第3章 chapter 3

寻址方式与指令系统

【学习目标】

指令系统是指计算机所能执行的全部指令的集合，其描述了计算机内全部的控制信息和“逻辑判断”能力，每种CPU都有自己的指令系统。可以说，指令系统的功能强弱大体上决定了计算机硬件系统功能的强弱。本章的学习目标：掌握8086 CPU指令系统中主要指令的功能、指令格式及对操作数的要求，指令执行对标志位的影响；理解寻址方式的含义，熟悉具体的7种寻址方式。

3.1 指令系统概述

3.1.1 指令的基本概念

计算机通过执行程序来完成指定的任务，而程序是由完成一个完整任务的一系列有序指令组成的。指令是控制计算机完成指定操作并能够被计算机所识别的操作命令，每条指令都明确规定了计算机必须完成的一套操作以及对哪一组操作数进行操作。每种计算机都有一套能反映计算机全部功能的指令，这些所有指令的集合称为该机的指令系统。指令系统定义了计算机硬件所能完成的基本操作，其功能的强弱在一定程度上决定了硬件系统性能的高低；指令系统也是计算机硬件和软件之间的桥梁，是汇编语言程序设计的基础。不同的计算机（或者说不同的微处理器）具有各自不同的指令系统，但同一系列的计算机的指令系统是向前兼容的。

机器指令是一组用二进制编码的指令，是计算机能够直接识别和执行的指令。例如8086 CPU中的指令INC AX，其机器指令形式为01000000，十六进制形式为40H，指令功能是将寄存器AX的内容加1后再回送给AX。所有机器指令都是这样用0、1组成的二进制代码形式，不易理解，也不便于记忆和书写，因此，人们就采用便于记忆并能描述指令功能的符号来表示机器指令。这种用助记符或符号来表示操作码或操作数的指令就是汇编指令。汇编指令与机器指令间是一一对应的。

8086 CPU的指令系统是Intel系列CPU的基本指令集。从80286到Pentium系列的指令系统是在8086 CPU基本指令集基础上的增强与扩充。

3.1.2 指令格式

1. 指令的组成

计算机中的指令通常由操作码和操作数两部分构成，其中，操作码部分规定计算机要执行的操作，操作数部分也称为地址码，用于描述该指令要操作的对象。

一个指令字中包含二进制代码的位数，称为指令字长度。指令字长度等于机器字长的指令，称为单字长指令；指令字长度等于半个机器字长的指令，称为半字长指令；指令字长度等于两个机器字长的指令，称为双字长指令。Intel 8086 的指令采用变字长格式，指令由 1～6 字节组成，其中第 1 字节至少包含操作码，大多数指令的第 2 字节表示寻址方式，第 3～6 字节表示的是一个或两个操作数。

2. 8086 汇编指令格式

8086 汇编语言的每条指令最多由 4 部分组成，其格式如下：

```
[标号:] 操作码  操作数 [;注释]
```

其中，方括号标识的表示该部分为可选部分。

1）标号

标号表示一条指令在代码段中的偏移地址。在汇编源程序中，只有在需要转向一条指令时，才为该指令设置标号，以便在转移或循环指令中直接引用这个标号。标号和其后的操作码之间必须用冒号“:”分隔。

2）操作码

操作码也称指令码或助记符，用来规定计算机要执行的具体操作（如传送、运算、移位、跳转等），通常用一些意义相近的英文缩写（即助记符）来表示。操作码是所有指令中必不可少的部分。

3）操作数

操作数是指令执行过程中参与指令操作的对象，它的表现形式比较复杂，可以是操作数本身，也可以是操作数地址或是地址的一部分，还可以是指向操作数地址指针或其他有关操作数据的信息。根据指令的不同，在指令中可以不含有操作数或者隐含操作数，即无操作数；也可以只含有一个操作数，即单操作数；或者是含有两个操作数，即双操作数。当是双操作数时，两个操作数间必须用逗号“,”分隔，并且称逗号左边的操作数为目的操作数，逗号右边的操作数为源操作数。操作数与操作码之间必须以空格分隔。

4）注释

注释是对有关指令及程序功能的标注性说明，以增加程序的可读性；注释是用户根据自己的需要可选的，不影响程序的执行，但要注意注释与操作数之间需用分号“;”分隔。

说明：读者要正确理解指令格式中方括号“[]”所表示的可选内容的含义。其中，“注释”可选项完全决定于用户，是真正的可选；但对于“标号”部分的可选项则是出于指令的

需要，对于需要的指令就一定要写，对于不需要的指令就不必写了。

存储器操作数及实例操作演示

3.1.3 操作数类型

8086指令中的操作数按其存放的地方，可分为立即操作数、寄存器操作数和存储器操作数三种类型。

1. 立即操作数

立即操作数简称立即数，是指具有固定数值的操作数（即常数），它不因指令的执行而发生变化。8086指令中的立即数具体可以是一字节、一个字或双字，可以是无符号数或带符号数。在指令书写中，立即数作为指令代码的一部分出现在指令中，作为源操作数使用；书写形式可以是二进制、十进制或十六进制，也可以是一个可以求出确定值的表达式，但数的取值范围必须符合相应的机器字长数的规定，如果取值超出了规定的范围，就会发生错误。存放时，立即数跟随指令操作码一起被存放在代码区。

2. 寄存器操作数

事先存放在CPU的8个通用寄存器和4个段寄存器中的操作数为寄存器操作数，在指令执行时只要知道寄存器名就可以寻找到操作数。寄存器操作数既可以作为源操作数使用，也可以作为目的操作数使用。

3. 存储器操作数

事先存放在内存中的操作数为存储器操作数，在指令中，只要知道存放操作数的存储单元的偏移地址就可以寻找到该操作数。由于8086指令系统中的操作数一般均为8位或16位字长，因此存储器操作数通常以字节和字类型居多，分别存放在一个和连续两个存储单元中，极个别指令中用到双字类型操作数。在指令中，存储器操作数都给出了偏移地址，段地址一般以隐含方式给出。存储器操作数既可以作为源操作数使用，也可以作为目的操作数使用。

说明：存储器操作数的表示有两种形式。

（1）若用数字形式表示的某存储单元的物理地址为M，该地址中的内容为N，则有$(M)=N$，即用圆括号将地址括起来表示该地址的内容，这种形式通常在文字描述中引用。

（2）带方括号“[]”的操作数表示存储器操作数，这种形式通常在程序指令中引用。

3.1.4 指令的执行

程序中要执行的所有指令，均保存在存储器中，当计算机需要执行一条指令时，首先根据这条指令的地址，访问相应的存储单元，取出指令代码，CPU再根据指令代码的要求以及指令中的操作数，去执行相应的操作。

3.2 寻址方式

寻址方式主要是指获得操作数所在的地址的方法。8086 CPU 的寻址分为两类：一类是寻找操作数的地址；另一类是寻找要执行的下一条指令的地址，即指令地址。后者主要在程序转移或过程调用时用来寻找目标地址或入口地址，这将在 3.3.4 节的转移和过程调用类指令中介绍。本节讨论针对操作数地址的寻址方式，并且如无特殊声明，一般都指源操作数，下面介绍这种寻址方式。

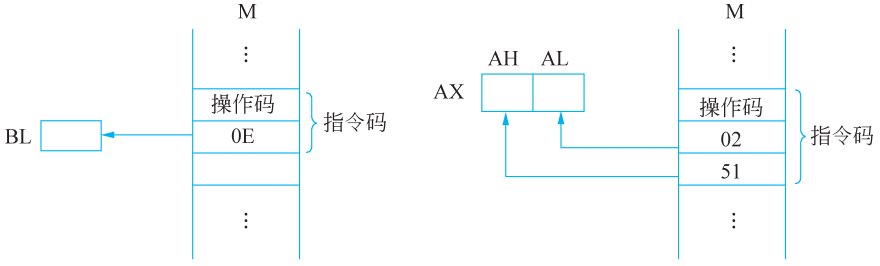
3.2.1 立即寻址

立即寻址方式所提供的操作数直接包含在指令中，此操作数紧跟在操作码后面，与操作码一起存放于内存的代码段中，在 CPU 取指令时，立即数随指令码一起取出并直接参与运算。立即寻址方式中的操作数只能用于源操作数，主要用来给寄存器或存储单元赋初值。

对于 16 位微型计算机系统，立即数可以是一个 8 位或 16 位的整数。若为 16 位，则存放时其低 8 位存放在相邻两个存储单元的低地址单元中，高 8 位存放在高地址单元中。例如：

```
MOV   BL,0EH            ;将 8 位立即数 0EH 传送到 BL 寄存器中
MOV   AX,5102H          ;将 16 位立即数 5102H 传送到 AX 寄存器中,AH=51H,AL=02H
MOV   DL,'6'            ;将字符'6'的 ASCII 码值 36H 送入寄存器 DL 中,'6'是立即数
```

机器执行立即寻址方式的指令过程如图 3.1 所示。



（a）MOV BL,0EH　　（b）MOV AX,5102H

图 3.1 立即寻址示意

3.2.2 寄存器寻址

寄存器寻址的操作数存放在 CPU 的某个寄存器中，在指令中写出指定的寄存器名即可。对于 8 位操作数，可用的寄存器是 AL、AH、BL、BH、CL、CH、DL 和 DH；对于 16 位的操作数，可使用 8 个通用寄存器 AX、BX、CX、DX、SI、DI、SP、BP 和 4 个段寄存器。例如：

```
MOV   AX, CX                ;将 CX 中的内容传送到 AX 中
```

若指令执行前 AX＝6545H，CX＝8932H，则指令执行后 AX＝8932H，而 CX 中的内

寻址方式动画演示（立即寻址与寄存器寻址）

容保持不变。

对于寄存器寻址方式，指令的操作码存放在代码段中，但操作数在CPU的寄存器中，执行指令时不必访问存储器就可获得操作数，故其执行速度较快。

寻址方式动画演示（5种存储器寻址）

3.2.3 存储器寻址

在存储器寻址方式中，操作数存放在内存的存储单元中，执行指令时，CPU要访问到操作数须先计算出存放该操作数的存储单元在内存中的物理地址，然后才能进行数据读/写的操作。

前已述及，8086 CPU对存储器采用分段管理，所以在指令中直接引用存储单元的物理地址较困难，而是直接或间接地给出存放操作数的偏移地址，并用方括号“[]”括起来，以达到访问操作数的目的，以下介绍具体的存储器寻址方式。

1. 直接寻址

在直接寻址方式中，指令中的操作数部分直接给出操作数的偏移地址，且该地址与操作码一起被放在代码段中。通常，直接寻址方式的操作数放在存储器的数据段中，这是一种默认方式，当然也可以使用段超越。例如：

```
MOV AX,[2000H]        ;将数据段中偏移地址为2000H和2001H的两单元的内容送到AX中
MOV AX,ES:[2000H]     ;将附加段中偏移地址为2000H和2001H的两单元的内容送到AX中
```

存储器操作数本身并不能表明数据的类型，需要通过另一个寄存器操作数的类型或别的方式来确定。因此对于以上两例，由于目的操作数AX为字类型，因此，作为存储器操作数的源操作数也应与之配套为字类型。假设DS＝3000H，则在执行MOV AX,[2000H]指令后，AX＝9310H。指令的寻址及执行过程如图3.2所示。

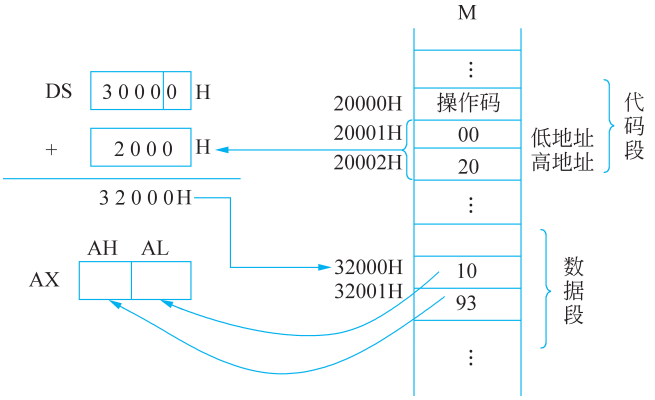


图3.2 MOV AX,[2000H]指令的寻址及执行过程

说明：在汇编语言源程序中，直接寻址方式不以在方括号内括一个16位常数这样的形式表示，常用变量名表示偏移地址。如上例中若用符号BUF代替地址2000H，则MOV AX,[2000H]指令可写成MOV AX,BUF。因为在源程序中若以MOV AX,[2000H]形式寻址，则汇编时，汇编程序会将源操作数[2000H]当成立即数汇编，即将该指

令汇编成 MOV AX,2000H,这样就变成了立即寻址,而不是直接寻址了,读者不妨试一试。

指令 MOV AX,BUF 中的 BUF 为存放操作数的存储单元的偏移地址,须在汇编源程序的开始处予以定义,该内容将在 4.2.2 节中介绍。

2. 寄存器间接寻址

在寄存器间接寻址方式中,操作数的偏移地址在指令指明的寄存器中,即寄存器的内容就是操作数的偏移地址,而操作数存放在存储器中。

对于寄存器间接寻址,存放操作数偏移地址的寄存器只能是 BX、BP、SI 和 DI,选择不同的寄存器涉及的段寄存器不同。指令中如果指定的寄存器是 BX、SI、DI,则操作数默认在数据段中,段地址由 DS 提供;如果指定的寄存器是 BP,则操作数默认在堆栈段中,段地址由 SS 提供,但允许段超越。

【例 3.1】 已知 DS=3000H,BX=1000H,(31000H)=56H,(31001H)=34H,试分析指令 MOV AX,[BX]的寻址情况。

由已知条件可计算出源操作数的物理地址=30000H+1000H=31000H,执行指令 MOV AX,[BX]操作的示意如图 3.3 所示。

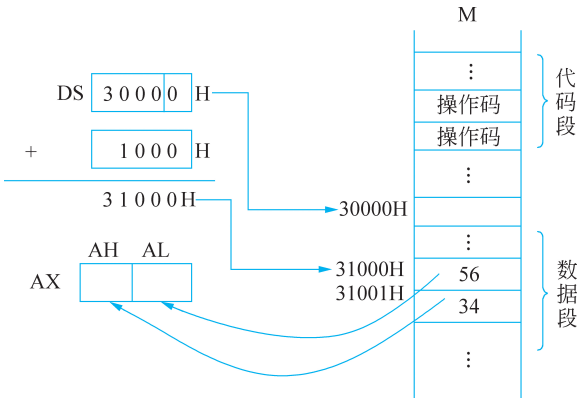


图 3.3 MOV AX,[BX]指令的寻址及执行结果

指令执行结果为 AX=3456H。

【例 3.2】 若已知 SS=8000H,BP=0400H,(80400H)=3AH,(80401H)=59H,则执行指令 MOV BX,[BP]后,结果为 BX=593AH。

寄存器间接寻址允许在指令中通过段超越前缀来存取其他段中的操作数。例如:

```
MOV  AX,ES:[BX]        ;将以 ES 的内容为段地址,以 BX 的内容为偏移地址的附加段中的
                       ;相应字内容送给 AX
```

3. 寄存器相对寻址

在寄存器相对寻址方式中,操作数的偏移地址由一个基址或变址寄存器与指令中指定的 8 位或 16 位位移量形成,即

$$EA=\begin{pmatrix}BX\\BP\\SI\\DI\end{pmatrix}+\begin{pmatrix}8\text{ 位}\\16\text{ 位}\end{pmatrix}\text{位移量}$$

对于寄存器相对寻址，段寄存器的引用规则与寄存器间接寻址方式相同，即指令中如果指定的寄存器是 BX、SI 或 DI，段地址默认由 DS 提供；如果指定的寄存器是 BP，段地址默认由 SS 提供，但允许段超越。例如：

```
MOV  SS: STR[SI], AX          ;SS 是段超越前缀，即 SS 为当前段寄存器
```

在汇编语言程序设计中，寄存器相对寻址的书写形式较灵活，允许有不同的书写形式，以下是与 MOV AX,COUNT [BX]指令等价的另外几种书写形式。

```
MOV  AX,[BX+COUNT]
MOV  AX,[COUNT+BX]
MOV  AX,[ BX] COUNT
```

【例 3.3】 假设 DS＝3000H，BX＝1000H，COUNT＝4000H，(35000H)＝90H，(35001H)＝19H，则在执行 MOV AX,COUNT [BX]指令后，AX＝1990H。该指令的寻址及执行结果如图 3.4 所示。

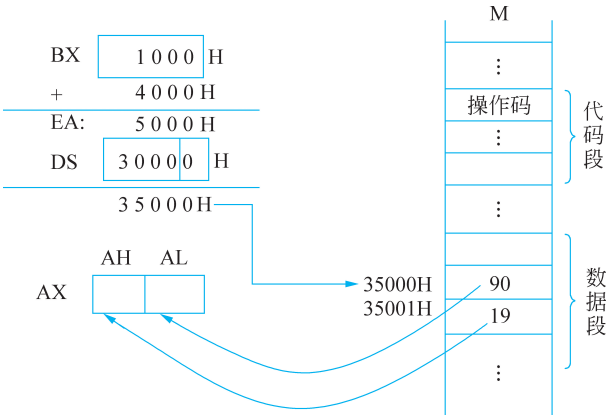


图 3.4 MOV AX,COUNT[BX]指令的寻址及执行结果

4. 基址变址寻址

在基址变址寻址方式中，操作数的偏移地址由一个基址寄存器(BX 或 BP)和一个变址寄存器(SI 或 DI)的内容相加而成，两个寄存器均由指令指出，即

$$EA=BX+\begin{pmatrix}SI\\DI\end{pmatrix}\quad\text{或}\quad EA=BP+\begin{pmatrix}SI\\DI\end{pmatrix}$$

基址变址寻址的操作数段地址引用随基址寄存器的不同而不同，指令中如果指定的基址寄存器是 BX，段地址默认由 DS 提供，如果指定的基址寄存器是 BP，则段地址默认由 SS 提供，但允许段超越，操作数的书写形式也有多种。

【例 3.4】 请分析指令 MOV AX,[BX][DI]的执行情况。设 DS＝2000H，BX＝

8000H，DI＝1000H，物理地址(29000H)的字单元内容为 5678H。

分析：对于指令 MOV AX,[BX][DI]，源操作数的偏移地址为 BX＋DI＝9000H，与段地址 DS＝2000H 形成的物理地址为 29000H，执行该指令，将(29000H)字单元中的内容送给 AX。因此，AX＝5678H。

说明：

(1) 与寄存器相对寻址一样，基址变址寻址的操作数书写形式较为灵活。对于例 3.4 中的 MOV AX,[BX][DI]指令，也可以写成 MOV AX,[BX＋DI]或 MOV AX,[DI][BX]等形式。

(2) 在实际编程应用中，基址变址寻址适用于对数组和表格的处理，可将首地址放在基址寄存器中，而用变址寄存器来访问数组或表格中的各个元素。

5. 基址变址相对寻址

在基址变址相对寻址方式中，操作数的偏移地址由一个基址寄存器、一个变址寄存器，同时还有一个由指令中指定的 8 位或 16 位位移量三者内容之和形成，即

$$EA=\begin{pmatrix}BX\\BP\end{pmatrix}+\begin{pmatrix}SI\\DI\end{pmatrix}+\begin{pmatrix}8\text{ 位}\\16\text{ 位}\end{pmatrix}\text{位移量}$$

基址变址相对寻址方式对操作数段地址的引用与基址变址寻址方式相同，即 BX 默认的段寄存器为 DS，BP 默认的段寄存器为 SS，允许段超越。基址变址相对寻址方式中操作数的书写形式也可采用多种等价形式。例如以下指令：

```
MOV  DX,disp [BX][SI]
MOV  DX,disp [BX+SI]
MOV  DX,[BX+SI+disp]
MOV  DX,[BX+SI]disp
MOV  DX,[BX] disp [SI]
```

这 5 种不同形式指令的寻址含义相同，都是将由 DS 段寄存器与偏移地址 BX＋SI＋disp 所形成的物理地址所对应的字存储单元中的内容传送到 DX 寄存器中。

【例 3.5】 设 DS＝3000H，BX＝2000H，SI＝1000H，位移量 disp＝0250H，物理地址＝DS×16＋BX＋SI＋disp＝30000H＋2000H＋1000H＋0250H＝33250H 的字单元内容为 3132H，则在执行指令 MOV DX,disp [BX][SI]后，DX＝3132H，该指令的寻址及执行过程如图 3.5 所示。

3.3 8086 CPU 指令系统

Intel 的 8086 CPU 指令系统是 80x86/Pentium 的基本指令集，提供了共 133 条指令，按功能可分为数据传送类、算术运算类、逻辑运算与移位类、串操作类、控制转移类和处理器控制类六大类。本节介绍部分常用指令。学习指令，需掌握指令的书写格式、指令功能、寻址方式、对操作数的规定以及指令对标志位的影响等，这是编写汇编程序的关键。

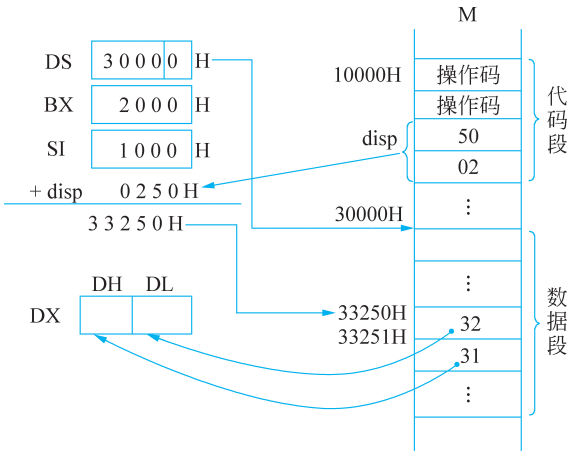


图 3.5　MOV DX,disp[BX][SI]指令的寻址及执行过程

汇编语言对指令的书写不区分大小写字母，本节在介绍具体的指令之前，先介绍本教材中要用到的一些符号所表示的含义。

mem：存储器操作数。

opr：泛指各种类型的操作数。

src：源操作数。

dest：目的操作数。

label：标号。

disp：8 位或 16 位位移量。

port：输入/输出端口，可用数字或表达式表示。

[ ]：整体上是一存储器操作数。

reg：通用寄存器。

count：移位次数，可以是 1 或 CL。

S_ins：串操作指令。

3.3.1　数据传送类指令

传送数据是计算机中量最大、最基本、最主要的操作，所以数据传送类指令也是实际程序中使用最多的指令，是指令系统中提供的传送种类和条数最多的一类指令，常用于将原始数据、中间运算结果、最终结果及其他信息在 CPU 的寄存器和存储器之间进行传送。根据所执行的功能不同，数据传送类指令分为通用数据传送指令、交换指令、堆栈操作指令、地址传送指令、标志寄存器传送指令和查表转换指令 6 种。

1. 通用数据传送指令 MOV

指令格式：

```
MOV dest,src
```

功能：将源操作数 src 的内容传送给目的操作数 dest，源操作数内容不变。

MOV 指令的主要数据传送方式如图 3.6 所示。

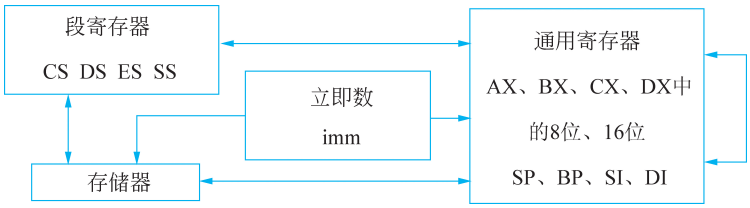


图 3.6 数据传送方式示意

图 3.6 中表示的是 MOV 指令能实现的其中大多数的数据传送方式，但也存在 MOV 指令不能实现的传送情况。以下为一些具体的规定，其中有些规定对后面其他的一些指令同样有效。

(1) 两个操作数的数据类型须匹配(即位数要一致，需同为字节类型或字类型数据)，两个操作数不能同时为段寄存器；也不能同时为存储器操作数。

(2) 操作数不能出现二义性，即至少一个操作数需明确类型。

(3) 代码段寄存器 CS 不能作为目的操作数使用，但可以作为源操作数。

(4) 立即数不能作为目的操作数使用，也不能直接传送给段寄存器。

(5) 指令指针寄存器 IP 既不能作为目的操作数使用，也不能作为源操作数使用。

下面列举几组正确的指令例子。

(1) 源操作数是寄存器操作数：

```
MOV  DL,AL          MOV  BP,SP          MOV  CX,BX
MOV  DS,BX          MOV  [BX],AH        MOV  [BX+SI],DX
```

(2) 源操作数是存储器操作数(其中，VARW 是字类型内存变量，以下同)：

```
MOV  AL,[2000H]     MOV  AX,ES:[DI]     MOV  AX,[BX]
MOV  AX,VARW        MOV  DX,[BX+SI]     MOV  AH,[BX+DI+100H]
```

(3) 源操作数是立即数：

```
MOV  AL,99H         MOV  BX,-100        MOV  WORD PTR [BX],300
```

下面是一些错误指令的例子(其中，VARA 和 VARB 为变量)：

```
MOV  BL,AX              ;操作数类型不匹配
MOV  ES,DS              ;两个操作数都为段寄存器
MOV  CS,AX              ;CS 不能为目的操作数
MOV  DS,100H            ;立即数不能直接传送给段寄存器
MOV  100H,AX            ;立即数不能作为目的操作数
MOV  VARA,VARB          ;VARA 和 VARB 都属于存储器操作数，不能直接进行
MOV  [BX],12H           ;两个操作数的类型都不明确
```

说明：立即数的类型是不明确的，不能把 16 位二进制数当作字类型的立即数，也不能把 8 位二进制数当作字节类型的立即数。在将立即数传送给存储器时，若存储器操作数的类型不明确，则需使用操作符 PTR 来明确其类型。BYTE 规定为字节，WORD 规定

为字。因此，对于指令 MOV [BX],12H，若将其修改为 MOV BYTE PTR [BX],12H 或 MOV WORD PTR [BX],12H 就是正确的指令了。有关 PTR 内容将在 4.1.4 节介绍。

MOV 指令不影响标志位。

2. 交换指令 XCHG

指令格式：

```
XCHG dest,src
```

功能：将源操作数和目的操作数的内容相互交换。

说明：指令中的两个操作数类型须匹配；交换只能在两个通用寄存器之间，通用寄存器与存储器操作数之间进行，不能同时为两个存储器操作数。例如：

```
XCHG  AL,BL        ;AL 和 BL 的内容相互交换
XCHG  SI,DI        ;SI 和 DI 的内容相互交换
XCHG  CX,[3450H]   ;CX 中的内容和由 DS:3450H 所形成的物理地址字单元中的内容进行交换
```

XCHG 指令不影响标志位。

3. 堆栈操作指令 PUSH 和 POP

指令格式：

```
PUSH src
```

功能：将指令中指定的 16 位操作数 src 压入堆栈，该操作数可以是通用寄存器操作数、存储器操作数或者是段寄存器操作数。

PUSH 指令的执行过程：

$$SP \leftarrow SP-2$$

$$(SP+1,SP) \leftarrow src$$

即先修正堆栈指针 SP 的值，然后将 16 位源操作数压入堆栈，压入的顺序是先高字节后低字节。

【例 3.6】 若给定 SP=00F8H，SS=4000H，AX=5102H，则在执行指令 PUSH AX 后，SP=00F6H，(400F6H)=5102H。该指令执行前后的堆栈变化示意如图 3.7 所示。

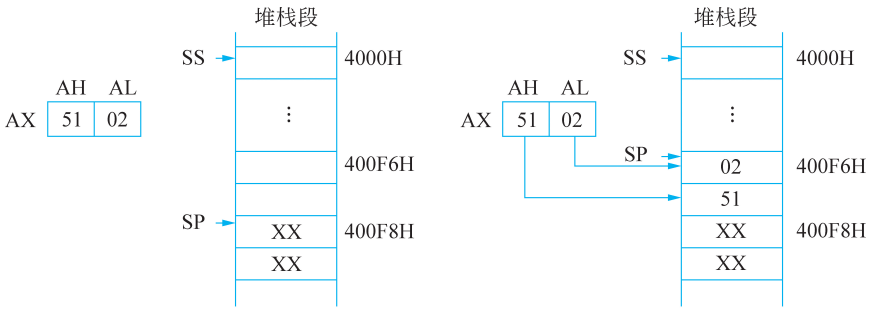


图 3.7 PUSH AX 指令执行前后的堆栈变化

执行 PUSH 指令，先将 SP 的值减 2 变为 00F6H，然后再将 AX 的内容 5102H 送入由 SS 和 SP 所形成的物理地址 400F6H 字单元中。

指令格式：

```
POP  dest
```

功能：将堆栈栈顶指针 SP 所指的 16 位字内容弹出给指令中指定的操作数。同 PUSH 指令一样，POP 指令中的操作数可以是一通用寄存器、一存储器或段寄存器操作数(CS 除外)。

POP 指令具体执行的操作：

$$dest \leftarrow (SP+1,SP)$$

$$SP \leftarrow SP+2$$

即先从栈顶弹出 16 位操作数给目的操作数，然后再修正堆栈指针 SP，使 SP 指向新的栈顶。

【例 3.7】 若给定 SP＝0100H，SS＝2000H，BX＝78C2H，(20100H)＝6B48H，则在执行指令 POP BX 后，SP＝0102H，BX＝6B48H。该指令执行前后堆栈变化示意如图 3.8 所示。

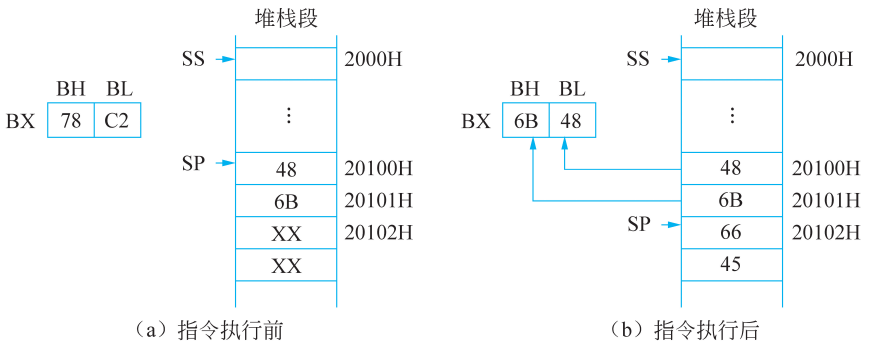


图 3.8 POP BX 指令执行前后的堆栈及 BX 变化情况

POP 指令执行时，首先将 SP＝0100H 所指的栈顶地址单元中的内容 6B48H 弹出来送入 BX 中，然后再将 SP 加 2 指向 0102H 单元，SP 在原来 0100H 的基础上增加了 2。

在程序设计中，堆栈是一种十分有用的结构，其经常用于子程序的调用与返回、中断处理过程中的断点地址保存以及暂时保存程序中的某些信息等。

8086 堆栈的使用规则如下。

(1) 堆栈的使用要遵循后进先出(LIFO)的准则。

(2) 堆栈的存取每次必须是一个字(16 位)，即堆栈指令中的操作数必须是 16 位。

(3) 8086 CPU 堆栈操作可以使用除立即寻址以外的任何寻址方式。

以上两条堆栈指令都不影响标志位。

4. 地址传送指令

地址传送指令传送的是存储器操作数的地址(偏移地址和段地址)，而不是内容。这

组指令都不影响标志位，指令中的源操作数都必须是存储器操作数。

1）取有效地址指令 LEA

指令格式：

```
LEA reg,mem
```

功能：将源操作数的有效地址送到指令中指定的寄存器中。源操作数只能是一存储器操作数，目的操作数只能是一 16 位的通用寄存器。例如：

```
LEA  BP,[3456H]          ;将偏移地址 3456H 送入 BP
LEA  BX,BUF              ;将变量 BUF 所指的存储单元的偏移地址送给 BX
```

【例 3.8】 设 BX＝1000H，DS＝6000H，[61032H]＝33H，[61033H]＝44H，试比较以下两条指令单独执行时的结果。

```
LEA  BX,50[BX]
MOV  BX,50[BX]
```

分析：执行第一条指令后，BX＝1032H；执行第二条指令后，BX＝4433H。

要特别注意 LEA 指令和 MOV 指令间的区别。从以上例子也可以看出：指令 LEA BX,50[BX]是将 BX 的内容 1000H 与 50 之和 1032H 作为存储器的有效偏移地址送入 BX 中；而指令 MOV BX,50[BX]则是将源操作数通过寄存器相对寻址方式所确定的物理地址 61032H 存储单元中的字内容送入 BX 中。

实际编程时，常通过 LEA 指令以使一个寄存器作为地址指针用，这个寄存器通常为 BX、BP、SI、DI。

2）地址指针装到 DS 和指定的寄存器指令 LDS

指令格式：

```
LDS  reg ,mem
```

功能：从指令的源操作数所指定的存储单元开始，将连续 4 字节存储单元中的第一个字内容送入指令中指定的通用寄存器中，而第二个字内容送入段寄存器 DS 中。

【例 3.9】 假设当前 DS＝3000H，存储单元(30100H)＝80H，(30101H)＝20H，(30102H)＝00H，(30103H)＝25H。则在执行指令 LDS SI，[0100H]后，SI＝2080H，DS＝2500H。

3）地址指针装到 ES 和指定的寄存器指令 LES

LES 指令的操作与 LDS 指令基本类似，所不同的只是第二个字内容送往的段寄存器是 ES，即将源操作数所指定的地址指针中的后 2 字节内容传送到 ES 段寄存器中。

地址传送类指令应用于串操作时，需建立初始的串地址指针。

5. 标志寄存器传送指令

与标志寄存器有关的传送指令共 4 条，它们都是无操作数指令，但操作数规定为隐含方式。利用这些指令，可以读出标志寄存器的内容，也可以对标志寄存器的某些标志位设置新值。

1）读取标志指令 LAHF

指令格式：

```
LAHF
```

功能：将标志寄存器的低 8 位读出后传送给 AH 寄存器。

LAHF 指令的操作情况如图 3.9 所示。

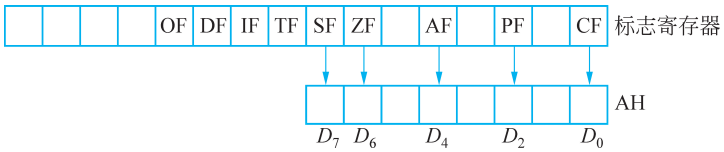


图 3.9 LAHF 指令的操作情况

2）设置标志寄存器指令 SAHF

指令格式：

```
SAHF
```

SAHF 指令与 LAHF 指令正好相反，它将 AH 寄存器内容送到标志寄存器的低 8 位。实际编程时，常用该指令修改某些状态标志位的值。

【例 3.10】 编写程序段，实现将标志寄存器中的 SF 标志位置为“1”。

程序段如下：

```
LAHF                          ;标志寄存器的低 8 位送到 AH
OR   AH,80H                   ;使用逻辑"或"指令将 SF 置为"1"
SAHF                          ;AH 的内容返回到标志寄存器
```

3）标志寄存器压栈指令 PUSHF

指令格式：

```
PUSHF
```

功能：先执行栈顶指针 SP 减 2 操作，然后再将 16 位标志寄存器的所有标志位送入到 SP 指向的栈顶字单元中，其操作过程与 PUSH 指令类似。

4）标志寄存器出栈指令 POPF

指令格式：

```
POPF
```

功能：POPF 指令与 PUSHF 指令刚好相反，执行时将堆栈栈顶指针所指的一个字内容弹出来送到标志寄存器中，然后再将 SP 的值加 2，其操作过程与 POP 指令类似。

实际编程时，常利用 PUSHF 和 POPF 指令保护和恢复标志位，即用 PUSHF 指令保护调用子程序之前的标志寄存器值，而在子程序执行之后，再利用 POPF 指令恢复这些标志位状态；利用 PUSHF 和 POPF 指令，也可以方便地改变标志寄存器中任一标志位的值。

【例 3.11】 编写程序段，将 TF 标志位置为“1”。

程序段如下：

```
PUSHF                    ;将当前标志寄存器的内容压入堆栈
POP  AX                  ;标志寄存器的内容弹出至 AX
OR  AH,01H               ;将 TF 位置为"1"
PUSH AX
POPF                     ;AX 的内容送至标志寄存器
```

以上指令中，LAHF 和 PUSHF 对标志位无影响，SAHF 和 POPF 会影响相应标志位。

6. 查表转换指令 XLAT

指令格式：

```
XLAT 或 XLAT  表首址
```

功能：实现将 AL 中的值变换成内存表格中的对应值。XLAT 指令的操作数是隐含的，所执行的操作是将 BX 中的值为基地址、AL 中的值为位移量所形成的偏移地址所对应的字节存储单元中的内容送到 AL 中，即执行 AL←[BX+AL]操作。XLAT 指令的功能可以用如下程序段代替。

```
ADD  BL,AL
ADC  BH,0                ;代替的条件是该指令不再产生进位
MOV  AL,[BX]
```

实际编程时，常使用该指令实现代码转换（将一种代码转换为另一种代码）或查表的功能。方法是：首先在数据段中建立一个长度小于 256 字节的数据表，将该表的首地址存放在 BX 中，将欲查找对象所在表中的地址下标值（数据表内位移量）存放在 AL 中，最后运用 XLAT 指令即可将该地址处的值送到 AL 中。

【例 3.12】 编程实现将十六进制数 0H～FH 的某一数转换为其字符对应的 ASCII 码值，假设存放 ASCII 码值的数据表首地址为 2000H，如图 3.10 所示。

以下是要取出十六进制数 B 所对应的 ASCII 码值的程序段。

```
MOV  BX,2000H      ;将 ASCII 码表首地址置入 BX 中，
                   ;BX=2000H
MOV  AL,0BH        ;将待转换的数据 0BH 在表中的位
                   ;移量送入 AL 中
XLAT               ;完成代码转换，AL=42H
```

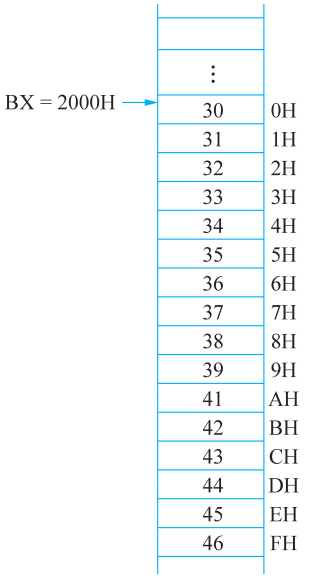


图 3.10 0H～FH 的十六进制数对应的 ASCII 码值

3.3.2 算术运算类指令

8086 CPU 的算术运算类指令包括针对二进制数运算的加、减、乘、除指令和十进制数算术运算调整指令，参加运算的操作数可以是字节、字，可以是带符号数或无符

号数。对本类指令的学习，除了掌握指令的格式及操作功能外，还需要掌握指令对标志位的影响，因为这类指令的执行结果大多要影响标志位。

1. 加法指令

1）加法指令 ADD

指令格式：

```
ADD dest,src
```

功能：将源操作数的内容和目的操作数的内容相加后的结果保存在目的操作数中，并根据结果设置相关标志位。

注意：

（1）源操作数和目的操作数应同时为带符号数或无符号数，且两者的数据类型应匹配。

（2）源操作数可以是 8/16 位的通用寄存器操作数、存储器操作数或立即数；目的操作数只能是与源操作数相匹配的通用寄存器或存储器操作数，且两者不能同时为存储器操作数。

以下是一些合法的加法指令例子：

```
ADD  AL,50H          ;AL ←AL+50H
ADD  BX,AX           ;BX ←BX+AX
ADD  AX,[BX+3000H]   ;DS:[BX+3000H]所指的字单元内容与 AX 内容相加，结果送入 AX 中
```

【例 3.13】 设指令执行前，AL＝66H，BL＝20H，执行指令 ADD AL，BL 之后，则 AL＝86H，BL 仍为 20H。

$$\begin{array}{rll} & 0110\ 0110 & \text{AL} \\ +) & 0010\ 0000 & \text{BL} \\ \hline & 1000\ 0110 & \text{AL} \end{array}$$

标志位影响情况为 CF＝0，ZF＝0，SF＝1，AF＝0，OF＝1，PF＝0。

2）带进位的加法指令 ADC

指令格式：

```
ADC dest,src
```

功能：ADC 指令的操作功能与 ADD 指令基本相同，唯一不同的是还要加上当前进位标志位的值。实际编程时，常使用该指令实现多字节/字的加法运算。由于 8086 一次最多只能实现两个 16 位的数据相加，因此对于多于 2 字节/字的加法，只能先加低 8 位（或低 16 位），再加高 8 位（或高 16 位），但在进行高部分内容相加时，需要考虑低部分的进位位，这时就需要使用 ADC 指令。

【例 3.14】 两个无符号双精度数的加法例子，求 0002F365H＋0005E024H＝？

假设目的操作数存放在 DX 和 AX 寄存器中，其中，DX＝0002H 存放高字部分，AX＝F365H存放低字部分；源操作数存放在 BX 和 CX 寄存器中，其中，BX＝0005H 存放高字部分，CX＝E024H 存放低字部分。

完成以上操作的双字加法指令序列：

```
ADD  AX, CX              ;低字内容相加
ADC  DX, BX              ;高字内容带进位相加
```

执行第一条指令的相加运算

```
                1111 0011 0110 0101    AX
             +) 1110 0000 0010 0100    CX
CF←1            1101 0011 1000 1001    AX
```

之后，AX=D389H，CF=1，AF=0，PF=0。

执行第二条指令的相加运算

```
        0000 0000 0000 0010    DX
        0000 0000 0000 0101    BX
   +)                    1    CF
        0000 0000 0000 1000    DX
```

之后，DX=0008H，CF=0，ZF=0，SF=0，AF=0，OF=0，PF=0。

该指令序列执行完后，相加的32位和存放在DX与AX中，DX=0008H，AX=D389H，即相加的结果为0008D389H。

3）加1指令INC

指令格式：

```
INC dest
```

功能：将指定操作数的内容加1后，再回送给该操作数。

实际上，指令中的操作数既是目的操作数，同时又是源操作数，其可以是任意一个8/16位的通用寄存器或存储器操作数，但不能是立即数。指令执行的结果将影响PF、AF、ZF、SF和OF，但不影响CF标志位。

实际编程时，INC指令常用于循环程序中对循环计数器的计数值或修改地址指针之用。例如，执行INC CX指令后，将CX寄存器中的内容加1后又回送到CX中。

2. 减法指令

1）减法指令SUB

指令格式：

```
SUB dest,src
```

功能：将目的操作数内容与源操作数内容相减之后的结果（差）存入目的操作数中。该指令对操作数的要求以及对标志位的影响与ADD指令完全相同。

【例3.15】 设SS=5000H，BP=2000H，AL=45H，存储单元(52008H)=87H。试分析指令SUB AL，[BP+8]执行后的情况。

执行指令SUB AL，[BP+8]，实际上执行的就是以下操作：

```
        0100 0101    AL
     -)1000 0111    (52008H)
  ------------------------------
  CF←1 1011 1110    AL
```

指令执行后：AL＝BEH，(52008H)单元内容仍为 87H。标志位的影响情况为 CF＝1，ZF＝0，SF＝1，AF＝1，OF＝1，PF＝1。

2）带借位的减法指令 SBB

指令格式：

```
SBB dest,src
```

功能：指令执行时，用目的操作数减去源操作数，还要减去标志位 CF 的值，并将相减的结果回送给目的操作数。SBB 指令对操作数的要求以及对标志位的影响与 SUB 指令完全相同。与 ADC 指令类似，在实际编程中，SBB 指令主要用于两个多字节/字二进制数的相减运算。

【例 3.16】 设指令执行前，DX＝0012H，AX＝7546H，CX＝0010H，BX＝9428H。编写程序段，完成 00127546H－00109428H 的运算。

双字减法指令段为

```
SUB  AX, BX          ;低字部分相减
SBB  DX, CX          ;高字带借位 CF 部分相减
```

第一条指令执行相减运算

```
          0111 0101 0100 0110   AX
     -)   1001 0100 0010 1000   BX
  ------------------------------
  CF←1    1110 0001 0001 1110   AX
```

之后，AX＝E11EH，CF＝1，AF＝1，PF＝1。

第二条指令执行相减运算

```
        0000 0000 0001 0010   DX
        0000 0000 0001 0000   CX
  -)                     1   CF
  ------------------------------
        0000 0000 0000 0001   DX
```

之后，DX＝0001H，CF＝0，ZF＝0，SF＝0，AF＝0，OF＝0，PF＝0。

该指令序列执行完后，相减的 32 位差值存放在 DX 与 AX 中，DX＝0001H，AX＝E11EH，即相减的结果为 0001E11EH。

3）减 1 指令 DEC

指令格式：

```
DEC dest
```

功能：完成对指令中指定操作数的内容减 1 后，又回送给该操作数。该指令对操作数的要求及对标志位的影响与 INC 指令完全相同，实际使用场合也与 INC 指令一样，通常用于在循环过程中对地址指针和循环次数的修改。

4）求补指令 NEG

指令格式：

NEG 指令及其操作与动画演示

```
NEG dest
```

功能：对指令中指定的操作数内容求补后，再将结果回送给该操作数。该操作数只能是通用寄存器操作数或存储器操作数。

对一个操作数求补实际上就相当于用 0 减去该操作数的内容，故 NEG 指令属于减法指令。该指令的间接求法是将操作数的内容按位求反末位加 1 后，再回送给该操作数。为什么可以这样？请读者思考。

该指令执行的效果是改变了操作数的符号，即将正数变为负数或将负数变为正数，但绝对值不变。由于带符号数在机器中是用补码表示的，因此，对于一个负数的操作数进行求补，实际上就是求其绝对值。例如：

设 BX 所指的字节单元内容为－4（补码为 1111 1100 B），则在执行 NEG BYTE PTR[BX] 后，其值变成了 0000 0100＝＋4（－4 的绝对值）。

从以上例子也可以看到，对一个负数的补码进行求补，得到的是该负数的绝对值。因此在实际编程时，常用 NEG 指令求负数的绝对值。

NEG 指令影响所有标志位。

5）比较指令 CMP

指令格式：

```
CMP dest,src
```

功能：CMP 指令除了不回送相减结果外，其他的均与 SUB 指令相同。例如：

设指令 CMP AL,CL 在执行之前，AL＝68H，CL＝9AH，该指令执行操作

$$\begin{array}{rll} & 0110\ 1000 & \text{AL} \\ -) & 1001\ 1010 & \text{CL} \\ \hline \text{CF}\leftarrow\underline{1} & 1100\ 1110 & \end{array}$$

之后，AL＝68H，CL＝9AH，CF＝1，ZF＝0，SF＝1，AF＝1，OF＝1，PF＝0。

在该例中，当把两个操作数作为无符号数比较时，被减数小于减数，不够减，有借位，CF＝1；当把两个操作数作为带符号数比较时，相减结果超出了带符号数所能表示的范围，因此 OF＝1，有溢出。

实际编程时，常利用 CMP 指令判断两数的大小或是否相等，在该指令后跟一条条件转移指令，根据比较的结果实现程序的分支。

【例 3.17】 在数据段的 BUF 存储单元开始处分别存放了两个 8 位无符号数，试比较它们的大小，并将较大者送到 MAX 单元。程序段如下：

```
      ⋮
LEA   BX,BUF          ;BUF 偏移地址送 BX,设置地址指针
MOV   AL,[BX]         ;第一个无符号数送 AL
INC   BX              ;BX 指针指向第二个无符号数
CMP   AL,[BX]         ;比较两个数
```

```
        JNC   NEXT                ;第一个数大于第二个数,即 CF=0,则转向 NEXT 处
        MOV   AL,[BX]             ;否则,第二个无符号数送至 AL(中间寄存器)
NEXT:   MOV MAX,AL                ;较大的无符号数送至 MAX 单元
            ⋮
```

3. 乘法指令

乘法指令分为无符号数乘法指令和带符号数乘法指令两种,它们唯一的区别是相乘的两个操作数是带符号数还是无符号数。

乘法指令的被乘数是隐含操作数,乘数需在指令中显式地写出来。执行指令时,CPU 会根据乘数是 8 位还是 16 位来自动选用被乘数是 AL 还是 AX。

1) 无符号数乘法指令 MUL

指令格式:

```
MUL opr
```

功能:将指令中指定的操作数与隐含的被乘数(都为无符号数)相乘,所得的乘积按表 3.1 中的对应关系存放。

表 3.1 乘法指令中乘数、被乘数与乘积的对应关系

乘数位数	隐含的被乘数	乘积的存放位置	举例
8 位	AL	AX 中	MUL BL
16 位	AX	DX 与 AX 中	MUL BX

MUL 指令对标志位 CF、OF 有影响,而对 SF、ZF、AF、PF 无定义。如果运算结果的高一半(AH 或 DX)为零,则 CF=OF=0;否则 CF=OF=1。

说明:

(1) 指令中的操作数可以使用除立即数以外的其他寻址方式,但当是寄存器寻址时,操作数只能是通用寄存器。

(2) 对标志位的"无定义"和"不影响"不同。无定义是指指令执行后,标志位的状态不确定;而不影响是指指令的结果不影响标志位,因而标志位应保持原状态不变。

2) 带符号数乘法指令 IMUL

该指令的格式和功能与 MUL 相同,只是要求两个操作数都须为带符号数。IMUL 指令对标志位的影响为:若乘积的高半部分是低半部分的符号位扩展,则 OF=CF=0;否则 OF=CF=1。

说明:IMUL 指令中对操作数寻址方式的规定同 MUL 指令,但表示形式为补码,乘积也是以补码形式表示的数。

【例 3.18】 MUL 指令和 IMUL 指令的乘法例子。

将以下指令中的立即数看作无符号数实现相乘。

```
MOV   AL,0B4H     ;AL=B4H=180
MOV   BL,11H      ;BL=11H=17
MUL   BL          ;AX=0BF4H=3060,高 8 位 0BH 不为 0,OF=CF=1
```