第 2 章　8086 的指令系统

计算机是通过指令序列来解决问题的，每种计算机都有支持的指令集合。计算机的指令系统就是指该计算机能够执行的全部指令的集合。Intel 8086 指令系统可分为 6 类：数据传送类指令，算术运算类指令，位操作类指令，控制转移类指令，串操作类指令，处理机控制类指令。

本章的重点是理解 8086 常用指令的功能，这是进行汇编语言程序设计的基础。为了更好地掌握每条指令，可以利用调试程序 DEBUG 作为实践环境进行指令汇编和单步执行，观察各种指令执行的实际效果。

下面将分类讲解每条指令。在学习每条指令时，请注意如下几方面。

- 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。
- 指令支持的寻址方式——该指令中的操作数可以采用何种寻址方式。1.6 节介绍了大多数指令支持的各种寻址方式，并给出本书采用的符号。
- 指令对标志的影响——该指令执行后是否对各标志位有影响，以及如何影响。
- 其他方面——该指令其他需要注意的地方，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。

2.1　数据传送类指令

数据传送是计算机中最基本、最重要的一种操作。传送指令也是最常使用的一类指令。数据传送指令的功能是把数据从一个位置传送到另一个位置。8086 有 14 种数据传送指令，实现寄存器和寄存器之间、主存和寄存器之间、AL/AX 与外设端口之间（见 5.4.1 节）的字与字节的多种传送操作。

数据传送类指令除标志寄存器传送指令外，均不影响标志位。指令介绍中不再说明。

2.1.1　通用数据传送指令

通用数据传送指令包括 MOV、XCHG 和 XLAT 指令，提供方便灵活的通用传送操作。

1．传送指令 MOV

传送指令 MOV 的格式如下：

```
mov     dest, src                    ; dest←src
```

MOV 指令把 1 字节或字的操作数从源地址 src 传送至目的地址 dest（其中，“←”表示赋值，下同）。源操作数可以是立即数、寄存器或主存单元，目的操作数可以是寄存器或主存单元，但不能是立即数。MOV 指令是采用寻址方式最多的指令，用约定的符号可以表达如下：

```
mov     reg/mem, imm                 ; 立即数送寄存器或主存
mov     reg/mem/seg, reg             ; 寄存器送寄存器（包括段寄存器）或主存
mov     reg/seg, mem                 ; 主存送寄存器（包括段寄存器）
```

```
mov     reg/mem, seg                    ; 段寄存器送主存或寄存器
```

也就是说，MOV 指令可以实现立即数到寄存器、立即数到主存的传送，以及寄存器与寄存器之间、寄存器与主存之间、寄存器与段寄存器之间的传送、主存与段寄存器之间的传送。

（1）立即数传送至通用寄存器（不包括段寄存器）或存储单元

```
mov     reg/mem, imm                    ; reg/mem←imm
```

【例 2.1】 立即数传送。

```
mov     al, 4                           ; al←4，字节传送
mov     cx, 0ffh                        ; cx←00ffh，字传送
mov     si, 200h                        ; si←0200h，字传送
mov     byte ptr[si], 0ah               ; ds:[si]←0ah，byte ptr 说明是字节操作
mov     word ptr[si+2], 0bh             ; ds:[si+2]←0bh，word ptr 说明是字操作
```

注意观察每条指令。例如，在上述第 2 条指令中，立即数（0FFH）使用了前导 0。因为在程序设计语言中字母开头通常表示标识符（如常量、变量、标号等），所以 MASM 规定十六进制数如果以字母开头需要添加前导 0，以便与标识符区别。同样，最后 2 条指令的立即数也使用了前导 0（0AH 和 0BH），如果缺少这个 0，则会被理解为寄存器（AH 和 BH）。

在包括传送指令的绝大多数双操作数指令中（除非特别说明），目的操作数与源操作数必须类型一致，或者同为字，或者同为字节，否则为非法指令。例如：

```
mov     al, 050ah                       ; 非法指令：050ah 为字，而 al 为字节
```

指定的寄存器有明确的字节或字类型，所以对应的立即数必须分别是字节或字。但在涉及存储器单元时，指令中给出的立即数可以理解为字，也可以理解为字节，此时必须显式指明。为了区别字节传送还是字传送，可用汇编操作符 byte ptr（字节）和 word ptr（字）指定。

注意，8086 不允许立即数传送至段寄存器，所以下列指令是非法的：

```
mov     ds, 100h                        ; 非法指令：不允许立即数至段寄存器的传送
```

（2）寄存器传送至寄存器（包括段寄存器）或存储单元

```
mov     reg/mem/seg, reg                ; reg/mem/seg←reg
```

【例 2.2】 寄存器传送。

```
mov     ax, bx
mov     ah, al
mov     ds, ax
mov     [bx], al
```

（3）存储单元传送至寄存器（包括段寄存器）

```
MOV     REG/SEG, MEM          ; REG/SEG←MEM
```

【例 2.3】 存储器传送。

```
mov     al, [bx]
mov     dx, [bp]              ; dx←ss:[bp]
mov     es, [si]              ; es←ds:[si]
```

8086 指令系统除串操作类指令外，不允许两个操作数都是存储单元，所以没有主存至主存的数据传送，要实现这种传送可通过寄存器间接实现。

【例 2.4】 buffer1 单元的数据传送至 buffer2 单元。buffer1 和 buffer2 是两个字变量。

```
mov     ax, buffer1           ; ax←buffer1（将 buffer1 内容送 ax）
mov     buffer2, ax           ; buffer2←ax
                              ; buffer1、buffer2 实际表示直接寻址方式
```

虽然存在通用寄存器和存储单元向 CS 段寄存器传送数据的指令，却不允许执行，因为这

样直接改变 CS 值将引起程序执行混乱。例如：

```
mov     cs, [si]              ; 不允许使用的指令
```

（4）段寄存器传送至通用寄存器（不包括段寄存器）或存储单元

```
mov     reg/mem, seg          ; reg/mem←seg
```

【例 2.5】 段寄存器传送。

```
mov     [si], ds
mov     ax, es
mov     ds, ax
```

注意，不允许段寄存器之间的直接数据传送。例如：

```
mov     ds, es                ; 非法指令：不允许 seg←seg 传送
```

2．交换指令 XCHG

交换指令用来将源操作数和目的操作数内容交换，其格式为：

```
xchg    reg, reg/mem          ; reg←→reg/mem，也可表达为：xchg  reg/mem, reg
```

XCHG 指令中操作数可以是字也可以是字节，可以在通用寄存器与通用寄存器或存储器之间对换数据，但不能在存储器与存储器之间交换数据。

【例 2.6】 用交换指令实现寄存器之间的数据交换。

```
mov     ax, 1234h             ; ax=1234h
mov     bx, 5678h             ; bx=5678h
xchg    ax, bx                ; ax=5678h，bx=1234h
xchg    ah, al                ; ax=7856h
```

【例 2.7】 用交换指令实现寄存器与存储器之间的数据交换。

```
xchg    ax, [2000h]           ; 也可以表达为 xchg  [2000h], ax
xchg    al, [2000h]           ; 也可以表达为 xchg  [2000h], al
```

3．换码指令 XLAT

换码指令用于将 BX 指定的缓冲区中，AL 指定的位移处的数据取出赋给 AL，格式为：

```
xlat    label
xlat                          ; al←ds:[bx+al]
```

这两种格式完全等效。第一种格式中，label 表示首地址的符号，既便于阅读也便于明确缓冲区所在的逻辑段；第二种格式也可以用 XLATB 助记符。实际的首地址在 BX 寄存器中。

【例 2.8】 将首地址为 100H 的表格缓冲区中的 3 号数据取出。

```
mov     bx, 100h
mov     al, 03h
xlat
```

换码指令常用于将一种代码转换为另一种代码，如扫描码转换为 ASCII 编码，数字 0～9 转换为 7 段显示码等。使用前，首先在主存中建立一个字节量表格，表格的内容是要转换成的目的代码，表格的首地址存放于 BX 寄存器中，需要转换的代码存放于 AL 寄存器，要求被转换的代码应是相对表格首地址的位移量。设置好后，执行换码指令，即将 AL 寄存器的内容转换为目标代码。

最后说明一点，因为 AL 的内容实际上是距离表格首地址的位移量，只有 8 位，所以表格的最大长度为 256，超过 256 的表格需要采用修改 BX 和 AL 的方法才能转换。

XLAT 指令中没有显式指明操作数，而是默认使用 BX 和 AL 寄存器。这种采用默认操作

数的方法称为隐含寻址方式，指令系统中有许多指令采用隐含寻址方式。

2.1.2 堆栈操作指令

堆栈是一个“先进后出”的主存区域，位于堆栈段中，使用 SS 段寄存器记录其段地址。堆栈只有一个出口，即当前栈顶。栈顶是地址较小的一端（低端），它用堆栈指针寄存器 SP 指定。在图 2-1(a)中，堆栈内还没有数据，是空的，此时栈顶和栈底指向同一个单元。

图 2-1　堆栈操作

堆栈有两种基本操作，对应有两条基本指令：进栈指令 PUSH 和出栈指令 POP。

1．进栈指令 PUSH

进栈指令先使堆栈指针 SP 减 2，然后把一个字操作数存入堆栈顶部。堆栈操作的对象只能是字操作数，进栈时，低字节存放于低地址，高字节存放在高地址，SP 相应向低地址移动 2 字节单元。

```
push    r16/m16/seg              ; sp←sp-2, ss:[sp]←r16/m16/seg
```

【例 2.9】 将 7812H 压入堆栈（见图 2-1(b)）。

```
mov     ax, 7812h
push    ax
```

再如，将主存单元 DS:[2000H]的一个字压入堆栈。

```
push    [2000h]
```

2．出栈指令 POP

出栈指令把栈顶的一个字传送至指定的目的操作数，然后堆栈指针 SP 加 2。目的操作数应为字操作数，字从栈顶弹出时，低地址字节送低字节，高地址字节送高字节。

```
pop     r16/m16/seg              ; r16/m16/seg←ss:[sp], sp←sp+2
```

【例 2.10】 将栈顶一个字的内容弹出送 AX 寄存器（见图 2-1(c)）。

```
pop     ax
```

再如，将栈顶一个字送入主存 DS:[2000H]：

```
pop     [2000h]
```

堆栈是系统中不可缺少的数据区域。堆栈可用来临时存放数据，以便随时恢复它们。堆栈也常用于在子程序间传递参数。在子程序中，通常需要保存被修改的寄存器内容，以便在返回时恢复它们，这时可用下例的方法。

【例 2.11】 现场的保护与恢复。

```
push    ax                    ; 进入子程序后（或调用子程序前）
push    bx
push    ds
…
pop     ds                    ; 返回主程序前（或调用子程序后）
pop     bx
pop     ax
```

注意：POP 指令的顺序与 PUSH 指令相反，因为堆栈是一个先进后出的区域，只有这样才能使各寄存器恢复原来内容。

8086 处理器的堆栈建立在主存区域中，使用 SS 段寄存器指向段基地址。堆栈段的范围由堆栈指针寄存器 SP 的初值确定，这个位置就是堆栈底部（不再变化）。堆栈只有一个数据出入口，即当前栈顶（不断变化），由堆栈指针寄存器 SP 的当前值指定栈顶的偏移地址，如图 2-2 所示。随着数据进入堆栈，SP 逐渐减小；数据依次弹出、SP 逐渐增大。随着 SP 增大，弹出的数据不再属于当前堆栈区域中；随后进入堆栈的数据也会占用这个存储空间。当然，如果进入堆栈的数据超出了设置的堆栈范围，或者已无数据可以弹出，即 SP 增大到栈底，就产生堆栈溢出错误。堆栈溢出，轻者使程序出错，重者导致系统崩溃。

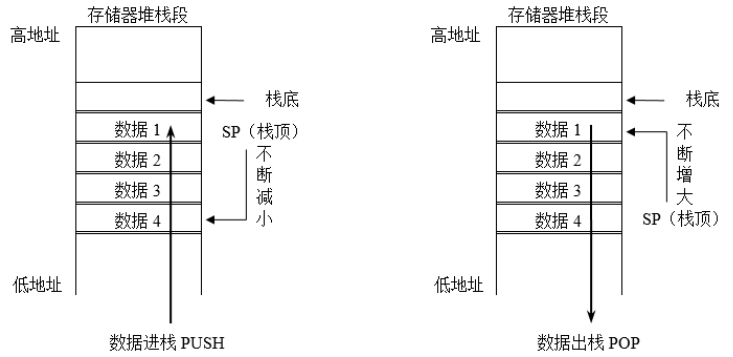

图 2-2　8086 处理器堆栈操作

堆栈操作常被比喻为“摞盘子”。盘子一个压着一个叠起来放进箱子里，就像数据进栈操作；叠起来的盘子应该从上面一个接一个拿走，就像数据出栈操作。最后放上去的盘子被最先拿走，就是堆栈的“后进先出”操作原则。不过，8086 处理器的堆栈段是“向下生长”的，即随着数据进栈，堆栈顶部（指针 SP）逐渐减小，所以可以想像成为一个倒扣的箱子，盘子（数据）从下面放进去。

3．堆栈的应用

堆栈是程序中不可或缺的一个存储区域。除堆栈操作指令外，还有子程序调用 CALL 和子程序返回 RET、中断调用 INT 和中断返回 IRET 等指令，以及内部异常、外部中断等情况都会使用堆栈、修改 SP 值（将在后续章节中逐渐展开）。

堆栈可用来临时存放数据，以便随时恢复它们。使用 POP 指令时，应该明确当前栈顶的数据是什么，可以按程序执行顺序向前观察由哪个操作压入了该数据。

既然堆栈是利用主存实现的，当然能以随机存取方式读写其中的数据。通用寄存器之一的堆栈基址指针 BP 就是出于这个目的而设计的。例如：

```
mov     bp, sp                ; bp←sp
```

```
mov     ax, [bp+4]          ; ax←ss:[bp+4], bp 默认与堆栈段配合
mov     [bp], ax            ; ss:[bp]←ax
```

利用堆栈实现主、子程序间传递参数就利用上述方法，这也是堆栈的主要作用之一。

堆栈还常用于子程序的寄存器保护和恢复。由于堆栈的栈顶和内容随着程序的执行不断变化，因此编程时要注意入栈和出栈的数据要成对，要保持堆栈平衡。

2.1.3 标志传送指令

1．标志寄存器传送

标志寄存器传送指令用来传送标志寄存器的内容，包括 LAHF/SAHF、PUSHF/POPF 指令。

（1）标志送 AH 指令 LAHF

LAHF 指令将标志寄存器 FLAGS 的低字节送寄存器 AH，即状态标志位 SF/ZF/AF/PF/CF 分别送入 AH 的第 7/6/4/2/0 位，而 AH 的第 5/3/1 位任意。

```
lahf                        ; ah←flags 的低字节
```

（2）AH 送标志指令 SAHF

SAHF 将 AH 寄存器内容送 FLAGS 的低字节，即根据 AH 的第 7、6、4、2、0 位相应设置 SF、ZF、AF、PF、CF 标志。由此可见，SAHF 和 LAHF 是一对功能相反的指令，它们只影响标志寄存器的低 8 位，而对高 8 位无影响。

```
sahf                        ; flags 的低字节←ah
```

（3）标志进栈指令 PUSHF

PUSHF 指令将标志寄存器的内容压入堆栈，同时栈顶指针 SP 减 2。这条指令可用来保存全部标志位。

```
pushf                       ; sp←sp-2, ss:[sp]←flags
```

（4）标志出栈指令 POPF

POPF 指令将栈顶字单元内容送标志寄存器，同时栈顶指针 SP 加 2。

```
popf                        ; flags←ss:[sp], sp←sp+2
```

【例 2.12】 置位单步标志 TF。

```
pushf                       ; 保存全部标志到堆栈
pop     ax                  ; 从堆栈中取出全部标志
or      ax, 0100h           ; 设置 d8=tf=1，而 ax 其他位不变
push    ax                  ; 将 ax 压入堆栈
popf                        ; 将堆栈内容取到标志寄存器，即 flags←ax
```

2．标志位操作

标志位操作指令可用来对 CF、DF 和 IF 三个标志位进行设置，除影响其所设置的标志外，均不影响其他标志。

```
clc                         ; 复位进位标志：CF←0
stc                         ; 置位进位标志：CF←1
cmc                         ; 求反进位标志：CF←~CF
cld                         ; 复位方向标志：DF←0
std                         ; 置位方向标志：DF←1
cli                         ; 复位中断标志，禁止可屏蔽中断：IF←0
sti                         ; 置位中断标志，允许可屏蔽中断：IF←1
```

许多指令的执行都会影响标志，上述指令提供了直接改变 CF、DF、IF 的方法。标志寄存

器中的其他标志需要用 LAHF/SAHF 或 PUSHF/POPF 指令间接改变。

2.1.4 地址传送指令

地址传送指令将存储器的逻辑地址送至指定的寄存器。

1．有效地址传送指令 LEA

LEA 指令将存储器操作数的有效地址传送至指定寄存器。

```
lea     r16, mem                ; r16←mem 的有效地址 ea
```

【例 2.13】 有效地址的获取。

```
mov     bx, 0400h
mov     si, 3ch
lea     bx, [bx+si+0f62h]       ; bx←bx+si+0f62h=0400h+3ch+0f62h=139eh
```

这里，BX 得到的是主存单元的有效地址，不是物理地址，也不是该单元的内容。

2．指针传送指令

LDS 和 LES 指令将主存中 MEM 指定的字送至 R16，并将 MEM 的下一字送 DS 或 ES 寄存器。实际上，MEM 指定了主存的连续 4 字节作为逻辑地址，即 32 位的地址指针。

```
lds     r16, mem                ; r16←mem，ds←mem+2
les     r16, mem                ; r16←mem，es←mem+2
```

【例 2.14】 地址指针的传送。

```
mov     word ptr [3060h], 0100h
mov     word ptr [3062h], 1450h
lds     si, [3060h]             ; ds=1450h，si=0100h
les     di, [3060h]             ; es=1450h，di=0100h
```

2.2 算术运算类指令

算术运算类指令用来执行二进制数及十进制数的算术运算：加、减、乘、除。这类指令会根据运算结果影响状态标志，有时要利用某些标志才能得到正确的结果。

2.1 节介绍的数据传送类指令中，除了标志为目的操作数的标志传送指令外，其他传送指令并不影响标志；也就是说，标志并不因为传送指令的执行而改变，所以没有涉及标志问题。但现在我们需要了解它们了。

2.2.1 状态标志

一方面，状态标志作为加、减运算和逻辑运算等指令的辅助结果，另一方面，用于构成各种条件、实现程序分支，是汇编语言编程中非常重要的方面。

1．进位标志 CF（Carry Flag）

处理器设计的进（借）位标志类似十进制数据加减运算中的进位和借位，不过只是体现二进制数据最高位的进位或借位。具体来说，当加减运算结果的最高有效位有进位（加法）或借位（减法）时，将设置进位标志为 1，即 CF=1；如果没有进位或借位，则设置进位标志为 0，

即 CF=0。换句话说，加减运算后，如果 CF=1，则说明数据运算过程中出现了进位或借位；如果 CF=0，则说明没有进位或借位。

例如，有两个 8 位二进制数 00111010 和 01111100，如果相加，运算结果是 10110110。运算过程中，最高位没有向上再进位，所以这个运算结果将使得 CF=0。但如果是 10101010 和 01111100 相加，结果是[1]00100110，出现了向高位进位（用[ ]表达），所以这个运算结果将使得 CF=1。

进位标志是针对无符号整数运算设计的，反映无符号数据加减运算结果是否超出范围、是否需要利用进（借）位反映正确结果。*N* 位无符号整数表示的范围是：0～2^N–1。如果相应位数的加减运算结果超出了其能够表示的这个范围，就是产生了进位或借位。

在上面例子中，二进制数据 00111010+01111100=10110110 被转换成十进制数表示是 58+124=182。运算结果 182 仍在 0～255 范围之内，没有产生进位，所以 CF=0。

对于二进制数据 10101010+01111100=[1]00100110，将它们转换成十进制数表示是 170+1 = 294=256+38。运算结果 294 超出了 0～255 范围，所以使得 CF=1。这里，进位 CF=1 表示了十进制数据 256。

2．溢出标志 OF（Overflow Flag）

把水倒入茶杯时，如果超出了茶杯容量，水会漫出来，这就是溢出的本意：一个容器不能存放超过其容积的物体。同理，处理器设计的溢出标志用于表示有符号整数进行加减运算的结果是否超出范围。若超出范围，就是有溢出，将设置溢出标志 OF=1，否则 OF=0。

溢出标志是针对有符号整数运算设计的，反映有符号数据加减运算结果是否超出范围。处理器默认采用补码形式表示有符号整数，*N* 位补码表达的范围是：-2^{N-1}～$+2^{N-1}$–1。如果有符号整数运算结果超出了这个范围，就是产生了溢出。

对上面例子的两个 8 位二进制数 00111010 和 01111100，按照有符号数的补码规则它们都是正整数，即十进制数 58 和 124。它们求和的结果是二进制数 10110110，即十进制数 182。运算结果 182 超出了−128～+127 范围，产生溢出，所以 OF=1。另一方面，按照补码规则，8 位二进制数结果 10110110 的最高位为 1，实际上表达的是负数，所以在溢出情况下的运算结果是错误的。

对于二进制数 10101010，最高位是 1，按照补码规则表达负数，求反加 1 得到绝对值，即十进制数−86。它与二进制数 01111100（十进制数表示为 124）相加，结果是[1]00100110。因为进行有符号数据运算，所以不考虑无符号运算出现的进位，00100110 才是我们需要的结果，即 38（−86+124）。运算结果 38 没有超出−128～+127 范围，将使得 OF=0。所以，有符号数据进行加减运算，只有在没有溢出情况下才是正确的。

注意，溢出标志 OF 和进位标志 CF 是两个意义不同的标志。进位标志表示无符号整数运算结果是否超出范围，超出范围后加上进位或借位运算结果仍然正确；而溢出标志表示有符号整数运算结果是否超出范围，超出范围运算结果不正确。处理器对两个操作数进行运算时，按照无符号整数求得结果，并相应设置进位标志 CF；同时，根据是否超出有符号整数的范围设置溢出标志 OF。应该利用哪个标志，则由程序员来决定。也就是说，如果将参加运算的操作数认为是无符号数，就应该关心进位；若认为是有符号数，则要注意是否溢出。

处理器利用异或门等电路判断运算结果是否溢出。按照处理器硬件的方法或者前面论述的原则进行判断会比较麻烦，这里给出一个人工判断的简单规则：只有当两个相同符号数相加（含

两个不同符号数相减）而运算结果的符号与原数据符号相反时，才产生溢出，因为此时的运算结果显然不正确，在其他情况下则不会产生溢出。

3．其他状态标志

零标志 ZF（Zero Flag）反映运算结果是否为 0。若运算结果为 0，则设置 ZF=1，否则 ZF=0。例如，8 位二进制数 00111010+01111100=10110110，结果不是 0，所以设置 ZF=0。如果是 8 位二进制数 10000100+01111100=[1]00000000，最高位进位有进位 CF 标志反映，除此之外的结果是 0，所以这个运算结果将使得 ZF=1。注意，零标志 ZF=1，反映结果是 0。

符号标志 SF（Sign Flag）反映运算结果是正数还是负数。处理器通过符号位可以判断数据的正负，因为符号位是二进制数的最高位，所以运算结果最高位（符号位）是符号标志的状态，即运算结果最高位为 1，则 SF=1，否则 SF=0。例如，8 位二进制数 00111010+01111100= 10110110，结果最高位是 1，所以设置 SF=1。如果是 8 位二进制数 10000100+01111100 = [1]00000000，最高位是 0（进位 1 不是最高位），所以这个运算结果将使得 SF=0。

奇偶标志 PF（Parity Flag）反映运算结果最低字节中“1”的个数是偶数还是奇数，便于用软件编程实现奇偶校验。最低字节中“1”的个数为零或偶数时，PF=1；为奇数时，PF=0。例如，8 位二进制数 00111010+01111100=10110110，结果中“1”的个数为 5 个，是奇数，故设置 PF=0。如果是 8 位二进制数 10000100+01111100=[1]00000000，除进位外的结果是零个“1”，所以这个运算结果将使得 PF=1。注意，PF 标志仅反映最低 8 位中“1”的个数是偶数或奇数，不管进行 16 位或 32 位操作。

加减运算结果将同时影响上述 5 个标志，表 2-1 总结了前面示例，便于对比理解。

表 2-1　加法运算结果对标志的影响

加法运算及其结果	CF	OF	ZF	SF	PF
00111010+01111100=[0]10110110	0	1	0	1	0
10101010+01111100=[1]00100110	1	0	0	0	0
10000100+01111100=[1]00000000	1	0	1	0	1

调整标志 AF（Adjust Flag）反映加减运算时最低半字节有无进位或借位。最低半字节（即 D_3 位向 D_4 位）有进位或借位时，AF=1，否则 AF=0。调整标志主要由处理器内部使用，用于十进制数算术运算的调整指令，用户一般不必关心。例如，8 位二进制数 00111010+ 01111100 = 10110110，低 4 位有进位，所以 AF=1。

2.2.2　加法指令

加法指令包括 ADD、ADC 和 INC 指令，执行字或字节的加法运算。

1．加法指令 ADD

加法指令 ADD 将源操作数与目的操作数相加，结果送到目的操作数，支持寄存器与立即数、寄存器、存储单元，以及存储单元与立即数、寄存器间的加法运算。

```
add     reg, imm/reg/mem            ; reg←reg+imm/reg/mem
add     mem, imm/reg                ; mem←mem+imm/reg
```

【例 2.15】 加法运算。

```
mov     al, 0fbh                    ; al=0fbh
```

```
add     al, 07h                         ; al=02h
mov     word ptr [200h], 4652h          ; [200h]=4652h
mov     bx, 1feh                        ; bx=1feh
add     al, bl                          ; al=00h
add     word ptr [bx+2], 0f0f0h         ; [200h]=3742h
```

ADD 指令按照状态标志的定义相应设置这些标志的 0 或 1 状态。例如二进制 8 位加法 07+FBH→02H 运算后，标志为 OF=0，SF=0，ZF=0，AF=1，PF=0，CF=1；用调试程序单步执行后，上述标志状态依次为 NV，PL，NZ，AC，PO，CY。

同样，进行二进制 16 位加法 4652H+F0F0H→3742H 运算后，标志为 OF=0，SF=0，ZF=0，AF=0，PF=1，CF=1；调试程序依次显示为 NV，PL，NZ，NA，PE，CY。注意，PF 仅反映低 8 位中“1”的个数，AF 只反映 D_3 对 D_4 位是否有进位。

2．带进位加法指令 ADC

ADC 指令除完成 ADD 加法运算外，还要加进位 CF，其用法及对状态标志的影响也与 ADD 指令一样。ADC 指令主要用于与 ADD 指令相结合实现多精度数相加。

```
adc     reg, imm/reg/mem                ; reg←reg+imm/reg/mem+cf
adc     mem, imm/reg                    ; mem←mem+imm/reg+cf
```

【例 2.16】 无符号双字加法运算。

```
mov     ax, 4652h                       ; ax=4652h
add     ax, 0f0f0h                      ; ax=3742h, cf=1
mov     dx, 0234h                       ; dx=0234h
adc     dx, 0f0f0h                      ; dx=f325h, cf=0
```

上述程序段完成 DX.AX=0234 4652H+F0F0 F0F0H=F325 3742H。

3．增量指令 INC

INC 指令对操作数加 1（增量），是一个单操作数指令，操作数可以是寄存器或存储器。

```
inc     reg/mem                         ; reg/mem←reg/mem+1
```

例如：

```
inc     bx
inc     byte ptr [bx]
```

设计加 1 指令和后面介绍的减 1 指令的目的是用于对计数器和地址指针的调整，所以它们不影响进位 CF 标志，对其他状态标志位的影响与 ADD、ADC 指令一样。

2.2.3 减法指令

减法指令包括 SUB、SBB、DEC、NEG 和 CMP，执行字或字节的减法运算，除 DEC 不影响 CF 标志外，其他减法指令按定义影响全部状态标志位。

1．减法指令 SUB

减法指令 SUB 使目的操作数减去源操作数，结果送目的操作数，支持的操作数类型同加法指令。

```
sub     reg, imm/reg/mem            ; reg←reg-imm/reg/mem
sub     mem, imm/reg                ; mem←mem-imm/reg
```

【例 2.17】 减法运算。

```
mov     al, 0fbh                    ; al=0fbh
```

```
sub     al, 07h                 ; al=0f4h, cf=0
mov     word ptr[200h], 4652h   ; [200h]=4652h
mov     bx, 1feh                ; bx=1feh
sub     al, bl                  ; al=0f6h, cf=1
sub     word ptr[bx+2], 0f0f0h  ; [200h]=5562h, cf=1
```

2．带借位减法指令 SBB

带借位减法指令 SBB 使目的操作数减去源操作数，还要减去借（进）位 CF，结果送到目的操作数。SBB 指令主要用于与 SUB 指令相结合，实现多精度数相减。

```
sbb     reg,imm/reg/mem         ; reg←reg-imm/reg/mem-cf
sbb     mem,imm/reg             ; mem←mem-imm/reg-cf
```

【例 2.18】 无符号双字减法运算。

```
mov     ax, 4652h               ; ax=4652h
sub     ax, 0f0f0h              ; ax=5562h, of=0, sf=0, zf=0, af=0, pf=0, cf=1
mov     dx, 0234h               ; dx=0234h
sbb     dx, 0f0f0h              ; dx=1143h, of=0, sf=0, zf=0, af=0, pf=0, cf=1
```

上述程序段完成 DX.AX=0234 4652H–F0F0 F0F0H=1143 5562H，有借位 CF=1。

3．减量指令 DEC

DEC 指令对操作数减 1（减量），是一个单操作数指令，操作数可以是寄存器或存储器。

```
dec     reg/mem                 ; reg/mem←reg/mem-1
```

同 INC 指令一样，DEC 指令不影响 CF，但影响其他状态标志。例如：

```
dec     cx
dec     word ptr [si]
```

4．求补指令 NEG

NEG 指令也是一个单操作数指令，对操作数执行求补运算，即用零减去操作数，然后将结果返回操作数。求补运算也可以表达成：将操作数按位取反后加 1。NEG 指令对标志的影响与用零做减法的 SUB 指令一样。

```
neg     reg/mem                 ; reg/mem←0-reg/mem
```

【例 2.19】 求补运算。

```
mov     ax, 0ff64h
neg     al                  ; ax=ff9ch, of=0, sf=1, zf=0, pf=1, cf=1
sub     al, 9dh             ; ax=ffffh, of=0, sf=1, zf=0, pf=1, cf=1
neg     ax                  ; ax=0001h, of=0, sf=0, zf=0, pf=0, cf=1
dec     al                  ; ax=0000h, of=0, sf=0, zf=1, pf=1, cf=1
neg     ax                  ; ax=0000h, of=0, sf=0, zf=1, pf=1, cf=0
```

5．比较指令 CMP

```
cmp     reg, imm/reg/mem    ; reg-imm/reg/mem
cmp     mem, imm/reg        ; mem-imm/reg
```

比较指令将目的操作数减去源操作数，但结果不回送目的操作数。也就是说，CMP 指令与减法指令 SUB 执行同样的操作，同样影响标志，只是不改变目的操作数。

CMP 指令用于比较两个操作数的大小关系。执行比较指令之后，可以根据标志判断两个数是否相等、大小关系等。所以，CMP 指令后面常跟条件转移指令，根据比较结果不同产生不同的分支。另外，CMP 指令的操作数与 ADD/ADC、SUB/SBB 指令都一样。

【例 2.20】 比较 AL 是否大于 100。

```
        cmp     al, 100             ; al-100
        jb      below               ; al<100，跳转到 below 执行
        sub     al, 100             ; al≥100，al←al-100
        inc     ah                  ; ah←ah+1
below:  …
```

2.2.4 乘法指令

乘法指令用来实现两个二进制操作数的相乘运算，包括两条指令：无符号数乘法指令 MUL 和有符号数乘法指令 IMUL。

```
        mul     r8/m8               ; 无符号字节乘：ax←al×r8/m8
        mul     r16/m16             ; 无符号字乘：dx.ax←ax×r16/m16
        imul    r8/m8               ; 有符号字节乘：ax←al×r8/m8
        imul    r16/m16             ; 有符号字乘：dx.ax←ax×r16/m16
```

乘法指令隐含使用一个操作数 AX 和 DX，源操作数则显式给出，可以是寄存器或存储单元。若是字节量相乘，则 AL 与 r8/m8 相乘得到 16 位的字，存入 AX 中；若是 16 位数据相乘，则 AX 与 r16/m16 相乘，得到 32 位的结果，其高字存入 DX，低字存入 AX 中。

乘法指令利用对 OF 和 CF 的影响，可以判断相乘的结果中高一半是否含有有效数值。如果乘积的高一半（AH 或 DX）没有有效数值，即对 MUL 指令高一半为 0，对 IMUL 指令高一半是低一半的符号扩展，则 OF=CF=0；否则 OF=CF=1。

乘法指令对其他状态标志的影响没有定义，即成为任意，不可预测。注意，这与对标志没有影响是不同的，没有影响是指不改变原来的状态。

【例 2.21】 无符号数 0B4H 与 11H 相乘。

```
        mov     al, 0b4h            ; al=b4h=180d
        mov     bl, 11h             ; bl=11h=17d
        mul     bl                  ; ax=0bf4h=3060d,of=cf=1（ax 高 8 位不为 0）
```

注意：含有结尾字母 D 表示这是一个十进制数，目的是便于与十六进制数进行比较。

这里 B4H 按照无符号整数编码是真值 180，与 17 相乘结果为 3060，即十六进制数 0BF4H。如果 B4H 按有符号整数编码（补码）理解，则真值是–76，与 17 相乘结果为–1292，用补码表示是 FAF4H。进行有符号乘法的程序片段如下：

```
        mov     al, 0b4h            ; al=b4h=-76d
        mov     bl, 11h             ; bl=11h=17d
        imul    bl                  ; ax=faf4h=-1292d,of=cf=1
                                    ; ax 高 8 位不是低 8 位的符号扩展，表示含有有效数字
```

计算二进制数乘法：B4H×11H。如果把它当做无符号数，用 MUL 指令，则结果为 0BF4H；如果看做有符号数，用 IMUL 指令，则结果为 FAF4H。由此可见，同样的二进制数看做无符号数与有符号数相乘，即采用 MUL 与 IMUL 指令，其结果是不相同的。

2.2.5 除法指令

除法指令执行两个二进制数的除法运算，包括无符号二进制数除法指令 DIV 和有符号二进制数除法指令 IDIV 两条指令。

```
        div     r8/m8               ; 无符号字节除：al←ax÷r8/m8 的商
```

```
                                ; ah←ax÷r8/m8 的余数
        div     r16/m16         ; 无符号字除: ax←dx.ax÷r16/m16 的商
                                ; dx←dx.ax÷r16/m16 的余数
        idiv    r8/m8           ; 有符号字节除: al←ax÷r8/m8 的商
                                ; ah←ax÷r8/m8 的余数
        idiv    r16/m16         ; 有符号字除: ax←dx.ax÷r16/m16 的商
                                ; dx←dx.ax÷r16/m16 的余数
```

除法指令隐含使用 DX 和 AX 作为一个操作数，指令中给出的源操作数是除数。如果是字节除法，AX 除以 R8/M8，8 位商存入 AL，8 位余数存入 AH。如果是字除法，DX.AX 除以 R16/M16，16 位商存入 AX，16 位余数存入 DX。余数的符号与被除数符号相同。

【例 2.22】 无符号数 0400H 除以 B4H。

```
        mov     ax, 0400h       ; ax=400h=1024d
        mov     bl, 0b4h        ; bl=b4h=180d
        div     bl              ; 商 al=05h，余数 ah=7ch=124d
```

同样，若是有符号数 0400H 除以 B4H，则

```
        mov     ax, 0400h       ; ax=400h=1024d
        mov     bl, 0b4h        ; bl=b4h=-76d
        idiv    bl              ; 商 al=f3h=-13d，余数 ah=24h=36d
```

除法指令 DIV 和 IDIV 对标志的影响没有定义，却可能产生溢出。当被除数远大于除数时，所得的商就有可能超出它所能表达的范围。如果存放商的寄存器 AL/AX 不能表达，便产生溢出，8086 CPU 中就产生编号为 0 的内部中断（见 2.4.5 节）。实用的程序中应该考虑这个问题，操作系统通常只提示出错。

对 DIV 指令，除数为 0，或者在字节除时商超过 8 位，或者在字除时商超过 16 位，则发生除法溢出。对 IDIV 指令，除数为 0，或者在字节除时商不在–128～127 范围内，或者在字除时商不在–32768～32767 范围内，则发生除法溢出。

2.2.6 符号扩展指令

8086 处理器支持 8 和 16 位数据操作，大多数指令要求两个操作数类型一致。但是，实际的数据类型不一定满足要求。例如，16 位与 8 位数据的加减运算，需要先将 8 位扩展为 16 位；16 位除法需要将被除数扩展成 32 位。不过，位数扩展后数据大小不能因此改变。

对无符号数据，只要在前面加 0 就实现了位数扩展、大小不变，这就是零位扩展（Zero Extension）。例如，8 位无符号数据 80H（=128），零位扩展为 16 位 0080H（=128）。8086 没有设计实现零位扩展的指令，需要时可以直接对高位进行赋值 0 实现。

对有符号数据（补码）表示，增加位数而保持数据大小不变，需要进行符号扩展（Flag Extension），即用一个操作数的符号位（即最高位）形成另一个操作数，增加的各位全部是符号位的状态。例如，8 位有符号数据 64H（=100）为正数，符号位为 0，（高位）符号扩展成 16 位是 0064H（=100）。再如：16 位有符号数据 FF00H（=–256）为负数，符号位为 1，符号扩展成 32 位是 FFFFFF00H（=–256）。典型的例子是真值“–1”，字节量补码表达是 FFH，字量补码是 FFFFH，双字量补码表达为 FFFFFFFFH。

8086 设计有 2 条符号扩展指令 CBW 和 CWD。CBW 指令将 AL 的最高有效位 D_7 扩展至 AH，即：如果 AL 的最高有效位是 0，则 AH=00；AL 的最高有效位为 1，则 AH=FFH，AL 不变。CWD 将 AX 的内容符号扩展形成 DX，即：如果 AX 的最高有效位 D_{15} 为 0，则 DX=0000H；

如果 AX 的最高有效位 D_{15} 为 1，则 DX=FFFFH。

```
cbw                         ; al 符号扩展到 ax
cwd                         ; ax 符号扩展到 dx 和 ax 寄存器对（dx.ax）
```

【例 2.23】 符号扩展

```
mov     al, 80h             ; al=80h
cbw                         ; ax=ff80h
add     al, 255             ; al=7fh
cbw                         ; ax=007fh
```

符号扩展指令常用来获得除法指令所需要的被除数。

【例 2.24】 进行有符号数除法 AX÷BX

```
cwd
idiv    bx
```

整数数据经过零位或者符号扩展增加了位数，大小没有变化，新扩展的位数只是数据的符号，并没有数值含义。反过来说，如果高位部分都是符号位，可以截断这些高位部分，也不改变数据大小。例如，真值–1 用 32 位补码表达为 FFFFFFFFH，高位都是符号位，所以截断高 16 位，得到 16 位表达是 FFFFH；其实“–1”用 8 位就可以表达了，所以可以再截断高 8 位。

这时，回过来理解乘法指令对标志 OF 和 CF 影响的设计原因。两个 *N* 位数据相乘，可能得到 2*N* 位的乘积。但如果乘积的高一半是低一半的符号位扩展，说明高一半不含有效数值，就可以放心地截断高一半而不影响正确的结果。

2.2.7 十进制调整指令

前面介绍的算术运算指令都是针对二进制数的。然而，十进制是我们日常使用的进制。为了方便进行十进制数的运算，8086 提供一组十进制数调整指令。这组指令对二进制数运算的结果进行十进制调整，以得到十进制数的运算结果。

十进制数在计算机中也要用二进制编码表示，这就是二进制编码的十进制数：BCD 码。8086 支持压缩 BCD 码和非压缩 BCD 码，相应地，十进制调整指令分为压缩 BCD 码调整指令和非压缩 BCD 码调整指令。

1．压缩 BCD 码调整指令

压缩 BCD 码是通常的 8421 码，它用 4 个二进制位表示一个十进制位，1 字节可以表示两个十进制位，即 00～99。压缩 BCD 码调整指令包括加法和减法的十进制调整指令 DAA 和 DAS，用来对二进制数加、减法指令的执行结果进行调整，得到十进制数结果。注意，在使用 DAA 或 DAS 指令前，应先执行加法或减法指令。DAA 指令跟在以 AL 为目的操作数的 ADD 或 ADC 指令后，对 AL 的二进制数结果进行十进制调整，并在 AL 中得到十进制数结果。DAS 指令跟在以 AL 为目的操作数的 SUB 或 SBB 指令之后，对 AL 的二进制数结果进行十进制调整，并在 AL 中得到十进制数结果。

```
daa                         ; al←将 al 中的加和调整为压缩 bcd 码
das                         ; al←将 al 中的减差调整为压缩 bcd 码
```

DAA 和 DAS 指令对 OF 标志无定义，按结果影响所有其他标志，其中 CF 反映压缩 BCD 码相加减的进借位状态。

【例 2.25】 压缩 BCD 码的加法运算。

```
mov     al, 68h             ; al=68h，表示压缩 bcd 码 68
```

```
mov     bl, 28h             ; bl=28h，表示压缩 bcd 码 28
add     al, bl              ; 二进制数加法：al=68h+28h=90h
daa                         ; 十进制调整：al=96h
                            ; 实现压缩 bcd 码加法：68+28=96
```

再如，压缩 BCD 码的减法运算：

```
mov     al, 68h             ; al=68h，表示压缩 bcd 码 68
mov     bl, 28h             ; bl=28h，表示压缩 bcd 码 28
sub     al, bl              ; 二进制数减法：al=68h-28h=40h
das                         ; 十进制调整：al=40h
                            ; 实现压缩 bcd 码减法：68-28=40
```

【例 2.26】 已知 AX=1234H，BX=4612H，计算 1234–4612 的差。

```
sub     al, bl
das
xchg    al, ah
sbb     al, bh
das
xchg    al, ah              ; ax=6622h，cf=1
```

把 1234H 和 4612H 认为是无符号十进制数，则利用借位 1，则 1234–4612=6622，结果正确。如果认为是有符号十进制数，则 1234–4612=–(4612–1234)= –3378，结果仍然正确吗？正确，因为用补码表示–3378 就是 6622（0000–6622=9999–6622+1）。实际上，位数为 n 的十进制整数 d，其补码定义为：10^n-d。

2．非压缩 BCD 码调整指令

非压缩 BCD 码用 8 个二进制位表示一个十进制位，实际上只是用低 4 个二进制位表示一个十进制位 0～9，高 4 位任意，但通常默认为 0。0～9 的 ASCII 编码是 30H～39H，所以 0～9 的 ASCII 编码（高 4 位变为 0）就可以认为是非压缩 BCD 码。

对非压缩 BCD 码，8086 提供 AAA、AAS、AAM 和 AAD 四条指令，分别用于对二进制数加、减、乘、除指令的结果进行调整，以得到非压缩 BCD 码表示的十进制数结果。由于只要在调整后的结果中加上 30H 就成为 ASCII 编码，所以这组指令实际上也是针对 ASCII 编码的调整指令的。

（1）加法的非压缩 BCD 码调整指令 AAA

```
aaa                         ; al←将 al 中的加和调整为非压缩 bcd 码
                            ; ah←ah+调整产生的进位
```

该指令跟在以 AL 为目的操作数的 ADD 或 ADC 指令之后，对 AL 进行非压缩 BCD 码调整。如果调整中产生了进位，则将进位 1 加到 AH 中，同时 CF=AF=1，否则 CF=AF=0。AAA 指令对其他标志无定义。另外，该指令使 AL 的高 4 位清 0。

【例 2.27a】 非压缩 BCD 码的加法运算。

```
mov     ax, 0608h           ; ax=0608h，表示非压缩 bcd 码 68
mov     bl, 09h             ; bl=09h，表示非压缩 bcd 码 9
add     al, bl              ; 二进制数加法：al=08h+09h=11h
aaa                         ; 十进制调整：ax=0707h
                            ; 实现非压缩 bcd 码加法：68+9=77
```

（2）减法的非压缩 BCD 码调整指令 AAS

```
aas                    ; al←将 al 中的减差调整为非压缩 BCD 码，ah←ah-调整产生的借位
```

该指令跟在以 AL 为目的操作数的 SUB 或 SBB 指令之后，对 AL 进行非压缩 BCD 码调整。如果调整中产生了借位，则将 AH 减去借位 1，同时 CF=AF=1，否则 CF=AF=0。AAS 指令对其他标志无定义。另外，该指令使 AL 的高 4 位清 0。

【例 2.27b】 非压缩 BCD 码的减法运算。

```
mov     ax, 0608h      ; ax=0608h，表示非压缩 BCD 码 68
mov     bl, 09h        ; bl=09h，表示非压缩 BCD 码 9
sub     al, bl         ; 二进制数减法：al=08h-09h=ffh
aas                    ; 十进制调整：ax=0509h
                       ; 实现非压缩 BCD 码减法：68-9=59
```

（3）乘法的非压缩 BCD 码调整指令 AAM

```
aam                    ; ax←将 ax 中的乘积调整为非压缩 BCD 码
```

该指令跟在以 AX 为目的操作数的 MUL 指令后，对 AX 进行非压缩 BCD 码调整。利用 MUL 相乘的两个非压缩 BCD 码的高 4 位必须为 0。AAM 指令根据结果设置 SF、ZF 和 PF，但 OF、CF 和 AF 无定义。

【例 2.27c】 非压缩 BCD 码的乘法运算。

```
mov     ax, 0608h      ; ax=0608h，表示非压缩 BCD 码 68
mov     bl, 09h        ; bl=09h，表示非压缩 BCD 码 9
mul     bl             ; 二进制数乘法：ax=08h×09h=0048h
aam                    ; 十进制调整：ax=0702h，实现非压缩 BCD 码乘法：8×9=72
```

（4）除法的非压缩 BCD 码调整指令 AAD

```
aad                    ; ax←将 ax 中的非压缩 BCD 码扩展成二进制数，即：
                       ; al←10×ah+al，ah←0
```

AAD 调整指令与其他调整指令的应用情况不同，是先将存放在 AX 寄存器中的两位非压缩 BCD 码数进行调整，再用 DIV 指令除以一个非压缩 BCD 码数，这样得到非压缩 BCD 码数的除法结果。其中，要求 AL、AH 和除数的高 4 位为 0。AAD 指令根据结果设置 SF、ZF 和 PF，但 OF、CF 和 AF 无定义。

【例 2.27d】 非压缩 BCD 码的除法运算。

```
mov     ax, 0608h      ; ax=0608h，表示非压缩 BCD 码 68
mov     bl, 09h        ; bl=09h，表示非压缩 BCD 码 9
aad                    ; 二进制扩展：ax=68=0044h
div     bl             ; 除法运算：商 al=07h，余数 ah=05h
                       ; 实现非压缩 BCD 码除法：68=7×9+5
```

十进制调整指令只针对要求 BCD 码运算的应用，并且要与对应的运算指令配合。表 2-2 总结了 8086 算术运算的各种情况。

表 2-2　8086 支持的 4 种数据类型的算术运算

	加法	减法	乘法	除法
无符号二进制数	ADD，ADC	SUB，SBB	MUL	DIV
有符号二进制数	ADD，ADC	SUB，SBB	IMUL	IDIV
压缩 BCD 码	ADD，ADC，DAA	SUB，SBB，DAS		
非压缩 BCD 码	ADD，ADC，AAA	SUB，SBB，AAS	MUL，AAM	AAD，DIV

2.3 位操作类指令

位操作类指令对二进制数的各位进行操作，包括逻辑运算指令和移位指令。

2.3.1 逻辑运算指令

逻辑运算指令用来对字或字节按位进行逻辑运算，包括 5 条指令：逻辑与 AND、逻辑或 OR、逻辑非 NOT、逻辑异或 XOR 和测试 TEST。

1．逻辑与指令 AND

AND 指令对两个操作数执行按位的逻辑与运算，即只有相“与”的两位都是 1 结果才是 1，否则结果为 0。逻辑与的结果送目的操作数。

```
AND     DEST, SRC                       ; DEST←DEST∧SRC（符号∧表示逻辑与）
```

AND 指令及后面介绍的其他双操作数逻辑指令 OR、XOR 和 TEST 指令，所支持的操作数组合同加减法指令一样。

```
逻辑运算助记符    reg, imm/reg/mem      ; reg←reg 逻辑运算 imm/reg/mem
逻辑运算助记符    mem, imm/reg          ; mem←mem 逻辑运算 imm/reg
```

在这两个操作数中，源操作数可以是任意寻址方式，目的操作数只能是立即数外的其他寻址方式，并且两个操作数不能同时为存储器寻址方式。

所有双操作数的逻辑指令均设置 CF=OF=0，根据结果设置 SF、ZF 和 PF 状态，而对 AF 未定义。

【例 2.28】 逻辑与运算。

```
mov     al, 45h
and     al, 31h                     ; AL=01H, CF=OF=0, SF=0, ZF=0, PF=0
```

AND 指令可用于复位一些位，但不影响其他位。这时只需将要置 0 的位同 0 相“与”，而维持不变的位同 1 相“与”就可以了。

再如，将 BL 中 D_0 和 D_3 清 0，其余位不变，则

```
and     bl, 11110110b
```

2．逻辑或指令 OR

OR 指令对两个操作数执行按位的逻辑或运算，即只要相“或”的两位中有一位是 1，结果就是 1，否则结果为 0。逻辑或的结果送目的操作数。所支持的操作数如 AND 指令。

```
or      dest, src                   ; dest←dest∨src（符号∨表示逻辑或）
```

【例 2.29】 逻辑或运算。

```
mov     al, 45h
or      al, 31h                     ; AL=75H, CF=OF=0, SF=0, ZF=0, PF=0
```

OR 指令可用于置位某些位，而不影响其他位。这时只需将要置 1 的位同 1 相“或”，维持不变的位同 0 相“或”即可。

再如，将 BL 中 D_0 和 D_3 置 1，其余位不变，则

```
or      bl, 00001001b
```

3．逻辑异或指令 XOR

XOR 指令对两个操作数执行按位的逻辑异或运算，即相“异或”的两位不相同时，结果

就是 1，否则结果为 0。其结果送目的操作数。所支持的操作数如 AND 指令。

```
        xor     dest, src                   ; dest←dest⊕src（符号⊕表示逻辑异或）
```

【例 2.30】 逻辑异或运算。

```
        mov     al, 45h
        xor     al, 31h                     ; AL=74H, CF=OF=0, SF=0, ZF=0, PF=1
```

XOR 可用于求反某些位，而不影响其他位。要求求反的位同 1 相“异或”，维持不变的位同 0 相“异或”。

再如，将 BL 中 D_0 和 D_3 求反，其余位不变，则

```
        xor     bl, 00001001b
```

XOR 指令经常给寄存器清 0，同时使 CF 清 0。例如：

```
        xor     ax, ax                      ; AX=0, CF=OF=0, SF=0, ZF=1, PF=1
```

4．逻辑非指令 NOT

NOT 指令对操作数按位取反，即原来为 0 的位变成 1，原来为 1 的位变成 0。NOT 指令是一个单操作数指令，该操作数可以是立即数外的任何寻址方式。注意，NOT 指令不影响标志位。

```
        not reg/mem                         ; reg/mem← ~reg/mem（符号~表示逻辑反）
```

【例 2.31】 逻辑非运算。

```
        mov     al, 45h
        not     al                          ; AL=0BAH, 标志不变
```

5．测试指令 TEST

TEST 指令对两个操作数执行按位的逻辑与运算，但结果不回到目的操作数。TEST 指令执行的操作与 AND 指令相同，但不保存执行结果，只根据结果来设置状态标志。

```
        test    dest, src                   ; dest∧src（符号∧表示逻辑与）
```

TEST 指令通常用于检测一些条件是否满足又不希望改变原操作数的情况。这条指令之后一般是条件转移指令，目的是利用测试条件转向不同的程序段。

【例 2.32】 TEST 指令用于测试某一（几）位是否（同时）为 0 或为 1。

```
        test    al, 01h                     ; 测试 al 的最低位 d0
        jnz     there                       ; 标志 zf=0，即 d0=1，则程序转移到 there
        …                                   ; 否则 zf=1，即 d0=0，顺序执行
there:  …
```

2.3.2 移位指令

移位（Shift）指令分成逻辑（Logical）移位指令和算术（Arithmetic）移位指令，分别具有左移（Left）或右移（Right）操作，如图 2-3 所示。

```
        shl     reg/mem, 1/cl     ; 逻辑左移: reg/mem 左移 1/cl 位，最低位补 0
                                  ; 最高位进入 cf
        shr     reg/mem, 1/cl     ; 逻辑右移: reg/mem 右移 1/cl 位，最高位补 0，最低位进入 cf
        sal     reg/mem, 1/cl     ; 算术左移，功能与 shl 相同
        sar     reg/mem, 1/cl     ; 算术右移: reg/mem 右移 1/cl 位，最高位不变，最低位进入 cf
```

4 条（实际为 3 条）移位指令的目的操作数可以是寄存器或存储单元。后一个操作数表示移位位数，该操作数为 1，表示移动 1 位；当移位位数大于 1 时，则用 CL 寄存器值表示，该操作数表达为 CL。

移位指令按照移入的位设置进位标志 CF，根据移位后的结果影响 SF、ZF、PF，对 AF 没有定义。如果进行 1 位移动，则按照操作数的最高符号位是否改变，相应设置溢出标志 OF：如果移位前的操作数最高位与移位后操作数的最高位不同（有变化），则 OF=1，否则 OF=0。当移位次数大于 1 时，OF 不确定。

【例 2.33】 移位指令的功能。

```
mov     cl, 4
mov     al, 0f0h
; al=f0h
shl     al, 1
; al=e0h, cf=1, sf=1, zf=0, pf=0, of=0
shr     al, 1                ; al=70h, cf=0, sf=0, zf=0, pf=0, of=1
sar     al, 1                ; al=38h, cf=0, sf=0, zf=0, pf=0, of=0
sar     al, cl               ; al=03h, cf=1, sf=0, zf=0, pf=1
```

图 2-3 移位指令

逻辑左移和算术左移实际上是同一条指令的两种助记符形式，两者完全相同，建议采用 SHL。在指令系统中还有类似的情况。采用多个助记符只是为了方便使用，增加可读性。

逻辑左移指令 SHL 执行一次移位，相当于无符号数乘 2；逻辑右移指令 SHR 执行 1 位移位，相当于无符号数除以 2，商在目的操作数中，余数由 CF 标志反映。

算术右移指令 SAR 执行 1 位移位，相当于有符号数除以 2。注意，当操作数为负（最高位为 1）且最低位有 1 移出时，SAR 指令产生的结果与等效的 IDIV 指令的结果不同。例如，–5（FBH）经 SAR 右移 1 位等于–3（FDH），而 IDIV 指令执行–5÷2 的结果为–2。

【例 2.34】 利用移位指令计算 DX←3×AX+7×BX，假设为无符号数运算，无进位。

```
mov     si, ax
shl     si, 1                ; si←2×ax
add     si, ax               ; si←3×ax
mov     dx, bx
mov     cl, 03h
shl     dx, cl               ; dx←8×bx
sub     dx, bx               ; dx←7×bx
add     dx, si               ; dx←7×bx+3×ax
```

2.3.3 循环移位指令

循环（Rotate）移位指令类似移位指令，但要从一端移出的位返回到另一端形成循环，分为不带进位循环移位和带进位循环移位，分别具有左移或右移操作，如图 2-4 所示。

```
rol     reg/mem, 1/cl        ; 不带进位循环左移
ror     reg/mem, 1/cl        ; 不带进位循环右移
rcl     reg/mem, 1/cl        ; 带进位循环左移
rcr     reg/mem, 1/cl        ; 带进位循环右移
```

前两条指令不将进位 CF 纳入循环位中。后两条指令将进位标志 CF 纳入循环位中，与操作数一起构成的 9 位或 17 位二进制数一起移位。

循环移位指令的操作数形式与移位指令相同，如果仅移动 1 次，可以用 1 表示；如果移位多次，则需用 CL 寄存器表示移位次数。循环移位指令按照指令功能设置进位标志 CF，不影响

SF、ZF、PF、AF 标志。对 OF 标志的影响，循环移位指令与前面介绍的移位指令一样。

【例 2.35】 将 DX.AX 中的 32 位数值左移 1 位。

```
SHL     AX, 1
RCL     DX, 1
```

【例 2.36】 把 AL 最低位送 BL 最低位，但保持 AL 不变。

```
ROR     BL, 1
ROR     AL, 1
RCL     BL, 1
ROL     AL, 1
```

利用移位或循环移位指令可以方便地实现 BCD 码转换。

【例 2.37】 AH 和 AL 分别存放着非压缩 BCD 码的两位，将其合并成为一个压缩 BCD 码存入 AL。

```
MOV     CL, 4
ROL     AH, CL   ; 也可以用 SHL  AH, CL
ADD     AL, AH   ; 也可以用 OR   AL, AH
```

图 2-4　循环移位指令

2.4　控制转移类指令

在 Intel 8086 中，程序的执行序列是由代码段寄存器 CS 和指令指针 IP 确定的。CS 包含当前指令所在代码段的段地址，IP 则是要执行的下一条指令的偏移地址。程序的执行一般依指令序列顺序执行，但有时需要改变程序的流程。控制转移类指令通过修改 CS 和 IP 寄存器的值来改变程序的执行顺序，包括 5 组指令：无条件转移指令、有条件转移指令、循环指令、子程序指令和中断指令。本节介绍指令功能本身，第 4 章中介绍指令应用。

一条指令执行后，需要确定下一条执行的指令，也就是确定下条执行指令的地址，这被称为指令寻址。程序顺序执行，下一条指令在存储器中紧邻着前一条指令，指令指针寄存器 IP 自动增量，这就是指令的顺序寻址。程序转移则控制程序流程从当前指令跳转到目的地指令，实现程序分支、循环或调用等结构，这就是指令的跳转寻址。目的地指令所在的存储器地址称为目的地址、目标地址或转移地址，指令寻址实际上主要是指跳转寻址，也称为目标地址寻址。8086 处理器设计有相对、直接和间接 3 种指明目标地址的方式，其基本含义类似于对应的存储器数据寻址方式。图 2-5 汇总了各种寻址方式（含 1.6 节的数据寻址）。

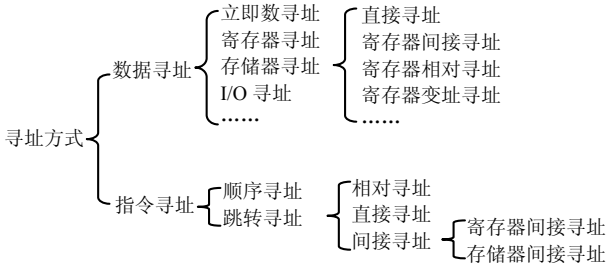

图 2-5　寻址方式