址。在一个段内，偏移量大于段界限的虚拟地址将没有意义。使用这样的虚拟地址，系统将产生异常。另外，对一个段进行段的属性不允许的访问，也将产生异常。例如，在80386中，如果要在一个只读段中进行写入，80386将检测到这种违反该段规定访问属性的操作，并产生异常。

图4.7表示一个段如何从虚拟地址空间，重新定位到线性地址空间。图的左侧表示虚拟地址空间，定义了A、B及C三个段，段容量分别为$Limit_A$、$Limit_B$及$Limit_C$。虚拟—线性地址的转换，由段基地址$Base_A$、$Base_B$及$Base_C$进行定义。图中虚线把虚拟地址空间中的段A、B及C与线性地址空间区域连接起来表示这种转换，段A的线性地址空间区域为$Base_A$到$Base_A+Limit_A$。段B的线性地址空间区域为$Base_B$到$Base_B+Limit_B$。段C的线性地址空间区域为

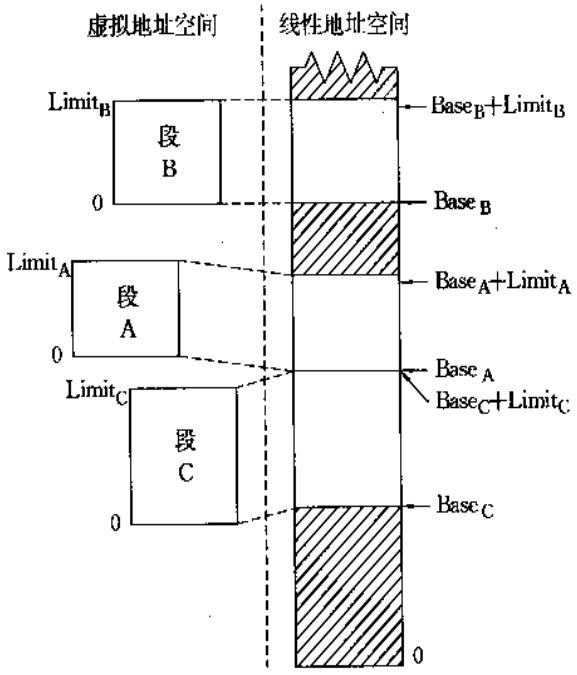


图4.7 虚拟—线性地址转换

$Base_C$ 到 $Base_C+limit_C$。图中可看到段 A 存储在紧接段 C 的上面，这表示 $Base_A=Base_C+Limit_C$，而段 A 的顶端与段 B 的开始有一段间隔，即表示 $Base_B>Base_A+Limit_A$。

每次对存储器访问要进行合法性检查，以保证合法地址的偏移量部分要在段的界限之内。例如在段 C 中，如有一个很大的偏移量(偏移量大于 $Limit_C$)，使该值进入了段 A 的范围，则不对段中的存储器进行寻址，而是产生一个异常，向操作系统报告段 C 产生的段界限的违反。这样的界限检查，使操作系统可以将段 A 分配在特权级 3 级，而段 C 分配在较内层的特权级，以保证一个在 3 级的程序，不能使用一个非法的在段 A 内的偏移量，去访问段 C 中的存储器。这样的检查和保护，对于那些线性地址空间彼此相邻的段也是一样。

段的基地址、界限及保护属性，存储在段的描述符中，在虚拟—线性地址转换过程中要对段描述符进行访问。段描述符又存储在存储器的段描述符表中，该描述符表是段描述符的一个数组。第 2 章介绍的段选择子，规定段描述符的位置，标识一个段。描述符表选择子及描述符，将在下面几节中做详细的介绍。

4.2.1 段描述符表

全局描述符表(GDT)及局部描述符表(LDT)，是包含段描述符表的两个特殊的段。描述符表存储在由操作系统维护、并由处理器中的存储管理硬件所访问的、特别的段中，这些段在受保护的存储器中存储，只能由操作系统软件进行访问，避免应用程序的软件修改地址转换信息。

虚拟地址空间分为相等的两半：一半由 GDT 进行映射，另一半由 LDT 进行映射。整个虚拟地址空间包含 2^{14}个段，GDT 映射的一半(即 2^{13}个段)是全局虚拟地址空间。LDT 映射的另一半(共 2^{13}个段)是局部虚拟地址空间。段描述符由描述符表(GDT 或 LDT)及描述符表内的描述符的序号进行定位。

当发生任务切换时，LDT 改变为新任务的 LDT，而 GDT 保持不变。因此，由 GDT 映射的那一半虚拟地址空间为系统中所有任务共用。但是，由 LDT 映射的那一半虚拟地址空间，则由任务的切换而改变。在系统中，所有任务共享的段用 GDT 进行映射，这样的段包括：含有操作系统的段及系统中所有任务的 LDT 段。这些 LDT 段可视为属于操作系统的数据。

LDT 段包含由各单个任务私有的描述符。几个任务可以共享一个公共的 LDT，在这种情况下，因为这些任务有相同的 LDT，因此相同的段集可由所有这些任务使用，而且所有这些任务共享同一 GDT。对于一个共享的段，两个任务也可在各自的 LDT 中，各有一个描述符与之对应。用这样的办法使段共享，即可以不必把段的描述符放在 GDT 中由所有的任务共享。但是，在这种情形下，这一被共享的段必须由操作系统进行特别的处理。这是因为，该段有两个描述符分别在两个不同的、必须一起进行更新的 LDT 中。

图 4.8 表示了在一个任务中的各个段如何能在 GDT 及 LDT 之间分开。图中共有 8 个段，分别存储两个应用程序(A 及 B)、操作系统及核心的段。系统中有两个任务，分别对应于两个应用程序，每个任务都有自己的 LDT。应用程序 A 在任务 A 中运行，任务 A 有映射 $Code_A$ 及 $Data_A$ 段的 LDT_A。$Code_A$ 及 $Data_A$ 分别包含任务 A 的代码及数据。类似地，应用程序 B 在任务 B 中运行，任务 B 有映射 $Code_B$ 及 $Data_B$ 段的 LDT_B。保存操作系统的两个段 $Code_{OS}$及 $Data_{OS}$，以及核心段 $Code_K$ 及 $Data_K$，由 GDT 进行映射。因此，这四个段可由两个任务共享。另外，LDT 段 LDT_A 及 LDT_B 也由 GDT 进行映射。

当任务 A 执行时，可访问的段集包括由 LDT_A 映射的 $Code_A$ 及 $Data_A$，加上由 GDT 映射

的操作系统段 $Code_{OS}$、$Data_{OS}$、$Code_K$ 及 $Data_K$。当任务 B 执行时，可访问的段集则改变为，由 LDT_B 映射的 $Code_B$ 及 $Data_B$ 加上由 GDT 映射的操作系统段及核心段。

本例图示出对虚拟地址空间如何进行划分，使系统中的各个任务通过使用每个任务各自的 LDT 而互相隔离开来。当任务 A 执行时，任务 B 的各段不是虚拟地址空间的一部分。所以，任务 A 没有任何办法去访问任务 B 中的存储器。类似地，当任务 B 执行时，任务 A 的段不能被任务 B 访问。LDT 的这种用法，使应用程序之间彼此隔离，满足了要求。

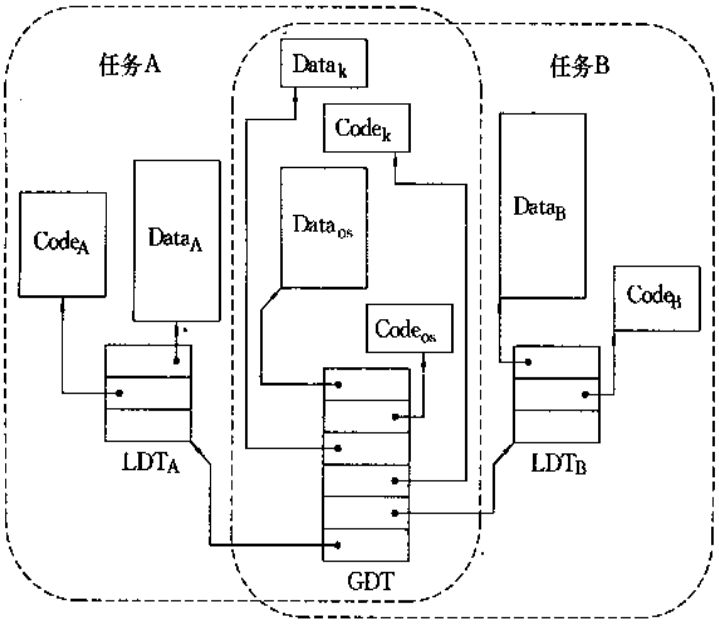


图 4.8 全局及局部地址空间

4.2.2 段选择子

在第 2 章介绍的段选择子，用作为虚拟地址两个部分中的段部分。段选择子标识一个段，也可把段选择子视为段的名字。如图 4.9 所示，段选择子长度为 16 个二进制位，其中包括 3 个二进制位构成的子字段。图中的 RPL(请求特权级)字段，占选择子的最低 2 位。RPL 在段保护模式中起着关键的作用。在位 2 中的 TI 位，指定包含段描述符的描述符表。TI=0 表示段描述符在 GDT 中；TI=1 则表示段描述符在 LDT 中。Index(索引)字段占选择子的高 13 位，该字段给出段描述符在 GDT 或 LDT 中的索引。

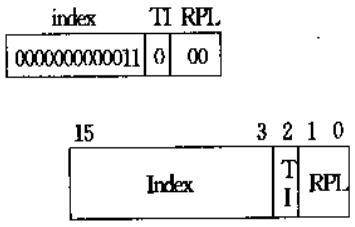


图 4.9 选择子格式

读者已经看到，一个选择子通过把段描述符定位到 GDT 或 LDT 中的某个位置而命名了一个段。TI 位确定包含段描述符的描述符表，Index 字段确定描述符在指定描述符表中的位置。Index 的值为描述符在描述符表中的偏移。Index=0 时，占用描述符表中的第 0 个表项，Index=k 时，占用描述符表中的第 k 个表项。描述符包含有访问一个段必须的全部信息，即包含段基地址、段的界限及段的属性等。我们可以把段选择子视为描述符的位置保持器或视为描

述符的间接指示器。

例 1 有一个描述符放在全局描述符表的第三个表项中，访问该描述符的请求特权级为 0，请写出访问该描述符的选择子。

- 描述符在全局描述符表中，所以 TI=0；
- 描述符在全局描述符表中的第 3 个表项，即索引为 3，所以 Index=3 或 11_b；
- 访问描述符的选择子的请求特权极为 0，即 RPL=0 或 00_b。

故所求选择子为

Index	TI	RPL
0000000000011	0	00

例 2 有一描述符的选择子，其内容为

Index	TI	RPL
1000011110000	1	11

，请解释其含义。

- TI=1 表示选择子访问的描述符在 LDT 即局部描述符表中；
- RPL=3 表示选择子的请求特权级为 3；
- Index=$1\times16^3+15\times16=4096+240=4336$，表示选择子所访问的描述符在描述符表中的 4336 表项中。

1. 空选择子

当选择子的 TI=0 而且 Index=0 时，我们把这样的选择子定义为空选择子。由于空选择子的 TI=0，故空选择子一定在 GDT 中，且占据 GDT 的第 0 个表项。在操作系统建立描述符表时，必须在 GDT 的第 0 个表项中建立一个假想的描述符，使之作为空选择子对应的描述符。由于该描述符永远不会被处理器访问，因此，事实上只需在 GDT 中把第 0 个表项留出即可。空选择子并不要求 RPL=0，所以空选选子具有如下形式：

Index	TI	RPL
0000000000000	0	RPL

关于空选择子有关的问题，将在 4.7.2 存储器数据访问一节中做更详细的讨论。

2. RPL 字段的用法

每当程序试图访问一个段时，要把当前特权级，与所访问段的特权级进行比较，以确定是否允许程序对该段的访问。使用选择子的 RPL 字段，将改变特权级的测试规则，在这种情况下，与之比较的特权级不是 CPL，而是 CPL 与 RPL 中处于更外层的特权级。

当带有由外层例程作为参数传入的选择子操作时，RPL 使操作系统减弱了 CPL 的作用。操作系统直接把所有选择子参数的 RPL 设置为传递选择子的程序的 CPL。这一点是通过使用第 3 章所描述的 ARPL 指令来完成的，然后，当操作系统使用选择子对段进行访问时，将使用主调程序的 CPL(存放在 RPL 中)，而不是操作系统的 CPL，来完成特权级的检查。这就保证了操作系统不以主调程序的名义对段进行访问(主调程序本身具有访问该段的权限除外。)

CPL 存放在 CS 寄存器的 RPL 字段内，每当一个代码段选择子装入到 CS 寄存器时，处理器自动地把 CPL 存储到 CS 的 RPL 字段。如果程序要对 CPL 的值进行检查，可以通过把 CS 选择子存储到通用寄存器或存储器的方法来实现。

4.2.3 段描述符

前面几节已经介绍了包含描述符的描述符表，以及用来确定描述符在描述符表中的位置

的段选择子。在本节我们要对描述符本身进行描述。在给出确切的段描述符的格式之前，先介绍段描述符的几个主要字段，即段基地址、段的界限、及段的属性。这些字段共占 8 个字节，示于 4.2.4 的存储段描述符格式一节的图 4.10 中。

4.2.3.1 基地址和界限

基地址(Base)规定段的起始线性地址，长 32 位。因为基地址长度与寻址地址的长度相同，所以任何一个段都可以从 32 位线性地址空间中的任何一个字节地址开始。

界限(Limit)为了保证段的长度可以覆盖整个线性地址空间，界限本身也应规定段的长度是 32 位。但是在描述符中只规定了 20 位作为界限使用。由于段的界限可以是以字节为单位或以 4K 字节为单位，在描述符属性位中有一位对此进行定义，并把该位称为粒度位，用 G 表示。所以，当粒度位规定界限以字节为单位时(即 G=0)，20 位的界限具有的范围为 1 个字节至 1M 字节，增量为 1 个字节；而当粒度位规定界限以 4K 字节为单位时(即 G=1)，20 位的界限具有的范围则为 4K 字节至 4G 字节，增量为 4K 字节。所以，当粒度位为 1 时，段的界限实际上就扩展成为 32 位。

例如，如果有一个段，在描述符中规定：

$$\text{Base} = 56781234$$
$$\text{Limit} = 0$$
$$G = 0$$

该描述符描述的段的长度只有一个字节，线性地址为 56781234。在此段内，只有偏移量等于 0 是合法的，利用此偏移量可以访问段内唯一的、地址为 56781234 中的字节。偏移量 0 以外的任何其它值都将引起异常。

当规定 4K 粒度的界限时，存放在描述符中的 20 位的界限所规定的范围用下面的公式进行计算：

$$\begin{aligned}\text{Segment} + \text{Limit} &= \text{Descriptor Limit} * 4\text{K} + 4095 \\ &= \text{Descriptor Limit} \ll 12 + \text{0FFFh}\end{aligned}$$

换句话说，描述符界限字段中的 20 位，给出段界限的高 20 位，段界限的低 12 位设置为 0FFFh，从而形成一个 32 位的段界限。由于段界限的最低 12 位固定设置为 0FFFh，所以，描述符中 20 位的界限值为 0 时所规定的段的实际界限为 4K，而当描述符中 20 位的界限值为 $2^{20}-1$(0FFFFFh)时，规定段的实际界限为 4G。因此，无论基地址为任何值，通过规定描述符的粒度为 1，及适当的界限值，可以使整个段覆盖全部 4G 的线性地址空间。

例 1 有一个存储段，描述符中有：

$$\text{Base} = 12345000\text{h}$$
$$\text{Limit} = 0$$
$$G = 1$$

该存储段描述的线性地址空间范围从线性地址 12345000h 开始，至 12345FFFh 结束，长度为 4K 字节。在此段中，合法偏移量的值为 0～0FFFh，分别对应于线性地址 12345000h～12345FFFh。

例 2 有一存储段，描述符中有

$$\text{Base} = 0$$
$$\text{Limit} = \text{0FFFFFh}$$
$$G = 1$$

该描述符描述的存储段覆盖了整个 4G 线性地址空间。根据段界限的计算式可算出描述符界限为 0FFFFF、G 为 1 时的段界限为：

$$0FFFFFh \ll 12 + 0FFFh = 0FFFFF * 1000h + 0FFFh = 0FFFFFFFFh$$

即段容量为 4G 个字节。因为基地址为 0，所以段内最大的地址为 4G－1，刚好在 4G 线性地址空间的顶部。

通过增加段的界限，可以使段的容量得到扩展。这对于那些要在内存中扩展容量的数据段是很有效的，但对于包含堆栈的段却没有什么帮助。回顾第 2 章的介绍，程序的堆栈存储在由 SS 段寄存器寻址的段中，ESP 寄存器保存有栈顶的偏移量，当有栈元压入堆栈时，堆栈在段内朝着偏移量较低的方向增长。由于程序堆栈向下增长，所以堆栈分配的原则通常是从段顶(即段的界限)开始，向下方朝向段内偏移量为 0 的方向增长。向下扩展的段为堆栈提供以自然的方向扩展的支持。

向下扩展的段的界限是反向的。当把段的界限视为 4G 时，在自然的向上扩展的段内，界限字段把偏移量分为两个子范围：从 0 到 Limit 的值是合法的偏移量，而从 Limit＋1 至 4G－1 则是非法的偏移量。而在向下扩展的段内，合法与非法的解释正好要颠倒过来，偏移量从 0 至 Limit 是非法偏移量，从 Limit＋1 到 4G－1 则是合法偏移量。

段的扩展方向是段的又一属性，这一属性也存放于段描述符的属性字段内，访问并使用该属性，可以决定如何使用界限字段，以检查界限的合法性。

4.2.3.2 段属性

在各种类型的段描述符中，段的特权级存放于属性字段内，并命名了描述符特权级，用 DPL 表示。该字段占 2 位，可表示四种特权级，用数字 0～3 表示。

存在位(或称 P 位)这是所有类型的段描述符中共有的第二个属性。如描述符的 P 位被置位，则表示在虚拟地址转换为物理地址过程中，使用该描述符是有效的；如 P 位未置位，则表示描述符是无效的，不能用于虚拟—物理地址的转换。若一定要使用这样的描述符进行地址转换，则操作系统将报告一个异常。

另一个所有描述符共有的属性位是描述符类型位(或 DT 位)，该位用于区别描述符的两种类型：如 DT 位为 1，表示描述符指示的是一个存储段；如 DT 位为 0，则描述符指示的是一个系统段或是一个门。LDT 是一种系统段，前面已经做过介绍，门则在本章稍后 4.2.5 节进行描述。

下面一节将描述存储段、系统段及门的描述符的编码。虽然存储段及系统段的格式非常类似，但它们各有不同的属性，因而在不同节内进行讨论。门描述符的格式则与存储段或系统段描述符的格式相去甚远，以下我们就一一加以介绍。

4.2.4 存储段描述符格式

存储段是存放可由程序直接进行访问的代码及数据的段。存储段描述符的格式见图4.10。上面一排是描述符 8 个字节的表示图，最低地址字节(假定地址为 m)在最右边，其余字节依次向左，直到最高字节，地址为 m＋7。描述符的属性(Attributes)字段安排在字节偏移量为 5 和 6 的字节，即地址为 m＋5、m＋6 的字节。在图 4.10 下面一排的展开图中，展示了该属性字段的位偏移及字节偏移。DT 位安排在字节 m＋5 中的第 5 位，即位 4。如 DT＝1，表示该描述符所描述的段为存储段，在图 4.10 中用 DT＝1 表示；如 DT＝0，则表示该段为系统段或门。

描述符的基地址(Base)分离为两片。基地址的位 23～位 0 安排在 m＋2 字节开始的连续

三个字节中，而位 31～位 24 放在 m+7 字节中。界限(Limit)字段也分离为两个部份，位 15～位 0 放在 m 及 m+1 两个字节中，而位 19～位 16 放在 m+6 字节的低四位中。段的属性被安排在 m+5 及 m+6 字节，以下就对属性字段内的各种属性分别予以讨论。

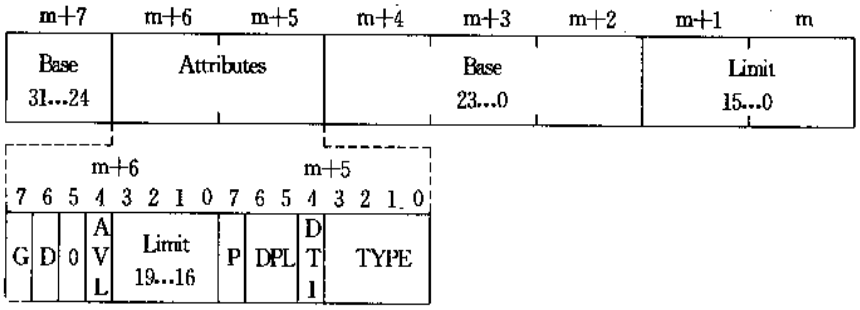


图 4.10 存储段描述符格式

G G 位是界限粒度属性位。G=0 表示界限粒度为字节，G=1 表示界限粒度为 4K 字节。应注意的是 G 位只对段界限的粒度起作用，而对段的基地址没有影响。即段基地址总以字节为粒度。

D 对 80386 软件，D 位总应为 1。在第 7 章将会讨论，80386 软件应置 D 位为 0。该位对可执行段、向下扩展段、或者由 SS 寄存器寻址的段，给出相应的默认值，具体规定如下：

① 可执行段使用 D 位，为段中的指令访问的地址或操作数设置默认值。D=1 表示指令使用的默认值是 32 位地址及 32 位或 8 位操作数，这是 80386 的正常使用情况；D=0 表示使用的默认值是 16 位的地址及 16 位或 8 位的操作数，与 286 系统兼容。使用指令前缀，可把指令的地址或操作数的长度改变为与默认值相反的值。

② 向下扩展的段使用 D 位，决定段的上部边界。D=1 表示段的上部界限为 4G；D=0 表示段的上部界限为 64K，这是为了与 286 兼容。

③ 由 SS 寄存器寻址的段，使用 D 位以确定隐式的堆栈访问指令(如 PUSH 及 POP 指令)要使用 32 位的 ESP 寄存器(此时 D=1)、还是使用 16 位的 SP 寄存器(此时 D=0)。后者是考虑与 286 兼容。

AVL AVL 位是软件可利用位。80386 对该位的使用未做规定，Intel 公司也保证今后开发生产的处理器只要与 80386 兼容，也不会对该位的使用做任何定义或规定。

P P 位已进行过讨论，即 P=1 表示描述符对地址的转换是有效的，P=0 表示描述符无效，且使用该描述符会引起异常。

DPL 描述符特权级，共 2 位，用于定义与段相联系的特权级。

DT 描述符类型位。区分描述符所指的段是存储段(DT=1)，还是系统段及门(DT=0)。

Type 类型字段。4 位长，定义存储段描述符的类型，对应代码示于表 4.1。表 4.1 中的读、写及执行属性不需做进一步的解释，向下扩展界限属性已刚刚讨论过，而代码的一致性属性在稍后的节中要做讨论。当把描述符的相应选择子装入到段寄存器时，表明描述符已由 80386 访问，并标记为“已访问”。“已访问”属性可以由操作系统软件进行测试，确定自上次由软件将该位清 0 后，该描述符是否已由处理器再次进行过访问。

表 4.1 存储段描述符类型

类　型	说　明	类　型	说　明
0	只读	8	只执行
1	只读、已访问	9	只执行、已访问
2	读/写	10	执行/读
3	读/写、已访问	11	执行/读、已访问
4	只读、向下扩展	12	只执行、一致码段
5	只读、向下扩展、已访问	13	只执行、一致码段、已访问
6	读/写、向下扩展	14	执行/读、一致码段
7	读/写、向下扩展、已访问	15	执行/读、一致码段、已访问

4.2.5 系统段及门

系统段是 80386 段机制使用的一种特别的段。门不是用来描述一个段，而是包含一个指针。系统段及门在本章稍后将详细描述，作为参考信息，这里先定义它们的描述符格式。

图 4.11 示出的是系统段描述符的格式。基地址和界限字段与存储段描述符完全相同。属性中的 G 位，AVL 位，P 位，DPL 字段也完全相同。类型字段仍为 4 位，但编码及含义则完全不一样。存储段的 D 位在这里不用，图 4.11 中以 X 表示。DT 段为 0，(标为 DT0)，以区别于存储段描述符类型，表示该选择子对应的是系统段或门描述符。

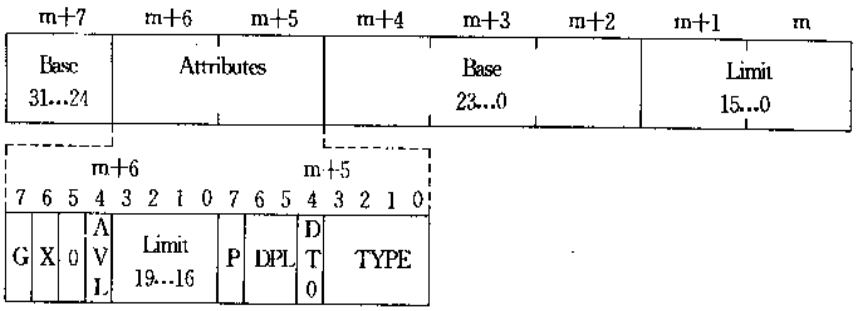


图 4.11 系统段描述符格式

与存储段描述符一样，在 m+5 字节包含有一个四位的类型字段，该字段确定描述符的类型，即描述符是系统段描述符还是门描述符。在描述符类型字段中，系统段及各种门描述符的编码示于表 4.2。前面已介绍过 LDT 系统段类型。其它的类型如“忙”的及“可用”的 386TSS 系

表 4.2 系统段和门的类型字段的编码

类型编码	定义	类型编码	定义
0	未定义	8	未定义
1	可用 286TSS	9	可用 386TSS
2	LDT	10	未定义
3	忙的 286TSS	11	忙的 386TSS
4	286 调用门	12	386 调用门
5	任务门	13	未定义
6	286 中断门	14	386 中断门
7	286 陷阱门	15	386 陷阱门

统段类型、386 调用门类型及任务门类型等在本章进行描述。而 386 陷阱门及中断门类型将在第 5 章进行描述。286TSS 系统段类型及 286 的各种门类型将在第 7 章讨论。286 的各种系统段与 386 的相应系统段相对应，只是位数是 16 位的。

图 4.12 示出门描述符的格式，门中包含 48 位的全指针及 16 位的属性。48 位的全指针，具有与第 2 章介绍的数据类型全指针相同的信息。在图 4.12 中，指针的选择子(Selector)存放在 m+2，m+3 字节，指针的偏移量分两个部分存放：低 16 位存放在 m 字节及 m+1 字节，高 16 位存放在 m+6 及 m+7 字节。在门描述符中，属性字段存放在 m+4 及 m+5 字节，具体安排如下：

P 存在位。P=1 表示门存在，即门有效；P=0 表示门不存在，即门无效。使用无效门将产生异常。

DPL 描述符特权级。定义门的特权级。

DT 描述符类型位。区分存储段描述符和系统段及门描述符，这里叙述的是系统段及门，故 DT 位为 0，图中以 DT0 表示。

Type 门的类型。用四位编码定义，详见表 4.2。

DwordCount 双字计数。在通过调用门调用另一过程时，通常要传递一些参数到被调用的过程，这些参数一般放在堆栈内。如果由于门的使用引起特权级的转换及堆栈的改变，则需将一个堆栈的参数复制到另一个堆栈。Dword Count 字段给出的值，即是要复制的双字参数的数量。下面就要说明，用这种方法可以把过程参数、通过程序堆栈在程序之间进行传递。此字段只对调用门有效。

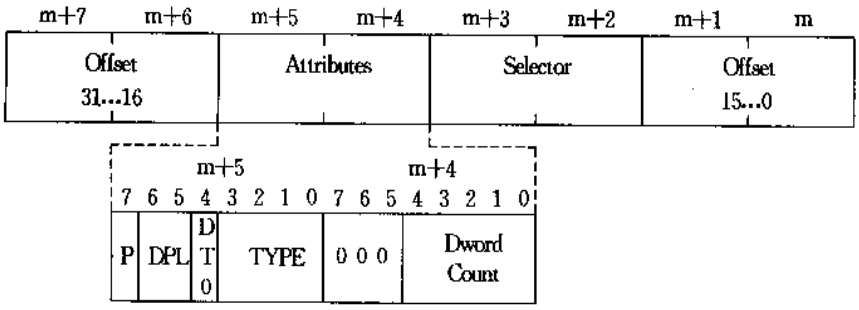


图 4.12 门描述符格式

4.2.6 描述符投影寄存器

为避免在每次存储器访问时，都要访问描述符表，读出描述符并对段描述符进行译码以得到描述符本身的各种信息，每个段寄存器都有一组与之相联系的描述符投影寄存器。在这些寄存器中，容纳有由段寄存器中的选择子确定的段的描述符信息。段寄存器对编程人员是可见的，这些段寄存器或者是第2章定义的六个段寄存器之一，或者是本章将要描述的寄存器之一。而与之相联系的容纳描述符的寄存器，则对编程人员是不可见的，故称之为投影寄存器。图 4.13中所示的是六个段寄存器及其投影寄存器。用实线画出的寄存器是段寄存器，用以表示这些寄存器对编程人员可见；用虚线画出的寄存器是投影寄存器，表示对编程人员不可见。

投影寄存器容纳有由相应段寄存器寻址的段的基地址、界限及属性。每当用选择子装入段寄存器时，CPU 硬件便自动地把描述符的全部信息装入对应的投影寄存器。因此，在多次访问

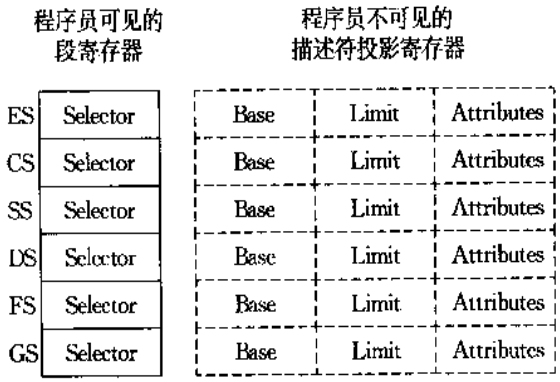


图 4.13 描述符投影寄存器

同一段时，就可以用投影寄存器中的基地址来访问存储器。投影寄存器存储在80386的芯片上，因而可以由段机制硬件进行快速访问。因为多数指令访问的数据是在其选择子已经装入到段寄存器之后进行的，所以使用投影寄存器可以得到很好的执行性能。

第3章中多段类指令一节所描述的每条指令，既把一个新的选择子装入到段寄存器中，也把相应的描述符信息装入到投影寄存器中。第3章把这些指令描述成编程人员可以见到的指令，即描述成把选择子装入到段寄存器的指令。本章将把此定义扩展到对应的描述符如何由这些指令进行检查的细节，以及在所有的检查都通过时，如何将描述符装入到投影寄存器的细节。如果进行的各种检查中有一种检查失效，则产生异常。这时任何选择子寄存器或投影寄存器都不发生变化。

投影寄存器包含有描述符信息的拷贝，因此操作系统必须注意，保证描述符表的改变要反映在投影寄存器中。否则一个段可能已经改变了描述符表中的段的基地址或界限，但投影寄存器中未做改变而保存着原来的值。进行这种处理的一种方便的方法，是在改变了描述符表中任意一个描述符之后，重新装入所有六个段寄存器，从而使描述符表中的最新信息重新装入到投影寄存器中。

4.3 分页机制

分页机制是存储管理机制的第二部分。分页机制在段机制之后进行操作，以完成虚拟一物理地址转换过程。段机制把虚拟地址转换为线性地址，分页机制进一步把该线性地址再转换为物理地址。

分页机制由 CR0中的 PG 位启用。如 PG=1，启用分页机制，并使用本节要描述的机制，把线性地址转换为物理地址。如 PG=0，禁用分页机制，直接把段机制产生的线性地址当做物理地址使用。

分页机制与段机制不一样，段机制对应和管理的是大小可变的存储器块，该存储器块的大小可以是若干个 K 字节，或若干兆字节甚至若干千兆字节。而分页机制管理的对象则是固定大小的存储器块，称之为页。分页机制把整个线性地址空间及整个物理地址空间都看成由页组成。在线性地址空间中的任何一页，可以映射为物理地址空间中的任何一页。图4.14示出分页机制如何把线性地址空间及物理地址空间划分为页，以及如何在这两个地址空间进行映射。图

4.14的左边是线性地址空间，并将其视为一个具有一页大小的固定块的序列。图4.14的右边是物理地址空间，也将其视为一个在页的序列。图中，用箭头把线性地址空间中的页，与对应的物理地址空间中的页联系起来。在这里，线性地址空间中的页，与物理地址空间中的页是随意地对应起来的。

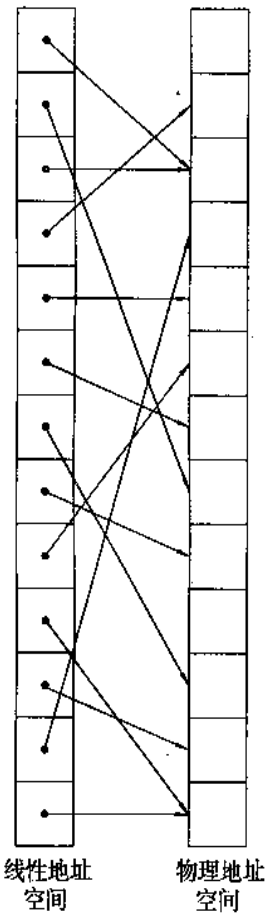


图 4.14 分页机制把线性地址转换为物理地址

80386使用4K 字节大小的页。每一页都有4K 字节长，并在4K 字节的边界上对齐，即每一页的开始地址都能被4K 整除。因此80386的分页机制，把4G 字节(2^{32})的线性地址空间，划分成2^{20}个页面，每页有2^{12}(即4K)字节大小。分页机制通过把线性地址空间中的页，重新定位到物理地址空间来进行管理。因为每个页面的整个4K 字节作为一个单位进行映射，并且每个页面都对齐在4K 字节的边界，因此，线性地址的低12位经过分页机制直接地作为物理地址的低12位使用。重定位函数也因此可看成是把线性地址的高20位，转换为对应物理地址高20位的转换函数。

线性—物理地址转换函数，可将其意义扩展为允许将一个线性地址标记为无效，而不是实际地产生一个物理地址。有两种情况可能使页被标记为无效：其一是线性地址是操作系统不支持的地址；其二是在虚拟存储系统中，线性地址对应的页存储在磁盘上，而不是存储在物理存储器中。在前一种情形下，程序因产生了无效地址而必须被终止。对于后一种情形，该无效地址实际上是请求操作系统的虚拟存储管理系统，把存放在磁盘上的页传送到物理存储器中，使该页能被程序所访问。由于无效页通常是与虚拟存储系统相联系的，这样的无效页通常称为未驻留页，并且用页表属性位中叫做存在位的属性位进行标识。未驻留页是程序可进行访问的页，但它不在主存储器中。对这样的页进行访问，形式上是发生了异常，实际上是通过异常进行缺页处理。

4.3.1 页表结构

分页转换函数，由一个称为页表的存储器常驻表描述，页表总是存储在物理地址空间中。我们可以把页表看成一个简单的2^{20}个物理地址数组，线性—物理地址映射函数就是一个简单的数组查询。线性地址的高20位形成对数组进行查询的索引，从而选择出相应页的物理地址。线性地址的低12位则给出所选择出页的页内偏移，将此偏移加到页的基地址上，即得到了最终的物理地址。由于页的基地址是4K 边界对齐的，所以页基地址的低12位为全0，这就表示低12位的偏移量并不需要真正地与页基地址进行相加操作，而只是简单地将其连接到页基地址的高20位即可。也就是说，页基地址提供物理地址的高20位，而偏移量提供物理地址的低12位。

每个页表项为32位长，由于32位的页表项中只有20位需用来存储物理地址，剩下的12位则用来表示页的属性，例如页是否存在，页的特权级等等。如果由线性地址经索引得到的页表项属性位中被标记为存在，则该页表项是有效的，并且页表项对应页的物理地址可从页表项中的

高20位字段中得到。如果页表项的属性位表示不存在，则向操作系统软件报告一个无效寻址的页异常。

4.3.1.1 两级页表结构

在80386中页表共含2^{20}个表项，每个表项占4个字节。如果把所有的页表项存储在一个表中，则该表将占4M 字节连续的物理存储空间。为避免使页表占有如此巨额的物理存储器资源，故对页表采用了两级表的结构。而且对线性地址的高20位的线性—物理地址转换也分为两步完成，每一步各使用其中的10位。

两级表结构的第一级称为页目录表，存储在一个4K 字节的页中。页目录表共有2^{10}(1K)表项，每个表项为4个字节长，并指向第二级表。线性地址的最高10位(即位31～位22)用来产生第一级表的索引，由索引得到的表项中，指定并选择了2^{10}个二级表中的一个表。

两级表结构的第二级称为页表，也刚好存储在一个4K 字节的页中，包含1K 个4字节的表项，每个表项包含一个页的物理基地址。第二级页表由线性地址的中间10位(即位21～位12)进行索引，以获得包含页的物理基地址的页表项，这个物理地址的高20位与线性地址的低12位形成了最后的物理地址，也就是页转换过程输出的物理地址。

图4.15示出了上述两级表结构的查找过程。寄存器 CR3(见4.4.1)通过指向页目录表而确定了页表结构的根。线性地址的高10位就用于索引此目录得到指向适当的二级页表的指针。线性地址中间10位用于索引页表，得到物理地址的高20位结果。线性地址的低12位不加改变的通过分页机制，并直接地和从页表得到的高20位物理地址连在一起形成了完整的物理地址。

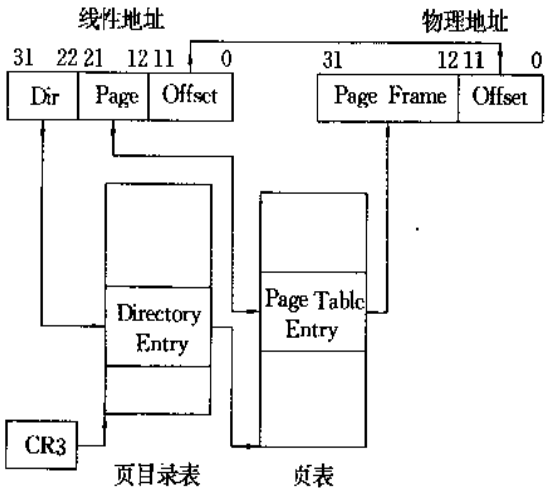


图 4.15 两级页表结构

4.3.1.2 不存在的页表

通过使用两级表结构，我们并没有解决前述需要4M 字节存储空间存储页表的问题。事实上，使用两级表结构反而使存储问题更严重了一点。因为除了需要4M 空间存储页表以外，还需另外增加一页用以存放页目录表。然而两极结构允许把页表分散存放在遍及存储器任意位置的页中，而不要求存放在一个连续的4M 字节的存储区域中。此外，两级表结构中对于线性地址空间中不存在的或未使用的部分不必分配页表。页目录表虽然必须驻留于物理存储器中，但二级表仅当在需要时才进行分配，于是页表结构的大小就对应于实际使用的线性地址空间

的容量。因为任何一个实际运行的程序使用的线性地址空间，都远小于4G，所以需要分配页表结构的大小也就远小于4M 字节。

页目录表中的每个表项有一个存在属性位，类似于页表项中的存在属性。在目录项中的存在属性，表明对应第二级表在页转换中是否可利用。如果目录项表明二级表存在，同通过访问二级表继续进行二级表的查找。如果存在位表明二级表不存在，则产生页异常，向操作系统报告这一非法线性地址的使用。目录项中的存在属性，使得操作系统只需分配覆盖实际使用的线性地址范围需要的二级表。

目录项中的存在属性也可用于把第二级页表存储在虚拟存储器中。这意味着在任何时刻，只有第二级页表的一个子集在物理存储器中，第二级表的其余部分，则可存储在磁盘上。在物理存储器中的页表的目录项，属性位标志为存在，表明这些目录项在页转换时是有效的；对存储在磁盘上的页表的目录项，则标志为不存在，表明这样的目录项在地址转换时无效。由于访问不存在的第二级页表产生的异常，将发送信号给操作系统，使其将物理存储器中缺少的页表从磁盘上读入。把页表存储在虚拟存储器，可以最大限度的减少存储分页机制转换表所需的物理存储器的总数量。

4.3.1.3 全局页表与局部页表

与段表结构不一样，对于页表，没有将其分解为全局表和局部表的规定。然而，通过使每一个任务共享线性—物理地址映射函数的一部分，可以定义线性地址空间的全局部分。由于在每一个任务中使用同样的线性—物理映射信息，因此，线性地址空间的这一部分，在系统的每一个任务中都进行同样的映射。这种映射使得这些页在所有任务中，都映射到同样的物理地址，因而线性地址空间的这一部分中的页被称为全局页。线性地址空间中余下的部分，局部于每一不同任务。通过建立页表结构，使得不同的地址映射函数，用于线性地址空间的局部范围。在线性地址局部部分中的页，称为局部页。

两级页表结构，通过共享第二级页表及全局页，支持部分线性地址空间的有效共享。图4.16表示这一技术。图中，页表结构由 A、B 两个任务使用，DIR_A 是任务 A 的目录页，DIR_B 是任务 B 的目录页，每个目录的第一表项(在偏移量0的位置)指向一个可以映射最多为2^{10}(1K)个全局页的全局页表。每个目录的第二个表项指向分别由各个任务使用的局部页表，每个这样的页表映射可达2^{10}个局部页。

在这一例子中，有至少4M 字节的线性地址空间全局于两个任务，并且各有另外的4M 字节局部于每一任务。另外的页表，可以按需要分配到全局区域，提供4M 字节块的线性地址全局部分的扩展。要注意的是，共享的页表，由两个页目录中同样的目录项所指向。这一点很重要，因为它保证了在两个任务中同样的线性地址范围，将映射到该全局区域。本例中，第一个4M 字节的线性地址空间(地址为0—3FFFFFh)在全局线性地址空间中。

共享映射线性地址空间全局部分的第二级表有几个优点：第一，只需要分配一组第二级页表映射全局页，这些页表由所有任务共享，对每一个任务不需要拷贝；第二，如果共享页的状态改变，只需要更新一个页表项，反映这一状态的改变；第三，如果一个全局页被换出到虚拟存储系统磁盘中或换入到内存，只需对一个页表项进行改变。

与段一样，操作系统的代码、数据及段描述符表，应存储在全局表中。而仅属于一个任务的代码和数据应存储在局部页中。段表及页表的建立，最好使得由 GDT 映射的段存储在全局页中，由 LDT 映射的段存储在局部于任务的页中。

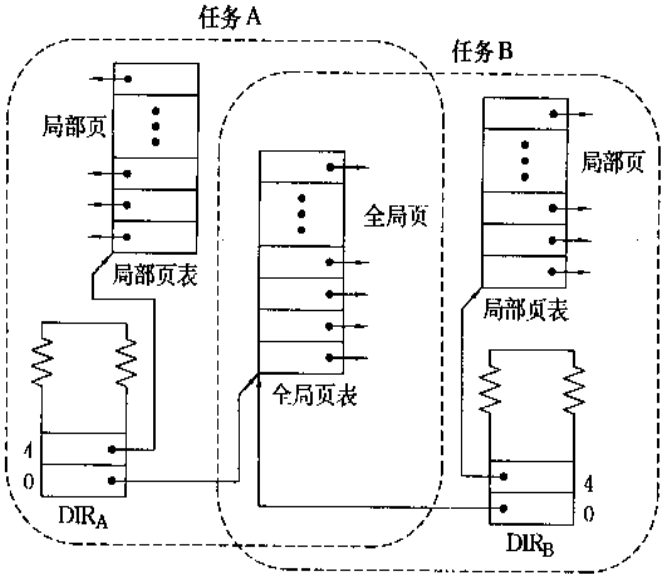


图 4.16 全局及局部页表

4.3.2 页表项格式

在页目录及页表中的表项使用如图4.17中的格式。位31—位12包含物理地址的高20位，表示物理地址空间中的“页框”。低12位包含页的属性。前面已讨论过存在位，在此简略描述一下其余属性位(以后将做详细讨论)。所示属性中内容为0的位是 Intel 公司为今后开发的处理器留下的保留位，在使用中这些位必须设置为0，以保证向上兼容性。

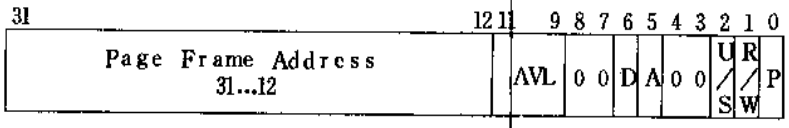


图 4.17 页目录表/页表的表项格式

P 位0为存在位，该位表示表项对地址转换有效(P=1)，或无效(P=0)。在页转换期间，无论在目录表或页表中遇到无效表项，都会产生异常。如果 P=0，表项中其余各位可供软件使用，如图4.18所示。80386不解释 P=0的表项中任何其它的位。

R/W 位1为 R/W 位，即读/写位。如该位为1，对页表指定的页可进行读、写或执行；如该位为0，该页可读或执行，但不能对该页进行写操作。然而读写位并不总是起作用的。当处理器正在超级特权之一(即0，1或2级)执行时，R/W 位被忽略。在目录表项中的 R/W 位，应用于该目录表项映射的所有各页。

U/S 位2为 U/S 位，即用户/系统位。如该位为1，页表指定的页可由在任何特权级(包括用户级即3级)下执行的程序访问；如该位为0，则该页只能由在超级特权级(即0、1或2级)执行的程序访问。在目录表项中的 U/S 位，应用于该目录表项映射的所有各页。

A 位5为 A 位，即访问位。页表项中的 A 位在对页表项映射的页进行任何访问之前，由处

理器设置为1。目录表项中的 A 位，在对目录表项映射的各页中的任何一页进行任何访问之前，由处理器设置为1。处理器永不清除 A 位，但操作系统软件为获取页使用情况的统计信息而周期地清除 A 位。

D 位6为 D 位，即已写标志位。第二级页表项中的 D 位，由处理器对页表项映射的页进行写访问之前将其设置为1。处理器不修改页目录中表项的 D 位。

AVL AVL 字段在位9～位11，供软件使用。处理器不能对其进行修改。

图4.18 不存在的页目录表/页表的表项

为了便于理解，下面举例说明线性地址如何通过两级页表转换为物理地址。

例 请将线性地址12345678h 转换为物理地址，假定 CR3中的值为8000h。

首先，将32位线性地址12345678h 分解为三部分：

31	2221	1211 0
0001001000	1101000101	011001111000

位31～位22用于目录表的索引，位21～位12用于所选目录表项指向的第二级页表的索引，位11—位0给出所选页表项指向的页中的页内字节偏移。

在目录表中，每个表项占四个字节，因此，线性地址中目录索引值需乘4，得到目录表的字节偏移。现在线性地址索引值为0001001000b＝48h，乘以4即左移2位后得到48h×4＝120h，这就是加到包含在 CR3中目录页的物理基地址上的偏移。已假定 CR3中的值为8000h，所以目录项由物理地址8000h＋120h＝8120h 中读出。

页表由目录项的高20位进行寻址，假定在8120h 地址中的目录项的值为00010021h，即页表的基地址为00010000，P 位(位0)及 A 位(位5)都是1，这表示第二级页表存在，而且目录项已被访问过。页表项的地址通过把线性地址的位21～位12作为索引加到在地址为00010000h 的页表上进行寻址得到。同样，由于每个页表项也占四字节，所以索引值也应乘以4以得到偏移，即偏移值＝345h×4＝D14h，因此，页表项从物理地址00010000h＋0D14h＝00010D14h 中读出。假定00010D14h 中的内容为54321021h，即页框地址为54321000h，属性字段为021，即页存在且可访问。

对应于原始线性地址12345678h 的物理地址通过把线性地址的低12位即678h 与页表中的页框地址相加得到，即：

页框地址	54321000h
页偏移	678h
物理地址	54321678h

现在，我们把转换过程再小结一下。在本例中，线性地址12345678h，通过查找两级页表而转换成了物理地址。寄存器 CR3指向物理基地址在8000h 处的页目录，目录项从地址8120h 读

出，并指向物理地址在10000h 处的页表，页表项从地址10D14h 中读出，再指向物理地址在54321000h 处的页框，最后的物理地址，由页框基地址与线性地址的页偏移相加，得到54321678h。

4.3.3 虚拟存储页的驻留及修改

P 位提供关键的属性，以支持具有分页机制的虚拟存储器。物理存储器中存在的虚拟地址空间中的页，被标志为存在(P=1)，在页表项中可以得到对应的物理地址。在物理存储器中不存在的页，被标志为不存在(P=0)。如果程序访问不存在的页，则产生页异常，使得操作系统把缺少的页从磁盘上读入，并将读入页所在物理地址存入到表项中。然后将该页标志为存在，再使引起异常的程序恢复运行。

A 位和 D 位协助虚拟存储器的有效执行。通过周期地检测及清除所有的 A 位，操作系统即可确定哪些页在最近一段时间未被访问。这些最近未被访问的页很可能就被选择出来，当存储器资源短缺时，将它们换出到磁盘存储器上去。当一个页从磁盘上读入时，D 位被设置为0。如果某页需从内存传送到磁盘上，若该页的 D 位仍然为0，则此时并不需要实际的磁盘写操作，而是简单地放弃内存中的页即可，因为内存中的页与磁盘上的页，具有完全相同的内容。然而如果要写到磁盘上的页的 D 位为1，则必须进行写入操作。因为 D=1表明，内存中的页自上次从磁盘读入内存后，已至少有过一次写入，页中的内容一般来说已经被破坏。即使在某些特殊情况下，虽然发生了一次甚至多次写入，页中的内容却碰巧并未被改变，例如页中某地址原内容为 A，执行程序过程中第一次把 A 改写为 B，以后又从 B 改为 A，然后不再对该页进行写入，由于发生过两次写入，所以页表项的 D 位已被置1，在需将此页换出到磁盘上时，虽然内容实际上未被改变，但仍要进行写磁盘操作。

对于虚拟存储系统另外一个重要的考虑，在页表结构中是不可见的。所有的指令，在发生页异常后都是可重新执行的。一旦发生异常的原因被消除，例如，执行某条指令所需的数据在磁盘上而未在内存，通过从磁盘上读入不存在的页，并将其标志为存在后，指令所需的数据已可读到，此时，引起异常的指令即可以直接地从页异常中断处理程序返回后重新执行。报告异常及从异常返回的机制在第5章介绍。页异常的原因及页异常处理程序可利用的信息，本章还将进一步说明。

4.3.4 页级保护

R/W 及 U/S 位，提供由段机制支持的保护属性的子集。这些页级保护属性概括于表4.3。分页机制只区分两种特权级，特权级0，1和2合在一起构成超级特权级；特权级3称为用户特权级。用户级的页可以规定为只允许读/执行或规定为读/写/执行。超级特权级的页对于超级用户总是可读/写/执行的，而对用户则不允许访问。与段的机制一样，外层用户级执行的程序只能访问用户级的页，而在超级用户级(0，1和2)执行的程序，即可访问用户级的页，也可访问超级特权级的页。与段机制的区别则是，在内层超级用户级执行的程序，对任何页都有读/写/执行访问权，即使用户级规定为只允许读/执行的页，内层程序也有对该页的写访问权。因而使得用户级规定为只允许读/执行的页，仍允许超级用户进行读/写/执行访问。

表 4.3 页级保护属性

U/S	R/W	用户访问权限	系统访问权限
0	0	/	读/写/执行
0	1	/	读/写/执行
1	0	读/执行	读/写/执行
1	1	读/写/执行	读/写/执行

正如在整个80386地址转换机制中分页机制在段机制后起作用一样，页级保护也在由段提供的保护之后起作用。首先，测试检查所有的段级保护，如果保护检查通过，则测试检查页级保护。例如，一个存储器的字节，仅当该字节存储在3级可访问的段内、并存储在标志为用户级的页内时，才可由在3级执行的程序访问。一个存储器的字节，仅当段及页都允许写入时，对该字节才是可写的。如果段的类型为读/写，而页规定为只允许读/执行，则不允许写访问。如果段的类型为只允许读/执行，则不管页保护如何规定，也不允许进行写访问。

页的保护属性用页目录项及页级属性的组合进行计算。页表项中的 U/S 及 R/W 位应用于该页表项映射的一个页。页目录项中的 U/S 及 R/W 位则应用于由该页目录项映射的所有1K 页。表4.4定义在两个页表级中的属性经组合后的页保护属性。经组合的属性通过对两个级的属性，执行“与”操作来形成。因此，组合属性有相对于两个级别更严格地限制。例如，假定页目录项有 U/S=1及 R/W=1，表示可由用户读/写/执行的页，但页表项有 U/S=1，R/W=0，表示用户只可读/执行。经组合的保护属性为 U/S=1及 R/W=0，这也是最后的用户得到的权限，即页目录中的写权限被抑制。

表 4.4 组合页保护属性

页目录 U/S	页表 U/S	组合 U/S
0	0	0
0	1	0
1	0	0
1	1	1
页目录 R/W	**页表 R/W**	**组合 R/W**
0	0	0
0	1	0
1	0	0
1	1	1

4.3.5 修改页表项的软件问题

本节介绍：当修改页表项时，操作系统软件应遵循的若干准则。另外分页机制转换高速缓存的使用，要求所有的系统遵从一定的准则。多处理器系统中，一个处理器要改变另一个并发执行的处理器的页表项，还要遵从一些附加准则。

4.3.5.1 分页机制转换高速缓存的相关性

为了避免在每次存储器访问时，都要访问常驻内存中的页表、从而提高访问内存的速度，80386处理器的硬件，把最近使用的线性—物理地址转换，存储在处理器内的页转换高速缓存中。在访问存储器页表之前总是先查阅高速缓存。仅当必需的转换不在高速缓存中时，才访问

两级页表进行虚拟—物理地址的转换。页转换高速缓存是页机制中与用于加速段转换的描述符投影寄存器相类似的对应部分，以加速页的查找。在其它计算机中，页机制转换高速缓存使用的术语是转换查找缓存或叫 TLB。

在分页机制转换高速缓存中的数据与页表中的数据的相关性，不是由80386处理器进行维护，但必须由操作系统软件保存。也就是说，处理器并不知道软件什么时候会修改页表，例如改变页的基地址或把页标志为不存在等。在一个合理的系统中，页表只能由操作系统修改，操作系统可以直接地在软件修改页表后通过刷新高速缓存来保证相关性。高速缓存的刷新通过装入处理器控制寄存器 CR3完成。实际过程可能用如下的两条代码实现：

```
MOV    EAX    CR3;          CR3中的值传送到 EAX
MOV    CR3    EAX;          从 EAX 将 CR3中的值送回并刷新高速缓存。
```

一个重要的修改页表项的特殊情况，不需要对页转换高速缓存进行刷新。这种情况是指修改不存在表项的任一部分，即使 P 位本身从 P=0改变为 P=1时也一样。因为无效的表项不会存入高速缓存，因此，当无效的表项被改变时，不需要刷新高速缓存。这也表明在从磁盘上读入一页使其存在时，不必刷新高速缓存。

4.3.5.2 *有关多处理器的问题*

在一个多处理器系统中，必须特别注意是否在一个处理器中执行的程序，会改变可能由另外的处理器同时访问的页表。在80386处理器中，每当要更新页表项，并设置 D 位或 A 位时，通过使用不可分的读/修改/写周期支持多处理器的配置。对于页表项的软件更新需借助于使用 Lock 前缀(见第3章)，从而保证修改页表的指令工作在不可分的读/修改/写周期中。在改变一个可能由另外的处理器使用的页表之前，最好使用一条加锁的 AND 指令在一个不可分的操作中将 P 位清除为0，然后，该表项可根据要求进行修改，并随后把 P 位置成1而使表项成为可用。

当修改页表项时必须及时通知(通常使用中断方式)系统中该表项已高速缓存的所有处理器刷新各自的页转换高速缓存，以撤消该表项的旧拷贝。在表项的旧拷贝被刷新之前，各处理器仍可以继续访问旧的页，并且可以设置正被修改的表项中的 D 位。如果这样做引起表项修改失败，则分页机制高速缓存最好在表项被标记为不存在之后，并在对表项进行另外的修改之前进行刷新。

4.4 处理器控制寄存器及系统段

本节描述控制段及分页机制操作的寄存器及存储器段。这些控制寄存器包含段的基地址、页转换表及控制处理器操作的控制位。控制寄存器只能由特权级为0的程序访问，即只能由最内层特权级的程序访问。如果较外层的程序试图往这些寄存器写入新的内容，则会产生异常，使操作系统核心采取适当的措施。

图4.19示出了80386的控制寄存器。由图可见，有四个32位的控制寄存器，分别名为 CR0，CR1，CR2，CR3；有两个48位的寄存器，名为 GDTR 及 IDTR；还有两个16位的选择子寄存器，名为 LDTR 及 TR。图中还示出了与 LDTR 及 TR 相联系的描述符投影寄存器。投影寄存器用虚线框出。对编程人员来说，投影寄存器是不可见的。

4.4.1 处理器控制寄存器

处理器的控制寄存器为 CR0～CR3，CR1为保留寄存器，可供今后开发的处理器使用。

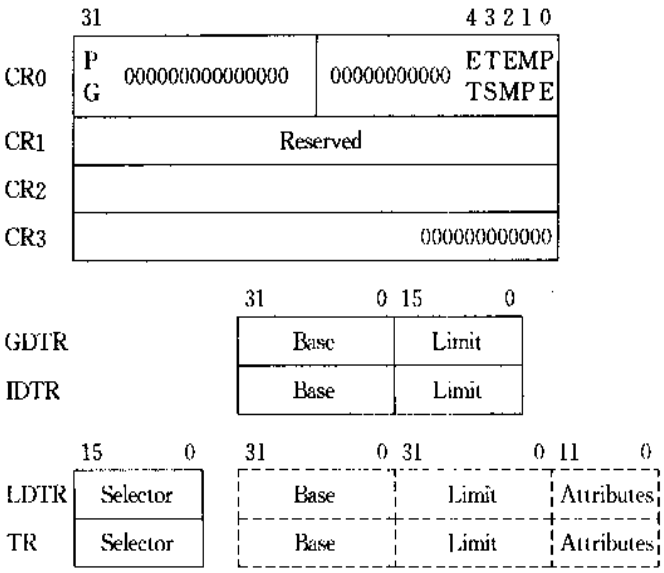


图 4.19 处理器控制寄存器

80386未对CR1进行定义、试图把CR1作为寄存器编码用在指令中，将引起无效指令操作异常(详见第5章)。CR0包含启用或禁用分页机制及保护机制的控制位、控制浮点协处理器操作的控制位。CR2及CR3由分页机制使用。CR0中的位30～位5及CR3中的位11～位0为保留位，但这些位不能为随意值，而是必须为0。

处理器控制寄存器可由在0级特权级执行的程序，通过使用特殊格式的MOV指令装入，详见第3章。

4.4.1.1 CR0协处理器控制位

CR0中的1～4位分别名为MP、EM、TS及ET位，控制80387浮点协处理器的操作。ET位选择与协处理器进行通讯时使用的协议。TS、MP及EM位确定浮点指令或WAIT指令是否产生设备不可使用(DNA)异常(见第5章)。第6章中的协处理器异常处理程序实例会告诉读者：如何使用DNA异常，为使用浮点运算的任务保存及恢复浮点寄存器。ET、TS、MP及EM的用法分述如下：

ET ET为扩展类型位。控制把浮点指令送到协处理器使用的协议。ET=1表示系统中配置有80387协处理器，因而使用32位协处理器协议。ET=0表示使用16位的协议与80287协处理器进行通讯。当EM位为1时，ET位不起作用。

TS TS为任务切换位。用于加快任务的切换。这一目的，通过只在必要时才进行协处理的切换来实现。每当要进行任务切换时，处理器把TS位置为1。当TS=1时浮点指令将产生DNA异常。而当TS=1，同时MP=1时，WAIT指令将产生DNA异常。

MP MP位为算术存在位。控制WAIT指令在TS=1时是否产生DNA异常。如果MP=1，在TS=1的情况下执行WAIT指令将产生异常；如果MP=0，则WAIT指令忽略TS条件。

EM 模拟位。控制浮点指令的执行是用软件模拟，还是由硬件执行。EM=1时，浮点指令产生异常，浮点指令由软件模拟；EM=0，硬件控制浮点指令传送到协处理器

表4.5给出了EM、TS及MP位产生DNA异常的用法。

表 4.5 EM、TS 及 MP 位与浮点 WAIT 指令

CR0位			指令类型	
EM	TS	MP	浮点	WAIT
0	0	0	执行	执行
0	0	1	执行	执行
0	1	0	DNA 异常	执行
0	1	1	DNA 异常	DNA 异常
1	0	0	DNA 异常	执行
1	0	1	DNA 异常	执行
1	1	0	DNA 异常	执行
1	1	1	DNA 异常	DNA 异常

当处理器复位时，ET 位被初始化，以指示系统中数字协处理器的类型。如果80387协处理器存在，则 ET 位被置为1；否则，如果80287协处理器存在，或者系统中没有协处理器，ET 位清除为0。参考 Intel 手册的数据，可得到80386、80387及80287在硬件复位时的更多的信息。

4.4.1.2 CR0保护控制位

CR0寄存器中的 PE 位(位0)及 PG 位(位31)，控制段及分页机制的操作。PE 位用于控制段机制。如 PE=1，处理器工作于保护方式；如 PE=0，处理器运行于实地址方式，且等同于8086(见第7章)。PG 用于控制分页机制。如 PG=1，启用分页机制，线性地址—物理地址的转换按本章描述的过程进行；如 PG=0，禁用分页机制，段机制产生的线性地址，直接地当做物理地址使用。

表4.6列出通过使用 PE 及 PG 位选择的处理器的方式。两个位分别为0或1，只有四种组合，但只有其中三种组合方式有效，而 PG=1且 PE=0是一种非法组合。因此，用首位为1末位为0的32位的值装入 CR0寄存器将产生通用保护异常。

表 4.6 PG、PE 位与处理器方式

PG	PE	执行方式
0	0	实地址(见第7章)
0	1	保护方式，禁用分页机制
1	0	非法组合
1	1	保护方式、启用分页机制

必须注意 PG 位及 PE 位的修改对程序的影响。PG 位的修改将使系统启用或禁用分页机制，因而只有当所执行的程序的代码和至少有一部分数据在线性地址空间和物理地址空间具有相同的地址的情况下，象在第6章解释的那样才能改变 PG 位。另外也要注意，在启用分页机制之前，即在把 PG 位置为1之前，要预先刷新分页机制的高速缓存。

在改变了 PE 位的值以后，程序必须立即执行一条 jump 指令，以刷新已经预取的，当然是在一种不正确的方式下的指令流。在 PE 位被置位之前，程序必须先初始化系统段及控制寄存器。如处理器使 CR0的 PE 及 PG 位初始化为0，则处理器在实地址方式下工作，使得引导程序可以把支持段及分页机制需要的寄存器及数据结构，在这些机制启用之前进行初始化。参考第6章关于如何对处理器进行初始化的实例，及第7章关于实地址方式的更详细的信息。

4.4.1.3 CR2及CR3

CR2及CR3寄存器由分页机制使用，CR3中的内容是第一级页表即页目录表的起始物理地址，由于目录是页对齐的，所以寄存器中仅高20位有效。低12位保留未用。经CR3中装入新的值时，低12的各位必须为0，而保存CR3中的值时，低12位被忽略。

使用第3章描述的指令：MOV CR3 rcg 装入CR3寄存器，会产生使分页高速缓存无效的副作用。CR3寄存器即使在CR0中的PG位为0时也可装入，这就可以进行分页机制的初始化。CR3在任务切换时要被改变，但是如果新任务中CR3的值，与原任务的CR3的值相同，则处理器不必刷新分页高速缓存，因而当任务共享页表时有较快的执行速度。

CR2用于发生页异常时报告出错信息。当发生页异常时，处理器把引起异常的线性地址保存于CR2中。操作系统中的页异常处理程序可以检查CR2的内容，从而查出线性地址空间中的哪一页引起本次异常。

4.4.2 段表基地址寄存器

GDTR、IDTR、LDTR及TR是系统中非常重要的一些特殊段的基地址寄存器，这些段包含有对段机制来说是很重要的表格。GDTR、IDTR及LDTR寻址包含描述符表的段，而TR寄存器则对一个特别的任务状态段(TSS)进行寻址。在任务状态段中，包含有当前正在执行的任务的重要信息。

GDTR指向GDT，48位的GDTR定义GDT的基地址及界限，其中32位为基地址，16位为界限。GDTR提供段表结构的“根”，LDT的描述符及当前任务的任务状态段的描述符都存放于GDT中，所以这些系统段可以用类似于其它段的方法，由选择子进行指定。由于GDT不能由GDT表本身之内的描述符进行定义，因而GDTR为GDT这一特别的系统段提供了一个伪描述符，确定了线性地址空间中段表结构的“根”。

IDTR包含一个用于规定中断描符表(IDT)的48位的伪描述符。IDT在线性地址空间中，直接地由一个伪描述符定根，而不是用GDT中的描述符定根。这样做是为了避免在访问中断描术符时使用一个间接级。IDT及描述述符类型详见第5章。

图4.20示出48位伪描述符的格式。高32位包含线性地址空间中的段的基地址，低16位包含界限字段，界限以字节为粒度，因而界限字段提供了从一个字节到64K字节的段表容量。由于描述符占8个字节长，因此操作系统最好为含有N个描述符的界限字段的值设置为8×N−1。

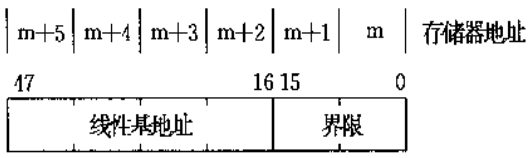


图 4.20 伪描述符格式

LDTR寄存器包含当前任务的LDT的选择子。该选择子必须确定一个由GDT映射的类型为LDT的段，亦即选择子中的TI位必须为0，以表示与选择子相联系的描述符存储于GDT中，而且描述符中的类型字段所表示的段的类型为LDT。

可以用一个空选择子装入LDTR，这表示当前的任务没有LDT。在这种情况下，如果用一个选择子，其TI位为1，表明对应描述符存储于LDT中，并用此描述符产生一个地址，系统将

产生异常。如 LDTR 用一个空选择子装入，则所有的地址必须使用通过 GDT 映射的段的选择子。

TR 寄存器包含当前任务的任务状态段的选择子(任务状态段的说明见下节)。选择子必须确定一个由 GDT 映射的段。选择子不能为空，所指定的段的类型必须为 TSS。TSS 由 TR 寄存器进行寻址，在 TSS 中包含有当前正在处理器上执行的任务的重要信息。

注意，所有这些系统段定义为线性地址空间的连续区域，就象其它段也定义成连续线性地址空间区域一样。因为这些段在线性地址空间中，所以，这些系统段可以由分页机制进行重定位。而且，在一个有很多任务的虚拟存储系统中，不活动任务的 LDT 及 TSS 段，可以以页为单位转换到磁盘上，并在需要时从磁盘上取回。因为 IDT 及 GDT 由所有的任务访问，所以这些系统段最好总是驻留在物理存储器中。

所有系统段都不能由程序直接进行访问，即使是0级的操作系统核心也不能直接进行访问。对专用于 LDT 及 TSS 段的描述符类型，如试图把名为这样的描述符之一的选择子装入到六个段寄存器之一，此时将引起故障。GDT 及 IDT 段则根本没有选择子或段描述符，为允许操作系统核心对这些段的内容进行检查或修改，必须用同样的基地址及界限定义一个只读或读写段，并具有适合于装入到 DS、ES、FS、或 GS 之一以供程序访问的属性。这些附加的段称为别名段，因为这些段为线性地址空间中同样的区域提供了不同的名字。

4.4.3 任务状态段格式

任务状态段(TSS)是包含有关任务的重要信息的特殊的段。活动任务的任务状态段由 TR 寄存器进行寻址。不活动任务的任务状态段包含着暂时固定的任务的各种信息。通过在 TSS 中保存任务的寄存器状态的完整映象，任务状态段可以保证任务的挂起及恢复。当任务被挂起时，当前处理器中各寄存器的值写入 TSS 相应字段。当任务恢复执行时，处理器各寄存器用保存在 TSS 中的值装入，重新建立起任务的状态，使任务继续运行，就好象该任务未被挂起一样。TSS 的安排示于图4.21，包含五种不同类型的信息：链接字段；内层栈指针；有关地址映射的基寄存器；寄存器保存区域；其它字段。

80386处理器定义 TSS 的前104个字节的格式。在此硬件定义的区域以上，操作系统还可以存储有关任务的若干附加信息。如上所述，必须为 TSS 定义一个别名："存储器段"，以保证操作系统可以读出或写入 TSS 中的内容。

由于 TSS 存储在线性地址空间中，因而可以由分页机制对 TSS 重新进行定位，甚至可以把 TSS 换出到虚拟存储系统的磁盘中。要把 TSS 从内存换出，只有一个特别的要求，即任务切换的中间不允许发生页异常。如任务状态段所在页不在存储器中，则对此任务状态段的访问将产生页异常，任务状态段所在页可从磁盘上读回内存。由于在任务切换操作期间，处理器只访问任务状态段的前104个字节，因此，采用一些简单的策略可以保证在任务切换中间不会发生页异常：一种策略是不要把 TSS 段换到磁盘上，或者把该段标志为不存在；另一种策略是，通过保证每个 TSS 段都不跨越页的边界，从而保证 TSS 被对换时，要么整个段存在，要么整个段不存在。事实上，这一点在设计时可以很容易地实现。这只要把每个段对齐在线性地址空间的128字节边界即可。如果 TSS 的基地址对齐在128字节边界，则由于4K 可被128整除，因而 TSS 的前104字节不会跨越页边界。

4.4.3.1 链接字段

链接字段放在 TSS 中偏移为0的位置，长度为32位，其中低16位为选择子，高16位未用。链

31		0	
I/O Permission Bitmap Offset	0000000000000000	T	64h
0000000000000000	LDT		60h
0000000000000000	GS		5Ch
0000000000000000	FS		58h
0000000000000000	DS		54h
0000000000000000	SS		50h
0000000000000000	CS		4Ch
0000000000000000	ES		48h
EDI			44h
ESI			40h
EBP			3Ch
ESP			38h
EBX			34h
EDX			30h
ECX			2Ch
EAX			28h
EFLAGS			24h
EIP			20h
CR3			1Ch
0000000000000000	SS2		18h
ESP2			14h
0000000000000000	SS1		10h
ESP1			0Ch
0000000000000000	SS0		8
ESP0			4
0000000000000000	LINK		0

图 4.21 任务状态段

接字段与 EFLAGS 寄存器中的 NT 位配合使用，用以链接由 CALL 指令或中断挂起的任务的任务状态段。通过执行涉及一个任务的 CALL 指令可以将任务挂起。这一方法，将在4.7.4任务切换细节一节中予以描述。中断处理可以引起任务切换，故一个任务也可由于接收到一个中断而挂起，这将在第5章中描述。如果当前的任务由 CALL 指令或中断而激活，则在 TSS 中的链接字段保存被挂起任务的 TSS 的选择子，且 NT 位被置为1，使链接字段有效。在返回时，由于 NT 位为1，此时的 IRET 指令，使该指令沿着链接字段恢复到链上的前一个任务去执行。

图4.22表示出了任务链的链接链。当前任务的 TSS 段(图中的任务 C)由 TR 寄存器寻址，

被挂起任务的任务状态段(图中的任务 A 及任务 B)由链接字段使其链接在一起(如图中的箭头线所示)。后面将会看到,当任务 C 正执行时 EFLAGS 寄存器中的 NT 位应该为1,以表明任务 C 是一个嵌套的任务,同时,任务 B 的 TSS_B 中 EFLAGS 寄存器映象的 NT 位也应该为1,表示任务 B 嵌套在另一个任务的里面,而任务 A 的 TSS_A 中的 NT 位应该为0,表明任务 A 没有嵌套在另一个任务里面。因此 NT=0表示链接链的结束,并且 TSS_A 的链接字段不会被访问。

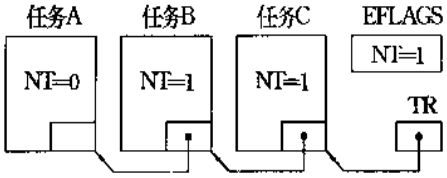


图 4.22 任务状态段的链接链

图4.22中表示的 TSS 链接链,由 CALL 指令或中断引起的一系列嵌套的任务切换所建立。当任务 A 正执行时,接收到的 CALL 指令或中断,产生从任务 A 到任务 B 的任务切换。由此,任务 A 被挂起,而任务 B 被激活,任务 B 中的链接字段指向 TSS_A。在任务 B 执行期间,另一个 CALL 或中断引起切换到任务 C,并使得任务 B 被挂起,任务 C 被激活,并且 TSS_C 中的链接字段指向 TSS_B,这一过程就形成了如图4.22的 TSS 链接链。

4.4.3.2 内层栈指针

前面已经说明,每一特权级如何通过使用各自的堆栈,避免由于共享栈而产生的保护问题。当从某一特权级改变到一个内层特权级时,内层特权级的堆栈,通过装入 TSS 堆栈区域中的适当的堆栈指针而初始化,把外层的堆栈指针压入到内层堆栈中,所以外层堆栈可以在内层向外层返回时得到恢复。特权级的改变及与之相联系的堆栈切换在4.6控制转移方法一节中描述。

任务状态段中有三个堆栈指针,它们都是48位的全指针,分别指向0级,1级及2级堆栈的栈顶,依次存放在 TSS 中偏移量为4、12及20开始的位置。当发生向内层转移时,则把适当的指针装入到 SS 及 ESP 寄存器以切换到内层的堆栈。没有指向3级堆栈的堆栈指针,因为3级是在最外层,所以任何一个向内层转移都不可能转移到3级。如果任务在3级挂起,则指向它的指针保存在 TSS 的 SS 及 ESP 寄存器映象中。

80386处理器从 TSS 中的堆栈指针区域读出内层堆栈指针,但从不向这一区域进行写入。这表明内层转移总是把内层堆栈初始化为同样的指针,在每次内层转移开始时,内层栈总是一个空栈。这无疑是正确的,因为不可能发生同级内层转移的递归。一旦某一内层转移发生时,返回到外层的唯一途径是相匹配的外层返回,把内层堆栈内容弹回外层,使内层栈变空。

4.4.3.3 有关地址映射的基寄存器

当任务切换发生时,要用新任务即 TSS 字段中的相应内容装入 LDTR 及 CR3寄存器。LDTR 装入的是 TSS 中偏移量为60h 的选择子,CR3装入的是偏移量为1Ch 中的双字。

在 CR3中装入新值,即把页表改变为新任务的页表,因而改变了线性-物理转换函数。同样地,装入 LDTR 寄存器,把 LDT 改变为新任务的 LDT,从而改变了由 LDT 映射的虚拟地址空间的虚拟—线性映射函数。如上所述,这种在任务之间改变转换函数的能力,是保护机制的一部分,使任务之间得以隔离。

请读者注意，在经任务切换进入某一任务时，80386处理器把 TSS 中 LDT 及 CR3字段读出装入到适当的寄存器中，但处理器并不向这些字段进行写入操作，即使从任务切换出去时也是一样。因此，如果程序改变了 LDTR 或 CR3，则必须在当前正执行的任务中把新值存储在 TSS 中。

4.4.3.4 寄存器保存区域

TSS 的寄存器保存区域用于保存通用寄存器、处理器控制寄存器及段寄存器。当 TSS 对应的任务正在执行时，保存区域是未定义的；在当前任务被切换出时，上述寄存器的当前值，就存储在保存区域。所以，当再次发生任务切换，并切换回原任务时，各寄存器的值可以从保存区域读出，使处理器恢复成该任务换出前的状态，从而使任务能够恢复执行。

寄存器保存区域，安排在 TSS 段中偏移量为20h～5Fh 区域中(见图4.21)。各个段寄存器都保存在一个32位的双字中，段寄存器中的选择子只有16位，放在双字的低16位，双字的高16位空着未用。

4.4.3.5 其它字段

I/O 许可位图在 TSS 中的偏移量，为从66h 开始的区域。它定义了可由 TSS 对应任务寻址的 I/O 端口地址，I/O 许可位图本身是 TSS 中的附加字段。

在偏移量为64h 处的字，为任务提供特别的属性。在80386中，只定义了一种属性，即调试陷阱属性。该属性用存储在64h 处字的最低位 T 来表示。为了与以后开发的处理器相兼容，字的其它位必须为0。如果发生任务切换，进入任务的 T 位为1，则在任务切换完成之后新任务的第一条指令执行之前产生调试陷阱。这一调试陷阱，使得软件可以在任务之间，根据需要有效地共享调试寄存器，而不会因具有这样的功能而加重了标准任务切换的负担。调试陷阱及调试寄存器详见附录 A。

4.5 对特权级敏感的指令

某些80386的指令，由于执行这些指令的程序所在特权级的不同，而有不同的执行结果。赋予了特权的指令，只能在特权级0执行；如果在其它级执行这些指令，则会产生异常。I/O 敏感指令只能由处在与 EFLAGS 寄存器中 IOPL 字段所表示特权级相同级或更内层级的任务执行，或者 I/O 许可位图(TSS 中的特别的数据结构)规定的可由当前任务访问的 I/O 地址访问指令可以执行。最后，修改 EFLAGS 寄存器的指令，如果不是在特权级0执行，则不会改变某些字段的值；如果不是在相对于 IOPL 同级或更内层级执行，也不会改变其它字段的值。

4.5.1 被赋予特权的指令

对关键的保护模式寄存器进行访问的指令是特权指令。80386规定，只有在特权级0执行的程序可以访问保护模式寄存器。由于对这些寄存器不能随意进行访问，从而保证了保护模式的完整性。表4.7列出的是80386的特权指令。请注意，装入 GDTR，IDTR，LDTR，TR 及 MSW (CR0的低16位)的指令是特权指令，而存储上述寄存器的指令不是特权指令。这表明，任何程序可以存储这些寄存器，但只有0级的程序可以改变寄存器的内容。与此相对比，控制寄存器及调试寄存器，除装入指令是特权指令以外，存储这些寄存器的指令也是特权指令。

表 4.7 特权指令

指令(格式)	功　能	指令(格式)	功　能
CLTS	清除 CR0中的 TS 位	LTR	装入 TR
HLT	停机	MOV CRn, reg	装入控制寄存器 n
LGDT	装入 GDTR	MOV reg, CRn	保存控制寄存器 n
LIDT	装入 IDTR	MOV DRn, reg	装入排错寄存器 n
LLDT	装入 LDTR	MOV reg, DRn	保存排错寄存器 n
LMSW	装入 MSW(CR0中的低16位)		

4.5.2 I/O 空间保护

有两种机制控制访问 I/O 地址空间及有关的 I/O 指令：

(1) 在 EFLAGS 寄存器中的 IOPL 字段；

(2) 在 TSS 中的 I/O 许可位图。

EFLAGS 寄存器中的 IOPL 字段，定义可以执行所有 I/O 有关的指令、及访问 I/O 空间中所有地址的最外层特权级。TSS 中的 I/O 许可位图，定义了在 I/O 空间中的哪些地址，可以由在任何特权级执行的程序进行访问。

I/O 特权级(IOPL)或更内层特权级执行的程序，可以执行表4.8列出的全部指令。如较外层的程序执行时，CLI 及 STI 指令将产生异常。IN，INS，OUT 及 OUTS 指令在外层执行时，仅当 I/O 许可位图中标为不可访问时，才产生异常。也就是说，这些指令中的任一指令，在相对于 IOPL 更外层的特权级执行时，在产生异常之前，先对 I/O 许可位图进行查询，如果位图允许指令对访问的地址进行访问，则指令的操作正常地进行，否则才产生异常。

表 4.8 I/O 敏感指令

指令	功能
CLI	清除 EFLAGS 中的 IF 位
STI	置位 EFLAGS 中的 IF 位
IN	从 I/O 地址读出数据
INS	从 I/O 地址读出字符串
OUT	从 I/O 地址写数据
OUTS	向 I/O 地址写字符串

由于每个任务有其自己的 TSS 及 EFLAGS 寄存器，所以每个任务可以有不同的 IOPL，并且可以定义不同的 I/O 许可位图。例如，执行游戏程序的任务，可以有一个允许访问操纵杆的位图，执行通讯应用程序的任务，可以有访问网络硬件的位图。

4.5.2.1 I/O 许可位图

I/O 许可位图定义了64K 个 I/O 地址空间中哪些地址可以由在任何特权级执行的程序访问。为此，在 TSS 中存储了一个64K 位的位串，位串中的每一位对应于一个 I/O 字节地址。位0对应于地址0，位1对应于地址1，位 N 对应于地址 N 等等。位图中为0的位，表示对应的 I/O 地址，可以由在任何特权级执行的程序访问；位图中为1的位，则表示对应的 I/O 地址，只能由在 IOPL 特权级或更内层的特权级执行的程序访问。如果一个在相对于 IOPL 更外层特权级的程序，试图对位图中为1对应的 I/O 地址进行访问，则将产生异常。

由于I/O地址空间可按字节进行寻址，而每条指令一次最多可访问四个字节。传输多个字节访问结果前，要逐个对访问字节的每一对应位进行检查，如访问字节对应的每一位都是0，则I/O被允许；如果对应位中任有一位为1，则I/O操作可能产生异常（是否产生异常还取决于特权级的情况）。

多位的检查，必须对任何可能长度的位，及任何对齐组合情况都能正常进行。因此，往往要求要读出位图中的两个字节进行检查。例如，某条指令要对位图中的两个位进行检查，其低位为某字节的最高位，高位为下一字节的最低位，也就是说，即使只有两个位，也可能需检查位图中的两个字节。由于一条指令最多只需检查位图中的四个位，因此，在最坏情况下，也只需检查位图中的两个字节。为使访问位图能尽可能快速的进行，80386总是读出位图中的两个字节，而不管是否有必要这样做。为了避免在位图映射的最高的I/O地址处发生问题，在包含有效映射信息的位图中，在最后一个位图字节之后，TSS界限范围之前，必须另加一个内容为全1的字节。这就使得在任何情况下都可以读出位图中的两个连续字节，即使在访问I/O地址空间的顶部时，也不会产生越界情况。I/O许可位图，可以存储在TSS的第一个64K字节内的任何位置，并且可以是8位的整倍数的任意长度。当前TSS中偏移量为66h中的字，定义I/O许可位图的起始偏移量。因为64K的字节的I/O地址空间需8K字节的位图，因此，I/O许可位图的起始偏移量必须小于56K。亦即I/O位图可以放在TSS中前56K字节内任何地方开始的区域中。位图的末端，取位图起始偏移量加上8K与TSS界限二者之中的较小者。当位图正好为8K时，则把TSS界限定义为大于起始偏移量+8K，而当位图小于8K时，则TSS的界限就放在贴近于位图末端的位置，使位图占有较小的存储。超过TSS界限的位取值为1，使得IOPL更外层访问对应的I/O地址时产生异常。位图可以为空，这只需使TSS界限小于或等于位图起始偏移即可。

例1 设置TSS界限为位图起始偏移+32，此时，在TSS位图中提供了32×8=256个I/O地址。即比IOPL特权级低的任务对于I/O地址空间中0～255地址是可寻址的（还取决于位图中的对应位是否为0），对于大于255的I/O地址进行寻址则是非法的。可见，只对前256个I/O空间中的地址进行寻址，相应的位图只占32字节，而不需TSS段中拿出8K字节供位图使用。

例2 设置TSS界限=起始偏移量-2。此时表明该TSS对应的任务不需访问I/O。

4.5.2.2 I/O地址空间访问

在本小节中，我们采用类似于C语言的程序，对I/O地址空间的访问进行详细说明。

为了帮助读者更好地理解本书所列各例程的程序，除对程序进行解释的文字说明之外，还配以相应的程序框图（个别程序例外），把程序、说明、及框图对照阅读、理解，就一定会对程序的逻辑功能、算法理解得较为清楚、透彻。未接触过C语言的读者，开始会觉得困难、生疏一点，但只要坚持把上述三者结合起来反复思考推敲，经过一段时间之后，就会对C程序逐渐熟悉起来，能逐渐做到直接理解C程序，根据C程序自己画出框图并对程序进行说明。读者不妨试试以检验和测试自己这方面的能力。

顺便说明一下，从本节起，本书将陆续采用若干用C语言编写的例程，用以解释80386中由微程序控制使硬件完成的相应动作。

现在我们返回到对I/O地址空间进行访问的说明。

(1) 例程名：AccessIO( )。

(2) 例程参数：

·IOAddress　要访问的I/O首地址；

·Lenth　访问的字节数；

·RW　访问方式；

·Data　数据地址。

(3) 程序动能：对当前任务的 I/O 地址空间进行访问。

(4) 程序清单：

```
Access IO (IOAddress,Length,RW,Data)
    int IOAddress,   /* 访问首地址   */
        Length,      /* 访问字节数   */
        RW,          /* 访问方式:0读,1写 */
        *Data;       /* 指向读出或写入数据的指针 */
{
    short int ptr2,ptrbits;
    int          BitOffset,ByteOffset,Mask,Result;
/* CPL≤IOPL 时,对所有 IO 地址都可进行访问   */
/* CPL>IOPL 时,需检查 IO 许可位图再确定是否允许进行访问 */
if(CPL>IOPL) {
    /* 从 TSS 中读出位图偏移 */
    AccessLinear(TR.Base+66h,2,0,0,&ptr2); /* 特权级0,读两个字节 */
    /* 计算位图内的字节偏移及位偏移 */
    ByteOffset=IOAddress >>3;
    BitOffset=IOAddress & 7;
    /* 形成屏蔽码 */
    Mask=(01111b≫(4-Length))≪BitOffset;
    /* 读出对应于被访 I/O 地址在位图中的两个字节,并检查是否越界 */
    if(TR.Limit<ptr2+ByteOffset+1)
        segmentException($GP,0);/* 越界时,产生出错码为0的 $GP 异常     */
    AccessLinear(TR.Base+ptr2+ByteOffset,2,0,0,&ptrbits);/* 特权级0,读两个字节 */
    /* 许可位图和屏蔽码相"与",结果为0表示允许访问,结果非0,表示不允许访问并产生异常 */
    Result=ptrbits & Mask;
    if(Result!=0)
        SegmentException($GP,0);/* $GP 异常,出错码为0 */
  }
  /* 如所有的检查通过,则进行实际的访问 */
  AccessPhysicalIO(IOAddress,Length,RW,&Data);
}
```

AccessIO()程序的流程，见图4.23

(5) 执行过程：

AccessIO( )例程首先把 CPL 与 IOPL 进行比较。当 CPL 相对于 IOPL 为同级或更内层时，即 CPL≤IOPL 时，所有的 I/O 地址都是可访问的，此时不需对位图进行检查。当 CPL 在 IOPL 的外层时，即 CPL>IOPL，才需对位图进行检查。

要检查位图偏移量，首先要从当前 TSS 段偏移量为66h 的字中，把位图的起始偏移读出。位图中所需要位的位置，要由被检查的 I/O 地址的字节及位偏移经计算获得。字节偏移由 I/O 地址右移3位(即除以8)得到，字节内的位偏移则为 I/O 地址取模8的余数。得到字节偏移后，从位图中读出以字节偏移开始的两个字节，以保证读出任何可能的长度及字节对齐情况下需要的位图中的位。如果要读的字节超过了 TSS 段的界限，则产生异常，从而支持定义的位图宽度小于8K 字节的情况。

长度屏蔽码的形成是基于 I/O 访问的大小，对于进行1～4字节的 I/O 访问，可以由1～4位构成。长度屏蔽码根据位偏移向左移位以使最低屏蔽位与对应于最低 I/O 地址的位图位对

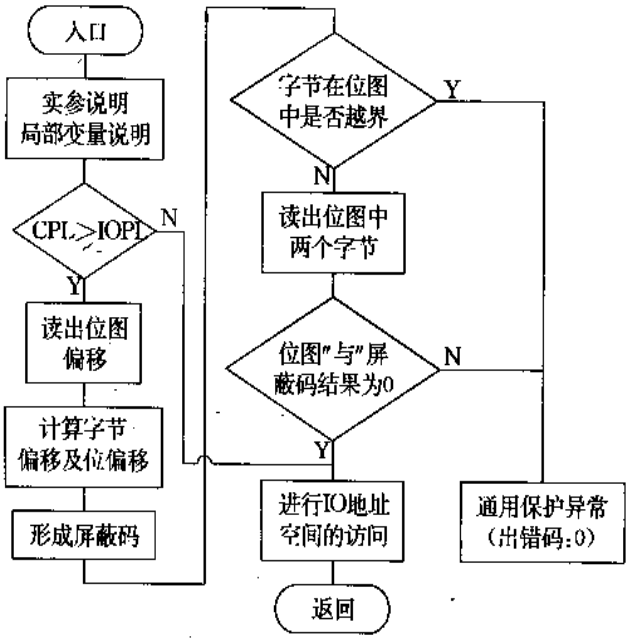


图 4.23 AccessIO( )的程序流程

齐。对从位图中读出的字节和长度屏蔽码进行 AND 操作，清除无关的各个位。如果结果为0，则允许 I/O 访问；结果为非0，则一个或多个访问中跨越的 I/O 地址是不可访问的，并产生一个段异常。

如果允许进行 I/O 访问，访问结果的内容，由一个对 AccessphysicalIO( )例程的调用带出。该例程抽取访问物理 I/O 地址的操作，这部分内容本书未予描述，因为在存储系统中的操作，超出了80386处理器的范围。该例程有与 AccessIO( )例程同样的参数。

小结上述过程，可概括为下列步骤：

·进行特权级的比较；

·找到 I/O 位图的起始地址；

·计算要访问的 I/O 地址的字节偏移、位偏移及 I/O 访问的屏蔽字；

·越界判断；

·读出位图中两个连续的字节并与屏蔽数字进行"与"操作；

·根据"与"操作结果进行物理 I/O 访问。

图4.24示出的位图，用于下面的例子，以图示 I/O 许可位图的操作。存储在当前任务 TSS 内的位图，开始于 TSS 内偏移为66h 的字中给出的偏移。在该例中，偏移量为 m。由每个双字映射的 I/O 地址，连同相对偏移量为 m 的位图开始处的双字地址一并在右面示出。最上面列出的是字节内的位偏移，及双字内的字节偏移。图4.24的位图，包含有17个字节，并加上在最高字节后的一个全1的附加字节。17个有效字节映射的17×8=136个 I/O 地址为0—135。

读者可以核对一下，位图允许访问的地址为2—9，12，13，15，20—24，27，33，34，40，41，48，50，52，53，58—60，62，63及96—127。其它的 I/O 地址，只能由相对于 IOPL 同级或更内层执行的程序进行访问。请读者注意，TSS 界限之前的最后一个字节 m+17字节是装填字节，该字节为全1字节。

例1 假定一个程序在相对于 IOPL 的外层执行，该程序有一个如图4.24的位图，并且试

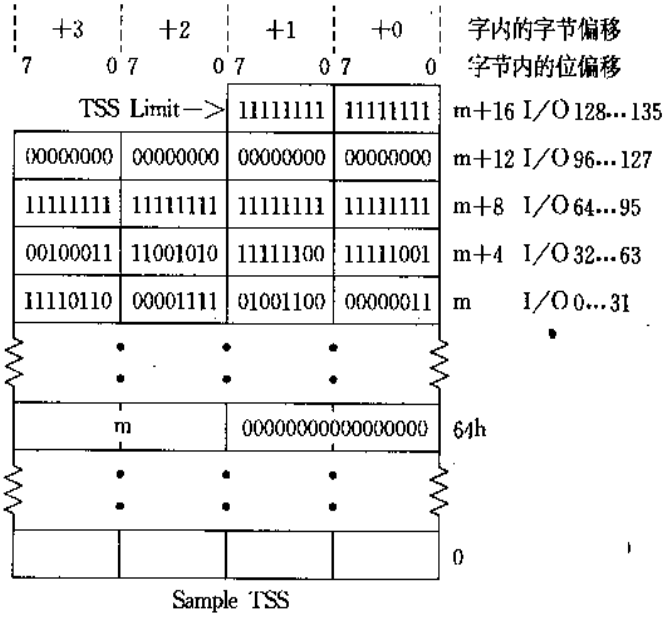


图 4.24 I/O 许可位图实例

图访问在 I/O 地址7的双字。因为访问四个字节，必须检查对应于 I/O 地址7—10的位图中的位。使用程序清单1中的算法，要访问四个字节，屏蔽码为1111。移位次数为地址除以8的余数，此例中移位次数为7，经移位后的屏蔽码为0780h。与此屏蔽码进行"与"操作的两个位图字节，在偏移量为7/8处，即位图中偏移为0和1的两个字节，内容为4C03h。0780h 与4C03h 相"与"，得到0400，即 AND 操作的结果为非0。这是因为操作要访问 I/O 地址7—10，而 I/O 地址10是不可访问的。这由位图中位置为10的位为1所表示。"与"操作的非零结果产生了异常，此过程可用算法表示如下：

$$\begin{aligned}\text{Offset} &= \text{Byte0, bit7}\\ \text{Length} &= 4\\ \text{Mask} &= [1111\text{b} \gg (4-\text{Length})] \ll 7\\ &= (1111\text{b} \gg 0) \ll 7\\ &= 1111\text{b} \ll 7 = 780\text{h}\end{aligned}$$

需读出的字节偏移＝bit/8＝7/8＝0

字节1，字节0的内容，即

位串＝0100110000000011

屏蔽＝0000011100000000

AND＝0000010000000000(非0结果)⇒产生异常

本例是一个要检查的位跨越位图中两个字节的情形。对应于 I/O 地址7的位，在从位图起始地址算起的偏移量为0的字节中，而 I/O 地址为8，9和10的位，则在位图的下一个字节中。

例2 假定一个程序在相对于 IOPL 的外层执行，其位图如图4.24，试图访问 I/O 地址33的字。在此例中，要对 I/O 地址33和34对应的位进行检查。使用上面的算法，需把两个1向左移位。移位次数为33经模8运算的余数结果为1，即把两个1向左移位一次，得到屏蔽码。I/O地址33所在位图字节为33/8，即偏移量为4。把屏蔽码与偏移量为4和5的位图字节内容进行"与"操

作，结果为0，因此允许进行所要求的访问。上述算法可写成：

$$
\begin{aligned}
\text{Offset} &= \text{Byte 4, bit1} \\
\text{Length} &= 2 \\
\text{Mask} &= [1111_b \gg (4 - \text{Length})] \ll 1 \\
&= (1111_b \gg 2) \ll 1 \\
&= 11b \ll 1 = 110b
\end{aligned}
$$

位图字节偏移为4，故读出字节4、5的内容形成的位串，及和屏蔽码的“与”操作结果为：

$$
\begin{array}{l}
\text{位串} = 1111\ 1100\ 1111\ 1001 \\
\text{屏蔽} = 0000\ 0000\ 0000\ 0110 \\
\hline
\text{AND} = 0000\ 0000\ 0000\ 0000
\end{array}
$$

结果为0，允许程序对指定 I/O 地址的访问！

4.5.3 改变 EFLAGS 的指令

对 EFLAGS 寄存器中某些字段的处理，随试图对这些字段进行访问和修改的程序所在特权级的不同而不同。在第2章已经提到，在 EFLAGS 中对 IF，IOPL 及 VM 位的处理，不同于对 EFLAGS 中其它字段的处理。IRET，CLI，STI 及 POPF 指令可用来改变 EFLAGS 中的这些字段。

IOPL 及 VM 位，只能由在特权级0执行的程序修改，IF 位只能由相对于 IOPL 同级或更内层特权级执行的程序进行修改。但是，一个在外层特权级的程序执行 POPF 或 IRET 指令，试图修改这些字段中的任何一个字段，并不产生异常，试图要修改的字段也未被修改，也不给出任何特别的信息。表4.9示出了对 EFLAGS 字段的特殊处理。

除此之外，POPF 指令不修改 VM 位，而 PUSHF 指令总是把0压入到 VM 位的位置。这对于避免通过测试确定系统是处于虚拟8086方式还是实地址方式是必要的。详见第7章。

表 4.9 EFLAGS 字段的特别处理

执行特权级	标志字段		
	VM	IOPL	IF
CPL=0	修改*	修改	修改
0<CPL≤IOPL	不变	不变	修改
IOPL<CPL	不变	不变	不变

* POPF 指令不能修改 VM 应除外

4.6 控制转移方法

本节描述在不同的代码段程序之间由段间跳转、调用及返回指令进行控制转移的方法。为此，提出两种控制转移的形式。第一种形式在本节讨论。实现控制转移的不同方法包括：直接转移到另一代码的偏移地址；通过门转移到另一代码段的入口(点)或通过任务门转移到另一任务。在这一形式中，我们要介绍必需的概念，以理解控制转移到当前代码段以外的所有不同的

方法。第二种形式在稍后讨论。可以使用一个或多个(JMP、CALL 及 RET 指令)进行转移。第5章将描述中断及 IRET 指令如何使用同样的方法实现转移到中断处理及从中断处理返回。

任务切换,即从一个任务控制转移到另一个任务,将在4.7.4进行详细的描述。任务切换是80386段模式最复杂的部分,但不是每个80386系统都必需的部分。通过对任务切换的描述,我们希望读者能得到一个其它控制转移方法的清楚的轮廓,而这些方法又是每个80386系统都必须使用的方法。

第3章对 JMP、CALL 及 RET 指令已做过描述,说明通过改变 CS 及 EIP 寄存器的内容,使其指向不同代码段的指令,达到转移到另一个代码段的目的。这是应用程序编程人员能看到的结果,即看到程序执行了控制转移指令后,马上开始执行由转移指令给出的、不同代码段的程序。在控制转移后,由目标选择子取出相应的描述符,在进行各种保护检查后,装入至 CS 描述符投影寄存器。控制转移的结果可以是转移至同一任务的不同特权级的代码段,也可转移至不同任务的不同特权级的代码段。

4.6.1 同一特权级、同一任务的转移

最简单的段间转移,是使用 JMP,CALL 或 RET 指令,使控制转移到同一任务中同一特权级的代码段。这种转移通过使新的 CS 选择子,对应于一个具有 DPL=CPL,且是一个存在的可执行的存储器段来实现;或对应于 DPL≤CPL,且是一个存在的、一致的可执行代码段来实现。这种控制转移类似于用一个选择子为数据段装入数据段寄存器,并可认为这是一种装入 CS 寄存器的方法。在装入过程中要对新的选择子进行若干检测。若检测通过,则与选择子相联系的描述符从描述符表中读出,再进一步对描述符进行检测。若对描述符的检测也通过,则把描述符装入到 CS 投影寄存器,新的选择子装入到 CS,而新的偏移量装入到 EIP,从而成功地完成控制转移。

一致的可执行段,是一种特别的存储器段。这种存储器段,为在多个特权级执行的程序,提供对子例程的共享支持,而不要求改变特权级。例如,通过把数值库例程放在一致的码段中,可以使不同级执行的程序共享数值库例程。这时,任何特权级的程序,可以使用段间调用指令,调用库中的例程,并在调用者具有的特权级执行该例程。

转移到一致的代码段的控制转移,将在调用者所在特权级执行;而不是在一致的代码段中,由 DPL 所表示的特权级执行。在一致的代码段中的 DPL,用来规定可以转移到一致的代码段的最内层特权级。DPL 的这种解释,正好与正常的 DPL 的解释相反;正常情况下,DPL 用来规定允许访问一个段的最外层特权级。一致的码段用 DPL 规定允许转移到的最内层的特权级,这就意味着,3级的程序可以转移到任何一致码段,而0级例程只允许转移到 DPL=0的一致码段。

4.6.2 同一任务不同特权级的转移

能在相同特权级的段之间进行转移的能力是重要的,但这还不够充分,因为80386有四种特权级,而不是只有一种!CALL 指令通过使用调用门,允许控制转移到更内层的特权级。RET 指令则相反,允许转移到外层的段,以返回到来自外层的 CALL 调用。JMP 指令,则不能在不同特权级之间进行转移。

支持向内层的调用,及向外层的返回,使应用程序可以直接调用内层的操作系统例程,以获得必要的系统服务。例如存储器分配及文件访问等。系统不支持向外层调用及向内层的返

回，因为操作系统不大可能要调用应用程序级的程序来得到服务。只有很少的情形，操作系统必须转移到外层。这时，先将外层程序的段指针及偏移量压入堆栈，再执行一次 RET 指令。对于一个刚装入并可开始执行的程序，这种技术是有用的。参见第6章初始化实例。

转移到内层特权级，必须小心地加以控制，以保证保护机制的完整性。只允许外层级通过操作系统规定的入口点转移到内层。操作系统还必须控制入口点的段及偏移量。否则，如果操作系统只控制段部分，则应用程序可以转移到段内的任何偏移量指示的位置。例如，可以规定一个刚刚超过程序代码之后的偏移量，更糟的是，外层可能规定一个位于一条指令中间的偏移量。在这种情况下，有可能造成灾难性的后果！

4.6.2.1 调用门

调用门提供了从外层对内层例程进行访问所需要的机制。调用门是特别的描述符类型，包含有指向入口点的指针。门描述符中，包含目标地址的段及偏移量的48位全指针。CALL 调用门，就象一个通过门的间接 CALL。外层例程要使用调用门，需为调用门规定一个选择子，作为装入到 CS 的选择子。当 CALL 指令检查新的 CS 描述符，并发现该描述符不是一个可执行存储段，而是一个调用门时，门中的指针被抽出，并作为控制转移的指针使用。使用门中的偏移量代替 CALL 指令中给出的偏移量来控制内层段的入口点。门中的选择子用来读出另一个描述符，该描述符就是调用的实际目标的可执行段。

图4.25示出调用门如何间接地规定调用的目标。注意，调用指令中的偏移量被丢弃，而代之以使用门中的偏移量。门的选择子所读取的描述符对应的段必须是一个可执行段。

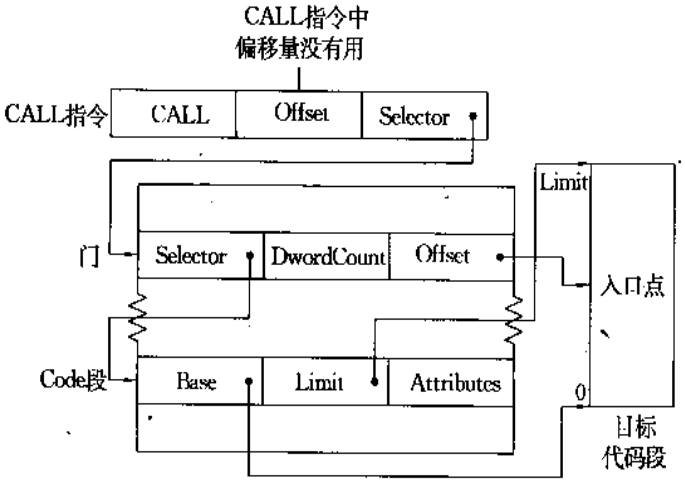


图 4.25 通过调用门的转移

门描述符中的 DPL 字段，使用与访问数据相同的特权级规则控制对门的访问。只有在同一级执行的过程，或者相对于门中 DPL 更内层的过程可以使用门。门的选择子的 RPL 字段也要进行检查，以保证 RPL 与 DPL 在同级或 RPL 是相对于 DPL 更内层的级。因此，门的 DPL 安排在最外层，则对该门的访问就总是被允许的。

特权级检查的一种不同的方法是应用门中的选择子所指向可执行段的描述符的 DPL，而忽视门中选择子的 RPL 字段。如果 DPL=CPL，或者如果是具有 DPL≤CPL 的一致代码段，则门的转移在同一级内进行。JMP 和 CALL 指令，可以使用调用门转移到同级的段。如果 DPL 小于 CPL，则执行一个向内层的转移。只有 CALL 指令，可以使用调用门转移到内层。不

允许 DPL>CPL，以防止转移到外层的调用。RET 指令不管特权级如何，不能使用调用门。

图4.26示出 CALL 指令在 CPL、门的特权级及由门指向的可执行段的特权级之间的关系。一个在2级执行的段 $Code_A$ 中的程序，可以访问2级和3级中的门，这些门必须指向2级或更内层的可执行段。在图中，允许的调用关系用实线表示，不允许的调用关系用虚线表示。

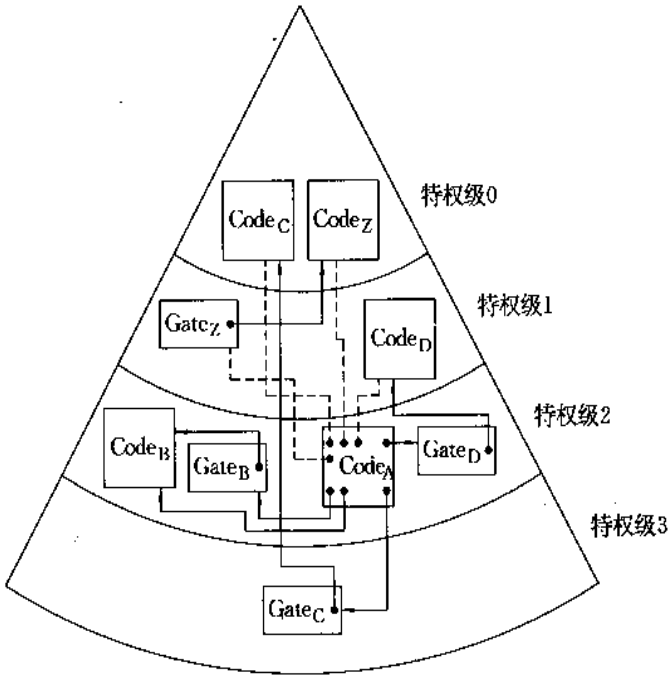


图 4.26 特权级和门及其间的调用

2级的门 $Gate_B$，指向2级的可执行段($Code_B$)。所以在2级使用门 $Gate_B$，不引起特权级的改变。事实上，如图所示，$Code_B$ 可以不使用门而直接访问。3级的门 $Gate_C$ 指向0级的段 $Code_C$，所以在2级使用门 $Gate_C$ 引起特权级转移。$Code_C$ 不可以被直接访问，而只可以通过门 $Gate_C$ 间接进行访问。与此类似，在2级的门 $Gate_D$，指向1级码段 $Code_D$，该码段不能由2级的程序直接访问，在2级使用门 $Gate_D$，也引起特权级的转移。在1级的 $Gate_Z$，不能在2级直接进行访问，0级段 $Code_Z$ 也是不可直接访问的。

4.6.2.2 堆栈切换

通过调用门到内层的 CALL，不仅切换特权级，把控制转移到一个新的码段，而且也切换到内层的堆栈段。如果 CALL 指令转移到同一级，则使用的堆栈段保持不变。即使 CALL 通过门转移到同级，堆栈也不改变。读者一定记得在 TSS 中包含有指向0级、1级及2级栈的指针。所以，使特权级发生变化的 CALL 指令要用 TSS 中适当的指针对 SS 及 ESP 寄存器进行初始化。因为 TSS 中的 ESP 指针通常设置成新栈段的高端限定值，因而在新的特权级建立起一个空栈。当然，新栈为空不会维持很久。

当用选择子为内层栈段装入 SS 寄存器时，同样要应用特权级的检查，犹如在内层特权级执行 MOV SS 选择子 指令一样。这表明段必须是读/写段，而且必须有 RPL=DPL=内层的 CPL。

用 TSS 的指针对新栈进行初始化之后，把原先的 SS 及 ESP 寄存器的值压入内层栈中，

使得相应的向外层返回可恢复原来的外层栈。此后，可以有0—31个双字(即最多124字节)从外层栈拷贝到内层栈。这些拷贝的双字，是调用之前压入到外层栈中的参数，而这些参数，则是原过程随CALL指令传递给新过程，供新过程使用的实参值。通过栈的拷贝，也就使内层的过程很容易在新栈对传递的实参进行访问。传递(即拷贝)的双字的数量，在调用门描述符的Dword Count字段中给出。

图4.27示出的是通过门转移到内层的CALL指令的栈切换操作。外层栈示于图的右面，包含有CALL指令之前已压入栈的四个双字参数(P1、P2、P3及P4)；内层栈示于图的左面，示出成功完成CALL指令之后，内层栈的栈指针及传递的参数分布。外层栈的SS及ESP首先压入内层栈，然后通过拷贝压入传递的四个参数(当然这是假定门的Dword Count字段表示有四个参数)，最后，把返回地址作为CS和EIP值压入堆栈。CALL指令之后，SS为内层程序寻址栈段，而ESP指向栈内由CALL指令压入的断点，即外层的EIP值。

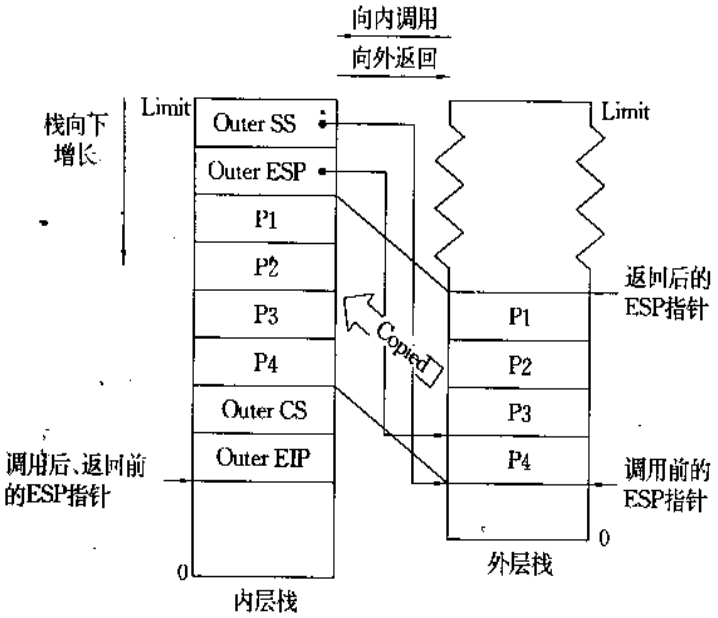


图 4.27 切换到内层栈

虽然调用门对于实现向内层的转移是最有用的，但也可以在调用门中规定一个当前特权级的代码段，或一致的代码段。这样使用调用门时，不会引起特权级的转移，也不会发生栈的切换。CALL指令及JMP指令，可以利用门转移到同一特权级的段。门的这种用法，类似于CALL指令或JMP指令的间接转移。

4.6.2.3 透明性

门的一个重要特性，是门对主调程序的透明性。即主调程序使用标准的段间CALL指令转移到不同的段，而不用区分在调用中给出的选择子是标志存储器段，还是标识一个门(用以间接规定新的段和新的偏移量)。因为操作系统控制描述符表的内容，操作系统可以使用门防止转移到不同特权级的例程。

使用门直接地调用操作系统过程，非常适合于多段编程模式。并且也是使用这种模式的主要优点。然而，由于单段编程模式不使用段间调用，透明性的优点也就不那么显著。在单段模式中，对操作系统过程的调用，典型地是一种对实用程序库例程的段内调用，该实用程序库例程

包含可以通过门的段间调用。

4.6.3 向外层返回

各种 CALL 指令通过调用门，把程序的控制从外层的例程转移到内层的例程。本节描述如何使用 RET 指令，从内层返回到外层。段间 RET 指令从栈中弹出返回指针，并且可以调整 ESP，跳过相应的调用之前压入堆栈的参数。返回指针的选择子部分，指向要返回的段。选择子的 RPL 字段确定返回后的特权级。选择子的 RPL 字段是要使用的，而不是对应描述符的 DPL 字段的值。这一点是必要的，因为返回可以是到一致的段，该段可以在 DPL 字段给定的级以外的特权级执行。

如果选择子的 RPL 字段规定相对于 CPL 更外层的级，则产生向外层返回。在 CPL 被调整到外层之后，指向外层栈的指针从内层栈中弹出，并装入到 SS 及 ESP，以恢复外层栈。然后，调整 ESP，跳过在相应的调用之前压入到外层栈中的参数。在外层恢复执行之前，检查数据段寄存器 DS、ES、FS 及 GS，以保证寻址的段在外层级是可访问的。如果段寄存器寻址的段在外层是不可访问的，则用一个空选择子装入该段，以避免在返回时发生保护空洞。

需注意的是参数字节由向外的返回弹离内层及外层栈。这一点是有意义的，因为有两个参数的拷贝。拷贝到内层栈的参数，必须被弹出以便访问指向外层栈的指针；外层栈的参数必须被弹出，以调整在返回完成以后要使用的栈。

图4.27，用于图示级间调用的栈切换操作，也图示出在匹配的向外返回期间发生的栈切换。在 RET 指令执行之前，内层栈包含返回地址、内层调用期间拷贝的四个双字参数及指向外层栈的 SS:ESP 指针。内层调用执行之前，外层栈使四个双字参数被压入。参数计数为16的 RET 指令将取消级间调用的影响。CS:EIP 返回地址从内层栈弹出，并且内层 ESP 用16调整以跳过参数字节。这就使 ESP 指针，指向了外层栈的 SS:ESP。返回 CS 选择子的 RPL 字段，指示一个外层级，所以 SS 及 ESP 值从栈中弹出，CPL 被改变到外层级，然后外层 ESP 用16调整，以跳过原始的参数。在返回之后，内层栈被废弃，SS 寻址外层栈，ESP 指针如图指向四个参数之上。

4.7 段机制的详细说明

在这一节中，我们要用类似 C 的表示法，详细描述段机制的操作。在以下的若干小节及第5章的相应节中，将提供一组子例程，精确地描述80386的存储管理模型。这些子例程，将逐步的增强读者关于存储管理的概念。但对于作为 C 编译器要求使用的述语，并不总在使用之前予以定义。这些子例程是相互依赖的，例如，大部分子例程调用其它子例程处理存储器访问。描述任务切换的子例程，引用若干级的子例程以描述复杂的切换操作。我们相信，使用子例程的方法，是提供这些材料的最好方法。因为这种方法使读者透彻地理解具体化在子例程中的概念。然后再使用这样的概念作为更高级的概括，去建立更复杂的操作。当访问存储器、装入段寄存器及执行段间控制转移时，这组子例程将提供严格的80386动作的全面的描述。

我们首先要描述的一个重要内容是关于异常。这些异常是要向操作系统报告发生了保护要求的违反。虽然有关异常的详细描述内容主要放在第5章，但本章详述了引起异常的条件，以及随同异常报告的信息。在专门说明异常之前，这里先给出一个在本章如何描述异常的简要的轮廓。有关的说明分成三个主要部分：

(1) 在存储器数据段中对数据的访问；

(2) 段间控制转移；

(3) 任务切换。

第一部分先把类似C语言的表示法加以扩展，使其包括结构定义和指针操作，然后再包括本章其余部分要用的结构类型定义和全局变量。此外，还包括分页机制的描述，这是存储访问的关键部分。

本节详细说明提供了一个80386上存储管理模型操作的格式说明，及确切的80386存储管理机制的规格说明。如果读者是第一次接触这方面的内容，可先阅读第5章及第6章，得到80386可用的全部操作系统设施的更全面概念，再返回到本节的内容，效果可能会更好。其他的读者会发现这些说明作为参考材料更为有用。

4.7.1 异常概述

如果程序试图违反段保护模式的操作，则在以下的详细指令说明中，可能发生几种不同类型的段异常。如果使分页机制启用，则即使是在段机制的操作期间，还可能发生页异常。第5章将描述异常如何中断程序。本章中的详细指令说明，描述了引起段及页异常的条件，以及当报告这些异常时，给出的对软件有用的信息。

4.7.1.1 段异常

本章涉及的段异常及在详细说明中使用的名称如下：

任务切换	$TS
不存在	$NP
栈段	$SS
通用保护	$GP

段异常报告一个出错码可由段处理程序软件发现及消除异常的原因。当段异常终止有问题的程序时，出错码向用户提供必要的诊断信息，其格式示于图4.28。出错码是由引起异常的选择子形成的一个16位的值。当没有选择子时，构成出错码选择子部分的值为0。把选择子装入段寄存器的操作，直接选取选择子的高14位，其中位2是TI位，位15～位3是索引字段。其它操作，例如数据访问，使用值为0的出错码。这是因为在操作中并不直接包含有选择子。选择子中的RPL未用，而用指示引起异常的2个位来代替。位0为EXT位，当处理某一异常或外部中断时，又发生了某种异常，这时将EXT位置为1；位1为IDT位，当从IDT中读出表项并产生异常时置为1，这只在中断或异常的处理期间才会发生。详见第5章。本章的详细说明确切地规定每种可能的异常如何形成出错码。

在详细说明中，凡是有调用子例程SegmentException( )的地方，表示该处已发生异常。在此子例程中有两个参数：

(1) 报告异常时使用的矢量号，用预定义的名为$TS、$NP，$SS及$GP的助记符指示。矢量号与异常的对应关系见第5章；

(2) 随异常报告的出错码中使用的选择子。

4.7.1.2 页异常

页异常也可由本章描述的操作产生。段异常可以有几种类型，而页异常只有一种类型。当页异常发生时，处理器把引起页异常的线性地址装入到CR2中。页异常还报告一个出错码，以有助于操作系统软件对其进行诊断。

页异常的出错码只使用三个二进制位(见图4.29)。位0为P位,表示页异常由于访问不存在的页(P=0),或访问违反保护权限的页(P=1)引起;位1和位2表示引起异常时的访问类型。位1为W位,W=1表示进行写访问,W=0表示进行读访问。位2为U位,U=1表示本次访问来自于特权级3(即用户级)执行的程序,U=0表示访问来自特权级0、1和2(即系统级)执行的程序。

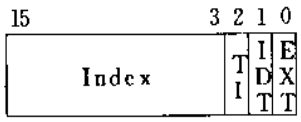


图 4.28 段异常出错代码格式

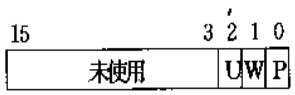


图 4.29 页异常出错代码格式

在详细说明中,页异常由对第5章定义的子例程 PageException( )的调用指示,该子例程有四个参数:

(1) 引起页异常的线性地址;

(2) 试图要进行访问的特权级,该特权级用于形成出错码的U位;

(3) 出错码的W位(1为写,0为读);

(4) 出错码的P位(1为保护异常,0为页不存在异常)。

4.7.1.3 指令说明的中止执行

与在详细说明中使用的其它子例程不一样,PageException( )及 SegmentException( )例程不是返回到它们的调用者,而是中止指令说明的执行,终止逆向沿嵌套过程调用的所有子例程。这就使得我们可以更简洁地描述正常的情形而仍精确地描述异常处理。

4.7.2 存储器数据访问

如第2章所述,为了访问存储器中的数据,必须规定虚拟地址的段部分及偏移量。段部分作为段寄存器给出,此寄存器已用包含所需要的数据的段的选择子装入。偏移量是一个段内的字节偏移,可以由基寄存器、带比例因子的变址寄存器及一个常数位移量的和代替。这表明访问存储器中的数据分为两步进行:首先,用一个包含数据的段的选择子装入段寄存器,然后,规定偏移量的内容。

存储器访问的说明分为三个部分,第一部分介绍类型及全局变量,例如用于描述寄存器的那些;第二部分描述使用段寄存器对存储器进行访问,最后说明装入段寄存器的过程。因为装入段寄存器本身要求进行几次存储器访问,所以我们先描述存储器的访问。

4.7.2.1 表示法、结构定义及全局变量

在深入地说明之前,需要先用两个新的构件扩展第3章介绍的C语言表示法。第一,我们使用 Struct 构件去建立结构或记录,作为几种基本数据类型,例如:int(表示整型)、Char(表示字符型)或其它结构的组合体。结构运算符使我们可以为段描述符及选择子等复杂结构,定义自身的数据类型;第二,我们使用指针访问运算符(*),去访问存储器中的数据,被访问数据的地址从指针变量中得到。如果 Ptr 是一个指向整型的指针,则构件 *Ptr 可用来命名由 Ptr 指向的整型数。在变量定义中,*运算符用于定义指针。结构和指针都是比较新的概念,虽然不是C语言所独有,但却是使C语言独具特色的很重要的内容。要深入的理解C程序及结构、指针等内容,请参考有关C语言的专门著作。为了急用,我们这里用一个小小的程序代码段来说明

指针的用法。

```
int I,J,*PTR; /* I,J 为整型,PTR 为指向整型的指针 */
PTR=&J;       /* J 的地址赋值给指针 PTR,即 PTR 指向 J */
*PTR=I;       /*将值 I 赋给 PTR 指向的地址,即 J=I */
```

此程序段中,第一行为说明语句行,int 表示整型,其后的 I,J 为变量,因为变量 I,J 放在 int 之后,所以 I,J 为整型变量。PTR 也是变量,前面有一"*"号,表示 PTR 是一个指针,指针中的内容为整型值。第二行是赋值语句,符号'&'表示地址,&J 表示变量 J 的地址。语句的含义为:把变量 J 的地址赋给指针 PTR。第三行也是赋值语句,表示把变量 I 的值赋给指针 PTR 指向的变量中。因为 PTR 指向变量 J,所以该语句使变量 J 的值等于变量 I 的值。

在 C 语言中,符号/*及*/之间的内容为注释,用以解释程序某行或某一段的作用。借助于注释的帮助,及此处对三行程序代码的解释,希望能帮助读者对以后遇到的程序有个粗略的理解。

此外,C 语言中使用得很多的几个逻辑运算符有必要在此顺便介绍一下,以方便读者阅读即将介绍的各个例程。这些运算符有:

符号"‖"表示逻辑"或",符号"&"表示逻辑"与",符号"!"表示逻辑"非",符号"==="表示相等逻辑判断,符号"!="表示不等逻辑判断。由这些逻辑运算符进行操作的结果皆为逻辑值。

数据访问及段寄存器装入例程的详细说明,使用了一些结构定义及全局变量。在介绍这些例程之前,先以 C 语言的描述方式列出有关结构及全局变量,并加以适当解释。这些结构及全局变量为:

```
Struct SelectorType{
        Bit(2) RPL;                 /*选择子请求特权级,用二位表示*/
        Bit(1) TI;                  /*指示本选择子指定的描述符所在的描述符表的标志,用一位
                                      表示。*/
                                    /*TI 为0,表示指定全局描述符表;TI 为1,表示指定局部描述
                                      符表。 */
        Bit(13) Index;              /*指定描述符在全局(局部)描述符表中的索引,用13位表示*/
}

Struct SegAttributes{
        Bit(5) DwordCount;            /*参数个数,用5位表示,可传送32个参数。
                                        此字段仅在描述符类型为调用门时使用*/
        filler(3);                    /*3位,未使用*/
        Bit(4)Type;                   /*属性类型*/
        Bit(1)DType;                  /*描述符类型。为0表示系统段或门,为1表示存储段*/
        Bit(2)DPL;                    /*描述符特权级*/
        Bit(1)P;                      /*存在位*/
        filler(6);                    /*6位,未使用*/
        Bit(1)D;                      /*指令或字的默认长度(16位或32位)*/
        Bit(1)G;                      /*界限粒度位:0表示字节粒度,1表示4K 字节粒度*/
}

Struct SegmentRegister{
        SelectorType   Selector;        /*可见的选择子寄存器*/
        SegAttributes  Attributes;      /*不可见的投影寄存器*/
        int            Base;            /*段基地址*/
        int            Limit;           /*段界限*/
}
```

```
SegmentRegister CS,SS,DS,ES,FS,GS,   /*数据段、代码段、栈段寄存器*/
                TR,LDTR,             /*系统段寄存器*/
                IDTR,GDTR;           /*中断描述符表、全局描述符表寄存器 */
int CR0,CR2,CR3;                     /*处理器控制寄存器*/
int CPL;                             /*当前特权级*/
int EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI;
                                     /*32位的通用寄存器*/
Struct EflagType{                    /*处理器标志寄存器*/
      Bit(1) CF;
      filler(1);
      Bit(1) PF;
      filler(1);
      Bit(1) AF;
      filler(1);
      Bit(1) ZF,SF,TF,IF,DF,OF;
      Bit(2) IOPL;
      Bit(1) NT;
      filler(1);
      Bit(1) RF,VM;
      filler(14);
}
EflagsType EFLAGS;
int        EIP;
```

在上述的结构定义中，使用了两个不是标准C语言结构的内部类型：Bit类型按所给定的位数定义结构字段，在结构定义之内，规定位字段从低位向高位分配；filler类型，用于定义结构中保留的相应位，表明这些位未被使用。

选择子的结构是由结构定义Selector-Type进行定义，该结构对RPL字段，TI位，及索引字段进行了详细分配。这一定义与图5.9给出的选择子格式的图形表示相一致。段属性由结构定义SegAttributes进行描述，用以描述在描述符中偏移量为4的第二个双字，包括了示于图4.10、4.11及4.12中门及段描述符的所有可能的属性字段及字段的组合。DT及TYPE字段确定余下属性字段的哪一字段是有效的。

段寄存器及相联系的投影寄存器的结构，由结构定义SegmentRegister进行描述，包括四个子字段，以反映图4.13中示出的结构：一个具有结构类型SelectorType的选择子；一个有结构类型SegAttributes的属性字段；以及两个32位的整数，描述基地址及段界限，段界限以整个32位的字节粒度界限存放。段寄存器装入时，在装入到段界限寄存器之前，要将4K粒度的界限转换为字节粒度界限。

有了如上的预定义类型之后，再对各寄存器进行进一步的定义如下，六个段寄存器CS、SS、、DS、ES、FS及GS定义成具有SegmentRegister结构类型。系统寄存器LDTR、TR、IDTR及GDTR也定义为SegmentRegister结构类型。但IDTR及GDTR不使用选择子及属性字段。CPL作为32位的整数定义，控制寄存器CR0、CR2及CR3也定义为32位整数，如同通用寄存器EAX、EBX、ECX、EDX、ESP、EBP、ESI及EDI一样。

EFLAGS寄存器的结构由结构定义EflagsType给出，并与图2.2的定义相一致。

EFLAGS寄存器就是这种类型，EIP寄存器定义为32位整数。

4.7.2.2 数据访问

如果选择子已装入段寄存器，对应的描述符已装入投影寄存器，则寄存器寻址的指令可以访问段内的数据。有关数据访问的例程包括：访问虚拟存储器的例程；有关分页机制的例程；及

访问物理存储器的例程。

1. 访问虚拟存储器的例程

(1) 例程名:AccessVirtual( )

(2) 例程参数:

- SReg　被访问段的段寄存器;
- Offset　段内的偏移量;
- Length　访问的字节数;
- RW　访问方式;
- IntNumber　异常类型;
- Data　数据地址。

(3) 程序功能:访问虚拟存储器

(4) 程序清单:

```
/*访问由段寄存器SReg寻址的段中的数据                                    */
AccessVirtual(SReg,Offset,Length,RW,IntNumber,Data)
SegmentRegister SReg;   /*  被访段所在段寄存器                          */
int Offset,Length;      /*  段内偏移及访问数据的长度                    */
int RW;                 /*  访问方式:0为读,1为写                        */
int IntNumber;          /*  发生异常时的类型:                           */
                        /*  访问SS段产生的异常为$SS,即IntNumber=$SS     */
                        /*  访问其它段产生的异常为$GP,即IntNumber=$GP   */
int *Data;              /*  指向要访问的数据                            */
{
  if(SReg.Attributes.P==0)              /*  段不存在                    */
     SegmentException(IntNumber,0);
  switch(SReg.Attributes.Type){
   case0:case1;         /*只读数据段          */
   case10:case11;       /*可读、可执行代码段  */
   case14:case15;       /*可读、可执行的一致代码段
                        /*如对没有写权限的段进行写操作将引起异常              */
     if(RW==1)
         SegmentException(IntNumber,0);     /*出错代码为0                   */
     break;
   case2:case3;         /*可读可写数据段                                   */
                        /*越界检查(向上扩展的情形)                         */
     if((Offset+Length-1)>SReg.Limit)
         SegmentException(IntNumber,0);     /*越界时的段异常出错码为0       */
     break;
   case4:case5;         /*只读向下扩展数据段*/
     /*对只读向下扩展段进行写操作时产生异常。如不进行写操作,则继续往下进行界限检查*/
     if(RW==1)
         SegmentException(IntNumber,0);     /*出错代码为0                   */
   case6:case7;                             /*可读可写向下扩展数据段        */
                                            /*越界测试*/
     if((Offset<=SReg.Limit)||((Offset+Length-1)>=4G))
         SegmentException(IntNumber,0);     /*出错代码为0*/
     break;
   case8:case9;                             /*只可执行代码段                */
   case12:case13;                           /*只可执行一致代码段            */
                                            /*只执行代码段不允许进行读写访问 */
     SegmentExeption(IntNumber,0);          /*出错代码为0*/
```

```
        break;
    }
    AccessLinear(SReg,Base+Offset,Length,CPL,RW,Data);
}
```

程序流程见图4.30

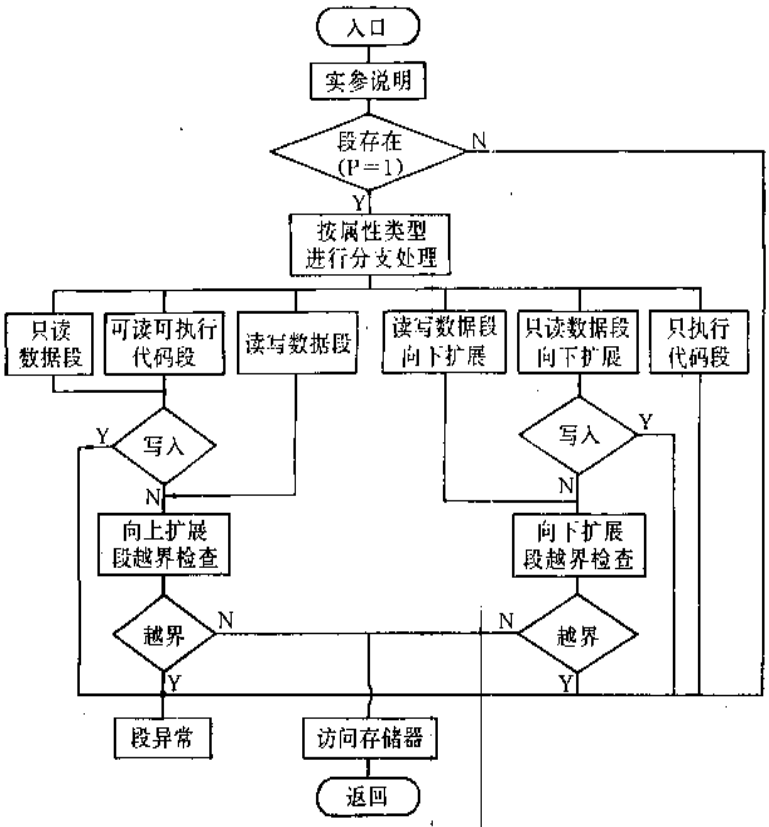


图 4.30 AccessVirtual( )的程序流程

(5) 执行过程

第一，对每个数据访问进行有关的保护检查。由于在装入段寄存器时，多数的保护检查已经完成，这样，就使得在实际访问存储器时的检查项目已减少到最低限度。因为进行存储访问的频度远远高于段寄存器装入操作的频度，因此，在装入段寄存器时完成尽可能多的检查项目，可以提高系统的性能。在数据访问时，仍需检查的项目有：

① 检查段寄存器，看看段寄存器中的选择子是否为空。这一检查通过测试描述符投影寄存器中的属性位 P 来完成；

② 检查偏移量，确保要访问的地址在段的界限之内。80386要求进行两种不同的段界限的检查：一种是正常的向上扩展的段的检查，另一种是向下扩展的段的检查；

③ 依照段的属性检查访问类型，以避免对只读的段进行写入访问或对只允许执行的段进行读出或写入访问。

上述任何一种检查失败，AccessVirtual( )例程向系统报告发现异常，并且不会进行实际的存储器访问。

第二，如果上述的所有检查都得到通过，则段机制允许进行存储器访问，并调用

AccessLinear( )例程，把数据的线性地址传送给分页机制。数据的线性地址，用段基地址与偏移量相加获得。

第三，如果所有的分页访问检查也通过，则指令可以正常地完成。请注意，这些数据访问直接由程序产生，CPL 是作为访问的特权级使用的。

2. 访问线性地址的例程

80386中的数据，有一个字节、两个字节、四个字节及八个字节等多种类型。访问数据时，无论是访问哪一种类型的数据，指令中给出的访问地址，总是指向被访数据的第一个字节的地址。此外，80386中，数据和程序都以"页"为单位进行组织，因此，访问数据(也包括读取指令)时，不能保证被访对象的各个字节总在同一页面。例如，访问地址在0FFFh 处的双字就会跨过两个页面：双字的第一个字节在 Page0，地址为0FFFh；双字的后三个字节在 Page1，地址为1000h～1002h。访问线性地址例程支持跨页访问的处理。在进行跨页访问时，该例程把跨页访问分解为对两个不同页的两次访问来完成。

(1) 例程名：AccessLinear( )。

(2) 例程参数：

- LAddress　起始线性地址；
- Length　访问的字节数；
- PL　访问的特权级；
- RW　访问方式；
- Data　访问地址。

(3) 程序功能：完成线性地址空间的读写访问

(4) 程序清单：

```
AccessLinear(LAddress,Length,PL,RW,Data)
    int LAddress,Length,/* 数据的线性地址及访问字节数 */
      PL,              /* 访问特权级                 */
      RW,              /* 访问方式:0读1写            */
      *Data;           /* 指向读出或写入数据的指针   */
{
    int PAddress,
        LAddress2,PAddress2,
        Len1,
        Len2;
  /* 判断指定的访问是否要跨页 */
  if((LAddress % 4096)+Length)>4096) {
  /* 进行跨页访问,要将访问分成两次不跨页的访问完成。*/
  /* 第一次访问从指定地址 LAddress 开始至页末:  */
        Len1=4096-(LAddress % 4096);
        PAddress=TranslateLinear(LAddress,PL,RW);
  /* 第二次访问下一页,从页首开始,字节数为总数 Length 减去第一次已访问数 Len1;  */
       LAddress2=LAddress+Len1;
       Len2=Length-Len1;
       PAddress2=TranslateLinear(LAddress2,PL,RW);
  /* 分两次访问物理存储器  */
       AccessPhysical(PAddress,Len1,RW,Data);
       AccessPhysical(PAddress2,Len2,RW,Data-Len1);
  }
  else{/* 不需跨页访问 */
        PAddress=TranslateLinear(LAddress,PL,RW);
```

```
        AccessPhysical(PAddress,Length,RW,Data);
    }
}
```

访问线性地址例程的程序流程见图4.31。

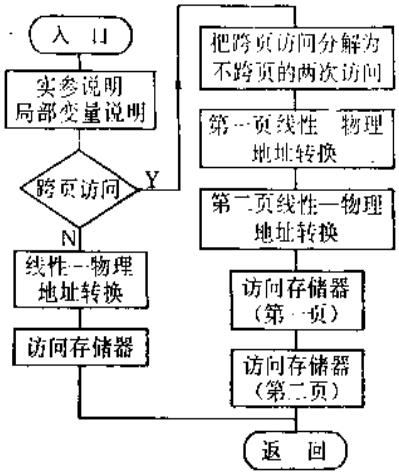


图 4.31 AccessLinear( )的程序流程

(5) 执行过程：

首先检查本次访问是否要进行跨页访问，并在需跨页时，将跨页访问分解为不跨页的两次访问。然后调用 TranslateLinear( )例程(有可能调用两次)，将线性地址转换为物理地址。最后，调用 AccessPhysical( )例程(有可能调用两次)，访问物理地址完成对线性地址的访问。

3. 线性—物理转换例程

线性—物理转换例程是分页机制的核心，用以完成从线性地址到实际物理地址的转换(映射)。

(1) 例程名：TranslateLinear( )

(2) 例程参数：

- LAddress 要转换的线性地址；
- PL 访问特权级；
- RW 访问方式。

(3) 程序功能：将线性地址转换为物理地址

(4) 程序清单：

```
/*分页机制未启用时,物理地址=线性地址,故直接将线性地址返回   */
/*分页机制已启用时,使用两级页表完成线性—物理转换*/
TranslateLinear(LAddress,PL,RW)
  int LAddress,          /* 要转换的线性地址      */
      PL,                /* 访问特权级            */
      RW;                /* 访问方式,0读1写       */
{     /*PTE结构定义页目录/页表结构的表项格式      */
  struct PTE{
      bit(1) P;          /* 为1表示转换有效       */
      bit(1) RW;         /* 为1表示表项可写       */
      bit(1)SU;          /* 为1表示用户级可访问  */
```

```
    filler(2);          /* 未用                    */
    bit(1)A;            /* 访问标志位              */
    bit(1)D;            /* 修改标志位              */
    filler(5)           /* 未用                    */
    bit(20)PageFrame    /* 物理地址高20位          */
   }
   PTE DEntry,          /* 页目录表项              */
    PEntry;             /* 页表表项                */
   int PAddress;
   /* 测试分页机制启用位PG(即控制寄存器CR0),如CR0=0,则表示分页机制未启用 */
   if((CR0&80000000h)==0)
       return(LAddress);
   /* 如未返回,继续往下执行,则表示PG位为1,已启用分页机制,并从CR3指向的页目录表中读出
      页表指针      */
   PAddress=(CR3 & 0FFFFF000h+((LAddress≫20)&0FFCh);
   AccessPhysical (PAddress,4,0,&DEntry);      /* 读目录表项 */
   if(DEntry.P==0)                              /* 页表不存在 */
     PageEXception(LAddress,PL,RW,DEntry.P);
   /* 设置页表已被访问的标志 */
   DEntry.A=1;
   AccessPhysical(PAddress,4,1,&DEntry);       /* 将已更新的页目录表项写回 */
   /* 从页表中读出被访问页的页表表项 */
   PAddress=(DEntry & DFFFFF000h)+((LAddress≫10) & 0FFCh);
   Accessphysical (PAddress,4,0,&PEntry);
   if((PEntry.P==0)      /* P=0表示页未驻留内存 */
     ‖((PL==3)&&      /* 特权级为3时,进行访问特权检查 */
         ((DEntry.U==0)‖PEntry.U==0
         ‖((DEntry.RW==0)&&RW==1))
         ‖((PEntry.RW==0)&&RW==1)))
       )
     )
    )
       PageException(LAddress,PL,RW,PEntry.P);
   /* 执行至此,表示访问合法,设置访问位,并在进行写访问时,设置修改位 */
   PEntry.A=1;
   if(RW==1)
      PEntry.D=1;
   AccessPhysical(PAddress,4,1,&PEntry);
   Return((PEntry & 0FFFFF000h)+(LAddress&0FFFh))
     /* 用页框地址与偏移量合成物理地址 */
}
```

线性—物理转换例程的程序流程见图4.32。

(5) 执行过程:

第一,检查CR0寄存器的PG位,以确定分页机制是否已启用。如果分页机制未被启用,则线性地址就是物理地址,因此,只需把输入的线性地址作为输出的物理地址直接返回即可;如果分页机制已被启用,则要对两级页表进行访问:首先用线性地址的最高10位的值作为索引访问页目录表,得到页表地址。再用次高10位的值作为索引访问页表,从而得到物理地址的页框地址,在访问两级页表进行线性—物理转换过程中,要对目录表项和页表项进行测试,如果任有一个表项标志为不存在,或检测到违反页保护的操作,则分页机制报告存储器访问发生发异常。

第二,如线性—物理转换有效,则修改页表中线性地址中间10位指定的页表项的属性:即

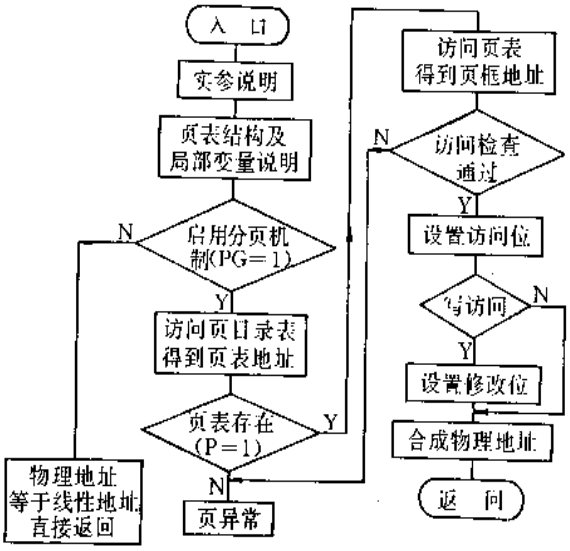


图 4.32 TranslateLinear( )程序流程

属性中的访问位置为1;如果本次访问是写访问,则设置该页已被改写标志。

最后,将页表项中的页框地址与线性地址中的页内偏移相加,形成线性地址经转换后得到的物理地址,并返回该地址。

4. 访问物理存储器的例程

访问物理存储器的例程 AccessPhysical( ),执行访问物理存储器的操作。该例程有四个输入参数:

- 要访问的物理地址;
- 要访问的数据的字节数;
- 访问的方式(读或写);
- 数据地址的指针。

执行这一例程进行的操作包括:用给定的物理地址经过译码选中指定的物理存储单元;根据数据访问的长度和方式,对物理存储器进行读写操作;读出时,把读出的物理地址中的数据放在数据地址指针指定的位置,写入时,将数据地址指针指定的数据写入物理存储器的指定存储单元中。这些操作是在80386处理器外部的存储器系统中进行的,所以这里不进行更细的说明。

4.7.2.3 装入段寄存器

存储器段中的数据在可由程序进行访问之前,必须先把段的选择子装入到80386的几个段寄存器中的某一个寄存器,使要被访问的段成为可以寻址的段。本小节将详细描述如何装入段寄存器及如何把段描述符装入到相应的投影寄存器。

用简单的装入指令只能把段装入到 DS、ES、FS、GS 及 SS 寄存器中,而 CS 只能由段间控制转移装入。当前代码段内的数据(也可以是代码段内的指令,这时该指令又兼作数据使用。)可以通过给定 CS 作为段寄存器及规定所要求的数据在段内的偏移量,对数据进行访问。由于装入到 CS 的段中的数据(包括指令)只能是或者允许执行、或者允许执行和读,所以,对当前的代码段只支持读访问。

装入段寄存器的例程及子例程有段寄存器的装入例程、读出描述符的例程及设置描述符

访问属性的例程。

1. 段寄存器的装入例程

(1) 例程名:SRegLoad( )

(2) 例程参数:

- SReg　要装入的段寄存器;
- Selector　段寄存器的选择子;
- GPorTS　异常类型。MOV、POP 及 Lsr 指令的段装入,使用 $GP;任务切换中的段装入,使用 $TS。

(3) 程序功能:将选择子装入指定段寄存器,段描述符装入描述符投影寄存器。

(4) 程序清单:

```
/* 将描述符装入 SS、DS、ES、FS 或 GS 投影寄存器  */
SRegLoad(SReg,Selector,GP or TS)
      SegmentRegister SReg;
      SelectorType    Selector;
      int GP or TS;    /* 除不存在异常外,矢量号13为 $GP 异常,矢量号10为 $TS 异常 */
{
  SegAttributes Attributes;
  SelectorType  GSelector;
  int           Base,Limit,GOffset;
  /* 装入 SS 段 */
  if(SReg==SS) {
    if(((Selector & 0FFFCh)==0)||(Selector.RPL!=CPL))     /* 选择子测试 */
      SegmentException(GP or TS,0);     /* 出错码为0 */
    /* 读出并测试描述符 */
    ReadDescriptor((Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);
    if((Attributes.DPL!=CPL)
        ||(Attributes.DType==0))     /* 描述符类型为系统段或门时产生异常 */
      SegmentException(GP or TS,0);     /* 出错码为0 */
    switch(Attributes.Type) {
      case2:case3:  /* 可读可写数据段 */
      case6:case7:  /* 可读可写向下扩展数据段 */
        /* 类型正确,检查存在位 */
        if(Attributes.P==0)
          SegmentException(12,selector);     /* 栈段异常,出错码为选择子 */
        break;
      case 0: case1:  /* 只读数据段  */
      case4:case5:  /* 只读,向下扩展数据段  */
      case8:case9:  /* 只执行代码段  */
      case10:case11:  /* 可读可执行代码段  */
      case12:case13:  /* 只执行一致代码段  */
      case14:case15:  /* 可读可执行一致代码段  */
        /* 类型不正确,产生段异常  */
        SegmentException(GP or TS,0);  /* 出错码为0 */
    }
  }
  else {     /* 装入 DS、ES、FS 或 GS */
    if((Selector &0FFFCh)==0)     /* 空描述符 */
    {
      /* 设置描述符无效标志 */
      SReg.Attributes.P=0;
      Return;
```

```
    }
    /* 装入并测试描述符 */
    ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);
    if(Attribute.DType==0)      /* 描述符类型为系统段或门时产生异常 */
      segmentException(GP or TS,0);  /* 出错码为0 */
    switch (Atrributes.Type) {
      case0:case1:     /* 只读数据段      */
      case2:case3:     /* 读写数据段      */
      case4:case5:     /* 只读向下扩展数据段 */
      case6:case7:     /* 可读写向下扩展数据段 */
      case10:case11:     /* 可执行可读代码段 */
        /* 类型正确,进行特权级检查 */
        if((Attributes.DPL<CPL)||Attributes.DPL<Selector.RPL))
          SegmentException(GP or TS,Selector);/* 出错码为选择子 */
      /* 一致码段作为数据段装入时,不检查 DPL */
      case 14:case15:
            /* 检查存在位 */
            if(Attributes.P==0)
              SegmentException(11,Selector);     /* 段不存在异常,出错码为选择子 */
            break
      case8:case9:     /* 只执行的代码段 */
      case12:case13:     /* 只执行的一致代码段 */
          /* 类型不正确,产生异常 */
          SegmentException(GP or TS,0);
    }
  }
  /* 设置访问位,把描述符装入投影寄存器 */
  SetAccessed(Selector);
  SReg.Attributes=Attributes;
  SReg.Base=Base;
  SReg.Limit=Limit;
}
```

SRegLoad( )程序的流程,见图4.33

(5) 执行过程:

第一,选择子测试。选择子有效性的检查是装入段寄存器执行过程的第一步。如果检查失败,则发生异常,指定要装入的段寄存器中的内容不发生变化。读者可以回忆一下。每个选择子有三个字段:

- TI 字段:仅一位,指定包含相应描述符的描述符表。
- RPL 字段:二位,指定选择子所具有的请求特权级。
- Index 字段:13位,指定描述符在描述符表中的索引。

选择子的测试,包括上述三个字段中后两项内容的测试。首先检查的选择子的索引字段,看看该字段中的值是否为0。如索引字段中的内容为0,则表明该选择子是一个空选择子。段寄存器中,DS、ES、FS 及 GS 寄存器可以装入空选择子,但 SS 寄存器中不能装入空选择子。如果把空选择子装入 DS、ES、FS 及 GS,则表明该段未被使用;相应描述符投影寄存器中的内容,除段属性的 P 位被置为0,表明该描述符投影寄存器中是无效描述符外,投影寄存器中其余的内容无任何意义。此后,在进行存储器访问时,如把装有空选择子的寄存器作为段寄存器使用,则会产生异常。把空选择子装入到 SS 寄存器,则在装入时立即产生异常,而不是要等到在装入 SS 后的第一次栈访问时才发生异常。此外,如把选择子装入 SS 寄存器,还要检查选择子的 RPL 字段,并且必须满足 RPL=CPL。

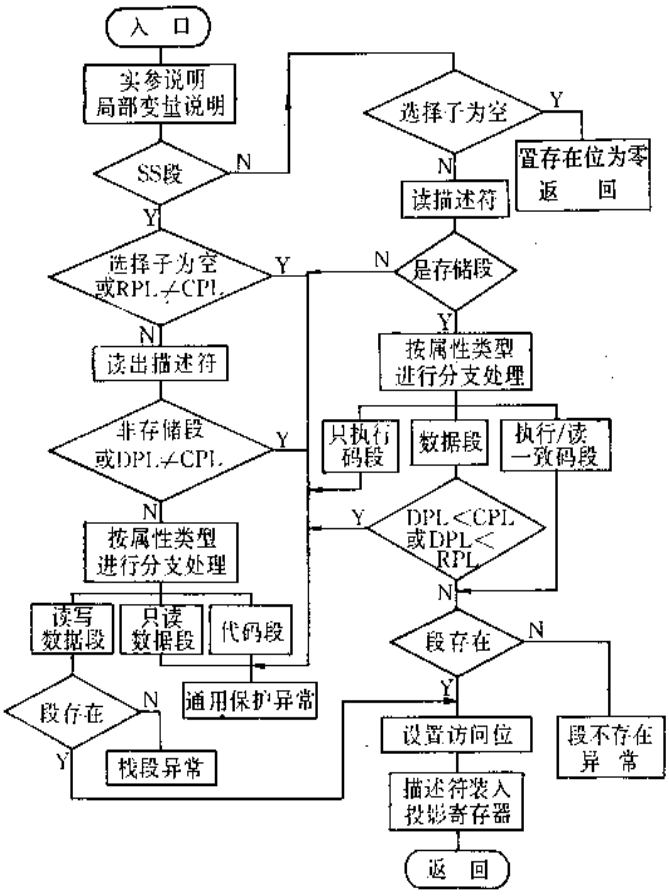


图 4.33 SRegLoad( )程序流程

第二，在选择子测试通过以后，即选择子非空，而且有效，则使用索引字段从 TI 位所指定的描述符表中读出描述符。

第三，对描述符进行检查。检查的内容有：属性、特权级及访问方式检查。读出描述符之后对描述符进行的各种检查，是为了保证要装入的段的合法性。这样的检查，只在段的装入时进行一次，不必在每次使用该段进行存储访问时都做这样的检查。正如对选择子进行的测试一样，装入段寄存器所进行的检查，与所要装入的段有关。一组检查是对装入 DS、ES、FS 及 GS 段寄存器所进行的，另一组检查是对装入 SS 段寄存器所进行的。对装入 SS 寄存器比对装入其它寄存器所进行的检查要更为严格。

对所装入的描述符进行特权级检查时，如装入的是 SS 寄存器，则要求描述符特权级(DPL)等于当前特权级(CPL)。因为已对请求特权级(RPL)进行过检查，并且有 RPL=CPL，所以，装入 SS 寄存器要求有 DPL=RPL=CPL；如装入的是 SS 以外的其它数据段寄存器，则要求 CPL 及 RPL 规定一个相对于 DPL 更内层的特权级，即有：

$$DPL \geqslant CPL \ \&\& \ DPL \geqslant RPL$$

如上式条件不满足，则向系统报告一个异常。因此，在 SRegLoad( )例程中使用的报告异常的条件，就是上述条件的“非”，即产生异常的条件为：

$$!(DPL \geqslant CPL \&\& DPL \geqslant RPL)$$
$$=!(DPL \geqslant CPL) \| !(DPL \geqslant RPL)$$

$$=DPL < CPL \| DPL < RPL$$

对访问方式进行检查的规则为：只有执行/读访问权限的段，可以装入 DS、ES、FS 或 GS 寄存器，但不能装入 SS 寄存器；一致代码段不进行特权级检查，具有执行/读访问权限的一致代码段，可以在任何特权级读出；装入 SS 寄存器必须具有读/写访问权限（理由很明显：因为堆栈段总是要进行写入操作）。

最后，如所有的检查都通过，则将读出的描述符进行分离、组合，装入投影寄存器，再调用 SetAccessed( )例程，设置描述符已被访问标志，完成此例程。

总结前述内容，段寄存器的装入可能有以下三种结果之一：

- 选择子或相应的描述符有问题，例程产生异常，段寄存器及投影寄存器内容都不改变；
- 把空选择子装入 DS、ES、FS 或 GS（不能装入 SS）；
- 把选择子装入段寄存器，描述符装入投影寄存器。

2. 读出描述符例程

(1) 例程名：ReadDescriptor( )

(2) 例程参数：

- Selector　段描述符的选择子；
- Attributes　向调用者返回段描述符属性的地址；
- Base　向调用者返回段描述符基地址的地址；
- Limit　向调用者返回段描述符界限的地址；
- GSelector　向调用者返回描述符选择子的地址；
- GOffset　向调用者返回描述符偏移量的地址。

(3) 程序功能：根据选择子给出的索引，从指定描述符表中读出段描述符，并装入描述符投影寄存器。

(4) 程序清单：

```
/* 从描述符表读出描述符的例程 */
/* 返回属性字段、基址、界限、门选择子及偏移 */
/* 调用者必须检查属性字段，以确定使用基址及界限还是使用门选择子及偏移 */
ReadDescriptor(Selector,Attributes,Base,Limit,GSelector,GOffset)
        SelectorType Selector;      /* 描述符的选择子 */
        SegAttributes *Attributes;      /* 存放段属性的地址 */
        int           *Base,      /* 存放段基址的地址 */
                      *Limit;      /* 存放段界限的地址 */
        SelectorType  *GSelector;      /* 存放门的选择子的地址 */
        int           *GOffset      /* 存放门的偏移量的地址 */
{
  int DTBase,Dword1,Dword2;
  if(Selector.TI==1) {      /* 局部描述符表 */
    if((LDTR.Attributes.P==0)
        ||(((Selector.Index*8)+7)>LDTR.Limit))
    /* 局部描述符表不存在，或描述符越界 */
        SegmentException($GP,Selector);      /* 出错码为选择子 */
      DTBase=LDTR.Base;
  }
  else{      /* 全局描述符表 */
      if(((Selector.Index *8)+7)>GDTR.Limit)      /* 描述符越界 */
          SegmentException($GP,Selector);      /* 出错码为选择子 */
      DTBase=GDTR.Base;
```

```
    }
    /* 读出描述符 */
    AccessLinear(DTBase+Selector.Index*8,4,0,0&Dword1);
                                        /* 特权级0读描述符第一个双字 */
    AccessLinear(DTBase+Selector.Index*8+4,4,0,&Dword2);
                                        /* 特权级0读描述符第二个双字 */
    /* 将分离存放的描述符基址、界限合并还原 */
    *Limit=(Dword1&0FFFFh)|(Dword 2 & 0F0000h);
    *Base=(Dword1≫16)|(Dword2 & 0FFh)≪16)
                        |(Dword2 & 0FF000000h);
    *Attributes=Dword2;
    if(*Attributes.G==1)
      *Limit=(*Limit<<12)|0FFFh;      /*4K 粒度时的界限处理*/
    /* 将门选择子及偏移量合并还原 */
    *GSelector=Dword1≫16;
    *GOffset=(Dword1&0FFFFh)|(Dword 2 & 0FFFF0000h);
}
```

ReadDescriptor( )程序的流程，见图4.34。

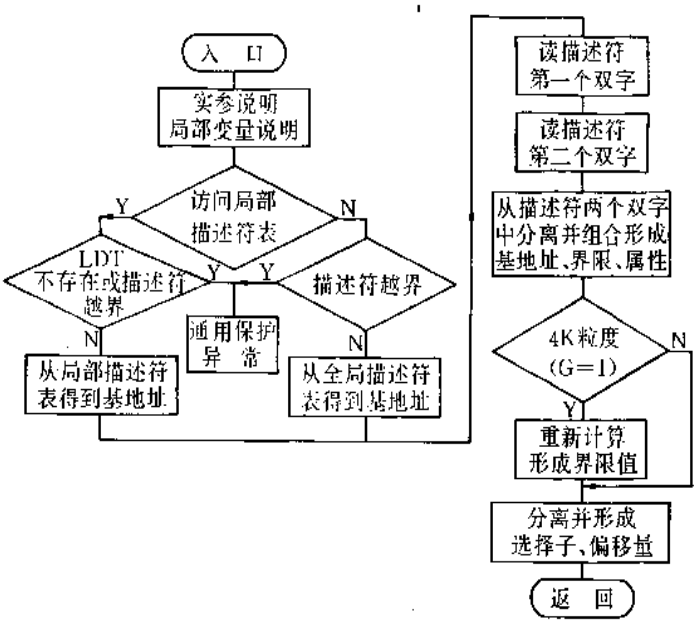


图 4.34 ReadDescriptor 程序流程

(5) 执行过程：

第一，检查由选择子索引字段指定的描述符是否部分地、甚至整个地超出了描述符表界限的范围。如超出指定描述符表界限的范围，则产生异常。若指定描述符表为局部描述符表，则还要检查局部描述符表是否为空。如为空，也将产生异常。这两种情况产生的异常，都把选择子作为出错码给出。

第二，计算描述符表项所在的线性地址，使用特权级0，分两次调用 AccessLinear( )读出两个描述符双字。

最后，分离描述符双字，并组合成描述符的基地址、界限、属性、门选择子及门偏移量送描述符投影寄存器并返回给主调程序。

在读出描述符时，发生页异常是可能的。这是因为描述符表所在的段与其它的段一样，也存储在线性地址空间中。这里还要提请读者注意，在 ReadDescriptor( )例程中，调用 AccessLinear( )例程读出描述符时，使用的是特权级0作为访问线性地址的读出特权级。这表明，所有对描述符表的读出操作，不是使用运行程序的特权级(CPL)，而是在特权级为0的情况下进行的。因此，描述符表可以存储在3级程序不能访问的页中，但允许3级程序把描述符表中的描述符装入到段寄存器中。

在第3章测试段属性指令的详细说明中，使用了取描述符例程。该例程与读描述符例程很相似，所以在这里做一对比介绍。

(1) 例程名：FetchDescriptor( )

(2) 例程参数：

- Selector　需取出的描述符的选择子；
- Dword1　取出的描述符的第一个双字；
- Dword2　取出的描述符的第二个双字；
- Attributes　返回属性。

(3) 程序功能：根据选择子给出的索引，从指定描述符表中读出段描述符。

(4) 程序清单：

```
/* 从描述符表取(读)描述符，正常时返回1，有问题返回0 */
/* 描述符的两个双字原样返回给主调程序 */
/* 描述符的第二个双字同时作为结构类型返回 */
FetchDescriptor(Selector,Dword1,Dword2,Attributes)
  SelectorType Selector;    /* 选择子 */
  int          *Dword1, * Dword2;    /* 存放描述符两个双字的地址 */
  SegAttributes * Attributes;    /* 存放属性的地址 */
{
  int DTBase;
  if(Selector. TI==1) {    /* 局部描述符表 */
    if(LDTR. Attributes. P==0)    /* 局部描述符表不存在 */
      ||(((Selector. Index * 8)+7)>LDTR. Limit))    /* 描述符表越界 */
      return(0);
    DTBase=LDTR. Base;
  }
  else {    /* 全局描述符表 */
    if(((Selector. Index * 8)+7)>GDTR. Limit)    /* 描述符表越界 */
        return(0);
    DTBase=GDTR. Base;
  }
  /* 读出描述符 */
  AccessLinear(DTBase+Selector. Index * 8,4,0,0,&Dword1);
            /* 使用特权级0读出描述符第一个双字 */
  AccessLinear(DTBase+Selector. Index * 8+4,4,0,0,&Dword2);
            /* 使用特权级0读出描述符第二个双字 */
  * Attributes=Dword2;
  Return(1)
}
```

FetchDescriptor( )的流程见图4.35。

(5) 执行过程：

首先检查由选择子索引字段指定的描述符是否超出描述符表的范围，如超出，则向主调程

序 返回0，以表示出错。如指定的描述符表为局部描述符表，则还要检查局部描述符表是否为空。如为空，也返回0表示出错。第二，用描述符表的基地址与选择子中的索引字段的值，计算描述符的线性地址，使用特权级0，分两次调用 AccessLinear ( )读出描述符双字。最后，向主调程序返回1，表示取描述符例程正常完成。描述符的属性及两个双字，则从调用函数的参数地址带回。

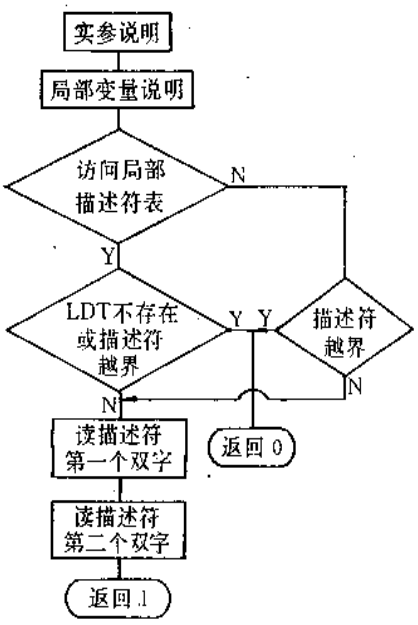


图 4.35 FetchDescriptor( )程序流程

通过比较可以发现，读描述符与取描述符这两个例程有如下两点区别：

① 如不能读出描述符，读描述符例程将产生异常；而取描述符例程则将不产生异常，并返回0，表示出错。

② 读描述符例程要将读出的两个描述符双字分解成基地址、界限等各种字段，并装入描述符投影寄存器；取描述符例程则直接返回取出的描述符双字。

3. 设置描述符访问属性例程

(1) 例程名：SetAccessed( )

(2) 例程参数：Selector 选择子

(3) 程序功能：对选择子指定的描述符设置访问属性位。

(4) 程序清单：

```
/*描述符被访问之后，在段描述符中设置描述符的访问位 */
SetAccessed(Selector)
  SelectorType Selector;      /*选择子*/
{
  int DTBase,Dword2;
  /*读出描述符的第二个双字，将描述符的访问位，即类型字 */
  /* 段的最低位置1后写回，类型检查由调用者进行          */
  if(Selector.TI==1){        /* 描述符在局部描述符表 */
    if(LDTR.Attributes.P==0
        ||(((Selector.Index*8)+7)>LDTR.Limit))
    /*局部描述符表不存在或描述符表越界 */
        SegmentException($GP,Selector);      /*出错码为选择子*/
    DTBase=LDTR.Base;
  }
  else {      /*描述符在全局描述符表*/
    if(((Selector.Index*8)+7)>GDTR.Limit)     /*描述符表越界*/
      SegmentException($GP,Selector);      /*出错码为选择子*/
    DTBase=GDTR.Base;
  }
  AccessLinear(DTBase+Selector.Index*8+4,4,0,0,&Dword2);
              /*使用特权级0读出描述符的第二个双字 */
  Dword2=Dword2|100h;      /*设置访问属性 */
  AccessLinear(DTBase+Selector.Index*8+4,4,0,1,&Dword 2);
              /*使用特权级0将修改了访问位后的第二个双字写回 */
```

}

SetAccessed( )程序流程见图4.36。

(5) 执行过程：

第一，选择子测试，检查选择子指定描述符是否越界。如指定局部描述符表，则局部描述符表不能为空。以上测试如失败，将产生异常，并将选择子作为出错码给出。

第二，用描述符表的基地址与选择子索引字段的值，计算描述符的线性地址，使用特权级0读出描述符的第二个双字(包含属性的双字)。

第三，将描述符属性中的访问位置为1，再使用特权级0，将已修改访问位后的描述符的第二个双字写回原处，从而完成本例程。

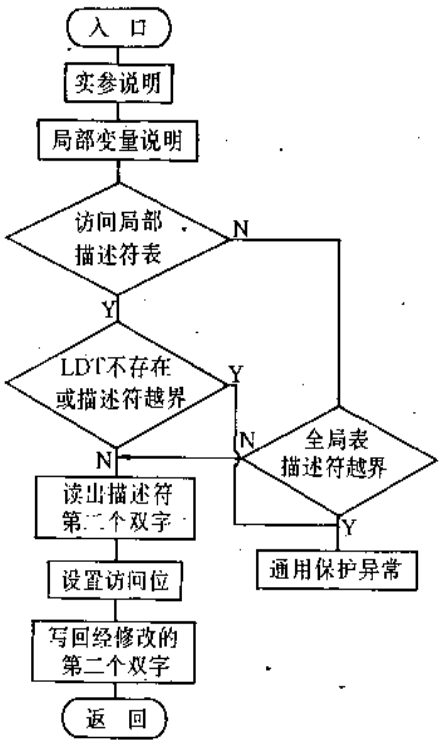


图 4.36 SetAccessed( )程序流程

4.7.2.4 用 LLDT 指令装入 LDTR 寄存器

局部描述符表是一个特别的系统段，该段最多可容纳$2^{13}=8K$ 个表项，每一表项都是一个描述符。所以，局部描述符表的段长为描述符字节数乘以表项数N(即8 * N)，最大长度为64K。访问局部描述符表通过访问局部描述符寄存器(LDTR)进行，该寄存器用 LLDT 指令(LLDT 例程)装入。

(1) 例程名：LLDT( )

(2) 例程参数：Selector 局部描述符表的描述符的选择子。

(3) 程序功能：装入局部描述符表寄存器

(4) 程序清单：

```
/*LLDT 指令的详细说明 */
LLDT(Selector)
  SelectorType Selector;
{
  SegAttributes Attributes;
  SelectorType GSelector;
  int          Base,Limit,GOffset;
  /*LLDT 指令只能在0级执行 */
  if(CPL!=0)
    SegmentException($GP,0);
  /*选择子测试*/
  if(Selector.TI==1)     /*局部描述符表的描述符必须在全局描述符表中，否则产生异常 */
    SegmentException($GP,0);
  if((Selector&0FFFCh)==0)
  /*将空选择子装入局部描述符表寄存器时，要将描述符的存在位置为0 */
  {
    LDTR.Selector=Selector;
    LDTR.Attributes.P=0;
    return;
  }
```

```
  /* 读出并测试描述符 */
  ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);
  if((Attributes.DType==1)||(Attributes.Type!=2))
  /*读出的描述符不是局部描述符表的描述符时产生异常 */
    SegmentException($GP,0);
  if(Attributes.P==0)
    SegmentException($NP,0);     /*不存在异常,出错码为0 */
  /*描述符装入投影寄存器,选择子装入 LDTR */
  LDTR.Attributes =Attributes;
  LDTR.Base       =Base;
  LDTR.Limit      =Limit;
  LDTR.Selector   =Selector;
}
```

LLDT 指令的程序流程见图4.37。

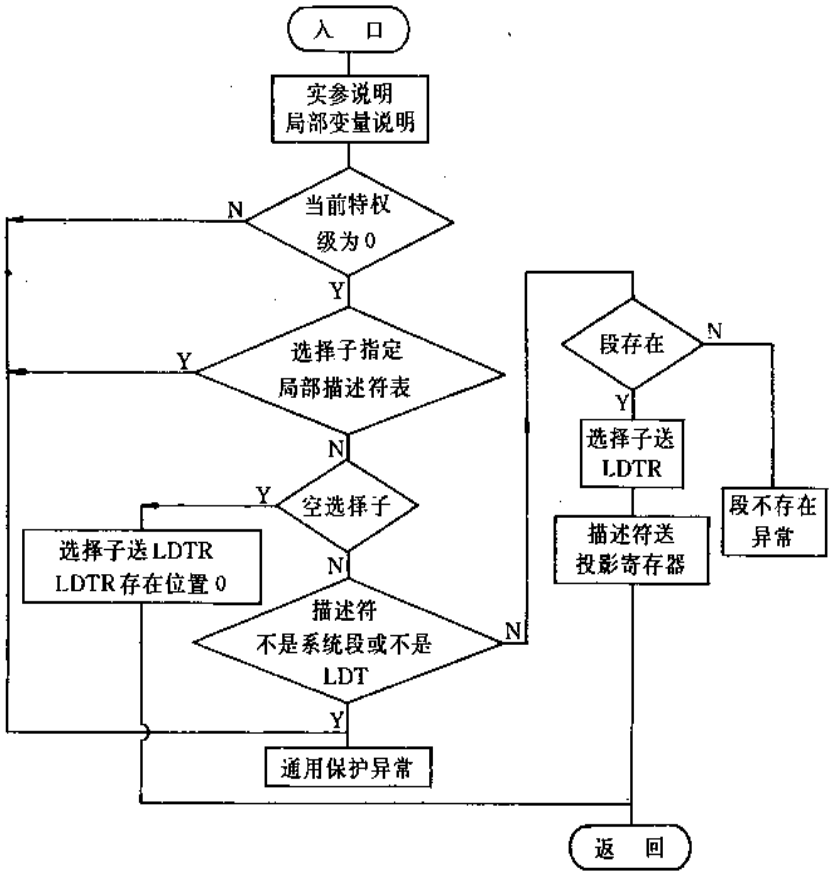


图 4.37 LLDT 指令的流程

(5) 执行过程:

第一,检查执行该指令的特权级是否为0级。LLDT 指令是特权指令,只能在0级执行。如在非0级执行该指令,则将产生异常。

第二,对选择子进行测试。局部描述符表的选择子,必须指示其描述符在全局描述符表中,否则产生异常。如选择子中的 Index 字段为0,则表示用空选择子装入 LDTR。用空选择子装入 LDTR 是允许的,这表示正执行的任务只有全局描述符表,没有局部描述符表。用空选择子装

入 LDTR 时，要把描述符的存在位置为0，表示正执行的任务没有局部描述符表这一特性，然后直接返回。

第三，如装入 LDTR 的不是空选择子，则调用 ReadDescriptor( )例程，将新的 LDT 段的描述符读出并进行测试：该描述符必须是一个系统段或门、类型为 LDT 并且存在。如果这些检查都通过，则把选择子装入到可见的 LDTR 选择子寄存器，描述符装入到 LDTR 投影寄存器。

4.7.2.5 用 LTR 指令装入 TR 寄存器

每当任务切换执行一个新任务时，需将新任务的任务状态段的选择子装入任务寄存器。新任务的任务状态段的描述符装入 TR 的投影寄存器中。

(1) 例程名：LTR( )

(2) 例程参数：Selector 新任务的任务状态段的选择子

(3) 程序功能：装入任务寄存器

(4) 程序清单：

```
/* LTR 指令的详细说明 */
LTR(Selector)
   SelectorType Selector;
{
  SegAttributes Attributes;
  SelectorType  GSelector;
  int           Base,Limit,GOffset;
  /* LTR 指令只能在0级执行 */
  if(CPL!=0)
    SegmentException($GP,0);
  /*描述符必须在全局描述符表，而且非空，否则产生异常*/
  if((Selector.TI==1)||((Selector&0FFFCh)==0))
      SegmentException($GP,0);
  /* 读出并测试描述符 */
  /* 描述符类型必须为可用的286TSS 或可用的386TSS */
  ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);
  if((Attributes.DType==1)
      ||(Attributes.Type!=1&&Attributes.Type!=9))
    SegmentException($GP,0);
  if(Attributes.P==0)
    SegmentException($NP,0);
  /*将描述符类型字段修改为忙的286TSS 或忙的386TSS */
  AccessLinear(GDT.Base+Selector.Index*8+4,4,0,0,&Dword2);
              /*使用特权级0读出描述符第二个双字*/
  Dword2=Dword2 |200h;     /*改为忙*/
  AccessLinear(GDT.Base+Selector.Index*8+4,4,0,1,&Dword2);
        /*使用特权级0写回描述符第二个双字*/
  /* 将描述符装入投影寄存器，选择子装入 TR */
  TR.Attributes =Attributes;
  TR.Base       =Base;
  TR.Limit      =Limit;
  TR.Selector   =Selector;
}
```

LTR 指令的流程见图4.38。

(5) 执行过程

第一，检查执行该指令的特权级是否为0级。LTR 指令是特权指令，只能在0级执行。如在

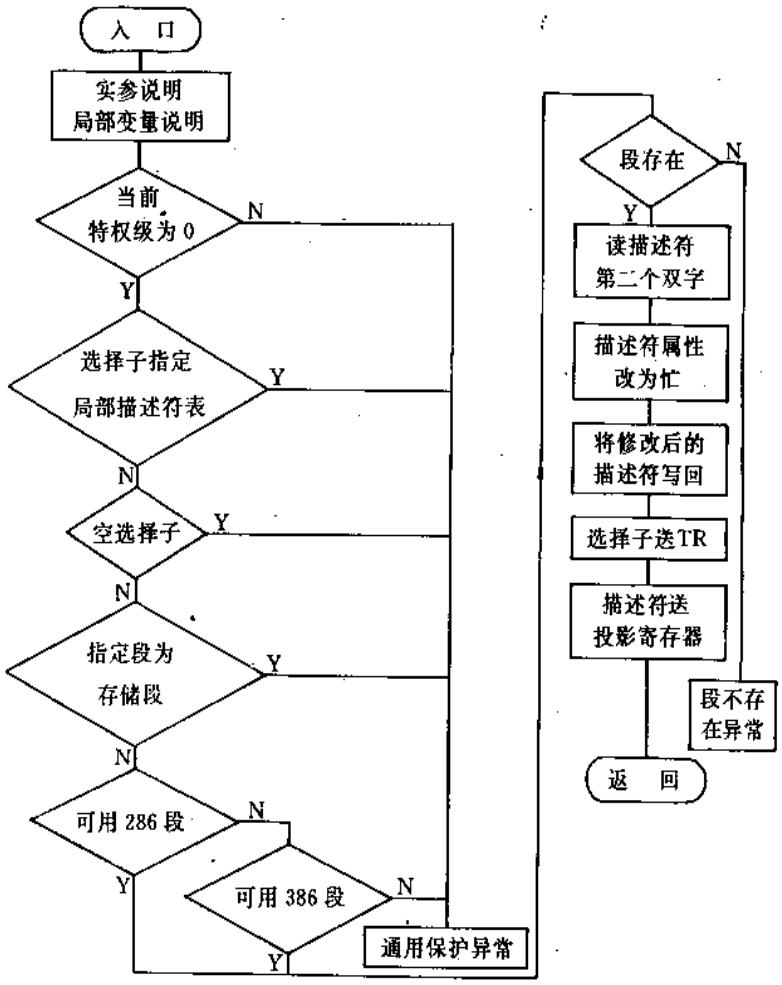


图 4.38 LTR 指令的流程

非0级执行该指令，则将产生异常。

第二，对选择子进行测试。任务状态段的选择子必须指示其描述符在全局描述符表中，同时选择子不能为空，否则将产生异常。

第三，调用 ReadDescriptor( )例程，读出任务状态段的描述符，并对描述符进行测试：该描述符必须是一个系统段或门、类型为可用286TSS 段或可用386TSS 段、并且存在位为1。否则，将产生异常。

第四，计算描述符的线性地址，调用 AccessLinear( )例程，读出描述符中包含属性位的双字，将该任务状态段标记为“忙”，再将此双字写回原处。

最后，将选择子装入 TR 寄存器，描述符装入 TR 投影寄存器，完成此例程。

4.7.3 控制转移

本节用类似 C 语言表示法的程序，描述段间控制转移的每一条指令。在这些指令的说明中，利用了刚刚叙述过的分页及段寄存器装入的某些例程。

控制转移包含有跳转、调用及返回等类型。成功的控制转移表明，系统将从正执行的代码段转去执行另一代码段，并且需要用新的代码段描述符装入 CS 投影寄存器。CS 描述符装入

例程，就用来完成各种控制转移类型都需要的这一功能。下面就分别叙述各类控制转移的例程。

4.7.3.1 CS 描述符装入例程

(1) 例程名：CSDescriptorLoad()

(2) 例程参数：

- ·Selector　要装入的描述符的选择子
- ·Attributes　要装入的描述符的属性
- ·Base　要装入的描述符的基地址
- ·Limit　要装入的描述符的界限
- ·GPorTS　异常情况时的异常类型

(3) 程序功能：把控制转移后的代码段描述符装入 CS 投影寄存器。

(4) 程序清单：

```
/* 把描述符装入 CS 投影寄存器的公共例程。本例程不装入 CS 的 */
/* 选择子，CS 选择子需由调用者另行装入      */
CSDescriptorLoad(Selector,Attributes,Base,Limit,GP or TS)
  SelectorType   Selector;
  SegAttributes  Attributes;
  int            Base,Limit,
                 GP or TS;     /* 除不存在异常外，均为 $GP 或 $TS 异常 */
{
  switch(Attributes.Type) {
    case8:case9:     /* 只可执行的代码段 */
    case10:case11:     /* 可执行可读的代码段 */
      /* 非一致代码段的特权级检查 */
      if((Attibutes.DPL!=CPL)||(Attributes.DPL<Selector.RPL))
        SegmentException(GP or TS,Selector);     /* 出错码为选择子 */
      break;
    case12:case13:     /* 只可执行的一致代码段 */
    case14:case15:     /* 可执行可读的一致代码段 */
      /* 一致代码段的特权级检查 */
      if(Attributes.DPL>CPL)
        SegmentException(GP or TS,Selector);     /* 出错码为选择子 */
      break;
    case0:case1:  /* 只读数据段 */
    case2:case3:  /* 可读写数据段 */
    case4:case5:  /* 只读向下扩展数据段 */
    case6:case7:  /* 可读写向下扩展数据段 */
      /* 类型错误，产生异常 */
      SegmentException(GP or TS,0);     /* 出错码为0 */
  }
  /* 检查存在位 */
  if(Attributes.P==0)
    SegmentException($NP,Selector);     /* 不存在异常，出错码为选择子 */
  /* 如所有检查通过，则程序往下执行，设置访问位， */
  /* 把描述符装入投影寄存器   */
  SetAccessed(Selector);
  CS.Attributes =Attributes;
  CS.Base       =Base;
  CS.Limit      =Limit;
}
```

CSDescriptorLoad( )的程序流程见图4.39。

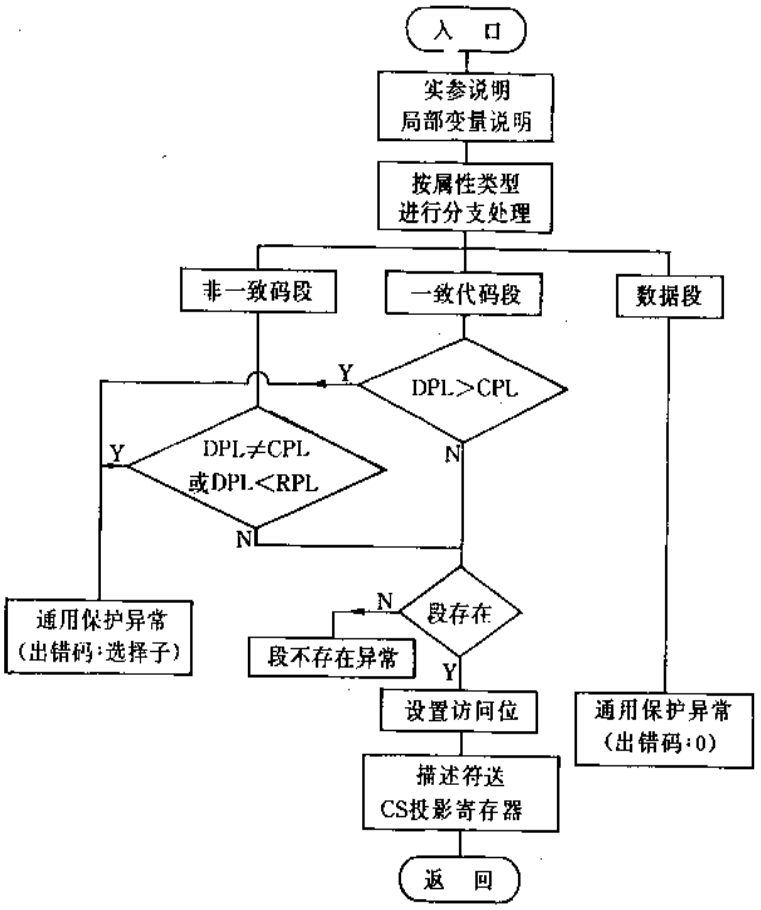


图 4.39 CSDescriptorLoad( )的程序流程

(5) 执行过程:

第一,对描述符的属性类型进行测试:

① 类型中必须有执行权限,没有执行权限的各种描述符属性类型都将产生异常;

② 可执行的非一致代码段,必须有 DPL=CPL,且有 DPL≥RPL,否则将产生异常;

③ 可执行的一致码段,必须有 DPL≤CPL,否则将产生异常。

第二,检查描述符的存在位。该位必须为1,否则将产生异常。如果上述检查通过,则调用SetAccessed( )例程,设置描述符的访问位。

最后,将描述符的属性、基地址、界限装入描述符投影寄存器完成本例程。

4.7.3.2 跳转指令

跳转指令可以实现直接的段间控制转移,也可通过调用门实现间接的段间控制转移,还可以通过任务状态段或任务门,实现任务间的切换。跳转指令是80386指令集中、比较复杂而又很重要的指令。跳转指令的例程及子例程包括跳转和通过门的跳转。

1. 普通跳转

(1) 例程名:JMP( )

(2) 例程参数:

·Selector 跳转的目标选择子

·Offset　跳转的目标偏移量

(3) 程序功能:实现程序的控制转移。

(4) 程序清单:

```
/* 跳转指令的详细说明 */
JMP(Selector,Offset)
  SelectorType Selector;
  int          Offset;
{
  SegAttributes Attributes;
  SelectorType  GSelector;
  int           Base,Limit,GOffset;
  /* 选择子为空则产生异常 */
  if((Selector & 0FFFCh)==0)
    SegmentException($GP,0);
  /* 读出并测试描述符 */
  ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);
  if(Attributes.DType==1){     /* 描述符为存储段 */
    /* 调用公共例程装入CS描述符 */
    CSDescriptorLoad(Selecor,Attributes,Base,Limit,$GP);
    /* 检查目标是否越界 */
    if(Offset>CS.Limit)
      SegmentException($GP,0);
    /* 选择子装入CS寄存器(末两位保持不变) */
    CS.Selector=Selector;
    CS.Selector.RPL=CPL;
    EIP          =Offset;
  }
  else{     /* 描述符为系统段或门 */)
    /* 特权级检查 */
    if((Attributes.DPL<CPL)||(Attributes.DPL<Selector.RPL))
      SegmentException($GP,Selector);     /* $GP 异常,出错码为选择子 */
    switch(Attributes.Type){
    case1:     /* 描述符属性类型为可用286TSS */
        if(Attributes.P==0)
          SegmentException($NP,Selector);/* $NP 异常,出错码为选择子 */
        TaskSwitch286(Selector,Attributes,Base,Limit,0);
                          /* 最后一个参数0表示无链接 */
        break;
    case5:     /* 属性类型为任务门 */
      if(Attributes.P==0)
          SegmentException($NP,Selector);     /* $NP 异常,出错码为选择子 */
      TaskGate(GSelector,0)     /* 无链接 */
      break;
    case9:     /* 属性类型为可用386TSS */
      if(Attributes.P==0)
        SegmentException($NP,Selector);     /* $NP 异常,出错码为选择子 */
      TaskSwitch(Selector,Attributes.Base,Limit,0);     /* 无链接 */
      break;
    case4:     /* 属性类型为286调用门 */
        if(Attributes.P==0)
            SegmentException($NP,Selector);
        JumpGate286(GSelector,GOffset,$GP);
        break;
    case12:     /* 属性类型为386调用门 */
```

```
        if(Attributes.P==0)
          SegmentException($NP,Selector);
        JumpGate(GSelector,GOffset,$GP);
        break;
        Default:    /*其它属性类型时产生异常 */
          SegmentException($GP,Selector);
    }
  }
}
```

JMP 指令的程序流程见图4.40。

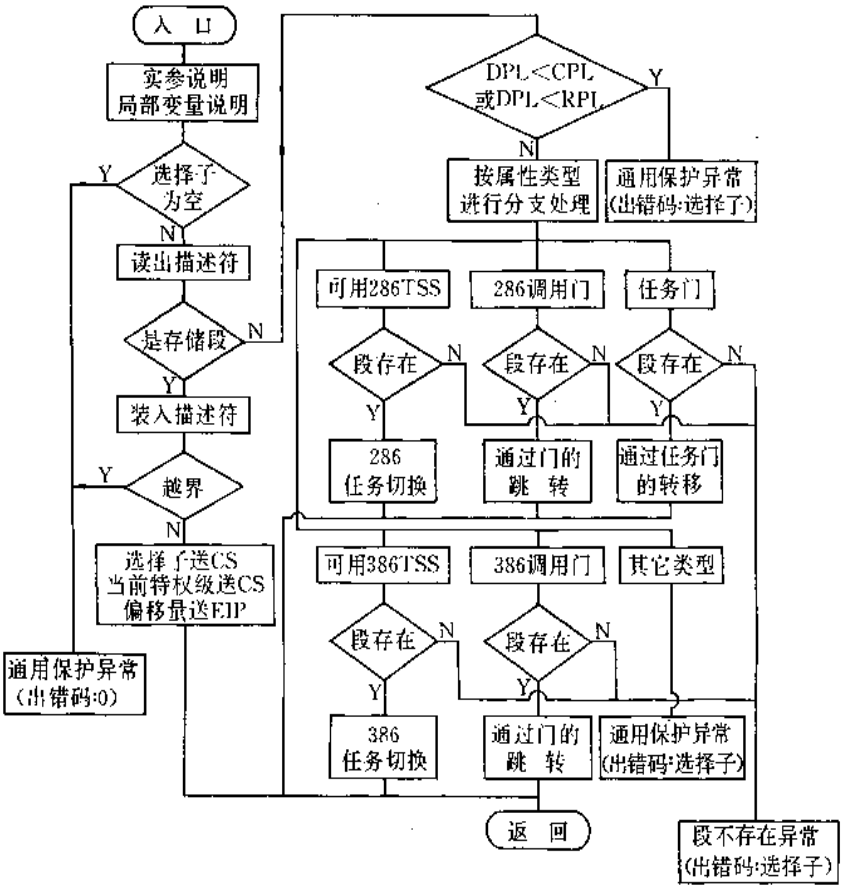


图 4.40 JMP 指令的程序流程

(5) 执行过程：

第一，对选择子进行测试。因为选择子直接地或间接地指向跳转目标，所以跳转指令中的选择子参数的索引字段的值不能为0，否则，被认为是空选择子，因为得不到跳转目标而产生异常。

第二，调用 ReadDescriptor( )例程读出并测试描述符。如果描述符属性中的 DT 位为1，则表明此次跳转是直接的段间转移，并调用 CSDescriptorLoad( )例程，完成段寄存器的装入。

最后如偏移量未超出描述符中的界限值，则将选择子装入 CS 寄存器。令 CS 中的 RPL 等于当前特权级 CPL，再把偏移量置入程序计数器 EIP，完成段间跳转的操作。

如果描述符属性中的 DT 位为0，则表明选择子所对应的描述符是系统段或门，即此次跳转要通过调用门进行间接的段间转移或通过任务状态段或任务门实现任务切换。这时，首先要

检查描述符属性字段中的 DPL 与 CPL 和 RPL 之间的关系，并且必须满足

$$DPL \geqslant CPL \ \&\& \ DPL \geqslant RPL$$

否则，因违反控制转移必须遵守的特权级规则而产生异常。在满足上式关系的情况下，根据描述符属性中类型字段的值执行如下操作

① 如类型字段的值表示为可用的286任务状态段，则调用 TaskSwitch286( )例程完成任务切换(见第7章)；

② 如类型字段的值表示为可用的386任务状态段，则调用 TaskSwitch 例程完成任务切换；

③ 如类型字段的值表示为任务门，则调用 TaskGate( )例程完成任务切换；

④ 如类型字段的值表示为286调用门，则调用 JumpGate286( )例程完成通过调用门的间接控制转移(见第7章)；

⑤ 如类型字段的值表示为386调用门，则调用 JumpGate( )例程完成通过调用门的间接控制转移；

⑥ 如类型字段的值不表示上述情况中的任何一种情况，则表示发生异常。

除最后一种情况外，无论是通过系统段或任务门的任务切换、还是通过调用门的间接转移，都要对描述符的存在位进行检查。如果存在位为0，都要产生异常。

2. 通过调用门的跳转

本例程是跳转指令的子例程。在跳转指令中，如描述符的类型是一个286调用门或386调用门，则跳转指令要调用 JumpGate286( )或 JumpGate( )完成通过调用门的跳转。此处要介绍的是通过386调用门的跳转，而通过286调用门的跳转留待第7章讨论和比较。

(1) 例程名：JumpGate( )

(2) 例程参数：

- Selector　调用门中的选择子
- Offset　调用门中的偏移量
- GP or TS 异常类型

(3) 程序功能：实现通过调用门的跳转。

(4) 程序清单：

```
/* 通过调用门的跳转 */
JumpGate(Selector,Offset,GP or TS)
  SelectorType Selector;
  int          Offset;
  int          GP or TS;     /*除不存在异常外,其它情况为$GP或$TS异常*/
{
  SegAttributes Attributes;
  SelectorType  GSelector;
  int           Base,Limit,GOffset;
  /*选择子测试如为空则产生异常*/
  if((Selector&0FFFCh)==0)
    SegmentException(GP or TS,0);
  /*读出并测试描述符 */
  ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);
  if(Attributes.DType==0)     /* 描述符不能是系统段或门*/
    SegmentException(GP or TS,Selector);
  /*装入CS描述符 */
```

```
  Selector.RPL=0;      /*忽略门中选择子的请求特权级*/
  CSDescriptorLoad(Selector,Attributes,Base,Limit,GP or TS);
  /*检查是否越界 */
  if(Offset>CS.Limit)
    SegmentException(GP or TS,0);
  /*选择子装入CS寄存器(末两位保持不变) */
  CS.Selector       =Selector;
  CS.Selector.RPL=CPL;
  EIP               =Offset;
}
```

JumpGate( )的程序流程见图4.41。

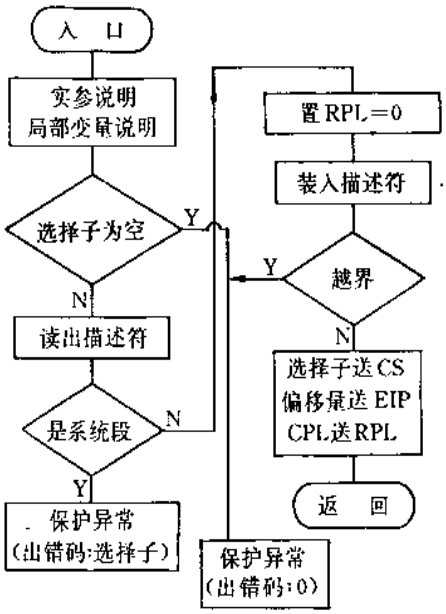


图 4.41 JumpGate( )的程序流程

(5) 执行过程:

第一,对选择子进行测试。由于该选择子是跳转指令读出的描述符(调用门)的选择子,必须指向一个目标码段的描述符,因此,该选择子必须非空。

第二,从指定描述符表中读出描述符(通过调用 ReadDescriptor( )例程完成。由于 JumpGate( )例程不支持门描述符的嵌套,所以此描述符不能是系统段或门,否则将产生异常。如果描述符的 DT 字段表示一个存储段,则将其传送给公共例程 CSDescriptorLoad( )例程。因为调用门中选择子的 RPL 字段要被忽略,所以在调用 CS 描述符装入例程之前,要将 RPL 字段置为0,使得在 CS 描述符装入例程中进行特权级测试时,由于 RPL=0,DPL≥RPL 恒为真,使对 RPL 的测试得以通过。最后,测试偏移量是否越界,如未越界,则将选择子装入 CS 寄存器,令 CS 寄存器中的 RPL=CPL,再将调用门中的偏移量置入程序计数器 EIP 完成本例程。

4.7.3.3 调用指令

从结构上看,调用指令与前述的跳转指令是两条非常类似的指令。这两条指令有一些共同的功能:实现程序的控制转移;通过任务状态段或任务门实现任务切换;通过调用门实现间接的转移。但这两者也有区别,基本的不同点是前者是带返回的控制转移,而后者不带返回。为了

实现带返回的控制转移，在调用指令中执行实现控制转移的操作之前，要预先做好返回安排，而在通过系统段或任务门进行任务切换时，要进行带链接的切换。调用指令有关的例程及子例程有调用例程、通过门的调用、栈切换及任务切换。除任务切换放在4.7.4专门介绍外，其余例程和子例程都在此逐一介绍。

1. 调用子程序的例程。

(1) 例程名：CALL( )

(2) 例程参数：·Selector　被调程序段描述符的选择子

·Offset　被调程序的入口偏移量

(3) 程序功能：实现带返回的控制转移

(4) 程序清单：

```
/* 段间调用指令的详细说明 */
CALL(Selector,Offset)
   SelectorType Selector;
   int          Offset;
{
     SegAttributes Attributes;
     SelectorType  GSelector;
     int           Base,Limit,GOffset;
  /* 选择子测试，为空时产生异常 */
  if((Selector&0FFFCh)==0)
    SegmentException($GP.0);
  /* 读出并测试描述符 */
  ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);
  if(Attributes.DType==1){      /* 描述符类型为存储段 */
    /* 调用公共例程装入 CS 描述符 */
    CSDescriptorLoad(Selector,Attributes,Base,Limit,$GP);
    /* 越界检查 */
    if(Offset>CS.Limit)
      SegmentException($GP,0);
    /* 返回地址 CS:EIP 压栈 */
    ESP--=4;      /* 即 ESP=ESP-4 */
    AccessVirtual(SS,ESP,4,1,$SS,&CS.Selector);      /* 写入4个字节：两个高字节为0，低字节为
                                                        CS 选择子 */
    ESP--=4;
    AccessVirtual(SS,ESP,4,1,$SS,EIP);
    /* 选择子装入 CS 寄存器(末两位保持不变)，偏移量装入指令指针寄存器 */
                                                                        */
    CS.Selector       =Selector;
    CS.Selector.RPL   =CPL;
    EIP               =Offset;
  }
  else{      /* 描述符类型为系统段或门 */
    /* 特权级检查 */
    if((Attributes.DPL<CPL)||(Attributes.DPL<Selector.RPL))
      SegmentException($GP,Selector);
    switch(Attributes.Type) {
      case1:      /* 描述符属性类型为可用的286TSS 段 */
        if(Attributes.P==0)
          SegmentException($NP,Selector);
        TaskSwitch286(Selector,Attributes,Base,Limit,1);      /* 带链接 */
        break;
```

```
        case5:     /* 描述符属性类型为任务门 */
          if(Attributes.P==0)
            SegmentException($NP,Selector);
          TaskGate(GSelector,1);     /* 带链接 */
          break;
        case9:     /* 描述符属性类型为可用的386TSS */
          if(Attributes.P==0)
            SegmentException($NP,Selector);
          TaskSwitch(Selector,Attributes,Base,Limit,1);     /* 带链接 */
          break;
        case 4:     /* 描述符属性类型为286调用门 */
          if(Attributes.P==0)
            SegmentException($NP,Selector);
          CallGate286(GSelector,GOffset,Attributes,DwordCount);
          break;
        case12:     /* 描述符属性类型为386调用门 */
          if(Attributes.P==0)
            SegmentException($NP,Selector);
          CallGate(GSelector,GOffset,Attributes,DwordCount);
          break;
      Default:     /* 其它属性类型时产生异常 */
        SegmentException($GP,Selector);
      }
    }
  }
```

CALL 指令的程序流程见图4.42。

(5) 执行过程

首先,对选择子进行测试,选择子必须非空,并以该选择子为参数调用 ReadDescriptor( )例程读出描述符,再根据对描述符的测试可能有的两种情况继续执行:

第一种情况,描述符属性中的 DT 位为1。这时,调用 CSDescriptorLoad( )例程,把描述符装入 CS 投影寄存器,再检查指令中偏移量参数的值与 ReadDescriptor( )例程读出的描述符的偏移量界限值之间的关系,即检查与 CS 投影寄存器中的偏移量之间的关系。如指令中偏移量参数的值≤CS 投影寄存器中偏移量界限的值,表示未发生越界。接下去的动作是返回点压栈、选择子装入 CS、CPL 送 CS、Offset 送程序计数器而完成本例程。与 JMP 指令相比,由于 CALL 在执行过程中增加了返回点压栈的动作,因此,在 CALL 指令的控制转移目标代码段(即被调用代码段)执行完毕之后,只需执行一个弹栈操作(RET 指令中介绍),即可返回到 CALL 指令的下一条指令。

第二种情况,描述符属性中 DT 位为0,即指令中选择子指定的描述符是系统段或门。这时,CALL 指令要通过调用门进行间接调用或通过任务门进行任务切换。和 JMP 相同,描述符特权级、选择子特权级和当前特权级必须满足

$$DPL \geqslant CPL \ \&\& \ DPL \geqslant RPL$$

在满足此关系式的情况下,根据描述符属性中类型字段的值执行如下操作:

① 如类型字段的值表示为可用的286任务状态段,则调用 TaskSwitch286( )例程,进行带链接的任务切换(见第7章);

② 如类型字段的值表示为可用的386任务状态段,则调用 TaskSwitch( )例程,进行带链接的任务切换;

③ 如类型字段的值表示为任务门,则调用 TaskGate( )进行带链接的任务切换;

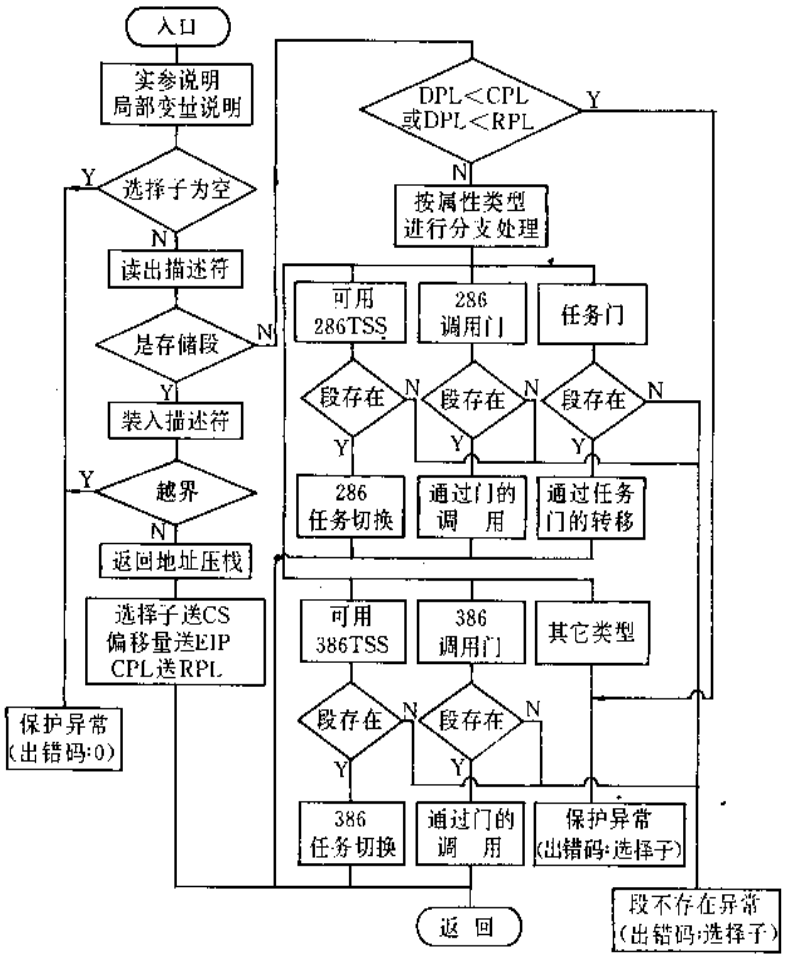


图 4.42 CALL 指令的程序流程

④ 如类型字段的值表示为286调用门,则调用 CallGate286( )例程进行带参数的间接调用(见第7章);

⑤ 如类型字段的值表示为386调用门,则调用 CallGate( )例程进行带参数的间接调用;

⑥ 如类型字段的值不表示上述情况中的任何一种情况,则表示发生异常。

除最后一种情况外,无论是通过系统段或任务门的任务切换,还是通过调用门的间接调用,都要对描述符的存在位进行检查。如果存在位为0,都要产生异常。

2. 通过调用门的调用子程序例程

本例程是 CALL 指令的子例程。在 CALL 指令中,如描述符的类型是一个286调用门或386调用门,则 CALL 指令要调用 CallGate286( )或 CallGate( )完成通过调用门带传递参数的调用。此处介绍的是通过386调用门的调用,而通过286调用门的调用留待第7章讨论。

(1) 例程名:CallGate( )

(2) 例程参数:

·Selector 调用门中的选择子

·Offset 调用门中的偏移量

·DwordCount 要求进行堆栈切换时,在堆栈间传送参数的双字个数。

(3) 程序功能:实现通过调用门的间接调用。

(4) 程序清单：

```
/* 通过调用门的 CALL 例程 */
CallGate(Selector,Offset,DwordCount)
        SelectorType Selector;
        int          Offset;
        int          DwordCount;     /* 要向内层调用传递的双字参数的数量 */
{
  SegAttributes Attributes;
  SelectorType GSelector;
  int          Base,Limit,GOffset;
  /* 选择子测试,为空时产生异常 */
  if((Selector&0FFFCh)==0)
      SegmentException($GP,0);
  /* 读出并测试描述符 */
  ReadDescriptor(Selector,&Attributes,&Base,&Limit,*GSelector,GOffset);
  if(Attributes.DType==0)     /* 描述符类型为系统段或门时产生异常 */
    SegmentException($GP,Selector);
  /* 调用内层非一致码段时,需切换到内层栈及内层 CPL */
  if((Attributes.Type>=8)&&(Attributes.Type<=11)
     &&(Attributes.DPL<CPL)&&(Attributes.P==1))
         InnerStack(Attributes.DPL,DwordCount,Selector);
  /* 装入 CS 描述符 */
  Selector.RPL=0;     /* 忽略门中选择子的 RPL 的影响 */
  CSDescriptorLoad(Selector,Attributes,Base,Limit,$GP);
  /* 越界检查 */
  if(Offset>CS.Limit)
    SegmentException($GP,0);
  /* 保护检查通过后,继续执行以下程序 */
  /* 返回地址 CS:EIP 压栈,压栈时,CS 放低16位 */
  ESP--=4;
  AccessVirtual(SS,ESP,4,1,$SS,&CS.Selector);     /* CS 压栈 */
  ESP--=4;
  AccessVirtual(SS,ESP,4,1,$SS,&EIP);     /* EIP 压栈 */
  /* 装入 CS 选择子及 EIP */
  CS.Selector       =Selector;
  CS.Selector.RPL   =CPL;
  EIP               =Offset;
}
```

CallGate( )的程序流程见图4.43。

(5) 执行过程：

第一，对选择子进行测试。因为该选择子是调用指令读出的描述符(调用门)中的选择子，必须指向一个目标码段的描述符，因此，该描述符必须非空。

第二，调用 ReadDescriptor( )例程，从指定描述符表中读出描述符并对其进行测试。由于 CallGate( )例程不支持门描述符的嵌套，所以此描述符不能再是系统段或门，否则将产生异常。如果调用门规定了一个向内层栈的切换，即同时满足条件：

- 描述符属性类型在8～11范围之内；
- 描述符特权级小于当前特权级；
- 描述符对应的段存在。

则调用 InnerStack( )例程，进行参数的拷贝和栈的切换。

第三，设置 RPL=0，忽略 RPL 在特权级测试中的作用，并调用 CSDescriptorLoad( )例

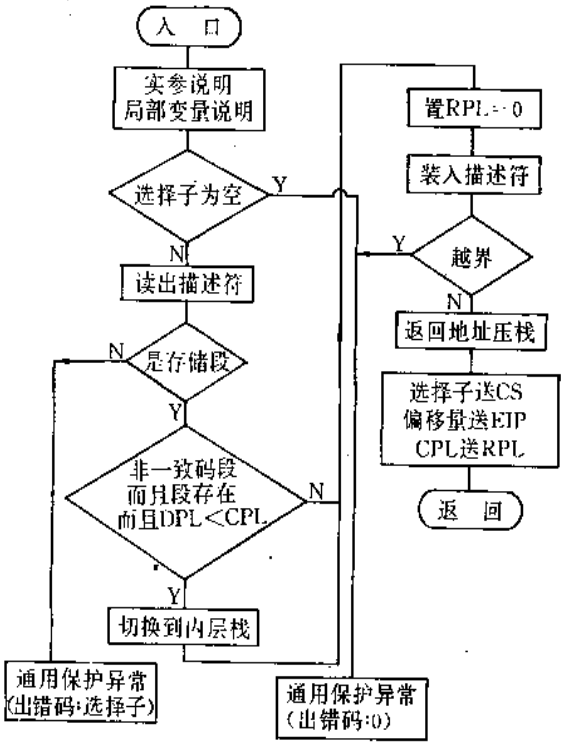


图 4.43 CallGate( )的程序流程

程,将描述符装入 CS 描述符投影寄存器。最后,测试转移目标偏移量是否越界?如未越界,则将返回地址压栈(CS 选择子及程序计数器)、用新的 CS 选择子装入 CS 寄存器、用控制转移目标偏移量装入程序计数器完成本例程。

3. 向内层的堆栈切换

主调程序通过调用门调用内层的函数过程时,要发生特权级的转移。在80386系统中,每一特权级都有自己的堆栈,有关信息都保存于任务状态段中。所以,在发生特权级转移时,还要同时进行与特权级转移相对应的堆栈切换。

(1) 例程名:InnerStack( )

(2) 例程参数:

- NewCPL 调用门的目标段的描述符特权级
- DwordCount 堆栈之间拷贝参数的双字数
- NewSelector 调用门中的选择子

(3) 程序功能:实现向内层栈的切换

(4) 程序清单:

```
/* 内层转移的栈切换例程 */
InnerStack(NewCPL,DwordCount,NewCSSelector)
int          NewCPL;      /* 切换后的特权级 */
int          DwordCount;      /* 拷贝到新栈的双字参数个数 */
SelectorType NewCSSelector;      /* 新的 CS 选择子 */
{
  SelectorType     NewSSSelector;
  SegmentRegister OldSS;
```

```
int                TSSOffset,NewESP,tempESP,OldCPL,Dword1;
/* 从 TSS 中读出新栈的 SS 及 ESP */
TSSOffset=NewCPL*8+4;
if((TSSOffset+7)>TR.Limit)
    SegmentException($TS,NewCSSelector);
AccessLinear(TR.Base+TSSOffset,0,0,&NewESP);
AccessLinear(TR.Base+TSSOffset+4,0,&NewSSSelector);
/* 保存原来的 SS 及 CPL 供参数拷贝时用 */
OldSS=SS;
OldCPL=CPL;
/* 新 CPL 取代原 CPL */
/* 描述符装入 SS 投影寄存器,选择子装入 SS 寄存器 */
CPL=NewCPL;
SRegLoad(SS,NewSSSelector,$TS);
SS.Selector=NewSSSelector;
/* 原 SS 及 ESP 压栈 */
NewESP=NewESP-4;
AccessVirtual(SS,NewESP,4,1,$SS,&OldSS.Selector);
NewESP=NewESP-4;
AccessVirtual(SS,NewESP,4,1,$SS,&ESP);
/* 拷贝参数:从老栈读出时使用原来的 CPL,写入新栈时使用新的 CPL */
/* 拷贝顺序:先压栈参数先拷贝至新栈(顺序也可相反,方向一致即可) */
/* 参数拷贝完成后,新栈指针指向栈顶 */
tempESP=ESP+DwordCount*4;     /* 指向第一个压栈的参数 */
for(i=1;i<=DwordCount;i++) {
  CPL=OldCPL;
  tempESP=tempESP-4;
  AccessVirtual(OldSS,tempESP,4,0,$SS,&Dword1);
  CPL=NewCPL;
  NewESP=NewESP-4;
  AccessVirtual(SS,NewESP,4,1,$SS,&Dword1);
}
ESP=NewESP     /* ESP 指向新栈的栈顶 */
}
```

InnerStack( )的程序流程见图4.44。

(5) 执行过程:

第一,从当前任务状态段中读出新栈的选择子及栈指针:

·由 NewCPL 计算新栈指针在 TSS 内的偏移量;

·调用 AccessLinear( )读出新栈指针;

·调用 AccessLinear( )例程读出新栈的选择子。

第二,进行堆栈的切换:

·保存原栈的选择子和特权级;

·使用新特权级作为当前特权级;

·调用 SRegLoad( ),把新栈描述符装入 SS 描述符投影寄存器;

·新栈选择子装入 SS 寄存器。

第三,压栈及参数拷贝:

·原栈的选择子及栈指针压入新栈;

·按 DwordCount 规定的数量将调用参数从原栈拷贝到新栈。

最后,当前栈指针赋值为新栈指针,完成向内层切换的例程。在执行过程的第三步,关于参

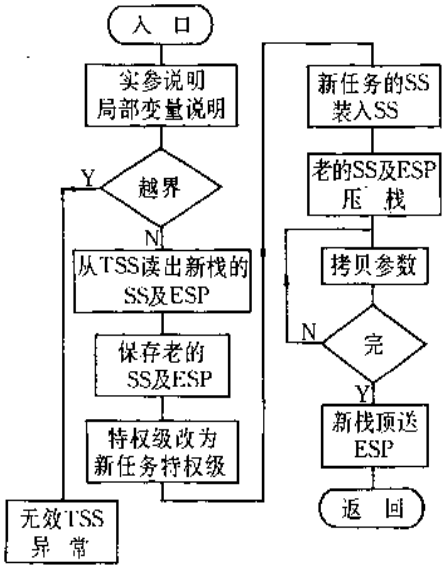


图 4.44 InnerStack( )的程序流程

数的拷贝操作，有三点请读者注意：第一，被拷贝的参数从堆栈中的某一地址开始，一直拷贝到堆栈的栈顶，即原栈栈顶中存放的是要拷贝的最后一个参数。所以，主调程序在通过调用门调用内层程序时，要预先将调用参数连续压入堆栈，并且在全部调用参数压栈之后，不能在堆栈中再压入任何其它内容；第二，被拷贝的参数，不是从原栈的栈顶开始、按弹出顺序从原栈读出，而是经过反推计算，从压入堆栈的第一个参数开始顺序读出，并压入新栈栈顶。所以，所传递的参数，在新栈与原栈中存放的次序是相同的；第三，从原栈读出参数时，当前特权级是原栈的特权级，向新栈写入参数时，当前特权级改变为新栈的特权级。所以，在拷贝过程中，当前特权级在不断地进行改变。最后一点非常重要，它对于保证页保护检查的正常工作是必需的，而且避免了使用内层特权级读出外层栈时可能出现的保护缺陷。

4.7.3.4 返回指令(RET)

返回指令是一条用来与调用指令配对使用的指令，该指令总是在子程序中使用，用来结束子程序的执行，返回到调用该子程序的下一条指令。

1. 返回主调程序的例程

(1) 例程名：RET( )

(2) 例程参数：ParmCount 从栈中跳过的字节数

(3) 程序功能：把程序控制返回到主调程序

(4) 程序清单：

```
/* 段内返回指令的详细说明 */
RET(ParmCount)
  int ParmCount      /*栈中要跳过的字节数 */
{
  SelectorType Selector,GSelector;
  SegAttributesAttributes;
  int          Base,Limit,Offset,GOffset;
  /*弹出返回地址 */
  AccessVirtual(SS,ESP,4,0,$SS,&Offset);
```

```
  ESP=ESP+4;
  AccessVirtual(SS,ESP,4,0,$SS,&Selector);
  ESP=ESP+4;
  /*选择子测试 */
  if(((Selector & 0FFFCh)==0)||(Seletor.RPL<CPL))
    SegmentException($GP,0);
  if(Selector.RPL>CPL) {
    /*向外层返回,恢复外层栈 */
    OuterStack(Selector.RPL,ParmCount);
  }
  /*读出并测试CS描述符 */
  ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);
  if(Attributes.DType==0)      /*描述符类型为系统段或门时产生异常 */
    SegmentException($GP,Selector);
  /*装入CS描述符 */
  CSDescriptorLoad(Selector,Attributes,Base,Limit,$GP);
  /*越界检查 */
  if(Offset>CS.Limit)
    SegmentException($GP,0);
  /*所有保护检查通过以后,装入CS选择子及EIP完成RET指令 */
  CS.Selector=Seletor;
  EIP=Offset;
}
```

RET 指令的程序流程见图4.45。

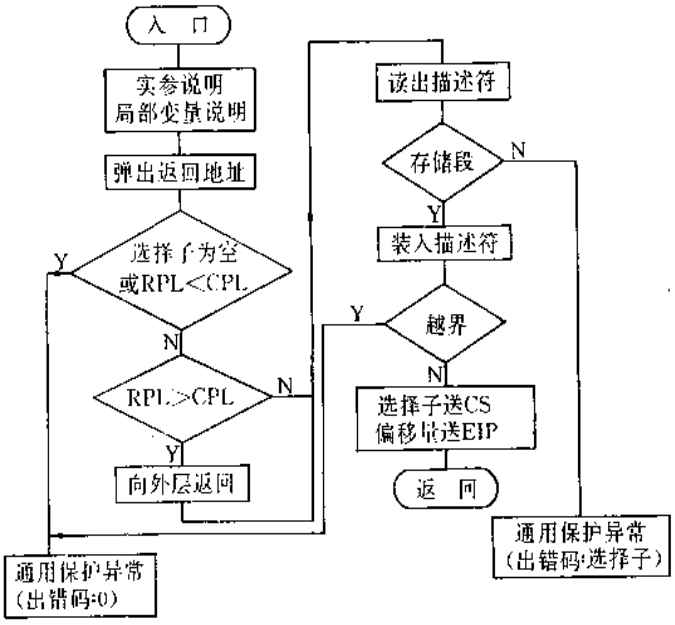


图 4.45 RET 指令的程序流程

(5) 执行过程

第一,分两次调用 AccessVirtual 弹出(读出)返回地址:CS 选择子(后16位有效)及偏移量(程序计数器指针)并通过改变堆栈指针,从堆栈中跳过指定个数的字节。

第二,对弹出的选择子进行测试:选择子必须非空(显然,主调程序的选择子不能为空);检查 RPL 与 CPL 之间的关系,如 RPL=CPL,则同层返回,不改变特权级。如 RPL>CPL,向外

层返回，改变特权级并进行栈切换。如 RPL<CPL，表示发生异常。一般来说，如果没有发生硬件故障，也没有在子程序中做特别的安排，对选择子进行的这两个方面的测试总是能够通过的。出于对系统安全方面的考虑，增加了这样的测试。类似的情况在别处还有，此后就不再做说明。

第三，调用 ReadDescriptor( )例程读出 CS 选择子指定的描述符及进行描述符类型的测试；再调用 CSDescriptorLoad( )例程把描述符装入描述符投影寄存器。

最后，把选择子装入 CS 寄存器，偏移量置入程序计数器，完成向主调程序的返回。

2. 向外层堆栈的切换

每当主调程序用调用门调用了内层的子程序、库函数，被调程序处理完毕之后，要进行一次从内层到外层的特权级转移，及进行对应的栈切换，恢复主调程序原来的特权级和原来的堆栈，并返回调用点的下一条指令恢复执行。如果主调程序的 CALL 发生了向内层的转移，则进栈保存的 CS 选择子中的 RPL 大于子程序的特权级(即当前特权级)CPL，所以 RET 指令在执行到进行选择子测试时，会有 RPL>CPL，从而调用本例程完成向外层的切换。

(1) 例程名：OuterStack( )

(2) 例程参数：

- NewCPL　切换后的特权级(即主调程序特权级)
- ParmCount 从堆栈中跳过的字节数

(3) 程序功能：实现向外层栈的切换

(4) 程序清单：

```
/* 切换到外层栈的例程 */
OuterStack(NewCPL,ParmCount)
  int NewCPL,      /* 返回到外层后的特权级 */
      ParmCount;     /* 从栈中跳过的参数字节数 */
{
  SelectorType NewSSSelector;
  int          NewESP;
  /* 从内层栈中弹出外层栈的 SS 及 ESP */
  AccessVirtual(SS,ESP,4.0 /* Read */,$SS,&NewESP);
  ESP=ESP+4;
  AccessVirtual(SS,ESP,4,0 /* Read */,$SS,&NewSSSelector);
  ESP=ESP+4;
  /* 用新 CPL 取代原 CPL */
  /* 选择子装入 SS 寄存器，描述符装入 SS 投影寄存器 */
  CPL=NewCPL;
  SRegLoad(SS,NewSSSelector,$GP);
  SS.Selector=NewSSSelector;
  /* 跳过指定字节数后的指针装入 ESP */
  ESP=NewESP+ParmCount;
  /* 检查由 DS、ES、FS、GS 所寻址的段，确认这些段能被外层的程序代码访问 */
  /* 否则，将选择子置为空，描述符的存在位置为0 */
  if( (DS.Attributes.Type<12) &&(DS.Attributes.DPL<CPL) )
    {DS.Selector=0;DS.Attributes.P=0;}
  if( (ES.Attributes.Type<12) && (ES.Attributes.DPL<CPL) )
    {ES.Selector=0;ES.Attributes.P=0;}
  if( (FS.Attributes.Type<12) && (FS.Attributes.DPL<CPL) )
    {FS.Selector=0;FS.Attributes.P=0;}
  if( (GS.Attributes.Type<12) && (GS.Attributes.DPL<CPL) )
```

```
    {GS. Selector=0;GS. Attributes. P=0;}
}
```

OuterStack( )的程序流程见图4.46。

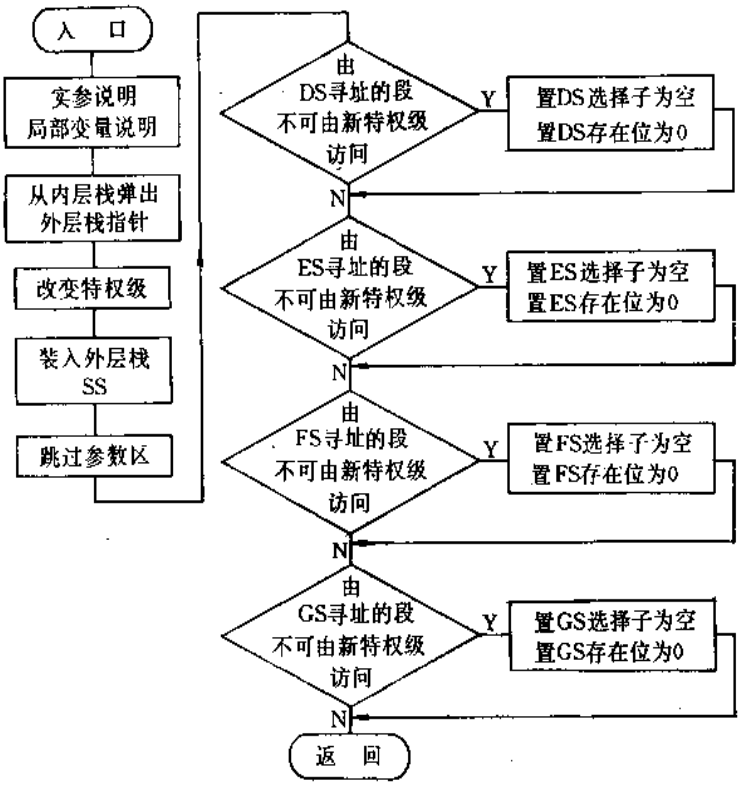


图 4.46 OuterStack( )的程序流程

(5) 执行过程：

第一，弹出目标栈(要恢复的栈)的选择子和栈指针，并把要恢复的特权级置为当前特权级之后，调用 SRegLoad( )例程，将读出的栈选择子装入 SS 寄存器、描述符装入描述符投影寄存器，并按要跳过的字节数调整栈指针的值。

第二，要检查由 DS、ES、FS 和 GS 所寻址的段，确信它们能被外层级的码访问；否则，要把它的选择子置为空，描述符的存在位置为0。

4.7.4 任务切换

像中断及 IRET 指令一样，CALL 及 JMP 指令也可以把程序控制转移到不同的任务。80386处理器可以把一个完整的任务切换作为正执行指令的一部分来完成。前面说明过的 TSS 系统段，将任务的各种状态信息存入 TSS 段中来支持任务切换。TSS 中包含有寄存器保存区域、链接任务字段及指向段和页表的指针，使80386可以保存及装入处理器寄存器、链接挂起的任务及改变虚拟—物理地址映射，从而完成一次完整的任务切换。用一条指令实现任务切换，并没有为80386增加任何功能。事实上，使用80386的指令集，用软件的方法，也能很容易地实现这里所描述的任务切换。但是，如果80386的任务模型与操作系统的任务模型紧密匹配，则硬件支持的任务切换操作，比起用软件方法就更为有效。当然，如果任务模型之间的差别甚大，最好还是使用软件方法的切换机制。

如果中断、JMP、CALL 或 IRET 指令的目标选择子确定一个 TSS 或任务门，则发生切换到 TSS 或由任务门指向的 TSS 的任务切换。任务门与调用门一样，都是间接地规定控制转移的目标。在任务门中，只使用指针指向的选择子部分，并且它对应一个 TSS。

在任务切换中，要进行特权级的检查，以保证 TSS 或任务门是可访问的。任务切换使用的特权检查，与调用门及数据段的装入所使用的检查相同。TSS 或任务门中的 DPL 所规定的特权级，相对于 CPL 及 RPL 必须处在同级或更外层的特权级。对任务门指向的 TSS 的 DPL 不进行特权级检查。

TSS 描述符的类型必须与转换的类型相适应。TSS 要么为"忙"、要么为"可用"，由 TSS 描述符中的类型表示。如果一个任务是当前正执行的任务、或者是用 TSS 中的链接字段沿挂起任务链接到当前任务上的任务，则该任务是"忙"的任务。否则，该任务为"可用"任务。CALL、JMP 或中断，必须转移到一个可用任务；而 IRET 则是转移到挂起任务链上的前一个任务，该任务必须是"忙"的任务。

前面已经述及，TSS 中的链接字段用于建立挂起任务链。如果因调用或中断任务切换到另一任务执行，则当前任务被挂起，并链接到挂起任务链的前端。如果执行一条 IRET 指令，则相匹配的任务从挂起任务链撤下，恢复为当前任务。IRET 指令所恢复的任务，是逆向沿挂起任务链恢复最后一个被挂起的任务，因而可实现嵌套返回。

任务切换不能递归。一个 CALL 或中断必须转到一个可用的 TSS、并且保持原来的 TSS 为忙，以避免任务的递归。可能产生嵌套中断的中断源，不能用任务切换来进行处理，在接收到由任务切换处理的中断以后，该中断必须完成，并执行一条 JMP 或 IRET 指令，标志该中断任务为可用，然后才能接收同一中断源的下一次中断。

EFLAGS 寄存器中的 NT 位，用来指示有挂起的任务链接到当前 TSS 这样一种情况。该位由 IRET 指令进行访问，以决定该返回是要求进行任务切换的返回、还是返回到当前任务内的另一程序。如果 NT＝1，则 IRET 沿链接字段返回到最近挂起的任务(即最后一个挂起的任务)；如果 NT＝0，则 IRET 指令从栈中弹出返回地址，返回到当前任务内的另一段程序。

任何特权级的程序都可以改变 NT 位的值。如果程序把 NT 位设置为1、然后再执行 IRET，则发生切换到当前 TSS 的链接字段指向的 TSS 的任务切换。如果链接字段尚未由相应的调用或中断设置，则可能产生难以预料的后果。因此，操作系统最好在跳转到 TSS 之前，对每个 TSS 字段小心地进行初始化。这样，即使有预谋的或有错误的程序把 NT 位置1，并执行 IRET，但操作系统的运行仍旧不会被破坏。

任务切换及有关的子例程有通过任务门的任务切换、通过任务状态段的任务切换、对任务状态段进行读/写访问及设置任务状态段标志等例程。以下逐一进行介绍。

1. 通过任务门的任务切换

(1) 例程名：TaskGate( )

(2) 例程参数：

- Selector　任务门中的选择子
- Linkage　链接标志：Linkage＝0表示无链接、Linkage＝1表示链接、Linkage＝－1表示解链。

(3) 程序功能：根据任务门中的选择子指定的 TSS 进行任务切换

(4) 程序清单：

```
/* 通过任务门进行切换 */
```

```
TaskGate(Selector,Linkage)
  SelectorType Selector;     /*任务门中的选择子 */
  int          Linkage;      /*链接字。0:无链接;1:链接;-1:解链 */
{
  SegAttributes Attributes;
  SelectorType  GSelector;
  int           Base,Limit,GOffset;
  /*选择子测试:选择子必须指定全局描述符表,而且非空,否则产生异常*/
  if( ((Selector & 0FFFCh) ==0
      ‖ (Selector. TI ==1))
    SegmentException( $GP,Selector);
  /*读出并测试描述符 */
  ReadDescriptor(Selector. &Attributes,&Base,&Limit,&GSelector,&GOffset);
  if (Attributes. DType == 1)     /*描述符类型为存储段时发生异常 */
    SegmentException( $GP,Selector);
  /*选择子指定的描述符必须是 TSS 段,再根据不同 TSS 段类型进行分支处理 */
  switch (Attributes. Type) {
  case 1:     /*描述符为可用286TSS */
    if (Linkage ==-1)
        SegmentException( $GP. Selector);
    if (Attributes. P ==0)
        SegmentException( $NP,Selector);
    TaskSwitch286(Selector. Attributes,Base,Limit,Linkage);
    break;
  case 3: /*描述符为忙的286TSS */
    if (Linkage !=-1)
       SegmentException( $GP,Selector);
    if (Attributes. P==0)
      SegmentException( $NP,Selector);
    TaskSwitch286(Selector,Attributes,Base,Limit,Linkage);
    break;
  case 9: /*描述符为可用386TSS */
      if (Linkage ==-1)
          SegmentException( $GP,Selector);
      if (Attributes. P==0)
          SegmentException( $NP,Selector);
      TaskSwitch(Selector,Attributes,Base,Limit,Linkage);
      break;
  case 11:     /*描述符为忙的386TSS */
        if (Linkage !=-1)
           SegmentException( $GP,Selector);
        if (Attributes. P==0)
            SegmentException( $NP,Selector);
        TaskSwitch(Selector,Attributes,Base,Limit,Linkage);
        break;
    Default: /*描述符为非 TSS 的其它系统段或门 */
        SegmentException( $GP,Selector);
    }
}
```

TaskGate( )的程序流程见图4.47。

(5) 执行过程:

第一,对选择子进行测试:选择子必须非空、指定一个 TSS 段的描述符且描述符必须在全局描述符表中。

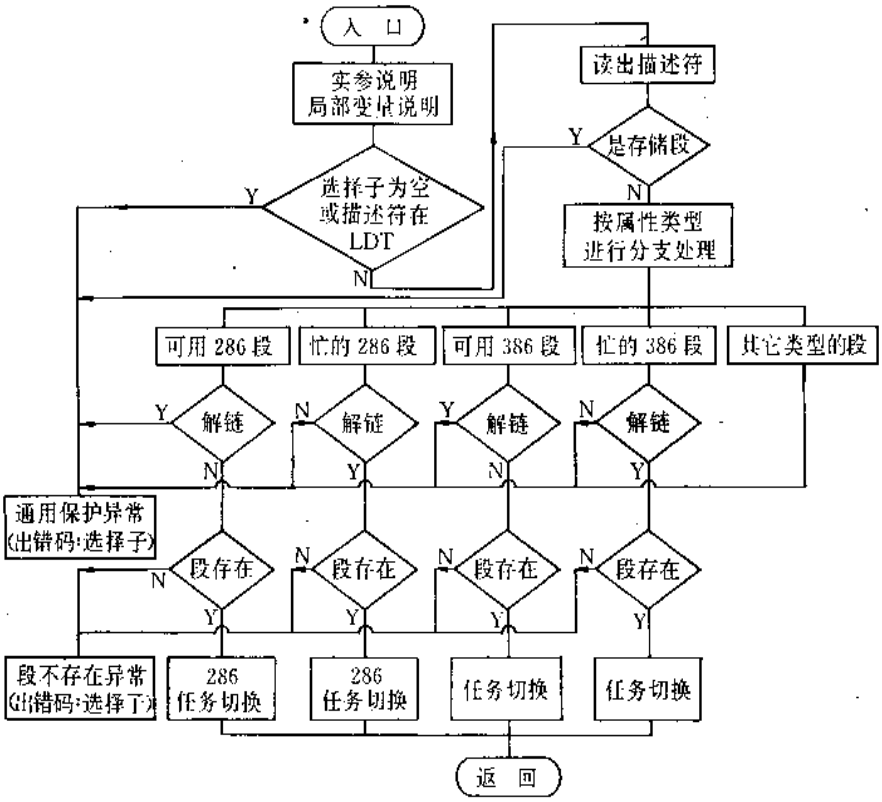


图 4.47 TaskGate( )的程序流程

第二,读出描述符并对其进行测试:

① 描述符属性中的 DT 位必须为0;

② 如描述符类型为可用286TSS,链接标志为非解链,则调用 TaskSwitch286( )进行任务切换,进入新任务;

③ 如描述符类型为忙的286TSS,链接标志为解链,则调用 TaskSwitch286( )进行任务切换,恢复前一任务;

④ 如描述符类型为可用386TSS,链接标志为非解链,则调用 TaskSwitch( )进行任务切换,进入新任务;

⑤ 如描述符类型为忙的386TSS,链接标志为解链,则调用 TaskSwitch( )进行任务切换恢复前一任务;

⑥ 如描述符类型为其它值,则产生异常。

2. 通过任务状态段的任务切换。

(1) 例程名:TaskSwitch ( )

(2) 例程参数:

·Selector 任务状态段的选择子

·Attributes 任务状态段的属性

·Base 任务状态段的基地址

·Limit 任务状态段的界限

·Linkage 任务状态段的链接标志

(3) 程序功能:根据 TSS 描述符进行任务切换。

(4) 程序清单:

```
/* 利用任务状态段进行任务切换 */
TaskSwitch(Selector,Attributes,Base,Limit,Linkage)
  SelectorType Selector;
  SegAttributesAttributes;
  int          Base,Limit,
               Linkage; /*链接字,0:无链接;1:链接;-1:解链 */
{
  /*任务状态段至少有104字节,否则产生异常 */
  if (Limit<103)
    SegmentException($TS,Selector);
  /*当前机器状态保存到老任务的 TSS */
  AccessTSSState(1 /* Write */);
  */(TR 及投影寄存器指向新任务的 TSS */
  TR.Base       =Base;
  TR.Limit      =Limit;
  TR.Attributes=Attributes;
  OldTSS        =TR.Selector; /*保存老任务的选择子供链接 */
  TR.Selector   =Selector;
  /*把新任务 TSS 中的内容装入硬件寄存器 */
  AccessTSSState(0 /* Read */);
  /*根据链接字的值进行不同处理 */
  /*如链接,老的 TSS 选择子存入新任务 TSS 中;设置嵌套位;新任务描述符设置忙标志 */
  if (Linkage ==1){
    AccessLinear(TR.Base,2.0/*Level 0*/,1/* Write */,&OldTSS);
    EFLAGS.NT=1;
    SetTSSBusy(Selector,1);
    }
  /* 如解链,则老任务 TSSR 描述符置为非忙 */
  else if (Linkage ==-1){
      SetTSSBusy(OldTSS.0);
      }
  /* 如无链接,则老任务 TSS 描述符标为非忙,新任务 TSS 描述符标为忙 */
  else if (Linkage ==0){
      SetTSSBusy(OldTSS,0);
      SetTSSBusy(Selector,1);
      }
  /*设置任务切换位及当前特权级 */
  CR0.TS=1;
  CPL=CS.Selector.RPL;
  /*各描述符投影寄存器的存在位预置为0 */
  LDTR.Attributes.Present=0;
  CS.Attributes.Present=0;
  SS.Attributes.Present=0;
  DS.Attributes.Present=0;
  ES.Attributes.Present=0;
  FS.Attributes.Present=0;
  GS.Attributes.Present=0;
  /*装入各个段的投影寄存器之前,先装入 LDTR */
  if (LDTR.Selector.TI ==1) /*LDTR 的选择子必须指定全局描述符表,否则产生异常*/
    SegmentException($TS,LDTR.Selector);
  if ( (LDTR.Selector & 0FFFCh)==0) {
      /*允许局部描述符表为空。如为空,将描述符存在位置为0 */
      LDTR Attributes.P=0;
```

```
    }
  /* 局部描述符表非空,则读出并测试描述符表的描述符 */
  else {
    ReadDescriptor(LDTR.Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);
    if ( (Attributes.DType ==1)
        ‖ (Attributes.Type !=2 /* LDT */)
        ‖ (Attributes.Present ==0) )
      SegmentException($TS,LDTR.Selector);
    /* 装入局部描述符表投影寄存器 */
    SetAccessed(LDTR.Selector);
    LDTR.Attributes =Attributes;
    LDTR.Base       =Base;
    LDTR.Limit      =Limit;
    }
  /* 装入 CS 投影寄存器 */
  JumpGate(CS.Selector,EIP,$TS);
  /* 装入 SS、DS、ES、FS 及 GS 投影寄存器 */
  SRegLoad(SS,SS.Selector,$TS);
  SRegLoad(Ds,DS.Selector,$TS);
  SregLoad(ES,ES.Selector,$TS);
  SRegLoad(FS,FS.Selector,$TS);
  SregLoad(GS,GS.Selector,$TS);
  /* 排错寄存器中的局部启用位置为0 */
  DR7.L0=0;
  DR7.L1=0;
  DR7.L2=0;
  DR7.L3=0;
  DR7.L0=0;
}
```

TaskSwitch( )的程序流程见图4.48。

(5) 执行过程:

第一,测试任务状态段的界限。为了保证因任务切换而离去的任务,在经过一段时间之后因任务切换而恢复执行时,不会改变离去时的运行环境,在任务状态段中定义了104个字节,用于保存各个任务所必需的全部信息。不同的任务,在任务状态段中还可以有数量不等的其它信息,也可以没有其它信息,所以,任务状态段的界限应大于或等于103,否则将产生异常。

第二,调用 AccessTSSState( )例程进行写访问,把通用寄存器、段寄存器、EIP 及 EFLAGS的当前值存入当前 TSS 中。保存的 EIP 的值为返回地址,指向引起任务切换指令的下一条指令。

第三,用新任务的 TSS 的选择子装入 TP 寄存器,并把描述符装入 TR 描述符投影寄存器。此后,任务执行情况(包括发生异常)都记录进新的任务中。

第四,调用 AccessTSSStatc( )例程进行读访问,把新任务 TSS 中存放的内容装入通用寄存器、段寄存器、EFLAGS 及 EIP。在新任务装入段寄存器的过程中,为了能正确地处理发生的异常,要先装入所有的选择子寄存器,然后再装入描述符。因为选择子寄存器已经装入、而描述符投影寄存器尚未装入,所以,要把所有的描述符投影寄存器的存在位都置为0,标志为不存在。

第五,根据链接标志位的要求,对新老任务状态段进行处理:

① Linkage=1,进行链接处理:调用 AccessLinear( )例程,将老任务状态段的选择子写入新任务状态段的第一个字的位置、新任务状态段的链接位 NT 设置为1、调用 SetTSSBusy( )

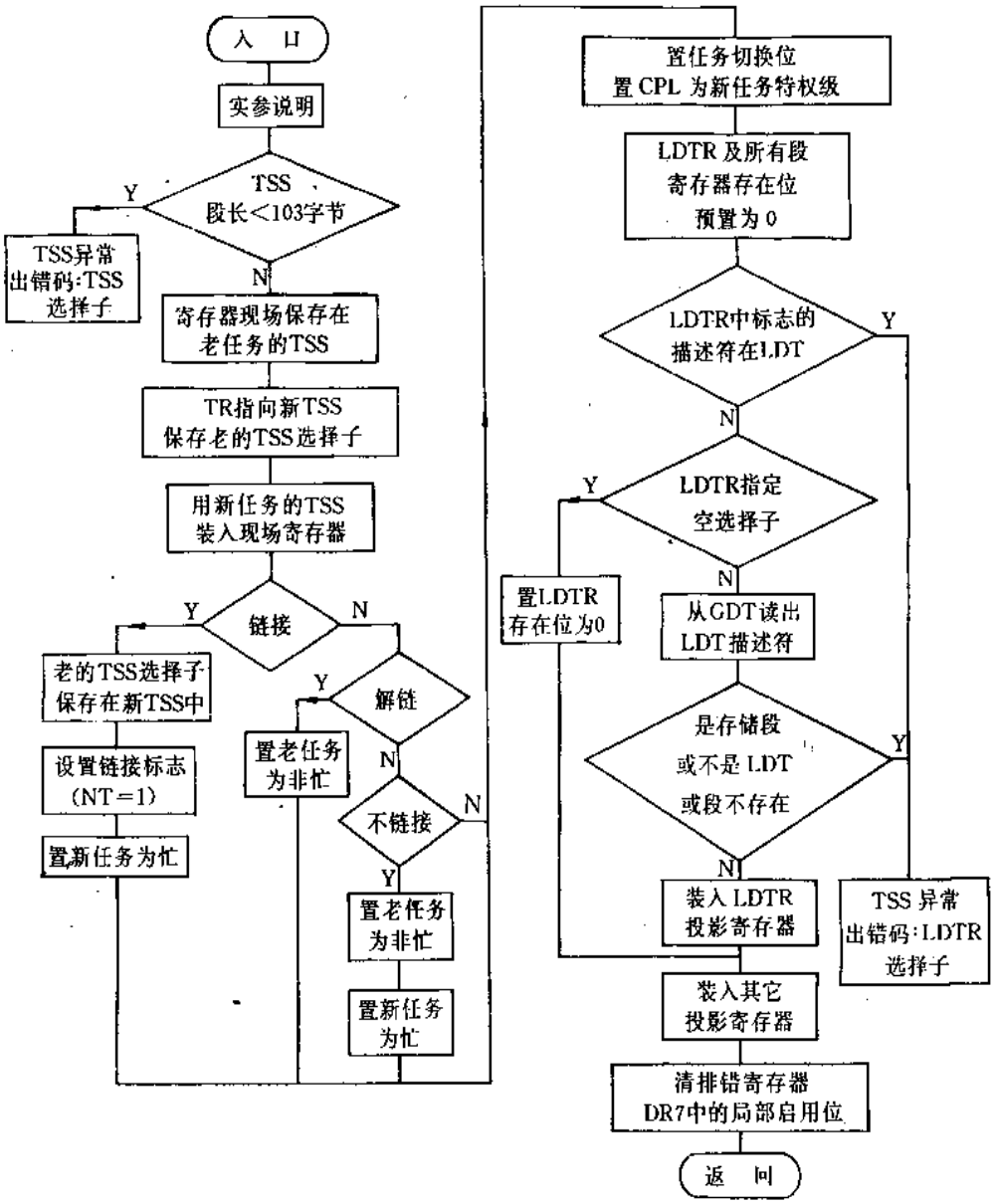


图 4.48 TaskSwitch( )的程序流程

例程将新任务状态段标志为忙；

② Linkage=−1，进行解链处理：调用 SetTSSBusy( )例程将老任务标志为“可用”；

③ Linkage=0，无链接处理：将老任务标志为可用、新任务标志为忙。

第六，把 CR0中的 TS 位置为1，用以标志已发生过任务切换，如新任务中使用了协处理器指令，则在使用该指令时产生自陷，从而把协处理器各寄存器的内容保存到老的任务中，并把新任务中的相应内容装入到这些寄存器中。由于很多任务都不使用协处理器指令，所以用这种延迟保存协处理器状态的方法，在很多情况下可以减少耗时较多的协处理器状态的保存与恢复操作。关于协处理器状态的保存/恢复，请参见第6章。

第七，把 TSS 中 CS 选择子的 RPL 作为新任务的特权级设置到 CPL。任务切换可以在一个任务的任何特权级发生，并可切换到另一任务的任何特权级。

第八，新任务状态段中局部描述符表的选择子的 TI 位必须为0，因为局部描述符表只能放在全局描述符表中，否则将产生异常。一个任务可以有自己的局部描述符表，也可以没有。当

任务没有局部描述符表时，任务状态段局部描述符表选择子的索引将为0。所以，如测试局部描述符表选择子的索引为0，则将局部描述符表的存在位置为0，表明新任务不使用局部描述符表。

第九，如新任务局部描述符表选择子为非空，则从全局描述符表中读出局部描述符表的描述符，并对描述符进行测试，且必须有：DT＝1、类型值为2、存在位为1。如满足这些条件，则将描述符装入投影寄存器，调用 SetAccessed( )例程设置局部描述符表描述符的访问位。

第十，调用 JumpGate( )例程装入 CS 寄存器，调用 SRegLoad( )例程装入堆栈段及各数据段(SS、DS、ES、FS 及 GS)寄存器及相应投影寄存器。

第十一，把排错寄存器 DR7中的局部启用位设置为0，以清除局部于老任务的各个断点及方式。排错寄存器的介绍请参见附录 A。

3. 保存/恢复任务状态段的例程

(1) 例程名：AccessTSSState( )

(2) 例程参数：RW

- RW＝0，读出任务状态段的内容装入硬件寄存器
- RW＝1，把硬件寄存器中的内容写入任务状态段。

(3) 程序功能：保存/恢复任务的运行环境。

(4) 程序清单：

```
/* 把机器状态保存在 TSS 或从 TSS 恢复机器的状态 */
AccessTSSState(RW)
  int     RW;  /* 访问 TSS 的方式:0为读,1为写 */
{
  /* 任务状态段至少含有104个字节,否则产生异常 */
  if (TR.Limit<103)
    SegmentException($TS,TR.Selector);
  /* 保存/恢复机器状态 */
  AccessLinear(TR.Base+20h,4,0 /* Level 0 */,RW,&EIP);
  AccessLinear(TR.Base+24h,4,0 /* Level 0 */,RW,&EFLAGS);
  AccessLinear(TR.Base+28h,4,0 /* Level 0 */,RW,&EAX);
  AccessLinear(TR.Base+2Ch,4,0 /* Level 0 */,RW,&ECX);
  AccessLinear(TR.Base+30h,4,0 /* Level 0 */,RW,&EDX);
  AccessLinear(TR.Base+34h,4,0 /* Level 0 */,RW,&EBX);
  AccessLinear(TR.Base+38h,4,0 /* Level 0 */,RW,&ESP);
  AccessLinear(TR.Base+3Ch,4,0 /* Level 0 */,RW,&EBP);
  AccessLinear(TR.Base+40h,4,0 /* Level 0 */,RW,&ESI);
  AccessLinear(TR.Base+44h,4,0 /* Level 0 */,RW,&EDI);
  AccessLinear(TR.Base+48h,2,0 /* Level 0 */,RW,&ES.Selector);
  AccessLinear(TR.Base+4Ch,2,0 /* Level 0 */,RW,&CS.Selector);
  AccessLinear(TR.Base+50h,2,0 /* Level 0 */,RW,&SS.Selector);
  AccessLinear(TR.Base+54h,2,0 /* Level 0 */,RW,&DS.Selector);
  AccessLinear(TR.Base+58h,2,0 /* Level 0 */,RW,&FS.Selector);
  AccessLinear(TR.Base+5Ch,2,0 /* Level 0 */,RW,&GS.Selector);
  /* LDTR 的选择子及 CR3只在恢复状态时访问 */
  if (RW==0) {
    AccessLinear(TR.Base+60h,2,0 /* Level 0 */,RW,&LDTR.Selector);
    AccessLinear(TR.Base+1Ch,4,0 /* Level 0 */,RW,&NewCR3);
    if (NewCR3 !=CR3) {
      /* 新任务的 CR3≠离去任务的 CR3时,刷新分页机制的线性——物理转换高速缓存 */
      FlushTLB;
```

```
        CR3=NewCR3;
        }
    }
}
```

AccessTSSState( )的程序流程见图4.49。

(5) 执行过程：

第一，检查任务状态段的界限，其值必须大于或等于103，以保证所有必须的信息都已保存在前104个字节中。

第二，调用 AccessLinear( )例程对任务状态段中各个通用寄存器、数据段和堆栈段选择子寄存器、标志寄存器、程序计数器在任务状态段相对应的单元进行规定的读或写访问以恢复或保存相应硬件寄存器内容。

第三，任务寄存器中，局部描述符表的选择子及控制寄存器的内容都是在整个任务执行的过程中不会发生改变的部分，因此，保存任务状态段时，不需对这些单元进行写访问的操作；但是在恢复任务状态段时，则需进行如下读操作：

① 调用 AccessLinear( )例程进行读访问，把局部描述符表的选择子装入局部描述符表选择子寄存器。每个任务经过切换进入时，都需把自己的局部描述符表装入并覆盖离去任务的局部描述符表；

② 调用 AccessLinear( )例程进行读访问，把新任务的控制寄存器 CR3读出并覆盖离去任务的 CR3中的内容。因为 CR3中的内容是页目录表的线性地址，所以当进入任务的 CR3≠离去任务的 CR3时，要刷新分页机制的线性——物理转换高速缓存。

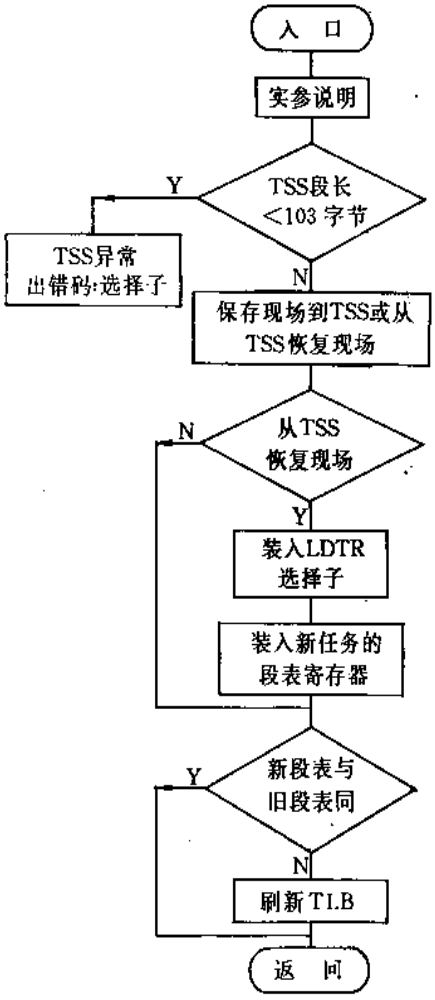


图 4.49 AccessTSSState( )的程序流程

4. 设置任务状态段标志的例程

(1) 例程名：SetTSSBusy( )

(2) 例程参数：

- Selector 要设置标志的任务状态段的选择子
- Busy 要设置的标志：Busy=0，设置为“可用”、Busy=1，任务状态段设置为“忙”。

(3) 程序功能：把任务状态段设置为忙/可用。

(4) 程序清单：

```
/* 改变任务状态段的忙/可用标志的例程 */
SetTSSBusy(Selector,Busy)
SelectorType Selector;   /* TSS 的选择子 */
int          Busy;       /* 要设置的状态:0为可用,1为忙 */
{
```

```
  int dword1;
  /* 调用本程序的调用者已保证选择子非空,而且在全局描述符表 */
  /* 读出描述符,修改类型字段的第二位                          */
  AccessLinear(GDT.Base+Selector.Index*8+4,4,0,0,&dword1);
  dword1=dword1&0FFFFFDFFh;  /*先将标志位清为非忙*/
  dword1=dword1|(Busy<<9);  /*再与参数Busy进行或操作*/
  AccessLinear(GDT.Base+Selector.Index*8+4,4,0,1,&dword1);
}
```

SetTSSBusy( )的程序流程见图4.50。

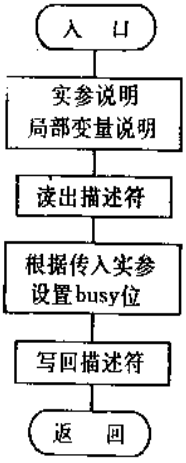


图 4.50 SetTSSBusy( )的程序流程

(5) 执行过程:

第一,从全局描述符表中,把由选择子指定的任务状态段的描述符的第二个双字读出来,由于"可用"386任务状态段的类型和"忙"的386任务状态段的类型分别为9和11(参见图4.11及表4.2)并位于描述符m+5字节的低4位中,即m+5字节低4位的值为1001表示任务状态段可用,低4位的值为1011表示任务状态段为忙。

第二,设置任务状态段的类型为指定状态:

① 将m+5字节第2位清除为0(用D,即1101进行按位"与"操作),使状态变为"可用"状态;

② 将例程中的参数Busy移位至第m+5字节的第2位(即左移9位),并与前一步得到的值进行或操作,使其置为所需要的状态:如Busy=0,经或操作后的结果仍为1001,相当于无论原来是什么状态,经此例程的上述操作后,已改变为"可用"386任务状态段的类型;如Busy位为1,经"或"操作后,第m+5字节低4位的结果变为1011,相当于经此例程的上述操作后,已改变为"忙"的386任务状态段类型。

第三,调用AccessLinear( )例程,将已设置好所需状态的描述符的第二个双字写回全局描述符表完成此例程。

第5章　中断及异常

中断和异常是特殊的控制转移方法，这种控制转移在正常编程的指令流之外完成。在正常的指令流中，通过执行一条CALL指令，可以实现从一个过程到另一个过程的控制转移。中断和异常实现程序的控制转移，则不是执行一条显式的指令，而是隐含的好像执行了一条CALL指令。该隐含的指令插入正在执行的程序的两条指令之间。这一条隐含的CALL指令，把程序控制转移到一个处理中断或异常的程序过程，该过程在完成之前，执行一条IRET指令(见第3章)，使其返回到被中断的程序。把程序控制转移到处理程序的过程就称为处理中断或异常。

中断由一个外部事件引起。中断被系统接受时，不执行显式的指令。通常，中断用于指示I/O设备的一次操作已经完成。外部事件及中断接受与正执行的指令没有关系。中断使程序控制转移到操作系统中的中断处理程序，并开始I/O设备的下一次操作：读数据、检查状态等。当中断处理程序完成时，该处理程序通过执行一条IRET指令使被中断的程序在断点处恢复执行。

异常是在指令执行期间检测到的不正常的或非法的条件，并与所执行的指令有直接的联系。例如，在指令执行期间检测到如第4章提到的段或页异常，使得指令不能成功完成；故软中断指令INT_n及INTO也归类于异常而不称为中断，这是因为执行这些指令产生异常事件；另一个异常的例子是排错机制(见附录A)。

中断和异常通常在两条指令之间进行处理。重复执行某一操作的串指令时(第3章)，串指令每完成一步操作，需更新指针及计数寄存器。由于执行时间不固定，而且执行时间相对较长，为了保证系统有良好的中断响应，每条串指令都允许在每次重复操作后响应中断。重复的串指令也允许报告一个异常而不丢失已完成各步的结果。例如，一条重复传送2000字符的指令，在执行至1901步时产生异常，而1900步是该指令成功执行的最后一个执行步，因此，产生异常时，保存有1900步的指针及计数的值。在异常的原因排除之后，恢复该指令的执行，并完成余下100步传送字符的操作。

每一种中断及异常，都有一个与之相联系的中断矢量，用8位2进制数字表示。分配给异常的矢量是前32个矢量编号，即0～31。例如，除法出错异常的矢量号为0，页异常的矢量号为14等。中断的矢量号可以分配给0～255范围内的任何值，但为了避免与异常矢量号发生冲突，故最好分配在32～255范围内。矢量号用来从中断描述符表中为给定的中断及异常选择处理程序。

在正常情况下，中断及异常可以在任意两条指令之间发生。然而，本章描述的机制可用来屏蔽中断及某些异常，从而使得被屏蔽的中断及某些被屏蔽的异常在屏蔽期间被忽略，直至屏蔽撤消后，这些中断及异常才能继续起作用。可以屏蔽的异常仅限于附录A中将描述的支持程序排错的那些异常，其它的异常(包括段异常、页异常等)不能被屏蔽，因而可以在任何时候发生。

5.1 中　断

如上所述，中断是由异步的外部事件引起的。外部硬件使用处理器芯片上特定的引脚，把中断信号传送给 80386 处理器。80386 支持两种类型的中断：它们是可屏蔽中断 INTR 及不可屏蔽中断 NMI。80386 有两个引脚分别对应这两种类型的中断。

5.1.1 INTR 中断

INTR 中断是可屏蔽中断，可以有任意的矢量号。外部硬件在发出 INTR 中断信息的同时，还向处理器给出一个 8 位的中断矢量号。Intel 公司的 8259A 可编程中断控制器芯片，可配合 80386 处理器的工作，并通过 80386 的单个中断引脚处理多达 64 个中断源。每个 8259A 可以支持 8 种中断，使用 9 个 8259A 中断控制器芯片，即可支持 64 个中断源，并给每一中断源分配不同的中断矢量。每个 8259A 芯片除了给 80386 提供矢量号之外，还要分配和判断中断的优先级。由于 80386 只提供一个 INTR 中断引脚，只识别一种优先级，所以由 8259A 芯片判断优先级就是一种很重要的功能。

INTR 中断，可以用 EFLAGS 寄存器中的 IF 位进行屏蔽。如果 IF 位为 1，INTR 中断可以在任何两条指令之间发生；IF 位为 0，INTR 中断被屏蔽。当 IF＝0 时，如有 INTR 中断产生，则将信号送给 80386 处理器，由于中断被屏蔽，使处理器并不理会这一中断信号的请求，该中断信号被挂起，这种挂起状态保持到 IF 被置为 1 或外部硬件撤消此中断请求为止。某些程序，在程序代码的某一特定区域(常称为程序的临界区)，为了保证执行程序的可靠性，当程序在该区域内执行时，不允许响应任何中断。利用 IF 位屏蔽中断的这一特性，可以达到使程序在临界区运行时不响应中断的目的。

5.1.2 NMI 中断

NMI 中断是不可屏蔽中断，即对 INTR 起屏蔽作用的 IF 位对 NMI 不起屏蔽作用。系统在响应 NMI 中断时，不从外部硬件接收中断矢量号。NMI 的中断矢量号是由系统固定分配的，在 80386 中，NMI 的中断矢量号为 2。

因为 NMI 是不可屏蔽中断，所以 NMI 仅限于系统出现某种紧急事件时使用。例如，系统中出现总线超时、电源故障、电网掉电等情况时，就在 NMI 端子上产生不可屏蔽中断信号。

由于 80386 芯片只提供一个输出端供 NMI 使用，因此，当一种紧急事件通过 NMI 由系统正进行处理时，在处理期间又发生了另一紧急事件；或两种紧急事件同时在同样的两条指令之间发生时。仅在这两种情况下，一种事件的 NMI 处理程序，可以屏蔽(实际上是封锁)另一个 NMI 中断请求。与 INTR 相比较可知：INTR 的屏蔽由 IF 位实现，对程序员是可见的，NMI 的屏蔽是由于对另一个 NMI 的处理引起的，对程序员不可见。应该说，产生 NMI 中断那样的紧急情况，在系统正常运行时是极少发生的，而一种 NMI 的处理，屏蔽另一种 NMI 则更是极其少有的。但是，系统对这种看来不大可能发生的情况，提供万一发生这一情况时的处理支持则是必需的。因此 80386 处理器的设计中，每当接受一个 NMI 中断，就在内部屏蔽了再次响应 NMI 事件，避免发生系统的嵌套中断。这一屏蔽过程直至处理程序执行 IRET 指令后才结束。所有的 NMI 处理程序，都在相应事件的处理结束之后，用 IRET 返回到被中断的程序。

5.2 异　常

异常是与特定指令的执行相联系的不正常的或非法的条件。80386 识别几种不同类别的异常，并赋予每一种类别不同的矢量号。当把异常通知系统时，根据引起异常的程序是否可被恢复而又把异常进一步分类为故障(Fault)、陷阱(Trap)或中止(Abort)。

Fault 是在引起异常的指令之前，把异常情况通知给系统的一种异常。当系统把程序控制转移到故障处理程序时，所保存的断点 CS 及 EIP 的值，指向引起故障的指令。在 80386 中，故障是可以重新执行的。这就是说，一旦异常的原因排除之后，处理程序的 IRET 指令使故障程序恢复执行，并使上次执行时引起异常的指令，在从处理程序返回之后再执行一次。

故障的检测，可以在一条指令开始执行之前，也可以在一条指令的执行期间进行。如果在执行指令的中间检测到故障，则中止故障指令，并把指令的源操作数恢复为指令开始执行之前的值。因此，当故障程序恢复之后，故障指令将重新执行，并且与前一次执行该指令有完全相同的输入条件，从而保证可以得到正确的结果。

Trap 是在引起异常指令之后，把异常情况通知给系统的一种异常。当系统把程序控制转移到异常处理程序时，所保存的断点 CS 及 EIP 的值，指向引起故障的指令的下一条应该执行的指令。这里要请读者注意，下一条应执行的指令，不一定就是下一条指令。因此，对于用陷阱这样一种类别通知系统发生异常情况时，系统并不是总能根据保存的断点，反推确定产生异常的指令。在一个陷阱通知给系统之前，引起陷阱的指令应正常完成，它有可能改变了寄存器或存储器。

Abort 是在系统出现严重情况(例如硬件故障或系统表中出现非法值或不一致的值)时，通知给系统的一种异常。引起中止的指令是无法确定的。产生中止时正执行的程序不能被恢复执行。系统接收中止异常之后，处理程序要重建各种系统表格，并可能需要重新启动操作系统。

第 4 章介绍过的段及页异常是故障的例子。在一条指令的执行期间，如果检测到段或页的异常，则停止该指令的执行，CS 及 EIP 指向该指令，并通知系统：一个故障已经产生。在附录 A 描述的单步及数据断点异常，是陷阱的一个很好的例子。如果在指令的执行期间检测到这样的异常，则让该指令正常完成，CS 及 EIP 指向下一条指令，并通知系统：一个陷阱已经产生。

第 3 章描述的软件中断指令 INT n 及 INTO 是不同中断的陷阱指令。执行这些指令时，将产生一个陷阱作为指令执行的一部分。该陷阱导致系统发生控制转移，并转到陷阱处理程序，而 CS 及 EIP 则指向下一条指令。

5.2.1 指令的重新执行

所有的指令在故障之后都是可以重新执行的。重新执行的处理，不需要操作系统软件的参与。在操作系统已经改正了故障的原因之后，故障处理程序返回时，执行一条 IRET 指令，使故障指令直接地再重新执行一次。

5.2.2 异常类型

本节用矢量号及每一类型的简要说明列出异常的类型。每一种异常都分类归于故障、陷阱或中止。某些异常还以出错码的形式提供某种附加的信息传递给异常处理程序。

多数异常都是十分的简单，并且在简要说明中做了适当的描述。第4章已对段及页异常进行了详细讨论；附录A对排错异常进行了详细讨论；协处理器出错异常在本章下面的专门的节中讨论。

1. 异常0——除法出错

除法出错是一种故障。当执行DIV指令或IDIV指令时，如果除数为0，或者相除后得到的商太大，在结果操作数中容纳不下时产生这一故障。在除法出错处理程序的入口，被保存的CS及EIP的值，指向产生故障的指令。除法出错不提供出错码。

2. 异常1——排错异常

排错异常的矢量号为1。排错异常有故障类型的，也有陷阱类型的。寄存器DR6，可以由排错异常处理程序访问，用以确定排错异常的原因及类型。在一条指令中，可以检测不止一个排错异常并使DR6中同时有几个位置为1。如果EFLAGS中的RF位为1，则给出排错故障标志。排错异常不提供出错码。

3. 异常3——单字节INT3

INT3是一条特别的一字节INT n指令。该指令由排错程序用来支持程序断点。INT3指令被看成是一种陷阱，而不是一个中断。在异常处理程序入口，保存的CS及EIP的值指向紧跟INT3的指令，即INT3指令后面的一个字节。INT3陷阱不提供出错码。

4. 异常4——溢出

溢出陷阱通过矢量4通知给系统。INTO指令通过矢量4提供一个条件陷阱：如果EFLAGS寄存器中的OF位为1，则INTO产生陷阱；OF位为0，则不产生陷阱，继续执行INTO后面的指令。在溢出异常处理程序的入口，保存的CS及EIP的值，指向INTO的下一条指令。溢出陷阱不提供出错码。

5. 异常5——边界检查

如果BOUND指令发现被测试的值超出了规定的范围，则发生边界检查故障。在边界检查处理程序的入口，保存CS及EIP的值，指向发生故障的BOUND指令。边界检查故障不提供出错码。

6. 异常6——无效操作码

从CS及EIP所指定的位置开始，如果连续一个或多个字节所包含的内容，不表示80386指令集中的任何一条指令，则发生无效操作码故障。有如下三种情况发生这样的故障：

- 操作码字段的内容不是一个合法的80386指令的代码；
- 要求使用存储器操作数指令时，却规定的是寄存器操作数；
- 不能被加锁的指令使用了LOCK前缀。

在无效操作码故障处理程序的入口，保存的CS及EIP的值，指向无效指令的第一个字节。无效操作码故障不提供出错码。

7. 异常7——设备不可用

设备不可用故障，支持80387数字协处理器。在不包括80387协处理器硬件的系统中，可用异常7替代协处理器的软件模拟器。在发生任务切换时，也可用异常7来推迟80387寄存器(硬件寄存器或软件模拟的寄存器)状态的内容切换，使得只有在另外的任务使用浮点指令时，才进行80387寄存器状态的切换。在设备不可用故障的入口，保存的CS及EIP的值，指向发生故障的指令。设备不可用故障不提供出错码。

设备不可用故障在下列情况下产生：

(1) 在执行浮点指令时，控制寄存器 CR0 中的 EM 位或 TS 位为 1；

(2) 在执行 WAIT 指令时，控制寄存器 CR0 中的 TS 位及 MP 位都为 1。

参见第 4 章关于 CR0 寄存器中 EM、TS 及 MP 位的讨论。

8. 异常 8——双重故障

当检测到段或页异常时，已有另外一个异常正通知给系统，则处理机试图向系统通知一个双重故障，而不是通知第二个异常。双重故障被分在中止异常那一类，即保存在双重故障处理程序的入口处的 CS 及 EIP 的值，不可能指向引起双重故障的指令，而且指令的重新启动不支持双重故障。

双重故障通常指示系统表中出现的严重问题，例如段描述表、页表、或中断描述表出现问题。在调试一个还不能正确处理系统表的操作系统时，双重故障异常可能是很有用的。对于一个产品系统，双重故障处理程序在重建系统表后，可能不得不重新启动操作系统。

当正通知一个段故障异常时，有可能又产生一个页故障。在这种情况下，通知给系统的是页故障异常而不是双重故障异常。然而如果正通知给系统一个段或页故障时，又检测到一个段故障或正通知给系统一个页故障时又检测到一个页故障，这时，都引起双重故障。

如果正通知一个双重故障的过程中，又检测到一个段或页异常，则处理机暂停执行指令，并进入关机方式。此时的关机方式类似于处理机执行一条 HLT 指令后的状态：处理机空转并维持到处理机接收到一个 NMI 中断或处理机被重新启动为止。在关机方式下，处理机不响应 INTR 中断。

9. 异常 9——协处理器段越界

协处理器段越界，是浮点指令操作数超出段界线时产生的异常，这种异常属于中止类异常，不提供出错码。

例如，有一条浮点指令，操作数占 8 个字节，存储在一个大小为 7FFFFFFDh 的段中，如果指定操作数首址的偏移量为 7FFFFFFCh，则操作数所占地址的偏移量范围应为 7FFFFFCh～80000003，故该操作数有 6 个字节不在指定段中，产生段越界。

因为引起协处理器段越界异常的指令不能被重新启动(事实上，每次执行总得到同样结果)，所以协处理器段越界异常是一种中止类的异常。在协处理器段越界异常处理程序中，必须先用 FINIT 指令使协处理器重新初始化，然后才能从处理程序返回。在异常处理程序的入口保存的 CS 及 EIP 的值，指向被中止的指令，但是这种中止不是系统的中止，而是只影响检测到这种异常时，正执行的指令所在的程序。

10. 异常 10——无效 TSS

当正从 TSS 装入选择子时，如果发生除了不存在异常以外的段异常时，则产生无效 TSS 故障。这种异常提供一个出错码，其内容包含有引起异常的段的选择子。出错码的格式及产生无效 TSS 异常的条件详见第 4 章。在那里，这一异常用符号 $TS 表示。

因为这种异常属于故障类型，所以，在异常处理程序的入口处保存的 CS 及 EIP 的值，指向引起这一故障的指令，或者该故障作为任务切换的一部分发生时，指向任务的第一条指令。

11. 异常 11——段不存在

当处理器访问除 SS 以外的其它有效的段描述符时，发现有 P=0，则产生段不存在异常。段不存在故障提供一个包含有引起该异常的段选择子的出错码。出错码的格式及产生此异常的条件详见第 4 章。在那里，这一异常用符号 $NP 表示。

因为这种异常属于故障类型，所以，在异常处理程序的入口处保存的 CS 及 EIP 的值，指

向引起这一故障的指令，或者该故障作为任务切换的一部分发生时，指向任务的第一条指令。

12. 异常 12——栈段异常

当处理机检测到用 SS 段寄存器进行寻址的与段有关的某种问题时，产生栈段故障。栈段故障异常提供一个其值依赖于检测条件的出错码。出错码的格式及产生此异常的条件详见第 4 章。在那里，这一异常用符号 $SS 表示。引起栈段故障异常的条件分为如下三类：

(1) 由 SS 寄存器寻址的段中的界限超出范围，将引起栈段故障，出错码为 0；

(2) 层间调用或中断期间，内层栈中的界限超出范围，将引起栈段故障，出错码包含有内层栈的选择子；

(3) 如果要装入到 SS 寄存器的描述符，其存在位为 0，则产生栈段故障，出错码包含有对不存在段进行访问的选择子。

(2)、(3) 两种情况都导致具有非 0 出错码的故障。由于不可能直接用出错码来区分开后面这两种异常情况，因此，在栈段故障处理程序中，要借助于考查出错码的选择子对应的描述符中的存在位。如果存在位为 1，则表明是内层栈中的界限违反引起的栈段故障，如果存在位为 0，则表明是不存在段引起的栈段故障。

栈段异常是一种故障，所以，在异常处理程序的入口处保存的 CS 及 EIP 的值，指向引起这一异常的指令，或者该异常作为任务切换的一部分发生时，指向新任务的第一条指令。这种异常是可以重新启动的。

在支持可以扩展栈段的系统中，如果栈段故障的出错码为 0，则表示需要对栈段进行扩展。

13. 异常 13——通用保护

一个没有预先分类的段异常，作为通用保护故障通知给系统。这种异常提供一个其值依赖于被检测条件的出错码。出错码的格式及产生通用保护故障异常的条件详见第 4 章。在那里，这一异常用符号 $GP 表示。

因为这种异常属于故障类型，所以，在异常处理程序的入口处保存的 CS 及 EIP 的值，指向引起这一故障的指令，或者该故障作为任务切换的一部分发生时，指向任务的第一条指令。

根据通用保护异常处理程序可能进行的响应，引起该异常的条件，可以分为如下两个类别：

(1) 指示违反保护方式但不终止程序的异常。这种异常在应用程序执行特权指令或 I/O 访问时发生；支持虚拟 8086 程序的系统或支持虚拟 I/O 访问的系统，需要模拟这些指令，并在模拟完成产生故障的指令之后，重新执行被中断的程序。这些异常报告给系统时，其出错码为 0。

(2) 指示违反保护方式并导致故障程序终止的异常。这种异常的出错码可能为 0，也可能为一个选择子的值。

两种异常可以通过检查引起故障的指令及出错码而予以区别。当出错码不为 0 时，肯定是第二种类型的异常。如果出错码为 0，则需进一步检查引起故障的指令。由于通用保护异常是一种故障，所以保存在故障处理程序入口处的 CS 及 EIP 指向产生故障的指令。如果该指令是系统支持的可以模拟的指令，且出错码为 0，则处理程序模拟该指令，并返回到故障指令的下一条指令，否则，故障程序被终止。

14. 异常 14——页异常

同时满足如下两个条件时产生页故障：

(1) 分页机制被启用，即寄存器CR0中的PG位为1；

(2) 指令在进行存储访问时，被访问的线性地址所在的页未驻留在内存，或用不适当的访问类型访问一内存页。

处理器把引起页故障的线性地址装入寄存器CR2，并提供一个出错码，指示引起页故障的存储器访问的类型。出错码的格式及产生页故障的条件详见第4章。

页故障是故障类型的异常，所以保存在页故障处理程序入口处的CS及EIP的值，指向引起故障的指令。由于页异常是可以重新启动的异常，所以，一旦引起页故障的原因被排除之后，即可从页故障处理程序通过执行一条IRET指令，直接地重新执行产生故障的指令。

15. 异常16——协处理器出错

协处理器出错故障通知给系统的矢量号为16。协处理器出错故障，表示协处理器发生了未被屏蔽的数字错误(如上溢或下溢)。在引起故障的浮点指令之后的下一条浮点指令或WAIT指令，把协处理器出错作为一个故障通知给系统。该故障不提供出错码。协处理器出错异常的更详细的说明容后述。

5.2.3 异常小结

表5.1概览了本节所讨论的异常条件，用与异常相联系的矢量号为序列出各种异常，包括引起给定异常的指令、每种异常的分类以及产生异常时是否提供出错码。

表5.1 异常一览表

矢量号	异常名	类 别	出错码	产生异常的指令
0	除法出错	故障	/	DIV，IDIV
1	排错	故障/陷阱	/	任何指令
3	单字节INT3	陷阱	/	INT3
4	溢出	陷阱	/	INTO
5	边界检查	故障	/	BOUND
6	非法操作码	故障	/	非法指令编码或操作数
7	设备不可用	故障	/	浮点指令或WAIT
8	双重故障	终止	有	任何指令
9	协处理器段越界	终止	/	访问存储器的浮点指令
10	无效TSS	故障	有	JMP、CALL、IRET、中断
11	段不存在	故障	有	装入段寄存器的任何指令
12	栈段异常	故障	有	·装入SS寄存器的任何指令 ·对以SS寄存器寻址的段中进行存储访问的任何指令
13	通用保护	故障	有	·任何特权指令 ·任何访问存储器的指令
14	页异常	故障	有	任何访问存储器的指令
16	协处理器出错	故障	/	浮点指令或WAIT
0～255	软中断	陷阱	/	INTn

5.3 中断及异常的优先级

本节描述在一条指令执行期间，如检测到不止一个中断和异常时，以什么样的顺序对检测到的中断及异常进行处理的细节。表 5.2 按中断优先由高至低的列出了 80386 所识别的中断及异常，当检测到多于一个中断或异常时，把优先级最高的中断或异常通知给系统，其它优先级较低的异常被废弃，而优先级较低的中断则保持悬挂。当把较高优先级的异常通知给系统时，废弃的较低优先级的异常，有可能在引起较高级异常的原因被排除之后，引起较低级异常的指令被重新执行时会又一次被检测到。

表 5.2 中断/异常的优先级

中断/异常类型	优先级
排错故障	最高
非排错故障	↓
陷阱指令 INTn、INTO	↓
排错陷阱	↓
NMI 中断	↓
INTR 中断	最低

例如，当 NMI 及 INTR 同时产生时，由于 NMI 有较高优先级而被系统接收，而 INTR 中断则保持悬挂。又如在同一条指令期间检测到排错陷阱和页故障时，页故障有较高优先级而被接收。排错异常被废弃。当排除页故障之后，重新执行引起故障的指令时，排错隐阱将再次产生并通知给了系统。当然这里假设在此期间没有冒出新的优先级更高的异常。

请注意，在一条指令期间检测到多个异常及中断的情形，与双重故障的情形是完全不相同的。这里所描述的情形，是要确定在一条指令的执行期间发生的多个中断及异常中，选择哪一种中断或异常通知给系统，而且这些同时发生的中断及异常中，只把一种中断或异常通知给系统。当正通知给系统一个高级的中断或异常时，又产生一个段或页异常，则废弃选择的中断或异常，转而通知系统产生了双重故障。而单条指令期间的多个故障不引起双重故障。

5.4 屏蔽中断及异常

某些条件及处理机标志将引起中断及排错异常被忽略或被屏蔽，在这种情况下，各种中断保持悬挂，而排错异常则被废弃。这样的情况有：

(1) 如 EFLAGS 寄存器中 IF 位为 0，则 INTR 中断被屏蔽。

(2) 如 IF 为 0 时，执行 STI 指令，则在 STI 指令及下面一条指令的执行期间，屏蔽 INTR 中断。

(3) 如 EFLAGS 寄存器中的 RF 位为 1，则屏蔽排错故障。

(4) 系统正处理一个 NMI 期间，则屏蔽任何新的 NMI 中断。

(5) 以 SS 寄存器为目的寄存器的 MOV 或 POP 指令，将在该指令及下面一条指令的执行期间屏蔽各种中断及排错异常。

前二种情形在本章前面已经提及，最后一种情况可实现新栈指针的装入，通过一个指令对，把新栈指针装到SS及ESP寄存器。这种装入SS寄存器的指令，对程序员是不可见的。如果MOV或POP到ESP指令，则两条指令之间不会产生中断和排错异常，保证能够安全的更新一个完整的栈指针。只要第二条指令中ESP的装入不发生段和页异常，将连续执行两条指令。第3章描述的LSS指令是80386的一条新的指令，该指令把存储器中一个48位的指针分别装入SS及一个通用寄存器(通常为ESP)中。因为该指令用一条指令可装入SS及ESP两个寄存器，所以该指令用于装入新栈的指针，并取代原来用两条指令装入新栈指针的老方法。

5.5 中断/异常的转移方法

本章提供描述中断/异常控制转移的两种层次，转移的结果，使得中断或异常相对应的矢量号所选择的处理程序开始执行。本节给出的第一种层次，是这些转移使用的方法的高级形式，后面各节中给出的第二种层次，是关于转移机制的详细描述，并以类似于C的表示法给出。这些方法和机制类似于第4章所述关于段间的CALL通过调用门及任务门使用的方法和机制。读者应透彻地理解有关的材料之后，再阅读本章以下各节的内容。

在发生中断及异常时要使用中断描述符表(IDT)。如第4章所描述，该表由IDTR寄存器指向。矢量号作为IDT表的索引，从IDT表中获取一个8字节的门描述符。图5.1中示出门描述符的格式。在门中包含48位的全指针及占用了16位的属性。48位的全指针包含的信息，与第2章介绍的全指针数据类型包含的信息相同，即包含32位的偏移量及16位的选择子。但是，这里的偏移量分成两个部分存放。如图可见，全指针的选择子部分，存放在m+2及m+3字节，偏移量的低16位存放在m及m+1字节，高16位存放在m+6及m+7字节。

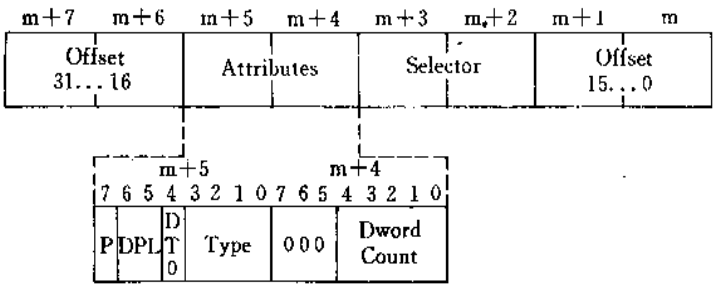


图5.1 门描述符格式

在门描述符中，属性位存放在m+4及m+5字节。各属性位的详细说明如下：

P 为存在位，P=1表示门有效；P=0表示门无效。使用无效门将引起异常。

DPL 为描述符特权级，定义与门相联系的特权级。门的DPL只在INTn及INTO时进行检查，以避免应用程序执行INTn时，使用了各种设备在中断表中使用的矢量号。对于所有其它的异常或中断，忽略门DPL。

DT 为DType位，该位用于区分存储段(DT=1)，及非存储段(即系统段或门，DT=0)。我们现在讨论的是门描述符，故此处有DT=0。

Type 为四位类型字段，用于定义门的类型，见表5.3。表5.3与第4章中的表4.2完全相同。第4章的讨论重点是说明描述符为系统段或门时，类型字段的编码。此处则强调系统段

或门的类型。为了阅读内容方便，该表也在此处列出。IDT 中的描述符必须具有下列类型之一：

- 任务门
- 286 中断门
- 286 陷阱门
- 386 中断门
- 386 陷阱门

DwordCount 该字段在此处的门中没有使用。

表 5.3 系统段及门描述符的类型

类 型	说 明	类 型	说 明
0	/	8	/
1	可用 286TSS	9	可用 386TSS
2	LDT	10	/
3	忙的 286TSS	11	忙的 386TSS
4	286 调用门	12	386 调用门
5	任务门	13	/
6	286 中断门	14	386 中断门
7	286 陷阱门	15	386 陷阱门

286 中断门及陷阱门在第 7 章讨论，本章描述通过任务门及 80386 的中断门和陷阱门进行的控制转移。通过任务门的转移，可使程序控制转移到不同任务的处理程序；而通过中断门及陷阱门的转移，只能使程序控制转移到当前任务的处理程序。中断、故障、陷阱及中止，可由三种门的类型(任务门，中断门，陷阱门)中的任何一种来进行处理。

通过任务门的转移，执行一个完整的任务切换，并开始执行另一个任务的处理程序。这种任务切换等同于通过任务门执行一个 CALL 指令实现任务切换所使用的方法。中断及陷阱门只进行一个简单得多的转移，使控制转移到当前任务的一个处理过程。这种控制转移类似于但并不等同于通过调用门执行一个 CALL 指令实现控制转移使用的方法

5.5.1 中断及陷阱门

如果经矢量号索引的 IDT 描述符，是一个 386 中断门或 386 陷阱门，则表示控制转移到当前任务的一个处理程序过程。与通过调用门的 CALL 指令一样，指向处理程序过程的 48 位指针，从中断门或陷阱门中获取。图 5.2 示出 IDT 中的中断门或陷阱门如何指定中断处理程序过程。门中的选择子用于访问 GDT 或 LDT 表中的描述符，该描述符必须指定一个可执行的存储器段。门描述符的偏移量规定处理程序入口点在段内的偏移量。

门中的 DPL 只在 INTn 及 INTO 时进行检查，以避免应用程序执行 INTn 时，使用了分配给各种设备在中断表中使用的矢量号。对于所有其它的异常或中断，门的 DPL 被忽略。

门的选择子必须指定一个存在的、可执行的存储段。该存储段描述符中的类型及 DPL 字段，确定中断或异常是转移到当前特权级的某一处理过程，还是要求转移到一个新的特权级的

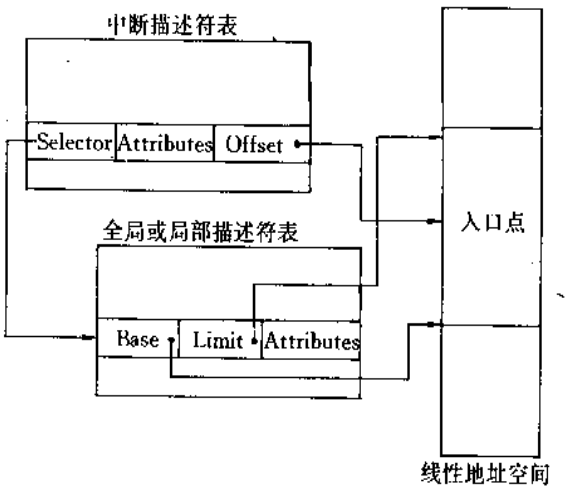


图 5.2 通过中断门或陷阱门的转移

处理过程。中断及异常可以转移控制到同一特权级或内层特权级；与之相匹配的 IRET 指令，则转移控制到同一特权级或外层特权级。这一点，满足 CALL 及 RET 指令的向内调用，向外返回的规则。

如果门中的选择子确定一个一致的代码段，且 DPL 与 CPL 同级或 DPL 位于相对 CPL 更内层的级；或者确定一个非一致的码段，且 DPL 与 CPL 同级，则表示转移到当前特权级的过程。一个非一致的代码段，且 DPL 在相对于 CPL 更内层的级，则表示转移到内层特权级。新的特权级是 DPL 的值所表示的级。与通过门执行 CALL 指令一样，作为特权级切换部分，栈段也切换到内层的栈。新栈的 SS 及 ESP 的初值，从当前 TSS 字段中获取（见第 4 章）。老的 SS 及 ESP 值被压入新栈，所以对应的 IRET 指令可以恢复外层级的栈指针。与通过调用门的 CALL 指令相区别的是，通过中断门或陷阱门的转移，在老栈和新栈之间不进行参数的拷贝，这时 Dword Count 字段被忽略不用。

一旦特权级及栈已被切换，如果需要，中断或异常处理的其余部分可继续进行。EFLAGS 寄存器压栈，然后，将 EFLAGS 寄存器中的 NT 及 TF 位置 0。TF=0 表示处理程序不允许单步执行。NT=0 表示处理程序返回时，IRET 返回到同一任务而不是一个嵌套任务。如果转移是通过一个中断门进行的，则 EFLAGS 中的 IF 位也被置为 0，使得进入处理程序后，INTR 中断被屏蔽。如果转移是通过一个陷阱门进行的，则 IF 位保持不变。

下一步，通过压入 CS 及 EIP 的当前值使返回指针压栈。然后，在把 RPL 字段设置到 CPL 后，用门中的选择子装入 CS，门中的偏移量装入 EIP，从而完成到处理程序过程的转移。

异常处理的最后一部分，是根据需要把出错码压入堆栈。出错码具有 16 位的值，其格式已在第 4 章描述。为了保证栈的边界对齐，实际压栈的出错码具有 32 位的值，其中高 16 位的值未做定义。

通过门完成转移之后，除了 EFLAGS 寄存器中的 IF 位的处理不同以外，中断门及陷阱门是完全相同的。对于中断门，在完成转移之后，把 IF 位设置为 0，使得在处理程序执行期间，屏蔽掉 INTR 中断；而陷阱门则保持 IF 位不变，即 IF 位原为 1，则通过陷阱门转移到处理程序之后仍允许 INTR 中断。因此，中断门最适宜于处理中断，而陷阱门最适宜于处理异常。

图 5.3 示出栈切换的操作，这种栈切换在通过中断门或陷阱门向内层转移期间发生。由

SS 及 ESP 寄存器进行寻址的外层栈示于右面，内层的新栈示于左面，压入的第一项(最高地址)是指向外层栈的指针，由老的 SS 及 ESP 的值给出，接着，老的 EFLAGS 值压栈；然后，返回地址(即外层的 CS 及 EIP 的值)压栈；最后，将出错码压栈。在处理程序的入口，SS 为内层栈寻址栈段，而 ESP 给出出错码的偏移量，该值即堆栈的栈顶。

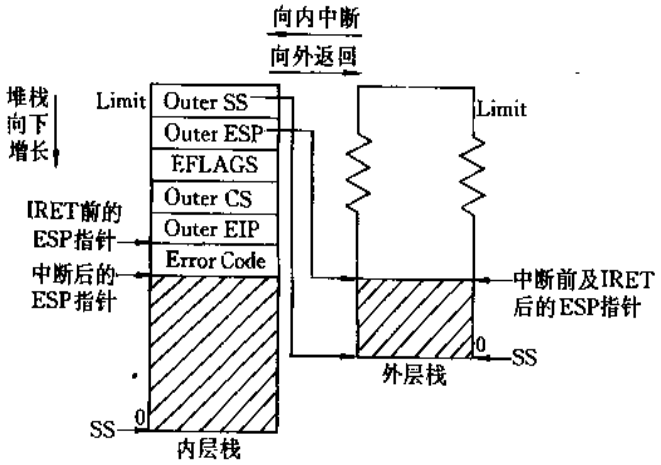


图 5.3 通过中断门或陷阱门向内层转移

5.5.2 NT=0 时的 IRET

IRET 指令用于从中断或异常处理程序的返回，由中断门或陷阱门转入的处理程序，由于在转入时 NT 位已被置 0，所以从处理程序返回时，将从当前栈的栈顶找出返回信息，并返回到同一级或外层级。

IRET 指令从栈顶弹出返回指针 EIP 及 CS，然后弹出 EFLAGS 值。弹出的 CS 选择子的 RPL 字段，确定返回后的特权级。如同第 4 章描述的 RET 指令一样，使用返回 CS 选择子的 RPL，而不是由选择子标识的段的 DPL，是为了返回到不在 DPL 给定的级执行的一致码段。

如果返回选择子的 RPL 与 CPL 相同，则不进行特权级的改变。若 RPL 规定了一个外层特权级，则需要特权改变；在这种情况下需要恢复外层栈的 ESP 及 SS 的值，把它们从内层栈中弹出，以完成 IRET。

IRET 需找到返回指针及在栈顶保存的 EFLAGS 值，所以，提供出错代码的异常处理程序，必须先从栈中传送出错代码，然后再执行 IRET 指令。

图 5.3 示出通过中断门或陷阱门的内层转移，也说明了内层 IRET 指令执行前后栈的变化，外层栈示于右面，当前栈为内层栈，示于左面。在执行 IRET 指令之前，处理程序必须从栈中弹出出错码。所以，ESP 指向压入的返回地址指针的 EIP 部分。返回指针的 EIP 及 CS 从栈中弹出，以恢复执行被中断的程序。而 EFLAGS 寄存器，则通过弹出保存的 EFLAGS 的值来恢复。对弹出的 CS 选择子的 RPL 字段进行检查，以确定是否要求进行外层的返回。在此例中，RPL 规定一个相对于 CPL 的外层级，所以，把保存的 ESP 及 SS 值弹出到对应的寄存器，以恢复外层栈。被中断的程序，用中断被接受时所保存的 SS、ESP、EFLAGS、CS 及 EIP 的值来恢复。

5.5.3 通过任务门的转移

如果由矢量号索引得到的 IDT 描述符是一个任务门，则表示要转移到不同任务的处理程序。与通过任务门的 CALL 指令一样，门中有一个 16 位的描述处理程序任务的 TSS 段的选择子。该选择子必须标识一个可用的 286TSS 或 386TSS 类型的段。通过 286TSS 段的转移在第 7 章进行讨论。通过任务门到一个可用的 386TSS，转向中断或异常处理程序的转移，等同于通过任务门到一个可用 386TSS 的 CALL 指令(详见第 4 章)，唯一的区别是此异常提供出错码，在完成任务切换之后，要把出错码压入新任务的堆栈中。为保证栈的边界对齐，16 位的出错码以 32 位的值压入，其中高 16 位的值未作定义。

通过任务门的转移，在进入中断或异常处理程序时，EFLAGS 寄存器中的 NT 位为 1，这表示在相应的 IRET 指令返回时，必须返回到一个嵌套的任务。IRET 指令将访问当前 TSS 中的链接字段，以获得要返回到的任务的 TSS 的选择子。参见第 4 章中关于任务切换及 TSS 中的链接字段，定义嵌套任务链的细节。

5.5.4 任务门与中断/陷阱门的比较

80386 的中断或异常允许使用中断门或陷阱门由当前任务之内的一个过程进行处理；也允许通过使用任务门由另外一个任务进行处理。在当前任务内的处理程序较为简单，并可以很快的转移到处理程序。但处理程序要负责保存及恢复处理器的寄存器内容。转到不同任务的处理程序要花费较长时间，这时，保存及恢复寄存器内容的开销已作为任务切换的一部分处理。使用当前任务内的处理程序的方法，在中断或异常信号产生时，对正执行任务的状态可直接进行访问，但是，这样就要求每一个任务内都包含一个处理程序。使用独立的任务来处理中断或异常，使处理程序得到较好的隔离，但在中断或异常信号产生时，对任务状态的访问变得较为复杂。

无效 TSS 异常必须由任务门进行处理，以保证处理程序有一个有效的任务环境。其它的异常通常在任务环境之内进行处理。在任务内，异常被检测并且不必屏蔽中断，所以最好使用陷阱门。由陷阱门指向的异常处理程序，是一个由所有任务共享的过程，所以该处理程序最好置于全局地址空间之内。如果各个任务要求有不同的处理程序，则全局异常处理程序，保存一个各处理程序的入口表，并为引起异常的任务调用相应的处理程序。

中断通常与正执行的任务没有关系，并可能从使用任务门提供的任务隔离中获得好处。要求较快响应的中断，通过中断门可以得到较好的处理。没有使用内装 386 任务切换机制的系统，也将使用中断门。因为中断可以在任意时刻产生，所以，通过中断门访问的中断处理程序，必须置于全局地址空间中。这些处理程序对于所有的任务都是可寻址的。第 6 章包含一个较长的例子。图示中断门的使用及处理程序如何安排在全局地址空间中。

当中断或异常被接受时，使用任务门可提供一个处理程序任务的自动调度。这种任务调度由 80386 硬件直接执行，并且越过包含在操作系统中的软件任务切换。这就为处理程序提供了一个快速的任务切换。

5.6 中断/异常的详细说明

本节介绍中断及异常控制转移方法的第二种层次：以类似 C 语言的表示法，说明中断/异

常的公共处理及进行存储器访问发生的段异常和页异常的处理。

5.6.1 中断/异常的公共处理

中断/异常的公共处理包括中断/异常的响应、转移到相应的处理程序及中断/异常的返回。这些例程分别为 Interrupt( )、IntTrapGate( )及 IRET。

5.6.1.1 中断/异常的响应

(1) 例程名：Interrupt( )

(2) 例程参数：

- VecNumber　与中断/异常相关的矢量号
- Check DPL　指示是否检查 IDT 中门描述符的 DPL

(3) 程序功能：用引起中断/异常事件的矢量号索引 IDT 表，根据 IDT 表中描述符指定的门的类型，分支到相应的处理程序。

(4) 程序清单：

```
/* 处理一个中断或异常 */
Interrupt (VecNumber, CheckDPL)
    int VecNumber,  /* 矢量号 */
       CheckDPL;    /* 检查 DPL 标志。  0:不检查 IDT 中门的 DPL,    */
                    /*                  1:  检查 IDT 中门的 DPL      */
{  SegAttributes Attributes;
   SelectorType Selector;
   int          Dword1, Dword2, Offset, Base, Limit;
   /* 越界检查 */
   if (VecNumber * 8+7>IDTR.Limit)
     SegmentException($GP, VecNumber * 8+2);
   /* 读出描述符 */
   AccessLinear (IDTR.Base+VecNumber * 8,    4,  0 /* PL  0 */, 0  /* Read */,
                 &Dword1);
   AccessLinear (IDTR.Base+VecNumber * 8+4,  4,  0 /* PL  0 */,0  /* Read */,
                 &Dword2);
   /* 获取属性、选择子及偏移 */
   Attributes=Dword2;
   Selector=Dword1>>16;
   Offset=(Dword1 & 0FFFFh) | (Dword2 * 0FFFF0000h);
   /* 测试描述符 DType 及 Type
   if  (
        (Attributes.DType==0)  /* 描述符不能是存储段 */
      || ((CheckDPL==1) && (Attributes.DPL<CPL) /* 如检查 DPL,必须有 DPL≥CPL */
      ||  (Attributes.Type<5)                   /* 属性类型必须是 5、6、7、14、15 */
      || ((Attributes.Type>7) && (Attributes.Type<14))
       )
     SegmentException ($GP, VecNumber * 8+2);   /* 否则产生异常 */
   /* 检查存在位 */
   if (Attributes.P==0)
     SegmentException ($NP,VecNumber * 8+2);
   /* 根据门的类型进行分支处理 */
   switch (Attributes.Type)  {
    5:     /* 任务门 */
      TaskGate (Selector, 1);    /* 参数 1 表示链接 */
      break;
    6:     /* 286 中断门 */
```

```
        IntTrapGate286(Selector, Offset, 1);   /* 参数 1 表示清中断标志 */
        break;
    7:      /* 286 陷阱门 */
        IntTrapGate286(Selector, Offset, 0);   /* 参数 0 表示不改变中断标志 */
        break;
    14:     /* 386 中断门 */
        IntTrapGate 386 (Selector, Offset, 1);   /* 清中断标志 */
        break;
    15:     /* 386 陷阱门 */
        IntTrapGate 386 (Selector, Offset, 0);   /* 不改变中断标志 */
        break;
    }
}
```

Interrupt( )的程序流程见图 5.4。

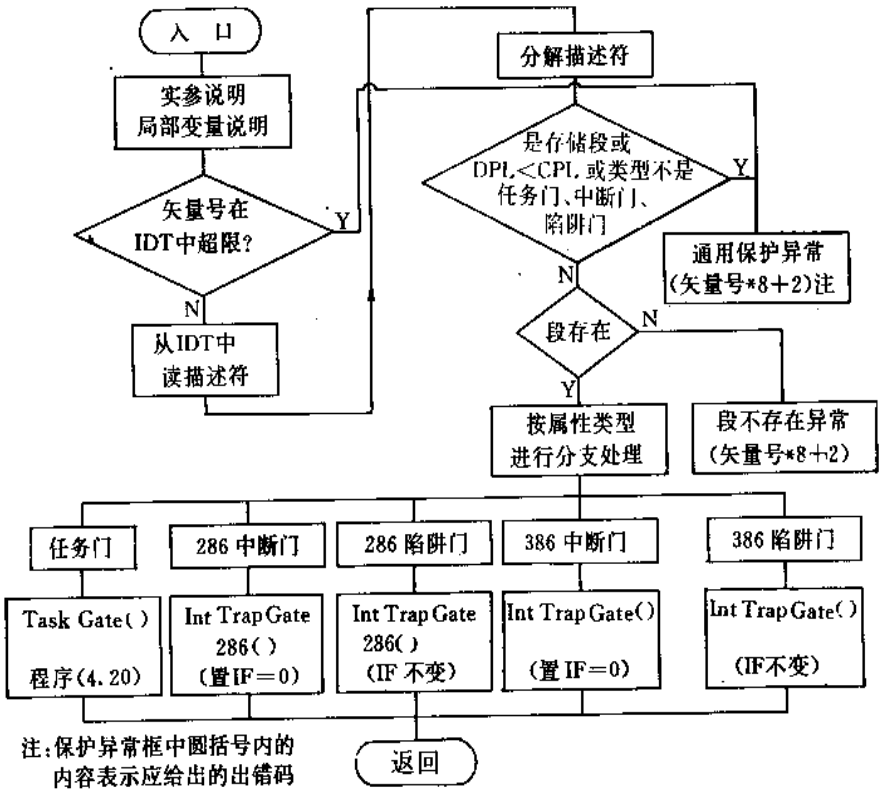


图 5.4 Interrupt( )的程序流程

(5) 执行过程:

第一,用 VecNumber 检索 IDT 表,得到门的描述符。检索中,要检查矢量号 VecNumber 的合法性。如矢量号指定的描述符超出 IDT 表的范围,则产生通用保护异常;如矢量号指定的描述符在 IDT 表的界限范围之内,则读出相应的门描述符。

第二,对读出的门描述符进行检查,这些检查包括:

① 描述符指定的段必须是一个非存储段。

② 如果传入的参数 Check DPL 为 1,则对 IDT 表中矢量号指定描述符的门的 DPL 进行检查,以保证 DPL 处在相对于 CPL 的同级或外层级,即应有 DPL≥CPL。

③ 类型字段的值必须为5、6、7、14或15这些值之一。这些值分别表示该描述符是一个任务门、286中断门、286陷阱门、386中断门或386陷阱门。

④ 描述符的P位必须为1，指示该描述符是一个有效的描述符。

上列各项检查中，前三项为合法性检查，其中任有一项检查失败，都产生$GP(通用保护)段异常。最后一项检查为有效性检查，如检查失败，则产生$NP(不存在)段异常。$GP及$NP异常都提供一个出错码，该出错码包含选择子索引字段中的矢量号，并把出错码的位1置为1，表示出错码与IDT有关。

最后，根据门中类型字段的值，确定响应中断应进行的进一步处理：

① 类型字段的值为5，表示要进行任务切换，并切换到由门中的选择子确定的TSS所描述的任务，此时分支到TaskGate( )程序(见第4章)。

② 类型字段的值为6或7，表示进行286方式的中断/陷阱处理，并分支到IntTrapGate 286( )程序(见第7章)。IntTrapGate 286 ( )例程中的第三参数，用于区分此次IntTrapGate 286( )调用是286中断门(类型值为6)调用，还是286陷阱门(类型值为7)调用。

③ 类型字段的值为14或15，表示进行386方式的中断/陷阱处理，并分支到IntTrapGate ( )程序。

5.6.1.2 386中断/异常的处理

(1) 例程名 IntTrapGate( )

(2) 例程参数：

- Selector 代码段选择子。选择子指定代码段中包含有处理程序的代码；
- Offset 处理程序入口地址偏移量；
- ClearIF 区分中断门/陷阱门的标志。

  Clear IF=1，表示中断门。

(3) 程序功能：完成到中断/陷阱处理程序的转移。

(4) 程序清单：

```
IntTrapGate (Selector, Offset, ClearIF)
SelectorType Selector;
Int          Offset,
             ClearIF;  /* 清除IF位的标志。   1: 清除IF位 */
                       /*                    0: IF不变 */
{
   SegAttributes Attributes;
   SelectorType  GSelector;
   int           Base, Limit, GOffset;
   /* 选择子测试，为空时产生异常 */
   if ( (Selector & 0FFFCh)==0)
     SegmentException ($GP, 0);
   /* 读出并测试描述符 */
   ReadDescriptor (Selector,&Attributes, &Base,&Limit,&GSelector,&GOffset);
   /* 描述符类型不是存储段时产生异常 */
   if (Attributes.DType==0)
     SegmentException($GP, Selector);
   /* 切换到内层栈及内层CPL */
   if ( (Attributes.Type>=8) && (Attributes.Type<=11)
      && (Attributes.DPL<CPL) && (Attributes.P==1) )
            InnerStack (Attributes.DPL, 0,Selector);
   /* 忽略门中选择子的RPL，装入CS描述符 */
```

```
  Selector.RPL=0;
  CSDescriptorLoad(Selector, Attributes.Base, Limit, $GP);
  /* 越界检查 */
  if (Offset>CS.Limit)
     SegmentException ($GP,0);
  /* 标志寄存器及返回地址压栈 */
  /* 修改标志寄存器,装入CS选择子及EIP */
  ESP=ESP-4;
  AccessVirtual (SS,ESP,4,1 /* Write */, $SS, &EFLAGS);
  ESP=ESP-4;
  AccessVirtual (SS,ESP,4,1 /* Write */, $SS, &CS.Selector);
  ESP=ESP-4;
  AccessVirtual (SS,ESP,4,1, /* Write */, $SS, &EIP);
  EFLAGS.TF=0;
  EFLAGS.NT=0;
  if (Clear IF==1)
    EFLAGS.IF=0;

  CS.Selector        =Selector;
  CS.Selector.RPL    =CPL;
  EIP                =Offset;
}
```

IntTrapGate( )的程序流程见图5.5。

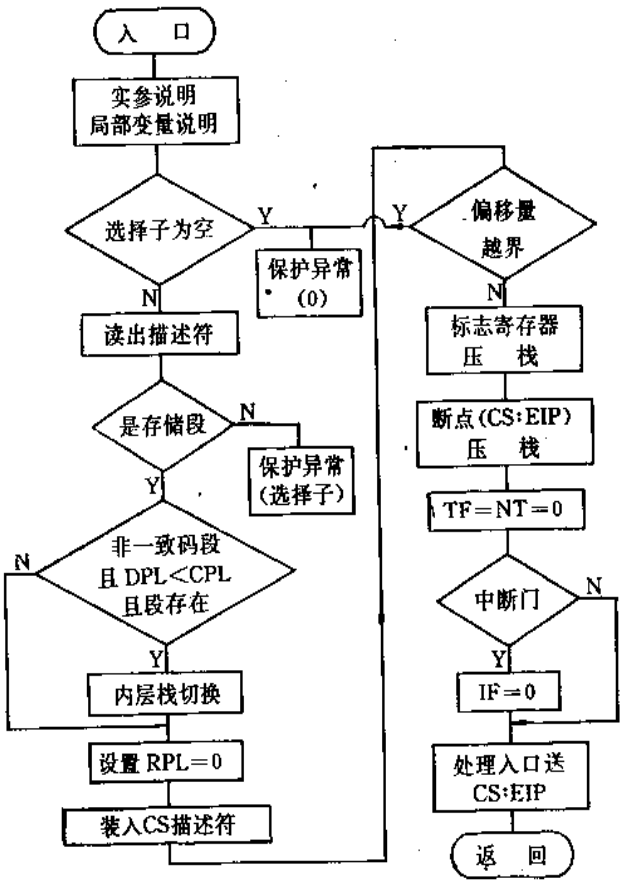


图5.5 IntTrapGate( )的程序流程

(5) 执行过程：

第一，对参数 Selector 进行测试，在选择子非空的情况下，用例程 ReadDiscriptor( )(见第4章)读出描述符。因为选择子指定处理程序的代码段，因此，读出的描述符表示一个存储器段，即有 DType 字段必须为1。如果描述符指定的代码段是一个存在的、非一致可执行的代码段，且有 DPL<CPL，则要进行特权级的转换，并调用 InnerStack( )例程(见第4章)完成栈的切换。

第二，调用 CSDescriptorLoad( )例程装入描述符，并完成对描述符的检查。由于门中选择子的 RPL 应予忽略，所以在调用 CSDescriptorLoad( )之前，已先将 RPL 置为0。注意：最后对偏移量参数 Offset 进行检查，如果 Offset 在 CS 段的界限之内，则执行中断处理的"可见"部分：即将 EFLAGS、CS 及 EIP 压栈；清除 EFLAGS 字段中的某些位；用门中的选择子及偏移量装入 CS 及 EIP。完成 CS 及 EIP 的装入，即完成了转到中断/异常处理程序的控制转移，此后，即实际进行中断/异常的处理。

5.6.1.3 中断/异常的返回——IRET 指令

(1) 例程名 IRET( )

(2) 程序功能：结束中断处理，返回被中断的程序

(3) 程序清单：

```
/* IRET 指令的详细说明 */
IRET ( )
{
   SelectorType  Selector, GSelector;
   SegAttributes  Attributes;
   int            Base, Limit, Offset, GOffset;
   /* 测试 NT 位，确定是否嵌套返回 */
   if (EFLAGS.NT==1)  /* 嵌套返回的处理 */
     {
      AccessLinear (TR.Base, 2, 0 /* PL 0 */, 0 /* Read */, &Selector);
      TaskGate (Selector, -1 /* Unlink */);
     }
   else {     /* 非嵌套返回，从栈中弹出中断返回信息 */
     AccessVirtual (SS, ESP, 4, 0 /* Read */, $SS, &Offset);
     ESP=ESP+4;
     AccessVirtual (SS, ESP, 4, 0 /* Read */, $SS, &Selector);
     ESP=ESP+4;
     AccessVirtual (SS, ESP, 4, 0 /* Read */, $SS, &NewEFLAGS);
     ESP=ESP+4;
   /* 选择子测试 */
   if (  ((Selector & 0FFFCh) ==0)  ||  (Selector.RPL<CPL))
     SegmentException ($GP, 0);
   if (Selector.RPL>CPL) {
     /* 返回到外层，恢复外层栈 */
     OuterStack (Selector.RPL, 0);
     }
   /* 读出并测试描述符，描述符是系统段或门时产生异常 */
   ReadDescriptor (Selector, &Attributes, &Base, &Limit, &GSelector, &GOffset);
   if (Attributes.DType==0)
     SegmentException ($GP, Selector);
   /* 装入 CS 描述符 */
   CSDescriptorLoad (Selector, Attributes, Base, Limit, $GP);
   /* 越界检查 */
```

```
    if (Offset>CS.Limit)
      SegmentException ($GP, 0);
    /* 装入标志寄存器、CS 选择子及 EIP(即被中断程序的断点地址) */
    EFLAGS=NewEFLAGS;
    CS.Selector=Selector;
    EIP=Offset;
    }
}
```

IRET 的程序流程见图5.6。

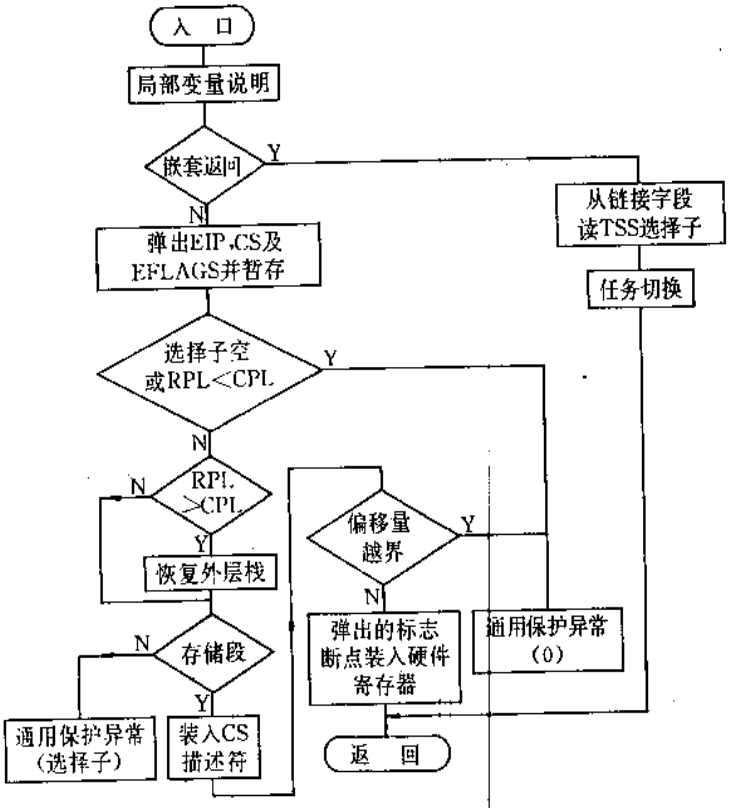


图 5.6 IRET 的程序流程

(4) 执行过程:IRET 指令通过测试 EFLAGS 寄存器的 NT 位,确定是嵌套返回还是当前任务内的返回。如 NT=1,则执行嵌套任务的返回。当前 TSS 中的链接字段保存有前一任务的 TSS 的选择子,读出该选择子,并调用例程 TaskGate( )以完成任务切换。

如 NT=0,则执行当前任务内的返回。其操作有:

- 从栈中弹出 EIP、CS、EFLAGS,返回后的特权级由弹出的 CS 选择子中的 RPL 确定;
- 如 RPL>CPL,则调用 OuterStack( )例程(见第4章)返回到当前 CPL 的外层级;
- 调用 ReadDescriptor( )例程读出描述符,及调用 CSDescriptor( )例程测试描述符并装入 CS 投影寄存器。
- 将已弹出的 EFLAGS、CS 及 EIP 装入相应寄存器完成 IRET。

因为 CS、EIP 中装入的值是中断时保存在栈中的断点,恢复了断点处有关寄存器被中断时的状态,保证了中断后的正确返回。

5.6.2 存储访问异常

在第4章，我们已使用例程 SegmentException( )及 PageException( )，用来报告段及页访问异常。本节要对这两个例程做详细的说明。在说明之前，先用如下的程序清单介绍几个全局变量及基本指令的取—执行循环。

```
/* 几个全局变量及基本指令的取—执行循环 */
/* EXT 及 OldEIP 为全局变量，在取出并执行新指令时赋值。*/
/* 这里的一小段程序规定 EXT、EIP 及 OldEIP 在一条指令内的处理。*/
int EXT;  /* 处理外部中断或异常时为1，用于为段异常形成出错码 */
int OldEIP;  /* 保存当前指令的 EIP，当执行该指令时，*/
             /* 如发生故障型的异常，则在异常处理完成以后，*/
             /* 重新执行由 OldEIP 所指向的指令 */
/* 以下为取出并执行一条指令的操作。
/* 异常处理例程通过转到此处的跳转，使正执行的指令 */
/* 被中止，并开始执行异常处理程序的第一条指令 */
DispatchInstruction:
    EXT=0;
    OldEIP=EIP;              /* 保存当前指令的指针 */
    FetchInstruction( );     /* 使 EIP 指向下一条指令 */
    Execute ( );             /* 执行当前指令 */
    Goto DispatchInstruction; /* 下一次的取—执行过程 */
```

全局变量 EXT 用于段异常时形成出错码。通常 EXT=0，当处理外部中断或处理除了 INTn 或 INTO 软件陷阱指令以外的异常时，EXT 被置为1。OldEIP 保存当前指令在 CS 寻址的代码段内的偏移量；如果发生故障，则 Old EIP 的值送回 EIP，所以在进行异常处理时，保存的 EIP 或在当前任务的栈中或在嵌套任务的 TSS 中，其值为故障指令所在地址，即保存的 EIP 指向引起故障的指令。

基本指令的取—执行循环从标号 Dispatch Instruction 开始。每条指令开始之前，先将 EXT 置为0，EIP 送 OldEIP，然后调用取指令例程——即例程 FetchInstruction( )(本书未给出)取出 EIP 所指向的指令，并根据所取指令的字节数对 EIP 进行相应增量，使其指向下一条指令。指令取出后，再通过调用 Execute( )来执行指令。根据取出的指令的不同，Execute 实际上有不同的内容，例如取出的指令被解释为 IRET 指令，则 Execute( )即为 IRET ( )。如果指令的执行成功地完成，则重新开始下一次取—执行循环。由于取—执行循环比较简单，我们就略去了用流程图来做进一步的解释。

在第4章已经说明，例程 PageException( )及 SegmentException( )不返回其调用者，而代之以中止指令的执行。中止的办法通过跳转到标号 DispatchInstruction 来完成。这一点很难用标准的 C 结构来描述，但对多数编程人员则是很熟悉的。在详细说明中未包括检测及报告双重故障，这是为了避免使讨论过于复杂化。

5.6.2.1 段异常例程

(1) 例程名：SegmentException( )

(2) 例程参数：

- VecNumber　异常的矢量号
- Errorcode　出错码，由引起段异常的选择子形成

(3) 程序功能：产生段异常

(4) 程序清单：

```
/* 产生段异常的子例程 */
SegmentException (VecNumber, ErrorCode)
    int VecNumber;  /* 异常号          */
    int ErrorCode;  /* 出错码的高15位 */
{
    /* 出错码的末位按 EXT 的值设置 */
    ErrorCode=(ErrorCode & 0FFFEh) | EXT;
    EXT=1;
    /* EIP 回退使其指向产生故障的指令 */
    /* 即介绍全局变量及基本指令取一执行循环的程序清单 */
    /* 中的 OldEIP                                     */
    EIP=OldEIP;
    /* 用 VecNumber 中给出的矢量号发异常信号 */
    Interrupt (VecNumber,0);   /* 不检查 IDT 门中的 DPL */
    /* 出错码压栈 */
    ESP=ESP-4;
    AccessVirtual (SS,ESP,4,1 /* write */, $SS, &ErrorCode);
    /* 终止故障指令的执行,转移到异常处理程序的第一条指令 */
    goto DispatchInstruction;
}
```

SegmentException( )的程序流程见图5.7。

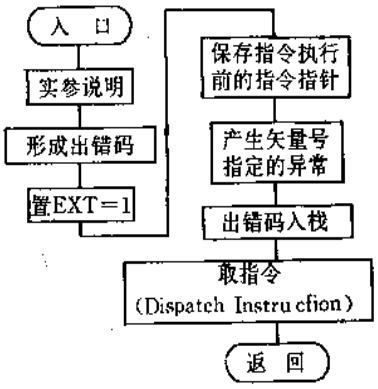


图 5.7 SegmentException( )的程序流程

(5) 执行过程:

第一,形成异常的出错码。该出错码由两部分组成,一部分为传递给例程的第二个参数ErrorCode的高15位(调用者在此处通常是给出某描述符的选择子),另一部分仅一位,即 EXT 位(段异常的出错码格式见第4章)。

第二,将已在取一执行循环中增量后的 EIP 回退,使其指向故障指令,其动作是将 OldEIP 的值赋予 EIP。

第三,调用例程 Interrupt( )完成中断/异常公共处理,最后,将出错码入栈。中止故障指令,并跳转到取一执行循环的起始标号,重新开始执行指令。由于在 Interrupt( )例程及随后的 IntTrapGate( )例程,已完成中断/异常的各种公共处理,并在 CS、EIP 中已置入处理程序的入口地址(此时,置入的是段异常处理程序的入口地址),所以,接下去执行的程序,是执行实际的异常处理程序。

5.6.2.2 页异常例程

(1) 例程名 PageException( )

(2) 例程参数:

- LAddress 引起异常的线性地址;
- U 欲进行访问的特权级,用于形成出错码的 U 位;
- W 访问方式位,W=1表示进行写访问,W=0为读访问,用于形成出错码的 W 位;
- Present 被访问页对应描述符的存在位,P=1表示页存在,P=0表示页不存在。Present用于形成出错码的 P 位。

(3) 程序功能：完成页异常的公共处理

(4) 程序清单：

```
/* 页异常处理子例程 */
PageException (LAddress,U,W,Present)
int LAddress,
    U,          /* 用户程序异常时为1,否则为0 */
    W,          /* 写访问时为1，读访问时为0 */
    Present;    /* 表项存在时为1,表项不存在为0 */
{
  int ErrorCode;
  /* 建立出错信息。产生页故障的线性地址存入 CR2,并 */
  /* 根据 U、W 及 Present 的值产生出错码 */
  CR2=LAddress;
  if (U==3)
    ErrorCode=4+2*W+Present;
  else
    ErrorCode=   2*W+Present;
  EXT=1;
  /* 使 EIP 回退至本指令的指令地址值,如本指令产生故障 */
  /* 则 EIP 指向了产生故障的指令。该地址已保存在 OldEIP 中 */
  EIP=OldEIP;
  /* 产生矢量号为14的异常 */
  Interrupt (14,0);  /* 不检查 IDT 门中的 DPL */
  /* 出错码压栈 */
  ESP=ESP-4;
  AccessVirtual (SS,ESP, 4, 1, /* write */, $SS, &ErrorCode);
  /* 终止产生异常的指令,转去执行异常处理程序的第一条指令 */
  goto DispatchInstruction;
}
```

例程 PageException( )的程序流程见图5.8。

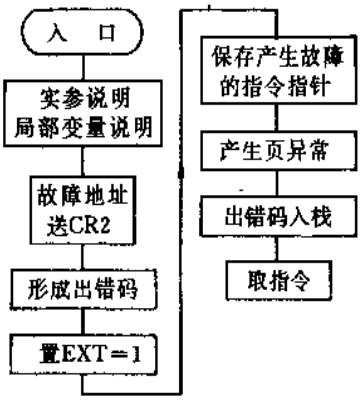


图 5.8 PageException( )的程序流程

(5) 执行过程：

第一，将引起页异常的线性地址保存在 CR2寄存器。以便在异常处理程序中，可以从 CR2中得到故障地址。

第二，根据参数 U、W 及 Present 形成出错码，算法为

ErrorCode=4*用户方式+2*W+Present

其中，用户方式=1，U=3，

用户方式=0，U=0、1、2，

第三，使 EIP 回退，重新指向引起页异常的指令。

第四，调用例程 Interrupt 完成对中断/异常的公共处理。最后，将出错码入栈，中止故障指令，并跳转到取一执行循环的起始标号，开始执行实际的页异常处理程序。

5.7 协处理器出错异常

下面讨论由80387数字协处理器检测到的异常。在第2章我们讨论了80387的状态字寄存器

及控制字寄存器，在这些寄存器中包含有对于异常处理来说是很重要的字段。第3章则列出了每条指令可能产生的异常。

协处理器出错异常可归为六种类别，见表5.4。表中给出了第3章指令说明中使用的异常的代码、对应的状态字寄存器中状态位的名字及对应的控制字寄存器中屏蔽位的名字。

表 5.4 协处理器出错异常

代码	状态	屏蔽	条　件
I,IS	IE	IM	无效，操作
D	DE	DM	非规格化操作数
Z	ZE	ZM	被零除
O	OE	OM	溢出
U	UE	UM	下溢
P	PE	PM	精度

5.7.1 屏蔽与未屏蔽的异常

控制寄存器中的屏蔽位决定了检测异常条件后采取的动作。如果屏蔽位为1，则异常被屏蔽，并在80387之内通过返回一个合理的结果值进行处理，然后程序继续进行。如屏蔽位为0，则异常是未屏蔽的，并以矢量号为16向系统报告。

注意，即使不用向系统报告协处理器的出错异常，仍需为屏蔽异常设置状态位，其目的是指示协处理器的中间计算中曾经发生过某种异常。因此，应用软件可以通过清除状态位、设置屏蔽位，然后进行一系列计算。由于设置了某种(或某些)异常的屏蔽位，在计算过程中如发生已屏蔽的异常，这样的异常并不会干扰预定的计算过程。但是，如果应用程序想要知道在这一系列的计算过程中是否发生过被屏蔽的异常，就可以在这一系列的计算完成以后，通过测试状态位达到这一目的。

如果一个未屏蔽的异常发生，则在下一次执行浮点指令或 WAIT 指令时作为故障报告。也就是说，保存在协处理器出错处理程序入口处的断点 CS 及 EIP 指向引起异常之后的下一条浮点指令，而不是引起异常的浮点指令本身。80387出错指针寄存器可以用指令 FSTENV 或 FSAVE 读出，以获得一个指针(在 FCS 和 FIP 中)指向引起协处理器出错异常的指令、该指令的操作码以及存储器操作数(如果有存储器操作数)。

未屏蔽异常进一步分为执行前或执行后异常。执行前异常在结果被存储之前报告，这时指令的源操作数未发生改变。执行后异常在存储值之后报告，这时，指令的源操作数可能有一个或多个已发生改变。执行前异常类似于一个故障，只有保存的返回指针不是指向引起故障的指令这一点与故障不同。类似地，执行后异常类似于一个陷阱。

异常 I、IS、D 及 Z 总是执行前异常；异常 P 总是执行后异常；异常 U 及 O，对于存储到存储器的指令(FST (P)、FIST(P)及 FBST(P))，是执行前异常，否则异常 U 及 O 是执行后异常(包括存储到累加器栈)。

协处理器出错处理程序可能只是简单地把出错显示在屏幕上或通知给排错程序，然后终止引起异常的程序。如果协处理器出错处理程序希望恢复引起异常的程序执行，则必须完成引起异常的浮点指令。其方法是在返回之前，对于执行前异常，解释该指令；对于执行后异常，调

整结果。异常处理程序可以读出状态字寄存器，并借此确定发生了哪一种未屏蔽异常，并可以使用指令指针寄存器 FCS 及 FIP 读出并译码指令。从协处理器出错处理程序返回之前，未屏蔽状态字的异常状态位，必须用执行指令 FCLEX、FLDENV 或 FRSTOR 的方法将其清除。

由80387提供的屏蔽异常响应在多数情况下都是最好的。我们建议，除无效操作以外的所有异常都是屏蔽异常。无效操作异常只有在合理的结果无法得到时的情况下才会发生(例如加+∞或零除以零)。

5.7.2 协处理器出错类别

有六种协处理器出错异常的类别，引起异常的浮点指令又分为计算指令或非计算指令两类。计算指令包括数据传送、算术运算，比较及超越指令等。下列指令为非计算指令类：

(1) 指令 FLD，带一个寄存器操作数或一个临时的实地址存储器源操作数；

(2) 指令 FST(P)，带一个寄存器操作数或一个临时的实地址存储器目的操作数；

(3) 指令 FXCH、FABS、FCHS 及 FXAM。

5.7.2.1 无效操作异常

有两类无效操作异常：一类为由于数字错误的无效操作异常，另一类为由于80387累加器栈的上溢或下溢造成的无效操作异常。状态字寄存器的 SF 位用于区分这两种情况。前者为 SF=0，后者为 SF=1。无效操作异常是一种执行前异常，所以在存储结果之前报告非屏蔽异常。

1. 无效数字操作(SF=0) 表5.5列出的任何一种情形都触发一个无效操作异常。如果无效操作异常是屏蔽的，则返回不定的静态 NaN 值(FIST(P)存储整型不定值及 FBSTP 存储 BCD 不定值除外)，对于返回两个结果的操作，这两个结果都以不定的静态 NaN 返回。特别的情形，见表5.5中的脚注说明。

如果无效操作异常是未屏蔽的，则产生协处理器出错异常，并且不存储结果，源操作数对于异常处理程序是可用的。

表 5.5 无效数字操作

计算指令的操作数的格式不正确
计算指令的操作数是一个 NaN
FCOM 或 FTST 的操作数是 NaN 或静态 NaN
FUCOM 的操作数是一个 NaN
FADD 的两个操作数为无穷大且符号相反
FSUB 的两个操作数为无穷大且符号相同
FMUL 的两个操作数为零和无穷大
FDIV 的两个操作数都为零或都为无穷大
FPREM 或 FPREM1 (注2) 的除数为0或被除数为无穷大
FCOS、FPTAN、FSIN 或 FSINCOS 的操作数为无穷大
FSQRT 或 FYL2X 的操作数小于零
FYL2XP1的操作数小于−1
FIST 或 FBSTP 的操作数是 NaN 或无穷或在目标中不适合

注1 如果无效操作异常被屏蔽，则条件码设置为"unordered"(不定)：C3、C2、C0全置为1

注2 如果无效操作异常被屏蔽，则 C2设置为1。

2. 累加器栈下溢/上溢(SF=1)　无效操作异常也可由累加器栈的下溢或上溢而触发。如果一条指令试图读一个标志为空的累加器，则发生下溢；而一条指令试图把结果压入累加器栈，而新栈的栈顶不标志为空，则发生上溢。但是，指令FST(P)或FXCH的目的寄存器不认为是压栈，所以这些指令可以对非空寄存器进行写入。状态字寄存器中C1条件代码位用于区分下溢或上溢：C1=1表示上溢，C1=0表示下溢。

5.7.2.2　非规格化异常

如果计算指令的源操作数是"非规格化"的，则报告非规格化异常。如果异常是屏蔽的，则状态字寄存器中的DE位被置位，且指令使用非格化的数继续进行。如果异常是未屏蔽的，则状态字寄存器的DE位被置位，产生处理器异常。因为非规格化异常是执行前异常，所以不存储结果。

非规格化数比规格化数的有效数位数要少，所以在某些算法中，必须避免使用非规格数，以防止计算精度下降。然而，在多数情形下，此异常应是屏蔽的，使得仍使用非规格化的操作数继续进行计算。

5.7.2.3　零除异常

如果指令FDIV或指令EYL2X试图用一个确定的操作数除以零，或者指令FXTRACT的操作数为0，则报告一个零除异常。如果异常是屏蔽的，则对于指令FDIV或FYL2X，结果为无穷大，结果的符号为两操作数的异或值。对于指令FXTRACT，结果为负无穷大。对于所有这些指令，状态字中的ZE位都置为1。如果异常不是屏蔽的，则ZE位置1、产生协处理器异常，而且不存储结果，因为零除异常是一个执行前的异常。

5.7.2.4　溢出异常

如果操作的最后结果太大，目的操作数格式无法表达，则报告一个溢出异常。如果异常是屏蔽的，则状态字寄存器中的OE位被置位，并根据控制字寄存器中的RC字段选择的舍入方式(见表5.6)存储结果。

表5.6　屏蔽的溢出结果

舍入方式	结果符号	屏蔽结果
邻近舍入	+	+∞
	−	−∞
向−∞方向舍入	+	最大的无穷正数
	−	−∞
向+∞方向舍入	+	+∞
	−	最大的无穷负数
向0值方向舍入	+	最大的无穷正数
	−	最大的无穷负数

如果异常是未屏蔽的，则OE位被置位，并报告一个协处理器异常。如果指令FST中产生溢出，且目的操作数为存储器，则不把结果存储到存储器中，这就使得异常处理程序可以采取适当的动作，使源数据在累加器栈中仍然可用。

如果目的操作数是累加器栈，则先存入结果，再报告协处理器出错异常。所存入的结果为

实际结果舍入后再除以2^{24576}。这里，$24576=3*2^{13}$。读者可以看出，通过对阶的再偏置，这时的阶的范围已得到扩展。

5.7.2.5 下溢异常

如果操作的最后结果太小，目的操作数格式无法表达，则报告一个下溢异常。如果异常是屏蔽的，则最后结果是非规格化的，并存入目的操作数。对于屏蔽异常，仅当非规格化的结果损失精度的情况下，状态字寄存器中的UE位才置位。这时，还同时产生精度异常。

如果下溢异常是未屏蔽的，则不管结果的精度是否损失，都要把UE位置1。如果执行目的操作数为存储器的FST指令时发生下溢异常，则结果不存入存储器，这就使得异常处理程序可以采取适当的动作，使源数据在累加器栈中仍然可用。

如果目的操作数是累加器栈，则先存入结果，再报告协处理器出错异常。存入的结果为实际结果舍入后再乘以2^{24576}。这样，通过对阶的再偏置而扩大了阶的范围。

5.7.2.6 精度异常

如果结果不能在目的操作数中精确地表示出来，或下溢结果有精度损失，则报告精度异常。请读者注意，状态字寄存器中的条件代码位C1用于指示计算指令中最后一次舍入的舍入方向，C1=0表示最后一次舍入操作为'舍'，所以计算的结果比实际应得的结果要小；C1=1表示最后一次舍入操作为'入'，所以计算的结果比实际应得的结果略大。如果异常是屏蔽的，则状态字寄存器中的PE位置位，经舍入后的结果存入目的操作数；如果异常是未屏蔽的，除屏蔽异常的动作以外，还产生一个协处理器出错异常信号。

当计算不精确时，精度异常可用来引起人们的注意，这在有些情况下（例如计算货币值时）是很重要的。在多数应用程序中，精度异常最好是屏蔽的。

5.7.3 协处理器出错异常的优先级

如果一条指令中检测到多个异常条件，不是将所有异常都报告给系统，而是只报告其中的一个。具体应报告多个异常中的哪一个，则由表5.7中给出的协处理器出错异常优先顺序决定。

表 5.7 协处理器出错异常的优先级

异　常	优先级
无效操作 · 栈下溢 · 栈溢出 · 格式不被支持 · 其它 零除 非规格化 溢出/下溢 精度	最高 ↓ 最低

第6章 程序举例

6.1 应用程序举例

为了加深对指令和第1、2章内容的理解、我们收集了11个程序实例(6个是关于整数指令的,5个是关于浮点指令的),进行分析。

此外,我们还在本章介绍多段指令、操作系统指令有关的实例。

6.1.1 语法

随着本书内容的不断深入,所涉及的语法知识面越来越大。回想一下,第2章介绍过寄存器寻址和内存寻址的语法,第3章叙述了多操作数指令格式以及80386、80387的全部汇编语言指令,本章又要和复杂的、能实现某些通用功能的指令序列打交道,所以,扩充更多的汇编语言语法知识是必要的。

可能有些读者会觉得本书介绍的汇编与他的机器上的汇编不太一样。这种由于机器的不同而带来的汇编之间的差异是正常的、不可避免的,但是不会妨碍我们对程序或对80386、80387汇编指令的理解。

6.1.1.1 类型

变量和内存访问的类型,实质上指的是被访问的字节数。只有明确类型信息,才能使汇编产生正确的指令。这是使用变量和访问内存首先要解决的问题。

有时类型信息可以从指令或其操作数得到。但在一般情况下,要求程序员对类型作显式规定。请看,

```
INC [ESI]
```

就是一条会引起混乱的指令,因为没有说明[ESI]究竟是字节,是字,还是双字。这样的指针被称为"无特征指针"。正确的指令应选用下面几种格式:

```
INC  byte ptr [ESI]
INC  word ptr [ESI]
INC  dword ptr [ESI]
```

再有,指令

```
MOV [EBX], 5
```

同样具有歧义性,因为立即数并不包含类型信息。下面写法就对了:

```
MOV word ptr [EBX], 5
```

双操作数格式,需由一个操作数,而且仅允许一个操作数来规定访问的类型。下面的指令

```
MOV [EBX], EAX
MOV [EBX], AX
MOV [EBX], AL
```

都是正确的,因为第二个操作数已指明了操作数的类型,也就是说我们从第二个操作数得到了明确的类型信息。表6.1列出了所有合法的类型:

表6.1 合法类型

类型	说　明
byte	1字节
word	2字节
dword	4字节
pword	6字节
qword	8字节
tword	10字节
short	段内标号,从－128字节到＋127字节
near	段内标号,(短的或长的)
far	段间标号,(在别的段内的标号)

读者要注意ptr,指针pointer的缩写,其意义也是不明确的。例如,byte ptr给了内存操作数(如[EBX])以类型,但它决不意味着[EBX]即是个内存指针。请你看一下双操作数格式就明白了,在MOV [EBX],EAX指令中,内存操作数[EBX]并无个性特征。

6.1.1.2 数据单元

你可以用db、dw或者dd来预定义某些存储器的字节、字或双字。在通常的定义格式中应包含存储器的变量名(通过它来访问的符号名)、大小和初始值。例如

```
fred db1          ;定义1字节的存储器,变量名为fred,初始值为1
joe dw 10,8,7     ;定义3个字长的存储器,变量名为joe,初始值为10,8,7
    dd ?          ;定义一个双字存储器,没有变量名,没有初始值
```

数据单元的初始化有两种规定的格式。第一种格式,以一个字符串来初始化用db定义的一串连续字节,如

```
string1 db   'Hello World'
```

它定义了11个字节的存储器,变量名为string1,初始化为一个字符串'Hello World'。

第二种格式是规定一个可以嵌套的DUP结构,来初始化多个存储器单元。如:

```
stack dd 50 dup (0)   ;它定义了50个双字存储器,变量名为stack,全部初始化为0
funny dw 2 dup (3,2dup (?),4); 这是个嵌套的DUP结构。它定义了8个字的存储器,变量名为funny,
                               初始化为3,?,?,4,3,?,?,4
```

6.1.1.3 偏移量

偏移量操作符(Offset)返回一个变量或标号的段内偏移量。

例如:

```
string 1 db   'Hello World'
string1_end db 0
string 2 db 'Goodbye World'
string2_end db 0
<----->
MOV ESI,Offset string 1
MOV EAX,Offset string 1－Offset string 2
```

第一个代码语句将变量string1的(在定义string1的段内的)偏移量送入ESI寄存器。第二个代码语句表示Offset提供的偏移量还可以参加算术运算。

6.1.1.4 标号

标号是对于代码序列中的单元的符号访问。标号提供了一种直接跳转到某个单元的极为方便的手段。

```
label1: MOV EAX,1
        ⋮
        LOOPNZ label1
```

6.1.1.5 PROC/ENDP

PROC 伪指令的作用和定义一个标号相当，它定义了一个称之为子程序的指令序列。描述这个子程序的指令序列以 ENDP 伪指令结束。这个子程序通常用 call 指令来调用。

```
JunkProc  PROC
    ⋮
JunkProc  ENDP
    ⋮
call JunkProc
```

6.1.1.6 EQU

EQU 汇编伪指令可以使一个符号与某个表达式等同起来。你可以用这个指令为常数、标号、地址，或寄存器规定各种各样的助记符。如：

```
reg1 EQU EAX
reg2 EQU EBX
```

用 reg1 和 reg2 来代替通用寄存器名。

其它的例子有：

```
Length EQU 100
My_word EQU word ptr [ESI+EBP]
```

在这一章中，你可以看到 EQU 伪指令被用于对和堆栈有关的临时或局部变量的助记符号访问。例如

```
Error EQU SS: byte ptr [EBP-1]
```

定义了名叫 Error 的一字节变量，使它位于堆栈段内，寻址办法为 SS 加上 EBP 寄存器内容减一的偏移量。

6.1.2 整数的例子

在这一节里，我们给出若干整数指令序列的例子。

6.1.2.1 符号除

在第 3 章关于 SAR(算术右移)指令的说明中，我们注意到 SAR 等效于除数为 2 的幂的符号除法，而商的舍入是朝着负无穷的方向，即舍入后的商总比商的真值要小。这是 SAR 指令和 IDIV 指令的相异之处，IDIV 指令的商的舍入总是朝着 0 的方向，即舍入后的商的绝对值总比商的真值的绝对值小。例如，−17 被 4 除，若用 SAR 指令得商为−5，而用 IDIV 指令得商为−4。

```
IDIV ECX
  执行前: EDX: FFFFFFFF  EAX: FFFFFFEF
          ECX: 00000004
  执行后: EDX: FFFFFFFF  EAX: FFFFFFFC
          ECX: 00000004
```

SAR EAX,CL

执行前：EAX：FFFFFFEF　ECX：00000002

执行后：EAX：FFFFFFFB　ECX：00000002

IDIV 指令的执行速度太慢，以寄存器为源操作数的 IDIV 指令要执行 43 个时钟周期。而用本节中介绍的程序，执行速度要快得多，而且能检测出产生不正确舍入的条件（如被除数小于 0），从而调整被除数，使得即使使用移位指令（它的速度比 IDIV 快得多）也能产生正确的舍入。参看图 6.1a。改进图 6.1a，得图 6.1b。

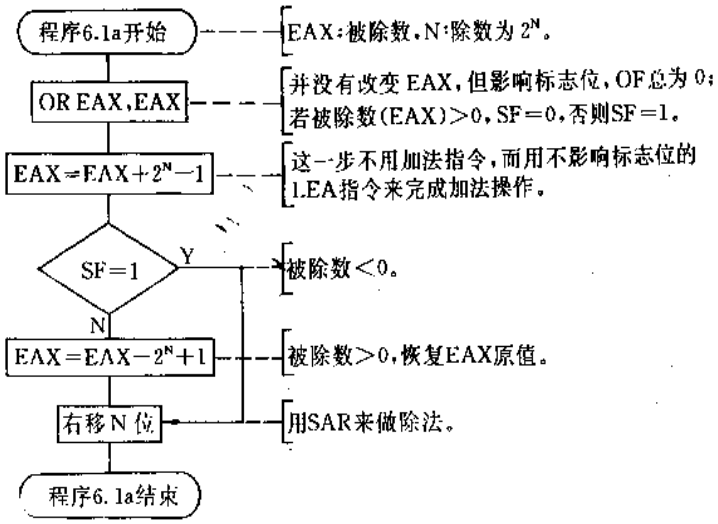


图 6.1a　用 SAR 做除法（除数为 2^N，N 为正整数）

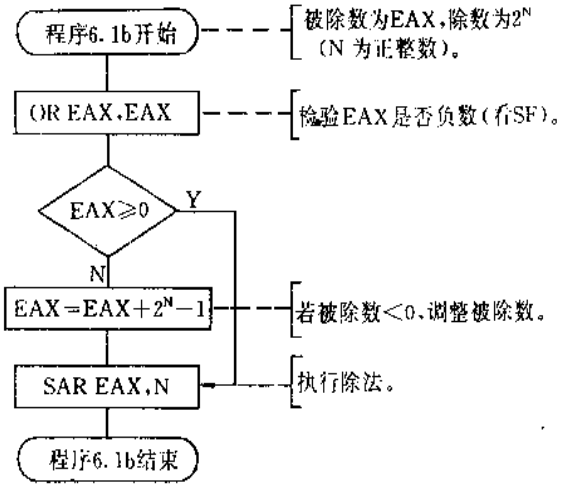


图 6.1b　用 SAR 做除法（改进）

程序清单如下：

程序 6.1：符号除法

```
;
;程序 6.1a
;
OR EAX, EAX
LEA EAX, [EAX+2^N-1]
JL Adj
SUB EAX,2^N-1
Adj:
```

```
SAR EAX,N
;
;程序 6.1b
;
OR EAX,EAX
```

```
JGE NoAdj
ADD EAX, 2^N-1
NoAdj:
    SAR EAX,N
```

这里大家可能注意到了，我们没有将程序 6.1 写成一个子程序。因为执行子程序调用和返回需要的时间；几乎要抵消快速的 2 的幂的 SAR 除法所节省下来的全部时间。所以为了获得所期望的性能提高，SAR 程序必须作为一个在线的代码来执行。如果你用了汇编的等号(EQU)或者宏，程序的格式就更简明了。

6.1.2.2 排序

下面举一个排序的例子。

所谓"排序"，就是将一个无序的数列变成一个有序——从小到大或从大到小——的数列的操作。

排序的方法很多。其中，"冒泡排序法"便是有名的一种。用一句形象的话来描述这种"冒泡法"，那就是"让最小(或最大)的数象气泡一样从水下冒出水面"。第一个冒出水面的"泡"是原数列中最小(或最大)的数；未冒出水面的数列仍然是无序的，但个数少了一个。第二个冒出水面的"泡"，是剩下的无序数列中最小(或最大)的数；留在水下的无序数列的数又少了一个。等 n 个泡从水中冒出，这 n 个数按"冒"出水面的时间先后组成了有 n 个元素的有序数列；而留在水中的无序数列则少了 n 个。直到气泡冒完，无序数列的全部元素重新组合成为一有序数列。

如何使无序数列中最小(或最大)的数"冒"出水面是一个关键。我们以最大数脱颖而出的过程为例，图解于图 6.2。

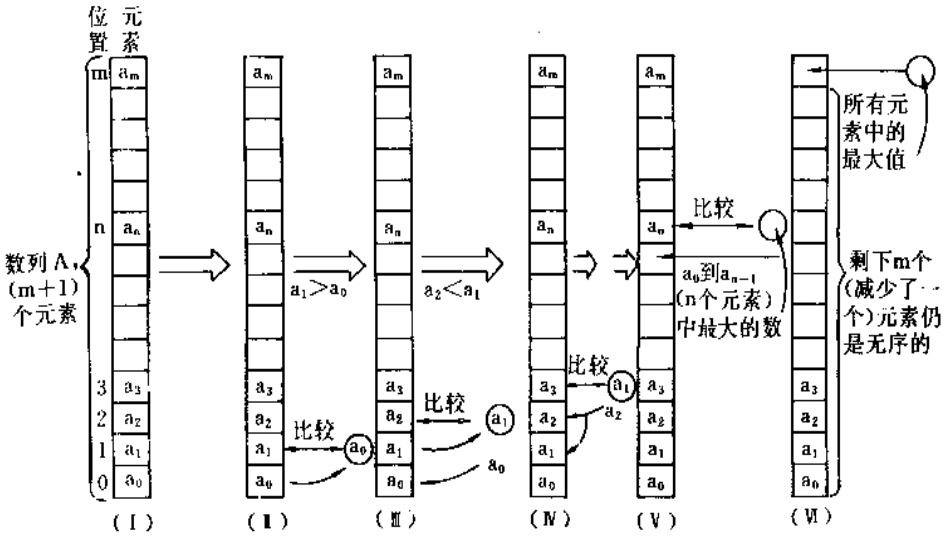


图 6.2 最大值"冒出水面"的过程

(Ⅰ) 具有(m+1)个元素的无序数列 A。

(Ⅱ) a_0 化为气泡，并从 0 位冒至 1 位。气泡的值(a_0)与位于 1 位的 a_1 进行比较。

(Ⅲ) 气泡(这时的值为 a_0)遇到了数值比它大的元素(a_1)。a_0 让位于 a_1，并被下压至与 a_1 相邻的空位。a_1 占据气泡。气泡(a_1)再与 a_2 进行比较。

(Ⅳ) a_2 的值比不过气泡，被下压至与其相邻的空位。气泡的内容不变，并继续上冒，与 a_3

进行比较。

(Ⅴ)气泡冒至第n位。因为不断地"优胜劣汰",唯最大之数方能存于气泡之中。所以气泡携带的,必为a_0到a_{n-1}(n个元素)中最大的数。气泡再与a_n进行比较。

(Ⅵ)最后,气泡携带数列A各元素中的最大值冒至顶部。m位空出,最大值占据m位。

下面来研究实现冒泡排序法的程序SORT。

有一无序数列A,它的第一个元素a_0在内存中的地址,存放在寄存器ESI之中。它的元素个数存放在寄存器ECX之中。

设计将EAX寄存器当作气泡,EDX寄存器存放气泡到达的层号(位置),(对照图6.1)。程序流程如图6.3:

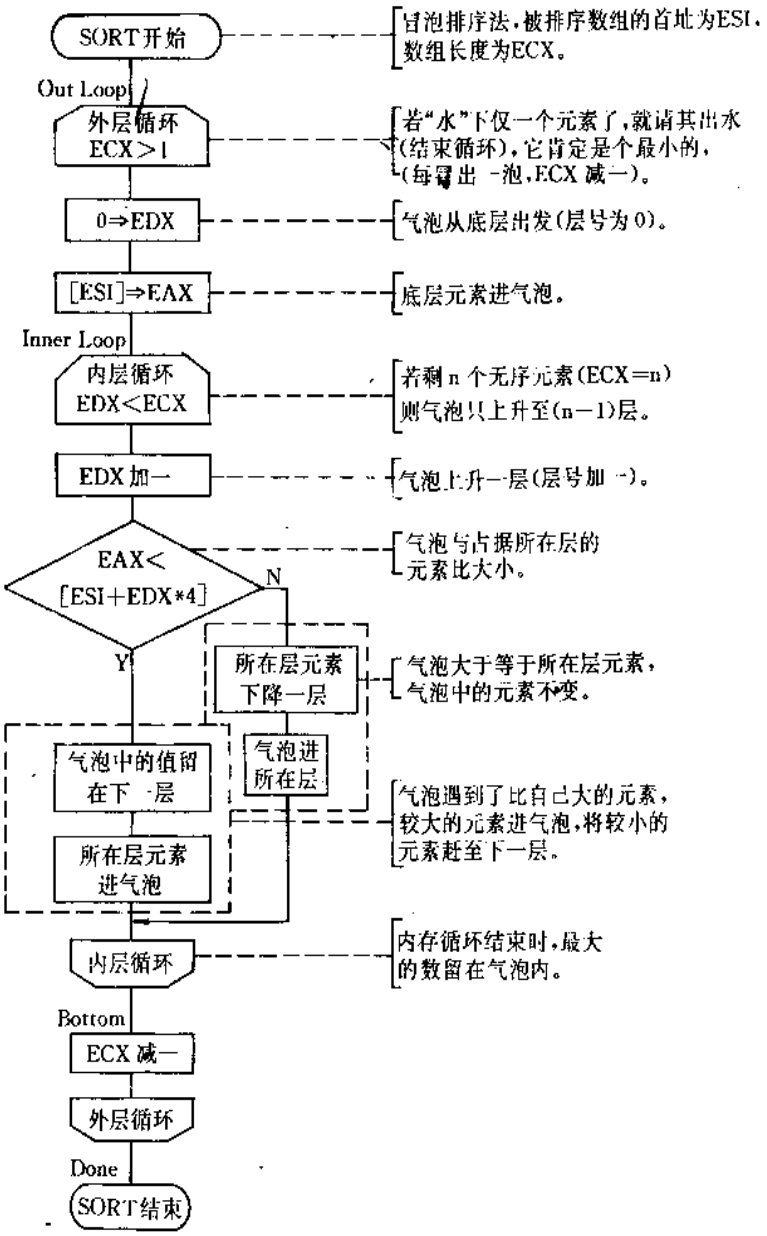


图6.3 SORT程序流程图

根据上述流程编制了程序 6.2。

程序 6.2：冒泡排序法

```
SORT PROC
OutLoop:
    CMP ECX,1
    JLE Done
    MOV EDX,0
    MOV EAX, [ESI]
Inner Loop:
    CMP EDX, ECX
    JGE Bottom
    INC EDX
    CMP [ESI+EDX * 4],EAX
    JG Swap
    MOV [ESI+EDX * 4-4], [ESI+EDX * 4]
    MOV [ESI+EDX * 4], EAX
    JMP InnerLoop
Swap:
    MOV [ESI+EDX * 4-4], EAX
    MOV EAX, [ESI+EDX * 4]
    JMP InnerLoop
Bottom:
    DEC ECX
    JMP OutLoop
Done:
    RET
SORT ENDP
```

6.1.2.3 阶乘

在数学上，阶乘是个很重要的函数，其公式为

$$n! = n * (n-1) * (n-2) * \cdots\cdots * 1$$

阶乘的程序设计，可以使用迭代法，也可以使用递归调用。一般来说，迭代法的速度比递归调用要快得多。一些先进的高级语言编译器对递归过程进行分析，并决定它们是否可以用迭代法来代替。

先介绍迭代法，然后再介绍递归过程调用。通过这一递归调用的例子，我们不但了解了递归调用的方法，而且还能了解到调用子过程时的“现场保护”的概念和方法。

1. 迭代法求阶乘

参数 n 存放在 EAX 寄存器内。计算结果(n!)也返回到 EAX 寄存器。为处理溢出，定义一变量 Error。

程序流程图见图 6.4。源代码见程序 6.3a。

程序 6.3a：迭代法求阶乘

```
Error EQU SS: byte ptr [EBP-1]
FACT PROC
    ENTER 4,0
    MOV Error, 0
    MOV ECX,EAX
    CMP ECX,0
    JB Fail
    MOV EAX, 1
    JE Done
top:
    IMUL EAX, ECX
    SETO BL
    OR Error, BL
    LOOP top
Done:
    CMP Error, 1
    JZ Fail
    LEAVE
    RET
Fail:
    MOV EAX, -1
    LEAVE
    RET
FACT ENDP
```

(2) 递归过程调用法求阶乘

程序 Fact 对所给的参数 n 进行检验，并调用过程_Fact。实现阶乘计算的过程递归调用包

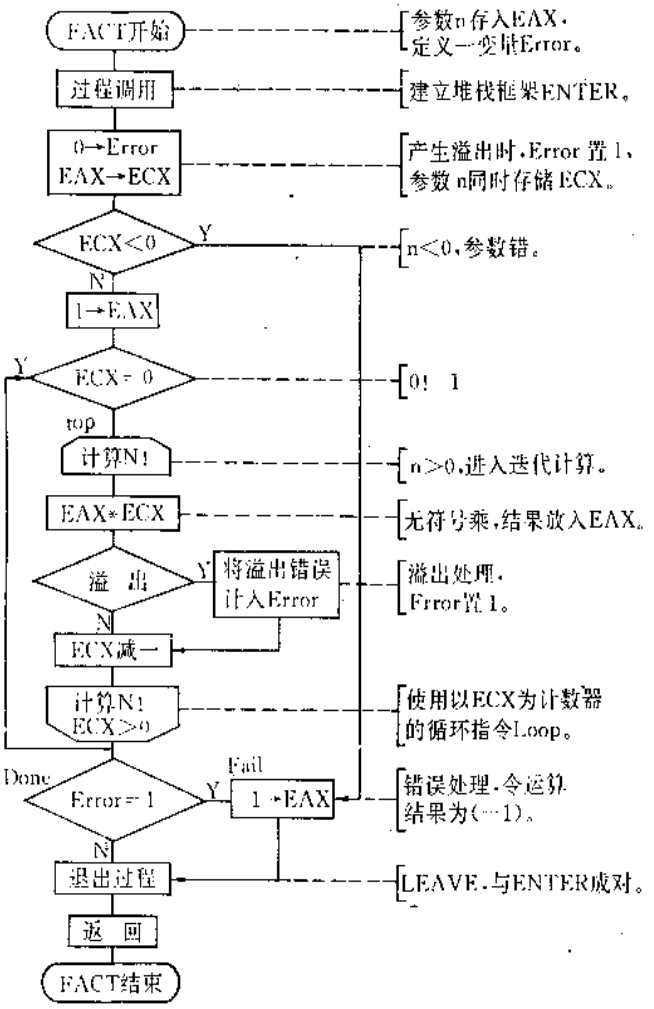


图 6.4 迭代法求阶乘(n!)的程序流程

含在过程 Fact 之中。若传递给 Fact 的参数 n 不为 0，_Fact 便递归调用参数为(n－1)的_Fact。待被调用的 Fact 返回，调用过程的 Fact 将返回结果与参数 n 相乘，便得出了(n!)的值。0! ＝1。

被调用的过程_Fact，叫做 Fact 的哑过程。在递归编程中，使用哑过程的做法是常见的。

参数 n 的取值范围：0≤n<14。在 32 位符号整数范围内，可以表示的最大阶乘为(13!)。

在调用任何子过程时，调用过程的寄存器状态是要保护的，以便在子过程返回后，调用过程利用这些状态继续执行运算。保存这些状态的方法有两种：

- 由调用程序将有用的寄存器内容保存起来；
- 由被调用过程将它使用的寄存器的原内容进行保存。

程序流程图见图 6.5 及图 6.6。源代码见程序 6.3b。

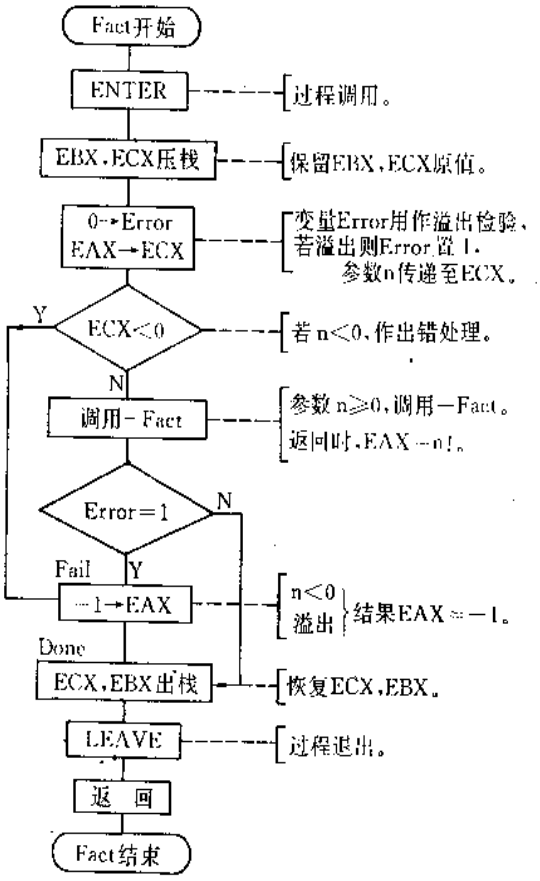


图 6.5 递归调用法求阶乘的主程序 Fact 程序流程图

程序 6.3b： 递归法求阶乘

```
主程序
Error EQU SS: byte ptr [EBP-1]
FACT PROC
  ENTER 4,0
  PUSH EBX
  PUSH ECX
  MOV Error,0
  MOV ECX, EAX
  CMP ECX, 0
  JB Fail
  CALL  Fact
  CMP Error, 1
  JNZ Done
Fail:
  MOV EAX,1
Done:
  POP ECX
  POP EBX
  LEAVE
  RET
FACT ENDP
递过程
 Fact PROC
  CMP ECX,0
  JZ done
  PUSH ECX
  DEC ECX
  CALL _Fact
  POP ECX
  IMUL EAX,ECX
  SETO BL
  OR Error, BL
  RET
done:
  MOV EAX,1
  RET
_Fact ENDP
```

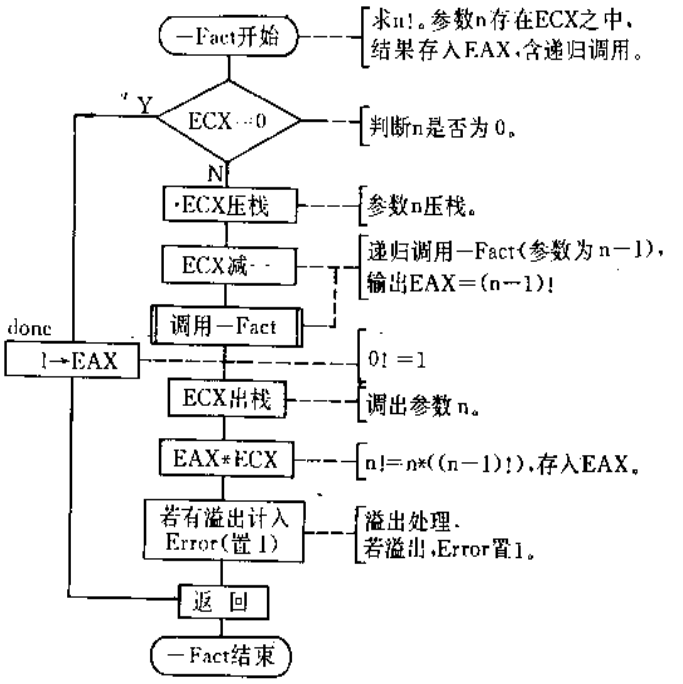


图 6.6 Fact 的哑过程 Fact 程序流程图

6.1.2.4 信号灯

在一个多用户、多任务系统中，两个或两个以上的进程同时申请使用一个盘、带、或文件的现象是常见的。协调进程间对于系统资源的竞争是多任务系统设计中的重要课题。当然这样大的题目不可能在这里被透彻地研究。但是，我们不难想到，交通指挥中的信号灯能使众多的车辆避免事故、安全驶达目的；那么，多个进程在同一资源上的“撞车”，是不是也可以用“信号灯”来解决呢？

我们可以规定内存中的某些字节作为系统资源的状态标志——信号灯。“信号灯”有两种状态：0 表示相应的资源已被某个进程所占(忙态)，1 表示相应的资源已被释放(空态)。当进程需要申请某个资源时，便去读对应于该资源的信号灯字节。当进程占用了状态为 1 的资源后，就立即将该标志置为“忙态”。当进程用完这个资源后，一定要修改状态，变成空态，以释放这个资源；否则别的进程便没有使用这个资源的机会了。

这里我们又想起一个问题：如果信号灯的“开关”不加以控制，任人随便拨动，则定使方寸大乱，后果不堪设想。于是必须规定：当信号灯一旦被读，在读/改/写全过程没有完成之前，系统中的其它进程不能去存取内存。这就叫做“信号灯标志的读/改/写不可分割。”

80386 的 LOCK 前缀可以在其后的指令执行期间将内存锁定，排斥其它进程对内存的访问，保证了后续指令执行的不可分割性。

本节介绍了两个程序：

GetSem：进程独占信号灯资源的程序，

FreeSem：进程释放信号灯资源的程序。

这两个程序可以用下面的方式调用：

```
MOV   ESI, Offset SemWord
      ;将 DS 段内的信号灯字节的首址送入 ESI 寄存器
MOV   ECI, SemBitWeNeed
      ;将信号灯内需要测试的一位送入 ECX 寄存器
CALL GetSem   ;等待信号标志变为空态,并占有它,再将它变为忙态
MOV   ESI,Offset SemWord
MOV   ECX,SemBitWeNeed
CALL  FreeSem   ;释放一个资源,使信号标志变为空态
```

GetSem 程序

程序流程见图 6.7。程序代码见程序 6.4。

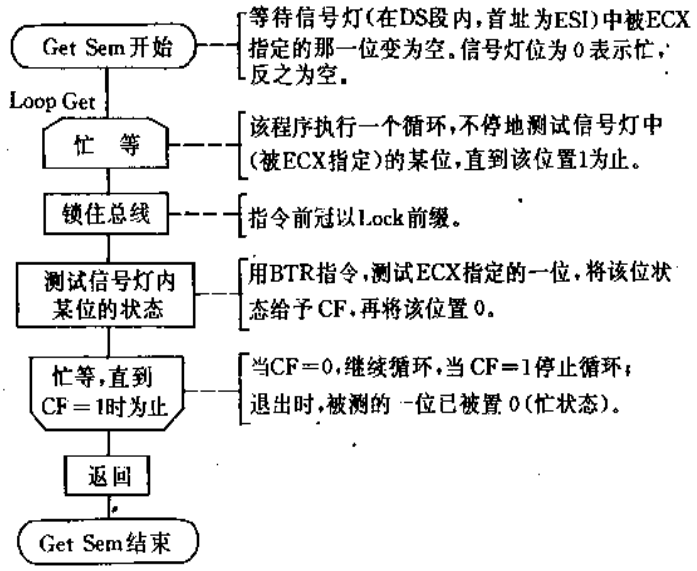


图 6.7 GetSem 程序流程

FreeSem 程序:

程序流程见图 6.8。程序代码见程序 6.4。

程序 6.4: 信号灯

```
GetSem PROC
LoopGet:
  LOCK BTR dword ptr [ESI],ECX
  JNC LoopGet
  RET
GetSem ENDP

FreeSem PROC
  LOCK BTS dword ptr [ESI],ECX
  JNC Done
  MOV EAX, -1
  Done:
    RET
FreeSem ENDP
```

这里再说明一下,以上确实不是一个好的程序。尤其是 GetSem,若遇设备非空,它就在那里一直等下去,直至设备空为止(这叫忙等,Busy Waiting)。如此处理,效率很低。望读者各显其能,编出比这更好的程序,放到你的系统里。

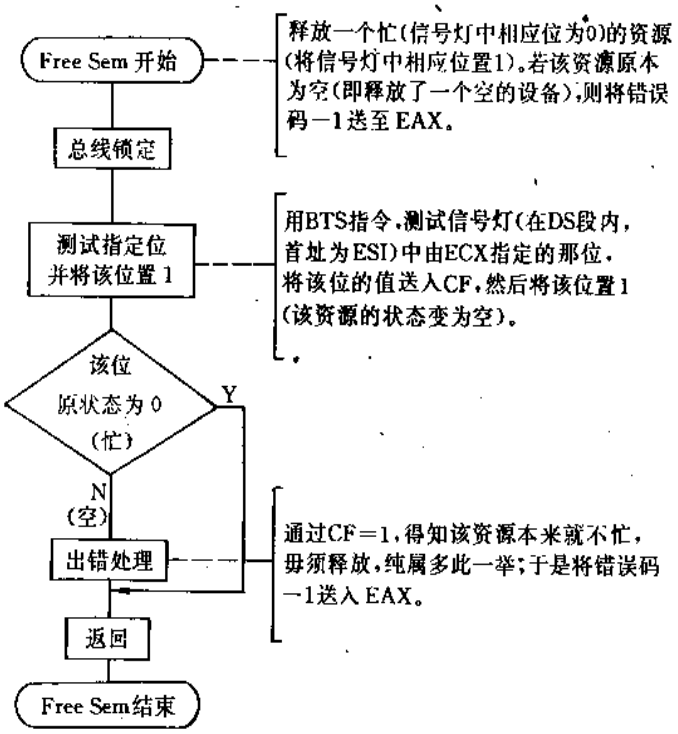


图 6.8 FreeSem 程序流程

6.1.2.5 字符串查找

在命令解释程序、文本编辑、以及数据库管理系统内,常常要处理在一个字符串中查找子串的问题。在基准测试(Benchmark)的标准集中有一个叫做 EDN Benchmarks 子集,就是由"子串查找"这一通用功能演变而来。大多数计算机制造商都根据这个标准子集来对照他们的处理器,以对付他们的竞争对手。下面就介绍一下为 80386 编的 EDN 字串查找基准(EDN string-search benchmark)。

1. BENCHE 程序的结构

BENCHE 程序即 EDN String-search benchmark。它的程序结构如图 6.9 所示。

输入: ESI 要查找的子串的指针(在 DS 段内)

EDI 被检索的字符串的指针(在 ES 段内)

ECX 字符串的长度

EDX 子串的长度

输出: 如发现字符串中确含要查找的子串,将子串在该字符串中的开始位置送入 EAX;若字符串中不含要查找的子串,则-1 送至 EAX。

变量: Srch_Len EQU SS: dword ptr [EBP-4]。

说明:

(1) 扫描之前的工作: 建立堆栈框架(ENTER); 方向标志清 0; 子串的第一个字节送 AL; EBX 指向子串的第二个字节; EDX 保存子串的剩余长度;ECX 保存为寻找第一个匹配字符,需要在字符串中扫描的最大长度;Srch_Len 保存此时的 ECX 值。

(2) 先在字符串中查找子串的第一个字符(Ⓢ模块);扫描字符串;如未找到,转Ⓝ模块;如

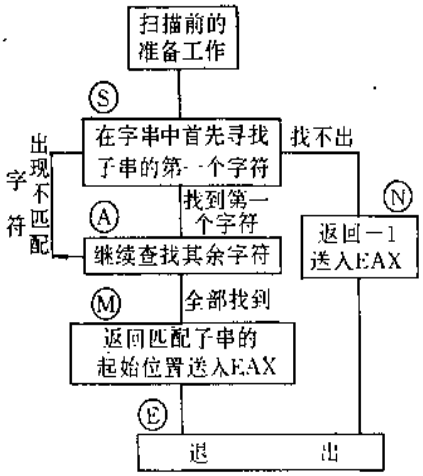


图 6.9　BENCHE 程序结构

找出第一个字符，转Ⓐ模块。

(3) 继续查找其余字符(Ⓐ模块)；若子串的所有字符都找到，转Ⓜ模块；若找到子串全部字符前，发现不匹配的字符，转Ⓐ重新查找。

(4) 返回匹配子串的起始位置，送入 EAX，退出(Ⓜ模块)。

(5) 查找失败，(－1)送 EAX，退出(Ⓝ模块)。

2. BENCHE 程序模块框图

(1) 扫描前的准备工作。见图 6.10。

(2) Ⓢ模块。见图 6.11。

(3) Ⓐ模块。见图 6.12。

(4) 计算返回值和退出。见图 6.13。

3. BENCHE 程序清单

程序 6.5：字符串查找

```
Srch_Len EQU SS: dword ptr [EBP-4]
BENCHE  PROC
  PUSH   EBP
  MOV    EBP,ESP
  SUB  ESP,4
  CLD
  LODSB
  MOV    EBX,ESI
  DEC    EDX,
  SUB    ECX,EDX
  MOV    Srch_Len,ECX
Scan_Loop:
  JECXZ  SHORT NotFnd
  REPNZ  SCASB
  JNE    SHORT NotFnd
  CMP    EDX,0
  JE     SHORT Match
  PUSH   ECX
  PUSH   EDI
```

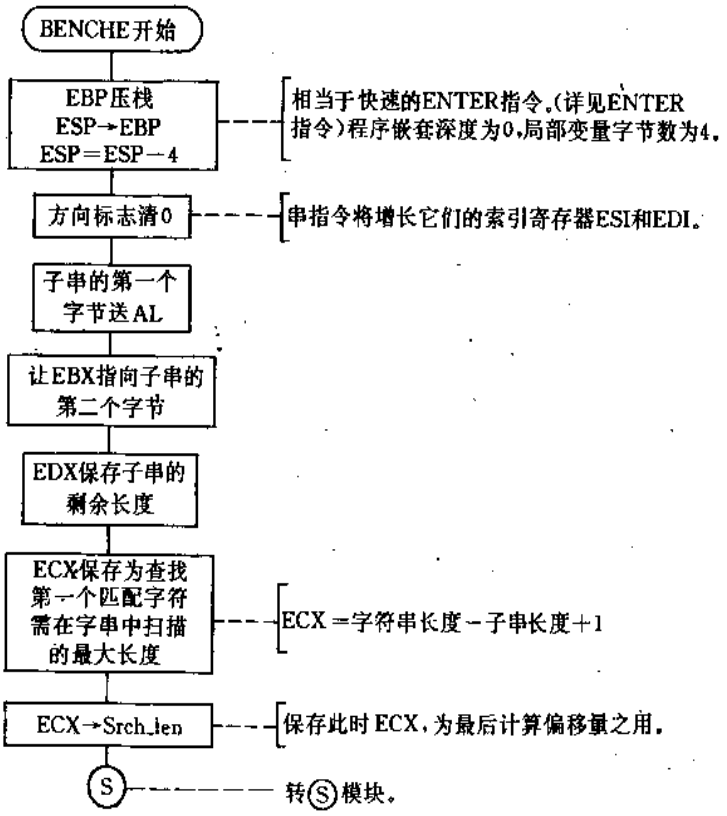


图 6.10 BENCHE 程序开始部分流程图

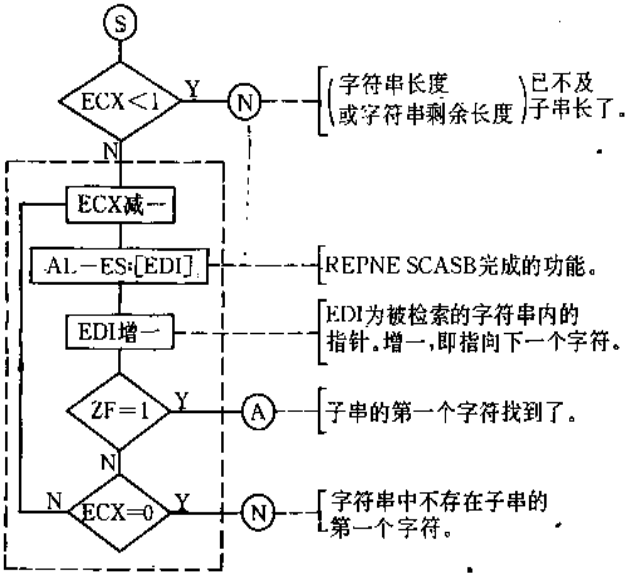


图 6.11 Ⓢ模块流程图

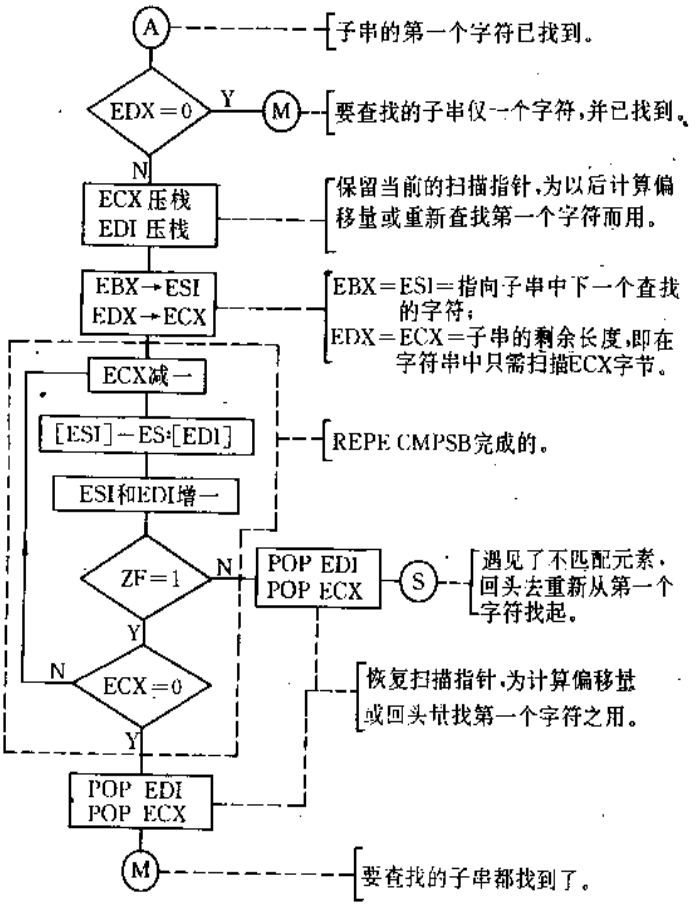


图 6.12 Ⓐ模块流程图

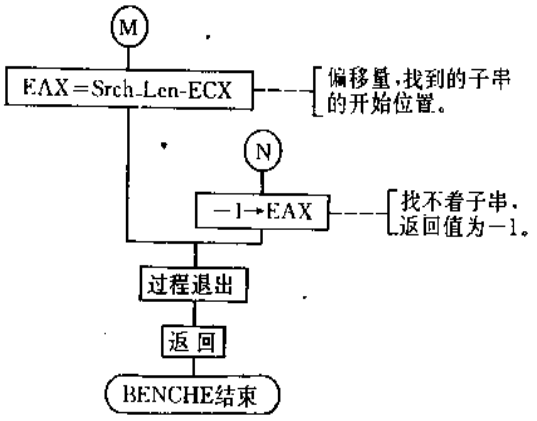


图 6.13 计算返回值和退出部分的流程图

```
MOV     ESI,EBX
MOV     ECX,EDX
REPE    CMPSB
POP     EDI
POP     ECX
```

```
  JNE     Scan_Loop
Match:
  MOV     EAX, Srch_Len
  SUB     EAX,ECX
  JMP     Epilogne
NotFnd:
  MOV     EAX,-1
Epilogne:
  LEAVE
  RET
BENCHE  ENDP
```

6.1.2.6 位块传送

图形运算和其它运算，常常要处理象位串那样的简单数据。最常见的一种位数据操作便是将位串或位块在内存中传送。

本节介绍两种位块传送程序：简单的和通用的。所谓简单的位块传送程序就是只考虑最最简单的情况。即只考虑目标位串对齐双字边界和被传送的位串长度为 32 的整数倍。这种最简单的位块传送虽无多大实用价值，但它能帮助我们理解位块传送的核心。在程序中我们使用了 80386 新增的 SHRD 和 SHLD 指令(双宽度移位指令)，使每次迭代可处理 32 位；否则每次迭代只能处理 16 位。

通用的位块传送没有上面的限制，所以它比简单的位块传送有用，但也复杂得多。

和常见的情形一样，这些程序都使用了局部的堆栈存储器。这样便于递归和管理局部的临时变量。EBP 常常用作指向这个堆栈的框架指针，而 ESP 则作为动态的栈顶指针。于是，局部临时变量常常通过 EBP 寄存器来寻址，比如：

BitCount EQU SS: dword ptr [EBP-4]

配对的 ENTER 和 LEAVE 指令便直接使用 EBP 作为栈框架指针。

一、程序：简单的位块传送 BitBlt

(1) 简化条件：源串可在任意位置；目标串对齐双字边界；传送的位数是 32 位的整数倍。

(2) 参数：

ESI：源串的双字地址。

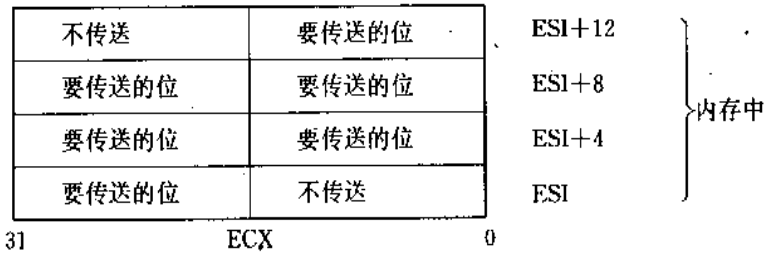
EDI：目标串的双字地址。

EBX：要传送的位串长度(以双字为单位)。

ECX：要传送的位串，在源串的第一个双字(地址为 ESI)中的偏移量。

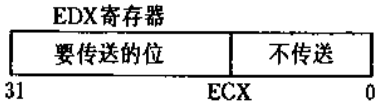
(3) 过程图解：

① 源串



EBX=3 (实际传送 3×32 位)

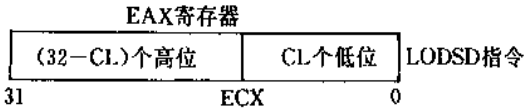
② 源串的第一个双字装入 EDX。ESI 增 4，以指向源串的第二个双字。



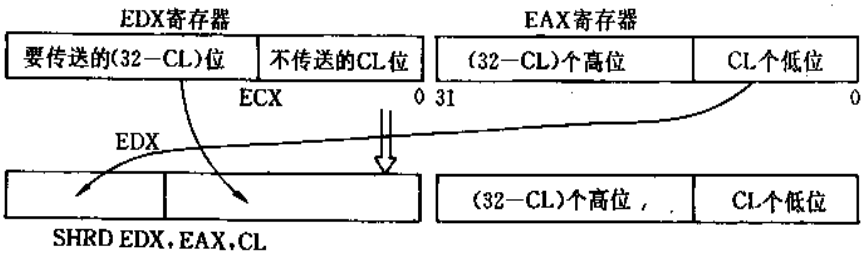
MOV EDX, dword ptr [ESI]

ADD EST, 4

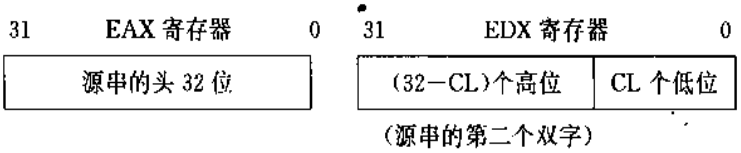
③ 源串中第二个双字装入 EAX。ESI 增 4,以指向源串的第三个双字。



④ EDX:EAX 双宽度右移 CL 位,使源串的头 32 位充满 EDX。EAX 不变。



⑤ EDX 与 EAX 交换。EAX 准备传送给目标串,EDX 已是源串第二个双字的内容了。



XCHG EDX, EAX

⑥ EAX 寄存器存入地址为 EDI 的目标串,完成了一次迭代,有 32 位数据实现了传送。(STOSD 指令)

⑦ 要传送的双字数目减一(DEC EBX)。判断一下还有无双字要传送。若有,重复第 2 步至第 7 步。

(4) 程序清单:

程序 6.6a: BitBlt

```
BitBlt  PROC
  MOV EDX, dword ptr [ESI]
  ADD ESI,4
  CLD
BitLoop:
  LODSD
  SHRD EDX, EAX,CL
  XCHG EDX,EAX
  STOSD
  DEC EBX
  JA BitLoop
  RET
BitBlt
```

二、程序:通用的位块传送程序 BitBlt

1. 综述

通用的位块传送程序 6b(以下简称"通用程序"),取消了简单的位块传送程序 6a(以下简

称“简单程序”)对于源位串和目标位串的种种限制。对于通用程序来说,被传送的位串长度可以是任意的,目标位串也不必对齐双字边界,如图 6.14 所示。

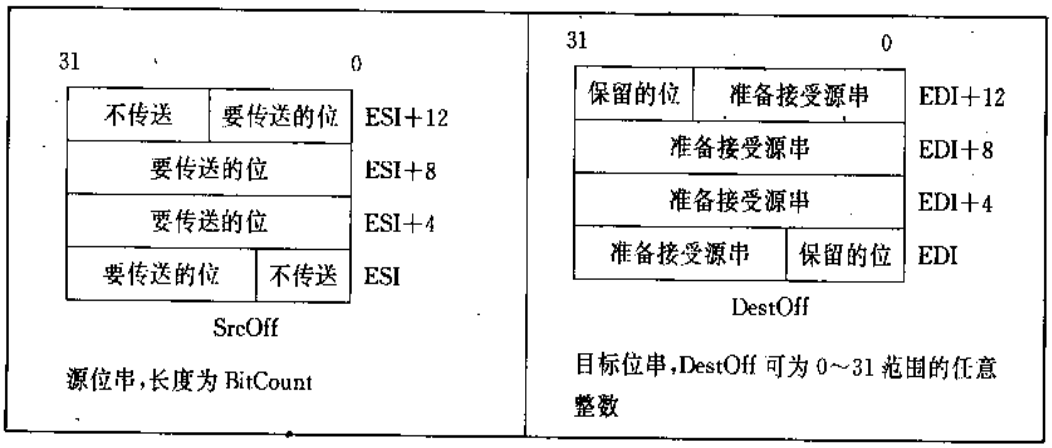


图 6.14 通用程序处理的源位串和目标位串
(SrcOff 与 DestOff 毋须相等)

通用程序要考虑各种可能的情况,所以程序复杂起来了。但需要过多关照的不外乎是第一个双字和最后一个双字。因为中间的双字,它们是对齐了双字边界的,而且可以传送的位串也正好是 32 位的整数倍,所以它们就和简单程序一致了。

理解到这一层,我们自然会明白通用程序为什么要分成三个部分:

第一部分(prelude)处理目标串的第一个双字;

第二部分(main)类似于简单程序的核心部分;

第三部分(postlude)处理目标串的最后一个双字。

在程序开始前,传递给程序的参数往往用寄存器来保存,它们是:

EAX: 要传送的源串的位数(在程序开始后,该参数将由变量 BitCount 来保存)。

ESI: 源串的双字地址

EDI: 目标串的双字地址

ECX: 待传送的源串,在其第一个双字(地址为 ESI)中的偏移量(在程序开始后,该参数将由变量 SrcOff 来保存)。

EDX: 源串传送的目的地,在目标串第一个双字(地址为 EDI)中的偏程量(在程序开始后,该参数将由变量 DestOff 来保存)。

在程序中使用到的局部变量有:

```
BitCount   EQU SS: dword ptr [EBP-4]
SrcOff     EQU SS: dword ptr [EBP-8]
DestOff    EQU SS: dword ptr [EBP-12]
SrcOffFlg  EQU SS: dword ptr [EBP-16]
```

上述寄存器和变量,除 SrcOffFlg 以外,都可以在图 6.14 中找到其说明并明了其含义。SrcOffFlg用作表征 SrcOff 大小的标志,当 SrcOff>DestOff 时,SrcOffFlg 置 1,否则置 0(该标志初始化为 0)。

2. 程序的第一部分: prelude

(1) 在处理目标串的第一个双字时,会碰到下面几种情况:

① 源串很短(DestOff+BitCount)≤32,仅传送一个双字就把源串全部传送完毕。

② SrcOff>DestOff。源串第一个双字中的要传送的位串,不能充满目标串第一个双字的空位,另要将源串第二个双字中的低位传到目标串第一个双字的高位。

③ SrcOff<DestOff。源串第一个双字的高位不能传送到目标串的第一个双字中,只能传送到目标串第二个双字的低位。

④ SrcOff=DestOff。当然这种情况处理起来要简单些。也可以归入(2)或(3),作为它们的一个特例来对待。

(2) 设计:见图 6.15

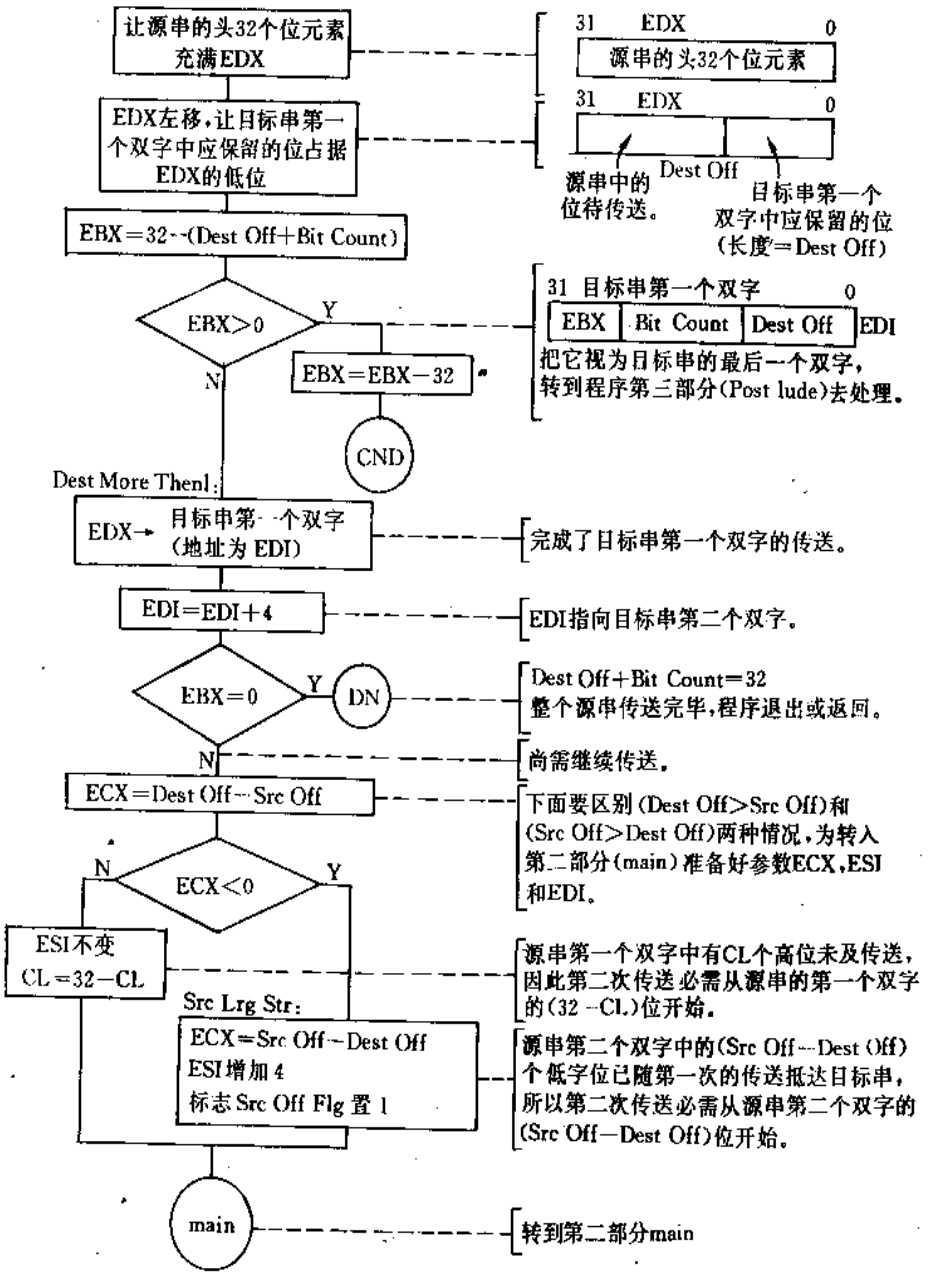


图 6.15 程序的第一部分:prelude

3. 程序的第二部分 main

(1) prelude 为 main 准备好了新的参数：

EDI——指向目标串的指针，

ESI——指向源串的指针，

ECX——尚未送出的源串在 ESI 指向的双字中的偏移量。

而且目标串是和双字边界对齐了的。所以这一部分几乎和简单程序 6a 完全一样，用不着多作解释。通用程序的初始参数仍保留在 BitCount、SrcOff 和 DestOff 之中。

(2) 设计：见图 6.16

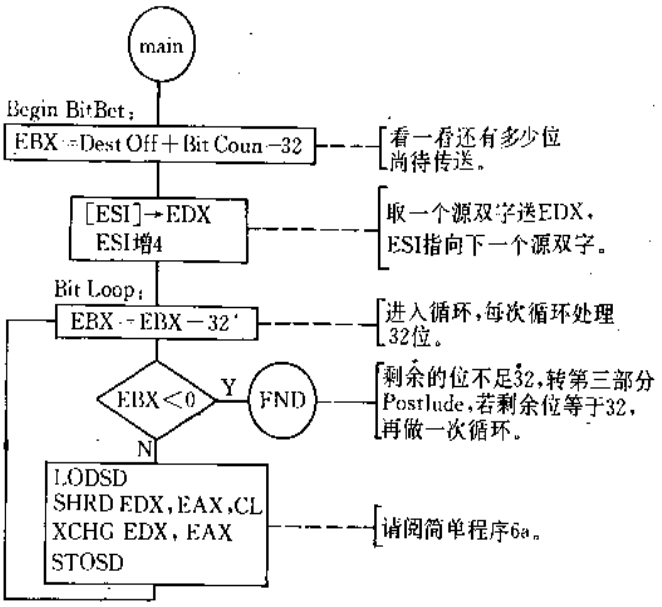


图 6.16 程序的第二部分：main

4. 程序第三部分：postlude

(1) postlude 是用来处理目标位串的最后的不足 32 个的位串(包括 0 个)。它要解决的关键问题是：既完成 BitCount 个位块的准确传送，又使得目标串最后一个双字中该保留的位得以保留。

(2) 设计：见图 6.17

5. 编码及源程序清单

程序 6.6b：通用的位块传送程序

```
BitCount    EQU SS: dword ptr [EBP-4]
SrcOff      EQU SS: dword ptr [EBP-8]
DestOff     EQU SS: dword ptr [EBP-12]
SrcOffFlg   EQU SS: dword ptr [EBP-16]

BitBlt  PROC
  ENTER  16,0
  CLD
;* * * *
;* * PRELUDE:
```

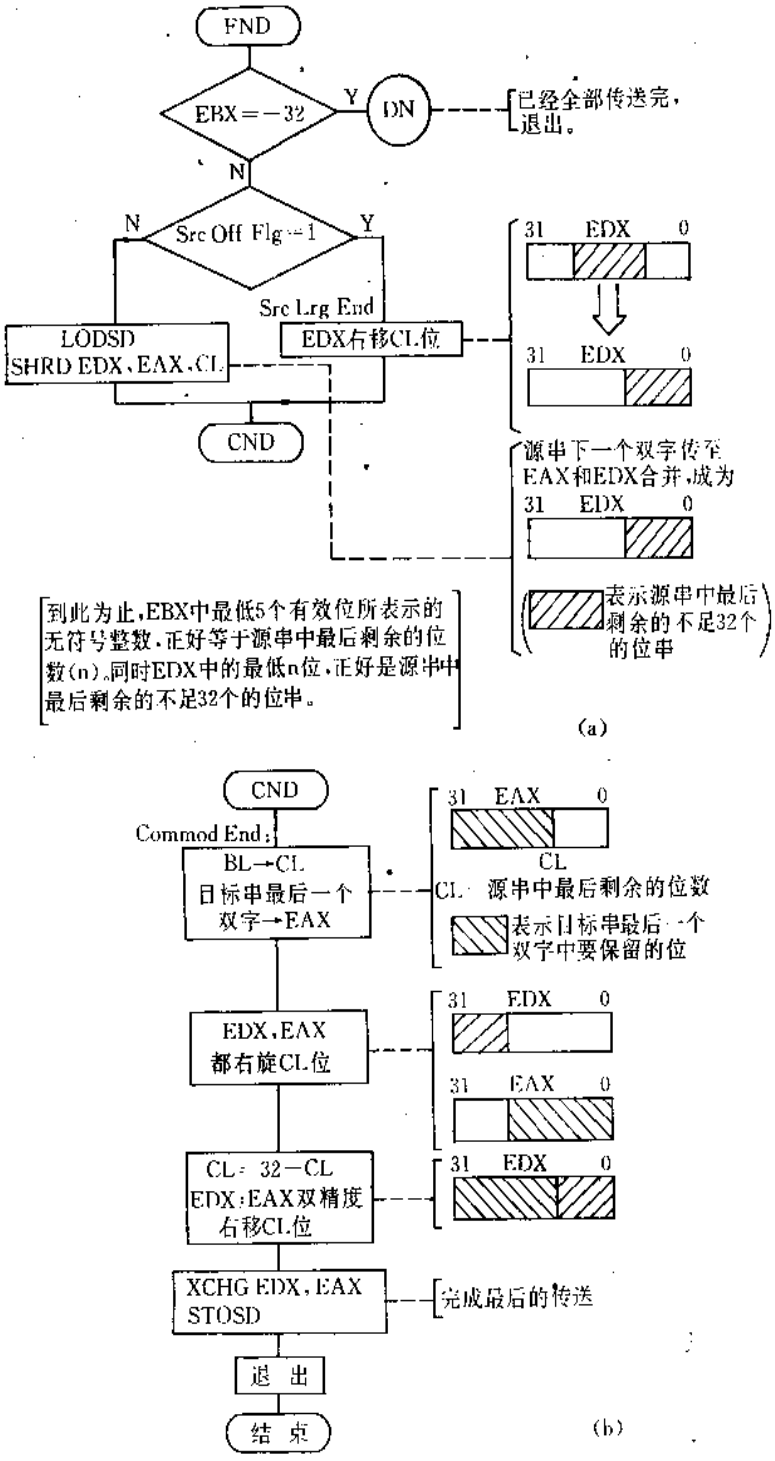


图 6.17 程序的第二部分：postlude

```
;* * * *
  MOV     BitCount, EAX
  MOV     SrcOff, ECX
  MOV     DestOff, EDX
  MOV     EDX, dword ptr [ESI]
  MOV     EAX, dword ptr [ESI+4]
  SHRD    EDX, EAX,CL            ;源串头 32 位充满 EDX
```

```
  MOV     EAX, dword ptr [EDI]
  MOV     ECX, DestOff
  ROR     EAX, CL
  SHLD    EDX, EAX, CL      ;让目标串第一个双字中应保留位占据 EDX 低位
  MOV     EBX,32
  SUB     EBX,DestOff
  SUB     EBX,BitCount
  JBE     DestMore Then 1   ;(DestOff+BitCount)≥32
  MOV     ECX,32
  SUB     ECX,EBX
  MOV     EBX,ECX
  JMP     Common End        ;(DestOff+BitCount)<32,转 postlude 处理
DestMore Then 1:
  MOV     dword ptr [EDI],EDX
  ADD     EDI,4
  JZ      Done              ;(DestOff+BitCount)=32
  SUB     ECX, SrcOff       ;若 SrcOff>DestOff, CL- SrcOff-DestOff
                            否则 CL=32-(DestOff-SrcOff)
  JL      SrcLrgStr
  MOV     BL,32
  SUB     BL,CL
  MOV     CL,BL
  JMP     Begin BitBlt

SrcLrgStr:
  MOV     ECX, SrcOff
  SUB     ECX,DestOff
  ADD     ESI, 4
  MOV     SrcOffFLg,1
;* * * *
;* *  MAIN:
;* * * *
Begin BitBlt:
  MOV     EBX,BitCount
  ADD     EBX,DestOff
  SUB     EBX,32
  MOV     EDX,dword ptr [ESI]
  ADD     ESI,4
BitLoop:
  SUB     EBX,32
  JL      FixEnd
  LODSD
  SHRD    EDX,EAX,CL
  XCHG    EDX,EAX
  STOSD
  JMP     BitLoop
;* * * *
;* *  POSTLUDE:
;* * * *
FixEnd:
  JZ      Done
  CMP     SrcOffFlg,1
  JZ      SrcLrgEnd
  LODSD
  SHRD    EDX, EAX,CL
```

```
    JMP     CommonEnd
SrcLrgEnd:
    SHR     EDX,CL
CommonEnd:
    MOV     EAX,dword ptr [EDI]
    MOV     CL,BL
    ROR     EDX,CL
    ROR     EAX,AL
    MOV     BL,32
    SUB     BL,CL
    MOV     CL,BL
    SHRD    EDX,EAX,CL
    XCHG    EAX,EDX
    STOSD

Done:
    LEAVE
    RET
BitBlt  ENDP
```

6.1.3 浮点数的例子

6.1.3.1 利用浮点的条件码实现程序的控制转移

1. 浮点的条件码

关于浮点的条件码，以及能改变这些条件码的指令，我们在第2章及第3章中都介绍过。例如

FCOM　OP_1

指令便是根据操作数 OP_1 和栈顶比较的结果来修改条件码的：

	C3	C2	C0
$ST>OP_1$	0	0	0
$ST<OP_1$	0	0	1
$ST=OP_1$	1	0	0
无序	1	1	1

利用浮点条件码来实现程序控制转移，是浮点很重要的应用。下面将这个技术介绍给读者。

2. 将80387的条件码变成80386的标志位

在80386中，实现控制转移的指令很多，但它们都和80387没有关系，即使是条件转移(如Jcc等)也是根据80386自身的标志位。因此要想利用80387的条件码来控制转移，首先要有一座“桥梁”，一座将80387的条件位过渡到80386的标志位的桥梁。

这座桥梁是这样来实现的：

FSTSW　AX

FAHF

FSTSW　AX 将80387的状态字复制到80386的AX寄存器，C3、C2和C0所在的状态字的高字节被复制到了AH。FAHF再将AH寄存器内容存入标志位寄存器。通过这座桥梁，

80387 的条件码进驻了 80386 的标志位寄存器。而且在 C3、C2、C0(80387 的)和 ZF、PF、CF(80386 的)之间存在固定的对应关系，如图 6.18 所示：

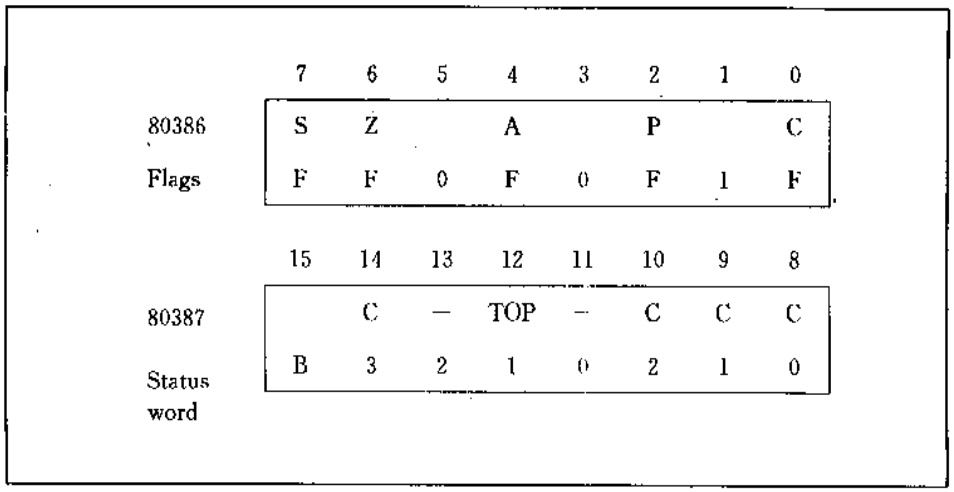


图 6.18　80387 的条件码转移至 80386 的标志位

3. 实现控制转移

程序 7 是一个根据 FCOM 指令执行后 80387 条件码的设置情况而跳转的表。栈顶下的第一个累加器与栈顶作比较，然后根据比较的结果作相应的跳转。其中的关键，就是上面介绍的那个“桥”。

举一反三，对于 FTST，FICOM，FUCOM 等测试和比较指令，也可以写出和程序 6.7 类似的跳转表。

程序 6.7：浮点标志

```
;通过比较的跳转表
FCOM   ST(1)    ;执行比较
FSTSW AX        ;将状态字拷贝到 80386 的 AX 寄存器
SAHF            ;将条件码送入标志寄存器
JP Error        ;ST 或 ST(1)是个 NaN——出错
JE STeqST1      ;ST=ST(1)
JB STltST1      ;ST<ST(1)
JA STgtST1      ;ST>ST(1)
```

6.1.3.2　计算余数

在第 3 章我们介绍部分余数指令时曾经说过这类指令的执行时间特别长，所以它们很可能停在求余数的半道上，只得部分余数(条件码 C2 反映了这个状态)。下面我们设计了一个中断服务程序(程序 8)，来检查这个减法是否做完以及是否需要迭代。在这个例子中，用到程序 6.7 将条件码转移至标志位的序列。

程序 6.8：部分余数

```
;利用 C2 条件码完成部分余数
RemLoop:
  FPREM          ;计算结果影响条件码 C2
  FSTSW AX
  SAHF           ;C2 送至 80386 的 PF 标志位
  JP RemLoop     ;若 C2=1，表明没有求到其正的余数，继续迭代
```

6.1.3.3　计算指数函数

80387 没有直接计算 10^x、l^x 和 Y^x 等指数的指令。但有计算($ST=2^{ST}-1$)的 F2×M1 指

令，以及计算($ST=ST(1)*\log_2 ST$)的 FYL2X 指令。利用指数函数及对数函数的知识导出：

$10^x=2^{x*\log_2 10}$

$l^x=2^{x*\log_2 l}$

$Y^x=2^{x*\log_2 Y}$

于是，利用程序计算 10^x、l^x、Y^x 的问题便迎刃而解了。请看程序 6.9。

程序 6.9：计算指数函数

```
;计算指数函数
;10^x     ;ST=x
FLDL2T                  ;ST=log2 10,ST(1)=x
FMULP ST,ST(1)          ;ST=x * log2 10
F2XM1                   ;ST=2^(x*log2 10)-1
FLD1                    ;ST=1,ST(1)=2^(x*log2 10)-1
FADDP ST,ST(1)          ;ST=2^(x*log2 10)=10^x
;l^x   ;ST=x
FLDL2E                  ;ST=log2 E, ST(1)=x
FMULP ST,ST(1)          ;ST=x * log2 l
F2XM1                   ;ST=2^(x*log2 l)-1=l^x-1
FLD1                    ;ST=1,ST(1)=2^(x*log2 l)-1=l^x-1
FADDP ST,ST(1)          ;ST=l^x
;Y * * X   ;ST=Y, ST(1)=X
FYL2X     ;ST=X * log2 Y
F2XM1     ;ST=Y^x-1
FLD1      ;ST=1, ST(1)=Y^x-1
FADDP ST,ST(1)          ;ST=Y^x
```

6.1.3.4 矩阵乘法

1. 概述

(1) 矩阵乘法的定义

假设 X 为 $i\times k$ 矩阵，Y 为 $k\times j$ 矩阵(注意 X 的列数与 Y 的行数相等)。则它们的乘积 $Z=XY$ 为 $i\times j$ 矩阵，Z 矩阵元素

$$Z_{mn}=\sum_{r=1}^{k}(X_{mr}*Y_{rn})\quad \begin{matrix}1\leqslant m\leqslant i\\ 1\leqslant n\leqslant j。\end{matrix}$$

(2) 以行为主格式(Row-Major Format)

形如二维数组的矩阵在内存中的映象，也和二维数组一样，采用以行为主的格式(详见第 1 章)。即：

- 同一行中的元素在内存中依次连续存放；
- 以行为单位依次连续存放，放完第一行，再放第二行，……，直至最后一行。

其结果，导致同一列的元素存放在跨度为矩阵列数的分立单元之中。

2. 矩阵乘法($Z=XY$)的程序设计

(1) 初始条件及寄存器变量的初始化：

① 两个相乘的矩阵及产生的乘积矩阵都在 DS 段内。它们在段内映象的偏移量分别为

ESI：X 矩阵在 DS 段内的偏移量

EDI：Y 矩阵在 DS 段内的偏移量

EDX(及变量 Z_{off})：Z 矩阵在 DS 段内的偏移量

② 矩阵的行数和列数

EAX（及变量 Di）：X 和 Z 的行数(i)

ECX（及变量 Dk）：X 的列数和 Y 的行数(k)

EBX（及变量 Dj）：Y 和 Z 的列数(j)

③ 循环变量

iCnt

jCnt

(2) 程序设计

程序设三层嵌套循环。

最内层循环执行

$$Z_{mn} = \sum_{r=1}^{k}(X_{mr} * Y_{rn}),$$

产生一个矩阵元素。m、n 不变，循环变量为 r，循环 k 次。

第二层循环产生 Z 的一行中的 j 个元素。m 不变，循环变量为 n，循环 j 次。最外层循环产生 Z 的所有行的 i×j 个元素。循环变量为 m，循环 i 次。

(3) 框图(图 6.19)

(4) 程序清单

程序 6.10：矩阵乘法

```
Di      EQU SS: dword ptr [EBP-4]
Dk      EQU SS: dword ptr [EBP-8]
Dj      EQU SS: dword ptr [EBP-12]
Zoff    EQU SS: dword ptr [EBP-16]
iCnt    EQU SS: dword ptr [EBP-20]
jCnt    EQU SS: dword ptr [EBP-24]

MatMult  PROC
  ENTER 24,0
  MOV   Di, EAX
  MOV   Dk, ECX
  MOV   Dj, EBX
  MOV   Zoff, EDX
  CMP   EAX,0
  JZ    Done
  CMP   ECX,0
  JZ    Done
  CMP   EBX,0
  JZ    Done
  MOV   iCnt,0
  MOV   EDX,0

Loop_i:
  MOV   jCnt,0

Loop_j:
  MOV   EAX,iCnt
  IMUL  EAX,Dk
  MOV   EBX,jCnt
  FLDZ
  MOV   ECX,Dk

Loop_k:
  DEC   ECX
  FLD   qword ptr [ESI+EAX * 8]
  INC   EAX
  FLD   qword ptr [EDI+EBX * 8]
  ADD   EBX,Dj
  FMULP  ST(1), ST
  FADDP  ST(1), ST
  CMP   ECX,0
  JA    Loop_k
  MOV   EAX,Zoff
  FSTP  qword ptr [EAX+EDX * 8]
  INC   EDX
  INC   jCnt
  MOV   EAX,Dj
  CMP   EAX,jCnt
  JA    Loop_j
  INC   iCnt
  MOV   EAX,Di
  CMP   EAX,iCnt
  JA    Loop_i

Done
  LEAVE
  RET
MatMult  ENDP
```

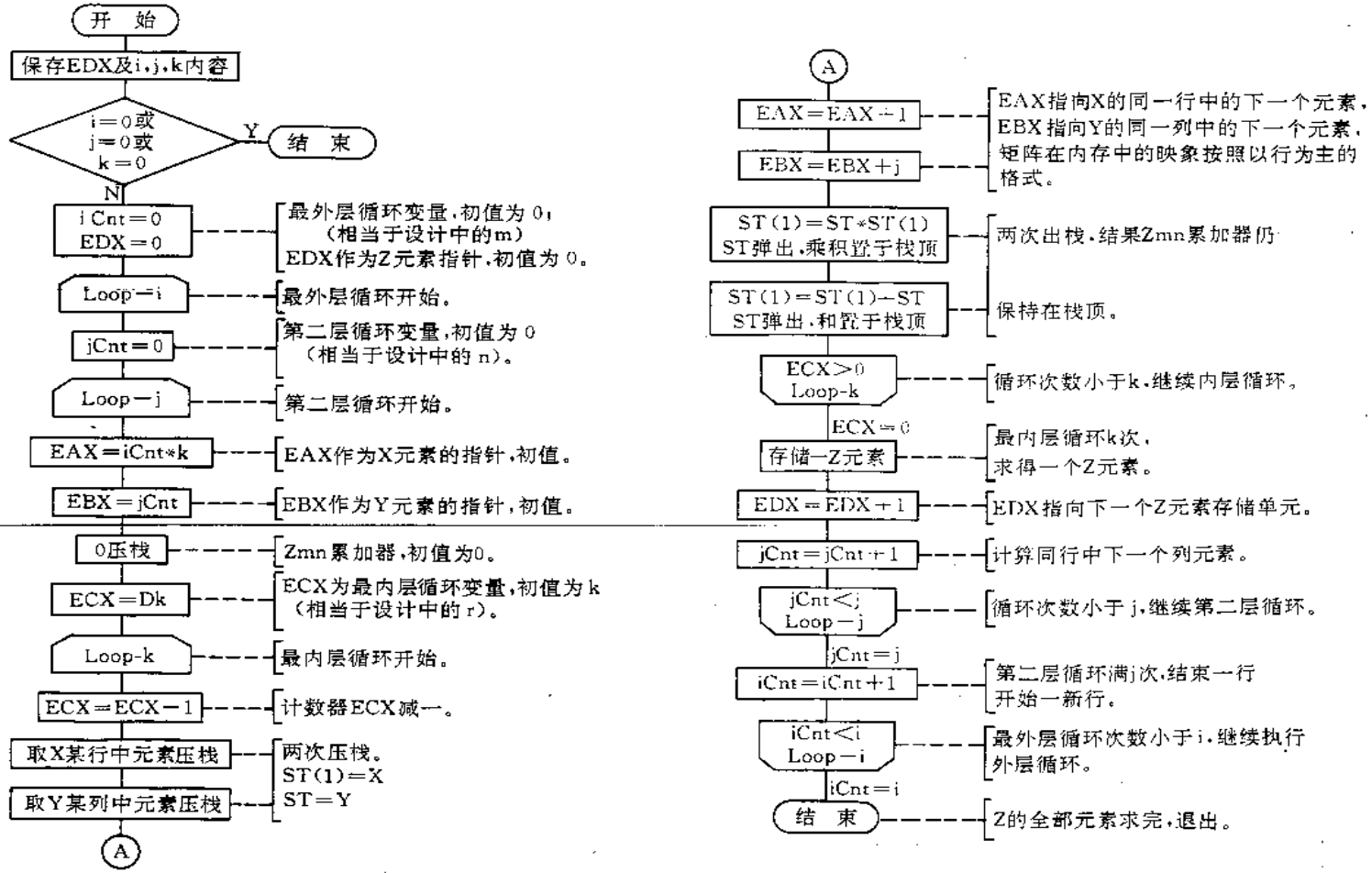


图 6.19 矩阵乘法程序流程

6.1.3.5 统计学

1. 概述

关于统计学，我们知道得最多的、最感兴趣的该是平均值和标准偏差。

统计学的对象是一组数。这组数据的个数为 n，它们的值为 X_i。这组数据的平均值

$$\mu = \frac{\sum_{i=0}^{n-1} X_i}{n}$$

标准偏差是相对于平均值的平均离差的度量，它等于

$$S = \sqrt{\frac{\sum (X-\mu)^2}{n-1}}。$$

上式不便于计算，因为先要求出平均值（第一次扫描整个数列），再求出数列的每个元素的离差的方均根值（第二次扫描整个数列）。

将上式稍作变换，变成适合于编程的公式：

$$S = \sqrt{\frac{\sum X^2 - \frac{\left(\sum X\right)^2}{n}}{n-1}}$$

程序 6.11 就是根据上式计算标准偏差的程序。其主循环计算 $\sum X$ 和 $\sum X^2$。然后再根据 $\sum X$ 和 $\sum X^2$，计算出 μ 和 S，分别放在 ST(1)和 ST 中。

过程的最后部分，主要依靠累加器堆栈来进行运算。这多少有点难懂。为了帮助理解堆栈的操作，我们在程序清单中给出了每步操作之后的堆栈数据。

2. 程序设计

(1) 寄存器和变量的初始化

ECX：数列元素个数

ESI：在 DS 段内的数列元素指针

Count：计数器，初始化为元素个数

(2) 框图 （图 6.20）

(3) 程序清单

程序 6.11：统计学

```
  Count  EQU SS: dword ptr [EBP-4]
STAT  PROC
  ENTER  4,0
  MOV   Count, ECX          ;保存元素个数

  FLDZ                      ;两个 0 压栈。一个为 X 的累加器，一个为 X² 的累加器，赋初值为 0。
  FLDZ                      ;
Loop Top:                   ;计算 ∑X 和 ∑X² 的循环开始
  FLD qword ptr [ESI+ECX*8]  ;取出一个元素，ST=Xi
  FADD ST(2), ST            ;累加 X，ST(2) = ∑Xi
  FMUL ST,ST(0)             ;栈顶 ST(0)=(Xi)²
  FADDP ST(1),ST            ;弹出一个栈顶，ST = ∑X²,ST(1) = ∑X
  LooPNZ Loop Top           ;ECX 减一，ECX 不为 0，再循环
```

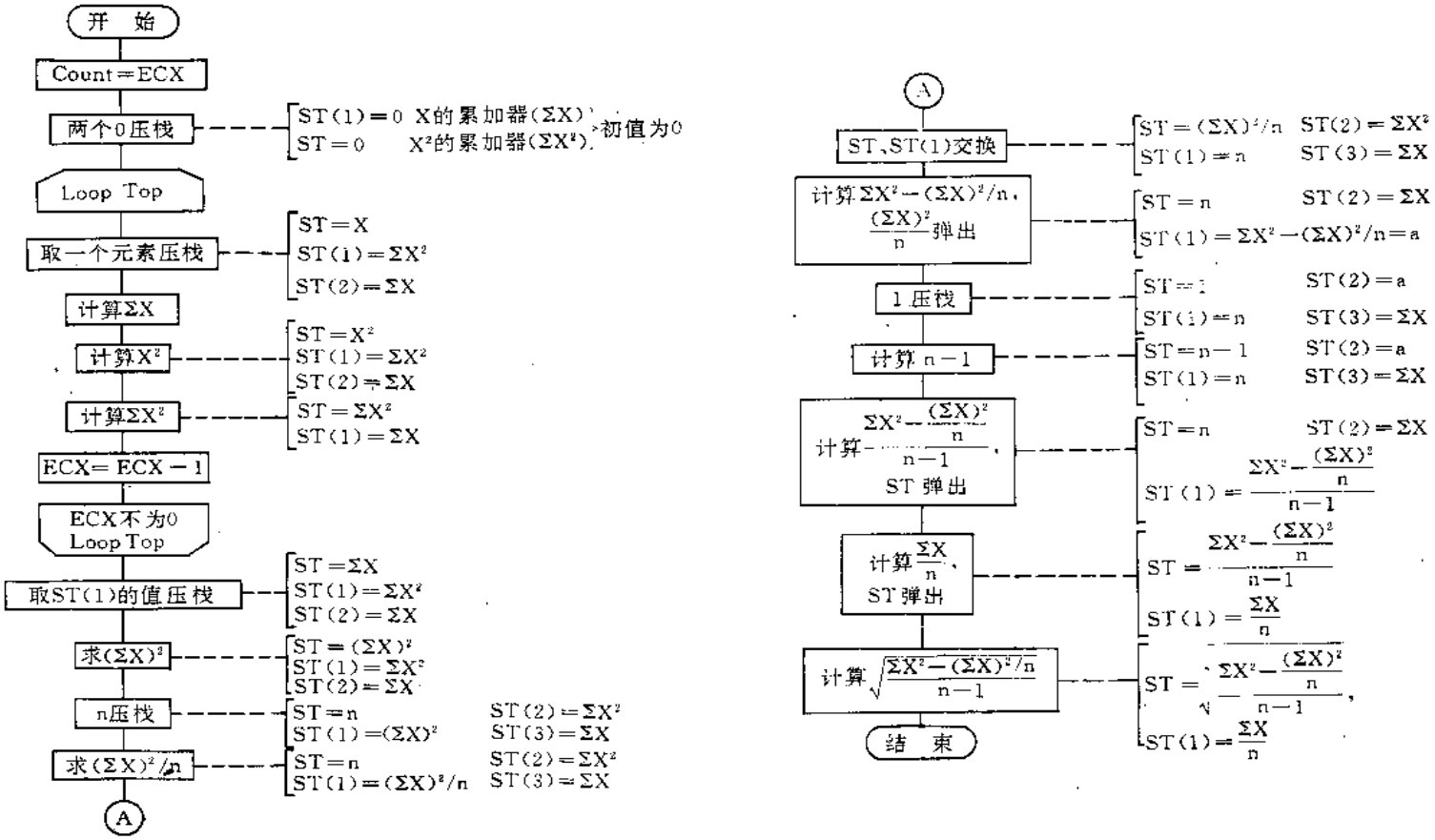


图 6.20 统计学程序流程

;堆栈状态表:	ST	ST(1)	ST(2)	ST(3)
;循环结束时	$\sum X^2$	$\sum X$		
FLD ST(1)	; $\sum X$	$\sum X^2$	$\sum X$	
FMUL ST,ST(0)	; $(\sum X)^2$	$\sum X^2$	$\sum X$	
FILD Count	; n	$(\sum X)^2$	$\sum X^2$	$\sum X$
FDIV ST(1), ST	; n	$\frac{(\sum X)^2}{n}$	$\sum X^2$	$\sum X$
FXCH ST,ST(1)	; $\frac{(\sum X)^2}{n}$	n	$\sum X^2$	$\sum X$
FSUBP ST(2),ST	; n	a	$\sum X$	$\left(a = \sum X^2 - \frac{(\sum X)^2}{n}\right)$
FLD1	; 1	n	a	$\sum X$
FSUBR ST,ST(1)	; n − 1	n	a	$\sum X$
FDIVP ST(2),ST	; n	S^2	$\sum X$	$\left(S^2 = \frac{a}{n-1}\right)$
FDIVP ST(2),ST	; S^2	$\frac{\sum X}{n}$		
FSQRT	; S	μ		

```
  LEAVE
  RET
STAT  ENDP
```

6.2 80386 的编程

上节中提供了一组实例用以说明和展示应用程序编程人员可以利用的 80386 的特性和指令。这些特性前几章中已有介绍。所提供的 80386 的特性和指令,是供操作系统编程人员使用的。本节进一步提供一个使用这些操作系统特性的实例(80386 初始化实例)。

6.2.1 句法

如 6.1 节所说明的,我们需要提供一些汇编语言的语法。多数新的结构与段的定义有关。

1. SEGMENT/ENDS

SEGMENT 及 ENDS 是汇编程序指令,用于定义段的开始和结束。这样的指令不会建立任何目标代码,而是告诉汇编程序如何去建立代码。

2. USE32/USE16

USE 是一条汇编程序的指令,该指令表示段的内容是 32 位的或 16 位的代码或数据。一个 USE32 段,将使汇编程序在假定段的描述符在 D=1 的条件下,建立目标代码。请注意,由于汇编程序不进行有关检查,如编程人员编程有错,且未设置描述符的 D 位,则程序将失败。因为在这种情况下,段内的数据或代码的长度不正确。

3. ORG

ORG 为汇编指令,用于定义段内的偏移量。

4. ASSUME

ASSUME 命令仅对代码段有意义。该命令用于指示在段内假定段寄存器的内容。不加上这条命令,汇编程序将给出警告信息,说明要寻址的段可能在当前是不可寻址的。请注意,该命

令并不产生将规定的段装入指定的段寄存器的任何代码。

5. 句法举例

用一个简单的例子说明上述各命令的用法：

```
Test_Code_Seg  SEGMENT  USE 32
ASSUME  DS:My_Segment
    MOV  AX, My_Segment_Sel
    MOV  DS,AX
    ...
    ...
    ORG  0FF0h
    ...
Test_Code_Seg  ENDS
```

此例中，该程序段的段名为 Test_Code_Seg，该段定义的范围由段名后的 SEGMENT 及 ENDS 指令限定。USE32 指示该段是一个 32 位的段。该段在汇编时假定 DS 寻址段名为 My_Segment 的段。因为汇编指令 ASSUME 并不将 My_Segment 装入 DS，所以在段中写了两行代码(前两行)。通过将 My_Segment 的选择子装入 DS 而使 DS 投影寄存器中有 My_Segment 的内容。最后，ORG 0FF0h 后面的代码定位于段内偏移量为 0FF0h 的位置。

6.2.2 初始化实例概述

本例介绍系统从初始状态开始转变为一个 32 位的系统，具有分页功能，可运行多个任务。在开始详细说明代码之前，首先介绍一下在完成本例之后机器应处于怎样的状态，将是有益的。了解这一点之后，通过什么办法使机器如何达到这样的状态就变得容易理解了。

本例为操作系统提供了一个简单的核心。一个实际的简单操作系统由上千行代码组成(复杂的操作系统达到百万行数量级代码，不可能作为这里的例子来介绍)，我们这里略去了很多内容，但用于展示多数操作系统的最基本的内容都已包括在本例中。

1. 假定条件

关于基础硬件，我们做如下假定：

(1) 操作系统代码驻留在 ROM 中，占用物理存储器的最高地址。假定 ROM 至少有 64K 字节大小(64K 时的开始地址为 FFFF0000h，末地址为 FFFFFFFFh)。

(2) RAM 从零地址开始，机器中至少配置有 48K 字节的存储器。

2. 多任务及保护

作为本例的最后结果，必要的操作系统表格应已经建立，一个单用户任务已被调用。我们已经注意到，要允许多个任务在此机器上同时运行。本例使用 2 个保护等级：0 级供操作系统使用，3 级供应用程序使用。操作系统由系统内的所有任务共享。每个任务有各自的私有代码及数据分区，存储在局部地址空间，因而对系统中的其它任务不可见。操作系统由所有任务共享，存储在全局地址空间。图 6.21 描述了这一多任务系统。同心圆代表保护级，本例中只使用 0 级和 3 级。在 3 级运行的任务用射线分开，图中表示有 N 个任务同时在系统中运行，相邻两条射线间的射线片表示一个任务。除了每个任务的虚拟地址空间的全局地址区域之外，一个任务的地址与其它任务的相同地址没有任何关系。

3. 虚拟地址空间

图 6.22 描绘每个任务的虚拟地址空间。虚拟地址空间由全局地址空间和局部地址空间两部分组成。全局地址空间由 GDT 映射，局部地址空间由 LDT 映射。全局地址空间包含如下 5

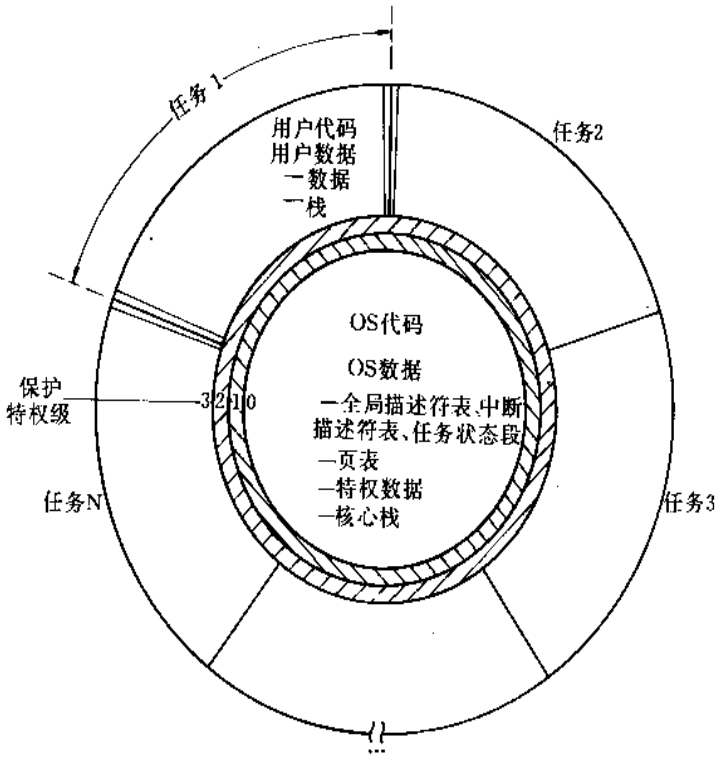


图 6.21　多任务系统

个段：

(1) GDT 本身，包含有在全局地址空间中定义的其他四个段的描述符；

(2) 第一个任务的 TSS。当有多个任务运行时，应为每一任务定义各自的 TSS；

(3) 操作系统数据段；

(4) 操作系统代码段；

(5) 第一个任务的 LDT，用以映射该任务局部地址空间中的各个段。当有多个任务运行时，应为每一任务定义各自的 LDT。

局部地址空间包含如下两个段：

(1) 用户代码段；

(2) 用户数据段(包含数据及栈)。

4. 线性地址空间

线性地址空间由图 6.23 示出。图中有四个线性地址空间，分述如下：

(1) 操作系统代码段(OS code)，段大小为 60K，起始地址为 FFFF0000h，末地址为 FFFFEFFFh。我们这里假定操作系统仅有 64K 大小，并可存放于一个 ROM 芯片中，仅是为了说明的方便。

(2) boot 代码段，段长 4K 字节，起始地址为 FFFFF000h，末地址为 FFFFFFFFh。这是 64K 操作系统的最高 4K 代码，这段代码工作于实地址方式。一旦系统进入保护方式，这部分地址即成为不可寻址的。

(3) 操作系统数据(OS data)段，段大小为 16M 字节，开始地址为线性地址 0，末地址为线性地址 00FFFFFFh。该段包含系统段表(GDT、IDT、TSS)、页表(目录表及第二级表)、操作系

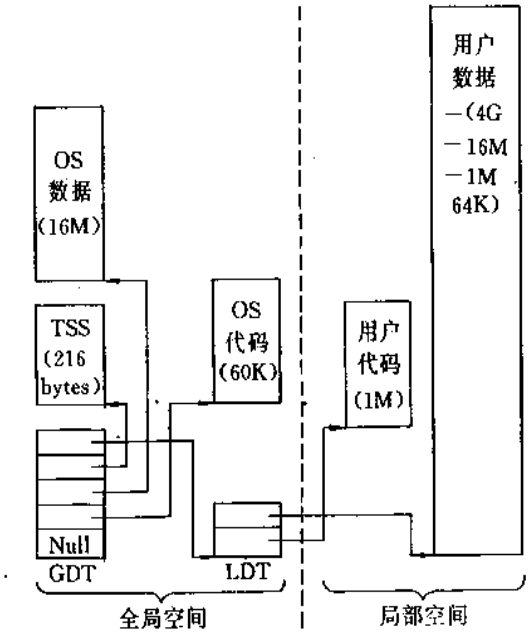


图 6.22 虚拟地址空间

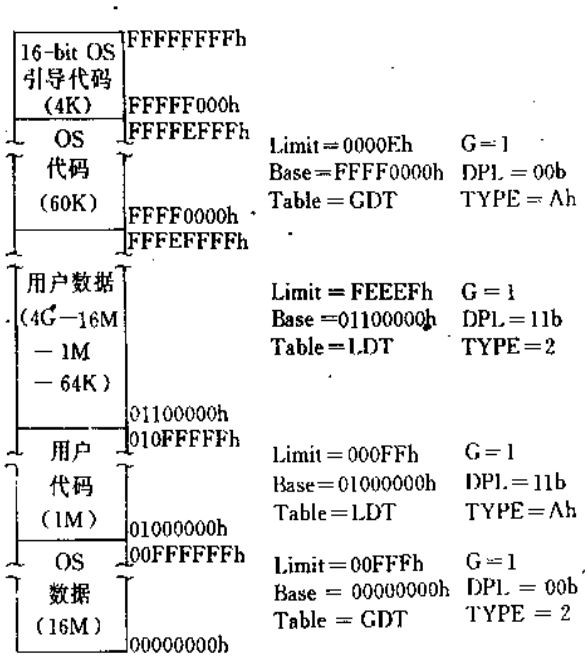


图 6.23 任务的线性地址空间

统局部数据及操作系统的栈。

(4) 用户代码段,段大小为 1M 字节。起始地址为线性地址 01000000h,末地址为 010FFFFFh。段的大小选为 1M 没有特别的含义。当任务被实际装入时,已经确切地知道代码段的实际大小,这时再对段的界限进行适当的调整。

(5) 用户数据段：在线性地址空间除操作系统及用户代码所占空间外，其余的全部地址空间都属数据段的空间，因此数据段的大小为：

4G-64K-16M-1M

这确实是一个非常大的地址空间！该段的起始地址为 01100000h，末地址为 FFFEFFFFh。在此空间中栈从段的顶部即地址 FFFEFFFFh 开始，并向下增长；数据从段的底部，即地址 01100000h 开始，并向上增长。

上述各段驻留在线性地址空间中的地址选择，除 boot 代码必须从物理地址的 FFFFFFF0h 开始，故我们选在最高 4K 地址外，其余各段也可以选择任意其它范围。

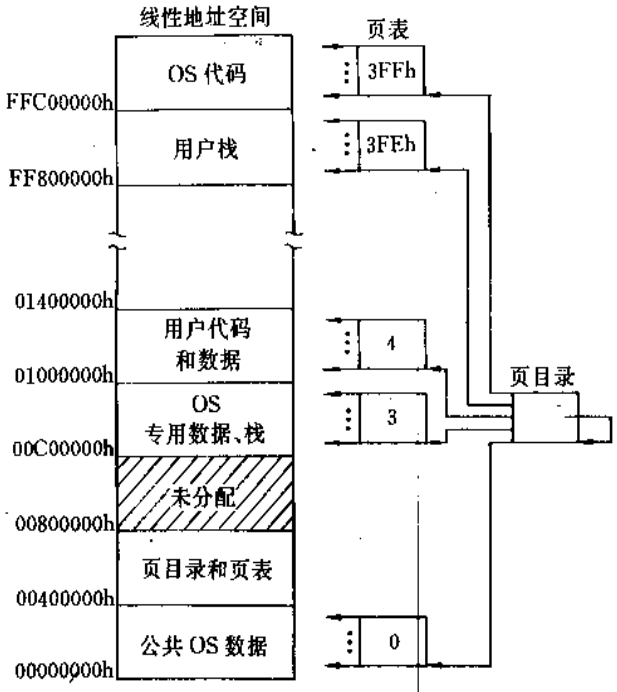


图 6.24　页表与线性地址空间的映射

5. 页映射

线性地址空间——经段机制转换后的地址空间，由图 6.24 所示的页表进行映射。在图 6.24 中，我们将线性地址空间分为 4M 字节大小的区域，每一区域恰好可由一个页表进行映射（一个页表含 1K 表项，每一表项映射一页即 4K 字节，所以每个页表共映射 1K×4K＝4M 字节）。图中共建立了五个页表，页目录中有五个表项分别指向这五个页表。页目录中还有第六个目录项，该目录项指向页目录本身。指向页目录的目录项，是把页表映射回线性地址空间，并允许对其进行访问的一种简单方法（下面将详细说明）。建立的六个目录项及页表为：

(1) 公共操作系统数据：GDT、IDT 及 TSS；

(2) 页目录及页表；

(3) 操作系统专用数据、操作系统其它数据及操作系统栈；

(4) 用户代码和数据；

(5) 用户栈；

(6) 操作系统代码。

6. 物理地址

物理地址空间由图 6.25 给出，共分配了 28 页物理地址。较低的 12 页定义在 RAM 区域，从地址 0 所在页开始，直到地址 0B000h 开始的那一页。另 16 页分配在存储器的高地址端，从地址 0FFFF0000h 开始直到地址 0FFFFFFFFh，这些页对应于存储器最高地址端的 64KB ROM 区域。

上面讨论了线性地址空间和物理地址空间，下面做一些比较：

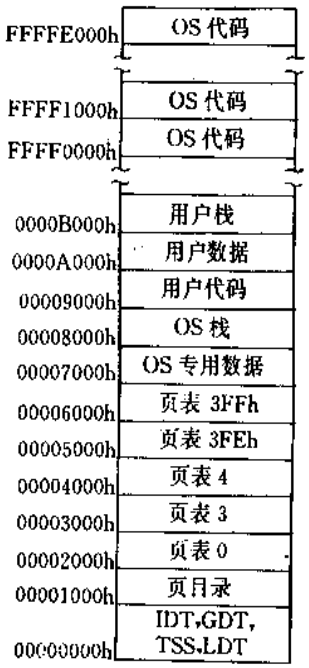


图 6.25 物理地址空间

(1) 操作系统代码区域都在两个地址空间中的最高 64K 字节。即线性地址与物理地址相同。这使得在机器初始化时的操作变得简单。操作系统代码在不同地址空间中可以有不同的地址，但那将使本例难于理解，特别是在系统的保护及分页机制启用之后更是如此。

(2) GDT、IDT、TSS 及 LDT 表都在两个地址空间中的相同的最低地址。如果不是象这样安排，则这些表必须重新建立并进行传送，而且在保护机制、分页机制启用之后，还需重新装入描述符。为避免这些问题，特意安排这些表在两种地址空间有相同地址的区域，因而只需装入和建立一次。

(3) 其他页的物理地址与线性地址不保证一致，这是为了尽量减少映射线性地址空间的地址所需物理存储器容量的总数。这是分页机制的主要优点之一：允许线性地址空间大于物理存储器容量。在我们的例子中，物理页与该页在对应的线性地址中出现的顺序大致相同(当然 1、2 两点除外)。然而，实际上在物理存储器中的页的地址可以是随意安排的。事实上，在虚拟存储系统中，每一物理页都可能被不断替换，由分页机制用一个活动页取代一个不活动页。分页机制不仅可以把线性地址空间的一部分换出，而且可以把页表换出，其目的是减少用于操作系统的物理存储容量。在换出页表时，要注意更新无效页表项。页目录不允许换出，必须常驻内存，改变页目录的位置只有更新页目录基寄存器(CR3)。

7. 任务切换

这里，我们不去注意建立任务及切换到新任务的细节，只是指出完成任务切换所需做的事情：

(1) 为新任务建立 LDT，并在 GDT 中建立一个表项指向该 LDT。新任务，除可能有不同的代码空间大小以外，与实例中的任务完全相同。

(2) 为新任务建立 TSS，并在 GDT 中建立一个表项指向该 TSS。新任务的 TSS 类似于实例中的 TSS。

(3) 为新任务建立页目录，映射全局地址空间的页表等同于第一个任务，而且这些页目录项被两个任务(及将来的所有任务)共享。建立指向局部页表(用户代码，数据和栈)的页目录项。

(4) 建立新任务局部地址空间的页表，这与第一个任务的页表类似，只是映射在不同的物理页。

(5) 为切换到新任务，要求装入 TR、LDTR 及 CR0。实际的任务切换可用第 4 章描述的方

法完成。

6.2.3 初始化实例说明

有了如上的基础之后，下面就开始深入到实例的若干细节(该实例的程序清单在本小节的说明之后列出)。可以看出该程序由若干短的程序按适当的顺序执行，形成了一个完整的初始化过程。

1. Equates

这是实例程序的第一部分，该部分进行如下定义：

- 定义助记符；
- 定义 GDT 及 LDT 的选择子；
- 定义页的物理地址(因为线性地址与物理地址不匹配，故使用的是助记符)，这些地址与图 6.25 表示物理地址空间的地址一致。

2. 段定义

接下来定义实例中的四个段(即 OS_Data_Seg、User_Code_Seg，User_Data_Seg 及 OS_Code_Seg)。这些段的定义分别与图 6.22、图 6.23 给出的虚拟及线性地址空间相一致。这里给出的段定义，使我们可以用符号对分布在本例其余部分的段进行访问。以这种方式声明的段不对存储器段初始化。

注意在 OS_Code 段中，不包括最高的 4K 存储器。操作系统代码的最高 4K 包含在名为 BootRom16 的一个独立的段中，即本例代码的最后一段程序。BootRom16 是一个 16 位的段，而 OS Code Seg 是一个 32 位的段。由于 80386 开始处于 16 位的操作方式，因此，需要有一小段程序使程序从 16 位的操作方式转到 32 位的操作方式。进入保护方式之后，这段 16 位代码段即变为不可被访问。本例中，我们使系统在 16 位方式下花费的时间减至最小，这与本书的目的——集中于 32 的 80386 是一致的。

3. 代码 1——冷启动

系统复位时，80386 以一种称为实地址的方式开始操作(第 7 章将做详细介绍)。实地址方式可以简单地视为 16 位的机器，最大的段为 64K 字节。实地址方式中，选择子左移四位与段内偏移量相加形成物理地址，因此，最大的物理地址为 FFFFFh，即整个地址空间为 1M 字节。复位时的机器状态见表 7.3，表中有些初始化机器状态在下面的叙述中要用到。

系统复位时，将 CS 寄存器初始化为 F000h，IP 初始化为 FFF0h。在实地址方式下，由 CS 和 IP 形成的 20 位地址为：

$$F000h \times 16 + FFF0h = FFFF0h$$

也就是说，复位后执行的第一条指令的地址在 F000h 段的 FFF0h 偏移处。通常，我们总是把系统复位后要执行的程序做成固件固化在 ROM 中，并占用内存空间的一个存储段，在我们这里的情形，应占用的段地址为 F0000h。但是，80386 是 32 位的系统，数据和地址都是 32 位的。如果把固件所占用的段地址定在 F0000h，则系统的内存地址空间被固件所在地址分成不连续的两个部分，造成对内存资源进行有效管理的困难。因此，80386 硬件规定，系统复位时，最高 12 位地址线(A31～A20)用一个触发器的“1”状态强制为高电平，从而使固件所在的段从 F000h 平移至 FFFF000h，第一条要执行的指令的地址也由 FFFF0h 变为 FFFFFFF0h。由于 FFFF000h 段是 32 位系统的地址空间的最后一个 64K 存储段，所以内存地址空间的其余部分构成一个完整的连续区域。最高 12 位地址线强制为“1”的状态持续至从 FFFFFFF0h 开始执

行的程序中，遇到的第一个段间转移指令为止。这是因为硬件设计中，每当执行段间转移指令时，都将前述触发器置为“0”的缘故。这一方法是所有机器在系统初起时，总是从定位在最高存储区域的固件程序的预定地址开始执行的常用方法，从而也要求 Boot ROM 总是置于存储器的最高端。

冷启动代码在 80386 初始化实例程序清单的最后，规定 ORG 为 0FFF0h 是为了满足硬件的要求。

4. 代码 2 —置初值

这段程序开始于标号 Start 16(程序清单中 BootRom16 段的开始处)。首先将 IF 位置 0，接着再将 IDT 的界限置 0。前者用于屏蔽任何可屏蔽的中断，后者用于万一发生并接收了不可屏蔽中断而使系统马上处于关机状态(见第 5 章)。为使 IDT 的界限为 0，需使用 LIDT 指令，而该指令要求有一存储器操作数，因此需将 DS 寄存器初始化，并指向 OS_Data_Seg。在本例中，因为 DS 寄存器初始化为 0，而 OS_Data_Seg 的偏移量也恰好为 0，所以初始化 DS 并指向 OS_Data_Seg 变得没有必要。然而，在多数情形下，要使用一个段必须先初始化段寄存器，所以在实例中沿用了常规办法。

5. 代码 3——建立 GDT，LGDT

代码的第三部分建立 GDT(全局描述符表)，并将 GDT 伪描述符装入处理器。

首先，要用适当的信息填入 GDT。本例中，我们有五个 GDT 表项(NULL、OS Code、OS Data、LDT 及 TSS)。表 6.2 列出了这五个描述符的内容。程序代码就是要将表 6.2 中描述符的内容进行适当分离，使其形成在图 4.10 中给出的描述符的格式。表 6.3 列出了这里使用的描述符的类型。

表 6.2

描述符名	描述符所在描述符表	界限 (LIMIT)	基地址 (BASE)	属性类型 (TYPE)	描述符类型 (DT)	特权级 (DPL)	存在位 (P)	保护方式 (D)	粒度位 (G)
NULL	GDT	00000	00000000	0	0	0	0	0	0
OS 代码	GDT	0000E	FFFF0000	A	1	0	1	1	1
OS 数据	GDT	FFFFF	00000000	2	1	0	1	1	1
用户代码	LDT	000FF	01000000	A	1	3	1	1	1
用户数据	LDT	FEEEF	01100000	2	1	3	1	1	1
TSS1	GDT	000D7	000000B0	9	0	0	1	0*	0
LDT1	GDT	0000F	00000188	2	0	0	1	0*	0

*注　系统段的 D 位无意义

表 6.3　描述符及描述符属性类型

描述符类型	属性类型	描述符说明
1	2	存储段——可读可写数据
1	A	存储段——可执行可读代码
0	9	系统段——可用 386TSS
0	2	系统段——LDT

然后在存储器中建立伪描述符。伪描述符的低位字为 GDT 的界限，其值为描述符表项数的 8 倍减 1，即 GDT 内的最大偏移量。伪描述符的高位双字是 GDT 的线性地址。

最后，用 LGDT 指令将伪描述符装入处理器。

6. 代码 4——进入保护方式

下一步是要设置机器状态字的 PE 位。在 PE 位置位之后，要立即执行一条跳转指令，清除芯片上的预取指令及译码序列。这一点之所以重要，是因为在 PE 位置位后，机器从实地址方式转入了保护方式，而已预取及译码的指令则从属于不再有效的实地址方式。至此，机器已进入了保护方式，但数据和偏移量还都是 16 位的。然后，再执行一条段间 JMP 指令，该指令转向 OS Code 段的 Start 32。由于 GDT 中定义 OS Code 段为 32 位的代码段，所以在 JMP 指令执行完毕之后，机器又进一步转变为一个 32 位机，这以后，机器就进入了我们在第 1 章～第 5 章所假定的状态。

7. 代码 5——装入段描述符

本段程序从标号 Start 32 开始(在清单中 32 位 OS 段的前部)。由于机器已处于保护方式，因此有必要重新装入段寄存器使得数据段变为可寻址的段。在前面的代码 2 中曾用选择子装入过 DS，但在那里是实地址方式，通过 DS 及偏移量形成的物理地址最多只寻址 1M 字节。在启用保护方式之后 DS 装入选择子，是为了在对应的投影寄存器中得到描述符。通过描述符不仅建立起各种保护信息，还使偏移量及基地址都扩展为 32 位，寻址范围扩大为 4G。装入 DS 的选择子还装入 ES，以供串操作指令使用。

8. 代码 6——建立 LDT、LLDT

任务的局部地址空间由 LDT(局部描述符表)进行映射。LDT 包含两个表项，一个为用户代码段，另一个为用户数据段(见图 6.22)。建立 GDT 时，GDT 中应有映射 LDT 的表项。LDT 本身也作为一个数据段存在，并定义为 OS Data 段的一部分。初始化实例中，当 80386 使用实际的 LDT 段，用 OS_Data 段中的别名建立 LDT 时，要用适当的信息填入 LDT。在实例中，也就是用户代码、用户数据两个段的有关信息。这些信息也列在表 6.2 中。程序代码将表 6.2 中 User Code 及 User Data 两行内容进行适当分离，使其符合描述符的格式，再写入 LDT 中。在 LDT 建立后，再把访问 LDT 的选择子装入 LDTR。此后，LDT 即可被访问。

很多保护模式的初始化(GDT、IDT、TSS)实际上既可在进入保护方式之前，也可在进入保护方式之后进行。我们这个实例的目标是要尽量减少 16 位程序代码的数量。所以除 GDT 在保护方式前进行初始化以外，其它的都放在进入保护方式之后才初始化。而装入 LDTR 只有在保护方式下才能进行，所以这部分代码放的位置就没有选择余地了。

9. 代码 7——建立 IDT

初始化中的一个很重要的部分是建立 IDT(中断描述符表)。在我们的实例中，当进入用户方式之后，用户任务访问操作系统服务的唯一途径，就是借助于异常处理机制。读者自然会问，在第 4 章已讨论过借助门不是也可在低特权级访问高特权级的程序吗？这当然是对的，但是实例中我们没有定义门。

事实上，IDT 表项本身即是一个门——中断门。(门描述符的格式见图 4.12)。由于整个操作系统在实例中对当前任务是可见的，所以在 IDT 表项中我们也不必使用任务门。实例中中断门的基本内容见表 6.4。如果一个异常或中断在特权级 3 被响应，则要进行特权级的转换，相应地还要改变所使用的栈。因此，响应中断之后，代码特权级从 3 级切换到 0 级，相应的堆栈也从原来使用的 3 级栈改变为使用 0 级栈。

表 6.4 中断门中的内容

字段	值
Offset	默认处理程序的偏移量
Selector	处理程序代码的段的选择子
Dword Count	0,不拷贝参数
Type	E,386 中断门类型
DPL	0

OS_Data 段中的别名用来访问 IDT。开始时,所有 17 个中断门都填入相同的信息。这 17 个中断级是由 80386 预先定义的,实例中也未增加另外的中断级。中断门中的信息是指向处理程序入口的,相同的中断门信息就指向相同的处理程序入口。在我们这里就指向系统的默认中断处理程序,该处理程序实际上是不做任何处理即返回。完成这一步之后,再对那些在系统中有实际意义的中断,由操作系统提供特别的处理程序,即将转入该处理程序所需的信息填入对应的 IDT 表项中,修改并替换原来已填入的默认信息。对于所有有意义的中断,替换相应 IDT 表项中的信息都完成之后,IDT 的建立即告结束。

10. 代码 8——建立页目录

段机制建立起来以后,即可着手建立分页机制。在第 4 章的讨论中已指出,页机制有两级存储器驻留转换表:即页目录表及页表。代码 8 讨论建立页目录,代码 9 讨论建立页表。

下面先讨论建立页目录。为了容易理解程序,先回忆一下有关页目录的知识:

- 每一任务有一页目录表,最多可有 1K 个页目录表项各指向一个页表;
- 每一页表最多可有 1K 个页表项,各指向一页;
- 每一页为 4K 字节。

页目录表项格式参见图 4.17。每个页目录表项格式除含有指向对应 4M 字节线性地址空间的页表地址外,还含有目录表项本身的若干属性。如属性中的存在位为 0,则该表项可视为空,对应的 4M 线性地址空间也就不存在。如果某一任务实际使用了 N 个页表(注意,不一定每一页表都用满 4M 字节),则应在页目录中填入 N 个目录项。填入的信息有:

(1) 存在位 (P=1);

(2) 用户/系统特权级;

(3) 读/写方式;

(4) 所指向页表的物理地址。

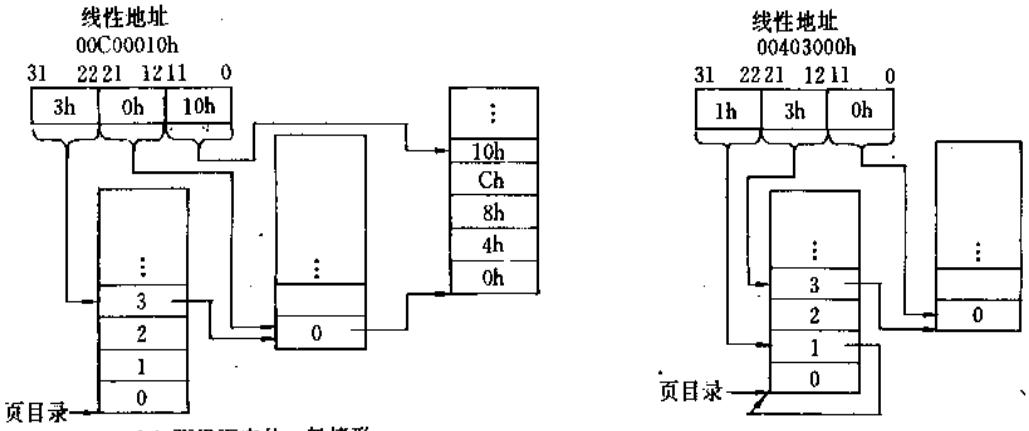
因此,建立页目录表的程序分为两个部分:第一部分将页目录表的所有 1K 个表项的存在位都置 0;第二部分将映射有效线性地址空间的表项用上述四个方面的信息填入。

实例中实际填入了六个页目录表项,页表地址可从前面的 EQU 指令中查到。要请读者注意的第二个表项,即 entry1,页目录表项指向的页表是页目录本身。这一点粗看起来容易使人发生混淆,但是,这确是把页目录和页表映射到线性地址空间中的一种很方便的方法。为此我们稍用一点篇幅并以图 6.26 为例加以说明(下面的例中我们假定页目录项(0000000001)$_b$ 指向页目录本身。

例 1 设线性地址为 00C00010h,将其分解为页目录项,页表项及页内偏移量的形式即变为:

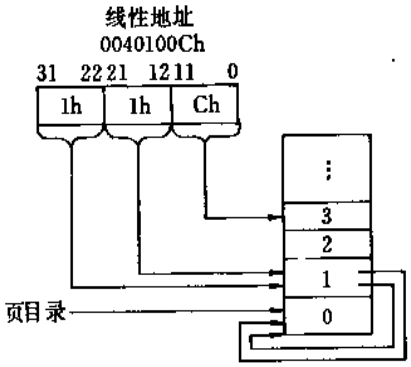
31	22 21	12 11	0
0000000011	0000000000	000000010000	

这是通过页目录 3、页表项 0 及偏移量所进行的正常的访问，见图 6.26(a)。



(a) 两级页表的一般情形

(b) 页目录表项指向页目录表



(c) 页目录表项及页表项都指向页目录表

图 6.26 指向页目录本身的页目录表项

例 2 设线性地址为 00403000h，将其分解为页目录项、页表项及页内偏移量的形式即变为：

31	22 21	12 11	0
0000000001	0000000011	000000000000	

这是以页目录为页表，并对第四个表项所指页中的 0 偏移量地址进行访问，见图 6.26(b)。

例 3 设线性地址为 0040100Ch。将此线性地址分解为页目录项、页表项及页内偏移量后的形式即变为：

31	22 21	12 11	0
0000000001	0000000001	000000001100	

这时的页表及页都是页目录本身，此线性地址即是对页目录表中偏移量为 12 的地址进行访

问，见图 6.26(c)。

11. 代码 9——建立页表

建立页表与建立页目录表有完全类似的过程。首先，将所有页表项标志为不存在，然后，将有效表项填入相应的信息。因为一个任务只有一个页目录表，所以建立页目录表只有一个这样的过程。而一个任务的页表则可能有多个(本实例中即有 5 个页表)，因此页表的建立就有多个这样的过程。前面已经讨论过，使线性地址等于物理地址在建立页表中是一个很重要的考虑。在代码 9 建立的页表中，下列线性地址等同于物理地址：

(1) 操作系统代码在 ROM 之外执行时，操作系统在以物理地址 FFFF0000h，FFFF1000h，……，FFFFF000h 开始的 16 个页应该赋予等同的线性地址，也就是使操作系统代码的线性地址等于物理地址；否则在分页机制启用之后，就会发生莫明其妙的跳转，产生难以预料的后果。

(2) GDT、IDT 及 TSS 存储在物理地址开始的第一页。对于该页，也赋予相同的线性地址，以避免分页机制启用之后，要重新装入相应寄存器及重建相联系的表。

表 6.5 列出的是代码 9 建立的 28 页页表对应的目录项、页表项及访问属性。

表 6.5 已分配页具体内容

页目录表项	页表表项	系统/用户	写权限	页内容
0	0	系统	无	IDT、GDT、TSS
1	0	系统	无	页表 0
1	1	系统	无	页目录
1	3	系统	无	页表 3
1	4	系统	无	页表 4
1	3FEh	系统	无	页表 3FE
1	3FFh	系统	无	页表 3FF
3	0	系统	无	OS 专用数据
3	3FFh	系统	无	OS 栈
4	0	用户	无	用户代码
4	100h	用户	有	用户数据
3FEh	3FFh	用户	有	用户栈
3FFh	3F0～3FFh	系统	无	OS 代码

12. 代码 10——启用分页机制

建立了页目录表及页表之后，即可启用分页机制。这一工作非常简单，只需用一条指令将 CR0 的 GP 位置 1 即可达到目的。但是不可忘记在启用分页机制之后，要用一条跳转指令清除指令预取队列及指令译码队列。这一点与进入保护方式后所要做的完全类似。分页机制被禁用的情况下，物理地址总是等于线性地址。然而分页机制启用之后，这一关系一般来说是不成立的，所以预取的指令队列及译码队列在 CR0 的 PG 位置 1 之后即变为无效，故做一次跳转是完全必需的。

13. 代码 11—装入操作系统栈指针

这部分程序很简单，总共才三行代码。第4章已经讨论过，不同特权级有不同的堆栈。本例使用两个特权级，故只需要两个堆栈。这里要提一下的是关于LSS指令的使用。LSS指令一次装入SS及ESP两个寄存器，其间不会由于中断而使SS及ESP的内容不一致而造成错误。在我们的程序中，到此为止，中断尚未开放，因而使用LSS指令也并非必需，但是，用LSS装入SS及ESP，而不是用两条指令完成同样的装入操作则是一个好的编程习惯。读者有这样的良好习惯，往往可以避免发生看起来非常简单然而又难以排除的编程错误。

14. 代码12——建立TSS

在特权级的改变或任务切换期间，需访问TSS以检索或存储信息。例如，系统中的第一次异常要求从用户特权级3转移到系统特权级0，作为特权转移的一部分，使用的栈也要加以改变。所以，期望能在TSS中的适当位置找到0级栈的选择子及偏移量。像这样的信息必须预先给出，这就是代码12的任务。而其它若干信息则是切换过程中自动装入的。代码12放入TSS中的信息有：

(1) 0级栈偏移量；
(2) 0级栈选择子；
(3) 页目录地址；
(4) LDT；
(5) I/O位图偏移量；
(6) 调试陷阱位(置为0)。

I/O位图偏移量在程序中置为8000，该值超过了TSS的界限，因而指的是一个空的位图。然后装入TR(任务寄存器)使TSS对处理机可见。还要注意，TSS有分配给浮点状态保存区的空间。保存区的最高字节设置为0，指示该任务尚未使用协处理器。至此，初始化的几乎所有方面都已完成。下面就准备装入用户任务。

15. 代码13——装入用户任务

实例中，假定有一个名字为Loader()的例程，该例程被调用后，即为用户装入任务，完成的功能有：

(1) 将用户任务从虚拟空间读入内存实空间。事实上只先读入一页，其他各页(如果有的话)在以后发生页故障时读入。

(2) 返回任务的开始地址，即第一条指令的偏移量，并存放在EAX寄存器。此后应将此值放入TSS中，以便调度到此任务后可从TSS中找到执行地址。

(3) 计算代码长度，并适当调整User_Code_Seg的界限及User_Data_Seg的基址。Loader()例程在实例中未予列出。

16. 代码14——为用户任务调用做准备

第4章已经讨论过，要从内层级通过调用切换到外层级是不可能的。系统初启时，工作于0级，而任务处于3级。为了能从0级转到3级执行用户任务，需要制造一个假的IRET环境，产生一个当前在0级运行是由于中断了用户程序而正做中断处理的假象。这样，用IRET启动系统的第一个用户任务就很自然了。代码14就制造这一假的IRET环境，在堆栈中压入如下信息：栈选择子；栈偏移量；标志；代码选择子；代码偏移量。

代码选择子与偏移量应为中断时的断点，但每个任务第一次执行时都在选择子及偏移量处写入程序要执行的第一条指令的地址，这就是Loader()返回在EAX中的地址。

注意，用在堆栈中的标志寄存器的映象中的IF位置位而启用中断。

17. 代码15——调用用户任务

实例的最后一步是执行IRET指令，该指令将把刚压入栈中的信息恢复至用户任务中，从而改变保护级、栈、CS:EIP及标志。IRET执行完毕后，即完成了对第一个用户任务的调用，实

例也就到此结束。以下为本实例的程序清单。

程序 6.12

```
;* * * * * * * * * * * * * * * * * * * * * * * * *
;* 80386 初始化实例程序清单
;* 使机器从复位状态开始,经初始化装入操作系统之后成为可用的系统,并调用第一个用户任务执行。
   本程序的主要任务是建立适当的段表及页表。实例由以下 15 步完成:
;* 1. 引导地址冷启动;
;* 2. 禁用中断、装入 DS 及使 IDT 为空;
;* 3. 建立 GDT 及 LGDT;
;* 4. 进入 32 位代码的保护方式;
;* 5. 装入段寄存器;
;* 6. 建立 LDT、LLDT;
;* 7. 建立 IDT、LIDT;
;* 8. 建立页目录;
;* 9. 建立页表;
;* 10. 启用分页机制;
;* 11. 装入 OS 栈;
;* 12. 建立 TSS、LTR;
;* 13. 装入用户任务;
;* 14. 为调用用户任务做准备;
;* 15. 调用用户任务。
;* * * * * * * * * * * * * * * * * * * * * * * * *
                    NAME    Initial
;各种常数
IDT_Entries         EQU     17
GDT_Entries         EQU     5       ;Null,OS Data,LDT   五个全局描述符表表项。
                                    ;OS Code, TSS
LDT_Entries         EQU     2       ;User Code, User Data   二个局部描述符表表项。
Page_Size           EQU     4096
Page_Entries        EQU     1024
Not_Present         EQU     0
PE                  EQU     1
PG                  EQU     80000000h
Int_Flag            EQU     0200h
;GDT 选择子
Null                EQU     0h      ;空描述符
OS_Code_Sel         EQU     8h      ;1,GDT,RPL=00
OS_Data_Sel         EQU     10h     ;2,GDT,RPL=00
LDT1_Sel            EQU     18h     ;3,GDT,RPL=00
TSS1_Sel            EQU     20h     ;4,GDT,RPL=00
;LDT 选择子
User_Code_Sel       EQU     7h      ;0,LDT,RPL=11
User_Data_Sel       EQU     0Fh     ;1,LDT,RPL=11
;物理地址常数,用于初始化页表
P_OS_GData          EQU     00000h
P_Page_Dir          EQU     01000h
P_Page_Tab0         EQU     02000h
P_Page_Tab3         EQU     03000h
P_Page_Tab4         EQU     04000h
P_Page_Tab3FE       EQU     05000h
P_Page_Tab3FF       EQU     06000h
P_OS_PData          EQU     07000h
P_OS_Stack          EQU     08000h
P_User_Code         EQU     09000h
```

```
P_User_Data        EQU     0A000h
P_User_Stack       EQU     0B000h
TSS_Size           EQU     104     ;TSS 段的最小段长。
FP_Save            EQU     112     ;浮点保存区域共 108 字节。另 4 个字节存放 FP 已被使用标
                                     志。
;* * * * * * * * * * * * *
;* * * * * * * * * * * * *
;* *OS 数据段
;* * * * * * * * * * * * *
;* * * * * * * * * * * * *
;包含段表及页表,该数据段的线性地址为 0
OS_Data_Seg        SEGMENT   RW   USE32
;IDT 表在线性地址为 0 的位置接下去是 GDT、TSS、LDT
IDT_Table          db  IDT_Entries * 8 DUP (?)
GDT_Table          db  GDT_Entries * 8 DUP (?)
TSS1_Table         db  (TSS_Size+FP_Save) DUP (?)
LDT1_Table         db  LDT_Entries * 8 DUP (?)
pdescr             db  6 dup (?)       ;为 IDTR、GDTR 建立伪描述符
;* * * * *
;* *物理地址空间中的页目录及页表区域,
;* *用于在启用分页机制之前初始化页表。
;* * * * *
                   ORG 00001000h              ;物理地址
Page_Dir           dd  Page_Entries DUP (?)
Page_Tab0          dd  Page_Entries DUP (?)   ;OS 公共数据(IDT、GDT、TSS)
Page_Tab3          dd  Page_Entries DUP (?)   ;OS 专用(栈)
Page_Tab4          dd  Page_Entries DUP (?)   ;用户代码及数据
Page_Tab3FE        dd  Page_Entries DUP (?)   ;用户栈
Page_Tab3FF        dd  Page_Entries DUP (?)   ;OS 代码
;* * * * *
;* *线性地址空间中的页目录及页表区域
;* *4M 线性地址空间供所有页目录及页表使用
;* * * * *
                   ORG 00400000h
L_Page_Tab0        dd  Page_Entries DUP (?)   ;OS 公共数据(IDT、GDT、TSS)
L_Page_Dir         dd  Page_Entries DUP (?)   ;页目录
                   ORG 00403000h
L_Page_Tab3        dd  Page_Entries DUP (?)   ;OS 专用(栈)
L_Page_Tab4        dd  Page_Entries DUP (?)   ;用户代码及数据
                   ORG 007FE000h
L_Page_Tab3FE      dd  Page_Entries DUP (?)   ;用户栈
L_Page_Tab3FF      dd  Page_Entries DUP (?)   ;OS 代码
;* * * * *
;* *OS 数据,4M 线性地址空间,例中未使用
;* * * * *
ORG 0800000h
;* * * * *
;* *OS 数据,4M 线性地址空间,OS 专用的数据和栈
;* * * * *
ORG        0C00000h     ;线性地址
Unshared_OS_Data        dd      ?
;不被共享的 OS 数据区域
temp     db      10 dup (?)
fptss    dw      ?
;* * * * *
```

```
;* *OS 栈
;* * * * *
ORG                 0FFFFFCh   ;栈顶
Top_OS_Stack        dd     ?
OS_Data_Seg         ENDS
;* * * * * * * * * * * * * * * * * *
;* * * * * * * * * * * * * * * * * *
;* *用户代码段
;* * * * * * * * * * * * * * * * * *
;* * * * * * * * * * * * * * * * * *
;此段安排在线性地址 1000000h 处
User_Code_Seg   SEGMENT   ER   USE32
  ;线性地址空间中,用户代码段的开始地址
  ;定义一个 1M 字节的分区
User_Code_Seg   ENDS
;* * * * * * * * * * * * * * * * * *
;* * * * * * * * * * * * * * * * * *
;* *用户数据段
;* * * * * * * * * * * * * * * * * *
;* * * * * * * * * * * * * * * * * *
;  此段安排在线性地址 1100000h 处
User_Data_Seg   SEGMENT   RW   USE32
  ;线性地址空间中,用户数据段的开始地址
  ;存储器的其余部分定义为用户的数据
  ORG 0FEEEFFFCh
  Top_User_Stack      dd ?
User_Data_Seg   ENDS
;* * * * * * * * * * * * * * * * * *
;* * * * * * * * * * * * * * * * * *
;* *32 位 OS 代码段
;* * * * * * * * * * * * * * * * * *
;* * * * * * * * * * * * * * * * * *
;32 位的 OS 代码安排在线性地址 0FFFF0000 处
OS_Code_Seg SEGMENT ER    USE32   ;Locate AT 0FFFF0000h
        Extrn     loader:NEAR
Default_Hand:
        ;<默认处理程序代码>
        IRET
DNA_Hand:
        ;<DNA 处理程序代码:协处理器异常的处理程序>
        IRET
Page_Fault_Hand:
        ;<页故障处理程序代码>
        IRET
Debug_Hand:
        ;<排错处理程序代码>
        IRET
Start32:
        ;* * * * *
        ;* *[5]将 OS 数据段的选择子装入 DS、ES,
        ;* *      使 OS_Data 可被寻址
        ;* * * * *
            MOV   AX,OS_Data_Sel
            MOV   DS, AX
            MOV   ES, AX
```

```
            ASSUME  DS: OS_Data_Seg
;*****
;**[6]建立LDT、LLDT
;*****
    MOV  EAX, offset LDT1_Table
    ;表项0,用户代码描述符
    ;Base=01000000h, Limit=000FFh, G=1,D=1,Type=A, DPL=3
    MOV  word ptr [EAX], 000FFh      ;Limit [15~0]=00FFh
    MOV  word ptr [EAX+2], 0000h     ;Base [15~0]=0
    MOV  byte ptr [EAX+4], 000h      ;Base [23~16]=0
    MOV  byte ptr [EAX+5], 0FAh      ;Type=A,DPL=3,P=1
    MOV  byte ptr [EAX+6], 0C0h      ;G=1,D=1,Limit [19~16]=0
    MOV  byte ptr [EAX+7], 01h       ;Base [31~24]=01h
    ADD  EAX,8
    ;表项1,用户数据描述符
    ;Base=01100000h, Limit=FEEEFh, G=1,D=1, Type=2,DPL=3
    MOV  word ptr [EAX], 0EEEFh      ;Limit [15~0]=0EEEFh
    MOV  word ptr [EAX+2], 0000h     ;Base [15~0]=0
    MOV  byte ptr [EAX+4], 010h      ;Base [23~16]=010h
    MOV  byte ptr [EAX+5], 0F2h      ;Type=2, DPL=3,P=1
    MOV  byte ptr [EAX+6], 0CFh      ;G=1,D=1,Limit [19~16]=0Fh
    MOV  byte ptr [EAX+7], 01h       ;Base [31~24]=01h
    MOV  AX,LDT1_Sel
    LLDT  AX
;*****
;**[7]建立IDT
;*****
    ;先将IDT各个表项填入,默认处理数据,
    ;再对需处理的中断的表项进行修改
    MOV  EAX,offset IDT_Table
    MOV  EBX, offset Default_Hand
    MOV  word ptr [EAX], BX          ;默认处理程序入口的低16位
    MOV  word ptr [EAX+2], OS_Code_Sel
    MOV  byte ptr [EAX+4], 00h       ;字计数为0
    MOV  byte ptr [EAX+5], 8Eh       ;类型为386中断门,特权级为0
    SHR  EBX,16
    MOV  word ptr [EAX+6], BX        ;默认处理程序入口的高16位
    ;用默认处理表项填满全部IDT表
    MOV  ESI, offset IDT_Table
    MOV  EDI, (offset IDT_Table+8)
    MOV  ECX, (IDT_Entries-1)*2
    REP  MOVSD
    ;修改非默认处理的中断表表项
    ;表项1: 排错
    MOV  EAX, (offset IDT_Table+(1*8))
    MOV  EBX, offset Debug_Hand
    MOV  word ptr [EAX],BX           ;处理程序入口低16位
    SHR  EBX, 16
    MOV  word ptr [EAX+6], BX        ;处理程序入口高16位
    ;表项7: 设备不可用
    MOV  EAX, (offset IDT_Table+(7*8))
    MOV  EBX, offset DNA_Hand
    MOV  word ptr [EAX],BX           ;处理程序入口低16位
    SHR  EBX, 16
    MOV  word ptr [EAX+6],BX         ;处理程序入口高16位
```

```
    ;表项 14:页故障
    MOV   EAX, (offset IDT_Table+(14*8))
    MOV   EBX, offset Page_Fault_Hand
    MOV   word ptr [EAX], BX                ;处理程序入口低 16 位
    SHR   EBX, 16
    MOV   word ptr [EAX+6], BX              ;处理程序入口高 16 位
    ;装入 IDTR
    MOV   word ptr pdescr, (IDT_Entries*8-1)
    MOV   dword ptr pdescr [2], offset IDT_Table
    LIDT  pword ptr pdescr
;*****
;**[8]建立页目录
;*****
    MOV EAX, Not_Present
    MOV   ECX, Page_Entries
    CLD
    MOV   EDI, offset Page_Dir
    REP STOSD      ;标记整个目录不存在
    ;表项 0: 共享的 OS 数据,用户不可访问。
    ;  安排在地址 0,所以线性地址=物理地址,而且装入的
    ;  GDT、IDT 及 TSS 在分页机制被启用后仍然有效。
    MOV   Page_Dir [0], P_Page_Tab0 OR 001h
    ;表项 1: 页目录指针,把页目录及页表映射到线性地址空间,
    ;  用户不可访问
    MOV   Page_Dir[4], P_Page_Dir OR 001h
    ;表项 3:OS 专用数据,用户不可访问
    MOV   Page_Dir[12], P_Page_Tab3   OR 001h
    ;表项 4: 用户代码及数据,用户可读写
    MOV   Page_Dir[16],P_Page_Tab4   OR 007h
    ;表项 1022: 用户栈,用户地址空间的最高端,用户可读写
    MOV   Page_Dir[4088], P_Page_Tab3FE   OR 007h
    ;表项 1023: 共享的 OS 代码,用户不可访问
    MOV   Page_Dir[4092], P_Page_Tab3FF   OR 001h
;******
;**[9]建立页表
;*****
    ;页表 0: OS 公共数据
    ;首先清除整个表
    MOV   EAX, NOT_Present
    MOV   ECX, Page_Entries
    CLD
    MOV   EDI, offset Page_Tab0
    REP   STOSD
    ;为 IDT、GDT 及 TSS 等分配存储页
    MOV   Page_Tab0 [0], P_OS_GData   OR 001h
    MOV   EAX, Not_Present
    MOV   ECX, Page_Entries
    CLD
    MOV   EDI, offset Page_Tab3
    REP   STOSD
    ;为专用数据和 OS 栈分配存储页
    MOV   Page_Tab3 [0000h], (P_OS_PData   OR   001h)
    MOV   Page_Tab3 [0FFCh], (P_OS_Stack   OR   001h)
    ;页表 4: 用户代码、数据和栈
    MOV   EAX, Not_Present
```

```
  MOV   ECX, Page_Entries
  CLD
  MOV   EDI, offset Page_Tab4
  REP   STOSD
  ;为用户代码及数据分配存储页
  MOV   Page_Tab4 [0000], (P_User_Code  OR  005h)
  MOV   Page_Tab4 [1024], (P_User_Data  OR  007h)
  ;页表 1022: 用户栈
  MOV   EAX, Not_Present
  MOV   ECX, Page_Entries
  CLD
  MOV   EDI, offset Page_Tab3FE
  REP   STOSD
  ;为用户栈分配存储页
  MOV   Page_Tab3FE[4092], (P_User_Stack  OR 007h)
  ;页表 1023: OS 代码。所有页面对用户可读,存储器最高 64K 地址
  ;的映射,使物理地址与线性地址相匹配
  MOV   EAX, 0FFFFF005h   ;最高页表项
  MOV   ECX,Page_Size
  LoopTop:
      SUB   ECX, 4
      MOV   Page_Tab3FF[ECX], EAX
      SUB   EAX, 1000h
      CMP   ECX, 4032
      JNZ   LoopTop
;* * * * *
;* *[10]启用分页机制
;* * * * *
  MOV   EAX, P_Page_Dir
  MOV   CR3, EAX
  MOV   EAX, CR0
  OR    EAX, PG
  MOV   CR0, EAX
  JMP   pflush
pflush:
;* * * * *
;* *[11]装入 OS 栈指针
;* * * * *
    MOV   dword ptr temp [0], offset Top_OS_stack
    MOV   word ptr temp [4], OS_Data_Sel
    LSS   ESP, pword ptr temp
;* * * * *
;* *[12]建立 TSS。只装入由 OS 访问的部分,其余部分或不需装入,
;* *      或在第一次向内层转移时装入
;* * * * *
    ; 0级栈偏移
    MOV dword ptr TSS1_Table [4], offset Top_OS_stack
    ; 0级栈选择子
    MOV   dword ptr TSS1_Table [8], OS_Data_Sel
    ; CR3
    MOV   EAX, CR3
    MOV   dword ptr TSS1_Table [1Ch], EAX
    ; LDT
    MOV   dword ptr TSS1_Table [60h], 0
    ; I/O 位图偏移
```

```
        MOV   word ptr TSS1_Table [66h], 8000h
        ;调试陷阱位
        MOV   word ptr TSS1_Table [64h], 0
        MOV   AX, TSS1_Sel
        LTR   AX
        初始化浮点状态字节(参见6.2.4)
        MOV   byte   ptr TSS1_Table[TSS_Size+FP_Save-1], 0
        MOV   fptss, 0
    ;* * * * *
    ;* *[13]装入用户任务
    ;* * * * *
        MOV   EAX, User_Code_Sel      ;用户代码段选择子
        MOV   ECX, User_Data_Sel      ;用户数据段选择子
        ;loader 从盘上取出第一个用户任务,并将装入到传递给该任务的
        ;选择子所指定的位置。Loader 将用户任务的入口地址偏移量
        ;置入 EAX 寄存器返回
        CALL loader

    ;* * * * *
    ;* *[14]为调用用户任务做准备:用户的 SS:ESP、flags、CS:EIP
    ;* *     压栈,使 IRET 调用用户任务
    ;* * * * *
        MOV   ECX, User  Data_Sel     ;用户栈
        PUSH  ECX
              ;用户栈指针
        MOV   ECX, Offset Top  User  Stack
        PUSH  ECX
        PUSHF
              ;启用中断
        OR SS: dword ptr [ESP], Int_Flag
        MOV   ECX, User_Code  Sel   ;用户任务的代码段
        PUSH  ECX
        PUSH  EAX                   ;用户任务开始地址的偏移
    ;* * * * *
    ;* *[15]调用用户任务
    ;* * * * *
        ;用 IRET 转入用户程序的执行
        IRET

OS_Code_Seg_ENDS

;16 位的引导程序代码,用汇编语言编写。此段程序安排在存储器的最高
;4K 地址,系统复位时形成的地址0FFFFF000即指向此段程序
BootRom16  SEGMENT EO USE 16   ;段地址0FFFF0000
           ORG     0F000h      ;偏移0F000
Start 16:
    ;* * * * *
    ;* *[2] 其它:CLI, DS, 空 IDT
    ;* * * * *
        CLI              ;关中断
        MOV   AX, 0
        MOV   DS, AX  ;实地址方式下初始化数据段
        ASSUME   DS: OS  Data  Seg

        ;置 IDT 为空,使任何中断都引起系统停机
```

```
    MOV   word ptr pdescr [0], 0
    MOV   dword ptr pdescr [2], 0
    LIDT  pword ptr pdescr
;* * * * *
;* *[3]建立 GDT
;* * * * *
    MOV   EAX, offset GDT_Table
    ;表项0: 空描述符
    MOV   dword ptr [EAX], 0
    MOV   dword ptr [EAX+4], 0
    ADD   EAX, 8

    ;表项1: OS 代码描述符
    ;Base=FFFF0000h, Limit=0000Eh,G=1,D=1,Type=A,DPL=0
    MOV   word ptr [EAX], 0000Eh        ;Limit [15..0]
    MOV   word ptr [EAX+2], 0000h       ;Base [15..0]
    MOV   byte ptr [EAX+4], 0FFh        ;Base [23..16]
    MOV   byte ptr [EAX+5], 09Ah        ;Type—代码描述符
                                        ;代码描述符 DPL=0
    MOV   byte ptr [EAX+6], 0C0h        ;G,D,Limit [19..16]
    MOV   byte ptr [EAX+7], 0FFh        ;Base [31..24]
    ADD   EAX,8

    ;表项2: OS 数据描述符
    ;Base=00000000h, Limit=0FFFFFh, G=1,D=1, Type=9,DPL=0
    ;把整个数据空间映射到 OS 地址空间
    MOV   word ptr [EAX], 0FFFFh        ;Limit [15..0]
    MOV   word ptr [EAX+2], 0000h       ;Base [15..0]
    MOV   byte ptr [EAX+4], 00h         ;Base [23..16]
    MOV   byte ptr [EAX+5], 092h        ;Type=数据描述符
                                        ;DPL=0
    MOV   byte ptr [EAX+6], 0CFh        ;G,D,Limit [19..16]
    MOV   byte ptr [EAX+7], 00h         ;Base [31..24]
    ADD   EAX,8

    ;表项3: LDT
    ;Base=(offset LDT1_Table), Limit=(LDT_Entries * 8),
    ;G=0,Type=LDT,DPL=00
    MOV   word ptr [EAX], (LDT_Entries * 8-1)     ;Limit [15..0]
    MOV   EBX, offset LDT1_Table
    MOV   word ptr [EAX+2], BX                    ;Base [15..0]
    SHR   EBX, 16
    MOV   byte ptr [EAX+4], BL                    ;Base [23..16]
    MOV   byte ptr [EAX+5], 082h                  ;Type=LDT
    MOV   byte ptr [EAX+6], 000h                  ;Limit [19..16]
    MOV   byte ptr [EAX+7], BH                    ;Base [31..24]
    ADD EAX, 8

    ;表项4: 任务的 TSS
    ;Base =(offset TSS1_Table), Limit=(TSS_Size+FP_Save),
    ;G=0, Type=可用 386 TSS, DPL=00
    MOV   word ptr [EAX], (TSS_Size+FP_Save-1)   ;Limit [15..0]
    MOV   EBX offset TSS1_Table
    MOV   word ptr [EAX+2], BX                    ;Base [15..0]
    SHR   EBX, 16
```

```
    MOV   byte ptr [EAX+4], BL                ;Base [23..16]
    MOV   byte ptr [EAX+5], 089h              ;Type=可用 386 TSS
    MOV   byte ptr [EAX+6], 000h              ;Limit [19..16]
    MOV   byte ptr [EAX+7], BH                ;Base [31..24]

    ;建立 GDT 伪描述符,并将其装入 GDTR
    MOV   word ptr pdescr[0], (GDT_Entries * 8-1)
    MOV   dword ptr pdescr [2], offset GDT_Table
    LGDT pword ptr pdescr

;* * * * *
;* *[4]进入保护方式
;* * * * *
    SMSW  AX
    OR    AX, PE
    LMSW  AX      ;设置保护启用位
    JMP   Flush
Flush:
    JMP   far ptr Start 32   ;通过 GDT 跳转到32位的 OS 代码

;* * * * *
;* *[1]冷启动代码
;* * * * *
    ORG   0FFF0h
    JMP   Start 16
BootRom16 ENDS
END
```

6.2.4 协处理器异常处理程序实例

本节的实例是协处理器设备不可用(DNA)异常处理程序。此例程是初始化实例的扩展。

6.2.4.1 概述

本例通过协处理不可用处理程序,说明故障处理程序的细节。下列原因之一都可产生设备不可用异常(矢量号7)。

(1) 执行浮点指令时,控制寄存器 CR0中的 EM 位为1,这时该指令不由硬件(协处理器)执行,而用软件模拟。

(2) 执行浮点指令时,控制寄存器 CR0中的 TS 位为1,这时表示快速任务切换方式。

(3) 执行 WAIT 指令时,控制寄存器 CR0中的 TS 及 MP 位皆为1。

在本例中,我们处理第二种情况——快速任务切换方式。如果系统要模拟协处理器,则只需简单地执行一个对例程 DNA_Emulate 的调用即可,该例程负责模拟协处理器的操作并完成指令要求进行的计算。

快速任务切换方式是一种优化的方式。采用这种方式,可以使协处理器的状态保存到下一次某任务对协处理器访问的时候为止。如果系统中多数任务不使用协处理器,用这种方式就会产生明显的好处。因为保存和重新装入协处理器的状态是十分费时的,而用这种方式正好是为了减少协处理器状态的保存及重新装入操作。

当 DNA 异常发生时,系统将调用协处理器设备不可用处理程序,该程序可以直接地插入实例1中。插入的位置在32位码段最前面的异常处理程序入口处,该处有默认中断处理、DNA 处理、页故障处理及排错处理等。本例的代码就可放在 DNA_Hand 标号之下,并替换

;<DNA 处理程序代码：协处理器异常处理程序>

的说明即可。

发生协处理器异常时，如果有另一个任务一直在使用协处理器，则协处理器的状态要保存到发生异常时的前一次使用协处理的任务的 TSS 中，然后将当前任务的协处理器状态装入协处理器中。为了能判定本次要使用协处理器的任务是否与前次使用协处理器的任务相同，在系统中需设置一个变量，用以保存最后一次使用协处理器任务的 TSS 的选择子。因此，当一个任务连续使用协处理器时，都不需保存及恢复协处理器状态，从而省去很多操作时间。只有当要使用协处理器的任务与前次的任务不一样时，才需保存状态到原任务，并恢复新任务的协处理器状态，同时，还更新保存 TSS 选择子的变量。

6.2.4.2 异常处理程序的详细说明

在初始化实例中，所有处理程序都是通过中断门转入的。在中断响应时，为了防止在处理程序中由于嵌套的中断可能产生的软件错误而关闭中断。协处理器异常处理程序首先要做的是状态的保存及恢复(如果必需)，要花费较多时间，因此，进入协处理器异常处理程序，在做其它事情之前，首先是开放并启用中断，并且只在绝对需要禁止中断时才再关闭中断。

1. 代码 1——模拟处理

代码 1检查 CR0的 EM 位，如果 EM 位为1，表明浮点指令由软件模拟而不是由硬件执行。在调用浮点指令的模拟程序(模拟处理不属本例范围，故略去)完成之后，立即返回。如果 EM 为0，则跳转到代码2。

2. 代码 2——确立当前任务 TSS 的基地址及 TSS 的选择子

为了确定是否某任务连续使用协处理器，需知道当前任务 TSS 的选择子，以便与保存前次使用协处理器任务的选择子的变量(在此程序中为 fptss)进行比较。为了保存状态，需要知道当前任务 TSS 的基地址，因为协处理器状态保存区在 TSS 最前面的部分，知道了 TSS 的基地址也就得到了保存区的地址。这两项任务由代码 2完成。其操作如下：

- 用 STR 指令得到当前任务的选择子，保存在 SI；
- 从 GDT 中获取 TSS 的描述符；
- 从描述符中分离出基地址，保存在 ECX。

3. 代码 3——保存原任务状态

本段程序做三件事情：

(1) 判断是否首次使用协处理器。如果是首次使用，则对协处理器做初始化处理。判断的标志是变量 fptss＝0。给 fptss 赋值为0是由初始化程序完成的。当任有一任务使用协处理器后，都有 fptss≠0。

(2) 判断本次使用协处理的任务是否与前次的任务相同。判断条件是：

fptss＝S1

是否成立。如条件成立，则立即返回；否则执行 3。

(3) 保存协处理状态，即将当前任务的协处理状态送入 TSS 前部协处理器状态保存区中保存。

4. 代码 4——装入当前任务的协处理器状态

装入当前任务的协处理器状态时，必须禁止程序被中断。这是因为这段程序在装入协处理器状态后还要更新变量 fptss 的值。如果不禁用中断，假定原任务的协处理器状态已经保存，新任务的协处理状态已经装入，但 fptss 尚未更新；这时协处理器的状态与 fptss 的值是不协

调一致的，前者是新任务状态，后者是老任务的选择子。如果恰好在这时有一中断请求并被响应，则本处理程序被再次调用时，将破坏老任务的协处理器状态。

然而，装入新任务的协处理器状态时禁止中断，在某些情况下是不能允许的，这是因为恢复状态要费很长的时间。要想在这段程序中既允许中断又不致发生破坏老任务的状态，当然也可用软件的办法来解决。这时需设置两个信号，而且异常处理程序也会变得比较复杂。我们这里仍使用这一不太好的设计，而把上述问题留给有兴趣的读者去解决。

协处理状态保存区的最高字节是一个标志，用以指示是否在 TSS 的协处理器保存区保存过状态。如果该字节为0，则表示该任务还未使用过协处理器，此次是第一次使用，因而不存在恢复状态的问题而简单地用 FNINIT 指令对协处理器进行初始化。如果该字节为1，则表明该任务有状态要从保存区装入协处理器。每个任务的 TSS 协处理器保存区最高字节的初始化，由上一节的初始化程序完成(见初始化实例，代码12的倒数第二行)。

这一字节由代码 4中标号为 DNA_Done 的第二个 MOV 指令使其置1。而 DNA_Done 的第一个 MOV 指令则把 TSS 的选择子保存在 fptss 变量中。

请读者注意区别 TSS 中保存区最高字节等于零所起作用与变量 fptss＝0所起作用的区别。前者是指某一任务尚未使用过协处理器，后者是指没有任何一个任务使用过协处理器。

下面列出协处理器异常处理程序的程序清单。

程序 6.13

```
;*********************************
;*设备不可用(DNA)处理程序
;*假定条件：
;*1. fptss 是最后一个使用协处理器的任务的 TSS 的选择子，如没有任务使用过
;*   协处理器，则 fptss 为0；
;*2. 在紧靠 TSS 的上端，有一个浮点保存区域；
;*3. TSS 的浮点保存区域为108字节＋4字节。最高的4个字节存放标志，
;*   用以指示保存区域中是否保存有协处理器的状态：
;*   1表示保存有协处理器状态，0表示没有保存。
;*
;*********************************

DNA_Hand:
    STI
    ;*****
    ;*[1]确定故障是否由模拟引起
    ;*****
        ;控制寄存器 CR0的安排；
        ;  31                    4  3  2  1  0
        ; |P                     E  T  E  M  P|
        ; |G                     T  S  M  P  E|
        MOV   EAX, CR0
        BT    EAX, 2
        JNC   DNA_Task_EX
        CALL  DNA_Emulate
        IRET

    ;*****
    ;**[2]确定当前任务 TSS 的基地址
    ;*****
```

```
DNA_Task_EX
    XOR   ESI, ESI
    STR   SI            ;ESI 存储当前任务的 TSS 选择子
    MOV   ECX, [ESI+offset GDT_Table+0]
    MOV   EDX, [ESI+offset GDT_Table+4]
    SHRD  ECX, EDX, 16
    AND   ECX, 000FFFFFFh
    AND   EDX, 0FF000000h
    OR    ECX, EDX      ;ECX 存放当前任务的基地址

;* * * * *
;*[3] 故障由任务切换引起。如果老任务的状态需保存,则将协处理器
;*     的状态保存至任务状态段保存区域中
;* * * * *
    XOR   EDI, EDI
    MOV   DI, fptss
    CMP   DI, 0
    JE    DNA_Restor      ;在此之前没有任何任务使用过协处理器
    CMP   DI, SI          ;如果上次使用协处理器的任务即当前正运行的任务,
                          ;则没有必要保存和恢复协处理器环境
    JNE   DNA_Save
    IRET
DNA_Save:
    ;确定上次使用协处理器的任务的 TSS 基地址
    MOV   EAX, [EDI+offset GDT_Table+0]
    MOV   EDX, [EDI+offset GDT_Table+4]
    SHRD  EAX, EDX, 16
    AND   EAX, 000FFFFFFh
    AND   EDX, 0FF000000h
    OR    EAX, EDX        ;EAX 已指向先前任务的 TSS 的基地址
    FNSAVE [EAX+TSS_Size] ;保存老任务的协处理器状态
;* * * * *
;*[4] 恢复新任务原来的协处理器状态
;*     此过程必须关中断运行
;* * * * *
DNA_Restor:
    CLI
    ;检查标志,如真,则恢复上一次的协处理器状态
    CMP   byte ptr [ECX+(TSS_Size+FP_Save-1)], 1
    JNE   DNA_Init
    FRSTOR [ECX+TSS_Size]
    JMP   DNA_Done
DNA_Init:
    FNINIT
DNA_Done:
    MOV fptss, SI     ;将任务的 TSS 选择子存入 fptss
    ;设置标志,表示协处理器已被本任务使用,清除 TS,置位 IF
    MOV   byte ptr [ECX+(TSS_SIZE+FP_Save-1)], 1
    CLTS
    STI
    IRET
```

第7章　80386的向下兼容性

本书前6章描述了80386作为一个32位处理器使用的情况。80386是8086、8088、80186、80188及80286系列的进一步发展和提高，本章则着重说明80386与前述各型处理器的兼容特性。

如第一章所述，8086、8088、80186、80188具有共同的结构。80286在这一共同结构的基础上，增加了存储管理及保护方式。因为使用了保护机制，使80286具有两种操作方式：一种是不使用保护特性的方式，我们称这种方式为实地址方式，在这种方式下使用80286，其行为与8086完全一样；另一种是使用保护特性的方式，即所谓的保护方式，在这种方式下使用80286，其行为与8086有着相当重要的区别。

由于8086、8088、80186、80188及实地址方式下的80286，对编程人员来说，这些处理器看起来是很相似的，在本章我们就以8086为代表来描述这些处理器。其间的细微差别则示于附录B。80286使用于保护方式，导致非常不同的编程模式，从而需分别地处理。

本章首先说明16位程序可用的基本机器寄存器及寻址方式；然后再描述8086的程序如何能在80386上运行。对于运行8086程序，80386有两种处理器工作方式可供使用：即实地址方式及虚拟8086方式。80386的实地址方式，与80286一样，是不使用保护机制的一种执行方式。虚拟8086方式则是80386特有的一种新的工作方式。在此方式下，80386既使用保护机制，又能执行8086的程序。本章的最后讨论80286程序的执行。

7.1　16位寄存器及寻址方式

16位的程序使用第2章所定义的寄存器的低16位。图7.1示出了这些寄存器。

(1) 8个通用寄存器AX、CX、DX、BX、SP、BP、SI及DI，是第2章所述相应通用寄存器的低16位。这8个寄存器中前4个寄存器AX、CX、DX及BX的前后8位还可分别以8位的寄存器进行访问。

(2) 两个16位的处理器控制寄存器IP及FLAGS分别是第2章描述的EIP及EFLAGS的低16位。

(3) 6个16位段寄存器(没有32位的段寄存器形式)中，FS、GS是80386新增加的，所以8086及80286上执行的16位的程序不能使用FS及GS。

存储器寻址类似于第2章讨论的存储器寻址机制。每个存储器访问都规定一个段地址及一个偏移量。因为16位的程序使用16位的偏移量，所以，最大的段大小为64K字节。表7.1列出了为产生16位程序的段偏移量可使用的寻址方式。16位的偏移量，通过把16位基寄存器、16位变址寄存器及8位或16位的位移相加而形成。这里的加法使用16位的算法，即结果以2^{16}为模，因此，位15的进位被忽略。附录H描述16位寻址方式的编码。

16位的寻址方式不象32位的寻址方式那样更为通用。在16位的寻址方式中，变址比例是不

可用的，只有 BX 及 BP 寄存器可用来作为基寄存器，SI 及 DI 寄存器可用来作为变址寄存器。当 BP 作为基址寄存器使用时，默认段寄存器是 SS，否则，默认的段寄存器是 DS。

16位寄存器的使用由指令的操作数长度控制，16位寻址方式的使用由指令的地址长度控制。第4章已说明代码段描述符中的 D 位，为操作数及地址设置默认的长度。通过把16位码段描述符中的 D 位清除为0，可为16位80286程序设置16位的默认长度。在实地址方式或虚拟8086方式下执行的程序，默认的长度为16位。附录 H 说明，对于任何指令，可使用一个指令前缀以取消操作数或地址的默认长度。

表 7.1　16位地址方式

基地址	+	变址值	+	偏移
{ BX, BP* }	+	{ SI, DI }		{ 8位值, 16位值 }

* 如 BP 是基寄存器，则 SS 为默认段寄存器

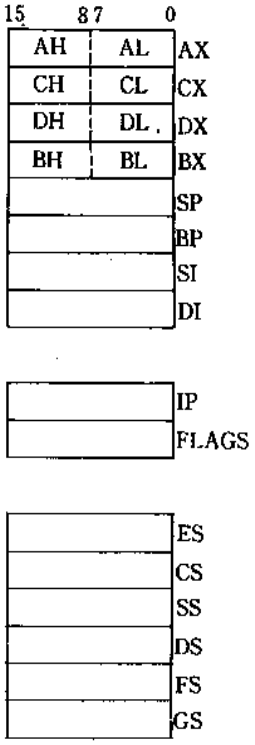


图 7.1　16位寄存器

80387使用的浮点寄存器与8087及80287相同。为8086/8087或80286/80287所写的任何程序都可使用第2章描述的浮点寄存器组。

7.2　执行8086程序

在80386上执行8086(及8088、80186、80188和实地址方式的80286)程序有两种执行方式：实地址方式及虚拟8086方式。80386在这两种方式下执行时，除附录 B 所列的若干细微差别之外，具有与8086同样的编程模型。

80386的实地址方式与80286的实地址方式相同。80386在实地址方式下执行程序时，其行为与8086相同，因而可直接地支持8086程序的执行。要使80386处于实地址方式，可通过将寄存器 CR0中的 PE 位清为0而禁用存储管理、保护及任务机制来实现。由于实地址方式中禁用保护机制，使80386好象在特权级0执行。80386工作于实地址方式时，中断与异常使用不同于第5章描述的机制进行处理。这种中断机制在本章实地址方式的中断处理一节描述。

当由硬件复位进行初始化时，80386开始以实地址方式执行，这就使得在启用保护方式之前，可使用自举代码。初始化系统中的各种表格见第6章。

虚拟8086方式是80386的一种新的工作方式，该方式支持在存储管理、保护及任务机制的环境中执行8086程序。称为虚拟机监控程序的操作系统软件，可创建一个在虚拟8086方式下执行8086程序的任务，执行这一任务会给人以这样的假象：好象该任务工作的环境就是一个8086环境。80386的任务机制，有可能使80386以虚拟8086方式执行一个任务的8086程序，以16位的保护方式执行另一个任务的80286程序，以及以32位的保护方式执行第三个任务的80386程序。

虚拟8086方式下执行的程序运行于特权级3，即运行于最外层的特权级。分页机制可用来

为虚拟8086方式下执行的程序分配及保护存储器资源。当程序在虚拟8086方式下执行时接收到一个中断或产生异常，则中断或异常通过切换到第4章所描述的保护方式进行处理，然后再用第4章描述的机制对中断或异常进行进一步的处理。从虚拟8086方式切换到保护方式将在进入及离开虚拟8086方式一节做详细的描述。

虚拟机监控程序，通过建立在虚拟8086方式下执行的几个任务，使几个8086程序可以同时执行。由于虚拟机监控程序软件及虚拟8086硬件设施共同提供的环境，使每一个这样的8086程序在80386上执行时，好象完全在8086上执行一样。在系统中的其它任务，可以执行16位保护方式的80286程序，也可执行32位保护方式的80386程序，而普通的任务切换，用于挂起一个任务并恢复另一个任务。这就使得80386运行各种各样任务的集合，某些任务工作于8086方式，另一些任务工作于保护方式，并在这些任务之间不断进行切换成为可能。这是一种功能很强的机制，用这种机制把大量的基于16位的 PC 程序，与更为复杂的最初为小型计算机及大型计算所写的32位的应用程序组合在一起。虚拟8086方式与80386的任务管理设施在一起，提供了关键的硬件设施支持这种功能强大的软件模型。

与保护方式及虚拟8086方式支持的功能强大的多任务模型相对比，实地址方式没有任务支持，因而只支持一种8086程序。这种8086程序以没有保护、没有存储管理、没有多任务支持的方式，对80386进行直接的访问。

下面，我们先讨论实地址方式及虚拟8086方式共同的机制，接着再讨论虚拟8086方式不同于实地址方式的设施。

7.2.1 段及寻址

8086的编程使用'段'的概念，支持一个二维存储器寻址模式，这种寻址模式与第2章介绍的模式相同，8086程序使用与保护方式同样的分为两部分的存储器地址，该地址包含一个段部分和一个偏移量部分。偏移量部分用表7.1中给出的16位地址模式之一产生。地址的段部分从段寄存器获取，该寄存器事先已用所需要的段地址装入。虽然地址的段部分象保护方式那样，取自段寄存器，但是在8086程序中对地址的段部分的处理与保护方式进行的处理是完全不同的。

8086存储器寻址，不使用段描述符表定义段的基地址、界限及属性。8086的段也有这些相应内容，但这些内容不需进行定义，而具有固定的值及属性。8086段的基地址为段寄存器内容乘以16，界限固定为64K 字节大小，属性固定为允许读、写或执行。实地址方式的段隐含 DPL＝0，这是因为实地址方式下执行程序时，犹如程序是在特权级0执行的一样。虚拟8086方式的段隐含 DPL＝3，因为虚拟8086方式下，程序在特权级3执行。由 CS 寻址的段隐含 D 位为0，所以8086程序的默认操作数及地址长度为16；由 SS 寻址的段也隐含有 D 位为0，所以栈访问操作使用16位的 ESP 寄存器。

段寄存器的值用于规定线性地址空间中段的基地址，其数值是段寄存器的值乘以16。16位的段寄存器中能容纳的最大值为$2^{16}-1$，所以最大的段基地址为$2^{20}-16$。这表明一个段可以在线性地址空间中第一个1M 字节范围内任一16字节边界的地址开始，并包含最多为64K 字节的范围。

段基地址与指令寻址方式规定的偏移量相加，得到指定存储器位置的线性地址。在图7.2中，计算线性地址使用32位的结构。因此，80386中段基地址与偏移量相加时，结果线性地址可以超过1M 字节，这种情况在段基地址大于或等于1M－64K＋16的任何值时，都有可能发生。

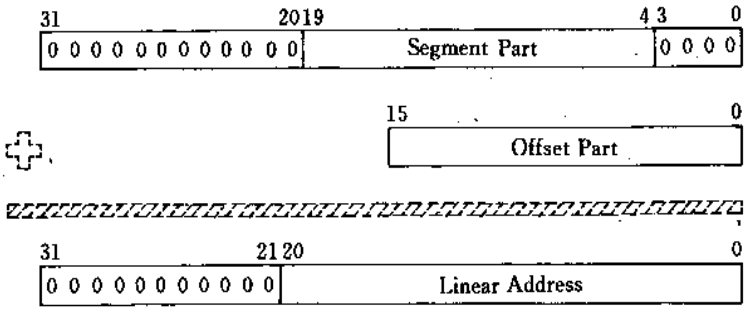


图7.2 实地址及虚拟8086线性地址格式

因为段寄存器的值被简单地视为一个16位的数值，在装入段寄存器时，不可能发生异常，所以，装入段寄存器不进行任何检查，并可使用任何16位的值。在保护方式下，装入段寄存器的值为0时，表明一个空选择子装入段寄存器，从而使该段寄存器不能用来进行寻址，否则会产生异常。但在实地址方式下，装入段寄存器的值为0具有实际的意义，表明段的基地址为线性地址0，这时，该段即为线性地址的前64K 范围。以下用两个例子说明实地址方式下的线性地址的计算。

例1 段寄存器的值为5678h，指令寻址方式给出偏移量为9，则结果线性地址为：

```
      56780h   段寄存器的值乘以16
+        9h    由寻址方式给出的偏移量
      56789h   结果线性地址
```

例2 段寄存器的值为0FFFFh，指令寻址方式给出偏移量为0FFFFh，则结果线性地址为：

```
      FFFF0h   段寄存器的值乘以16
+     FFFFh    寻址方式给出的偏移量
     10FFEFh   结果线性地址
```

例2中的结果线性地址，对8086来说，超过了存储器操作数的寻址范围，但在80386中，无论是实地址方式还是虚拟8086方式，该结果线性地址都是有效的。在实地址方式中，该结果地址即为物理地址。在虚拟8086方式中，如启用了分页机制，则该地址可映射为4G 空间内的任何物理地址；分页机制被禁用时，该地址即为物理地址。所以在实地址方式或虚拟8086方式中，分页机制被禁用时，可以产生的最大的物理地址为10FFEFh，即略大于1M。

7.2.1.1 *访问存储器时的异常*

80386在实地址或虚拟8086方式下执行时，如果对存储器进行访问，按寻址方式计算得到的偏移量大于段的界限64K 时，将报告栈段异常或通用保护异常。很明显，使用32位寻址方式产生的偏移量会超过64K。8086程序虽然只使用16位的寻址方式，但当偏移量的值接近64K，而被访操作数多于一个字节时，仍可能发生被访操作数超过段界限的情形。例如，一个四字节的操作数，第一个字节存放于相对于某一段值时，偏移量为0FFFEh 的位置，由于后两个字节超出64K 偏移量范围，对这一操作数的访问将产生异常。然而在8086中遇到这种情况时并不产生

异常，而是使操作地址回到偏移量为0的地址。

进行存储器访问时，访问地址超过64K界限时产生异常，由访问时使用的段寄存器确定异常的性质。如果使用的段寄存器是SS，则产生栈段异常；如果使用的是SS以外的其它段寄存器，则产生通用保护异常。

当分页机制被启用时，在虚拟8086方式下执行的程序，如果产生一个无效线性地址，或违反保护模式，也将报告一个分页异常(详见第4章)。

实地址方式或虚拟8086方式下执行程序，还可能产生的异常有除法出错异常及第5章所描述的大多数异常情形。但是装入段寄存器则不会发生异常，因为在实地址方式或虚拟8086方式下段寄存器的内容不表示一个描述符的选择子，而仅仅是一个16位的数值。

7.2.1.2 实地址及虚拟8086方式中的存储器访问

在实地址方式及虚拟8086方式下执行装入段寄存器的指令，与在保护方式下执行这些指令是不相同的。这时，我们不使用第4章为这些指令所描述的语法，而用第3章中的说明来描述装入段寄存器指令的动作。

当处理器在实地址方式或虚拟8086方式下执行时，第4章描述的例程AccessVirtual()，改变为例程Access 8086 Virtual()。在这两种方式下，不访问描述符投影寄存器，段的基地址通过把段选择子寄存器中的值乘以16而得到。对段的偏移量及段的长度要进行检查，以保证对存储器的访问是在64K的段的界限之内。在这些方式下，所有的段都是可读、可写及可执行的。因本程序相对比较简单，故未做流程图分析。下面列出Access 8086 Virtual()的程序清单：

```
/* 访问由段寄存器SReg寻址的段中的数据 */
Access 8086 Virtual (SReg, Offset, Length, RW, IntNumber, Data)
Segment Register SReg;
int             Offset, Length,   /* 段内偏移及数据长度 */
                RW,               /* 访问方式：0为读，1为写 */
                IntNumber,        /* 访问SS寄存器时为$SS, */
                                  /* 访问其它寄存器时为$GP */
                *Data;            /* 指向读或写数据的指针 */
{
   /* 被访数据必须在64K的段的范围之内，否则发生异常 */
   if (Offset+Length>10000h)
      Segment Exception (IntNumber, 0);
   /* 如检查通过，则访问线性地址 */
   Access Linear (SReg, Selector*16+Offset, Length, CPL, RW, Data)
}
```

7.2.1.3 模拟8086的20位地址运算

8086支持20位物理地址空间，但80386在计算线性地址时，用的是32位的运算。如果不对由第20位进位到第21位的进位进行适当的处理，则8086的程序在80386上将不能正确地运行。如果处理器在实地址方式或虚拟8086方式下执行，且分页机制被禁用，则线性地址直接地作为物理地址，由80386用来驱动外部硬件。这时，要求外部硬件模拟8086的20位地址空间。最简单的硬件解决办法是只使用低20位的地址线，忽略高12位地址线的作用，使其只支持1M物理存储器的寻址范围；另一种解决办法，是当执行8086程序时，用外部逻辑强制使地址线20(即地址的第21位)为0。

在虚拟8086方式中，可用分页机制来模拟20位的循环线性地址。这时，分页机制把8086地址运算中结果地址为100000h至10FFEFh范围的线性地址所映射的物理地址，映射到地址为0—0FFEFh范围的线性地址所映射的相同的物理地址。分页机制还可更进一步，把所有超过

1M 的线性地址所映射的物理地址，都映射为1M 以内地址所映射的相同的物理地址，即在实际的线性——物理地址映射之前，先把线性地址进行一次模2^{20}的处理，经过这样的处理之后，线性地址第21位以上的所有各位，在线性——物理映射中实际上已不再起作用。这与硬件只使用线性地址的20个低位所得到的是完全相同的结果。用分页机制模拟20位寻址的这种用法可用图7.3加以说明。在图中，我们只图示出8086方式下地址运算中实际上可能出现的最大线性地址的情形，即当段地址为最大值($2^{16}-1$)，偏移量为最大值($2^{16}-1$)，所产生的最大线性地址

$$(2^{16}-1)*16+2^{16}-1=10FFEFh$$

时的情形。实际的映射关系是：

线性地址	物理地址
0—00FFFFh	0—00FFFFh
010000h—0FFFFFh	010000h—0FFFFFh
100000h—10FFFFh	0—00FFFFh

7.2.1.4 无效指令

实地址方式及虚拟8086方式不支持表7.2中列出的指令。如在这两种方式下使用表7.2中的任何一条指令，将产生无效操作码异常。

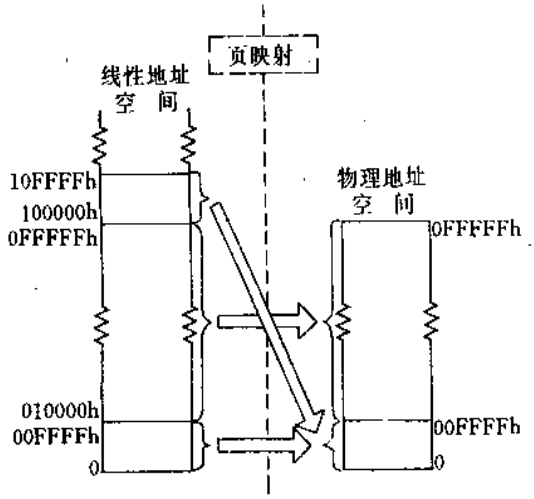


图 7.3 页机制模拟20位地址的寻址

表 7.2 实地址方式和虚拟8086方式中的无效指令

ARPL
LAR
LLDT
LSL
LTR
SLDT
STR
VERR
VERW

7.2.2 8086中 FLDENV、FSTENV 及 FNSTENV 指令的格式

第3章描述的 FLDENV、FSTENV 及 FNSTENV 指令为存储器装入或存储控制字、状态字、特征字及出错指针寄存器。第2章已描述过这些指令在保护方式下执行时的存储器格式，图7.4示出这些指令在实地址方式或虚拟8086方式下执行，且操作数大小为16时使用的格式。在8086上执行的程序为这些指令规定16位的操作数长度。在80386上可以使用32位的操作数，但这样使用没有用处，故不予推荐。

图7.4的控制字、状态字、特征字寄存器的值与第2章给出的格式相同，但出错指针以不同的格式存储。在图7.4中，为出错指针给出的是20位的线性地址，而不是以两个部分的地址格式

给出。

FRSTOR、FSAVE 及 FNSAVE 指令装入和存储浮点累加器以及 FLDENV、FSTENV 及 FNSTENV 指令处理的寄存器。如果这些指令由8086程序使用16位的操作数执行，则以图7.4给出的格式存储控制字、状态字、特征字及出错指针寄存器，然后，ST(0)装入或存储在偏移量为14处，ST(1)装入或存储在偏移量24处，以此类推，直到 ST(7)，存储在偏移量为84处。ST(0)～ST(7)中的每一个，在所有方式中都以80位的临时实地址格式存储。

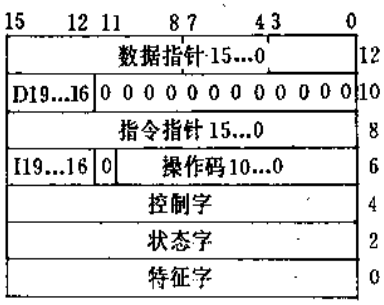


图 7.4 FLDENV、FSTENV 和 FNSTENV 指令的8086格式

7.2.3 虚拟8086方式的若干问题

本节专门描述虚拟8086方式的特性。判断处理器是否是在虚拟8086方式下执行，使用的标志是 EFLAGS寄存器中 VM 位的状态。当 VM＝1时，即表明处理器执行于虚拟8086方式；VM＝0，表明处理器没有处于虚拟8086方式。所以，要进入虚拟8086方式，即应先将 EFLAGS 中的 VM 位置1。相反要离开虚拟8086方式，就要把 VM 的值置为0。下面，首先描述对 I/O 特权级敏感的指令的一些改变，然后，再描述在保护方式和虚拟8086方式下互相切换使用的方法。

7.2.3.1 虚拟8086方式中的 IOPL

当80386工作于虚拟8086方式时，访问 EFLAGS 寄存器的指令对 IO 特权级敏感。在80386的所有工作方式下(实地址方式、虚拟8086方式及保护方式)，CLI 和 STI 指令对 IOPL 都敏感。在虚拟8086方式下，且 IOPL 小于3(虚拟8086方式的程序的特权级)时执行 PVSHF、POPF、INTn 及 IRET 指令，将产生通用保护异常(出错码设置为0)。这就保证了所有可以改变或存储 EPLAGS 中的 IF 位的指令对 IOPL 敏感，使操作系统软件支持一个“虚拟 EFLAGS”寄存器。

在虚拟8086方式下执行 IN、OUT、INS 或 OUTS 等 I/O 指令时，这些指令在测试 I/O 位图之前不对 IOPL 进行检查。相反，这些 I/O 指令总是访问存储在当前 TSS 中的 I/O 许可位图，并当所有许可位都为0时，允许此次 I/O 访问；而当许可位中至少有一个为1，或超出 TSS 界限时，产生通用保护异常。也就是说，在虚拟8086方式下执行 I/O 指令时，忽略 IOPL 字段的作用。

由于虚拟8086方式下执行 I/O 指令时不检查 IOPL，而 PUSHF、POPF、INTn 及 IRET 指令检查 IOPL，所以，虚拟8086方式相比于保护方式对 IOPL 字段有完全不同的解释。虚拟8086方式下，IOPL 用于保护 EFLAGS 寄存器中的 IF 位，而对 I/O 指令不起作用；在保护方式下，IOPL 影响 I/O 指令及某些可以改变 IF 位的指令。

1. IOPL 小于3时的虚拟8086方式

操作系统软件可以把 IOPL 设置为小于3的值，使得每当要执行一条可能修改或存储 IF 位的指令时，使虚拟8086方式的程序产生异常。通用保护异常处理程序，则支持一个与80386中实际的 EFLAGS 寄存器不相同的、虚拟的 EFLAGS 寄存器。此虚拟寄存器中的 IF 位在虚拟8086方式程序的控制之下进行置位；而实际的 EFLAGS 寄存器中的 IF 位，当执行虚拟8086程序且中断被启用时，即被置位。操作系统软件可以用虚拟的 EFLAGS 寄存器的 IF 位来确定：

向虚拟8086任务报告中断(虚拟 IF 位为1);或使中断保持悬挂(虚拟 IF 位为0)。PUSHF 指令产生一个异常,使异常处理程序通过把虚拟的 EFLAGS 寄存器压栈,来模拟实际的EFLAGS 寄存器。

2. IOPL 等于3时的虚拟8086方式

用户可以在虚拟8086方式下,且 IOPL=3时执行程序。这时,执行访问 IF 位的指令不会产生异常,而使虚拟8086程序绕过 INTR 中断屏蔽,从而使要执行大量 IOPL 敏感指令的8086程序可以全速执行。执行这样的程序时,如 IOPL 小于3,则执行过程中要产生很多的保护异常,要花费很大的开销,重复地执行异常处理程序来模拟这些指令。

使用这一方法的危险是,一个错误的甚至是预谋的虚拟8086程序会使 INTR 中断被长时间的屏蔽,使操作系统由于不能及时响应中断而发生错误。

要避免出现这样的危险,需要在 INTR 中断被禁用时间太长的情况下,使用一个定时器来产生一个 NMI 中断。定时器的时间长度,应略小于为了能及时响应中断所能容忍的最大值,并且,该值在每次响应 INTR 中断后,对定时器的值重新进行初始化。定时器 NMI 中断处理程序则设法对可能产生的 INTR 中断进行处理。这样一来,既可以最大限度的避免产生过多的保护异常,又保证系统中的所有 INTR 中断总能得到及时响应。

IOPL 等于3时执行虚拟8086程序的另一个问题是,INTn 指令有可能不是产生一个通用保护异常。例如,当 n 所对应的中断矢量的 IDT 门的 DPL 等于3时,INTn 指令就将产生一个中断陷阱。因此,对于那些必须使 IDT 门的 DPL 等于3的矢量号 n,如使用 INTn 指令,则需在陷阱处理程序中检查所保存现场的 EFLAGS 寄存器的 VM 位,区分是否由于虚拟8086方式使用 INTn 进入了陷阱处理程序。

7.2.3.2 进入和离开虚拟8086方式

80386可以在保护方式及虚拟8086方式两者之间进行切换。正如中断与异常一样,这种方式的切换可以在单个任务之内,也可以是任务切换操作的一部分。在下列情况下,需从保护方式切换到虚拟8086方式:

(1) 在保护方式下执行 IRET 指令,且 EFLAGS 寄存器中的 NT 位为0,则从栈中弹出 EIP、CS 及 EFLAGS 的值,并返回到同一任务内原来被中断的过程。如果执行 IRET 指令的 CPL=0,且被弹出的 EFLAGS 中的 VM 位为1,则原先被中断的过程应处于虚拟8086方式,这时的 IRET 指令完成保护方式到虚拟方式的切换。如果执行 IRET 指令时的 CPL≠0,VM 位不改变,则 IRET 返回到一个保护方式的程序。需注意,IRET 指令的操作数必须为32位。如果执行 IRET 指令的操作数为16位,这时的保护方式只能是80286的保护方式,80286不存在虚拟8086方式,所以 IRET 指令只恢复低16位值,且置 VM 为0。

(2) 一个任务切换到由386TSS 描述的另一任务,如果新任务 TSS 的 EFLAGS 中的 VM 位为1,则新任务将转换为虚拟8086方式下工作。在访问任务门或 TSS 的 JMP 和 CALL 指令时,以及在任务门的中断或执行 IRET 时,EFLAGS 寄存器中的 NT 位为1,都会引起任务切换。因为286TSS 只存储 EFLAGS 中的低16位,所以,使用286TSS 的任务切换期间,使 VM 位置为0。

当用信号报告一个中断或发生一个异常时,则会发生从虚拟8086方式到保护方式的切换。即中断或异常的处理分为两个步骤:第一步,切换到保护方式;第二步,在保护方式下用第5章所介绍的中断机制进行处理。该机制根据中断或异常的矢量号在 IDT 中的表项,使用如下两种方法之一转移到对应的处理程序:

(1) 如果中断或异常在 IDT 的表项中包含的是一个386陷阱门或386中断门，则中断或异常在同一任务内进行处理。这里使用的门的类型，必须是386的门类型，所以，在保护现场寄存器内容时，所保存的是32位的值，EFLAGS 寄存器的 VM 位的值也得以保存。相匹配的 IRET 指令也才可能按原样恢复。此外，中断或异常处理程序必须在特权级0，否则，将产生通用保护异常，并把新的选择子置于出错码中。因为处理程序在特权级0中，所以，相匹配的 IRET 指令使 VM 位恢复为1。

(2) 如果中断或异常在 IDT 的表项中，包含的是一个任务门，则中断或异常在另一任务中进行处理。在这种情形下，当前 TSS 必须是386TSS 格式，使其可以保存32位 EFLAGS 的值。新任务的类型可以是：由286 TSS 描述的16位保护方式的程序；由386 TSS 描述的32位保护方式的程序(TSS 应有 VM=0)；或由386 TSS 描述的另一虚拟8086任务(TSS 中有 VM=1)。

程序在虚拟8086方式下执行时，EFLAGS 寄存器中的 NT 位被忽略，所以不可能用 IRET 指令完成一个任务切换，并使其工作于虚拟8086以外的工作方式。在虚拟8086方式下执行的 IRET 指令，将弹出 IP、CS 及 EFLAGS 寄存器，以恢复被中断的程序，而不考虑 NT 位的值。

1. 任务内的方式转换

当处理器正执行于虚拟8086方式时，如果响应由386陷阱门或中断门处理的中断或异常，则80386在同一任务内把工作方式从虚拟8086方式改变为保护方式。相匹配的 IRET 指令，通过从保护方式切换回虚拟8086方式，恢复被中断的虚拟8086方式的程序。前面已经说明，方式的转换还包括特权级的转换，因为虚拟8086方式的程序总在3级执行，处理程序则必须在0级执行。

2. 通过386陷阱门或中断门的中断/异常

除了在正常的中断处理开始之前，要将虚拟8086程序的 GS、FS、DS 及 ES 段寄存器压入新栈(0级栈)以外，虚拟8086程序通过陷阱门及中断门的转移，也使用第5章所描述的同样的机制转移到处理程序。为保持使堆栈对齐，在段寄存器压栈时，一律按32位的值压入，段寄存器的值放在低16位，高16位则空着不用。然后，在进入中断或异常处理程序之前，用空选择子装入段寄存器使其初始化。

把四个数据段寄存器 GS、FS、DS 及 ES 保存在堆栈中，并在进入保护方式的中断处理程序之前，用空选择子装入这些段寄存器，使得处理程序可以把段寄存器作为有效的保护方式的值进行保存和恢复，并与进入处理程序时正在执行的程序的执行方式无关。如果进入处理程序时，正执行程序的执行方式为保护方式，则 GS、FS、DS 及 ES 中存放的是有效的选择子，执行处理程序时，一开始就将这些寄存器压栈，并在处理程序执行 IRET 指令前弹回予以恢复。如果进入处理程序时，正执行程序的执行方式是虚拟8086方式，由于段寄存器中存放的不是段选择子，而是实际段地址除以16后的值，因此，一般情况下，段寄存器 GS、FS、DS 及 ES 中存放的段值对保护方式来说是无效的段选择子。所以，在虚拟8086方式下执行程序时，如遇中断或异常进入中断/异常处理程序，在处理程序中要执行如下附加的操作：进入时，保存 GS、FS、DS 及 ES 的值，再用空选择子装入(见下面 InnerStack'()例程)；返回前，恢复 GS、FS、DS 及 ES 原来的值(见 IRET'()例程)。有了这些附加的操作，保护方式下执行的程序与虚拟8086方式下执行的程序在遇到中断或异常时，中断/异常处理程序就完全统一了起来。

图7.5示出的是：在中断或异常期间发生的任务切换操作。这些中断或异常通过386陷阱门或386中断门，从在3级执行的虚拟8086程序转移到0级保护方式的中断处理程序。外层栈由 SS

及 SP 寄存器寻址。此处，SS 包含虚拟8086段寄存器的值，而不是保护方式的段选择子。内层的新栈示于图的左面，现场寄存器按如下顺序保存在新栈中：

- 保存数据段寄存器，并按 GS、FS、DS 及 ES 的次序压栈；
- 保存外层栈指针，即将 SS 及 ESP 压栈；
- 保存标志寄存器，EFLAGS 压栈时，要使 VM 位置1；
- 保存返回地址，即 CS 及 EIP 压栈；
- 出错码压栈。

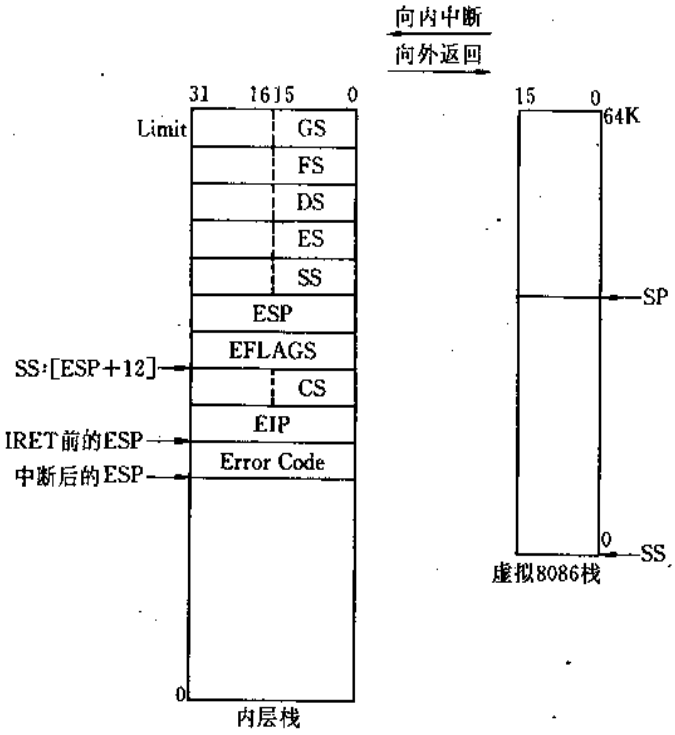


图 7.5 虚拟8086方式通过陷阱门或中断门的栈切换

在处理程序的入口，SS 寻址内层栈段，ESP 则指向该段的出错码。由于处理程序在保护方式下执行，所以 SS 寄存器放的是内层栈的选择子，SS 描述符投影寄存器由该选择子对应的描述符装入。同样，新的选择子寻址一个保护方式的码段，CS 描述符投影寄存器由新选择子对应的描述符装入。

中断或异常处理程序可以检查保存在堆栈中的 EFLAGS 映象，根据有关标志位的值，确定被中断程序的工作方式。如果保存的 EFLAGS 映象中有 VM=1，则被中断的程序工作于虚拟8086方式；否则，被中断的程序工作于保护方式。从图7.5可以看出：当有出错码被压栈时，EFLAGS 映象保存于相对于 ESP 的偏移量为12的位置；无出错码压栈时，EFLAGS 相对于 ESP 的偏移量为8。

虚拟8086方式下的中断/异常处理、中断返回及任务切换，和80386工作于保护方式下的相应例程相比，具有极为相似的执行过程，只是在虚拟8086方式下执行的例程中，增加了对标志寄存器中 VM 位的判断。为了区别，我们把虚拟8086方式下对应于第5章中的中断/异常处理例程 IntTrapGate()、中断返回例程 IRET 及第4章中的任务切换例程 TaskSwitch()分别命名

为 IntTrapGate′()、IRET′及 TaskSwitch′()。以下几段的说明中，我们先列出带有符号一撇“′”的程序清单，然后再和第4章或第5章中的相应程序进行对比说明。

IntTrapGate′()程序

```
/* 包括虚拟8086方式的完整的中断/异常例程 */
IntTrapGate′ (Selector, Offset, ClearIF)
  SelectorType Selector;
  int          Offset,
               ClearIF;  /* 清除 IF 位的标志：为1时，清除 IF */
                         /*                   为0时，IF 不变 */
{
  SegAttributes Attributes;
  SelectorType  GSelector;
  int           Base, Limit, GOffset;
  /* 选择子测试，为空时产生异常 */
  if ((Selector & 0FFFCh)==0)
    SegmentException ($GP, 0);
  /* 读出并测试描述符 */
  ReadDescriptor (Selector, &Attributes, &Base, &Limit,&GSelector, & GOffset);
  if (Attributes.DType==0)  /* 描述符类型为系统段或门时产生异常 */
    SegmentException ($GP, Selector);
  /* 切换到内层栈及内层 CPL */
  if ( (Attributes.Type>=8) && (Attributes.Type<=11)
      && (Attributes.DPL<CPL) && (Attributes.P==1))
            InnerStack (Attributes.DPL, 0, Selector);
  /* 用户程序为虚拟8086方式时，特权级必须为0，否则产生异常 */
  if ((CPL !=0) && (EFLAGS.VM==1))
     SegmentException ($GP, Selector);
  /* 装入 CS 描述符 */
  Selector.RPL=0;  /* 忽略门中选择子的 RPL */
  CSDescriptorLoad (Selector, Attributes, Base, Limit, $GP);
  /* 越界检查 */
  if (Offset>CS.Limit)
    SegmentException ($GP, 0)
  /* 标志寄存器及返回地址压栈 */
  ESP=ESP-4;
  AccessVirtual (SS,ESP, 4,1 /* Write */, $SS, &EFLAGS);
  ESP=ESP-4;
  AccessVirtual (SS,ESP, 4,1 /* Write */, $SS, &CS. Selector);
  ESP=ESP-4;
  AccessVirtual (SS,ESP, 4,1 /* Write */, $SS, &EIP);
  /* 修改标志寄存器，装入 CS 选择子及 EIP */
  EFLAGS.VM=0; /* 为 IntTrapGate′( )所增加 */
  EFLAGS.TF=0;
  EFLAGS.NT=0;
  if (Clear IF==1)
  EFLAGS.IF=0;
  CS.Selector       =Selector;
  CS.Selector.RPL   =CPL;
  EIP               =Offset;
}
```

把程序清单 IntTrapGate′()与第5章中的程序清单 IntTrapGate()相比较可以知道，IntTrapGate′()作了以下两处更改：

(1) 当被中断的程序工作于虚拟8086方式时，中断处理程序必须工作于0级。因此，在读出

处理程序代码段描述符之后，增加了对描述符特权级的检查，并且当特权级不为0时，产生GP异常。清单如下：

```
if ( (CPL!=0) && (EFLAGS.VM==1))
    SegmentException ($GP, Selector);
```

(2) 为保证处理程序工作于保护方式，在此公共的中断处理中，要使标志寄存器的VM位置0，即语句

```
EFLAGS.VM=0;
```

除以上两处更改之外，程序清单IntTrapGate′()与程序清单IntTrapGate()完全一样。所以，此处略去了IntTrapGate′()的流程，请读者参见图5.5阅读理解IntTrapGate′()程序。

虚拟8086方式下转移到内层栈的例程InnerStack′()的程序清单列出如下：

InnerStack′()程序

```
/* 为内层转移进行栈切换的例程 */
InnerStack′ (NewCPL, DwordCount, NewCSSelector)
    Int            New CPL,          /* 切换后的特权级 */
                   DwordCount;       /* 拷贝到新栈的双字参数个数 */
    SelectorType NewCSSelector;  /* 新的CS选择子 */
{
   SelectorType   NewSSSelector;
   SegmentRegister OldSS;
   int            TSSOffset, NewESP, tempESP, OldCPL, Dword1;
   /* 从TSS中读出新栈的SS及ESP */

   TSSOffset=NewCPL*8+4;
   if ( (TSSOffset+7)>TR.Limit)
     SegmentException ($TS,NewCSSelector);
   AccessLinear (TR.Base+TSSOffset,     0 /* PL 0 */, 0 /* Read */, &NewESP);
   AccessLinear (TR.Base+TSSOffset+4, 0 /* PL 0 */, 0 /* Read */, &NewSSSelector);
   /* 保存原来的SS及CPL供参数拷贝时用 */
   OldSS=SS;
   OldCPL=CPL;
   /* 用新CPL取代当前CPL，并将描述符装入投影寄存器，选择子装入段寄存器 */
   CPL=NewCPL;
   SRegLoad (SS,NewSSSelector, $TS);
   SS.Selector=NewSSSelector;
   /* 被中断的程序为虚拟8086方式时的处理 */
   if  (EFLAGS.VM==1)
      {NewESP=NewESP-4;
       AccessVirtual (SS,NewESP, 4, 1 /* Write */, $SS, &GS.Selector);
       GS.Selector=0; GS.Attributes.P=0;
       NewESP=NewESP-4;
       AccessVirtual (SS,NewESP, 4, 1 /* Write */, $SS, &FS.Selector);
       FS.Selector=0; FS.Attributes.P=0;
       NewESP=NewESP-4;
       AccessVirtual (SS,NewESP, 4, 1 /* Write */, $SS, &DS.Selector);
       DS.Selector=0; DS.Attributes.P=0;
       NewESP=NewESP-4;
       AccessVirtual (SS,NewESP, 4, 1 /* Write */, $SS, &ES.Selector);
       ES.Selector=0; ES.Attributes.P=0;
      }
   /* SS及ESP压栈 */
   NewESP=NewESP-4;
```

```
    AccessVirtual (SS,NewESP, 4, 1 /* Write */, $SS, &OldSS.Selector);
    NewESP=NewESP-4;
    AccessVirtual (SS,NewESP, 4, 1 /* Write */, $SS, &ESP);
    /* 拷贝参数                          */
    /* 读出参数时使用原CPL及原栈指针*/
    /* 写入参数时使用新CPL及新栈指针*/
    tempESP=ESP+DwordCount*4;
    for (i=1; i<=DwordCount; i++){
        CPL=OldCPL;
        tempESP=tempESP-4;
        AccessVirtual (OldSS, tempESP, 4, 0, $SS, &Dword1); /* 从原栈读出参数 */
        CPL=NewCPL;
        NewESP=NewESP-4;
        AccessVirtual (SS,NewESP, 4, 1, $SS, &Dword1);  /* 将参数写入新栈 */
    }
    ESP=NewESP;     /* ESP指向新栈栈顶 */
}
```

程序清单InnerStack′()是由第4章的InnerStack()进行修改之后而来的。因为虚拟8086方式进入中断之后，所有数据段寄存器要压栈保存，并使各数据段选择子置为空选择子。所以，该清单的修改内容是：装入新栈指针之后，在满足EFLAGS.VM=1的条件下，执行各段寄存器压栈及相应选择子清0的动作。清单如下：

```
if (EFLAGS.VM==1)
{
    NewESP=NewESP-4;
    AccessVirtual (SS,NewESP, 4, 1, $SS, &GS.Selector);
    Gs.Selector=0; GS.Attributes.P=0;
    NewESP=NewESP-4;
    AccessVirtual (SS,NewESP, 4, 1, $SS, &FS.Selector);
    FS.Selector=0; FS.Attributes.P=0;
    NewESP=NewESP-4;
    AccessVirtual (SS,NewESP, 4, 1, $SS, &DS.Selector);
    DS.Selector=0; DS.Attributes.P=0;
    NewESP=NewESP-4;
    AccessVirtual (SS,NewESP, 4, 1, $SS, &ES.Selector);
    ES.Selector=0; ES.Attributes.P=0;
}
```

我们这里也将程序清单InnerStack′()的流程略去，请参见图4.44阅读此例程。

3. 用NT=0的IRET返回到虚拟8086程序

执行IRET指令时，如果EFLAGS寄存器中的NT位为0，则该指令弹出栈中的信息返回到同一任务内的被中断程序。如果返回后的程序是虚拟8086方式的程序，则在正常的IRET过程完成之后，ES、DS、FS及GS寄存器内容从堆栈中弹出。这些寄存器中的值，以及装入到SS及CS中的值，都按8086方式进行解释，即所有这些值都解释为段地址，而不是保护方式的选择子。

作为响应中断时保存现场，把被中断任务各寄存器的值压入内层栈的逆过程，图7.5也可用来解释由IRET指令退出中断程序返回到被中断程序，并用内层栈中的值恢复被中断时的现场的过程。这时，当前栈为内层栈，并以SS投影寄存器规定段基地址、ESP规定偏移量进行寻址。为保证各现场寄存器的正确恢复，在IRET之前，先要调整ESP指针，使其由指向出错码调整为指向断点，然后执行IRET，恢复各现场寄存器，其过程为：

A14

· 弹出 EIP、CS 及 EFLAGS 寄存器；

· 弹出 ESP 及 SS，恢复外层栈；

· 弹出 ES、DS、FS 及 GS。

IRET′的程序清单如下：

IRET′()程序

```
/* 包括返回到虚拟8086方式的 IRET 指令 */
IRET′()
{
   SelectorType Selector, GSelector;
   SegAttributes Attributes;
   EFLAGSType NewEFLAGS;
   int Base, Limit, Offset, GOffset, NewESP, NewSS;
   /* 测试 NT 位，确定是否嵌套返回 */
   if (EFLAGS.NT==1)
     {       /* 嵌套返回 */
      AccessLinear (TR.Base, 2,0 /* PL 0 */, 0 /* Read */, &Selector)
      TaskGate (Selector, -1 /* Unlink */); /* see Chapter 4. */
     }
   else {/* 非嵌套返回，从栈中弹出中断返回信息 */
      AccessVirtual (SS,ESP, 4, 0 /* Read */ , $SS, &Offset);
      ESP=ESP+4;
      AccessVirtual (SS,ESP, 4, 0 /* Read */ , $SS, &Selector);
      ESP=ESP+4;
      AccessVirtual (SS,ESP, 4, 0 /* Read */ , $SS, &NewEFLAGS);
      ESP=ESP+4;
      if ( (NewEFLAGS.VM==1) && (CPL==0) )
      { /* 返回虚拟8086方式的程序的处理 */
        AccessVirtual (SS,ESP, 4, 0 /* Read */ , $SS, &NewESP);
        ESP=ESP-4;
        AccessVirtual (SS,ESP, 4, 0 /* Read */ , $SS, &NewSS);
        ESP=ESP-4;
        /* 从栈中弹出 ES、DS、FS 及 GS 的值送 ES、DS、FS 及 GS 段寄存器 */
        AccessVirtual (SS,ESP, 4, 0 /* Read */ , $SS, &ES.Selector);
        ESP=ESP-4;
        AccessVirtual (SS,ESP, 4, 0 /* Read */ , $SS, &DS.Selector);
        ESP=ESP-4;
        AccessVirtual (SS,ESP, 4, 0 /* Read */ , $SS, &FS.Selector);
        ESP=ESP-4;
        AccessVirtual (SS,ESP, 4, 0 /* Read */ , $SS, &GS.Selector);
        /* 指向外层虚拟8086栈，恢复用户特权级 CPL=3 */
        ESP=NewESP;
        SS.Selector=NewSS;
        CPL=3;
        }
      else {
        /* IRET 回到保护方式的选择子测试 */
        if ( ((Selector & 0FFFCh)==0) || (Selector.RPL<CPL))
          SegmentException ( $GP, 0);
        if (Selector.RPL>CPL) {
          /* 返回外层，调用 OuterStack()恢复外层栈 */
          OuterStack (Selector.RPL,0);
        }
        /* 读出并测试描述符 */
```

```
        ReadDescriptor (Selector, &Attributes, &Base, &Limit, &GSelector, &GOffset);
        if (Attributes.DType==0)
          SegmentException ($GP, Selector);
        /* 装入CS描述符 */
        CSDescriptorLoad (Selector, Attributes, Base, Limit, $GP);
        /* 越界检查 */
        if (Offset>CS.Limit)
          SegmentException ($GP,0);
      }
      /* 装入标志寄存器、CS选择子及EIP */
      EFLAGS=NewEFLAGS;
      CS.Selector=Selector;
      EIP=Offset;
    }
}
```

程序清单 IRET′与第5章中的 IRET 相比，增加了返回后的程序是否为虚拟8086方式的程序的判断及虚拟8086方式时，各数据段寄存器的恢复操作。阅读程序清单 IRET′可参考图5.6的流程图。

4. 转移到不同任务

在虚拟8086方式下响应中断或异常时，如果中断矢量对应 IDT 表项包含一个任务门，则该中断或异常将转到一个不同的任务。因为保护方式下，执行 JMP、CALL 或 NT=1时的 IRET 指令，也完成一次任务切换，所以，任务切换可以是从虚拟8086方式的程序切换到保护方式的程序或另一个虚拟8086程序；也可以是保护方式的程序切换到保护方式的程序或虚拟8086方式的程序。如果一个任务经切换后，离开了虚拟8086方式，则任务切换过程中保存在 TSS 中的 EFLAGS 寄存器映象，应使 VM 位为1，用以记录切换前是虚拟8086方式；如果经任务切换，进入的新任务的 TSS、EFLAGS 寄存器中，发现 VM=1，则切换后进入虚拟8086方式。

与切换到保护方式相比较，任务切换进入虚拟8086方式或离开虚拟8086方式，除段寄存器的装入与保存不一样以外，其它过程完全相同。虚拟8086方式是保护方式下的8086方式，所以段寄存器的内容为段地址，不以选择子方式进行解释。

虚拟8086方式下任务切换 TaskSwitch′()的程序清单如下：

TaskSwitch′()程序

```
/* 包含有虚拟8086方式任务的任务切换 */
TaskSwitch′ (Selector, Attributes, Base, Limit, Linkage)
    SelectorType   Selector;
    SegAttributes  Attributes;
    int            Base, Limit,
                   Linkage;     /* 链接字。0：无链接；1：链接；-1：解链 */
{
    if Limit<103)  /* 任务状态段不足104字节长时产生异常 */
      SegmentException ($TS,Selector);
    /* 当前机器状态保存到老任务的TSS */
    AccessTSSState (1 /* Write */);
    /* TR及投影寄存器指向新TSS */
    TR.Base        =Base;
    TR.Limit       =Limit;
    TR.Attributes  =Attributes;
    OldTSS         =TR.Selector;  /* 保存老任务的选择子供链接 */
    TR.Selector    =Selector;
```

```
/* 把新任务 TSS 中的内容装入各硬件寄存器 */
AccessTSSState (0 /* Read */);
/* 根据链接字的值进行处理 */
if (Linkage==1 /* Link */) {
    /* 如链接,则新任务中保存老任务的 TSS 选择子,设置嵌套位, */
    /* 新任务的 TSS 描述符置为忙                              */
    AccessLinear (TR.Base, 2,0 /* Level 0 */, 1 /* Write */, &OldTSS);
    EFLAGS.NT=1;
    SetTSSBusy (Selector, 1);
    }
else if (Linkage==-1) {/* 如解链,则置老任务的 TSS 描述符为非忙 */
    SetTSSBusy (OldTSS, 0);
    }
else if (Linkage==0) { /* 如无链接,则:  */
    SetTSSBusy (OldTSS, 0);  /* 置老任务的 TSS 描述符为非忙 */
    SetTSSBusy (Selector, 1);  /* 置新任务的 TSS 描述符为忙 */
    }
CR0.TS=1;  /* 设置任务切换位 */
if (EFLAGS.VM==1) /* 虚拟8086方式时,切换后的特权级为3级 */
  CPL=3;
else                    /* 不是虚拟8086方式时,切换后的特权级从 CS 的 RPL 中获取 */
    CPL=CS.Selector.RPL
/* 为 LDTR 及段寄存器装入描述符投影寄存器 */
/* 将所有描述符的存在位预置为0            */
LDTR.Attributes.Present=0;
CS.Attributes.Present=0;
SS.Attributes.Present=0;
DS.Attributes.Present=0;
ES.Attributes.Present=0;
FS.Attributes.Present=0;
GS.Attributes.Present=0;
/* 装入 LDTR */
if (LDTR.Selector.TI==1  /* 如 LDTR 的选择子在局部描述符表中则产生异常 */
  SegmentException ($TS,LDTR.Selector);
if ((LDTR.Selector & 0FFFCh)=0) {
  /* 允许局部描述符表为空,但需将存在位设置为0,使以后   */
  /* 对局部描述符表的访问产生异常 */
  LDTR.Attributes.P=0;
  }
else { /* 读出并测试描述符 */
  ReadDescriptor (LDTR.Selector,&Attributes, &Base, &Limit,
                  &GSelector, &GOffset);
  if ( (Attributes.DType==1)
      || (Attributes.Type !=2 /* LDT */)
      || (Attributes.Present==0))
    SegmentException ($TS,LDTR.Selector);
  /* 装入局部描述符表投影寄存器 */
  SetAccessed (LDTR.Selector);
  LDTR.Attributes =Attributes;
  LDTR.Base       =Base;
  LDTR.Limit      =Limit;
  }
if (EFLAGS.VM==0) {/* 虚拟8086方式时不执行以下装入操作 */
  /* 装入 CS 投影寄存器 */
  JumpGate (CS.Selector, EIP,STS);
  /* 装入 SS,DS,ES,FS 及 GS 投影寄存器 */
```

```
        SRegLoad (SS,SS.Selector, $TS);
        SRegLoad (DS,DS.Selector, $TS);
        SRegLoad (ES,ES.Selector, $TS);
        SRegLoad (FS,FS.Selector, $TS);
        SRegLoad (GS,GS.Selector, $TS);
        }
    /* 排错寄存器DR7中的局部启用位置为0 */
    DR7.LE0=0;
    DR7.LE1=0;
    DR7.LE2=0;
    DR7.LE3=0;
}
```

此程序是由第4章的任务切换例程 TaskSwitch()稍经修改而来的。修改的内容为：

(1) 如切换后为虚拟8086方式的程序，则切换后的 CPL 固定为 CPL=3，而不是从选择子中的 RPL 字段装入 CPL。

(2) 新任务为虚拟8086方式的任务时，只需装入段寄存器，没有装入段描述符投影寄存器的动作。

阅读理解 TaskSwitch'()的例程，请参考图4.47的流程。

7.2.4 实地址方式的若干问题

实地址方式是80386最基本的工作方式。以下我们将从三个方面说明实地址方式下执行的程序所具有的特性：

- 开机时80386处于实地址工作方式；
- 实地址方式及保护方式之间的相互转换；
- 实地址方式使用不同的中断机制。

7.2.4.1 初启时80386的状态

80386在复位时，处理器处于实地址工作方式，各个有关寄存器用表7.3中给出的值进行初始化。表7.3中未列出的其它寄存器在复位时的状态是未定义的，这些寄存器必须由软件进行初始化。Intel 公司的技术手册，有关于复位80386处理器过程的详细说明。

表 7.3 寄存器的初始化值

寄存器	初始化值	寄存器	初始化值
EAX	自测试特征	ES	0
ECX	未定义	CS	0F000h
EDX	构件/版本号	SS	0
EBX	未定义	DS	0
ESP	未定义	FS	0
EBP	未定义	GS	0
ESI	未定义	IDTR	Base=0,Limit=3FFh
EDI	未定义	CR0	0或10h
EFLAGS	00000002h	DR7	0
EIP	0FFF0h		

复位之后，如果要求进行自测试，则在 EAX 寄存器中存放标志自测试结果的值。如果测

试通过，则 EAX 中的内容为0；否则 EAX 中为非0。如果复位后不要求进行自测试，则 EAX 中的内容是未定义的(通常为非0)。关于复位后自测试的选择及自测试的测试内容，可参考80386技术手册。

EDX 寄存器用 component/revision ID 进行初始化，component ID 放在 DH 寄存器，revision ID 放在 DL 寄存器。对于80386，component＝3，但今后386系列处理器将使用不同的 component ID。早期的86系列处理器不支持这一特性，DX 中的值复位时是随机的。DL 寄存器中的 revision ID 值则是处理器每一重要版本的序号。

程序指针 CS 及 EIP 寄存器，在复位时分别置初值为0F000h 及0FFF0h，在实地址工作方式，CS 及 EIP 形成20位的程序指令地址。80386是32位的地址总线，寻址4G 存储空间，复位后，系统应转入初始引导程序，这部分程序一般都是作为固件，并安排在存储器的高端地址。第4章介绍地址空间分配时，已考虑将初始引导程序安排在0FFFF0000h～0FFFFFFFFh 区域，在此区域中，地址线16～31的各位都应为1。现在 CS 及 EIP 的初值已保证地址线的16～19位为1，而地址线的最高12位，即20～31位，则由硬件强制置1，这可由硬件提供一个标志触发器实现。系统每次复位时，该标志触发器置位，并在访问 CS 寄存器时向地址线的最高12位“或”入一个“1”信号。该标志触发器的置位状态，使初始引导程序工作于内存实际地址空间以外的另一存储段中。由于段间转移指令要重新装入 CS 寄存器，因而每当执行段间转移指令时，硬件逻辑保证向地址总线高12位提供“1”信号的标志触发器复位。因此，标志触发器在初始引导程序执行的第一个段间控制转移指令时复位为0。此后，再次访问 CS 寄存器时，不再向高12位地址线提供“1”信号，使程序正常地工作于前1M 地址空间。由于标志触发器在复位时置位，而在每次段间转移指令时都使其复位，所以初始引导程序只在复位后至引导程序的第一条段间转移指令之间执行一次，以后，由于实地址方式无法寻址1M 空间以外的地址，引导程序不可能再次执行。多数80386系统都把初始引导程序做成 ROM 固件，并在固件中，除固化初始引导程序代码之外，还把操作系统的一部分固化在 ROM 中。在执行过段间转移指令之后，如欲再次访问 ROM 固件中的内容，由于为该固件设定的地址不是实地址方式所能寻址的范围，如不提供某种特别的机制(例如类似于复位时的机制)，则这种对 ROM 固件内容的访问，只能在80386保护方式下才能实现。

此外，复位时的初始化还有如下内容：

(1) 栈段、数据段寄存器 SS、DS、ES、FS 及 GS 被初始化为0，使其寻址最低64K 存储空间；

(2) EFLAGS 寄存器被初始化为2，禁止响应中断、禁止单步工作方式，使初始化引导程序不受外界中断的影响连续地执行；

(3) IDTR 经初始化，使其基地址为0，界限为03FFh。所以中断表为0地址开始的连续1024字节区域，包含有256个中断表项。这与8086相兼容；

(4) CR0初始化为0或10h。即除 ET 位之外，所有 CR0中的其它位都置为0。ET 位的状态取决于复位时检查到的系统配置中是否有80387芯片。如配置有80387芯片，则 ET 位置为1，CR0为10h；如无80387芯片，ET 位为0，即 CR0为0。由于保护方式标志位 PE 及分页机制标志位 PG 都在 CR0中，无论 CR0初始化为0或10h，PE、PG 位皆为0，所以处理器复位后工作于实地址方式。

(5) DR7被初始化为0，禁用所有的调试断点。

除上述的硬件初始化以外，还有如下由软件进行初始化的内容：

(1) 给栈指针赋初值。由于中断必须使用堆栈，因此，只有在栈指针被赋初值之后，才能启用中断；

(2) 初始化中断表。所有准备启用的中断，其对应的中断表项中，在允许响应中断之前，必须预先给出中断门的全部信息。

虽然80386在复位时，同时禁止了对外部中断事件的响应，但处理器内部的非屏蔽中断仍然起作用，并可能在复位后立即发生。如果硬件不提供另外的特别机制，初始引导程序执行期间，仍然会被 NMI 中断所打断。这种情况在一般情况下是不多见的，但是如果万一发生这种情况，并发生在软件的初始化完成之前，初始化引导程序将发生混乱，只好对系统重新进行复位操作。

7.2.4.2 实地址方式到保护方式的转换

复位之后，初始化引导程序要对系统段寄存器赋初值，并把 CR0中的 PE 置为1，使系统转入保护方式。因为实地址方式及保护方式的寻址规则有很大区别，所以必须立即执行一条段间转移使已在实地址方式下预取的指令及已译码的指令变为无效。这里的段间转移指令通常是跳转到指令的下一条。从形式上看，该指令跳转到下一条，好象空操作一样似乎毫无意义，但这条看来没有意义的指令却是改变工作方式后必不可少的。

在进入保护方式之后，需继续进行如下工作：

(1) 用有效的保护方式选择子或空选择子装入各个段寄存器；

(2) 用段间控制转移指令重新装入 CS 寄存器；

(3) 用一个可用386(或286)TSS 类型的段的选择子装入 TR 寄存器；

(4) 用 LDT 类型的系统段的选择子(或空选择子)装入 LDTR。

与前述各个段寄存器的装入不一样，GDTR 及 IDTR 必须在系统进入保护方式之前装入，装入的内容，是相应 GDT 段及 IDT 段的基地址及界限。

7.2.4.3 保护方式到实地址方式的转移

由两种方法完成从保护方式到实地址方式的转移：

(1) 使用外部硬件复位80386系统；

(2) 用对应于 PE 位为0的值装入 CR0寄存器。

第一种方法已在前面进行了叙述。第二种方法提供的是软件方式的切换，也是在正常情况下必须使用的方法，并必须遵循如下规则：

(1) 当前正执行的程序所在代码段的大小必须为64K；

(2) 装入 CR0中的数据使 PE 位为0之前，SS、DS、ES、FS 和 GS 必须装入一个选择子，使其访问一个存在的读/写向上扩展数据段，并且 DPL=0，大小为64K；

(3) 装入 CR0中的数据必须使 PE 位为0；

(4) 装入 CR0指令之后，必须紧跟一条直接的段间 JMP 跳转指令。该跳转指令将使预取指令变为无效，并转移到物理存储器最低1M 字节中的实地址方式的程序。

在从保护方式转到实地址方式期间，所有的中断(包括非屏蔽中断)都应予以禁止。这种禁止响应所有中断的状态，一直持续到 IDTR 被重新装入，并指向在实地址方式所用格式的中断表为止。切换回实地址方式的另一可选方案是使用虚拟8086方式，执行8086的程序。

在从保护方式转移回实地址方式之前，必须关闭分页机制。这时，程序所在的代码段及使用的数据段是等同映射的，即线性地址等于物理地址。关闭分页机制的方法，是使用一条装入 CR0的指令，用装入的数据使 CR0中的 PG 位改变为0。再执行一条段间转移，使预取指令无

效，最后再装入 CR3，刷新分页机制高速缓存。

7.2.4.4 实地址方式的中断处理

处理器在实地址方式下执行时发生的中断及异常，要使用不同于保护方式所用方法的另外的机制进行处理。

在实地址方式中，中断描述符表是一个含有四个字节的表项数组，由 IDTR 进行寻址。数组的每一表项可形成一个指针，其格式如图7.6所示，指向中断处理程序的第一条指令。图7.6的格式中，低16位为偏移量，高16位为段寄存器的值。响应中断后，将它们的值装入 IP 及 CS，即可使系统执行相应的中断处理程序。

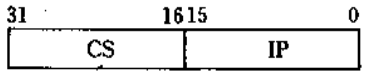


图 7.6 实地址方式中断表表项格式

实地址方式进行中断及异常处理时，INT 指令按16位的指令进行解释。首先，16位的 FLAGS 寄存器压栈，然后，查阅中断表，找出装入 CS 及 IP 的指针，并在将此入口指针装入后开始处理程序的执行，最后，将 EFLAGS 寄存器中的 TF 及 IF 位置为0，禁止单步及中断功能。实地址方式时，异常的处理不会压入出错码。

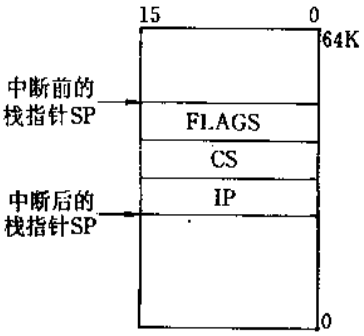


图 7.7 实地址方式中断/异常和 IRET 时栈的用法

第3章描述的 IRET 指令可用来从实地址方式的中断及异常处理中返回。这时，IRET 指令按第3章所描述的语义，而不用第5章为保护方式描述的扩展语义进行解释。并从栈中把 IP、CS 及 FLAGS 的值弹出，以恢复被中断的实地址方式程序。

图7.7示出实地址方式中断期间压入的值，及实地址方式 IRET 指令期间弹出的值。堆栈的栈宽为16位，与8086相一致，压入现场寄存器时的次序为 FLAGS、CS、IP。完成中断或异常公共处理，并在转入中断或异常处理程序时，SP 指向压入映象的 IP。中断返回时，相匹配的 IRET 将以相反的次序弹出 IP、CS 及 FLAGS 的值，恢复被中断的程序。

实地址方式下中断/异常处理程序 InterruptREAL()的程序清单如下：

InterruptREAL()程序

```
/* 实地址方式下中断/异常的处理 */
InterruptREAL (VecNumber)
int VecNumber;  /* 矢量号 */
{
    /* 16位的寄存器 FLAGS、CS、IP 压栈 */
    SP=SP-2;
    Access8086Virtual (SS,SP,2,1, $SS, &FLAGS);
    SP=SP-2;
    Access8086Virtual (SS,SP,2,1, $SS, &CS.Selector);
    SP=SP-2;
    Access8086Virtual (SS,SP,2,1, $SS, &IP);
    EFLAGS.TF=0;    /* 断开单步功能 */
    EFLAGS.IF=0;    /* 关中断 */
```

```
/* 检查矢量号是否越界,并在越界时产生异常 */
if (VecNumber * 4+3>IDTR.Limit)
   SegmentException ($GP,0);
/* 从中断描述符表读出矢量号对应的处理程序入口地址 CS:IP */
/* 并将其装入 CS 及 IP 寄存器                          */
AccessLinear(IDTR.Base +VecNumber * 4, 2,0,0,&IP);
                        /* 用特权级0读出段偏移装入 IP      */
AccessLinear(IDTR.Base +VecNumber * 4+2,2,0,0,&CS.Selector);
                        /* 特权级0读出段地址装入 CS        */
}
```

程序清单中的例程 Access8086Virtual()用于访问实地址方式栈的存储器。如果指定的中断或异常的中断表项超过 IDTR 的界限,将产生通用保护异常。本程序比较简单,清单中的注释也较清楚,故不再做更详细的说明。

7.3 执行80286保护方式程序

在80286上保护方式下执行的程序也可在80386上执行。80286的应用程序及操作系统不加任何修改,即可在80386上执行。运行80286程序有两个选项:

(1) 在80386上运行80286操作系统,并在此操作系统的支持下运行80286应用程序。这种方法只使用80386的16位子集,把80386当成一个快速的80286使用。

(2) 在80386上运行32位的操作系统,并通过在应用程序及32位操作系统之间加入一个16位的界面库,而运行80286应用程序。80286应用程序使用的对16位操作系统函数的调用,就通过此界面库转换成对80386操作系统中32位函数的调用。80286的寄存器、寻址方式及指令是80386的子集,且80286的存储管理、保护及任务模型也是80386模型的一个子集,因此,80386可以运行80286的操作系统,80286支持与80386同样的存储器段,只是基地址及界限的范围较小。80286也支持80386相对应的系统段及门,且其类型具有如下关系:

- 286任务门类型=386任务门类型=5;
- 286其它门类型=386相应门类型-8=(4,6,7);
- 286TSS 类型=386TSS 类型-8=(1,3)。

80286使用的支持存储段的格式与80386使用的格式相同(见图4.10)。但是,80386使用全部8个字节,而80286只使用较低的6个字节,最高两个字节不用。最高两个字节包含的信息有:

- 高位基地址　　8位
- 高位界限　　4位
- G 位　　1位
- D(B)位　　1位
- 未用位　　2位

因此,80286中,应把基地址的最高8位视为0,故地址空间范围为$2^{24}=16M$;把界限最高4位视为0,故最大界限为$2^{16}=64K$;G 位为0,故总是字符粒度单位;D(B)位为0,故为286的保护方式,且为代码段、栈段及向下扩展的栈段提供16位的默认值。

286的系统段描述符有与存储段描述符同样的限制,也不使用最高两个字节。在各种类型的门中,最高两个字节为偏移量的高16位,286视这高16位为0,故286门的偏移量为$2^{16}=64K$,

A14

同时，286调用门把 Dwordcount 字段解释为字计数，而不是双字计数，所以，使用286调用门的段间调用所拷贝的参数字节数，是2 * Dwordcount 个字节。

286调用门、陷阱门和中断门向堆栈压入16位的数据，分别是386压入各寄存器的低16位值。除此以外，所有门的其它操作与386相同。使用这些门从进入的程序中返回，可使用 RET 及 IRET 指令，其操作数的长度应为16位。

286的任务机制与386的任务机制相同，但是由286的 TSS 进行任务切换时，保存或恢复的寄存器的状态只有16位，并省略 CR3字段及 I/O 许可位图。

第3章中曾描述过的 FLDENV、FRSTOR、FSTENV、FNSTENV、FSAVE 及 FNSAVE 指令，装入和存储控制字、状态字、标志字及出错指针寄存器。如果这些指令以16位长的操作数执行，则只保存和恢复每个双字格式的低16位。出错指针的偏移量部分也限制为16位，并省略操作码字段。

附录A 排　　错

正如所有编程人员所知，在一个程序经过若干设计步骤并最后完成编码以后，该程序还要经过调试（或称为排错），以排除编码错误甚至逻辑错误。要把程序中某些莫明其妙的错误找出来，不仅需要技巧，而更需要工具。已开发出来的各种各样的软件调试程序（或称排错程序），则是用来帮助编程人员找出他们所写程序中的各种毛病的工具。一般来说，软件调试程序有如下一些特点：

(1) 程序步进执行，观察每一行程序执行后的结果是否与预想的一致；

(2) 在程序中设置断点，观察一段程序执行完毕后的结果是否与预想的一致；

(3) 程序中某一变量变为某一预定值时暂停，观察循环或某些变量达到某一特定值时的运行效果。

现在，软件调试程序已经使很多重要的程序正确地实现预想的功能，起着非常重要的作用。

本附录不讨论也不提供专门的软件调试工具，而只向读者介绍 80386 硬件支持的基本调试设施。利用这些基本调试设施，软件开发人员即可开发出功能全面的适合调试各种程序的方便的调试工具。

80386 的调试设施包括：

- 一字节的陷阱指令；
- 单步指令；
- 断点检测；
- 任务切换时的自陷。

80386 的断点设施是对 86 系列调试设施的有效的扩展。

A.1 术　　语

首先，需要澄清本附录中所要使用的术语。

调试异常　作为执行指令的一部分而检测到的异常。

调试处理程序　当检测到一个调试中断或启用的调试异常后，接受控制转移的例程。

调试故障　在一条指令中，指令虽已取出，但实际操作尚未执行之前检测到的调试异常。产生故障的情况有：调试寄存器保护，指令断点。

调试陷阱　在一条指令中，指令的实际操作完成之后检测到的调试异常。产生调试陷阱的情况有：任务切换；单步指令；数据断点。

A.2 调试断点

一个断点，允许编程人员对特定的线性地址设置特定的条件；当程序访问到该线性地址并满足指定的条件时，即跳转到异常处理程序。80386 可支持同时设置四个断点条件，因此，编程

人员可在程序中的四个位置设置条件，使其转向异常处理程序。这四个断点的每一个断点，都可以是如下三种不同类型的任何一种：

- 只在指令地址与断点地址一致时，断点有效；
- 数据写入地址与断点地址一致时断点有效；
- 数据读出地址或数据写入地址与断点地址一致时断点有效。

A.2.1 调试寄存器

为支持提供四个调试断点，在 80386 中增加了八个寄存器，编号为 DR0—DR7。这八个寄存器中有四个用于断点，两个用于控制，另两个保留未用。对这八个寄存器的访问，只能在 0 级特权级进行。在其他任何特权级对这八个寄存器中的任何一个寄存器进行读或写访问，都将产生无效操作码异常。此外，这八个寄存器还可用 DR6 及 DR7 中的 BD 位和 GD 位进行进一步的保护，使其即使是在 0 级也不能进行读出或写入。

对这些寄存器的访问，使用通常的 MOV 指令：

MOV reg DRi

该指令将调试寄存器 i 中的内容读至通用寄存器 reg 中；

MOV DRi reg

该指令将通用寄存器 reg 中的内容"写至"调试寄存器 i 中。此处，i 的取值可以为 0,1,2,3,6,7 中的任何值。

图 A.1 示出了这八个调试寄存器。这些寄存器的功能如下：

DR0—DR3 寄存器 DR0—DR3 包含有与四个断点条件的每一个相联系的线性地址(断点条件则在 DR7 中)。因为这里使用的是线性地址，所以，断点设施的操作，无论分页机制是否启用，都是相同的。

DR4—DR5 保留。

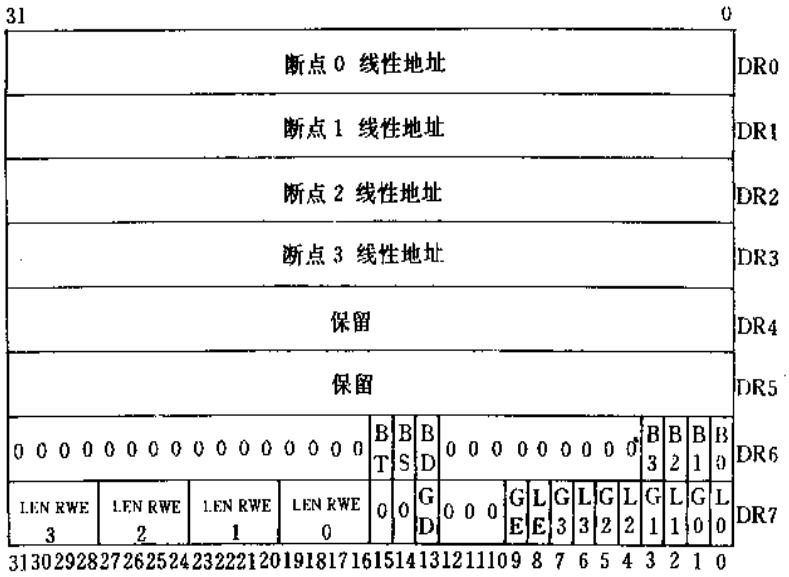


图 A.1 排错寄存器

DR6 DR6 是调试状态寄存器。当一个调试异常产生时，处理器设置 DR6 的相应位，用于

异常的类型及/或引起异常的断点。

DR7 调试控制寄存器，分别对应四个断点寄存器的控制位，对断点的启用及断点类型的选择进行控制。所有断点寄存器的保护也在此寄存器中规定。

1. DR6 各位的功能

B0—B3 当断点线性地址寄存器规定的条件被检测到时，将对应的 B0—B3 位置 1。置位 B0—B3 与断点条件是否被启用无关。即 B0—B3 的某位被置 1，并不表示要进行对应的断点异常处理。

BD 如下一条指令要对八个调试寄存器之一进行读或写时，则在指令的边界置 1BD 位。在一条指令内，每当即将读写调试寄存器时，也置 1 BD 位。置 1 BD 位与 DR7 中 GD 位启用与否无关。

BS 如果单步异常发生时，BS 位被置 1。单步条件由 EFLAGS 寄存器中的 TF 位启用。如果程序由于单步条件进入调试处理程序，则 BS 位被置 1。与 DR6 中的其它位不同的是，BS 位只在单步陷阱实际发生时才置位，而不是检测到单步条件就置位。

BT BT 位，对任务切换导致 TSS 中的调试陷阱位被启用而造成的调试异常，指示其原因。对这一条件，在 DR7 中没有启用位。

DR6 中的各个标志位，在处理机的各种清除操作中不受影响，因此，调试异常处理程序在运行以前，应清除 DR6，以避免下一次检测到异常条件时，受到原来的 DR6 中状态位的影响。

2. DR7 各位的功能

LEN LEN 为一个两位的字段，用以指示断点的长度。每一断点寄存器对应一个这样的字段，所以共有四个这样的字段分别对应四个断点寄存器。LEN 的四种译码状态对应的断点长度(字节数)如表 A.1 所示。

这里，如果断点是多字节长度，则必须按相应多字节边界进行对齐。如果对应断点是一个指令地址，则 LEN 必须为 00。

RWE RWE 也是两位的字段，用以指示引起断点异常的访问类型。共有四个 RWE 字段分别对应四个断点寄存器，RWE 的四种译码状态对应的访问类型如表 A.2 所示。

表 A.1 断点字节数

LEN	说　　明
0 0	断点为一字节
0 1	断点为两字节
1 0	保留
1 1	断点为四字节

表 A.2 引起断点异常的访问类型

RWE	说　　明
0 0	指令
0 1	数据写
1 0	保留
1 1	数据读和写

GE/LE GE/LE 位分别指示准确的全局/局部数据断点。如果 GE 或 LE 被置位，则处理器将放慢执行速度，使得数据断点准确地把产生断点的指令报告出来。如果这些位没有置位，则处理机在执行数据写的指令接近执行结束稍前一点报告断点条件。建议读者每当启用数据断点时，启用 LE 或 GE。降低处理机执行速度除稍微降低一点性能以外，不会引起别的问题。但是，对速度要求严格的代码区域除外。这时，必须禁用 GE 及

LE，并且必须容许某些不太精确的调试异常报告。

L0—L3/G0—G3 L0—L3 及 G0—G3 位分别为四个断点寄存器的局部及全局启用信号。如果有任一个局部或全局启用位(Li 或 Gi)被置位，则由对应断点寄存器 DRi 规定的断点被启用。

GD GD 位启用(由 DR6 的 BD 标志的)调试寄存器保护条件。注意，处理程序在每次转入调试异常处理程序入口处清除 GD 位，从而使处理程序可以不受限制地访问调试寄存器。

前述的各个 L 位(即 LE，L0—L3)是有关任务的局部位，使调试条件只在特定的任务启用。而各个 G 位(即 GD，G0—G3)是全局的，调试条件对系统中的所有任务皆有效。在每次任务切换时，处理器都要清除 L 位。

A.2.2 断点地址识别

LEN 字段及断点线性地址的组合，规定调试异常检查的四个线性地址的范围。上面已经提到，断点线性地址必须对齐于 LEN 规定的多字节长度的相应长度边界。事实上，处理器在检查断点时，根据 LEN 规定的长度，忽略线性地址的相应低位。例如，当 LEN=11 时，线性地址的最低两位被忽略，即把线性地址最低两位视为 00，因而按四字节边界对齐。而当 LEN=01 时，线性地址的最低位被忽略，即把线性地址的最低位视为 0，因而按两字节边界对齐。

对于由断点线性地址及 LEN 规定的地址范围内类型正确的任何字节的访问都产生异常，数据的访问及指令的取出，都要按所有四个断点地址范围进行检查。如果断点地址范围的任何字节匹配，访问的类型也匹配，则断点异常被报告。

表 A.3 给出识别数据断点的几个例子，这里假定所有断点被启用，而且设置了正确的访访问类型。

表 A.3 断点判别的实例

		地址		长度	断点
寄存器内容	DR0	00FF02	1,	LEN=00	
	DR1	00CC32	2,	LEN=01	
	DR2	0D0004	4,	LEN=11	
	DR3	01FF00	4,	LEN=11	
引起异常的访问		00FF02	1		B0=1
		00CC33	1		B1=1
		0D0007	2		B2=1
		00FEFF	4		B0=1
		01FF00	4		B3=1
		01FF03	4		B3=1
不引起异常的访问		00FF01	1		—
		00FF00	2		—
		00CC34	1		—
		01FEFF	1		—
		0D0000	4		—

按表 A.3 中规定的地址及长度得到可能产生断点异常的地址(范围)为：

① 00FF02h

② 00CC32h，00CC33h

③ 0D0004h，0D0005h，0D0006h，0D0007h

④ 01FF00h，01FF01h，01FF02h，01FF03h

在引起异常的访问一栏中，第3行从0D0007h开始，要访问2个字节，所以访问的地址是0D0007h及0D0010h。因为0D0007是断点异常地址范围第③种情况中的一个地址，所以访问到0D0007时即产生了对应于DR2所设置的断点异常，使B2=1。又如第4行中从00FEFFh开始，要访问4个字节，所以访问的地址是00FEFFh，00FF00h，00FF01h及00FF02h，因为00FF02h地址是断点异常地址范围第一种情况中的地址，所以访问到00FF02h时，产生对应于DR0所设置的断点异常，使B0=1。引起异常的访问的其它各行，请读者自己验证。此外，不难验证不引起异常的访问一栏中，所有的访问都不会访问到DR0～DR3中规定的断点地址及范围中的任一地址，读者不妨试试。

要设置地址未对齐于相应边界的断点，必须定义多个断点条件，使各个断点条件的组合正确地覆盖所要求的地址范围。例如，有一个四字节的数据存放在地址00FFFEh，要求访问该地址的数据时产生异常。由于该数据占用的地址为00FFFEh～010001h范围，未在四字节边界对齐，因此需设置两个断点：即地址为00FFFEh、LEN=01的断点及地址为010000h，LEN=01的断点。两个断点规定的断点长度都为2(LEN=01)，地址00FFFEh及010000h又都对齐于两字节边界，所以这两个断点地址的设置是合法的。

A.2.3 代码断点与数据断点的比较

指令访问断点与数据访问断点之间有如下几点区别：

(1) 在RWE字段的设置不同。指令断点，RWE=0；数据断点，RWE≠0。

(2) LEN的设置不同。指令断点的长度只能是00即一字节；数据断点的长度可以是1、2、4字节。由于很多指令的长度超过一个字节(事实上，指令长度为1～15字节)，所以指令断点必须设置在指令的第一个字节。

(3) 指令断点属故障类型，数据断点属陷阱类型。指令断点故障在指令执行前被检查及报告，数据断点陷阱则在对断点位置进行读和写之后才被报告。可以看出，数据断点陷阱对断点数据未进行写保护。要保护断点的数据不会被写入操作破坏，必须先在调试处理程序中保存断点原来数据的拷贝。

由于指令断点在指令执行之前被报告，因此，很明显，对该指令不能简单的重新执行。因为每一次新的执行都简单地重复产生故障，所以，如果调试处理程序不禁用断点，则这种故障就会形成无限的循环。为解决这一问题，就需用到80386中EFLAGS的RF位。当RF位置位时，任何指令断点将被忽略，因此，在RF位保持为置位状态时，指令断点将不再起作用。但RF位的置位状态不会长久保持。事实上，处理器的内部逻辑保证，在任何一条指令成功完成之后，都将RF位清零，因此，RF位的置位状态最多只保持一条指令的时间。也就是说，在RF位置位后的下一条指令，指令断点不起作用，这样只要在重新执行该指令之前，将RF置1，即可保证该指令断点不会形成无限循环，而且，也不影响紧接的下一条指令也设置指令断点。

RF位的置位，不是用某一个操作直接将EFLAGS的RF位置1来完成。每当进入一个故障处理程序，处理器保存中断现场时，需把断点等信息压栈。当把EFLAGS寄存器压栈时，推入栈中的EFLAGS的RF位是1，因此用IRET指令退出故障处理程序时，从栈中弹出的EFLAGS寄存器标志位中的RF为1，从而将RF位置位。读者可以看到，RF位不仅在断点故

障时，从处理程序返回后重新执行该断点指令的状态为1，而且在其它故障时，从相应故障处理程序返回后重新执行引起那一故障的指令，RF的状态也为1。

A.3 其它调试功能

断点设施是80386为调试程序提供的最重要的功能，但并不是唯一的功能，80386的其它调试功能如下述。

A.3.1 TSS中的调试陷阱

每当通过TSS发生任务切换时，TSS中的T位使调试处理程序被调用，这就为调试程序管理某些任务的活动提供了一个方便的方法。DR6中的BT位指示对该位的检测，DR7中对该位没有特别的启用位。

如果调试处理程序是通过任务门使用的，则不能设置对应TSS的调试陷阱位。否则，将发生调用调试处理程序的无限循环。

A.3.2 INT3

一个断点指令提供调试程序的另一种方法。按这种方法，要求作为断点指令的第一个字节用INT3指令替代。因此，程序执行到预先需要的断点处遇到断点指令，并进入INT3处理程序。在一些使用INT3显得不足的地方，还需使用断点寄存器。这样的情况有：

(1) 由ROM提供的代码中，不可能插入INT3指令。

(2) 由于使用了INT3，原来的程序代码被修改，使执行此代码的其它任务也被中断。

(3) INT3不能执行数据断点。

在另外一些情形，使用INT3则很有用：

(1) 单步及断点设施仅仅进入调试处理程序，而对调试处理程序的调试，INT3则是唯一方便的方法。

(2) 代码中可以插入任意数量的INT3指令，而断点设施只能提供最多四个断点。

(3) 早期86系列的各种型号处理器，没有80386提供的断点设施，INT3指令在这些处理器中是执行任何断点的唯一方法。

概括地说，除了某些特别情形之外，建议使用INT3指令在代码中执行断点，保留断点寄存器用于数据断点。

A.3.3 程序的步进执行

单步功能对程序调试者来说，是一个方便的调试手段。通过一条一条地执行指令，对操作数据、操作指令及操作结果的观察和分析，可以帮助调试人员判断出执行某一条指令时，是否发生了硬件错误(这时实际结果与应得结果不一致)，或是否有软件逻辑错误(实际结果与应得结果相一致，但与主观认为应得的结果不一致)。80386的单步功能通过陷阱来实现。单步陷阱在EFLAGS寄存器中的TF位置位时启用(见第2章)。在一条指令开始执行时，如有TF=1，则在指令执行的末尾产生调试异常，并进入调试处理程序。在这里，“指令开始执行时，TF=1”这一条件是重要的。有此条件的限制，使TF位置为1的指令不会产生单步陷阱。每次产生单步陷阱之后，在进入调试处理程序之前要将TF位清除。此外，在处理中断或异常时，也清除

TF 位。

如果外部中断与单步中断同时发生，则单步中断被优先处理，并清除 TF。在调试处理程序第一条指令执行之前，如仍有悬挂的中断请求，则响应并处理中断。因此，中断处理是在没有单步启用的情况下完成的。如果希望在中断处理程序中使用单步功能，则需先把中断处理程序的第一条指令设置为断点，当程序运行到断点处停下来之后，再启用单步功能。

附录B　80386,80286和8086之间的比较

本附录介绍8086、80286和80386之间的差别。但不再重复关于寄存器集、内存寻址、指令编码、80286分段、实模式分段以及实模式中断处理等方面的差别,因为它们在第9章中已经讲过了。

B.1　8086和80386的比较

B.1.1　80386比8086增加的指令

表B.1列出了80386比8086增加的指令。

表B.1　80386有的而8086没有的应用指令

指　　令	说　　明
PUSH imm	将立即数压栈
PUSHA and POPA	所有8个通用寄存器的压栈和出栈
IMUL reg, imm	立即数的符号乘法
IMUL reg, reg/mem, imm	立即数的符号乘法
IMUL reg, reg/mem	非特征化的符号乘
RCR/RCL reg/mem, imm	循环次数为立即数
ROR/ROL reg/mem, imm	循环次数为立即数
SAL/SAR reg/mem,imm	移位次数为立即数
SHL/SHR reg/men, imm	移位次数为立即数
INS/OUTS	字符串的输入输出(通过端口)
ENTER/LEAVE	过程的进入和退出
BOUND	数组边界检验
LSS/LFS/LGS	段加载指令
Jcc 32-位偏移量	长偏移量的控制的条件转移
BT/BTC/BTR/BTS	位操作
BSF/BSR	位扫描
SHRD/SHLD	双宽移位指令
SETcc	字节的条件置1
MOVSX/MOVZX	传送并符号/零扩展
MOV DRx, reg; reg, DRx	与调试寄存器之间的传送
MOV CRx, reg; reg, CRx	与控制寄存器之间的传送
FS and GS prefixes	FS和GS的段前缀

80386具有的有关保护模式(4级保护、描述符、描述符表、任务和门)和分页机制及支持它们的指令,在8086里统统没有。表B.2列出了8086没有的80386的保护模式指令。

除了这些新的指令以外,所有作用于段寄存器的指令(PUSH、POP、MOV、LES、LDS)和所有改变控制流的指令(CALL、INT、INTO、IRET、JMP、RET)在80386保护模式下的语义

和它们在 8086 下的语义也完全不同了。

表 B.2　8086 没有的 80386 保护模式指令

指　令	说　明	指　令	说　明
ARPL	调整请求特权级	LTR	装入任务寄存器
CLTS	清任务切换标志	SGDT	存储全局描述符表
LAR	装入存取权字节	SLDT	存储局部描述符表
LGDT	装入全局描述符表	SIDT	存储中断描述符表
LIDT	装入中断描述符表	SMSW	存储机器状态字
LLDT	装入局部描述符表	STR	存储任务寄存器
LMSW	装入机器状态字	VERR	检验要读的段
LSL	装入段界限	VERW	检验要写的段

B.1.2　8086 和 80386 的某些指令在执行上的差别

1. 除法异常

对于除法出错，8086 的 CS：IP 指向下一条指令。而 80386 的 CS：EIP 指向引起异常的指令。

2. 无定义的操作码

在 8086 没有 定义的操作码，在 80386 中或者引起无效操作码异常(异常 6)，或者执行一条新的指令。

3. PUSH SP 的值

由 PUSH SP 指令压栈的值，对于 8086 为算后增量形式，而对于 80386 则为算前增量形式。

4. 移位和循环

8086 的所有移位和循环指令的移位计数是取 8 位(二进制)，而 80386 则取 5 位。

5. 冗余前缀

8086 不限制指令的长度，而 80386 限制指令的长度不大于 15 字节。于是，冗余的前缀可能导致非法操作码异常。

6. 超出 64K 的访问

在 64K 段边界上，8086 将地址重新定为 0。所以假若段为 64K，就会引起边界违反，这和在 80386 实地址模式下是一样的。对内存操作数的访问，以及对连续代码的访问，都会有这种情况出现。

7. LOCK

80386 对 LOCK 前缀只限用于某些指令(详见第 3 章)。而 8086 对 LOCK 的使用没有限制。

8. 单步

单步异常的优先级高于外部中断。所以，外部中断处理程序不会执行单步。

9. 标志寄存器

8086 和 80386 的标志寄存器的 12 位到 15 位(4 位)是不一样的，详见第 2 章。

10. 不可屏蔽中断

80386 中 NMI 不能中断 NMI 处理程序，而 8086 能。

11. 协处理器错

80386 协处理器出错使用向量号 16。8086 可以使用任何向量号。

12. 协处理器指令的前缀

当一个协处理器异常处理程序被调用时，被 8086 保存的 CS：IP 值不会保留前缀（如果该值本来有前缀，也丢失了）。80386 则指向协处理器指令的开头，若有前缀，予以保留。

13. 一兆字节的处理

8086 仅有 20 位（一兆）物理地址空间。如果物理地址（段基址＋偏移量）超出一兆，只截取 20 位（一兆）以内的地址值。而 80386，因有 32 根地址线，不会发生因超过一兆字节而截短物理地址。

14. 重复的串操作指令

对于重复的串操作指令，被 8086 保存的 CS：IP 将丢失它的前缀（如果它有前缀的话）。80386 则指向重复的串操作指令的开关，并包括前缀。

B.2 80286 和 80386 的比较

B.2.1 80386 增加的指令

80286 和 80386 的最重要差别在于：

- 32 位地址线
- 数据类型
- 分页式内存管理。

为了支持以上功能，80386 增加了好几条指令。同时 80386 还增加了若干应用指令。在多段和操作系统方面的差别我们已在第 9 章讲过。表 B.3 列出了 80386 新增的指令。

表 B.3 80386 较之 80286 新增的指令

指　　令	说　　明
LSS/LFS/LGS	段加载指令
Jcc 32-位偏移量	长偏移量条件跳转
BT/BTC/BTR/BTS	位操作
BSF/BSR	位扫描
SHRD/SHLD	双宽度移位
SETcc	根据条件设置字节
MOV SX/MOV ZX	传送并符号/零扩展
IMUL reg，reg/mem	无特征符号乘法
MOV DRx，reg；reg，DRx	与调试寄存器之间的传送
MOV CRx，reg；reg，CRx	与控制寄存器之间的传送
FS and GS prefixes	FS 和 GS 的段前缀

B.2.2 80286 和 80386 在执行方面的差别

LOCK：80386 限制 LOCK 前缀仅用于某些指令（请看第 3 章）。而 80286 不限制 LOCK 的使用。

附录C　80387,80287和8087之间的比较

C.1　80287(和8087)与80387的比较

本节列出了80287和80387之间的差别,这些差别也同样存在于8087和80387之间。而下一节(C.2)介绍的是仅存在于8087与80387之间的差别,但它们不存在于80287和80387之间。

C.1.1　新增指令

表C.1给出了80387较80287/8087新增的指令。

表C.1　80387较80287/8087新增的指令

指　　令	说　　明
FPREM1	部分余数——IEEE
FUCOM/FUCOMP/FUCOMPP	无序比较
FSIN	正弦
FCOS	余弦
FSINCOS	正弦和余弦

C.1.2　指令执行

除了表B.1中的新增指令外,在某些指令的执行上还有下列一些差别。

1. 参数(自变量)的范围扩大

FSCALE、FPTAN、FPATAN和FZXM1的参数的有效值范围扩大了。详见第3章。

2. FPTAN

状态字的第10位(C2)置为1,表示FPTAN并未最后完成,而80287/8087对该位未作定义。

3. 非规格数(Denormal)操作

80387的FBSTP、FDIV、FIST、FISTP、FPREM和FSQRT指令可对非规格数进行操作,而80287/8087不能。

4. FPREM

在80287/8087中条件码(CC3—CC0)不能可靠地设置起来,而80387能。

5. FLD扩展的实型非规格数80387中非规格数不产生非规格数异常,而80287/8087会产生异常。

6. FXTRACT

80387中操作数0则产生零除异常并将ST(1)置为负无穷,而80287/8087则将ST(1)置0并不报告异常。80387对于正无穷不产生异常,而80287/8087则产生无效操作异常。

7. FLD常数

80387 进行舍入控制，而 80287/8087 不进行。

8. FLD 短实型数/长实型数

对于 80387，一个非规格数(Denormal)的装入，使得该数转换为扩展精度；而 80287/8087 则将该数转换成未规格数(Unnormal)。当装入一信号的 NaN，80387 会产生无效操作数异常，而 80287/8087 不会。

9. FSETPM

FSETPM 为 80287 的指令，和 80387 的 FNOP 指令相当。

10. 超越函数

由 80387 产生状态字的 round-up 位，而 80287/8087 对此无定义。

11. 其它

下面列出的差别，不是和某个实际指令有关，而是和初始化、数据类型、异常报告、以及标记字、状态字、控制字有关。

(1) 复位 RESET

80386 在复位时，状态字的 IE 和 ES 位置位，控制字的 IM 位复位，以报告 80387 的存在。据此，80386 可判断出存在的协处理器到底是 80387 还是 80287。

(2) 静的 NaN

80287 只有一种 NaN，即信号 NaN。而 80386 还支持静的 NaN，不会因遇上静的 NaN 而产生异常。

(3) 不支持的格式

无论遇到什么样的不支持的格式，80387 都会产生一个非法的操作码异常。这些不支持的格式为：伪 NaN、伪 0、伪无穷数、和未规格数。而 80287/8087 支持上述格式，若遇上这些格式也不产生异常。同时，80287 定义伪 0 和未规格数为有效(标记字为 00b)，而 80387 将伪 0 和未规格数划为特殊数据(标记字为 106)。

(4) 有效操作

那些会使 80287/8087 产生无效操作异常的 FSQRT、FDIV 以及 FPREM 操作，或者转换为 BCD 码或整数的操作，并不能使 80387 产生异常。

(5) 非规格数异常

80387 中超越指令及 FXTRACT 指令会产生非规格数异常，而 80287 不会。

(6) 溢出异常

当舍入模式置为截短，80387 对于溢出异常的屏蔽响应是最正的或最负的有效数，而 80287/8087 则为正负无穷数。若溢出异常未被屏蔽，80387 将标出精度误差，而 80287 不标。同时，若不屏蔽，80387 存入堆栈的结果将根据精度控制或操作码进行舍入，而 80287/8087 则不舍入。

(7) 下溢异常

若下溢异常被屏蔽，80387 只有在下溢引起精度丢失时才标出下溢异常。而 80287 则在舍入控制为向零的方向舍入的情况下才会标志下溢异常，不管该下溢是否引起了精度丢失。

如果下溢未被屏蔽，而且结果要被存入堆栈，则 80387 的结果要按照精度控制或操作码的要求进行舍入，而 80287/8087 的结果不要进行舍入。

(8) 异常的优先级

80387 的非规格化异常(the denormalization exception)，不管是屏蔽的或不屏蔽的，都有

相同的优先级。而 80287/8087 的非规格化异常，当它未屏蔽时，它的优先级要超过所有其它的异常。这样，80387 省去了不必要的非规格化数的规格化过程。

(9) 无穷数的控制

80387 只支持无穷数的仿射解释。而 80287/8087 则支持无穷数的仿射解释和投影解释。

(10) 堆栈的溢出和下溢

在堆栈故障的情况下，80387 除了以状态字的位 0 来表示无效异常外，还将状态字的位 6 置位，并且用位 10 的置位来表示堆栈的下溢。而 80287/8087 则仅反映无效异常。

(11) 状态字的条件位

FINIT 指令、未完成的 FPREM 指令及硬件复位，可以使 80387 状态字的条件位复位，而不能修改 80287 的条件位。

(12) 标记字

在执行 FLDENV 和 FRSTOR 指令时，装入标记字的值和寄存器中的值不一致是有可能的。80387 只相信标记字的空/非空位(值 11 表示空，而 00.01 和 10 表示非空)，而且在所有其它的情况下只分析非空的寄存器内容。而 80287/8087 依赖标记值决定操作数类型，但不分析寄存器内存。

C.2 8087 和 80387 的比较

除了上述 80287/8087 和 80387 之间的差别外，8087 还有只属于它自己的和 80387 的差别。80387 和 8087 在异常处理方面的差别是它们之间的最主要差别。8087 的异常处理程序，到了 80387 必须更新；而 8087 的应用程序几乎不用修改便能在 80387 上运行。

C.2.1 FENI/FNENI 和 FDISI/FNDISI

80387 忽略了 FENI/FNENI 和 FDISI/FNDISI 指令(不改变 80387 的状态)。用于这些指令的 8087 异常处理程序都不和 80387 兼容。

C.2.2 数字异常

正如附录 A 中所说，8087 和 80387 的异常处理的方法迥然不同。8087 通过中断矢量处理中断，而 80387 将中断处理为中断 16。另外，8087 通过分立的中断控制器来处理中断。而这种处理数字异常的方法在 80387 则是无效的。

C.2.3 异常指针

详见第 9 章，8087 和 80387 的存储指令和数据指针的格式是不一样的。

C.2.4 段

若指令落在段界之外(异常 13)，或者操作数位于段之外(异常 9 和 13)，80387 会产生一个段故障。8087 则不会。

C.2.5 异常

当 80387 执行一浮点指令，可能会产生一指示仿真或任务切换的异常(80386CRO 寄存器

的 EM 和 TS 位)。而 8087 不会。

C.2.6 协处理器指令的前缀

当一个协处理器异常处理器被调用,被 8087 保存的 CS:IP 的值不包含前缀,尽管这个前缀存在。而 80387 则指向包含前缀(若有前缀的话)的协处理器指令的开始。

C.2.7 WAIT 指令

在第 3 章中说过,80387 和 80386 会自动同步,而不要求编程者用显式的 WAIT 指令去强制同步(除了在将数据从 80387 存入内存的指令之后)。虽然,含有不必要的 WAIT 指令的程序能正常运行,但对 80387 来说,无论在代码空间或者执行时间上,都是一种浪费。

附录 D　二进制、十六进制、十进制数对照表

十六进制	二进制	十进制	十六进制	二进制	十进制
0	0	0	A	1010	10
1	1	1	B	1011	11
2	10	2	C	1100	12
3	11	3	D	1101	13
4	100	4	E	1110	14
5	101	5	F	1111	15
6	110	6	10	10000	16
7	111	7	11	10001	17
8	1000	8	⋮	⋮	⋮
9	1001	9			

附录 E　以 2 为底的幂函数

幂函数	十进制的值	幂函数	十进制的值	幂函数	十进制的值
2^0	1	2^{11}	2,048	2^{22}	4,194,304
2^1	2	2^{12}	4,096	2^{23}	8,388,608
2^2	4	2^{13}	8,192	2^{24}	16,777,216
2^3	8	2^{14}	16,384	2^{25}	33,554,432
2^4	16	2^{15}	32,768	2^{26}	67,108,864
2^5	32	2^{16}	65,536	2^{27}	134,217,728
2^6	64	2^{17}	131,072	2^{28}	268,435,456
2^7	128	2^{18}	262,144	2^{29}	536,870,912
2^8	256	2^{19}	524,288	2^{30}	1,073,741,824
2^9	512	2^{20}	1,048,576	2^{31}	2,147,483,648
2^{10}	1,024	2^{21}	2,097,152	2^{32}	4,294,967,296

附录 F　ASCII 码表

低 4 位	高 3 位							
	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	—	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

NUL	空	DLE	数据脱钩
SOH	标题的开始	DC1	设备控制 1
STX	文本开始	DC2	设备控制 2
ETX	文本结束	DC3	设备控制 3
EOT	传送结束	DC4	设备控制 4
ENQ	查询	NAK	负响应
ACK	响应	SYN	同步空闲
BEL	铃响(发声的)	ETB	块传送结束
BS	退格	EM	媒体的尾部
HT	水平制表符	SUB	置换
LF	换行	ESC	换码
VT	垂直制表符	FS	文件分隔符
FF	进页	GS	组分隔符
CR	回车	RS	记录分隔符
SO	移位出	US	单元分隔符
SI	移位进	DEL	删除

附录G　80386操作码表

本附录内的操作码表用来解释80386的目标码。操作码的高4位为该表的行索引；而操作码的列索引则采用操作码的低4位。如操作码为OFH，则查阅双字节操作码表，并利用该操作码的第二个字节去检索该表的行和列。

G.1 缩略语表

操作数用形如Zg的双字符码来标识。第一个为大写的字母规定了寻址方式。第二个为小写字母规定了操作数类型。

G.1.1 表示寻址方式的符号(操作数标识码的第一个字母)

A　直接寻址；该指令没有MODRM字节；操作数的地址在指令中；没有可利用的基址寄存器，索引寄存器，或比例因子；例如远程的跳转JMP(EA)。

C　MODRM字节中的reg字段选择的是控制寄存器；例如MOV(0F20,0F22)。

D　MODRM字节中的reg字段选择的是调试寄存器；例如MOV(0F21,0F23)。

E　跟在操作码后面的MODRM字节规定了操作数。该操作数可以是一个通用寄存器，也可以是一个内存地址。如果操作数是一个内存地址，这个地址是根据段寄存器和下面四个值当中的任一个或某种组合计算出来的：基址寄存器，变址寄存器，比例因子，位移量。

F　标志寄存器。

G　MODRM字节中的reg字段选择的是通用寄存器；例如ADD(00)。

I　立即数。操作数的值是在指令的后续字节之中。

J　指令中包含一个被加到指令指针寄存器的相对偏移量；如短转移指令JMP short，LOOP。

M　MODRM规定只能访问内存；BOUND,LES,LDS,LSS,LFS,LGS。

O　指令没有MODRM字节；操作数的位移量含在指令编码之中，视地址的尺寸属性，或为字，或为双字。没有可用的基址寄存器、变址寄存器或比例因子。如MOV (AO—A3)。

R　MODRM字节中的mod字段规定只能访问通用寄存器；如MOV (0F20- 0F24, 0F26)。

S　MODRM字节中的reg字段选择的是段寄存器；如MOV(8C,8E)。

T　MODRM字节的reg字段选择的是测试寄存器；如MOV (0F24,0F26)

X　内存地址由DS：SI决定；如MOVS,COMPS,OUTS,LODS,SCAS。

Y　内存地址由ES：DI决定；如MOVS, CMPS,INS,STOS。

G.1.2 表示操作类型的符号

a	内存中的两个单字操作数或两个双字操作数(取决于操作数的尺寸属性,只用于BOUND)。
b	字节(与操作数的尺寸属性无关)。
c	字节或字,取决于操作数的尺寸属性。
d	双字(与操作数的尺寸属性无关)。
p	32 位或 48 位指针,取决于操作数的尺寸属性。
s	6 字节伪描述符
v	字或双字,取决于操作数的尺寸属性。
w	字(与操作数尺寸属性无关)。

G.1.3 表示寄存器的符号

如果在操作码中明确规定了操作数是一个具体的寄存器,则直接标出这个寄存器的名字:如 AX,CL,ESI 等等。该寄存器的名字已经能够表示它究竟是 32 位的、16 位的,还是 8 位的。如果该寄存器的大小还取决于操作数的尺寸属性的话,那么我们就把它表示为 eXX。例如 eAX 表示:若操作数尺寸为 16 位的话则用 AX,若操作数尺寸为 32 位的话则用 EAX。

G.2 80386 操作码表

单字节操作码表

	0	1	2	3	4	5	6	7
0	ADD						PUSH	POP
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	ES	ES
1	ADC						PUSH	POP
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	SS	SS
2	AND						SEG	DAA
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=ES	
3	XOR						SEG	AAA
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=SS	
4	INC 通用寄存器							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH 通用寄存器							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	PUSHA	POPA	BOUND	ARPL	SEG	SEG	Operand	Address
			Gv,Ma	Ew,Rw	=FS	=GS	Size	Size
7	条件短转移(Jb)							
	JO	JNO	JB	JNB	JZ	JNZ	JBE	JNBE
8	立即数第 1 组			GrpI	TEST		XCHG	
	Eb,Ib	Ev,Iv		Ev,Ib	Eb,Gb	Ev,Gv	Eb,Gb	Ev,Gv
9	NOP	XCHG 字或双字寄存器与 eAX 之间						
		eCX	eDX	eBX	eSP	eBP	eSI	eDI
A	MOV				MOVSB	MOVSW/D	CMPSB	CMPSW/D
	AL,Ob	eAx,Ov	Ob,AL	Ov,eAX	Xb,Yb	Xv,Yv	Xb,Yb	Xv,Yv
B	MOV 直接数字节到字节寄存器							
	AL	CL	DL	BL	AH	CH	DH	BH
C	移位第 2 组		RET 近		LES	LDS	MOV	
	Eb,Ib	Ev,Ib	Iw		Gv,Mp	Gv,Mp	Eb,Ib	Ev,Iv
D	移位第 2 组				AAM	AAD		XLAT
	Eb,1	Ev,1	Eb,CL	Ev,CL				
E	LOOPNE	LOOPE	LOOP	JCXZ	IN		OUT	
	Jb	Jb	Jb	Jb	AL,Ib	eAX,Ib	Ib,AL	Ib,eAX
F	LOCK		REPNE	REP	HLT	CMC	一元的第 3 组	
				REPE			Eb	Ev

单字节操作码表

	8	9	A	B	C	D	E	F
0	OR						PUSH CS	2-byte escape
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
1	SBB						PUSH DS	POP DS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
2	SUB						SEG =CS	DAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
3	CMP						SEG =CS	AAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
4	DEC 通用寄存器							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	POP 至通用寄存器							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	PUSH Ib	IMUL GvEvIv	PUSH Ib	IMUL GvEvIb	INSB Yb,DX	INSW/D Yv,DX	OUTSB DX,Xb	OUTSW/D DX,Xv
7	条件短转移(Jb)							
	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
8	MOV				MOV Ew,Sw	LEA Gv,M	MOV Sw,Ew	POP Ev
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev				
9	CBW	CWD	CALL Ap	WAIT	PUSHF Fv	POPF Fv	SAHF	LAHF
A	TEST		STOSB Yb,AL	STOSW/D Yv,eAX	LODSB AL,Xb	LODSW/D eAX,Xv	SCASB AL,Xb	SCASW/D eAX,Xv
	AL,Ib	eAX,Iv						
B	MOV 直接数字或双字到字或双字寄存器							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
C	ENTER Iw,Ib	LEAVE	RET far		INT 3	INT Ib	INTO	IRET
			Iw					
D	ESC 转到协处理器指令集							
E	CALL Av	JMP			IN		OUT	
		Jv	Ap	Jb	AL,DX	eAX,DX	DX,AL	DX,eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC 第4组	间接 第5组

双字节操作码表(第一个字节为 0FH)

	0	1	2	3	4	5	6	7
0	第 6 组	第 7 组	LAR Gv,Ew	LSL Gv,Ew			CLTS	
1								
2	MOV Cd,Rd	MOV Dd,Rd	MOV Rd,Cd	MOV Rd,Dd	MOV Td,Rd		MOV Rd,Td	
3								
4								
5								
6								
7								
8	长距离的条件转移(Jv)							
	JO	JNO	JB	JNB	JZ	JNZ	JBE	JNBE
9	条件置字节(Eb)							
	SETO	SETNO	SETB	SETNB	SETZ	SETNZ	SETBE	SETNBE
A	PUSH FS	POP FS		BT Ev,Gv	SHLD EvGvIb	SHLD EvGvCL		
B			LSS Mp	BTR Ev,Gv	LFS Mp	LGS Mp	MOVZX Gv,Eb	MOVZX Gv,Ew
C								
D								
E								
F								

双字节操作码表(第一个字节为 0FH)

	8	9	A	B	C	D	E	F
0								
1								
2								
3								
4								
5								
6								
7								
8	长距离的条件转移(Jv)							
	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
9	SETS	SETNS	SETP	SETNP	SETL	SETNL	SETLE	SETNLE
A	PUSH GS	POP GS		BTS Ev,Gv	SHRD EvGvIb	SHRD EvGvCL		IMUL Gv,Ev
B			第 8 组 Ev,Ib	BTC Ev,Gv	BSF Gv,Ev	BSR Gv,Ev	MOVSX	
							Gv,Eb	Gv,Ew
C								
D								
E								
F								

由 MODRM 的 5,4,3 位决定的操作码

mod	nnn	R/M

组号	000	001	010	011	100	101	110	111
1	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
2	ROL	ROR	RCL	RCR	SHL	SHR		SAR
3	TEST Ib/Iv		NOT	NEG	MUL AL/eAX	IMUL AL/eAX	DIV AL/eAX	IDIV AL/eAX
4	INC Eb	DEC Eb						
5	INC Ev	DEC Ev	CALL Ev	CALL Ep	JMP Ev	JMP Ep	PUSH Ev	
6	SLDT Ew	STR Ew	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
7	SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Ew		LMSW Ew	
8					BT	BTS	BTR	BTC

附录 H　80386 指令格式和时钟周期次数

表 H.1 给出了所有指令的编码图和时钟周期次数。随后是指令编码的详细说明，完整地描述了 80386 的编码结构和 80386 指令中所有字段的定义。

H.1　80386 指令编码和时钟周期次数

表 H.1 中给出的时钟周期次数除以 80386 的工作频率，等于该指令的执行时间。例子请看第 3 章的指令集。

H.1.1　80386 的指令摘要及时钟周期次数

表 H.1　80386 的指令摘要及时钟周期次数

指令	格式	时钟周期次数：实地址模式或虚拟 8086 模式	时钟周期次数：保护的虚地址模式	注：实地址模式或虚拟 8086 模式	注：保护的虚地址模式
通用数据传送					
MOV＝传送：					
寄存器到寄存器/内存	1 0 0 0 1 0 0 w　mod reg r/m	2/2	2/2	b	h
寄存器/内存到寄存器	1 0 0 0 1 0 1 w　mod reg r/m	2/4	2/4	b	h
立即数到寄存器/内存	1 1 0 0 0 1 1 w　mod 0 0 0 r/m　立即数	2/2	2/2	b	h
立即数到寄存器(短格式)	1 0 1 1 w reg　立即数	2	2		
内存到累加器(短格式)	1 0 1 0 0 0 0 w　全位移量	4	4	b	h
累加器到内存(短格式)	1 0 1 0 0 0 1 w　全位移量	2	2	b	h
寄存器/内存到段寄存器	1 0 0 0 1 1 1 0　mod sreg3 r/m	2/5	18/19	b	h,i,j
段寄存器到寄存器/内存	1 0 0 0 1 1 0 0　mod sreg3 r/m	2/2	2/2	b	h
MOVSX＝带符号扩展的传送					

续表

指令	格式	时钟周期次数：实地址模式或虚拟8086模式	时钟周期次数：保护的虚地址模式	注：实地址模式或虚拟8086模式	注：保护的虚地址模式
从寄存器/内存到寄存器					
	00001111 | 1011111w | mod reg r/m	3/6	3/6	b	h
MOVZX=带零扩展的传送					
从寄存器/内存到寄存器					
	00001111 | 1011011w | mod reg r/m	3/6	3/6	b	h
PUSH=压栈					
寄存器/内存	11111111 | mod 110 r/m	5	5	b	h
寄存器(短格式)	01010 reg	2	2	b	h
段寄存器(ES、CS、SS或DS)(短格式)					
	000sreg2 110	2	2	b	h
段寄存器(ES、CS、SS、DS、FS或GS)					
	00001111 | 10 sreg3 000	2	2	b	h
立即数	011010s0 立即数	2	2	b	h
PUSHA=所有通用寄存器压栈					
	01100000	18	18	b	h
POP=从栈中弹出					
寄存器/内存	10001111 | mod 000 r/m	5	5	b	h
寄存器(短格式)	01011 reg	4	4	b	h
段寄存器(ES、CS、SS或DS)(短格式)					
	000 sreg2 111	7	21	b	h,i,j
段寄存器(ES、CS、SS、DS、FS或GS)					
	00001111 | 10 sreg 3 001	7	21	b	h,i,j
POPA=弹出至所有寄存器					
	01100001	24	24	b	h
XCHG=交换					
寄存器/内存和寄存器					
	1000011w | mod reg r/m	3/5	3/5	b,f	f,h
寄存器和累加器(短格式)	10010 reg 时钟周期数虚拟8086模式	3	3		
IN=从端口输入：					
固定的端口	1110010w | 端口号 | +26	12	6*/26**		m
可变的端口	1110110w | +27	13	7*/27**		m
OUT=向端口输出：					
固定的端口	1110011w | 端口号 | +24	10	4*/24**		m
可变的端口	1110111w | +25	11	5*/25**		m
LEA=将有效地址装入通用寄存器					
	10001101 | mod reg r/m	2	2		

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
段控制					
LDS=将指针装入DS	11000101 mod reg r/m	7	22	b	h,i,j
LES=将指针装入ES	11000100 mod reg r/m	7	22	b	h,i,j
LFS=将指针装入FS	00001111 10110100 mod reg r/m	7	25	b	h,i,j
LGS=将指针装入GS	00001111 10110101 mod reg r/m	7	25	b	h,i,j
LSS=将指针装入SS	00001111 10110010 mod reg r/m	7	22	b	h,i,j
标志控制					
CLC=清进位标志	11111000	2	2		
CLD=清方向标志	11111100	2	2		
CLI=清中断启用标志	11111010	3	3		m
CLTS=清任务切换标志	00001111 00000110	5	5	c	l
CMC=进位标志的反码	11110101	2	2		
LAHF=将标志装入AH寄存器	10011111	2	2		
POPF=弹出至标志寄存器	10011101	5	5	b	h,n
PUSHF=将标志寄存器压栈	10011100	4	4	b	h
SAHF=将AH装入标志	10011110	3	3		
STC=置进位标志	11111001	2	2		
STD=置方向标志	11111001	2	2		
STI=置中断启用标志	11111011	3	3		m
算术指令					
ADD=加					

续表

指令	格式			时钟周期次数		注	
				实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
寄存器到寄存器	000000dw	mod reg r/m		2	2		
寄存器到内存	0000000w	mod reg r/m		7	7	b	h
内存到寄存器	0000001w	mod reg r/m		6	6	b	h
立即数到寄存器/内存							
	100000sw	mod 000 r/m	立即数	2/7	2/7	b	h
立即数到累加器(短格式)							
	0000010w	立即数		2	2		
ADC=带进位的加法							
寄存器到寄存器	000100dw	mod reg r/m		2	2		
寄存器到内存	0001000w	mod reg r/m		7	7	b	h
内存到寄存器	0001001w	mod reg r/m		6	6	b	h
立即数到寄存器/内存							
	100000sw	mod 010, r/m	立即数	2/7	2/7	b	h
立即数到累加器(短格式)							
	0001010w	立即数		2	2		
INC=增量							
寄存器/内存	1111111w	mod 000 r/m		2/6	2/6	b	h
寄存器(短格式)	01000 reg			2	2		
SUB=减							
从寄存器减寄存器							
	001010dw	mod reg r/m		2	2		
从内存减寄存器	0010100w	mod reg r/m		7	7	b	h
从寄存器减内存	0010101w	mod reg r/m		6	6	b	h
从寄存器/内存减立即数							
	100000sw	mod 101 r/m	立即数	2/7	2/7	b	h
从累加器减立即数(短格式)							
	0010110w	立即数		2	2		
SBB=带借位的减法							
从寄存器减寄存器							
	000110dw	mod reg r/m		2	2		
从内存减寄存器	0001100w	mod reg r/m		7	7	b	h
从寄存器减内存	0001101w	mod reg r/m		6	6	b	h
从寄存器/内存减立即数							
	100000sw	mod 011 r/m	立即数	2/7	2/7	b	h
从累加器减立即数(短格式)							
	0001110w	立即数		2	2		
DEC=减量							

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
寄存器/内存	1111111w | reg 001 r/m	2/6	2/6	b	h
寄存器(短格式)	01001 reg	2	2		
CMP=比较					
寄存器与寄存器	001110dw | mod reg r/m	2	2		
内存与寄存器	0011100w | mod reg r/m	5	5	b	h
寄存器与内存	0011101w | mod reg r/m	6	6	b	h
立即数与寄存器/内存					
	100000sw | mod 111 r/m 立即数	2/5	2/5	b	h
立即数与累加器(短格式)					
	0011110w 立即数	2	2		
NEG=改变符号	1111011w | mod 011 r/m	2/6	2/6	b	h
AAA=加法之后的ASCII调整					
	00110111	4	4		
AAS=减法之后的ASCII调整					
	00111111	4	4		
DAA=加法之后的十进制调整					
	00100111	4	4		
DAS=减法之后的十进制调整					
	00101111	4	4		
MUL=无符号乘法:					
累加器与寄存器/内存					
	1111011w | mod 100 r/m				
乘数·字节		9-14/12-17	9-14/12-17	b,d	d,h
·字		9-22/12-25	9-22/12-25	b,d	d,h
·双字		9-38/12-41	9-38/12-41	b,d	d,h
IMUL=符号整数乘法					
累加器和寄存器内存					
	1111011w | mod 100 r/m				
乘数·字节		9-14/12-17	9-14/12-17	b,d	d,h
·字		9-22/12-25	9-22/12-25	b,d	d,h
·双字		9-38/12-41	9-38/12-41	b,d	d,h
寄存器与寄存器/内存					
	00001111 | 10101111 | mod reg r/m				
·字		9-22/12-25	9-22/12-25	b,d	d,h
·双字		9-38/12-41	9-38/12-41	b,d	d,h
寄存器/内存与立即数、送寄存器					
	011010sl | mod reg r/m 立即数				

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
·字		9-22/12-25	9-22/12-25	b,d	d,h
·双字		9-38/12-41	9-38/12-41	b,d	d,h
DIV=无符号除法					
累加器被寄存器/内存除					
	1111011w mod110r/m				
除数—字节		14/17	14/17	b,c	e,h
字		22/25	22/25	b,e	e,h
—双字		38/41	38/41	b,e	e,h
IDIV=符号整数除法					
累加器被寄存器/内存除					
	1111011w mod111r/m				
除数 字节		19/22	19/22	b,e	e,h
—字		27/30	27/30	b,e	e,h
—双字		43/46	43/46	b,e	e,h
AAD=除法之前的ASCII调整					
	11010101 00001010	19	19		
AAM=乘法之后的ASCII调整					
	11010100 00001010	17	17		
CBW=将字节转换成字					
	10011000	3	3		
CWD=将字转换成双字					
	10011001	2	2		
逻辑指令					
移位循环指令					
不通过进位标志(ROL,ROR,SAL,SAR,SHL,SHR)					
寄存器/内存,次数为1					
	1101000w mod TTT r/m	3/7	3/7	b	h
寄存器/内存,次数为CL					
	1101001w mod TTT r/m	3/7	3/7	b	h
寄存器/内存,次数为立即数					
	1100000w mod TTT r/m 8位立即数	3/7	3/7	h	h
通过进位标志(RCL,RCR)					
寄存器/内存,次数为1					
	1101000w mod TTT r/m	9/10	9/10	b	h
寄存器/内存,次数为CL					
	1101001w mod TTT r/m	9/10	9/10	b	h

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
寄存器/内存,次数为立即数					
	1 1 0 0 0 0 0 w | mod TTT r/m | 8位立即数	9/10	9/10*	b	h
	TTT 指令				
	0 0 0 ROL				
	0 0 1 ROR				
	0 1 0 RCL				
	0 1 1 RCR				
	1 0 0 SHL/SAL				
	1 0 1 SHR				
	1 1 1 SAR				
SHLD=双宽度左移					
寄存器/内存,次数为立即数					
	0 0 0 0 1 1 1 1 | 1 0 1 0 0 1 0 0 | mod reg r/m | 8位立即数	3/7	3/7		
寄存器/内存,次数为CL					
	0 0 0 0 1 1 1 1 | 1 0 1 0 0 1 0 1 | mod reg r/m	3/7	3/7		
SHRD=双宽度右移					
寄存器/内存,次数为立即数					
	0 0 0 0 1 1 1 1 | 1 0 1 0 1 1 0 0 | mod reg r/m | 8位立即数	3/7	3/7		
寄存器/内存,次数为CL					
	0 0 0 0 1 1 1 1 | 1 0 1 0 1 1 0 1 | mod reg r/m	3/7	3/7		
AND=与					
寄存器到寄存器					
	0 0 1 0 0 0 d w | mod reg r/m	2	2		
寄存器到内存	0 0 1 0 0 0 0 w | mod reg r/m	7	7	b	h
内存到寄存器	0 0 1 0 0 0 1 w | mod reg r/m	6	6	b	h
立即数到寄存器/内存					
	1 0 0 0 0 0 0 w | mod 1 0 0 r/m | 立即数	2/7	2/7	b	h
立即数到累加器(短格式)					
	0 0 1 0 0 1 0 w | 立即数	2	2		
TEST="与",影响标志,结果丢弃					
寄存器/内存和寄存器					
	1 0 0 0 0 1 0 w | mod reg r/m	2/5	2/5	b	h
立即数和寄存器/内存					
	1 1 1 1 0 1 1 w | mod 0 0 0 r/m | 立即数	2/5	2/5	b	h

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
立即数和累加器(短格式)					
	1010100w 立即数	2	2		
OR=或					
寄存器到寄存器					
	000010dw mod reg r/m	2	2		
寄存器到内存	0000100w mod reg r/m	7	7	b	h
内存到寄存器	0000101w mod reg r/m	6	6	b	h
立即数到寄存器/内存					
	1000000w mod 001 r/m 立即数	2/7	2/7	b	h
立即数到累加器(短格式)					
	0000110w 立即数	2	2		
XOR=异或					
寄存器到寄存器					
	001100dw mod reg r/m	2	2		
寄存器到内存	0011000w mod reg r/m	7	7	b	h
内存到寄存器	0011001w mod reg r/m	6	6	b	h
立即数到寄存器/内存					
	1000000w mod 110 r/m 立即数	2/7	2/7	b	h
立即数到累加器(短格式)					
	0011010w 立即数	2	2		
NOT=进制反码串操作指令					
	1111011w mod 010 r/m 时钟周期次数 虚拟8086模式	2/6	2/6	b	h
CMPS=比较字节/字					
	1010011w	10	10	b	h
INS=从DX口输入字节/字					
	0110110w +29	15	9*/29**	b	h,m
LODS=将字节/字装入AL/AX/EAX					
	1010110w	5	5	b	h
MOVS=传送字节/字					
	1010010w	7	7	b	h
OUTS=将字节/字输出至DX口					
	0110111w +28	14	8*/28**	b	h,m
SCAS=扫描字节/字					
	1010111w	7	7	b	h
STOS=将字节/字从AL/AX/EAX存入内存					
	1010101w	4	4	b	h

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
XLAT=查表转换					
	11010111	5	5		h
重复串操作指令，重复次数由CX或ECX内容决定					
REPE CMPS=比较字符串(发现不匹配字符)					
	11110011 10100111w	5+9n	5+9n	b	h
REPNE CMPS=比较字符串(发现匹配的字符)	时钟周期次数				
	11110010 1010011w 虚拟8086模式	5+9n	5+9n	b	h
REP INS=输入字符串					
	11110010 0110110w +27+6n	13+6n	7+6n*/27+6n**	b	h,m
REP LODS=装入字符串					
	11110010 1010110w	5+6n	5+6n	b	h
REP MOVS=传送字符串					
	11110010 1010010w	7+4n	7+4n	b	h
REP OUTS=输出字符串					
	11110010 0110111w +26+5n	12+5n	6+5n*/26+5n**	b	h,m
REPE SCAS=扫描字符串					
(发现不是AL/AX/EAX)					
	11110011 1010111w	5+8n	5+8n	b	h
REPNE SCAS=扫描字符串					
(发现是AL/AX/EAX)					
	11110010 1010111w	5+8n	5+8n	b	h
REP STOS=存储字符串					
	11110010 1010101w	5+5n	5+5n	b	h
位操作指令					
BSF=前向位扫描					
	00001111 10111100 mod reg r/m	10+3n	10+3n	b	h
BSR=反向位扫描					
	00001111 10111101 mod reg r/m	10+3n	10+3n	b	h
BT=位测试					
寄存器/内存，立即数					
	00001111 10111010 mod 100 r/m 8位立即数	3/6	3/6	b	h
寄存器/内存，寄存器					
	00001111 10100011 mod reg r/m	3/12	3/12	b	h
BTC=位测试并求反					
寄存器/内存，立即数					
	00001111 10111010 mod 111 r/m 8位立即数	6/8	6/8	b	h

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
寄存器/内存,寄存器					
	00001111 | 10111011 | mod reg r/m	6/13	6/13	b	h
BTR=位测试并复位					
寄存器/内存,立即数					
	00001111 | 10111010 | mod 110 r/m | 8位立即数	6/8	6/8	b	h
寄存器/内存,寄存器					
	00001111 | 10110011 | mod reg r/m	6/13	6/13	b	h
BTS=位测试并置位					
寄存器/内存,立即数					
	00001111 | 10111010 | mod 101 r/m | 8位立即数	6/8	6/8	b	h
寄存器/内存,寄存器					
	00001111 | 10101011 | mod reg r/m	6/13	6/13	b	h
控制转移指令					
CALL=调用					
段内直接调用	11101000 全位移量	7+m	7+m	b	r
寄存器/内存					
段内间接调用	11111111 | mod 010 r/m	7+m/10+m	7+m/10+m	b	h,r
段间直接调用	10011010 无符号全偏移量,选择符	17+m	34+m	b	j,k,r
仅保护模式(段间直接调用)					
通过调用门到同一特权级			52+m		h,j,k,r
通过调用门到不同特权级(无参数)			86+m		h,j,k,r
通过调用门到不同特权级(X为参数个数)			94+4x+m		h,j,k,r
从286任务到286 TSS			273		h,j,k,r
从286任务到386 TSS			298		h,j,k,r
从286任务到虚拟8086任务(386 TSS)			217		h,j,k,r
从386任务到286 TSS			273		h,j,k,r
从386任务到386 TSS			300		h,j,k,r
从386任务到虚拟8086任务(386 TSS)			217		h,j,k,r
段间间接	11111111 | mod 011 r/m	22+m	38+m	b	h,j,k,r
仅保护模式(段间间接调用)					
通过调用门到同一特权级			56+m		h,j,k,r
通过调用门到不同特权级(无参数)			90+m		h,j,k,r
通过调用门到不同特权级(X个参数)			98+4x+m		h,j,k,r

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
从286任务到286 TSS			278		h,j,k,r
从286任务到386 TSS			303		h,j,k,r
从286任务到虚拟8086任务(386TSS)			221		h,j,k,r
从386任务到286TSS			278		h,j,k,r
从386任务到386TSS			305		h,j,k,r
从386任务到虚拟8086任务(386TSS)			221		h,j,k,r
JMP=无条件转移					
短跳转	11101001 8位偏移量	7+m	7+m		r
段内直接	11101001 全位移量	7+m	7+m		r
段内,通过寄存器/内存间接					
	11111111 mod 100 r/m	7+m/10+m	7+m/10+m	b	h,r
段间直接	11101010 无符号全偏移量,选择符	12+m	27+m		j,k,r
仅保护模式(段间直接转移)					
通过调用门到同一特权级			45+m		h,j,k,r
从286任务到286TSS			274		h,j,k,r
从286任务到386TSS			301		h,j,k,r
从286任务到虚拟8086任务(386TSS)			218		h,j,k,r
从386任务到286TSS			270		h,j,k,r
从386任务到386TSS			303		h,j,k,r
从386任务到虚拟8086任务(386TSS)			220		h,j,k,r
段间间接	11111111 mod 101 r/m	17+m	31+m	b	h,j,k,r
仅保护模式(段间间接)					
通过调用门到同一特权级			49+m		h,j,k,r
从286任务到286TSS			279		h,j,k,r
从286任务到386TSS			306		h,j,k,r
从286任务到虚拟8086任务(386TSS)			222		h,j,k,r
从386任务到286TSS			275		h,j,k,r
从386任务到386TSS			308		h,j,k,r
从386任务到虚拟8086任务(386TSS)			224		h,j,k,r
RET=从调用返回					
段内	11000011	10+m	10+m	b	g,h,r
段内,SP加立即数					
	11000010 16位位偏量	10+m	10+m	b	g,h,r
段间	11001011	18+m	32+m	b	g,h,j,k,r

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
段间,SP加立即数					
	11001010 | 16位位移量	18+m	32+m	b	g,h,j,k,r
仅保护模式(RET):					
到不同特权级					
段间			68		h,j,k,r
段间,SP加立即数			68		h,j,k,r
条件跳转					
注:时钟周期数给出了"跳与不跳"的两种数据					
JO=溢出则跳(OF=1)					
8位位移量	01110000 | 8位位移量	7+m or 3	7+m or 3		r
全位移量	00001111 | 10000000 全位移量	7+m or 3	7+m or 3		r
JNO=不溢出则跳(OF=0)					
8位位移量	01110001 | 8位位移量	7+m or 3	7+m or 3		r
全位移量	00001111 | 10000001 全位移量	7+m or 3	7+m or 3		r
JB/JNAE=低于/不高于等于时跳(CF=1)					
8位位移量	01110010 | 8位位移量	7+m or 3	7+m or3		r
全位移量	00001111 | 10000010 全位移量	7+m or 3	7+m or 3		r
JNB/JAE=不低于/高于等于则跳(OF=0)					
8位位移量	01110011 | 8位位移量	7+m or 3	7+m or 3		r
全位移量	00001111 | 10000011 全位移量	7+m or 3	7+m or 3		r
JE/JZ=等于/为0则跳(ZF=1)					
8位位移量	01110100 | 8位位偏量	7+m or 3	7+m or 3		r
全位移量	00001111 | 10000100 全位移量	7+m or 3	7+m or 3		r
JNE/JNZ=不等于/不为0则跳(ZF=0)					
8位位移量	01110101 | 8位位移量	7+m or 3	7+m or 3		r
全位移量	00001111 | 10000101 全位移量	7+m or 3	7+m or 3		r
JBE/JNA=低于等于/不高于则跳(CF=1或ZF=1)					
8位位移量	01110110 | 8位位移量	7+m or 3	7+m or 3		r
全位移量	00001111 | 10000110 全位移量	7+m or 3	7+m or 3		r
JNBE/JA=不低于等于/高于则跳(CF=0且ZF=0)					
8位位移量	01110111 | 8位位移量	7+m or 3	7+m or 3		r
全位移量	00001111 | 10000111 全位移量	7+m or 3	7+m or 3		r
JS=有符号则跳(SF=1)					
8位位移量	01111000 | 8位位偏量	7+m or 3	7+m or 3		r

续表

指令	格式			时钟周期次数		注	
				实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
全位移量	00001111	10001000	全位移量	7+m or 3	7+m or 3		r
JNS=无符号则跳(SF=0)							
8位位移量	01111001	8位位移量		7+m or 3	7+m or 3		r
全位移量	00001111	10001001	全位移量	7+m or 3	7+m or 3		r
JP/JPE=(PF=1)则跳							
8位位移量	01111010	8位位移量		7+m or 3	7+m or 3		r
全位移量	00001111	10001010	全位移量	7+m or 3	7+m or 3		r
JNP/JPO=(PF=0)则跳							
8位位移量	01111011	8位位移量		7+m or 3	7+m or 3		r
全位移量	00001111	10001011	全位移量	7+m or 3	7+m or 3		r
JL/JNGE=小于/不大于等于则跳(SF≠OF)							
8位位移量	01111100	8位位移量		7+m or 3	7+m or 3		r
全位移量	00001111	10001100	全位移量	7+m or 3	7+m or 3		r
JNL/JGE=不小于/大于等于则跳(SF=OF)							
8位位移量	01111101	8位位移量		7+m or 3	7+m or 3		r
全位移量	00001111	10001101	全位移量	7+m or 3	7+m or 3		r
JLE/JNG=小于等于/不大于则跳(ZF=1或SF≠OF)							
8位位移量	01111110	8位位移量		7+m or 3	7+m or 3		r
全位移量	00001111	10001110	全位移量	7+m or 3	7+m or 3		r
JNLE/JG=不小于等于/大于则跳(ZF=0且SF=OF)							
8位位移量	01111111	8位位移量		7+m or 3	7+m or 3		r
全位移量	00001111	10001111	全位移量	7+m or 3	7+m or 3		r
JCXZ=CX为0则跳							
	11100011	8位位移量		9+m or 5	9+m or 5		r
JECXZ=ECX为0则跳							
	11100011	8位位移量		9+m or 5	9+m or 5		r
(JCXZ和JECXZ的区别在于地址尺寸不同)							
LOOP=循环CX次							
	11100010	8位位移量		11+m	11+m		r
LOOPZ/LOOPE=为0/等于则循环							
	11100001	8位位移量		11+m	11+m		r
LOOPNZ/LOOPNE=不为0时则循环							
	11100000	8位位移量		11+m	11+m		r
字节的条件置1							
注:时钟周期数给出了寄存器/内存两种情况下的值							

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
SETO=溢出时字节置1(OF=1)					
到寄存器/内存	00001111 10010000 mod 000 r/m	4/5	4/5		h
SETNO=不溢出时字节置1(OF=0)					
到寄存器/内存	00001111 10010001 mod 000 r/m	4/5	4/5		h
SETB/SETNAE=低于/不高于等于时字节置1(CF=1)					
到寄存器/内存	00001111 10010010 mod 000 r/m	4/5	4/5		h
SETNB/SETAE=不低于/高于等于时字节置1(CF=0)					
到寄存器/内存	00001111 10010011 mod 000 r/m	4/5	4/5		h
SETE/SETZ=等于/为0时字节置1(ZF=1)					
到寄存器/内存	00001111 10010100 mod 000 r/m	4/5	4/5		h
SETNE/SETNZ=不等于/不为0时字节置1(ZF=0)					
到寄存器/内存	00001111 10010101 mod 000 r/m	4/5	4/5		h
SETBE/SETNA=低于等于/不高于时字节置1(CF=1或ZF=1)					
到寄存器/内存	00001111 10010110 mod 000 r/m	4/5	4/5		h
SETNBE/SETA=不低于等于/高于时字节置1(CF=0且ZF=0)					
到寄存器/内存	00001111 10010111 mod 000 r/m	4/5	4/5		h
SETS=有符号时字节置1(SF=1)					
到寄存器/内存	00001111 10011000 mod 000 r/m	4/5	4/5		h
SETNS=无符号时字节置1(SF=0)					
到寄存器/内存	00001111 10011001 mod 000 r/m	4/5	4/5		h
SETP/SETPE=(PF=1)时字节置1					
到寄存器/内存	00001111 10011010 mod 000 r/m	4/5	4/5		h
SETNP/SETPO=(PF=0)时字节置1					
到寄存器/内存	00001111 10011011 mod 000 r/m	4/5	4/5		h
SETL/SETNGE=小于/不大于等于时字节置1(SF≠OF)					
到寄存器/内存	00001111 10011100 mod 000 r/m	4/5	4/5		h
SETNL/SETGE=不小于/大于等于时字节置1(SF=OF)					
到寄存器/内存	00001111 01111101 mod 000 r/m	4/5	4/5		h
SETLE/SETNG=小于等于/不大于时字节置1(ZF=1或SF≠OF)					
到寄存器/内存	00001111 10011110 mod 000 r/m	4/5	4/5		h
SETNLE/SETG=不小于等于/大于时字节置1(ZF=0且SF=OF)					
到寄存器/内存	00001111 10011111 mod 000 r/m	4/5	4/5		h
ENTER=进入过程					
	11001000 16位偏移量, 8位嵌套级				

续表

指令 格式	时钟周期次数		注	
	实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
L=0	10	10	b	h
L=1	12	12	b	h
L>1	15+4(n-1)	15+4(n-1)	b	h
LEAVE=退出过程				
11001001	4	4	b	h
中断指令				
INT=中断				
类型指定 11001101 类型	37		b	
类型3 11001100	33		b	
INTO=若OF=1,中断4				
11001110				
若OF=1	35		b,e	
若OF=0	3	3	b,e	
BOUND=若检测值超出范围,中断5				
01100010 mod reg r/m				
若超出范围	44		b,e	e,g,h,j,k,r
若在范围之内	10	10	b,e	e,g,h,j,k,r
仅保护模式(INT)				
INT:类型指定				
通过中断或自陷门到同一特权级		59		g,j,k,r
通过中断或自陷门到不同的特权级		99		g,j,k,r
通过任务门从286任务到286TSS		282		g,j,k,r
通过任务门从286任务到386TSS		309		g,j,k,r
通过任务门从286任务到虚拟8086模式		226		g,j,k,r
通过任务门从386任务到286TSS		284		g,j,k,r
通过任务门从386任务到386TSS		311		g,j,k,r
通过任务门从386任务到虚拟8086模式		228		g,j,k,r
通过任务门从虚拟8086模式到286TSS		289		g,j,k,r
通过任务门从虚拟8086模式到386TSS		316		g,j,k,r
通过自陷门或中断门从虚拟8086模式到特权级0		119		
INT:类型3				
通过中断或自陷门到同一特权级		59		g,j,k,r
通过中断或自陷门到不同的特权级		99		g,j,k,r
通过任务门从286任务到286TSS		278		g,j,k,r

续表

指令 格式	时钟周期次数		注	
	实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
通过任务门从286任务到386TSS		305		g,j,k,r
通过任务门从286任务到虚拟8086模式		222		g,j,k,r
通过任务门从386任务到286TSS		280		g,j,k,r
通过任务门从386任务到386TSS		307		g,j,k,r
通过任务门从386任务到虚拟8086模式		224		g,j,k,r
通过任务门从虚拟8086模式到286TSS		285		g,j,k,r
通过任务门从虚拟8086模式到386TSS		312		g,j,k,r
通过自陷门或中断门从虚拟8086模式到特权级0		119		
INT0				
通过中断或自陷门到同一特权级		59		g,j,k,r
通过中断或自陷门到不同的特权级		99		g,j,k,r
通过任务门从286任务到286TSS		280		g,j,k,r
通过任务门从286任务到386TSS		307		g,j,k,r
通过任务门从286任务到虚拟8086模式		224		g,j,k,r
通过任务门从386任务到286TSS		282		g,j,k,r
通过任务门从386任务到386TSS		309		g,j,k,r
通过任务门从386任务到虚拟8086模式		226		g,j,k,r
通过任务门从虚拟8086模式到286TSS		287		g,j,k,r
通过任务门从虚拟8086模式到386TSS		314		g,j,k,r
通过自陷门或中断门从虚拟8086模式到特权级0		119		
BOUND:				
通过中断或自陷门到同一特权级		59		g,j,k,r
通过中断或自陷门到不同的特权级		99		g,j,k,r
通过任务门从286任务到286TSS		254		g,j,k,r
通过任务门从286任务到386TSS		284		g,j,k,r
通过任务门从286任务到虚拟8086模式		231		g,j,k,r
通过任务门从386任务到286TSS		264		g,j,k,r
通过任务门从386任务到386TSS		294		g,j,k,r
通过任务门从386任务到虚拟8086模式		243		g,j,k,r
通过任务门从虚拟8086模式到286TSS		264		g,j,k,r
通过任务门从虚拟8086模式到386TSS		294		g,j,k,r
通过自陷门或中断门从虚拟模式到特权级0		119		
中断返回				
IRET=中断返回 11001111	22			g,h,j,k,r

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
保护模式(IRET)					
到同一特权级(任务内)			38		g,h,j,k,r
到不同的特权级(任务内)			82		g,h,j,k,r
从286任务到286TSS			232		h,j,k,r
从286任务到386TSS			265		h,j,k,r
从286任务到虚拟8086任务			214		h,j,k,r
从286任务到虚拟8086模式(任务内)			60		
从386任务到286TSS			271		h,j,k,r
从386任务到386TSS			275		h,j,k,r
从386任务到虚拟8086任务			224		h,j,k,r
从386任务到虚拟8086模式(任务内)			60		
处理器控制					
HLT—停机	11110100	5	5		1
MOV=和控制寄存器/调试寄存器/测试寄存器之间的数据传送					
从寄存器到CR0/CR2/CR3	00001111 00100010 11 ccc reg	10/4/5	10/4/5		1
从CR0—3到寄存器	00001111 00100000 11 eee reg	6	6		1
从寄存器到DR0—3	00001111 00100011 11 eee reg	22	22		1
从寄存器到DR6—7	00001111 00100011 11 eee reg	16	16		1
从DR6—7到寄存器	00001111 00100001 11 eee reg	14	14		1
从DR0—3到寄存器	00001111 00100001 11 eee reg	22	22		1
从寄存器到TR6—7	00001111 00100110 11 eee reg	12	12		1
从TR6—7到寄存器	00001111 00100100 11 eee reg	12	12		1
NOP=空操作	10010000	3	3		
WAIT=等待BUSY*引脚为负	10011011	6	6		
处理器扩充指令					
处理器扩充换码	11011TTT mod L l. l. r/m	请看80287			h

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
	TTT和LLL位为协处理器的操作码信息	/80387关于时钟周期数的数据			
前缀字节					
地址尺寸前缀	01100111	0	0		
LOCK=总线锁定前缀					
	11110000	0	0		m
操作数尺寸前缀	01100110	0	0		
段超越前缀					
CS:	00101110	0	0		
DS:	00111110	0	0		
ES:	00100110	0	0		
FS:	01100100	0	0		
GS:	01100101	0	0		
SS:	00110110	0	0		
保护控制指令					
ARPL=从寄存器/内存调整RPL					
	01100011 mod reg r/m	N/A	20/21	a	h
LAR=从寄存器/内存装入存取权限					
	00001111 00000010 mod reg r/m	N/A	15/16	a	g,h,j,p
LGDT=装入全局描述符表寄存器					
	00001111 00000001 mod 010 r/m	11	11	b,c	h,l
LIDT=装入中断描述符表寄存器					
	00001111 00000001 mod 011 r/m	11	11	b,c	h,l
LLDT=将本地描述符表寄存器装入寄存器/内存					
	00001111 00000000 mod 010 r/m	N/A	20/24	a	g,h,j,l
LMSW=从寄存器/内存装入机器状态字					
	00001111 00000001 mod 110 r/m	10/13	10/13	b,c	h,l
LSL=从寄存器/内存装入段界限					
	00001111 00000011 mod reg r/m				
字节粒度界限		N/A	20/21	a	g,h,j,p
页粒度界限		N/A	25/26	a	g,h,j,p
LTR=从寄存器/内存装入任务寄存器					
	00001111 00000000 mod 011 r/m	N/A	23/27	a	g,h,j,i

续表

指令	格式	时钟周期次数		注	
		实地址模式或虚拟8086模式	保护的虚地址模式	实地址模式或虚拟8086模式	保护的虚地址模式
SGDT=存储全局描述符表寄存器	00001111 00000001 mod 000 r/m	9	9	b,c	h
SIDT=存储中断描述符表寄存器	00001111 00000001 mod 001 r/m	9	9	b,c	h
SLDT=将本地描述符表寄存器存入寄存器/内存	00001111 00000000 mod 000 r/m	N/A	2/2	a	h
SMSW=存储机器状态字	00001111 00000001 mod 100 r/m	10/13	10/13	b,c	h,l
STR=将任务寄存器存入寄存器/内存	00001111 00000000 mod 001 r/m	N/A	2/2	a	h
VERR=检验段是否可读	00001111 00000000 mod 100 r/m	N/A	10/11	a	g,h,j,p
VERW=检验段是否可写	00001111 00000000 mod 101 r/m	N/A	15/16	a	g,h,j,p

*If CPL≤IOPL **If CPL>IOPL

注：+若I/O的权限允许在虚拟8086模式下和端口之间进行I/O，则所指示的时钟周期次数才管用。若I/O位图不具备权限，则异常13发生；时钟周期采用INT 3指令的。

*若CPL≤IOPL. **若CPL>IOPL

表H.1的"注"

"注"中a到c仅用于80386实地址模式。

a　这是个保护模式指令。如在实地址模式下执行，将引起异常6(非法操作码)。

b　在实地址模式，如果操作数的访问部分地或全部地超出了CS、DS、ES、FS或GS界限的最大值FFFFh，则发生异常13故障(通用的保护违犯)。在实地址模式，如果操作数的访问部分地或全部地超出了SS界限的最大值，则发生异常12故障(堆栈段界限违犯或未出现)。

c　该指令在实模式下可能被执行。它的主要目的在于将CPU初始化为保护模式。

"注"中d到g用于80386的实地址模式和保护的虚地址模式。

d　80386采用了early-out乘法算法。真正的时钟周期数取决于操作数(乘数)中最高有效位的位置。表中给出了时钟周期数的最小值和最大值。真正的周期数的计算公式为：

$$真正周期数=\begin{cases}([\log_2|m|],3)_{max}+6 & (m\neq 0)\\ 9 & (m=0)\end{cases}$$

其中m为乘数。

e　是否发生异常将取决于操作数的值。

f　不管有没有LOCK#前缀，LOCK#会自动生效。

g　在描述符表存取期间，LOCK#生效。

注中h到r仅用于80386的保护的虚地址模式：

h　如果在CS、DS、ES、FS或GS中的内存操作数，由于段界限违犯或存取权违犯而不能使用，便发生异常13故障(通用保护违犯)。如果栈界限被违反则发生异常12(堆栈段界限违犯或者未出现)。

i　对于段加载操作，若要避免异常13故障(通用保护违犯)，必须使CPL、RPL和DPL符合特权级规则。段的描述符必须标为"存在"，否则发生异常11(CS、DS、ES、FS、GS不存在)。如果SS寄存器被加载，且检测出堆栈段为"不存在"，则发生异常12(堆栈堆界限违犯或未出现)。

j　为维护多处理器系统中描述符的安全性，由该指令引起的所有对GDT和LDT中的段描述符的存取，都将使LOCK"自动生效。

k　如果访问另一代码段的JMP、CALL、INT、RET和IRET指令违犯了特权规则，则引起异常13(通用保护违犯)。

l　若CPL大于0(0为最高特权级)则引起异常13故障。

m　若CPL大于IOPL则引起异常13故障。

n　若CPL大于IOPL，标志寄存器的IF位不更新。只有当CPL=0时，IOPL和标志寄存器的VM字段才被更新。

o　MSW(CR0)的PE位不能被这条指令复位。如果想要复位PE位，得用MOV(至CR0)指令。

p　用于选择符操作数的任何特权规则遭到违犯都不会引起保护异常，只使零标志清0。

q　如果协处理器的内存操作违犯了段界限或者段存取权限，那么在ESC指令执行之前将发生异常13(通用保护违犯)。如果操作数的起始地址违犯了栈的界限，则引起异常12故障(堆栈段界限违犯或未出现)。

r　JMP、CALL、INT、RET或IRET指令的目标必须在定义的代码段界限之内，否则将引起异常13(通用保护违犯)。

H.1.2　有关指令的时钟周期次数的假设

(1) 已完成对该指令的取指和译码操作，指令已做好执行前的一切准备。

(2) 总线周期不需要等待状态。

(3) 设有延迟处理器对总线存取的本地总线HOLD请求。

(4) 在指令执行期间未检测到异常。

(5) 在计算有效地址时，不使用两个通用寄存器的分量。在表中所示的时钟周期次数之中只利用了一个寄存器、比例因子和偏移量。因此，如果有效地址的计算使用了两个寄存器的分量的话，则应在表中所示的时钟周期数上面再加上个1。

H.1.3　关于时钟周期数表示法的约定

(1) 如给出两个时钟周期次数的话，小的那个对应着寄存器操作数，大的那个对应着内存操作数。

(2) n=重复的次数。

(3) m=在下一条执行指令中的分量数，这里，完整的偏移量算作一个分量，完整的立即数算作一个分量，指令的每一个其它字节，和每一个前缀都算作一个分量。

H.2 指令编码

H.2.1 指令的一般格式

图 H.1 表示了指令的一般格式，所有指令的编码都是这一般格式的子集。指令的一般格式可分解为四个组成部分：

其基本部分为操作码，占一至两个字节。其余的三个部分，有的指令有，有的指令没有。它们顺次是：地址说明（由一至两个 MODRM 字节组成）、地址偏移量（由零至四个字节组成）和立即数（由零至四个字节组成）。详见图 H.1。

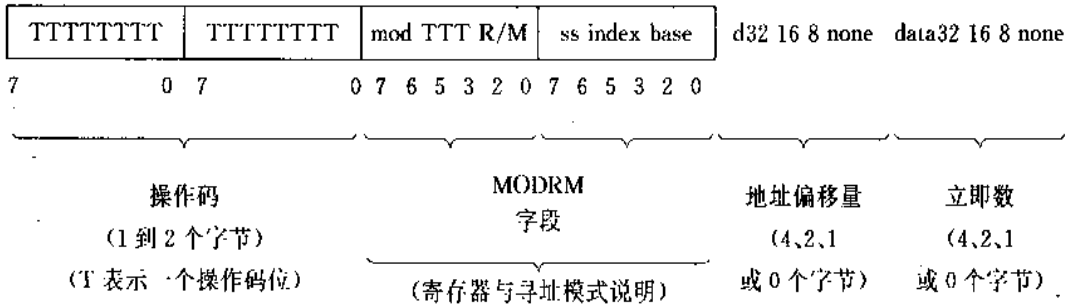


图 H.1 指令的一般格式

在基本的操作码内部，又可以分为若干个较小的编码段（字段）。这些字段因操作的类型而异。它们定义了诸如操作方向、偏移量尺寸、寄存器编码、符号扩展等等信息。

几乎所有的访问内存操作数的指令，在操作码的后面，总有一至两个 MODRM 字节。

寻址模式里往往含有地址偏移量。这个偏移量直接写在 MODRM 字节之后，常有 8 位、16 位，或 32 位三种格式。

若指令规定了立即数操作数，那么这个立即数一定是这个指令的最后一个字段。

图 H.1 中的 mod、R/M 等等是出现在上述四大部分中的更小的字段。可能出现在 80386 指令集中的所有的小字段，用表 H.2 列出。

表 H.2 80386 指令内的字段

字段名	说　明	位数
w	用于表示数据是字节还是全尺寸 （全尺寸即指 16 位或 32 位）	1
d	表示数据操作的方向	1
s	表示立即数据字段是否被符号扩展	1
reg	通用寄存器说明	3
MODRM	寻址模式说明（有效地址可以是一个寄存器）	2 位用作 mod 3 位用作 R/M
ss	比例变址寻址模式的比例因子	2
base	被用作基址寄存器的通用寄存器	3
index	被用作变址寄存器的通用寄存器	3
sreg2	选择 CS,SS,DS,ES 之一的段寄存器说明	2
sreg3	选择 CS,SS,DS,ES,FS,GS 之一的段寄存器说明	3
tttn	对于条件指令，表示一个断言的条件或“非”条件	4

注意：表 H.1 表示的是各个独立指令的译码

H.2.2 指令集的32位扩展

对于80386,86/186/286指令集在两个正交的方向上被扩展：增加了所有16位指令的32位格式,以支持32位数据类型；32位的寻址方式对所有访问内存的指令都有效。这正交的指令集扩展是根据代码段描述符中的缺省(D)位和两个指令前缀来完成的。

指令默认的究竟是16位操作,还是32位操作,取决于代码段描述符中D位。代码段的描述符会给出默认的长度(32位或是16位),以及当执行时代码段的有效地址。在实模式或虚拟8086模式,因没有代码段描述符,80386内部认D值为0(即默认的尺寸为16位,与8086/80186/80286兼容)。

操作数尺寸前缀和有效地址尺寸前缀,可以独立地超越操作数尺寸和有效地址尺寸的默认值。这两个前缀可以单独使用,也可同时使用,放在超越默认值的指令的操作码字节的前头。但它的作用范围仅限于这一条指令,对别的指令不产生影响。

操作数尺寸前缀与有效地址尺寸前缀的出现将分别使操作数尺寸和有效地址尺寸取默认值的反面的值。例如,一个默认为32位的数据操作,可以在操作数尺寸前缀的作用下变为16位的数据操作。默认有效地址尺寸为16位的指令,受有效地址尺寸前缀的作用,可以变成一条使用32位有效地址计算的指令。

在80386的所有模式中,包括实模式和虚拟8086模式,上述32位扩展都是有效的。而在这些模式中,默认值总是16位的,于是若要规定32位的操作数或地址,必须使用这两个前缀。

除了规定的之外,带有8位和16位操作数的指令不能影响扩展的寄存器的高位的内容。

H.2.3. 指令字段的编码

实际上指令还可以细分为一系列表示寄存器选择,寻址模式等等的字段。下面我们来解释这些字段的定义。

1. 操作数长度字段(W)

执行数据操作的指令,有执行32位操作的和执行16位操作的两种。在这两种指令中,W字段均起着限制操作的尺寸的作用,请看下表：

W字段的值	在16位数据操作中实际的操作数尺寸	在32位数据操作中实际的操作数尺寸
0	8位	8位
1	16位(全操作数尺寸)	32位(全操作数尺寸)

若W=0,实际的操作数尺寸为8位。只有当W=1时,才实现相应的全操作数尺寸。

2. 通用寄存器字段(reg)

reg字段规定了指令操作的通用寄存器。reg字段有时出现在操作码字节中,有时以MODRM字节中的reg字段的形式出现,有时还充作MODRM字节的R/M字段。reg字段的编码含义与指令中是否含有W字段有关,与W字段(若有W字段的话)的值有关。表H.3为指令中不含W字段时reg的编码;表H.4为W字段出现在指令之中时reg字段的编码。

表 H.3　指令中不含 W 字段时 reg 字段的编码

REG 字段	在 16 位数据操作时选择的寄存器	在 32 位数据操作时选择的寄存器	REG 字段	在 16 位数据操作时选择的寄存器	在 32 位数据操作时选择的寄存器
000	AX	EAX	100	SP	ESP
001	CX	ECX	101	BP	EBP
010	DX	EDX	110	SI	ESI
011	BX	EBX	111	DI	EDI

表 H.4　在 W 字段出现在指令中时 reg 字段的编码

REG 字段	在 16 位数据操作时由 reg 字段指定的寄存器(它们是 reg 和 W 的函数)		REG 字段	在 32 位数据操作时由 reg 字段指定的寄存器(它们是 reg 和 W 的函数)	
	(当 W=0)	(当 W=1)		(当 W=0)	(当 W=1)
000	AL	AX	000	AL	EAX
001	CL	CX	001	CL	ECX
010	DL	DX	010	DL	EDX
011	BL	BX	011	BL	EBX
100	AH	SP	100	AH	ESP
101	CH	BP	101	CH	EBP
110	DH	SI	110	DH	ESI
111	BH	DI	111	BH	EDI

3. 段寄存器字段(sreg)

如果 sreg 字段为 2 位，用来规定 80286 的四种段寄存器中的一种，请看下面的表 H.5。有时，sreg 字段占据了 3 位，则除了可以在 ES、CS、SS、DS 中进行选择外，选择的对象还可以包括 FS 和 GS(80386 的段寄存器)，请看表 H.6

表 H.5　2 位的 sreg2 字段

2 位的 sreg2 字段	指定的寄存器	2 位的 sreg2 字段	指定的寄存器
00	ES	10	SS
01	CS	11	DS

表 H.6　3 位的 sreg3 字段

3 位的 sreg3 字段	指定的段寄存器	3 位的 sreg3 字段	指定的段寄存器
000	ES	100	FS
001	CS	101	GS
010	SS	110	无用
011	DS	111	无用

4. 寻址方式

对于大多数指令来说，当前指令的寻址方式要由一至两个 MODRM 字节来说明。只有少数的、寻址方式早已确定了的特殊指令(如 PUSH，POP)是例外。

大多数 MODRM 字段为单字节。如果在 32 位寻址方式下规定使用双字节的 MODRM 字

段，则 MODRM 字段的 R/M=100，mod=00、01、或 10。这时，该 32 位的寻址方式应为 mod、SS、index 和 base 字段的函数。

在 MODRM 的图示中(图 H.1)有 3 位标为"TTT"的字段。这 3 个 TTT 位，有时用作主操作码的扩展，也有时用作 reg 字段。

如果指令要求 16 位寻址，则用 16 位的地址分量来计算有效地址，MODRM 字段被解释为 16 位的寻址说明。如果指令要求 32 位寻址，则用 32 位的地址分量来计算有效地址，MODRM 字段被解释为 32 位的寻址说明。详见表 H.7、H.7 续、H.8、H8 续、H.9 及 H.9 续。

表 H.7　带 MODRM 字节的 16 位寻址方式的编码(a)

MODRM	有效地址	MODRM	有效地址
00 000	DS：[BX+SI]	10 000	DS：[BX+SI+d16]
00 001	DS：[BX+DI]	10 001	DS：[BS+DI+d16]
00 010	SS：[BP+SI]	10 010	SS：[BP+SI+d16]
00 011	SS：[BP+DI]	10 011	SS：[BP+DI+d16]
00 100	DS：[SI]	10 100	DS：[SI+d16]
00 101	DS：[DI]	10 101	DS：[DI+d16]
00 110	DS：d16	10 110	SS：[BP+d16]
00 111	DS：[BX]	10 111	DS：[BX+d16]
01 000	DS：[BX+SI+d8]	11 000	寄存器　请看下面续表
01 001	DS：[BX+DI+d8]	11 001	寄存器—请看下面续表
01 010	SS：[BP+SI+d8]	11 010	寄存器—请看下面续表
01 011	SS：[BP+DI+d8]	11 011	寄存器—请看下面续表
01 100	DS：[SI+d8]	11 100	寄存器—请看下面续表
01 101	DS：[DI+d8]	11 101	寄存器—请看下面续表
01 110	SS：[BP+d8]	11 110	寄存器—请看下面续表
01 111	DS：[BX+d8]	11 111	寄存器—请看下面续表

表 H.7　带 MODRM 字节的 16 位寻址方式的编码(b)

MODRM	16 位数据操作时使用的寄存器(它们是 R/M 和 W 的函数)		MODRM	32 位数据操作时使用的寄存器(它们是 R/M 和 W 的函数)	
	(当 W=0)	(当 W=1)		(当 W=0)	(当 W=1)
11 000	AL	AX	11 000	AL	EAX
11 001	CL	CX	11 001	CL	ECX
11 010	DL	DX	11 010	DL	EDX
11 011	BL	BX	11 011	BL	EBX
11 100	AH	SP	11 100	AH	ESP
11 101	CH	BP	11 101	CH	EBP
11 110	DH	SI	11 110	DH	ESI
11 111	BH	DI	11 111	BH	EDI

表 H.8　带单字节 MODRM 的 32 位寻址方式的编码(a)

MODRM	有效地址	MODRM	有效地址
00 000	DS：[EAX]	10 000	DS：[EAX+d32]
00 001	DS：[ECX]	10 001	DS：[ECX+d32]
00 010	DS：[EDX]	10 010	DS：[EDX+d32]
00 011	DS：[EBX]	10 011	DS：[EBX+d32]
00 100	转到双字节	10 100	转到双字节
00 101	DS：d32	10 101	SS：[EBP+d32]
00 110	DS：[ESI]	10 110	DS：[ESI+d32]
00 111	DS：[EDI]	10 111	DS：[EDI+d32]
01 000	DS：[EAX+d8]	11 000	寄存器—请看下面续表
01 001	DS：[ECX+d8]	11 001	寄存器　请看下面续表
01 010	DS：[EDX+d8]	11 010	寄存器—请看下面续表
01 011	DS：[EBX+d8]	11 011	寄存器—请看下面续表
01 100	转到双字节	11 100	寄存器　请看下面续表
01 101	SS：[EBP+d8]	11 101	寄存器—请看下面续表
01 110	DS：[ESI+d8]	11 110	寄存器—请看下面续表
01 111	DS：[EDI+d8]	11 111	寄存器　请看下面续表

MODRM	16 位数据操作时使用的寄存器(它们是 R/M 和 W 的函数)	
	(当 W=0)	(当 W=1)
11 000	AL	AX
11 001	CL	CX
11 010	DL	DX
11 011	BL	BX
11 100	AH	SP
11 101	CH	BP
11 110	DH	SI
11 111	BH	DI

表 H.8　带单字节 MODRM 的 32 位寻址方式的编码(b)

MODRM	32 位数据操作时使用的寄存器(它们是 R/M 和 W 的函数)		MODRM	32 位数据操作时使用的寄存器(它们是 R/M 和 W 的函数)	
	(当 W=0)	(当 W=1)		(当 W=0)	(当 W=1)
11 000	AL	EAX	11 100	AH	ESP
11 001	CL	ECX	11 101	CH	EBP
11 010	DL	EDX	11 110	DH	ESI
11 011	BL	EBX	11 111	BH	EDI

表 H.9 带双字节 MODRM 的 32 位寻址方式的编码

MOD BASE	有效地址	MOD BASE	有效地址
00 000	DS：[EAX+(scaled index)]	01 100	SS：[ESP+(scaled index)+d8]
00 001	DS：[ECX+(scaled index)]	01 101	DS：[EBP+(scaled index)+d8]
00 010	DS：[EDX+(scaled index)]	01 110	DS：[ESI+(scaled index)+d8]
00 011	DS：[EBX+(scaled index)]	01 111	DS：[EDI+(scaled index)+d8]
00 100	SS：[ESP+(scaled index)]	10 000	DS：[EAX+(scaled index)+d32]
00 101	DS：[d32+(scaled index)]	10 001	DS：[ECX+(scaled index)+d32]
00 110	DS：[ESI+(scaled index)]	10 010	DS：[EDX+(scaled index)+d32]
00 111	DS：[EDI+(scaled index)]	10 011	DS：[EBX+(scaled index)+d32]
01 000	DS：[EAX+(scaled index)+d8]	10 100	SS：[ESP+(scaled index)+d32]
01 001	DS：[ECX+(scaled index)+d8]	10 101	DS：[EBP+(scaled index)+d32]
01 010	DS：[EDX+(scaled index)+d8]	10 110	DS：[ESI+(scaled index)+d32]
01 011	DS：[EBX+(scaled index)+d8]	10 111	DS：[EDI+(scaled index)+d32]
SS	比例因子	INDEX	变址寄存器
00	x1	000	EAX
01	x2	001	ECX
10	x4	010	EDX
11	x8	011	EBX
		100	没有变址寄存器
		101	EBP
		110	ESI
		111	EDI

**特请读者注意：

若 index 字段为 100，表示"没有变址寄存器"，这时 SS 字段必须为 00。如果 index 为 100 而 SS 不为 00，则有效地址不确定。

5. 操作方向(d)

在许多双操作数的指令中，d 字段用来指示哪个是源操作数，哪个是目的操作数。

d 操作方向

0 寄存器/内存←寄存器

"reg"字段指示的是源操作数

"mod R/M"或"mod SS index base"字段指示的是目的操作数

1 寄存器←寄存器/内存

"reg"字段指示的是目的操作数

"mod R/M"或"mod SS index base"字段指示的是源操作数

6. 符号扩展(S)

S 字段主要用在带立即数据字段的指令中，仅当 8 位的立即数被移至 16 位或 32 位的目标时，S 字段才产生影响；否则无用。

S的值	对8位立即数的影响	对16/32位立即数的影响
0	无	无
1	让8位立即数符号扩展为16位或32位目标数	无

7. 条件测试(tttn)

用于条件指令(条件跳转、以及根据条件字节置1)。其中n指被测标志位的值。tttn的编码请看表H.10。

表H.10 条件测试(tttn)的编码

助记符	条件	tttn	助记符	条件	tttn
O	溢出	0000	S	符号	1000
NO	不溢出	0001	NS	无符号	1001
B/NAE	低于/不高于等于	0010	P/PE	奇偶校验/偶	1010
NB/AE	不低于/高于等于	0011	NP/PO	不奇偶校验/奇	1011
E/Z	等于/为0	0100	L/NGE	小于/不大于等于	1100
NE/NZ	不等于/不为0	0101	NL/GE	不小于/大于等于	1101
BE/NA	低于等于/不高于	0110	LE/NG	小于等于/不大于	1110
NBE/A	不低于等于/高于	0111	NLE/G	不小于等于/大于	1111

8. 控制寄存器或调试寄存器(eee)

用于控制寄存器、调试寄存器和测试寄存器的装入和存储。

(1) 当解释为控制寄存器字段

eee码	寄存器名
000	CR0
010	CR2
011	CR3

(2) 当解释为调试寄存器字段

eee码	寄存器名
000	DR0
001	DR1
010	DR2
011	DR3
110	DR6
111	DR7

(3) 当解释为测试寄存器字段

eee码	寄存器名
110	TR6
111	TR7

附录 I 机器指令解码

第一个字节		第二个字节	第 3—7 字节	汇编 386 指令格式	
16 进制	二进制				
D8	1101 1000	MOD 000 R/M	SIB，位移量	FADD	单精度实数
D8	1101 1000	MOD 001 R/M	SIB，位移量	FMUL	单精度实数
D8	1101 1000	MOD 010 R/M	SIB，位移量	FCOM	单精度实数
D8	1101 1000	MOD 011 R/M	SIB，位移量	FCOMP	单精度实数
D8	1101 1000	MOD 100 R/M	SIB，位移量	FSUB	单精度实数
D8	1101 1000	MOD 101 R/M	SIB，位移量	FSUBR	单精度实数
D8	1101 1000	MOD 110 R/M	SIB，位移量	FDIV	单精度实数
D8	1101 1000	MOD 111 R/M	SIB，位移量	FDIVR	单精度实数
D8	1101 1000	1100 0 REG		FADD	ST,ST(i)
D8	1101 1000	1100 1 REG		FMUL	ST,ST(i)
D8	1101 1000	1101 0 REG		FCOM	ST(i)
D8	1101 1000	1101 1 REG		FCOMP	ST(i)
D8	1101 1000	1110 0 REG		FSUB	ST,ST(i)
D8	1101 1000	1110 1 REG		FSUBR	ST,ST(i)
D8	1101 1000	1111 0 REG		FDIV	ST,ST(i)
D8	1101 1000	1111 1 REG		FDIVR	ST,ST(i)
D9	1101 1001	MOD 000 R/M	SIB,位移量	FLD	单精度实数
D9	1101 1001	MOD 001 R/M		保留	
D9	1101 1001	MOD 010 R/M	SIB,位移量	FST	单精度实数
D9	1101 1001	MOD 011 R/M	SIB,位移量	FSTP	单精度实数
D9	1101 1001	MOD 100 R/M	SIB,位移量	FLDENV	14 到 28 字节 * * *
D9	1101 1001	MOD 101 R/M	SIB,位移量	FLDCW	2 个字节
D9	1101 1001	MOD 110 R/M	SIB,位移量	FSTENV	14 到 28 字节 * * *
D9	1101 1001	MOD 111 R/M	SIB,位移量	FSTCW	2 个字节
D9	1101 1001	1100 0 REG		FLD	ST(i)
D9	1101 1001	1100 1 REG		FXCH	ST(i)
D9	1101 1001	1101 0000		FNOP	
D9	1101 1001	1101 0001		保留	
D9	1101 1001	1101 001—		保留	
D9	1101 1001	1101 01——		保留	
D9	1101 1001	1101 1 REG		*(1)	
D9	1101 1001	1110 0000		FCHS	
D9	1101 1001	1110 0001		FABS	
D9	1101 1001	1110 001--		保留	
D9	1101 1001	1110 0100		FTST	
D9	1101 1001	1101 0101		FXAM	
D9	1101 1001	1110 011—		保留	

续表

第一个字节		第二个字节	第3—7字节	汇编386指令格式	
16进制	二进制				
D9	1101 1001	1110 1000		FLD1	
D9	1101 1001	1110 1001		FLDL2T	
D9	1101 1001	1110 1010		FLDL2E	
D9	1101 1001	1110 1011		FLDPI	
D9	1101 1001	1110 1100		FLDLG2	
D9	1101 1001	1110 1101		FLDLN2	
D9	1101 1001	1110 1110		FLDZ	
D9	1101 1001	1110 1111		保留	
D9	1101 1001	1111 0000		F2XM1	
D9	1101 1001	1111 0001		FYL2X	
D9	1101 1001	1111 0010		FPTAN	
D9	1101 1001	1111 0011		FPATAN	
D9	1101 1001	1111 0100		FXTRACT	
D9	1101 1001	1111 0101		FPREM1	
D9	1101 1001	1111 0110		FDECSTP	
D9	1101 1001	1111 0111		FINCSTP	
D9	1101 1001	1111 1000		FPREM	
D9	1101 1001	1111 1001		FYL2XP1	
D9	1101 1001	1111 1010		FSQRT	
D9	1101 1001	1111 1011		FSINCOS	
D9	1101 1001	1111 1100		FRNDINT	
D9	1101 1001	1111 1101		FSCALE	
D9	1101 1001	1111 1110		FSIN	
D9	1101 1001	1111 1111		FCOS	
DA	1101 1010	MOD 000 R/M	SIB,位移量	FIADD	短整数
DA	1101 1010	MOD 001 R/M	SIB,位移量	FIMUL	短整数
DA	1101 1010	MOD 010 R/M	SIB,位移量	FICOM	短整数
DA	1101 1010	MOD 011 R/M	SIB,位移量	FICOMP	短整数
DA	1101 1010	MOD 100 R/M	SIB,位移量	FISUB	短整数
DA	1101 1010	MOD 101 R/M	SIB,位移量	FISUBR	短整数
DA	1101 1010	MOD 110 R/M	SIB,位移量	FIDIV	短整数
DA	1101 1010	MOD 111 R/M	SIB,位移量	FIDIVR	短整数
DA	1101 1010	110 ———		保留	
DA	1101 1010	1110 0———		保留	
DA	1101 1010	1110 1000		保留	
DA	1101 1010	1110 1001		FUCOMPP	
DA	1101 1010	1110 101—		保留	
DA	1101 1010	1110 11——		保留	
DA	1101 1010	1111————		保留	
DB	1101 1011	MOD 000 R/M	SIB,位移量	FILD	短整数
DB	1101 1011	MOD 001 R/M	SIB,位移量	保留	
DB	1101 1011	MOD 010 R/M	SIB,位移量	FIST	短整数

续表

第一个字节		第二个字节	第3—7字节	汇编386指令格式	
16进制	二进制				
DB	1101 1011	MOD 011 R/M	SIB,位移量	FISTP	短整数
DB	1101 1011	MOD 100 R/M	SIB,位移量	保留	
DB	1101 1011	MOD 101 R/M	SIB,位移量	FLD	扩展的整数
DB	1101 1011	MOD 110 R/M	SIB,位移量	保留	
DB	1101 1011	MOD 111 R/M	SIB,位移量	FSTP	扩展的整数
DB	1101 1011	110——————		保留	
DB	1101 1011	1110 0000		* * (1)	
DB	1101 1011	1110 0001		* * (2)	
DB	1101 1011	1110 0010		FCLEX	
DB	1101 1011	1110 0011		FINIT	
DB	1101 1011	1110 0100		* * (3)	
DB	1101 1011	1110 0101		保留	
DB	1101 1011	1110 011—		保留	
DB	1101 1011	1110 1——		保留	
DB	1101 1011	1111 ——————		保留	
DC	1101 1100	MOD 000 R/M	SIB,位移量	FADD	双精度实数
DC	1101 1100	MOD 001 R/M	SIB,位移量	FMUL	双精度实数
DC	1101 1100	MOD 010 R/M	SIB,位移量	FCOM	双精度实数
DC	1101 1100	MOD 011 R/M	SIB,位移量	FCOMP	双精度实数
DC	1101 1100	MOD 100 R/M	SIB,位移量	FSUB	双精度实数
DC	1101 1100	MOD 101 R/M	SIB,位移量	FSUBR	双精度实数
DC	1101 1100	MOD 110 R/M	SIB,位移量	FDIV	双精度实数
DC	1101 1100	MOD 111 R/M	SIB,位移量	FDIVR	双精度实数
DC	1101 1100	1100 0 REG		FADD	ST(i),ST
DC	1101 1100	1100 1 REG		FMUL	ST(i),ST
DC	1101 1100	1101 0 REG		* (2)	
DC	1101 1100	1101 1 REG		* (3)	
DC	1101 1100	1110 0 REG		FSUBR	ST(i),ST
DC	1101 1100	1110 1 REG		FSUB	ST(i),ST
DC	1101 1100	1111 0 REG		FDIVR	ST(i),ST
DC	1101 1100	1111 1 REG		FDIV	ST(i),ST
DD	1101 1101	MOD 000 R/M	SIB,位移量	FLD	双精度实数
DD	1101 1101	MOD 001 R/M		保留	
DD	1101 1101	MOD 010 R/M	SIB,位移量	FST	双精度实数
DD	1101 1101	MOD 011 R/M	SIB,位移量	FSTP	双精度实数
DD	1101 1101	MOD 100 R/M	SIB,位移量	FRSTOR	94或108字节* * *
DD	1101 1101	MOD 101 R/M	SIB,位移量	保留	
DD	1101 1101	MOD 110 R/M	SIB,位移量	FSAVE	94或108字节* * *
DD	1101 1101	MOD 111 R/M	SIB,位移量	FSTSW	2个字节
DD	1101 1101	1100 0 REG		FFREE	ST(i)
DD	1101 1101	1100 1 REG		* (4)	
DD	1101 1101	1101 0 REG		FST	ST(i)

续表

第一个字节		第二个字节	第3—7字节	汇编386指令格式	
16进制	二进制				
DD	1101 1101	1101 1 REG		FSTP	ST(i)
DD	1101 1101	1110 0 REG		FUCOM	ST(i)
DD	1101 1101	1110 1 REG		FUCOMP	ST(i)
DD	1101 1101	1111 ————		保留	
DE	1101 1110	MOD 000 R/M	SIB,位移量	FIADD	字整数
DE	1101 1110	MOD 001 R/M	SIB,位移量	FIMUL	字整数
DE	1101 1110	MOD 010 R/M	SIB,位移量	FICOM	字整数
DE	1101 1110	MOD 011 R/M	SIB,位移量	FICOMP	字整数
DE	1101 1110	MOD 100 R/M	SIB,位移量	FISUB	字整数
DE	1101 1110	MOD 101 R/M	SIB,位移量	FISUBR	字整数
DE	1101 1110	MOD 110 R/M	SIB,位移量	FIDIV	字整数
DE	1101 1110	MOD 111 R/M	SIB,位移量	FIDIVR	字整数
DE	1101 1110	1100 0 REG		FADDP	ST(i),ST
DE	1101 1110	1100 1 REG		FMULP	ST(i),ST
DE	1101 1110	1101 0———		*(5)	
DE	1101 1110	1101 1000		保留	
DE	1101 1110	1101 1001		FCOMPP	
DE	1101 1110	1101 101—		保留	
DE	1101 1110	1101 11——		保留	
DE	1101 1110	1110 0 REG		FSUBRP	ST(i),ST
DE	1101 1110	1110 1 REG		FSUBP	ST(i),ST
DE	1101 1110	1111 0 REG		FDIVRP	ST(i),ST
DE	1101 1110	1111 1 REG		FDIVP	ST(i),ST
DF	1101 1111	MOD 000 R/M	SIB,位移量	FILD	字整数
DF	1101 1111	MOD 001 R/M	SIB,位移量	保留	
DF	1101 1111	MOD 010 R/M	SIB,位移量	FIST	字整数
DF	1101 1111	MOD 011 R/M	SIB,位移量	FISTP	字整数
DF	1101 1111	MOD 100 R/M	SIB,位移量	FBLD	压缩的BCD码
DF	1101 1111	MOD 101 R/M	SIB,位移量	FILD	长整数
DF	1101 1111	MOD 110 R/M	SIB,位移量	FBSTP	压缩的BCD码
DF	1101 1111	MOD 111 R/M	SIB,位移量	FISTP	长整数
DF	1101 1111	1100 0 REG		*(6)	
DF	1101 1111	1100 1 REG		*(7)	
DF	1101 1111	1101 0 REG		*(8)	
DF	1101 1111	1101 1 REG		*(9)	
DF	1101 1111	1110 0000		FSTSW AX	
DF	1101 1111	1110 0001		保留	
DF	1101 1111	1110 001—		保留	
DF	1101 1111	1110 01——		保留	
DF	1101 1111	1110 1———		保留	
DF	1101 1111	1111 ————		保留	

注：

SIB　SS index base 的缩写。请看图 H.1 和表 H.2。

*　该编码不由语言翻译器产生。但如果 80387 在指令流中遇上该指令，则按如下执行：

(1) FSTP ST(1)　　(6) FFREE ST(1) 以及弹出堆栈

(2) FCOM ST(1)　　(7) FXCH ST(1)

(3) FCOMP ST(1)　　(8) FSTP ST(1)

(4) FXCH ST(1)　　(9) FSTP ST(1)

(5) FCOMP ST(1)

**　该编码能由语言翻译器产生；然而 80387 把它们当作 FNOP 处理。它们和下列 8087，80287 指令对应。

(1) FENI

(2) FDISI

(3) FSETPM

***　传送的操作数的尺寸取决于对该指令起作用的 80386 操作数尺寸属性。

附录 J　80387 对于 80386 指令集的扩展

80387 指令不外乎表 J.1 所示的五种格式。指令至少有两个字节，并以 11011b 打头，这是浮点指令特有的标记。访问内存操作数寻址方式和 80386 相同。

表 J.1　80387 指令格式

	指令									可选的字节	
	第一个字节				第二个字节						
1	11011	OPA		1	MOD		1	OPB	R/M	SIB	DISP
2	11011	MF		OPA	MOD		OPB		R/M	SIB	DISP
3	11011	d	p	OPA	1	1	OPB		ST(i)		
4	11011	0	0	1	1	1	1	OP			
5	11011	0	1	1	1	1	1	OP			
	15～11	10	9	8	7	6	5	4 3 2 1 0			

注：

OP＝指令操作码，有时又分成 OPA 和 OPB 两个字段。

MF＝内存格式

00　32 位实数

01　32 位整数

10　64 位实数

11　64 位整数

p＝出栈

0　不出栈

1　操作之后弹至栈外

ESC＝11011

d＝目标

0　目标为 ST(0)

1　目标为 ST(i)

ST(i)＝寄存器堆栈元素 i

000　栈顶

001　第二个栈元素

⋮

⋮

111　第八个栈元素

MOD 和 R/M 字段的解释和 80386 一样，请看附录 H。

SIB 字节(Scale Index Base)和 DISP(displacement)有时出现在含有 MOD 和 R/M 字段的指令之中。它们的出现取决于 MOD 和 R/M 的值，这也和 80386 一样。

表 J.2 的指令摘要含有下列五点假定：

(1) 指令已被预取指和解码，并准备执行。

(2) 总线周期不要求等待状态。

(3) 没有要求处理器延迟存取总线的本地总线 HOLD 请求。

(4) 在指令执行期间未检测到异常。

(5) 在计算有效地址时，不使用两个寄存器的内容。如果使用到两个寄存器内容的话，则时钟另须加一。

表 J.2 80387 对 80386 指令集的扩展

指令	编码			时钟周期范围			
	字节 0	字节 1	可选字节 2～6	32 位实数	32 位整数	64 位实数	64 位整数
数据传送							
FLD=装入(注 a)							
整数/实数内存到 ST(0)	ESC MF 1	MOD 000R/M	SIB/DISP	20	45—52	25	61—65
长整数内存到 ST(0)	ESC 111	MOD 101R/M	SIB/DISP		56—67		
扩展的实数内存到 ST(0)	ESC 011	MOD 101R/M	SIB/DISP		44		
BCD 内存到 ST(0)	ESC 111	MOD 100R/M	SIB/DISP		266—275		
ST(i)to ST(0)	ESC 001	11000 ST(i)			14		
FST=存储							
ST(0)到整数/实数内存	ESC MF1	MOD 010 R/M	SIB/DISP	44	79—93	45	82—95
ST(0)到 ST(i)	ESC 101	11010 ST(i)			11		
FSTP=存储并且出栈							
ST(0)到整数/实数内存	ESC MF 1	MOD 011 R/M	SIB/DISP	44	79—93	45	82—95
ST(0)到长整数内存	ESC 111	MOD 111 R/M	SIB/DISP		80—97		
ST(0)到扩展的实数	ESC 011	MOD 111 R/M	SIB/DISP		53		
ST(0)到 BCD 内存	ESC 111	MOD 110 R/M	SIB/DISP		512—534		
ST(0)到 ST(i)	ESC 101	11001 ST(i)			12		
FXCH=交换							
ST(i)和 ST(0)	ESC 001	11001 ST(i)			18		
比较							
FCOM=比较							
整数/实数内存到 ST(0)	ESC MF 0	MOD 010 R/M	SIB/DISP	26	56—63	31	71—75
ST(i)到 ST(0)	ESC 000	11010 ST(i)			24		
FCOMP=比较并且出栈							
整数/实数内存到 ST	ESC MF 0	MOD 011 R/M	SIB/DISP	26	56—63	31	71—75
ST(i)到 ST(0)	ESC 000	11011 ST(i)			26		
FCOMPP=比较并出栈两次							
ST(1)到 ST(0)	ESC 110	1101 1001			26		
FTST=测试 ST(0)	ESC 001	1110 0100			28		
FUCOM=无序比较	ESC 101	11100 ST(i)			24		
FUCOMP=无序比较并出栈	ESC 101	11101 ST(i)			26		

续表

指令	编码 字节 0	字节 1	可选字节 2～6	时钟周期范围 32位实数	32位整数	64位实数	64位整数
FUCOMPP=无序比较并出栈两次							
	ESC 010	1110 1001		26			
FXAM=检验 ST(0)	ESC 001	11100101		30—38			
常数							
FLDZ=将+0.0装入 ST(0)	ESC 001	1110 1110		20			
FLD1=将+1.0装入 ST(0)	ESC 001	1110 1000		24			
FLDPI=将 π 装入 ST(0)	ESC 001	1110 1011		40			
FLDL2T=将 $\log_2(10)$装入 ST(0)	ESC 001	1110 1001		40			
FLDL2E=装入 $\log_2(e)$到 ST(0)	ESC 001	1110 1010		40			
FLDLG2=将 lg(2)装入 ST(0)	ESC 001	1110 1100		41			
FLDLN2=将 ln(2)装入 ST(0)	ESC 001	1110 1101		41			
算术指令							
FADD=加							
整数/实数内存和 ST(0)	ESC MF 0	MOD 000 R/M	SIB/DISP	24—32	57—72	29—37	71—85
ST(i)和 ST(0)	ESC dP 0	11000 ST(i)		23—31[b]			
FSUB=减							
整数/实数内存和 ST(0)	ESC MF 0	MOD 10 R R/M	SIB/DISP	24—32	57—82	28—36	71—83[c]
ST(i)和 ST(0)	ESC dP 0	1110 R R/M		26—34[d]			
FMUL=乘							
整数/实数内存和 ST(0)	ESC MF 0	MOD 001 R/M	SIB/DISP	27—35	61—82	32—57	76—87
ST(i)和 ST(0)	ESC dP 0	1100 1 R/M		29—57[e]			
FDIV=除							
整数/实数内存和 ST(0)	ESC MF 0	MOD 11 R R/M	SIB/DISP	89	120—127[f]	94	136—140[g]
ST(i)和 ST(0)	ESC dP 0	1111 R R/M		88[h]			
FSQRT[1]=平方根	ESC 001	1111 1010		122—129			
FSCALE=ST(0)乘以 2 的 ST(1)							
次方	ESC 001	1111 1101		67—86			
FPREM=部分余数	ESC 001	1111 1000		74—155			
FPREM1=部分余数							
(IEEE)	ESC 001	1111 0101		95—185			
FRNDINT=ST(0)舍入为整数	ESC 001	1111 1100		66—80			
FXTRACT=摘取 ST(0)的内容							
	ESC 001	1111 0100		70—76			
FABS=ST(0)的绝对值	ESC 001	1110 0001		22			
FCHS=改变 ST(0)的符号	ESC 001	1110 0000		24—25			
超越函数							
FCOS[k]=ST(0)的余弦	ESC 001	1111 1111		123—772[j]			
FPTAN[k]=ST(0)的部分正切	ESC 001	1111 0010		191—497[j]			
FPATAN=部分反正切	ESC 001	1111 0011		314—487			

续表

指令	编码			时钟周期范围			
	字节 0	字节 1	可选字节 2~6	32 位实数	32 位整数	64 位实数	64 位整数
FSIN[k]=ST(0)的正弦	ESC 001	1111 1110		122—771[j]			
FSINCOS[k]=ST(0)的正弦和余弦	ESC 001	1111 1011		194—809[j]			
F2XM1[i]=$2^{ST(0)}-1$	ESC 001	1111 0000		211—476			
FYL2X[m]=ST(1)*$\log_2$(ST(0))	ESC 001	1111 0001		120—538			
FYL2XP1[n]=ST(1)*$\log_2$(ST(0)+1.0)	ESC 001	1111 1001		257—547			
处理器控制							
FINIT=初始化 NPX	ESC 011	1110 0011		33			
FSTSW AX=存储状态字	ESC 111	1110 0000		13			
FLDCW=装入控制字	ESC 001	MOD 101 R/M	SIB/DISP	19			
FSTCW=存储控制字	ESC 101	MOD 111 R/M	SIB/DISP	15			
FSTSW=存储状态字	ESC 101	MOD 111 R/M	SIB/DISP	15			
FCLEX=清除异常	ESC 011	1110 0010		11			
FSTENV=存储环境	ESC 001	MOD 110 R/M	SIB/DISP	103—104			
FLDENV=装入环境	ESC 001	MOD 100 R/M	SIB/DISP	71			
FSAVE=保存状态	ESC 101	MOD 110 R/M	SIB/DISP	375—376			
FRSTOR=恢复状态	ESC 101	MOD 100 R/M	SIB/DISP	308			
FINCSTP=栈指针加一	ESC 001	1111 0111		21			
FDECSTP=栈指针减一	ESC 001	1111 0110		22			
FFREE=释放 ST(i)	ESC 101	1100 0 ST(i)		18			
FNOP=空操作	ESC 001	1101 0000		12			

阴影部分的指令对 80287/8087 无效。

注：

a. 当从内存装入单精度或双精度零时，要加 5 个周期。

b. 若 d=1，该范围加 3 个时钟。

c. 若 R=1，每个范围加 1 个时钟。

d. 若 d=0，该范围加 3 个时钟。

e. 典型值为 52(若 d=0，为 46－54，典型值为 49)。

f. 若 R=1，该范围加 1 个时钟。

g. 若 R=1，为 135—141。

h. 若 d=1，该范围加 3 个时钟。

i. $-0 \leqslant ST(0) \leqslant +\infty$。

j. 这些时钟周期范围仅对 $|x|<\pi/4$ 的操作数而言。若操作数超出了这个范围，则须增加至多 76 个时钟，以降低操作数的绝对值。

k. $0 \leqslant |ST(0)| < 2^{63}$。

l. $-1.0 \leqslant ST(0) \leqslant 1.0$。

m. $0 \leqslant ST(0) < +\infty$，$-\infty < ST(1) < +\infty$。

n. $0 \leqslant |ST(0)| < (2-SQRT(2))/2$，$-\infty < ST(1) < +\infty$。