第 4 章　80x86 的寻址方式与基本指令

本章主要介绍在应用程序中常用的 80x86 基本指令内容。包括：

- 操作数的寻址方式和转移目标位置的寻址方式。
- 数据处理类指令：数据传送指令、算术指令、逻辑指令、串处理指令。
- 控制转移类指令。
- 处理器控制类指令。

全面而准确地理解每条指令的功能和应用，是编写汇编语言程序的关键，所以，在学习指令时要注意以下几个方面。

- 指令的功能——该指令能够实现何种操作。通常指令助记符就是指令功能的英文单词或其缩写形式。
- 指令中操作数的寻址方式——该指令中的操作数可以采用何种寻址方式。
- 指令对标志的影响——该指令执行后是否对各个标志位有影响，以及如何影响。
- 其他方面——该指令其他需要特别注意的地方，如指令执行时的约定设置、必须预置的参数，隐含使用的寄存器等。
- 在叙述指令时，使用符号约定如下：

src　源操作数(Source)。

dst　目的操作数(Destination)

imm　立即数。8/16/32 位立即数记作：*imm*8/*imm*16/*imm*32。

reg　通用寄存器。8/16/32 位寄存器记作：*reg*8/*reg*16/*reg*32。

seg　段寄存器，代表的是 CS, DS, SS, ES, FS, GS。

mem　内存单元。8/16/32 位等内存单元记作：*mem*8/*mem*16/*mem*32。

acc　累加器。8/16/32 位累加器对应 AL/AX/EAX。

cnt　计数器。8/16/32 位计数器对应 CL/CX/ECX。

4.1　指令系统概述

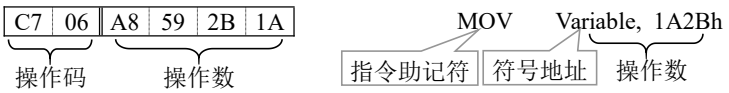
CPU 是计算机系统中的控制和执行部件，它的所有功能最终都可分解为若干个基本的操作，如数据传送、加法、减法、转移等。这些 CPU 可以执行的基本操作便是**计算机指令**，所有这些的基本操作构成了计算机的**指令系统**，它反映了计算机所具有的最基本的硬件功能。

和数据表示一样，在计算机内部，指令也必须用二进制编码来表示，才能为 CPU 所识别，并执行指令所指示的操作。

用二进制编码表示的指令叫做**机器指令**，它通常是若干个字节，由**操作码**和**操作数**两部分组成。操作码确定计算机要执行的具体操作，如传送、运算、移位、转移等，是指令中不可缺少的组成部分；操作数指出指令执行时所需要的操作数据，它可以是数的本身，也可以是存放数的存储单元(寄存器或内存)。

例如，如图 4.1(a)所示的一条 8086 机器指令。其中操作码是“C7 06”，表示执行的是“16 位数据传送”操作；操作数是“A8 59 2B 1A”，指出了两个操作数据：16 位数 1A2Bh 和 2 字节内存单元[59A8h]。该指令功能是：将 1A2Bh 传送到 2 字节单元[1000h]。

当使用“助记符”来表示操作码(称为**指令助记符**)，如用“MOV”来表示上述操作码；用符号来表示内存单元地址(称为**符号地址**)，如用“Variable”表示上述字单元的地址，机器指令就被“符号化”了，如图 4.1(b)所示。经过符号化的机器指令就是**汇编指令**，它与机器指令有着直接对应关系。



(a) 机器指令　　(b) 汇编指令

图 4.1　机器指令与汇编指令示例

计算机中的指令有些不需要操作数，大多数指令采用一个或两个操作数，还有少数指令需要更多的操作数。大多数运算型指令需要两个操作数，例如：MOV Variable, 1A2Bh。这条指令将 16 位数 1A2Bh 存入 Variable，其中 1A2Bh 是指令处理数据的来源，称为**源操作数**，Variable 供指令存放处理的结果，称为**目的操作数**。在 80x86 汇编指令中，目的操作数一般在最左边，源操作数在目的操作数的右边。

操作数可以放在操作码之后，称为**立即数**；也可以存放在 CPU 内部的寄存器中，称为**寄存器操作数**；大多数情况下操作数是存放在内存中，称为**内存操作数**。指令指定了操作数的位置，在执行时根据指定的位置找到所需的数据。这种寻找操作数的过程称为**寻址**，而寻找操作数的方法称为**寻址方式**。操作数有多种寻址方式，详见 4.2.1 节。

通常所说的计算机指令都是指汇编格式的指令，一台计算机所支持的全部指令，就是该计算机的**指令系统**。根据指令的功能，可将指令分类为：数据传送、算术逻辑运算类、移位操作类、转移类、堆栈操作类、输入输出类等。不同类型的计算机具有不同的指令系统，有的计算机指令系统很简单，只有二三十条指令，有的计算机指令系统复杂，支持的指令多达几百条，但是无论繁简，其所支持的指令都可分成这几类。

80x86 系列在不断发展，相应指令也在不断丰富。最初的 8086 指令系统有 117 条指令，如今的 Pentium 4 除了最初的 8086 指令集外，还具有浮点运算、MMX、3DNow、SSE、SSE2、SSE3 等多个扩展指令集。这本教材只是讲述汇编语言程序设计，所以只介绍 80x86 指令系统中常用的基本指令，其它指令的介绍，参见相应的资料手册。

4.2　数据处理类指令

计算机程序对数据的处理主要由数据处理类指令来完成，主要包括：数据传送指令、算术指令、逻辑指令、串处理指令等。

4.2.1　操作数的寻址方式

数据处理指令所处理的数据从何处取，处理的结果放到何处去？这些便是操作数寻址的问题。一般来说，存在多种方法获取操作数据及操作结果的存放位置，这些方法统称为操作数寻址方式。

例如，需要完成的操作是：将 1A2Bh 传送到 DX 中去。那么从何处能取到 1A2B 呢？如果 1A2B 放在操作码后面，那么可直接从指令中取到，这便是**立即寻址方式**；如果 1A2B 放在寄存器中，如放在 AX 中，那么指令执行时可从 AX 中取到，也就是说源操作数是**寄存器寻址方式**；如果 1A2B 是存于[0200h]单元中，那么执行时可以从字单元[0200h]中取到，此处的源操作数是**内存操作数寻址方式**。同样，该操作处理的结果是放到寄存器 DX 中去，所以此处的目的操作数也寄存器寻址方式。

为了方便，我们以“MOV *dst*, *src*”形式的指令来说明操作数的寻址方式。这是一条数据传送指令，其功能是：将源操作数 *src* 传送到目的操作数 *dst* 中。

一般来说，操作数据及操作结果主要有三种存放形式：放在指令的操作数中，存放在寄存器中，存放在内存单元中。与之对应的有三种操作数：立即操作数、寄存器操作数和内存操作数。因而有三种基本寻址方式：立即寻址方式、寄存器寻址方式和内存操作数寻址方式(又包括多种寻址方式)。除此之外，80x86 还有少数隐含操作指令、端口输入输出指令等，这些将在后续章节中陆续介绍。

1. 立即寻址

操作数作为指令的一部分直接放在指令操作码之后，这种操作数称为立即数。在实地址模式中，立即数可以是 8 位或 16 位；在保护模式中，立即数可以是 8 位或 32 位。

在指令手册中，立即数一般用 *imm* 表示，8/16/32 位立即数记作：*imm*8/*imm*16/*imm*32。

例 4.1　下列汇编指令中的源操作数均为立即寻址方式。

(1) MOV　DX, 1A2Bh

(2) MOV　Variable, 1A2Bh

指令(2)经汇编后的机器代码如图 4.2 所示，1A2B 放在操作码之后。

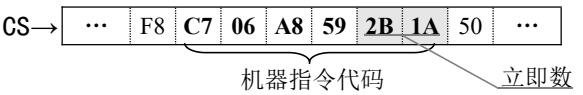


图 4.2　指令(2)经汇编后的机器代码示意图

立即数不对应任何寄存器或内存单元，所以立即数只能用作源操作数。

2. 寄存器寻址

操作数存放在寄存器中。在机器格式指令中，寄存器以编号形式出现在指令代码中，在汇编指令中，则直接用寄存器名来表示操作数。

在指令手册中，寄存器寻址一般用 *reg* 符号表示，可以是 *reg*8：AL, AH, BL, BH, CL, CH, DL, DH，也可以是 *reg*16：AX, BX, CX, DX, SP, BP, SI, DI，还可以是 *reg*32：EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI。此外，段寄存器 CS, DS, SS, ES, FS, GS 也可作为寄存器操作数使用。

例 4.2　下列指令中的操作数使用了寄存器寻址。

(1) MOV　Variable，AX　　；源操作数是寄存器寻址方式

(2) MOV　EDX, 12345678h　；目的操作数是寄存器寻址方式

使用寄存器寻址方式的操作数，相应的操作数据已存于寄存器中，或执行的结果数据存入目的地是寄存器，在执行时不用访问内存单元，所以，寄存器寻址方式要比内存操作数寻址方式的执行速度快。

3. 内存操作数寻址

实际上，大多数情况下操作数是存放在内存单元中，此时在指令中需要给出存放操作数的内存单元地址等信息。

在 80x86 汇编格式指令手册中，一般用符号 *mem* 表示内存操作数，8/16/32 位等内存操作数记作：*mem*8/*mem*16/*mem*32。

特别注意，**一条 80x86 指令最多只能有一个操作数使用内存操作数寻址**。

例 4.3 下列指令中的操作数使用了内存操作数寻址。

(1) MOV [59A8h], AX ；目的操作数是 59A8h 号字单元，是内存操作数寻址

(2) MOV EDX, [1000h] ；源操作数是 1000h 号双字单元，是内存操作数寻址

这里直接用地址编号来表示内存单元，但在汇编源程序中，一般很少直接使用地址编号，而是用符号来表示内存单元地址，例如，用 Variable 来表示地址编号 59A8h，那么指令：MOV [59A8h], AX，就写为：MOV Variable, AX。虽然它们形式有所不同，但本质上是一回事，都是使用内存操作数寻址。

内存操作数的单元地址可以直接用地址编号给出，也可全部或部分放在寄存器中。在例 4.3(1)中，直接用地址编号表示 59A8h 号字单元，即[59A8h]；如果将 59A8h 存放到 BX，那么该字单元也可表示为[BX]；如果 BX 中存放的是 59A0h，那么[BX+8]对应的也是 59A8h 号字单元。也就是说同一个内存单元可以有多种寻址方式。

在 80x86 中，内存单元的地址(偏移地址)EA＝基址+变址+位移量。三个分量可有不同组合，从而使用内存单元有以下 5 种不同的寻址方式。

1) 直接寻址

指令所需要的操作数据存放在内存单元中，指令码中直接包括了操作数的地址，放在指令操作码之后。即 EA=位置量。

直接寻址的汇编格式为：[地址编号] 或 符号地址 或 符号地址表达式

例 4.4 下列指令使用了直接寻址方式的操作数。

(1) MOV [100h]，2B1Ah ；目的操作数是直接寻址方式(直接地址)

(2) MOV Variable, 2B1Ah ；目的操作数是直接寻址方式(符号地址)

(3) MOV ES: Variable, 2B1Ah ；目的操作数是直接寻址方式(符号地址)

(4) MOV AX, Variable+4 ；源操作数是直接寻址方式(地址表达式)

(5) MOV EAX, [100Ah] ；源操作数是直接寻址方式(直接地址)

(6) MOV EAX, SS: [100Ah] ；源操作数是直接寻址方式(直接地址)

如果执行前数据段区和堆栈段区内容如图 4.3 所示，那么指令(5)执行后，EAX 内容为 44434241h；指令(6)执行后，EAX 内容为 038B9A04h(因为使用了不同的段寄存器)。

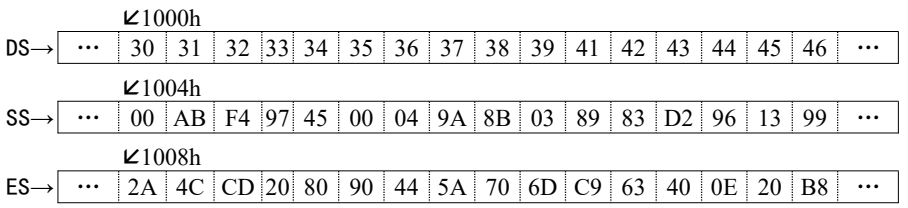


图 4.3 例 4.4～例 4.8 所用的各段内存区内容

2) 寄存器间接寻址

指令所需要的操作数据存放在内存中，其地址存放在寄存器中。即 EA=寄存器。

寄存器间接寻址的格式为：[寄存器]。

可用于间接寻址的寄存器有：BX, SI, DI 和 BP，以及所有的 32 位通用寄存器。注意，(E)BP, ESP 用作间接寻址寄存器时，则默认的段寄存器是 SS。

例 4.5　指出在下列指令中使用寄存器间接寻址方式的操作数。

```
(1)  MOV    [BX], AX          ; 目的操作数是寄存器间接寻址方式
(2)  MOV    [EBP], CX         ; 目的操作数是寄存器间接寻址方式
(3)  MOV    EAX, [BP]         ; 源操作数是寄存器间接寻址方式
(4)  MOV    EAX, ES: [BP]     ; 源操作数是寄存器间接寻址方式
(5)  MOV    EAX, [EBX]        ; 源操作数是寄存器间接寻址方式
```

如果执行前，EBP 内容为 0800100Ch，EBX 内容为 00001006h，数据段区、堆栈段区和附加段区内容如图 4.3 所示，那么指令(3)执行后，EAX 内容为 8389038Bh；指令(4)执行后，EAX 内容为 5A449080h；指令(5)执行后，EAX 内容为 39383736h。

3) 寄存器相对寻址

这种寻址也叫直接变址寻址。指令所需要的操作数据存放在内存中，内存单元地址是寄存器内容与指令中指定的位移量之和，即：EA＝[寄存器]+位移量。

寄存器相对寻址的格式为：disp[寄存器]　或　[寄存器+disp]。

其中，disp 是位移量，是指令指定的一个 8 位或 16/32 位的补码数。

可用于相对寻址的寄存器有：BX, SI, DI 和 BP，以及所有的 32 位通用寄存器。注意，(E)BP、ESP 用作相对寻址寄存器时，则默认的段寄存器是 SS。

例 4.6　下列指令中使用了寄存器相对寻址方式的操作数。

```
(1)  MOV    4[BX], AX         ; 目的操作数是寄存器相对寻址方式
(2)  MOV    −4[EBP], CX       ; 目的操作数是寄存器相对寻址方式
(3)  MOV    AX, [BP−2]        ; 源操作数是寄存器相对寻址方式
(4)  MOV    AX, ES: −2[BP]    ; 源操作数是寄存器相对寻址方式
(5)  MOV    AX, 8[EBX]        ; 源操作数是寄存器相对寻址方式
```

如果执行前，EBP 内容为 0800100Ch，EBX 内容为 00001006h，数据段区、堆栈段区和附加段区内容如图 4.3 所示，那么指令(3)执行后，AX 内容为 9A04h，取自堆栈段区字单元[100Ah]；指令(4)执行后，AX 内容为 20CDh，取自附加段区字单元[100Ah]；指令(5)执行后，AX 内容为 46455h，取自数据段内存区字单元[100Eh]。

4) 基址变址寻址

指令所需要的操作数据存放在内存中，内存单元地址是基址寄存器内容与变址寄存器内容之和，即：EA=基址寄存器+变址寄存器。

基址变址寻址的格式为：　(1) [基址寄存器][变址寄存器]

(2) [基址寄存器+变址寄存器]

可用作基址寄存器的有：BX 和 BP，以及所有的 32 位通用寄存器；可用作变址寄存器的有：SI 和 DI，以及除 ESP 之外的所有 32 位通用寄存器。注意，(E)BP, ESP 用作基址变址寻址寄存器时，则默认的段寄存器是 SS。

例 4.7　下列指令中使用了基址变址寻址方式的操作数。

```
(1) MOV     [BX][SI], AX        ; 目的操作数是基址变址寻址方式
(2) MOV     [BP+SI], CX         ; 目的操作数是基址变址寻址方式
(3) MOV     AX, [BP][DI]        ; 源操作数是基址变址寻址方式
(4) MOV     AX, DS: [BP][DI]    ; 源操作数是基址变址寻址方式
(5) MOV     AX, [EAX+EDX]       ; 源操作数是基址变址寻址方式
```

如果执行前，BP 内容为 100Ch，DI 内容为 0002h，数据段区、堆栈段区如图 4.4 所示，那么，指令(3)执行后，AX 内容为 8389h，取自堆栈段内存区的单元[100Eh]；指令(4)执行后，AX 内容为 4645h，取自数据段内存区的单元[100Eh]。

5) 相对基址变址寻址

指令所需要的操作数据存放在内存中，内存单元地址是基址寄存器内容、变址寄存器内容以及位移量之和，即：EA＝基址寄存器 + 变址寄存器 + 位移量。

基址变址寻址的格式为：(1) disp[基址寄存器][变址寄存器]

(2) [基址寄存器+变址寄存器+disp]

其中，disp 是位移量，是指令指定的一个 8 位或 16/32 位的补码数。

可用作基址寄存器的有：BX 和 BP，以及所有的 32 位通用寄存器；可用作变址寄存器的有：SI 和 DI，以及除 ESP 之外的所有 32 位通用寄存器。注意，(E)BP, ESP 用作基址变址寻址寄存器时，则默认的段寄存器是 SS。

例 4.8 下列指令中使用了相对基址变址寻址方式的操作数。

```
(1) MOV     100[BX][DI], AX     ; 目的操作数是基址变址寻址方式
(2) MOV     100[BP+DI], CX      ; 目的操作数是基址变址寻址方式
(3) MOV     AL, 2[BP][SI]       ; 源操作数是基址变址寻址方式
(4) MOV     AL, DS: 2[BP][SI]   ; 源操作数是基址变址寻址方式
(5) MOV     AL, 20[EAX+EDX]     ; 源操作数是基址变址寻址方式
```

如果执行前，BP 内容为 100Ch，SI 内容为 0001h，数据段区、堆栈段区如图 4.3 所示，那么，指令(3)执行后，AL 内容为 83h，取自堆栈段区的单元[100Fh]；指令(4)执行后，AL 内容为 46h，取自数据段区的单元[100Fh]。

4.2.2 数据传送指令

高级语言中的赋值语言一般由数据传送类指令来实现。数据传送指令的主要功能是将数据传送到寄存器或内存单元中，又可分为 4 类：通用数据传送指令——MOV, PUSH, POP, XCHG 等；地址传送指令——LEA, LDS, LES 等；标志位传送指令——LAHF, SAHF, PUSHF, POPF 等；输入输出指令——IN, OUT 等。为了方便，将类型转换指令也放在这一节。这些指令除了和标志位有关的传送指令外，均不影响标志位。

1. 通用数据传送指令

数据传送类指令是使用最频繁的一类指令，主要包括以下指令：

```
基本传送  MOV          传送
          XCHG         交换
          MOVZX        符号扩展传送
          MOVSX        零扩展传送
堆栈类    PUSH         压栈
```

```
          POP              出栈
          PUSHA/PUSHAD     所有 16/32 位通用寄存器进栈
          POPA/POPAD       所有 16/32 位通用寄存器出栈
其他      XLAT,            查表换码
```

1) MOV 指令(MOVe)

格式：MOV 目的操作数，源操作数。

功能：数据传送，将源操作数传送到目的储存单元中，源地址单元内容不变。

操作数的寻址方式：

```
MOV    reg/mem, imm        ; 立即数⇒寄存器/内存
MOV    reg/mem/seg, reg    ; 寄存器⇒寄存器/内存/段寄存器
MOV    reg/seg, mem        ; 内存⇒寄存器/段寄存器
MOV    reg/mem, seg        ; 段寄存器⇒寄存器/内存
```

关于操作数的寻址方式可参照 80x86 的指令手册。在编写程序时，若操作数不符合寻址要求，则汇编时不通过。此外需要注意：一般情况下，源、目的操作数类型长度要一致；立即数要在目的操作数类型值范围内；当两个操作数类型均不明确时，必须至少指定一个操作数类型。为方便以后叙述，将指定类型的伪操作符罗列如下：(伪操作符有关概念在以后章节中叙述)

Byte Ptr	指定数据类型为字节类型(8 位数据)
Word Ptr	指定数据类型为 2 字节(字)类型(16 位数据)
DWord Ptr	指定数据类型为 4 字节(双字)类型(32 位数据)
FWord Ptr	指定数据类型为 6 字节类型(48 位数据)
QWord Ptr	指定数据类型为 8 字节(四字)类型(64 位数据)
TByte Ptr	指定数据类型为 10 字节类型(80 位数据)

例 4.10　指出下列汇编指令是否正确。

```
MOV CS, DX           ; 错误, CS 不能用作目的操作数
MOV DX, CS           ; 正确
MOV DS, DX           ; 正确
MOV AL, 0A0Dh        ; 错误, 0A0Dh 超过 8 位数范围
MOV Variable, [SI]   ; 错误, 不支持 MOV mem, mem, 即内存⇒内存
MOV EAX, DX          ; 错误, 两操作数类型不一致
MOV [EAX], 41h       ; 类型不明确。可改为 MOV [EAX], DWord Ptr 41h
MOV DS, 1000h        ; 错误, 不支持 MOV seg, imm, 即立即数⇒段寄存器
MOV GS, CS           ; 错误, 不支持 MOV seg, seg, 即段寄存器⇒段寄存器
```

例 4.11　将双字单元[1000h]的内容传送到双字单元[2000h]中。

解：80x86 不支持 MOV *mem*, *mem* 指令格式，故用两条 MOV 指令来实现。

```
MOV EAX, DWord Ptr [1000h]  ; [1000h]⇒EAX
MOV DWord Ptr [2000h], EAX  ; EAX⇒[2000h]
```

2) XCHG 指令(eXCHanGe)

格式：XCHG 操作数 1，操作数 2。

功能：数据交换，操作数 1 与操作数 2 单元的内容互相交换。

由于两操作数即是源操作数，又是目的操作数，故它们的位置顺序无关紧要。

操作数的寻址方式为：

```
XCHG    reg/mem, reg    ; 寄存器/内存⇔寄存器
```

例 4.12 指出下列汇编指令是否正确。

```
XCHG    EAX, EBX            ; 正确
XCHG    AX, 0A0Dh           ; 错误，不支持 XCHG  reg, imm
XCHG    Variable, [EAX]     ; 错误，不支持 XCHG  mem, mem
XCHG    DS, Word Ptr[BX]    ; 错误，不支持 XCHG  mem, seg
XCHG    DS, BX              ; 错误，不支持 XCHG  reg, seg
XCHG    AX, BL              ; 错误，类型长度不一致
```

例 4.13 将双字单元[1000h]中内容与双字单元[2000h]中内容互换。

解：80x86 不支持 XCHG *mem*, *mem* 寻址方式指令，故用如下指令实现。

```
MOV     EAX, [100h]     ; [100h] ⇒EAX
XCHG    [200h], EAX     ; EAX⇔ [200h]
```

5) PUSH 指令(PUSH onto the stack)

格式：PUSH 源操作数。

功能：数据进栈。当源操作数为 16 位时，(E)SP 内容减 2，即向低地址端调整 2 字节；当源操作数为 32 位时，(E)SP 中内容减 4，即向低地址端调整 4 字节。

注意，进栈数据只能是 16/32 位数据。8086CPU 只能是 16 位。

操作数的寻址方式：PUSH *reg*/*mem*/*seg*/*imm* (8086 不支持 PUSH *imm*)。

例 4.18 指出下列汇编指令是否正确。

```
PUSH    AX              ; 正确
PUSH    CS              ; 正确
PUSH    AL              ; 错误，进栈数据必须是 16 位或 32 位
PUSH    Word Ptr 41h    ; 在 32 位 CPU 中正确，在 8086 中是错误的
PUSH    125h            ; 类型不明确，可指定为 PUSH DWord Ptr 125h
```

6) POP 指令(POP from the stack)

格式：POP 目的操作数。

功能：数据出栈。从堆栈中弹出数据到目的操作数所确定的单元中。若操作数为 16 位，(E)SP 向高地址端调整 2 字节；若操作数为 32 位时，(E)SP 向高地址端调整 4 字节。

注意，出栈数据只能是 16/32 位数据。8086CPU 只能是 16 位。

操作数寻址方式：POP *reg*/*mem*/*seg*。

例 4.19 指出下列汇编指令是否正确。

```
POP AX  ; 正确
POP CS  ; 错误，CS 不能用作目的操作数
POP AL  ; 错误，出栈数据必须是 16 位或 32 位
POP 41h ; 错误，不支持 POP  imm
```

堆栈操作的主要指令是 PUSH 和 POP 指令。另外，由于堆栈操作与(E)SP 密切相关，所以，一般在程序中不要将(E)SP 作为数据寄存器等使用。

例 4.20 使用 PUSH 和 POP 指令，将字(16 位)单元[1000h]内容和字单元[2000h]内容分别放入 EAX 低 16 位和高 16 位。

解：先将两个 16 位数按存放顺序要求进栈，再从堆栈弹出一个 32 位数到 EAX。

```
PUSH    Word Ptr[2000h] ; EAX 高 16 位数进栈
PUSH    Word Ptr[1000h] ; EAX 低 16 位数进栈
```

```
POP     EAX                ; 弹出 32 位数⇒EAX
```

假设字单元[1000h]内容为 1A2B，字单元[2000h]内容为 3C4Dh，执行该指令序列时，堆栈状态及 EAX 内容变化如图 4.4 所示。

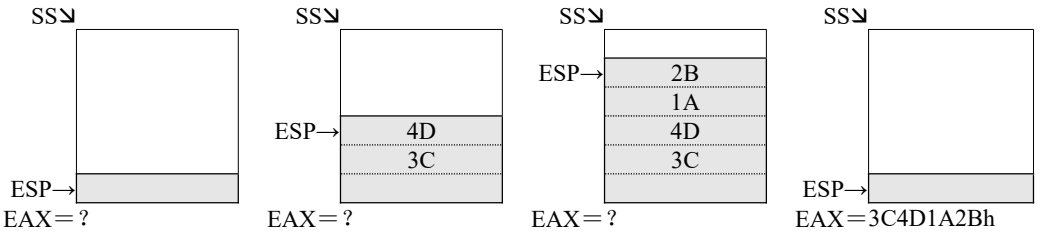


图 4.4　堆栈操作状态变化示意图

9) XLAT 指令(transLATe)

格式：XLAT。

功能：查表换码。将以(E)BX 基址，以 AL 内容为位移量的字节单元内容传送给 AL。

这条指令使用隐含操作数。在指令执行前约定：必须已经建立一个字节表，表首地址已经放入基址寄存器(E)BX；查找项的位移量已经放入 AL。

例 4.21　AL 存放的是一个 0～15 的数，用 XLAT 指令将它转换成十六进制数。

解：设数据区内从偏移地址 1000h 开始，连续 16 个字节单元中存放的是十六进制数(ASCII 码)，如图 4.3 所示，那么转换的指令如下：

```
MOV EBX, 1000h  ; 字节表首地址⇒EBX
XLAT            ; 查找数码⇒AL
```

若指令执行前 AL 内容为 0Bh，那么指令执行后 AL 内容为 42h('B'的 ASCII 码)。

2. 地址传送指令

1) LEA 指令(Load Effective Address)

格式：LEA 寄存器，源内存操作数。

功能：有效地址送寄存器。将内存操作数的偏移地址(EA)传送至目的寄存器中。

操作数的寻址方式为：LEA *reg*16/*reg*32, *mem*。

注意，目的操作数须是 16/32 位通用寄存器。若是 16 位寄存器则只装入 EA 的低 16 位。

例 4.22　说明下列两条指令的区别。

(1) LEA　AX, [EBX][ESI]

(2) MOV AX, [EBX][ESI]

解：指令(1)取源内存操作数的偏移地址 EA，并存入 AX，而指令(2)取 EA 所对应的内存 2 字单元内容，并存入 AX。前者取的是地址，而后者取的是单元中的内容。若 EBX＝1000h，ESI＝8h，假设数据区内容如图 4.3 所示，那么指令(1)执行后，AX 内容为 1008h，是内存单元的偏移地址；指令(2)执行后，AX 内容为 3938h，即内存单元[1008h]的内容。

2) LDS/LSS/LES/LFS/LGS 指令(Load DS/SS/ES/FS/GS with pointer)

这一组指令的格式和操作数寻址方式完全一样。

格式：LDS/LSS/LES/LFS/LGS 目的寄存器，源内存操作数。

功能：内存指针送寄存器和段寄存器指令。将源内存操作数中的低 2/4 字节内容传送到

目的寄存器，及高 2 字节内容传送至段寄存器 DS/SS/ES/FS/GS。

操作数的寻址方式： LDS/LSS/LES/LFS/LGS *reg*16, *mem*32 (16 位地址模式)；

LDS/LSS/LES/LFS/LGS *reg*32, *mem*48 (32 位地址模式)。

假设数据段区内容如图 4.3 所示，执行 LES AX, DWord Ptr[1001h]指令后(16 位模式)，ES 内容为 3433h，AX 内容为 3231h；执行 LES EAX, FWord Ptr[1001h]指令后(32 位模式)，ES 内容为 3635h，EAX 内容为 34333231h(仅作为示例，不考虑内容合法性)。

3. 标志位传送指令

1) LAHF 指令(Load AH with Flags)

格式：LAHF。

功能：标志送 AH。将 FLAGS 的低 8 位送至 AH，包括了 SF, ZF, AF, PF, CF。

2) SAHF 指令(Store AH into Flags)

格式：SAHF。

功能：AH 送标志寄存器指令。将 AH 内容送至标志寄存器低 8 位。

3) PUSHF/PUSHFD 指令(PUSH Flags onto the stack by word/Double word)

格式：PUSHF/PUSHFD。

功能：标志寄存器进栈指令。16/32 位标志寄存器 FLAGS/EFLAGS 内容进栈。

4) POPF/POPFD 指令(POP Flags from the stack by word/Double word)

格式：POPF/POPFD。

功能：标志寄存器出栈。从堆栈弹出 16/32 位数据到标志寄存器 FLAGS/EFLAGS。

4. 输入输出指令

80x86 有一组专门的输入输出指令来读写 I/O 端口，从而实现与外部设备交换数据。而且只能使用累加器来接收、发送数据。

1) IN 指令(INput)

格式：IN 累加器，端口地址。

累加器可以是 AL, AX, EAX；端口地址可以是 *imm*8，或存放在 DX 中。

功能：从端口输入。把 1/2/4 字节端口中的数据传送给 AL/AX/EAX。

80x86 系统的端口地址范围是 0000h～FFFFh。使用 *imm*8 形式端口，则指令中的端口地址范围只能是 00h～FFh；使用 DX 来存放端口地址，则指令中的端口地址范围是 0000h～FFFFh。此外，和内存操作数不同的是，IN 指令的源操作数据是来自 I/O 端口。

2) OUT 指令(OUTput)

格式：OUT 端口地址，累加器。

累加器可以是 AL, AX, EAX；端口地址可以是 *imm*8，或存放在 DX 中。

功能：端口输出指令。把 AL/AX/EAX 中的内容输送到 1/2/4 字节端口。

使用 *imm*8 形式端口，则指令中的端口地址范围只能是 00h～FFh；使用 DX 来存放端口地址，则指令中的端口地址范围是 0000h～FFFFh。此外，和内存操作数不同的是，OUT 指令的目的操作数据将传送到 I/O 端口中去。

例 4.23 说明下列 IN/OUT 指令执行功能。通过这个例子体会 IN/OUT 指令的用法。

```
IN  AL, 20h     ; 20h 端口的 1 字节内容⇒AL(imm8 直接)
```

```
IN  AX, 20h      ; 20h 端口、21h 端口中的 2 字节内容⇒AX(imm8 直接)
IN  AL, 378h     ; 378h 超出 imm8 范围，此指令不正确
MOV DX, 378h
IN  EAX, DX      ; 4 字节端口 378h 的内容⇒EAX，即依次取 378h~37Bh 端口数据(DX 间接)
OUT DX, AL       ; 将 AL 内容⇒378h 端口
OUT DX, AX       ; 将 AX 内容⇒378h 和 379h 端口
OUT DX, EAX      ; 将 EAX 内容⇒378h～37Ch 端口
```

5. 类型转换指令

1) CBW 指令(Convert Byte to Word)

格式：CBW。

功能：将字节类型数据转换成字类型指令。将 AL 内容符号扩展到 AH，形成 AX 中的 16 位数据。即若 AL 最高位为 0，则将 AH 置为 0；若 AL 最高位为 1，则将 AH 置为 FFh。

2) CWD 指令(Convert Word to Double-word)

格式：CWD。

功能：字类型转换为双字类型指令。将 AX 内容符号扩展到 DX，形成 DX:AX 中的 32 位数据。即若 AX 最高位为 0，则将 DX 置为 0；若 AX 最高位为 1，则将 DX 置为 FFFFh。

4.2.3　算术运算指令

算术运算指令用来执行加、减、乘和除四则运算。

由于 80x86 采用补码表示有符号数，所以对于加法、减法运算采用相同的运算规则，但是，对于乘法、除法运算，则需要使用不同的运算规则。

算术运算指令的执行结果一般会影响标志位 CF, ZF, SF, OF, AF, PF。所以，在学习中，一方面要掌握各指令的功能，另一方面要清楚指令对各标志位的影响。

1. 加法指令

基本的加法指令包括 3 条指令：ADD 加法、ADC 带进位加法和 INC 加 1。

80x86 采用补码表示有符号数，所以对于有符号数和无符号数采用相同的加法运算规则，也就是说，下面介绍的加法指令，既可用于无符号数运算，也可用于补码数运算。但要注意的是，对于无符号数加法，其运算结果溢出与否是通过 CF 指示出来的；对于补码数加法，其运算结果溢出与否是通过 OF 指示出来的。

1) ADD 指令(ADDition)

格式：ADD 目的操作数，源操作数。

功能：加法指令。将源操作数与目的操作数相加，结果存入目的操作数。

操作数的寻址方式：　ADD *reg*, *reg*/*mem*/*imm*；

　　ADD *mem*, *reg*/*imm*。

注意，该指令根据执行的结果设置 CF, AF, PF, ZF, SF, OF 的状态。

2) ADC 指令(ADd with Carry)

格式：ADC 目的操作数，源操作数。

功能：带进位加法指令。即将源操作数、目的操作数和 CF 相加，结果存入目的操作数。

操作数的寻址方式：　ADC *reg*, *reg*/*mem*/*imm*；

ADC *mem*, *reg*/*imm*。

该指令根据执行的结果设置 CF, AF, PF, ZF, SF, OF 状态。

3) INC 指令(INCrement)

格式：INC 操作数。

功能：加 1 指令。操作数自身加 1，即将操作数加 1，结果再存入操作数。

操作数的寻址方式：INC *reg*/*mem*。

与前两条指令不同的是，该指令影响 AF, PF, ZF, SF 和 OF，但不影响 CF。

例 4.24 完成两个 64 位数相加，被加数存放在[1000h]单元中，加数放在[2000h]单元中，结果放到[3000h]单元中。

解：用两次 32 位加指令来完成，在作高 32 位加法时，须考虑从低 32 位传来的进位，所以采用 ADC 指令。指令序列如下：

```
MOV ESI, 0  ; 0⇒EAX
MOV EAX, 1000h[ESI*4]   ; 取被加数的低 32 位
ADD EAX, 2000h[ESI*4]   ; 低 32 位相加
MOV 3000h[ESI*4], EAX   ; 保存结果的低 32 位
INC ESI                 ; 指向下一个 32 位
MOV EAX, 1000h[ESI*4]   ; 取被加数的高 32 位
ADC EAX, 2000h[ESI*4]   ; 高 32 位相加
MOV 3000h[ESI*4], EAX   ; 保存结果的高 32 位
```

2. 减法指令

基本的减法指令包括 5 条指令：SUB 减法、SBB 带借位减法、DEC 减 1、NEG 求补、CMP 比较。

减法指令既可用于无符号数运算，也可用于补码数运算。对于无符号数减法，其运算结果溢出与否是通过 CF 指示出来的；对于补码数减法，其运算结果溢出与否是通过 OF 指示出来的。

1) SUB 指令(SUBtraction)

格式：SUB 目的操作数，源操作数。

功能：减法指令。即目的操作数减去源操作数，结果存入目的操作数。

操作数的寻址方式： SUB *reg*, *reg*/*mem*/*imm*；

SUB *mem*, *reg*/*imm*。

该指令根据执行的结果设置 CF, AF, PF, ZF, SF, OF 的状态。

2) SBB 指令(SuBtract with Borrow)

格式：SBB 目的操作数，源操作数。

功能：带借位减法指令。即目的操作数减去源操作数，再减去 CF，结果存入目的操作数。

操作数的寻址方式： SBB *reg*, *reg*/*mem*/*imm*；

SBB *mem*, *reg*/*imm*。

该指令根据执行的结果设置 CF, AF, PF, ZF, SF, OF 的状态。

3) DEC 指令(DECrement)

格式：DEC 操作数。

功能：减 1 指令。操作数自身减 1，即操作数减去 1，结果再存入操作数。

操作数的寻址方式：DEC *reg*/*mem*。

与前两条指令不同的是，该指令影响 AF, PF, ZF, SF 和 OF，但不影响 CF。

4) NEG 指令(NEGate)

格式：NEG 操作数。

功能：求补指令。操作数各位取反再加 1(求补)，即将 0 减去操作数，结果存入操作数。

操作数的寻址方式：NEG *reg*/*mem*。

该指令影响 CF, AF, PF, ZF, SF, OF。只有当操作数为 0 时，求补运算的结果才使 CF＝0，其他情况则均为 1；只有当操作数为−128(8 位运算)或−32 768(16 位运算)或-2^{31}(32 位运算)时，求补运算的结果才使 OF＝1，其他情况则均为 0。

5) CMP 指令(CoMPare tow Operand)

格式：CMP 目的操作数，源操作数。

功能：比较指令。两操作数比较大小，根据目的操作数减去源操作数的运算结果，从而影响标志位。

操作数的寻址方式：　CMP *reg*, *reg*/*mem*/*imm*；

　　CMP *mem*, *reg*/*imm*。

该指令影响 CF, AF, PF, ZF, SF, OF。

这条指令除了相减结果不保存外，其他情况与 SUB 指令完全相同。

例 4.25　完成 64 位数相减，被减数存放在[1000h]单元中，减数放在[2000h]单元中，结果放到[3000h]单元中。

解：通过两次 32 位减操作来完成，其中进行高 32 位减法时，须考虑从低 32 位传来的借位，所以采用 SBB 指令。编制的指令如下：

```
SUB ESI, ESI             ; 0⇒EAX
MOV EAX, 1000h[ESI*4]    ; 取被减数的低 32 位
SUB EAX, 2000h[ESI*4]    ; 低 32 位相减
MOV 3000h[ESI*4], EAX    ; 保存结果的低 32 位
MOV EAX, 1000h[ESI*4+4] ; 取被减数的高 32 位
SBB EAX, 2000h[ESI*4+4] ; 高 32 位相减
MOV 3000h[ESI*4+4], EAX ; 保存结果的高 32 位
```

3. 乘法指令

和加法、减法指令不一样，对于无符号数和补码数，80x86 使用不同的处理规则，所以有两类乘法指令：MUL 无符号数乘法和 IMUL 有符号数乘法。

1) MUL/IMUL 指令(unsigned/sIgned MULtiple)

格式：MUL/IMUL 源操作数。

功能：无/有符号数乘指令。

　　8 位(字节)操作：　AL×源操作数 ⇒AX；

　　16 位(字)操作：　AX×源操作数 ⇒DX:AX；

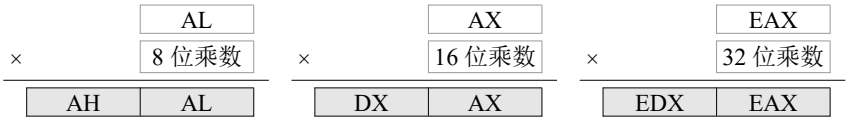
　　32 位(双字)操作：　EAX×源操作数⇒EDX:EAX。

操作数寻址方式：MUL/IMUL *reg*/*mem*。

两者的区别在于：MUL 的操作数内容看作无符号数，IMUL 操作数内容看作补码。

在乘法指令中，被乘数隐含在 AL(8 位运算)或 AX(16 位运算)或 EAX(32 位运算)中；乘数也就是源操作数，决定了乘法是 8 位运算，还是 16 位运算或者 32 位运算。两个 8 位数相

乘所得结果为 16 位，存放在 AX 中，如图 4.5(a)所示；两个 16 位数相乘所得结果是 32 位，存放在 DX,AX 中，如图 4.5(b)所示，其中，DX 存放高 16 位，AX 存放低 16 位；两个 32 位数相乘所得结果是 64 位，存放在 EDX,EAX 中，如图 4.5(c)所示，其中，EDX 存放高 32 位，EAX 存放低 32 位。



(a) 8 位乘法运算　(b) 16 位乘法运算　(c) 32 位乘法运算

图 4.5　乘法指令运算示意图

乘法指令不影响除 CF 和 OF 以外的标志位。

对于 MUL 来说，如果乘积的高半部分为 0，则 CF 和 OF 均为 0，否则 CF 和 OF 均为 1；对 IMUL 来说，如果乘积的高半部分是低半部分的符号扩展，则 CF 和 OF 均为 0，否则均为 1。通过测试这两个标志位，就能够知道乘积的高半部分是否有效数字。

例 4.26　分别将字节单元[1000h]和[2000h]中内容作为：① 8 位无符号数；② 8 位有符号数。求这两个数的乘积，所得 16 位数结果放到 2 字节单元[3000h]中去。

解：对于加法和减法，无符号数与有符号数用相同指令。但对于乘法，则要用不同的指令。无符号乘法用 MUL 指令，有符号数用 IMUL 指令。实现的指令如下：

```
① 作为 8 位无符号数的指令        ② 作为 8 位有符号数的指令
MOV AL, [1000h]                  MOV     AL, [1000h]
MUL Byte Ptr[2000h]              IMUL    Byte Ptr[200h]
MOV [3000h], AX                  MOV     [3000h], AX
```

设字节单元[1000h]中内容为 FEh、字节单元[2000h]中内容为 05h，则：指令①执行后，字单元[3000h]中的内容是 04F6h，CF＝OF＝1；指令②执行后，字单元[3000h]中的内容是 FFF6h，CF＝OF＝0。尽管两操作数内容相同，但由于编码约定不同而选择不同的乘法指令，因而所得结果也不一样。

4. 除法指令

对应无符号数和补码数，分别有 DIV 无符号数除法和 IDIV 有符号数除法。要求被除数的位数必须是除数的两倍。DIV/IDIV 指令(unsigned/sIgned DIVision)说明如下。

格式：DIV/IDIV 源操作数。

功能：无/有符号数除法指令。

8 位(字节)操作：　AX 除以源操作数，　商存入 AL，　余数存入 AH；

16 位(字)操作：　DX,AX 除以源操作数，　商存入 AX，　余数存入 DX；

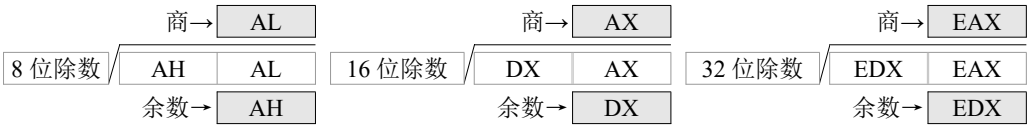
32 位(双字)操作：　EDX,EAX 除以源操作数，　商存入 EAX，　余数存入 EDX。

操作数的寻址方式为：DIV/IDIV *reg*/*mem*。

两者的区别在于：DIV 的操作数约定是无符号数，商和余数均为无符号数；IDIV 操作数约定是补码，商和余数均为有符号数，余数符号与被除数符号相同。

在除法中，被除数隐含在 AX(8 位运算)或 DX:AX(16 位运算)或 EDX:EAX(32 位运算)中，除数即源操作数，决定了除法是 8 位运算，还是 16 位运算或者 32 位运算。16 位数除以 8 位

数，商是 8 位，存放在 AL 中，余数是 8 位，存放在 AH 中，如图 4.6(a)所示；32 位数除以 16 位数，商是 16 位，存放在 AX 中，余数是 16 位，存放在 DX 中，如图 4.6(b)所示；64 位数除以 32 位数，商是 32 位，存放在 EAX 中，余数是 32 位，存放在 EDX 中，如图 4.6(c)所示。



(a) 8 位除法运算　(b) 16 位除法运算　(c) 32 位除法运算

图 4.6　除法指令运算示意图

一条除法指令可能导致两类错误：一类是除数为零，另一类是商溢出。当除法运算所得的商超过表示范围时，就产生商溢出。例如，2000h 除以 20h，所得商为 100h，超出 AL 所能表示的范围，导致除法溢出。当产生这两类除法错误时，处理器就会产生一次中断处理。

除法指令对所有标志位无定义。

例 4.27　分别按：① 无符号数；② 符号数，用 2 字节单元[1000h]中的数除以字节单元[2000h]中的数，所得商和余数分别存放到单元[3000h]和[4000h]中去。

解：无符号数除法用 DIV 指令计算，有符号数用 IDIV 指令计算。指令序列如下：

```
① 作为 8 位无符号数指令序列      ② 作为 8 位有符号数指令序列
MOV     AX, [1000h]              MOV     AX, [1000h]
DIV     Byte Ptr[2000h]          IDIV    Byte Ptr[2000h]
MOV     [3000h], AL              MOV     [3000h], AL
MOV     [4000h], AH              MOV     [4000h], AH
```

设字单元[1000h]中内容为 0105h，字节单元[2000h]中内容为 81h。指令序列①执行后，字节单元[3000h]，[4000h]的内容分别是 02h 和 03h，即有：105h＝02h×81h+03h。由于 16 位补码 0105h 表示的数是 261，8 位补码 81h 表示的数是−127，且有 261＝−127×(−2)+7，所以指令序列②执行后，字节单元[3000h]和[4000h]的内容分别是 FEh 和 07h。

4.2.4　逻辑指令

逻辑指令包括逻辑运算指令和移位指令。

1. 逻辑运算指令

逻辑运算是按位操作的，包括：AND, OR, NOT, XOR 和 TEST 指令。其中 AND, OR, XOR 和 TEST 都是双操作数指令，形式相似，操作数的寻址方式与算术运算指令相同，对标志位的影响也相同，即 CF=0，OF=0，AF 无定义，SF, ZF 和 PF 根据结果设置。

1) AND 指令

格式：AND 目的操作数，源操作数。

功能：逻辑与。执行按位“逻辑与”操作，目的操作数∧源操作数⇒目的操作数。

操作数的寻址方式：　AND *reg*, *reg*/*mem*/*imm*；

AND *mem*, *reg*/*mem*。

根据结果设置 SF, ZF 和 PF，CF=0，OF=0，AF 无定义。

2) OR 指令

格式：OR 目的操作数，源操作数。

功能：逻辑或。执行按位“逻辑或”操作，目的操作数∨源操作数⇒目的操作数。

操作数的寻址方式： OR *reg*, *reg*/*mem*/*imm*；

OR mem, reg/mem。

根据结果设置 SF, ZF 和 PF，CF=0，OF=0，AF 无定义。

3) NOT 指令

格式：NOT 目的操作数。

功能：逻辑非指令。执行按位取反操作，即：¬目的操作数⇒目的操作数。

操作数的寻址方式：NOT *reg*/*mem*。

不影响标志位。

4) XOR 指令(eXclusive OR)

格式：XOR 目的操作数，源操作数。

功能：逻辑异或。执行按位“逻辑异或”操作,目的操作数∀源操作数⇒目的操作数。

操作数的寻址方式： XOR *reg*, *reg*/*mem*/*imm*；

XOR *mem*, *reg*/*mem*。

根据结果设置 SF, ZF 和 PF，CF=0，OF=0，AF 无定义。

5) TEST 指令

格式：TEST 目的操作数，源操作数。

功能：测试指令。按位逻辑与进行测试，根据目的操作数与源操作数的逻辑与运算结果设置标志位。

操作数的寻址方式为：TEST *reg*, *reg*/*mem*/*imm*；

TEST *mem*, *reg*/*mem*。

根据结果设置 SF, ZF 和 PF，CF=0，OF=0，AF 无定义。

TEST 与 AND 都是两操作数按位“逻辑与”，但 TEST 的结果不保存。

例 4.29 指出下列逻辑指令执行时的状态变化。

```
MOV AX, 43E9h   ; 43E9h⇒AX
AND AX, AX      ; AX 无变化,但 CF=OF=0,SF=0,ZF=0,PF=0
AND AL, 6Eh     ; b0,b4,b7 清 0,AL:68h,CF=O,F=0,SF=0,ZF=0,PF=0
OR  AH, AH      ; AH 无变化,CF=OF=0,SF=0,ZF=0,PF=0
OR  AH,80h      ; b7 置 1,AH:C3h,CF=OF=0,SF=1,ZF=0,PF=1
XOR AX, 304h    ; b2,b8,b9 变反,AX:C06Ch,CF=OF=0,SF=1,ZF=0,PF=1
NOR AX          ; AX 各位取反,AX:3F93h,标志位无变化
```

例 4.30 80x86 标志寄存器中，DF, TF, IF 和 SF 分别对应位 8, 10, 9 和 7。若将 DF, TF 清 0，IF 置 1，SF 变反，其余保持不变，则可用如下的指令序列：

```
PUSHF               ; 标志寄存器低 16 位进栈
POP     AX          ; 标志寄存器低 16 位⇒AX
AND     AX, 0FAFFh  ; b8,b10 清 0,即 DF 和 TF 清 0
OR      AX, 0200h   ; b9 置 1,即 IF 置 1
XOR     AX, 0080h   ; b7 变反,即 SF 变反
PUSH    AX
POPF                ; 从堆栈中弹出 16 位数据⇒标志寄存器低 16 位
```

2. 移位指令

移位指令包括：逻辑移位指令、算术移位指令和循环移位指令。这些指令都是按指令规定的方式，对目的操作数执行向左或向右移动若干个二进制位数的操作。

双操作数移位指令格式是：

指令助记符 *reg*/*mem*, *imm*8/CL (对于 8086，*imm*8 只能取 1)

三操作数的双精度移位指令格式是：(80386 及后续 CPU 提供)

指令助记符 *reg*/*mem*, *reg*, *imm*8/CL

1) SHL 指令(SHift logical Left)

格式：SHL 目的操作数，移动位数。

功能：逻辑左移指令。目的操作数逻辑左移，最后移出的位进入 CF，最低位用 0 填充，如图 4.7(a)所示。

操作数的寻址方式为：SHL *reg*/*mem*, *imm*8/CL。

本指令影响 CF, OF, SF, ZF, PF，而 AF 不确定。其中 OF 在左移 1 位时有效，否则不确定。左移 1 位后，若符号位改变，则 OF=1，否则 OF=0。

2) SHR 指令(SHift logical Right)

格式：SHR 目的操作数，移动位数。

功能：逻辑右移指令。目的操作数逻辑右移，最后移出的位进入 CF，最高位用 0 填充，如图 4.7(b)所示。

操作数的寻址方式为：SHR *reg*/*mem*, *imm*8/CL。

本指令影响 CF, OF, SF, ZF, PF，而 AF 不确定。其中 OF 在右移 1 位时有效，否则不确定。右移 1 位后符号位改变，则 OF=1，否则 OF=0。

3) SAL 指令(Shift Arithmetic Left)

格式：SAL 目的操作数，移动位数。

功能：算术左移指令。SAL 与 SHL 是同一条指令，即一个操作码对应的两个助记符。

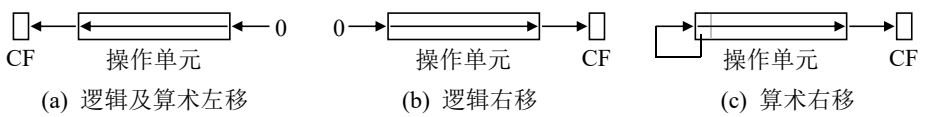
4) SAR 指令(Shift Arithmetic Right)。

汇编格式：SAR 目的操作数，移动位数。

功能：算术右移指令。目的操作数算术右移，最后移出的位进入 CF，高位用符号位填充，如图 4.7(c)所示。

操作数的寻址方式为：SAR *reg*/*mem*, *imm*8/CL。

本指令影响 CF, OF, SF, ZF, PF，而 AF 不确定。其中 OF 在右移 1 位时有效，否则不确定。右移 1 位后，OF=0。



(a) 逻辑及算术左移　(b) 逻辑右移　(c) 算术右移

图 4.7　逻辑及算术移位操作示意图

5) ROL 指令(ROtate Left)

格式：ROL 目的操作数，移动位数。

功能：循环左移。目的操作数循环左移，最后移出的位进 CF。如图 4.8(a)所示。

操作数的寻址方式：ROL *reg*/*mem*, *imm8*/CL

本指令影响 CF, OF, SF, ZF, PF，而 AF 不确定。其中 OF 在左移 1 位时有效，否则不确定。左移 1 位后，若符号位改变，OF=1，否则 OF=0。

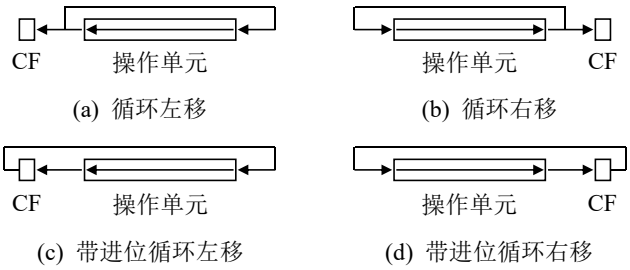


图 4.8　循环移位操作示意图

6) ROR 指令(ROtate Right)

格式：ROR 目的操作数，移动位数。

功能：循环右移。目的操作数循环右移，最后移出的位进 CF。如图 4.8(b)所示。

操作数的寻址方式为：ROR *reg*/*mem*, *imm8*/CL。

本指令影响 CF, OF, SF, ZF, PF，而 AF 不确定。其中 OF 在右移 1 位时有效，否则不确定。右移 1 位后，若符号位改变，则 OF=1，否则 OF=0。

7) RCL 指令(ROtate Left through Carry)

格式：RCL 目的操作数，移动位数。

功能：带进位循环左移指令。目的操作数和 CF 一起进行循环左移。如图 4.8(c)所示。

操作数的寻址方式为：RCL *reg*/*mem*, *imm*8/CL。

本指令影响 CF, OF, SF, ZF, PF，而 AF 不确定。其中 OF 在左移 1 位时有效，否则不确定。左移 1 位后，若符号位改变，则 OF=1，否则 OF=0。

8) RCR 指令(ROtate Right through Carry)

格式：RCR 目的操作数，移动位数。

功能：带进位循环右移指令。目的操作数和 CF 一起进行循环右移。如图 4.8(d)所示。

操作数的寻址方式为：RCR *reg*/*mem*, *imm*8/CL。

本指令影响 CF, OF, SF, ZF, PF，而 AF 不确定。其中 OF 在右移 1 位时有效，否则不确定。右移 1 位后，若符号位改变，则 OF=1，否则 OF=0。

4.3　控制转移类指令

在 80x86 系统中，处理器的 CS:(E)IP(32 位地址模式用 EIP，16 位地址模式用 IP)总是指向下一条要执行的指令在内存中的地址，因而，控制转移指令实际上通过改变 CS:(E)IP 来达到控制程序的执行流程。这类指令包括：无条件转移指令、条件转移指令、循环指令、子程序调用和返回指令，以及中断调用和中断返回指令。

如果控制转移指令仅能改变(E)IP，则是近转移或段内转移；如果既可改变(E)IP，又可改变 CS，则是远转移或段间转移。

4.3.1　无条件转移指令

80x86 指令系统中，JMP(JuMP)类指令是无条件转移指令。

格式：JMP 目标地址。

功能：无条件转移指令。无条件转移到目标地址，执行从该地址开始的指令序列。

目标地址有两种寻址方式，一种是直接寻址方式：目标地址作为指令的一部分，直接存放在操作码后面，如图 4.14 中的虚线方框标注的 JMP 指令；另一种是间接寻址方式：目标地址存放在寄存器或内存单元中，如图 4.14 中的实线方框标注的 JMP 指令。

图 4.14 仅仅为了说明 JMP 指令格式，在实际程序中不会出现如此“糟糕”的转移代码。示例中有两个代码段：由 CS 指示的段(左边部分，正在执行的代码段)和由段地址 0011 指示的段(右边部分)。注意，这段代码并不是汇编语言源代码，而是机器指令的“反汇编”(仅用指令助记符表示的机器指令，没有符号地址)。下面在此示例代码基础上，说明 JMP 指令格式及其汇编指令的用法。

```
地址      机器指令          汇编指令
CS:0100  2B C0            SUB AX, AX
CS:0102  05 34 12         ADD AX, 1234
CS:0105  78 03            JS  010A
CS:0107  E9 F8 FF         JMP Near Ptr 0102
CS:010A  3D 00 A0         CMP AX, A000
CS:010D  77 02            JA  0111
CS:010F  EB 10            JMP Short 0121
CS:0111  7B 09            JPO 011C
CS:0113  2E               CS:
CS:0114  FF 2E 18 01      JMP DWord Ptr[0118]
CS:0118  87 01 11 00      DB  87,01,11,00
CS:011C  EA 6A 01 11 00   JMP 001A:016A
CS:0121  BE 00 01         MOV SI, 0100
CS:0124  FF E6            JMP SI
CS:0126  CC               INT 3
```

```
地址      机器指令          汇编指令
 0011:0168  C3             RET
 0011:0169  90             NOP
→0011:016A  8B 1E 13 99    MOV BX,[9913]
 0011:016E  83 FB 00       CMP BX,0
 0011:0171  7E 13          JLE 0186
 0011:0173  8B 0E E1 99    MOV CX,[99E1]
 0011:0177  8B 16 DF 99    MOV DX,[99DF]
 0011:017B  8B C1          MOV AX,CX
 0011:017D  0B C2          OR  AX,DX
 0011:017F  74 05          JZ  0186
 0011:0181  B8 00 42       MOV AX,4200
 0011:0184  CD 21          INT 21
 0011:0186  CC             INT 3
 0011:0187  B4 40          MOV AH,40
 0011:0189  CC             INT 3
```

图 4.14　JMP 指令格式的代码示例

1. 无条件直接转移指令

转移的目标地址直接用标号标识出来，其格式为：

```
JMP 标号
```

此处标号直接标识指令的存放位置。

1) 段内直接近转移

当标号和转移指令同属于一个代码段，就是直接近转移，其格式为：

```
JMP Near Ptr 标号
```

在实际程序中，可直接写成“JMP 标号”形式，这样，只要标号和转移指令是在相同的代码段，汇编程序就会将它翻译为直接近转移指令。

在直接近转移指令对应的机器指令中，操作码后面是目标的偏移地址的相对位移量，即目标地址和 JMP 指令的下一条指令的地址的相对位移。目标地址在当前位置之前用负数表示，在当前位置之后用正数表示，所以操作码后面的位移量用补码表示。

在 16 位地址模式下，操作码后面是一个 8 位或者 16 位补码，对应的转移范围是：−128～+127 或者−32768～+32767(−32K～+32K−1)个字节；在 32 位地址模式下，操作码后面是一个 8 位或者 32 位补码，对应的转移范围是：−128～+127 或者−2147483648～+2147483647(−2G～

+2G−1)个字节。

例如，图 4.14 中 CS:0107 处的指令“E9 F8 FF”，E9 是操作码，F8 FF 是位移量。补码 FFF8 表示的数是−8，由此算出目标地址为 010Ah−8＝0102h。所以机器指令 EB F8 FF 对应汇编指令为 JMP 0102。

在示例中直接使用地址编号表示转向的目标地址，但是在源程序中则应该用标号来表示，例如，CS:102～10A 的指令，在汇编源程序中可编写成如下形式：

```
Loc1:   ADD     AX, 1234h
        JS      Loc2
        JMP     Near Ptr Loc1
Loc2:   CMP     AX, 0A000h
```

特别地，当位移量用 8 位补码表示，这样形式的转移指令称为直接短转移，它的转移目标范围是：−128～+127 个字节。如图 4.14 中 CS:010F 处的指令“EB 10”，EB 是操作码，10 是位移量，由此可以计算出转移的目标地址：0111h+10h＝0121h。所以机器指令 EB 10 对应的汇编指令是 JMP 0121。

短转移指令的格式为：JMP Short 标号。

2) 段间直接远转移

当标号和转移指令不在同一个代码段，就是直接远转移，其格式为：

```
JMP Far Ptr 标号
```

在实际程序中，可直接写成“JMP 标号”的形式，只要标号和转移指令是在不同的代码段，汇编程序就会将它翻译成直接远转移指令。

在直接远转移指令所对应的机器指令中，操作码后直接存放目标的偏移地址和段地址。如图 4.14 中 CS:011C 处的指令“EA 6A 01 11 00”，EA 是操作码，6A 01 和 11 00 分别是目标的偏移地址 016A 和 0011。该机器指令对应的汇编格式指令为 JMP 0011:016A。(在汇编源程序中应该用标号代替目标地址)

2. 无条件间接转移指令

目标地址不直接用标号标识，而是间接地放在寄存器或内存单元中，其格式为：

```
JMP reg/mem
```

1) 段内间接近转移

寄存器或者内存单元存放的是转移目标的偏移地址。其格式为：

```
JMP reg16/mem16 (16 位地址模式)
JMP reg32/mem32 (32 位地址模式)
```

如图 4.14 中 CS:0124 处的指令“JMP SI”，此处 SI 中存放的是目标的偏移地址。

2) 段间间接远转移

转移目标的偏移地址和段地址都存放在内存单元中。其格式为：

```
JMP mem32 (16 位地址模式,低字是偏移地址,高字是段地址)
JMP mem48 (32 位地址模式,低 4 字节是偏移地址,高字是段选择器)
```

如图 4.14 中 CS:0113 处的指令“CS:JMP DWord ptr [0118]”，此处双字单元 CS:[0118] 中前两字节存放的是偏移地址，后两字节存放的是段地址。

再如，JMP FWord Ptr 100h[EBX]，6 字节单元[EBX+100h]中低 4 字节是转移目标的偏移地址，高字则是转移目标的段选择器。

说明：所有的直接近转移(包括稍后介绍的条件转移、直接子程序调用等)采用的是相对地址转移，所以直接近转移指令符合程序的重定位要求；所有的间接转移及直接远转移所用的是绝对地址转移，所以使用了这些指令的程序是不能够进行重定位的。

在 16 位地址模式下，段内直接转移的目标仅有−32K～+32K−1，所以经常使用远转移指令；在 32 位地址模式下，段内直接转移的目标范围较大，可确定−2G～+2G−1，所以应用程序极少使用远转移指令。

4.3.2　条件转移指令

80x86 条件转移类指令主要是根据一个或多个运算结果标志位的状态来控制程序转移的一类指令，影响的标志位有：CF, PF, AF, ZF, SF, OF 等，因而条件转移指令有 16 条之多。

为简化和方便说明起见，常用 J*cc* 来代表这类指令的助记符。其格式如下：

```
Jcc  标号
```

其中，J*cc* 代表所有条件指令助记符；标号用以标识要转移的目标位置，必须与转移指令同处于一个代码段(所以是直接近转移)。

所有的条件转移指令都是段内直接近转移，是相对地址转移。在机器指令格式中，目标地址的相对位移量直接放在指令操作码之后，如图 4.14 中的指令 JS 010A，其转移目标地址的相对位移量是 03，直接放在操作码 78 之后。

在 8086 和 80286 中，条件转移都是使用短转移格式，目标地址只能在相对的−128～+127 个字节的范围内。在 80386 及后续处理器中，除了短转移格式外，还提供近转移格式指令，其目标地址的范围是：16 位地址模式，−32KB～+32KB−1；32 位地址模式，−2GB～+2GB−1。

根据测试条件的不同，可将条件转移类指令分为以下四组。

1. 根据单个标志位的值来决定是否转移的指令

测试的标志位有：CF, PF, ZF, SF 和 OF 共 5 个，每个标志位可以取 0 和 1，因此，这组指令有 10 条，每条指令对应每个标志位的一种取值。这组指令见表 4.1。

表 4.1　简单条件转移指令表

Jcc	检 测 条 件	功 能 描 述
JE/JZ	ZF=1	若相等/为 0，则转移(Jump if Zero, or Equal)
JNE/JNZ	ZF=0	若不等/不为 0，则转移(Jump if Not Zero, or Not Equal)
JS	SF=1	若为负数，则转移(Jump if Sign)
JNS	SF=0	若为正数，则转移(Jump if Not Sign)
JC	CF=1	若有进位，则转移(Jump if Carry)
JNC	CF=0	若无进位，则转移(Jump if Not Carry)
JO	OF=1	若有溢出，则转移(Jump if Overflow)
JNO	OF=0	若无溢出，则转移(Jump if Not Overflow)
JP/JPE	PF=1	若有偶数个 1，则转移(Jump if Parity, or Parity Even)

Jcc	检 测 条 件	功 能 描 述
JNP/JPO	PF=0	若有奇数个 1，则转移(Jump if Not Parity, or Parity Odd)

2. 比较两个无符号数，并根据比较结果转移的指令

为了与有符号数比较大小的“大于”或“小于”相区别，无符号数中的大于、小于关系分别用“高于”、“低于”来表述。两个无符号数比较时，应根据 CF 来判断它们的大小。具体来说，两数比较是通过减法操作来完成的，两个无符号数相减，若不够减，则最高位有借位，CF=1，否则，CF=0。所以，当 CF=1 时，说明被减数低于减数；当 CF=0 且 ZF=0 时，说明被减数高于减数；当 CF=0 且 ZF=1 时，说明被减数等于减数，见表 4.2。

需要说明的是，JAE/JNB 与 JNC 虽然助记符不同，但对应同一条机器指令。同样，JB/JNAE 与 JC 也是同一条指令。

表 4.2　无符号数比较条件转移指令表

Jcc	检 测 条 件	功 能 描 述
JA/JNBE	CF∨ZF=0	若高于/不低于等于，转移(Jump if Above, or Not Below or Equal)
JAE/JNB	CF=0	若高于等于/不低于，转移(Jump if Above or Equal, or Not Below)
JB/JNAE	CF=1	若低于/不高于等于，转移(Jump if Blow, or Not Above or Equal)
JBE/JNA	CF∨ZF=1	若低于等于/不高于，转移(Jump if Blow or Equal, or Not Above)

3. 比较两个有符号数，并根据比较结果转移的指令

两个有符号数比较时，应根据 SF 来判断它们的大小。两数比较通过减法操作来完成，当两个有符号数相减时，在结果没有溢出的情况下，即 OF=0，若 SF=1，则说明被减数小于减数；若 SF=0，则说明被减数不小于减数。但是如果运算结果产生溢出，即 OF=1，此时 SF 显示的正负性正好与应该得的正确结果值的正负性相反，也就是说，若 OF=1 时，SF=0 表示被减数小于减数，SF=1，表示被减数大于减数。

因此，当 OF=0 且 SF=1，或者 OF=1 且 SF=0 时，即 SF∀OF=1，表示被减数一定小于减数。当 OF=0 且 SF=0，同时 ZF=0，或者 OF=1 且 SF=1，ZF=0 时，表示被减数一定大于减数，即测试大于的条件为(SF∀OF)∨ZF=0。转移指令见表 4.3。

表 4.3　有符号数比较的条件转移指令

Jcc	检 测 条 件	功 能 描 述
JG/JNLE	(SF∀OF)∨ZF=0	若大于/不小于等于，则转移(Jump if Greater, or Not Less or Equal)
JGE/JNL	SF∀OF=0	若大于等于/不小于，则转移(Jump if Greater or Equal, or Not Less)
JL/JNGE	SF∀OF=1	若小于/不大于等于，则转移(Jump if Less, or Not Greater or Equal)
JLE/JNG	(SF∀OF)∨ZF=1	若小于等于/不大于，则转移(Jump if Less or Equal, or Not Greater)

条件判断和转移操作通常是由比较指令和条件转移指令来实现的。下面举例说明。

例 4.36　数据段内存区[1000h]字单元和[2000h]字单元分别存放两个有符号数，编写指令序列，将两个数中较大者存放到[3000h]字单元中。

```
        MOV     AX, [1000h]     ; [1000h]⇒AX
```

```
        CMP     AX, [2000h]     ; 两个有符号数比较
        JGE     Loc10           ; 大于等于,即 AX 中已经是大数,转移
        MOV     AX, [2000h]     ; [2000h]⇒AX
Loc10:  MOV     [3000h],AX      ; AX⇒[3000h]
```

例 4.37　将数据段内存区首地址为 1000h、长度为 500h 字节的内存块中的内容复制到首地址为 2000h 的内存块中，并且将所有小写字母转换成大写。

解：26 个小写英文字母的 ASCII 码是 61h～7Ah，可以据此判断一个 ASCII 码是不是小写字母。若是小写字母，则 ASCII 码减 32 后即为大写字母的 ASCII 码。指令序列如下：

```
        CLD                 ; 置 DF 为 0,即设置为正向传送方式
        MOV     SI, 1000h   ; DS:SI:源内存块的首地址(DS 不须设置)
        MOV     AX, DS      ; DS⇒AX
        MOV     ES, AX      ; AX⇒ES
        MOV     DI, 2000h   ; ES:DI:目的内存块的首地址
        MOV     CX, 500h    ; 内存块中的字符个数
Loc10:  LODSB               ; DS:[SI]⇒AL,SI+1⇒SI,取 1 个 ASCII 码到 AL
        CMP     AL, 61h     ; 与'a'比较
        JB      Loc20       ; 低于'a'则转移
        CMP     AL, 'z'     ; 'z'就是 ASCII 码 7Ah,与'z'比较
        JA      Loc20       ; 高于'z'则转移
        SUB     AL, 20h     ; 转换成大写字母
Loc20:  STOSB               ; AL⇒ES:[DI],DI+1⇒DI,即保存 1 个 ASCII 码
        DEC     CX          ; CX-1⇒CX
        JNZ     Loc10       ; 运算结果不为 0,转移
```

4.3.3　循环指令

80x86 专门针对(E)CX 提供了设计一组循环指令。

1) LOOP 指令(LOOP)

格式：LOOP　地址标号。

功能：先执行(E)CX−1⇒(E)CX，之后若(E)CX≠0，则转移。

2) LOOPZ/LOOPE 指令(LOOP while zero, or Equal)

格式：LOOPZ/LOOPE 地址标号。

功能：先执行(E)CX−1⇒(E)CX，之后若 ZF＝1 且(E)CX≠0，则转移。

3) LOOPNZ/LOOPNE 指令(LOOP while NonZero, or Not Equal)

格式：LOOPNZ/LOOPNE 地址标号。

功能：先执行(E)CX−1⇒(E)CX，之后若 ZF＝0 且(E)CX≠0，则转移。

例 4.38　用 LOOP 指令改写例 4.37 中的指令序列。

```
        CLD                 ; 置 DF 为 0,即设置为正向传送方式
        MOV     SI, 1000h   ; DS:SI:源内存块的首地址(DS 不须设置)
        MOV     AX, DS      ; DS⇒AX
        MOV     ES, AX      ; AX⇒ES
        MOV     DI, 2000h   ; ES:DI:目的内存块的首地址
        MOV     CX, 500h    ; 内存块中的字符个数
Loc10: LODSB                ; DS:[SI]⇒AL,SI+1⇒SI,取 1 个 ASCII 码到 AL
        CMP     AL, 61h     ; 与'a'比较
        JB      Loc20       ; 低于'a'则转移
```

```
        CMP     AL, 'z'          ; 'z'就是 ASCII 码 7Ah,与'z'比较
        JA      Loc20            ; 高于'z'则转移
        SUB     AL, 20h          ; 转换成大写字母
Loc20:  STOSB                    ; AL⇒ES:[DI],DI+1⇒DI,即保存 1 个 ASCII 码
        LOOP    Loc10            ; CX-1⇒CX, 若 CX≠转移
```

例 4.39 下列指令序列实现的寻找存储块中第一个空格。

```
        CLD
        MOV     AL, 20h
        MOV     DI, 100h
        MOV     CX, 16h
Loc10:  INC DI
        CMP     AL [DI-1]
        LOOPNE  Loc10
```

4.3.5 子程序调用指令与子程序返回指令

所有机器的指令系统都有子程序调用指令和子程序返回指令。当主程序转向子程序时，使用调用指令，而在子程序执行结束时，安排一条返回指令，使子程序返回到主程序。为保证正确的返回，每次调用子程序时，自动将下一条指令地址保存到堆栈中，返回时根据堆栈中先前保存的地址，转移到主程序继续执行。所以，子程序调用与返回指令是配套使用的。

在 80x86 中可用调用指令 CALL 在主程序中调用子程序，可用返回指令 RET 在子程序中返回主程序，继续往下执行。CALL 指令有段内调用(近调用)和段间调用(远调用)两种格式，与之对应的 RET 指令也有段内返回(近返回)与段间返回(远返回)两种格式。

为了便于说明，我们特地编写如图 4.15 所示的一段指令。在这段指令中有两个代码段，分别列示在两边。下面以图 4.15 示例代码来介绍 CALL 指令与 RET 指令。

地址	机器指令	汇编指令
1012:0100	B9 02 10	MOV CX,1002
1012:0103	E8 1B 00	CALL 0121
1012:0106	9A 60 01 11 01	CALL 0111:0160
1012:010B	BA 21 01	MOV DX,0121
1012:010E	FF D2	CALL DX
1012:0110	C7 06 00 10 74 01	MOV Word Ptr[1000],0174
1012:0116	C7 06 02 10 11 01	MOV Word Ptr[1002],0111
1012:011C	FF 1E 00 10	CALL DWord Ptr[1000]
1012:0120	CC	INT 3
1012:0121	89 C8	MOV AX,CX
1012:0123	01 C0	ADD AX,AX
1012:0125	C3	RETN

地址	机器指令	汇编指令
0111:0160	51	PUSH CX
0111:0161	E8 15 00	CALL 0179
0111:0164	83 C4 02	ADD SP,02
0111:0167	CB	RETF
0111:0168	55	PUSH BP
0111:0169	89 E5	MOV BP,SP
0111:016B	8B 46 04	MOV AX,[BP+04]
0111:016E	01 C0	ADD AX,AX
0111:0170	5D	POP BP
0111:0171	C2 02 00	RETN 0002
0111:0174	51	PUSH CX
0111:0175	E8 F0 FF	CALL 0168
0111:0178	CB	RETF
0111:0179	55	PUSH BP
0111:017A	89 E5	MOV BP,SP
0111:017C	8B 46 04	MOV AX,[BP+04]
0111:017F	01 C0	ADD AX,AX
0111:0181	5D	POP BP
0111:0182	C3	RETN

图 4.15 CALL 与 RET 指令示例

1. CALL 指令(CALL a procedure)

格式：CALL　目标地址。

功能：子程序调用指令。首先把该指令之后的地址进栈，然后转移到目标地址，去执行从该地址开始的指令。

与 JMP 指令类似，CALL 指令中的目标地址有直接寻址和间接寻址两种方式。

从功能方面看，CALL 指令与 JMP 指令很相似，都是无条件地转移到目的地址处，并执行该处指令，但是，它们是不同类型的指令：通过 CALL 指令实现的转移，可以用 RET 指令返回，而 JMP 却不能。

1) 段内直接近调用

目标地址直接以子程序名、标号等给出，且目标与 CALL 指令同处一个段。

格式：CALL Near Ptr 子程序名。

功能：首先把该指令之后的地址(返回地址)的偏移地址(16/32 位)进栈，再转向子程序入口地址，执行子程序的第一条指令。转移后 CS 内容没有变化。

注意，与此格式的调用指令相配套的是段内近返回指令。

在段内直接近调用指令所对应的机器指令中，目标地址以相对位移量的形式直接存放在操作码之后。在 16 位地址模式下，这样格式的指令是一条 3 字节指令：1 个字节的操作码，以及其后的 2 个字节的 16 位的位移量，如图 4.15 中 1012:0103 处的指令“E8 1B 00”，E8 是操作码，1B 00 是位移量，由此计算出子程序开始地址(目标地址)是：0106h+001Bh＝0121h；在 32 位地址模式下，这样格式的指令是一条 5 字节指令：1 个字节的操作码，以及其后的 4 个字节的 32 位的位移量。

2) 段间直接远调用

目标地址直接以子程序名、标号等给出，且目标与 CALL 指令不在同一个段。

格式：CALL Far Ptr 子程序名。

功能：首先把该指令之后的地址(返回地址)，按段、偏移(16/32 位)的次序进栈，再转向子程序入口地址，执行子程序的第一条指令。转移后，CS 内容一般会发生变化。

注意，与此格式的调用指令相配套的是段间远返回指令。

在段间直接远调用所对应的机器指令中，转移的目标地址以段、偏移的次序直接存放在操作码之后。在 16 位地址模式下，这样格式的指令是一条 5 字节指令：1 个字节的操作码，以及其后的 2 个字节偏移地址和 2 字节的段地址，例如，图 4.15 中 1012:0106 处的指令“9A 60 01 11 01”，其中 9A 是操作码，60 01 是偏移地址 0160h，11 01 是段地址 0111；在 32 位地址模式下，这样格式的指令是一条 7 字节指令：1 个字节的操作码，以及其后的 4 个字节的 32 位的偏移地址和 2 字节的段选择器。

在汇编源程序中，很少直接用具体地址来指明子程序的入口，而代之以子程序名、标号等。而且也可直接用“CALL 子程序名”的形式来调用子程序，汇编程序在处理这条指令时，将根据子程序的定义情况确定是生成 Near Ptr 格式的调用指令，还是生成 Far Ptr 格式的调用指令。关于子程序定义等内容将在以后章节中详细介绍。

3) 段内间接调用

目标的偏移地址不直接给出，而是存放在寄存器或内存单元中。

格式：CALL *reg*/*mem*。

功能：首先把该指令之后的地址(返回地址)的偏移地址(16/32 位)进栈，再转向由 *reg*/*mem* 所确定的子程序入口，执行子程序的第一条指令。转移后，CS 内容没有变化。

注意，与此格式的调用指令相配套的是段内近返回指令。

例如，图 4.15 中 1012:010E 处的指令“CALL DX”，其转向的偏移地址存放在 DX 中。

4) 段间间接远调用

目标的偏移地址和段地址不直接给出，而是存放在内存单元中。

格式：CALL *mem*　(16 位地址模式是 *mem*32，32 位地址模式是 *mem*48)。

功能：首先把该指令之后的地址(返回地址)，按段、偏移(16/32 位)的次序进栈，然后转向由 *mem* 所确定的子程序入口地址，执行子程序的第一条指令。

注意，与此格式的调用指令相配套的是段间远返回指令。

例如，图 4.15 中 1012:011C 处的指令“CALL DWord Ptr[1000]”，其转向的子程序入口的偏移地址存放在[1000h]单元中，段地址存放在[1002h]单元中。

2. RET 指令(RETurn from procedure)

格式：RET/RET [*imm*16]。

功能：子程序返回指令。首先从堆栈栈顶弹出返回的目标地址，然后转移到该地址处执行。若其后有 *imm*16，则还要执行(E)SP+*imm*16⇒(E)SP。

从功能方面看，RET 也属于转移类指令，只不过它须与 CALL 指令配套使用，实现从子程序中返回，继续主程序的执行。

1) 段内近返回

格式：RETN/RETN *imm*16。

功能：子程序执行结束，返回主程序继续执行。即：首先从堆栈栈顶弹出返回的偏移地址(16/32 位)，再转移到该地址处执行。转移后的 CS 内容没有变化。当其后带有 *imm*16，则(E)SP 还会向高地址端调整 *imm*16 个字节单位，即(E)SP+*imm*16⇒(E)SP。

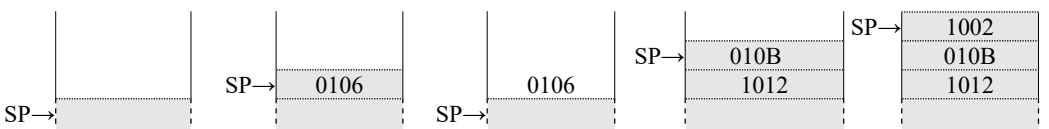
2) 段间远返回

格式：RETF/RETF imm16。

功能：子程序执行结束，返回主程序继续执行。即：首先从堆栈栈顶，以偏移(16/32 位)、段的次序弹出返回的目标地址，再转移到该目标地址处执行。当其后带有 *imm*16，则(E)SP 还会向高地址端调整 *imm*16 个字节单位，即(E)SP+*imm*16⇒(E)SP。

在汇编源程序中，常用“RET”或“RET *imm*16”形式，汇编程序在汇编这条指令时，根据子程序的定义情况来确定生成近返回指令(RETN)，还是远返回指令(RETF)。

为进一步说明 CALL 和 RET 的配套使用，从 1012:0100 开始执行图 4.15 中的指令序列，到 1012:0120 处终止。先后执行的指令有：(以指令所在的地址表示)1012:0100, **0103**, 0121, 0123, **0125**, **0106**, 0111:0160, **0161**, 0179, 017A, 017C, 017F, 0181, **0182**, 0164, **0167**, 1012:010B, **010E**, 0121, 0123, **0125**, 0110, 0116, **011C**, 0111:0174, **0175**, 0168, 0169, 016B, 016E, 0170, **0171**, **0178**, 1012:0120。执行期间堆栈变化如图 4.16 所示。



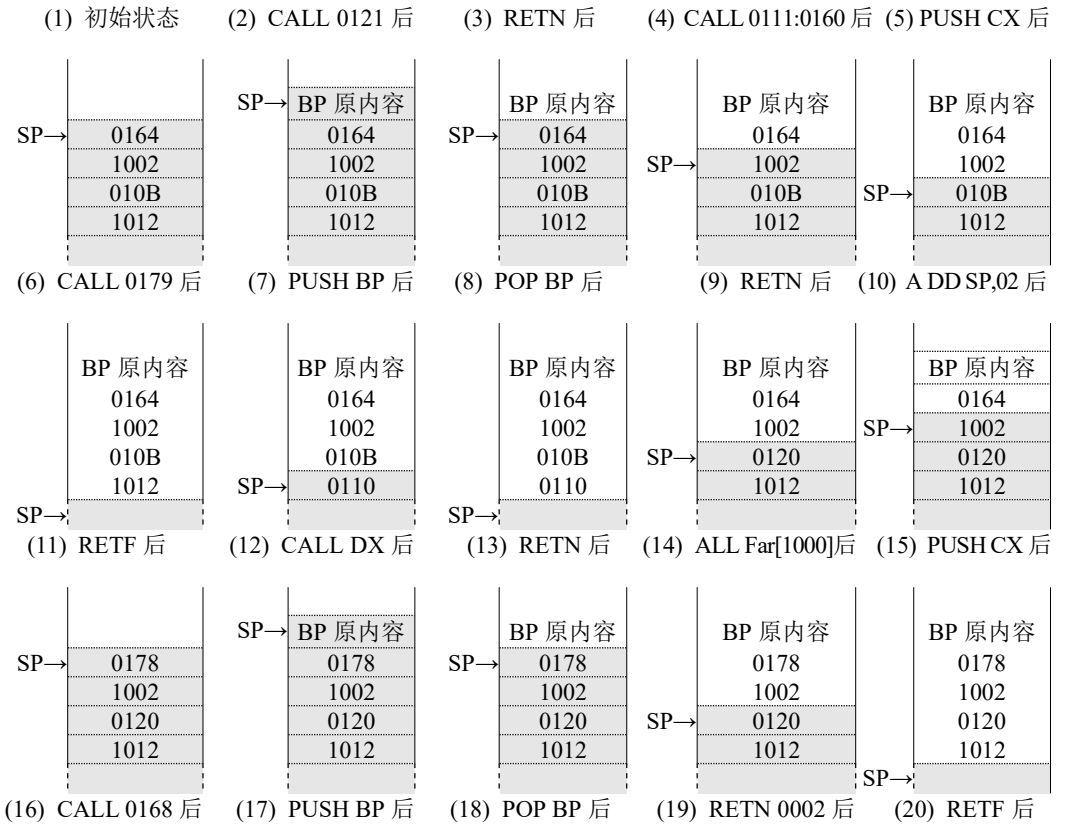


图 4.16　执行图 4.15 中指令序列的堆栈状态变化示意图

4.3.6　中断调用指令与中断返回指令

中断调用和中断返回类似于子程序间接远调用和返回，只不过中断服务子程序的入口地址保存在专门的内存区域，此外，在转移到中断服务子程序前，还需要把标志寄存器的内容保存入栈，以便当从中断服务子程序返回时，再将标志寄存器恢复到调用前的状态。

80x86 的中断分为外中断和内中断。两种类型中断的处理机制一样，不同的是外中断的中断源来自 CPU 外部，如 I/O 设备的中断请求，而内中断的中断源来自 CPU 内部，如执行 INT 指令，执行除法指令产生溢出。有关外中断处理的问题将在以后章节专门介绍，这里只介绍和中断相关的中断调用指令和中断返回指令。

由于在保护模式下，80x86 的中断处理机制比较复杂，为了简化起见，以下仅在实地址模式下介绍中断调用指令与中断返回指令。

在 80x86 系统中，中断服务子程序的入口地址称为**中断向量**。当 80x86 在实地址模式下工作时，内存中的最低 1KB 区域专门用来保存中断向量，称为**中断向量表**。每 4 字节保存一个中断向量：16 位偏移地址和 16 位段地址，并按顺序编号为：00h, 01h, …, FFh(0, 1, …, 255)，这样中断向量表保存有 256 个中断服务子程序的入口地址，相应的编号称为中断类型号。其对应关系是：中断类型号为 n，则该类型号中断服务子程序入口地址保存在地址为 0000h:$4\times n$ 的 4 字节单元中，其中低 2 字节存放的是偏移地址，高 2 字节存放的是段地址。80x86 实地址模式下的中断向量表如图 4.17 所示。

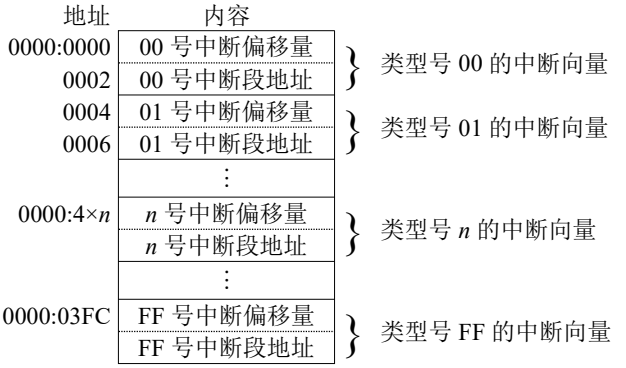


图 4.17　80x86 实地址模式下的中断向量表

从汇编指令的形式上看，与中断相关的指令主要有 INT，INTO 和 IRET。

1) INT 指令(INTerrupt)

格式：INT *imm*8。

功能：中断调用指令。产生一次类型号为 *imm*8 的中断：首先 FLAGS 进栈，再将 IF 和 TF 清 0；然后该指令之后的地址(返回地址)，按段、偏移的次序进栈；最后转向类型号为 *imm*8 的中断向量，即双字单元 0000h:[4×*imm*8]所确定的中断服务子程序入口地址处执行。

由于一条 INT 指令相当于产生一次中断，其处理过程与外部中断处理过程一样，所以，INT 指令又称为软中断指令。

2) INTO 指令(INTerrupt Overflow)

格式：INTO。

功能：若 OF＝1，产生一次类型号为 4 的中断，相当于 INT 4；否则顺序执行。

3) IRET 指令(RETurn from Interrup)

格式：IRET。

功能：中断处理结束，返回中断发生处继续执行。即：首先从堆栈中以“偏移地址、段地址、16 位标志”这样的次序弹出转向的目标地址和 FLAGS，再转移到该目标地址去执行。

4.4　其他类指令

4.4.1　标志位处理指令

80x86 提供有 7 条无操作数指令，专门用于设置或清除标志位。

CLC 指令(CLear Carry)	进位标志位清 0，即：0⇒CF；
CMC 指令(CoMplement Carry)	进位标志位取反，即：¬CF⇒CF；
STC 指令(SeT Carry)	进位标志位置 1，即：1⇒CF；
CLD 指令(CLear Direction)	方向标志位清 0，即：0⇒DF；
STD 指令(SeT Direction)	方向标志位置 1，即：1⇒DF；
CLI 指令(CLear Interrupt)	中断标志位清 0，即：0⇒IF；
STI 指令(SeT Interrupt)	中断标志位置 1，即：1⇒IF。

4.4.2　其他指令

1) NOP 指令(No OPeration)

格式：NOP。

功能：空操作指令。不执行任何操作。

说明：该指令的机器码(90h)占 1 个字节的内存单元，在调试程序时用它占有一定的存储单元，以便在正式运行时用其他指令取代。

2) HLT 指令(HaLT)

格式：HLT。

功能：停机指令。使 CPU 处于“什么也不干”的暂停状态，等待 I/O 中断发生。

说明：退出暂停状态有以下三种方法：中断、复位或 DMA 操作。实际使用时，该条指令往往出现在程序等待硬中断的地方，一旦中断返回，就可使 CPU 脱离暂停状态，继续 HLT 指令的下一条指令，实现了软件与外部硬件同步的目的。

3) WAIT 指令(WAIT white test pit not asserted)

格式：WAIT。

功能：等待指令。不断测试 WAIT 引脚。

说明：若测试到 WAIT 引脚状态为 0，则 CPU 处于暂停状态；若一旦测试到引脚状态为 1，则 CPU 脱离暂停状态，继续往下执行。

4) LOCK 指令(LOCK bus)

格式：LOCK XXXX 指令。

功能：总线封锁指令。使 LOCK 引脚输出低电平信号。

说明：实际使用中，CPU 的引脚与总线控制器 8289 的引脚相连。执行 LOCK 指令后，CPU 通过引脚送出一个低电平信号，总线控制器封锁总线，使其他处理器得不到总线控制权。这种状态一直延续到指令之后的指令执行完为止。

LOCK 总线封锁指令也叫前缀指令，可放在任何一条指令的前面。

5) CPUID 指令

格式：CPUID

功能：返回 CPU 的 ID 信息，放入 EAX, EBX, ECX, EDX 中。

6) RDTSC 指令

格式：RDTSC

功能：将自启动以来 CPU 运行的时钟周期数放入 EDX:EAX。

7) ESC 指令(ESCape)

格式：ESC *mem*。

功能：换码指令。指定内存单元中内容为由协处理器执行的指令。自 80486 起，这条指令码已成为未定义指令。