第 3 章　汇编语言基础

汇编语言使用指令进行编程，是一种面向机器的低级语言。

指令是对计算机硬件发出的指示、命令。例如，下面的一组二进制代码是一条指令：

```
10001011 11000011
```

它要求 8086 微处理器把 BX 寄存器的内容传送到 AX 寄存器中，其中左面的 8 位二进制代码 10001011 表示本条指令要进行的操作：在两个 16 位寄存器之间进行数据传送，称为操作码。后面的一个字节指出两个用于传送数据的寄存器，称为操作数。

用二进制代码表示的指令可以由 CPU 直接执行，称为机器指令。用机器指令编写程序的规范称为机器语言，编写出来的程序称为机器语言程序。

机器指令难以记忆，容易出错，不便于阅读和维护，为此，用一些符号来表示上面的操作码和操作数。例如，用 MOV 表示进行数据传送的操作码，用 AX、BX 表示各自的寄存器。这样，上面的指令可以另写为

```
MOV    AX, BX
```

显然这样的表达方式更为清晰，更便于记忆和使用。符号 MOV 因此称为助记符。用符号、助记符书写的指令称为符号指令。用符号指令书写程序的规范称为汇编语言，用它编写的程序称为汇编语言源程序。

微处理器不能直接识别和执行符号指令，所以汇编语言源程序要经过翻译，转换成对应的机器语言程序，才能够交计算机执行。这个翻译的操作称作汇编，由汇编程序完成。

汇编语言和机器语言都是面向机器的低级语言，是计算机的母语。使用汇编语言编程可以对计算机的硬件直接进行操控，实现计算机硬件能够实现的所有功能。此外，汇编语言编写的程序具有占用内存空间小，执行效率高的优点。

3.1　数据定义与传送

现代计算机可以处理数值数据、文字、声音、图形信息，这些信息在计算机内都是用一组二进制代码来表示的，统称为数据。计算机运行的过程，就是数据的传输、加工的过程。

本节介绍数据在计算机内的表示、数据在程序中的定义、数据传送指令及其应用。

3.1.1　计算机内数据的表示

1. 数据组织

计算机内的信息按一定的规则组织存放。

1）位

位(bit，b)是信息的最小表示单位。一位二进制可以描述一个开关的状态(称为开关量)，例如：用“1”表示接通，“0”表示断开。

2) 字节

字节(Byte,B)是计算机内信息读写、处理的基本单位,由 8 位二进制数组成。8 位二进制代码有 2^8 种不同的组合,可以表示 256(2^8)个不同的值,可以用来存放一个范围较小的整数,一个西文字符,或者 8 个开关量。

一个字节内的 8 位从右(低位)向左(高位)从 0 开始编号,依次为 b_0, b_1, …, b_7,如图 3-1(a)所示。其中,b_0 称为最低有效位(Least Significant Bit, LSB),b_7 称为最高有效位(Most Significant Bit, MSB)。

3) 字(Word)和双字(Double Word)

字和双字分别由 16 和 32 位二进制组成,或者说,分别由 2 或 4 字节组成。

字由 16 位二进制(2 字节)组成,可以存放一个范围较大的整数或一个汉字的编码。它的 16 个二进制位仍然从右(低位)向左(高位)从 0 开始编号,依次为 b_0, b_1, …, b_{15},如图 3-1(b)所示。其中,b_0～b_7 称为低位字节,b_8～b_{15} 称为高位字节。

双字由 32 位二进制组成(4 字节),可以存放范围更大的整数或者一个浮点格式表示的单精度实数。它的 32 个二进制位中,b_0～b_7,b_8～b_{15},b_{16}～b_{23},b_{24}～b_{31},分别称为低位字节、次低位字节、次高位字节、高位字节,如图 3-1(c)所示。

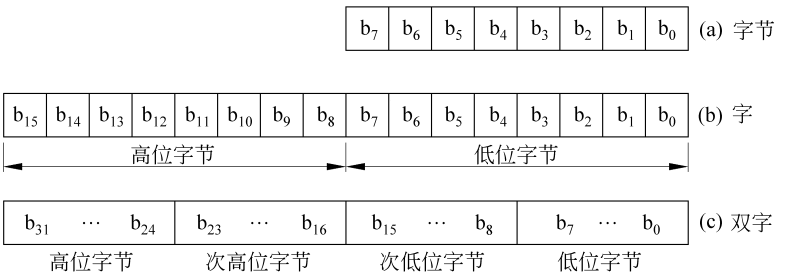


图 3-1 数据组织

2. 无符号数的表示

所谓无符号数是正数和零的集合。储存一个正数或零时,所有的位都用来存放这个数的各位数字,无须考虑它的符号,无符号数因此得名。

可以用字节、字、双字或者更多的字节来存储和表示一个无符号数。

用 N 位二进制表示一个无符号数时,最小的数是 0,最大的数是 2^N-1(N 位二进制数 111…111)。一个字节、字、双字无符号数的表示范围分别是 0～255、0～65 535、0～4 294 967 295。

一个无符号数需要增加位数时,可以在它的左侧添加若干个“0”,称为零扩展。例如,将 8 位无符号数 1011 0011 扩展为一个字时,低位字节置入这个无符号数,高位字节填“0”,结果为 0000 0000 1011 0011(插入空格是为了便于阅读,书写时没有这个要求)。

3. 有符号数的表示

计算机内用“补码”来表示一个有符号数,可以用字节、字、双字或者更多的字节来存储一个有符号数的补码。

补码表示法用最高有效位(MSB)表示一个有符号数的符号,“1”表示符号为负,“0”表示符号为正。

符号位之后的其他二进制位用来存储这个数的有效数字。正数的有效数字不变，负数的有效数字取反后最低位加 1。用字节记录一个有符号数时，$[+11011]_{补}=[+001\ 1011]_{补}=0\ 001\ 1011$，$[-11011]_{补}=[-001\ 1011]_{补}=1\ 110\ 0100+1=1\ 110\ 0101$。

对于正数 $X=+d_{n-2}d_{n-3}\cdots d_2d_1d_0$ 来说，$[X]_{补}=0\ d_{n-2}d_{n-3}\cdots d_2d_1d_0=X$。

对于负数 $Y=-d_{n-2}d_{n-3}\cdots d_2d_1d_0$，$[Y]_{补}=1\ \overline{d_{n-2}d_{n-3}\cdots d_2d_1d_0}+1=1111\cdots111-|Y|+1=2^n-|Y|=2^n+Y$。表 3-1 列出了用 8 位二进制代码表示的部分数值的补码。

表 3-1　部分数值的补码

真值(十进制)	二进制表示	补　　码
+127	+111 1111	0 111 1111
+1	+000 0001	0 000 0001
+0	+000 0000	0 000 0000
−0	−000 0000	0 000 0000
−1	−000 0001	1 111 1111
−2	−000 0010	1 111 1110
−127	−111 1111	1 000 0001
−128	−1000 0000	1 000 0000

用一个字节存储有符号数补码时，可以表示 127 个正数(1～127)，128 个负数(−1～−128)，1 个“0”(0000 0000)。其中，$[-1]_{补}=1\ 111\ 1111$，$[-128]_{补}=1\ 000\ 0000$。

如果把一个数的补码的所有位(包括符号位)“取反加 1”，将得到这个数相反数的补码。把“取反加 1”这个操作称为“求补”，$[[X]_{补}]_{求补}=[-X]_{补}$。例如，$[5]_{补}=0\ 000\ 0101$，$[[5]_{补}]_{求补}=[0000\ 0101]_{求补}=1\ 111\ 1011=[-5]_{补}$。

已知一个负数的补码，求这个数自身(真值)时，可以先求出这个数相反数(正数)的补码。例如：已知$[X]_{补}=1\ 010\ 1110$，符号位为 1，X 是一个负数。求 X 的真值可以遵循以下步骤。

(1) $[-X]_{补}=[[X]_{补}]_{求补}=[1\ 010\ 1110]_{求补}=0\ 101\ 0001+1=0\ 101\ 0010$(一个正数的补码)。

(2) $-X=[+101\ 0010]_2=+82D$(后缀'D'表示该数为十进制数)。

(3) $X=-82$。

一个补码表示的有符号数需要增加它的位数时，对于正数，在左侧添加若干个“0”，对于负数，则添加若干个“1”。上述操作实质上是用它的符号位来填充增加的“高位”(无论该数是正是负)，称为符号扩展。例如，$[-5]_{补}=1\ 111\ 1011$(8 位)$=1\ 111\ 1111\ 1111\ 1011$(16 位)，$[+5]_{补}=0\ 000\ 0101$(8 位)$=0\ 000\ 0000\ 0000\ 0101$(16 位)。

4. 字符编码

计算机处理的对象除了数值数据之外，还有大量的文字信息。文字信息以字符为基本单元。计算机内用若干位二进制来表示一个字符，这组二进制代码称为该字符的编码。

计算机内常用的字符编码是 ASCII(American Standard Code for Information

Interchange,美国信息交换标准编码)。它规定用 7 位二进制表示一个字母、数字或符号,包含 128 个不同的编码。由于计算机用 8 位二进制组成的字节作为基本存储单位,一个字符的 ASCII 码一般占用一个字节,低 7 位是它的ASCII 码,最高位置"0",或者用作"校验位"。

ASCII 编码的前 32 个(编码 00H～1FH)用来表示控制字符,例如 CR(回车,编码 0DH),LF(换行,编码 0AH)。

ASCII 编码 30H～39H 用来表示数字字符 0～9。它们的高 3 位为 011,低 4 位就是这个数字字符对应的二进制表示。例如,'5'=011 0000B + 0101B=011 0101B=35H。

ASCII 编码 41H～5AH 用来表示大写字母 A～Z。它们的高 2 位为 10。

ASCII 编码 61H～7AH 用来表示小写字母 a～z。它们的高 2 位为 11。小写字母的编码比对应的大写字母大 20H。例如,'A'=41H,'a'=61H,'a'－'A'='b'－'B'=…=20H。

完整的 ASCII 编码请参阅附录。

5. BCD 码

用一组二进制来表述一位十进制数,组间仍然按照逢十进一的规则进行。这种用二进制表示的十进制数编码称为 BCD 码(Binary Coded Decimal)。

有两种不同的 BCD 码。

压缩 BCD 码用一字节存储 2 位十进制数,高 4 位二进制表示高位十进制数,低 4 位二进制表示低位十进制数。例如,$[25]_{压缩BCD}$=0010 0101B。可以用相同数字的十六进制数来书写压缩 BCD 码。例如,用 25H 表示十进制数 25 的压缩 BCD 码。

非压缩 BCD 码用一字节存储 1 位十进制数,低 4 位二进制表示该位十进制数,对高 4 位的内容不作规定。例如,数字字符'7'的 ASCII 码 37H 就是数 7 的非压缩 BCD 码。

从上面的叙述可以看出,计算机内的一组二进制编码和它们的"原型"之间存在着"一对多"的关系。计算机内的一字节代码 96H,可以代表十进制数 96D 的压缩 BCD 编码,代表有符号数－106 的补码、无符号数 150,甚至还可以是通信用的同步字符(SYN)的偶校验 ASCII 码。所以,面对计算机内的一组二进制编码,可能无法准确地知道它究竟代表什么。知道它真面目的应该是这组二进制信息的主人,如汇编语言程序员。

3.1.2 数据的定义

1. 数据段

汇编语言程序以段为单位书写。常见的情况是,数据定义在数据段里,程序写在代码段内。

每个段有一个开始语句,一个结束语句。下面是一个例子:

```
DATA  SEGMENT                    ;DATA 数据段开始
;在这里定义数据;
⋮
DATA  ENDS                       ;DATA 数据段结束
```

汇编语言对大小写字母不加区分,DATA 与 data 被认为是相同的名字。

在汇编语言里,一行只能写一个语句。一个语句是一条指令,或者是一条伪指令,或者是对程序的注释。

第一行的语句

```
DATA   SEGMENT                    ;DATA 数据段开始
```

告诉汇编程序：一个名为 DATA 的段从本行开始了。

标识符 SEGMENT 表示一个段的开始。这个单词已经被汇编语言固定使用，称为保留字，用户不能把它用作其他用途。这个语句形式上与一条符号指令类似，但是汇编后不会产生机器指令代码，这样的语句称为伪指令，对应的操作称为伪操作。

DATA 是程序员给这个段起的名字。

程序员应该给每个段起一个含义清晰的名字。本章中，你会看到命名为 DATA_DSEG 的数据段，命名为 CSEG_CODE 的程序段，命名为 STACK_SSEG 的堆栈段。段的名字用字母或下划线开始，不能与保留字重名。

汇编语言把分号后面的文字看作是对程序的说明，称为注释，它不参加汇编，也不产生结果。注释可以跟在指令或伪指令的后面，也可以是分号开始的独立的一行。

段定义的最后一行 DATA　ENDS 表示命名为 DATA 的一个段到此结束。ENDS 是一个保留字，它也是一个伪操作，汇编时不产生代码。

数据的定义语句就写在这两行的中间。不能在一个段的内部再定义另一个段，不同的段互相独立。

2. 数据定义

1）伪操作 DB(Define Byte，定义字节数据)

用来定义字节数据。所谓定义数据，就是给出数据，为它分配存储单元，把它们用标准的格式存储到数据段中。例如，下面的定义将产生图 3-2 所示的结果。

```
DATA    SEGMENT
X       DB  -1, 255, 'A', 3+2, ?
        DB  "ABC", 0FFH,11001010B
Y       DB  3 dup(?)
DATA    ENDS
```

上面的例子里，定义了多项数据。

偏移地址	内容
0000H	11111111
0001H	11111111
0002H	01000001
0003H	00000101
0004H	00000000
0005H	01000001
0006H	01000010
0007H	01000011
0008H	11111111
0009H	11001010
000AH	00000000
000BH	00000000
000CH	00000000
000DH	

图 3-2　字节数据的定义

（1）用 DB 定义的第 2 行表示在数据段存储 5B 的数据，数据之间用逗号分隔。数据按照它们出现的先后顺序存储在数据段里。所有数据由汇编程序翻译成等值的二进制代码存储。

（2）DB 定义的数据，每个数据占用 1B 存储空间。如果是无符号数，应为［0～255］。有符号数用补码存储，应为［－128～127］。综合起来，在［－128～255］的数据都可以用 DB 来定义和储存，超出以上范围则无法存入，汇编程序将报告错误。

（3）可以出现用单/双引号括起来的单个/多个字符，每个字符占 1B 空间，按照它们出现的顺序用 ASCII 代码存储。

（4）可以出现简单的可以求出值的表达式，如第 2 行的“3＋2”，与直接写“5”效果相同。“?”表示一个尚未确定的值，在程序运行时写入，一般先用“0”填充这个单元。

（5）X 是程序员起的一个名字，代表了后面的若干个数据。因为这些数据的值在程序

里可以被改变，所以 X 也称为变量名。和代数里的含义不同，汇编语言里变量名的真正含义不是它所代表的变量的值，而是表示后面第一个数据的地址。

(6) 数据在一行写不下时，可以另起一行，仍用 DB 定义，不要重复写相同的变量名。

(7) 一般的情况下，段内的偏移地址从 0 开始，但是也可以不从 0 开始。无论怎样，数据之间的相对顺序是固定的。

(8) DUP 称为重复定义符，表示定义若干个相同的数据。本例中

```
DB 3 DUP(?)
```

等效于

```
DB  ?, ?, ?
```

同样

```
DB  2 DUP(5)
```

等效于

```
DB  5, 5
DB  2 DUP(2, 3, 4 DUP(?))
```

等效于

```
DB  2, 3, ?, ?, ?, ?, 2, 3, ?, ?, ?, ?
```

2) 伪操作 DW(Define Word，定义字数据)

用来定义字数据，每个数据占用 2B 空间，数据的高位存放在地址较大的单元里。用 DW 定义的数据范围应为 [－32768～65535]。

3) 伪操作 DD(Define Double word，定义双字数据)

用来定义双字数据，每个数据占用 4B 空间，数据的高位存放在地址较大的单元里。用 DD 定义的数据范围应为 $[-2^{31} \sim 2^{32}-1]$。

下面的定义将产生如图 3-3 所示的结果。

```
DSEG  SEGMENT
    Z    DW  -2, -32768, 65535, 'AB'
    W    DD  12345678H, -400000
         DW  Z, W-Z
DSEG  ENDS
```

(1) 有符号数据自动转换成它的补码，如 DW 定义的－2 成为 0FFFEH(以 A～F 开始的十六进制数都会在首部添加一个 0，以便与其他标识符区别)。高 8 位 0FFH 存放在偏移地址为 0001H 的存储单元里，低 8 位 0FEH 存放在偏移地址为 0000H 的存储单元里，用 0000H 代表这个字数据的地址(字地址)。

偏移地址	内容	变量名
0000H	0FEH	Z
0001H	0FFH	
0002H	00H	
0003H	80H	
0004H	0FFH	
0005H	0FFH	
0006H	42H	
0007H	41H	
0008H	78H	W
0009H	56H	
000AH	34H	
000BH	12H	
000CH	80H	
000DH	0E5H	
000EH	0F9H	
000FH	0FFH	
0010H	00H	
0011H	00H	
0012H	08H	
0013H	00H	

图 3-3　字/双字数据的定义

(2) 字符串 AB 构成一个“字”数据,'A'的 ASCII 码成为高位,'B'的 ASCII 码成为低位,这和 DB 定义时有些区别。

(3) “DW　Z, W－Z”把 Z 的偏移地址,W 和 Z 偏移地址之差存放到存储器中。W 和 Z 的偏移地址之差实际上代表了用变量名“Z”定义的所有数据占用存储器的字节数。类似地,还可以把一个变量名写在用 DD 定义的一行中,它占用 4B 空间,地址较小的 2B 内容存放这个变量名的偏移地址,地址较大的 2B 内容存放这个变量名所在段的段基址。

4) 伪操作 DQ 和 DT 用来定义 8B、10B 数据。

例如:

```
DQ  12345678ABCDEF00H
DT  112233445566778899AAH
```

大多数情况下,数据定义在一个独立的段里。有时候,为了某种需要,数据也可以定义在其他段内,比如说,定义在程序段中。

3.1.3　数据的传送

据统计,在机器指令程序中,大约有 30%的指令属于数据传送指令。本小节从数据传送指令入手,介绍汇编语言程序的基本格式和编写方法。

1. 指令格式

1) 8086 指令格式

汇编语言程序由若干个语句组成,每个语句占据源程序的一行。这些语句分成以下 3 类。

(1) 指令语句:包含一条符号指令,与一条机器指令相对应,汇编以后成为这条机器指令的二进制代码,这个代码称为目标(Object)。

(2) 伪指令语句:一条说明性的语句。有的伪指令语句汇编后没有结果,例如用 SEGMENT 定义一个段的开始,汇编程序只是在内部对定义的段名进行登记,不产生目标。有的伪指令汇编后产生目标,例如用 DB 定义的一个/若干个字节数据,汇编后产生与这些数据对应的二进制代码。

(3) 注释行:书写说明性文字,不进行汇编,也不产生目标。

下面就是一个指令行的例子:

```
BEGIN:  MOV  AX, 0                ;将 AX 寄存器清"0"
```

指令语句的一般格式如下:

```
[标号:]  操作码  [操作数]  [;注释]
```

标号是程序员给这一行起的名字,如上例的 BEGIN,后面必须加上冒号。大多数的行不需要标号。标号用字母开始,不允许使用保留字作为标号。方括号表示这项内容可以不出现。

操作码是这条指令需要完成的操作,用指令助记符表示,如上例中的 MOV。操作码本身就是保留字。

操作数是指令的操作对象。指令的操作数可以是 0～3 个,操作数之间用逗号隔开。大多数指令有两个操作数,右面的操作数称为源操作数,左面的操作数称为目的操作数。源操

作数参与指令操作，但是不存入结果，因此内容不会改变。目的操作数参与指令操作，还保存指令的操作结果。指令执行后，目的操作数的内容被更新。上例中，常数 0 是源操作数，寄存器 AX 是目的操作数，AX 的内容在指令执行后被改变。

[；注释] 用来添加一些说明，例如说明本行指令的功能。在重要指令处添加注释是一个良好的习惯。

2）操作数

操作数有寄存器操作数、立即数操作数和存储器操作数 3 种类型。

（1）寄存器操作数。寄存器操作数包括段寄存器，通用数据、地址寄存器。例如，把寄存器 AX 的内容送入 DS 寄存器可以用下面的指令：

```
MOV  DS, AX
```

其中，AX 是源操作数，写在右边，指令执行后，它的内容不会被改变。DS 是目的操作数，写在左边，指令执行后，它的内容被改变。

寄存器 IP 和 FLAG 不能作为操作数出现在指令中。

（2）立即数操作数。二进制/十进制/十六进制常数，可求出值的表达式，字符，标号等都可以用作操作数。例如，把常数 300 送入 BX 寄存器可以用下面的指令：

```
MOV  BX, 300              ;也可以写成 MOV  BX, 140*2+20
```

立即数不能用作目的操作数。

存储器操作数的表示方法比较灵活，将在下面单独介绍。

3）存储器操作数

为了对存储器的一个单元进行访问，需要给出这个单元的段基址和偏移地址。

大多数情况下，指令自动使用 DS 寄存器的内容作为操作数的段基址，为此编写汇编源程序时，首先做的事情就是把数据段的段基址装入 DS 寄存器。

存储器操作数的偏移地址可以由几个部分组合而成，合成后得到的偏移地址称做有效地址（Effective Address，EA）。

给出偏移地址的方法有直接和间接两种。

（1）直接（偏移）地址。顾名思义，所谓直接地址就是在指令里直接写出存储单元的偏移地址。例如：

```
MOV  AL, [1200H]          ;从 DS:1200H 读 1B 内容，送入 AL，方括号不能少
MOV  AX, [1200H]          ;从 DS:1200H 读 1W 内容（[1200H]和[1201H]2B 内容），送入 AX
```

从上面的例子可以看到，用常数书写的地址，可以代表 1B 的地址，也可以代表一个字的地址，没有固定的属性。

上面的常数地址虽然一目了然，但一般情况下没有实用价值，因为一般都不知道一个具体的地址单元里存放的是哪一个变量。这种写法容易导致错误，可读性也不好。

假设已经进行如下定义：

```
DATA SEGMENT
    A        DB  12, 34, 56
    ARRAY    DW  55, 66, 77, 88, 99
```

```
DATA ENDS
```

把 DATA 代表的段基址装入 DS 后，现在需要取出变量 A 的前两个数据 12，34 分别送入 BL，BH 寄存器，可以用下面的指令：

```
MOV  BL, A                    ;也可以写作  MOV  BL, [A]
MOV  BH, A+1                  ;也可以写作  MOV  BH, [A+1] 或 MOV  BH, A[1]
```

这里的 A 代表数据 12 的偏移地址，它如同上面的[1200H]一样，是一个直接地址，A＋1 是数据 34 的偏移地址。经过上面的数据段定义之后，A，A＋1 都是已知的地址。如注释所说明的，这两项都可以加上方括号，效果相同。

可能会想到用一条指令同时取出 2B 内容：

```
MOV    BX, A                  ;把变量[A]送入 BL,变量[A+1]送入 BH
```

这条指令是错误的。这是因为源操作数是字节变量，而目的操作数是字，两个不同类型的操作数不能直接传送。

(2) 间接(偏移)地址。所谓间接地址，就是把存储单元的偏移地址事先装入某个寄存器，需要时通过这个寄存器来找到这个存储单元，所以也称为寄存器间接寻址。如上例所示，为了把这两个数据装入 BL，BH 寄存器，可以这样编程：

```
MOV    SI, OFFSET A           ;把变量 A 的偏移地址装入 SI
                              ;OFFSET 是保留字,表示取出后面变量的偏移地址
MOV    BL, [SI]               ;SI 中存放变量"A"的地址,变量 A 的第一个值送入 BL
MOV    BH, [SI+1]             ;第二个值送入 BH,也可以写作 MOV BH, 1[SI]
```

对于 16 位 80x86 微处理器，只有 BX、BP、SI、DI 这 4 个寄存器可以用做间接寻址。不另加说明的话，使用 BP 时自动用 SS 的值作为段基址，使用 BX、SI、DI 时自动用 DS 的值作为段基址。

如果要取出字数组 ARRAY 的第 3 个成员“77”送入 AX，下面 3 种方法效果相同。

方法 1：

```
MOV    AX, ARRAY[4]           ;ARRAY 代表数组首地址,位移量=4,直接寻址
                              ;也可以写作"MOV  AX, ARRAY+4"
```

方法 2：

```
MOV    BX, OFFSET ARRAY       ;数组首地址装入 BX
MOV    AX, [BX+4]             ;第 3 个元素距数组首元素 4 个字节
```

方法 3：

```
MOV    BX, 4                  ;数组第 3 个成员距数组首地址的位移量装入 BX
MOV    AX, ARRAY[BX]          ;ARRAY 代表数组首地址,BX 中是位移量
```

可以用两个寄存器联合起来寻址。但是只能分别从(BX/BP)和(SI/DI)中各选出一个使用。同样，出现 BP 意味着使用 SS 作为段基址寄存器。

下面都是合法的寻址方式例子：

```
MOV    AX, ARRAY[4]        ;直接寻址,EA=ARRAY+4,使用 DS
MOV    AX, [BX]            ;寄存器间接寻址,使用 DS
MOV    AX, [BP+2]          ;寄存器相对寻址,BP 中存放首地址,位移量 2,使用 SS
MOV    AX, ARRAY [BX]      ;寄存器相对寻址,ARRAY 为首地址,BX 中存放位移量,使用 DS
MOV    AX, [BX+SI]         ;基址(BX)变址(SI)寻址,使用 DS
MOV    AX, [BP+DI+2]       ;相对基址变址寻址,使用 SS
```

2. 程序段

假设已定义数据段为 DATA,程序段的常见格式如下:

```
        CODE      SEGMENT
        ASSUME    CS: CODE, DS: DATA
START:  MOV       AX, DATA
        MOV       DS, AX
        ;其他指令
        MOV       AX, 4C00H
        INT       21H
        CODE      ENDS
        END       START
```

代码段的开始、结束和数据段类似。这里定义了一个名为 CODE 的程序段。

START 是第一条指令的标号。应注意标号与变量名的区别:标号出现在指令行前面,标号与指令之间用冒号(:)分开。本程序的执行从标有 START 的第一条指令开始,它的地址称为这个程序的入口地址。

标号 START 开始的两条指令用于装载数据段寄存器 DS。进入程序后,代码段寄存器 CS 已经由操作系统设置为代码段的段基址,数据段的段基址则需要由用户装入到 DS 中。

程序中,ASSUME 伪指令用来指定段和段寄存器之间的对应关系,供汇编程序使用。使用多个数据段时,可以清晰地看出 ASSUME 伪指令的作用。

假设有两个数据段定义如下:

```
DATA   SEGMENT
   A   DB 55
DATA   ENDS
DSEG   SEGMENT
   X   DB 10
DSEG   ENDS
```

代码段中对段的说明如下:

```
ASSUME  DS: DATA, ES: DSEG, CS: CODE
```

假设各段的段基址已经装入对应寄存器,并假设变量 A 和 X 的偏移地址都是 0000H。指令

```
MOV  AL, A
```

自动按照

```
MOV  AL, DS:[0000H]
```

的格式汇编，结果正确。

指令

```
MOV  DL, X
```

自动按照

```
MOV  DL, ES:[0000H]
```

的格式汇编，结果正确。

指令中的 DS：和 ES：指出数据所在的段。上面两条指令汇编以后，都能够正确地执行，取到正确的数据：(AL)＝55、(DL)＝10。

但是，如果这样来取数据：

```
MOV    SI, OFFSET A                ;将 A 的偏移地址装入 SI
MOV    DI, OFFSET X                ;将 X 的偏移地址装入 DI
MOV    AL, [SI]                    ;取 A 的值送给 AL
MOV    DL, [DI]                    ;取 X 的值送给 DL
```

执行的结果：(AL)＝55、(DL)＝55，这不是预期的结果。

出现错误的原因在于，虽然已经把 X 的偏移地址装入 DI，但是执行指令 MOV DL，[DI]时，仍然会使用 DS 寄存器中的段基址。为了避免上述错误，取 X 值的指令要改为

```
MOV    DL, ES: [DI]
```

这条指令显式地指定了段基址，汇编出来的机器指令比 MOV DL，[DI]多 1B，称为段跨越前缀。

注意：ASSUME 伪指令仅仅说明了段和段寄存器之间的对应关系，段基址的装入仍然需要程序员通过指令实现。

程序的最后两条指令用来结束程序运行，返回操作系统。指令 INT 21H 表示调用由操作系统提供的 21H 号服务程序。这个程序可以提供从键盘输入、显示器输出、文件操作等许多的服务，本次需要完成的服务的种类由 AH 中的功能号指定。本例中 AH＝4CH，表示返回操作系统的操作。装入 AL 中的代码称为返回代码，用 00H 表示正常返回。

与数据段相似，伪指令 CODE ENDS 表示代码段 CODE 结束。

最后一行 END 伪指令表示整个程序到此结束，在它下面书写的任何代码都不会被汇编成目标。因此，所有的段都应该写在 END 伪指令之前。这一行里的标号 START 定义这个程序的入口地址。如果在 END 之后没有写上入口标号，汇编程序会把整个源程序第一行作为入口，不管这第一行究竟是指令还是数据，这可能导致程序不能正常地执行。

综上所述，一个较完整的汇编语言源程序可以包含如下内容：

- 数据段定义；
- 代码段定义；
- 程序结束伪指令。

3. 基本传送指令

传送指令是使用最频繁的指令，要熟练地掌握使用。

1) MOV(Move,传送)指令

MOV 指令的一般格式如下：

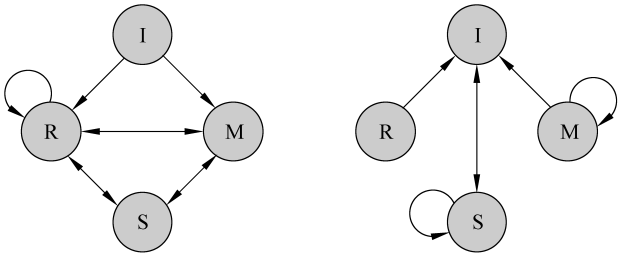
```
MOV  dest, src
```

MOV 指令把一个源操作数(source,src)传送到目的操作数(destination,dest)。指令执行后,源操作数的内容不变,目的操作数的内容与源操作数相同。例如,指令执行前,(AX)＝2345H,(BX)＝1111H。指令 MOV　AX,BX 执行后,(AX)＝1111H,(BX)＝1111H。BX 的内容复制到 AX 寄存器内。

源操作数可以是寄存器操作数、存储器操作数、立即数。

目的操作数可以是寄存器操作数、存储器操作数。

图 3-4 列出了正确的和错误的数据传送方向。图中,I、R、M、S 分别代表立即数、寄存器、存储器、段寄存器操作数。



(a) 正确的数据传送操作　　(b) 错误的数据传送操作

图 3-4　数据传送操作

MOV 指令的使用有如下限制。

(1) 源操作数与目的操作数可以是字节、字或双字,但必须具有相同的类型。

(2) 源操作数与目的操作数不能同时为存储器操作数。

(3) 目的操作数不能是立即数。

(4) FLAG、IP 不能用作操作数。

(5) 对于段寄存器作为操作数的 MOV 指令。

① 源操作数与目的操作数不能同时为段寄存器。

② 目的操作数是段寄存器时,源操作数只能是寄存器或存储器,不能是立即数。

③ CS 不能用作目的操作数。

假设变量 X_BYTE 用 DB 定义,变量 Y_WORD 用 DW 定义,它们所在段在 ASSUME 伪指令中与 DS 寄存器相对应。下面都是正确的传送指令：

```
MOV    AL, 30H            ;字节传送指令,执行后(AL)=30H
MOV    AX, 30H            ;字传送指令,30H=0030H,执行后(AX)=0030H
MOV    AL, -5             ;字节传送指令,[-5]补=0FBH,执行后(AL)=0FBH
MOV    AX, -5             ;字传送指令,[-5]补=0FFFBH,执行后(AX)=0FFFBH
MOV    CX, DX             ;字传送指令,将 DX 寄存器内容送入 CX
MOV    AX, CS             ;字传送指令,将 CS 寄存器内容送入 AX
MOV    X_BYTE, 30H        ;字节传送指令,执行后存储单元(X_BYTE)=30H
MOV    [BX], AX           ;字传送指令,将 AL 内容送入 DS:[BX],将 AH 内容送入 DS:[BX+1]
```

```
MOV    CX, Y_WORD          ;字传送指令,将存储单元 DS:[Y_WORD]内容送入 CL
                           ;将存储单元 DS:[Y_WORD +1]内容送入 CH
MOV    DX, [SI]            ;字传送指令,将 DS:[SI]内容送入 DL,将 DS:[SI+1]内容送入 DH
MOV    [BP], BL            ;字节传送指令,将 BL 寄存器内容送入 SS:[BP]处一个字节
```

使用立即数作为源操作数时,该立即数会按照目的操作数的类型进行扩展。如果立即数本身没有符号,进行零扩展,如果立即数本身有符号,进行符号扩展。

仍然假设变量 X_BYTE 用 DB 定义,变量 Y_WORD 用 DW 定义,下面是错误使用 MOV 指令的例子:

```
MOV    AX, X_BYTE          ;类型不匹配
MOV    X_BYTE, [BX]        ;不允许同时为内存操作数
MOV    CS, AX              ;CS 不允许作为目的操作数
MOV    DS, CS              ;不允许同时为段寄存器
MOV    DS, 2300H           ;目的操作数为段寄存器时,源操作数不能为立即数
MOV    DS, DATA            ;DATA 是已定义的数据段名,相当于立即数
MOV    AX, [DX]            ;不能用 DX 进行存储器间接寻址
MOV    CL, 300             ;源操作数超出范围
MOV    [BX], 20            ;无法确定操作数类型
```

可以用“类型 PTR”指定,或强行改变操作数的类型:

```
MOV    BYTE PTR[BX], 20H          ;将 1B 立即数 20H 送入 DS:[BX]
MOV    WORD PTR[BX], 20H          ;将立即数 20H 送入 DS:[BX],将 00H 送入 DS:[BX+1]
MOV    BYTE PTR[Y_WORD], 20H      ;将立即数 20H 送入字变量 Y_WORD 的第一字节
MOV    AL, BYTE PTR[Y_WORD]       ;将字变量 Y_WORD 的第一字节送入 AL 寄存器
MOV    WORD PTR[X_BYTE], 20H      ;将 2B 立即数 00 20H 送入变量 X_BYTE 开始的 2 字节
```

变量名一经定义,就已经具有明确的类型,要谨慎使用“类型 PTR 变量名”操作数。

MOV 指令执行之后,FLAG 寄存器内各标志位的状态不会发生变化。

2) LEA(Load Effective Address,装载有效地址)指令

LEA 把源操作数的偏移地址装入目的操作数。它的一般格式如下:

```
LEA  REG16, MEM
```

REG16 表示一个 16 位通用寄存器,MEM 是一个存储器操作数。上面的指令把存储器操作数的有效地址 EA 存入指定的 16 位寄存器。

假设变量 X 的偏移地址为 048CH,(BP)=1820H,(SI)=0068H

```
LEA    DX, X                  ;执行后,(DX)=048CH
LEA    BX, 4[BP][SI]          ;执行后,(BX)=4+1820H+0068H=188CH
```

上面第一条指令的功能等同于

```
MOV   DX, OFFSET X
```

但是它们是两条不同的指令。

利用 MOV 和 LEA 指令,可以编写简单的数据传送程序。

【例 3-1】 编写程序,把字节数组 ARRAY 的 4 个元素清“0”。

```
;第一个汇编语言源程序,文件名 EX301.ASM
;程序的功能:把字节数组 ARRAY 的 4 个元素清"0"
DATA    SEGMENT
ARRAY   DB  4 DUP (0FFH)
DATA    ENDS
;
CODE    SEGMENT
    ASSUME    CS: CODE, DS: DATA
START:  MOV   AX, DATA
        MOV   DS, AX
        MOV   ARRAY, 0            ;第 1 个元素清"0"
        MOV   ARRAY+1, 0          ;第 2 个元素清"0"
        MOV   ARRAY+2, 0          ;第 3 个元素清"0"
        MOV   ARRAY+3, 0          ;第 4 个元素清"0"
        MOV   AX, 4C00H
        INT   21H                 ;正常返回操作系统
CODE    ENDS
        END   START
```

这个程序里,源操作数使用立即数,目的操作数使用直接地址的寻址方式。

如果把数组 ARRAY 的首地址事先装入地址寄存器,常数 0 装入 AX,则程序更简捷:

```
MOV    AX, 0
LEA    BX, ARRAY              ;数组 ARRAY 首地址装入 BX
MOV    WORD PTR [BX], AX      ;第 1、第 2 个元素清"0"
                              ;"WORD PTR"可省略
MOV    [BX+2], AX             ;第 3、第 4 个元素清"0"
```

上面的程序里,BX 寄存器存放数组 ARRAY 的首地址,可以通过 BX 访问数组的各个元素。这种存放地址的寄存器或者存储单元称为地址指针,简称指针(Pointer)。

4. 其他传送指令

1) 地址传送指令 LDS, LES

指令格式如下:

```
LDS    REG16, MEM32           ;从存储器取出 4B,分别送入 REG16 和 DS 寄存器
LES    REG16, MEM32           ;从存储器取出 4B,分别送入 REG16 和 ES 寄存器
```

地址传送指令从存储器取出 4B,前面的 2B(地址较低)送入指定的 16 位寄存器,后面的 2B(地址较高)送入由指令操作码包含的段寄存器。

例如,指令 LDS SI, [BX]从 DS:[BX]处取出 32 位二进制,两个低地址字节送入 SI,两个高地址字节送入 DS 寄存器。指令执行后 DS 寄存器的内容被刷新。

这两条指令的执行不影响标志位。

2) 扩展传送指令 CBW, CWD

把累加器(AL/AX)的操作数符号扩展为 16/32 位,送入目的寄存器。指令格式如下:

```
CBW                           ;将 AL 寄存器内容符号扩展成 16 位,送入 AX
```

```
CWD                          ;将 AX 寄存器内容符号扩展成 32 位,送入 DX(高位)和 AX(低位)
```

指令助记符 CBW 是 Convert Byte to Word 的缩写,其余类似。这组指令主要用于有符号数除法前对被除数的位数进行扩展。

设有(AX)=8060H,分别执行下面指令后的结果如下:

```
CWD                          ;(DX)=0FFFFH,(AX)=8060H
CBW                          ;(AX)=0060H
```

3) 交换指令 XCHG

XCHG 指令交换源、目的操作数的内容,要求两个操作数有相同的类型,不能为立即数,不能同时为存储器操作数。指令格式如下:

```
XCHG  REG/MEM, REG/MEM
```

例如,(AX)=5678H,下面指令执行后的结果如下面右侧所示:

```
XCHG  AH, AL                 ;执行后(AX)=7856H
```

4) 换码指令 XLAT

换码指令用 AL 寄存器的内容查表,结果存回 AL 寄存器。要求表格的首地址事先存放在 DS:BX 中。指令格式如下:

```
XLAT                         ;AL←DS:[BX+AL]
XLAT  MEM16                  ;以 MEM16 所在段寄存器为段基址,以 BX 为偏移地址查表
```

设(AL)=0000 1011B=0BH,下面程序执行后,AL 中的二进制数改变为对应的十六进制数字符的 ASCII 代码 0100 0010('B')。

```
TABLE  DB  "0123456789ABCDEF"
⋮
PUSH   DS                    ;保护 DS 寄存器内容
MOV    BX, SEG  TABLE        ;取出 TABLE 所在的段基址送入 BX,"SEG"表示取"段基址"
MOV    DS, BX                ;段基址从 BX 转送入 DS
LEA    BX, TABLE             ;取出 TABLE 的偏移地址
XLAT                         ;查表,(AL)=0100 0010('B')
POP    DS                    ;恢复 DS 寄存器内容
```

5. 堆栈

和数据段、代码段一样,堆栈(STACK)也是用户使用的存储器的一部分,用来存放一些临时性的数据和其他信息,例如函数使用的局部变量、调用子程序的入口参数、返回地址等。

1) 堆栈段结构

下面是一个堆栈段的定义:

```
SSEG    SEGMENT STACK
        DW 6 DUP(?)
SSEG    ENDS
```

在 SEGMENT 伪指令中增加 STACK 表示该段是堆栈。有了这项说明,操作系统在装

入这个程序时，会自动地把 SSEG 的段基址置入 SS，堆栈段的字节数（本例中为 12＝000CH）置入 SP。这样，SP 指向了这个堆栈的栈顶。

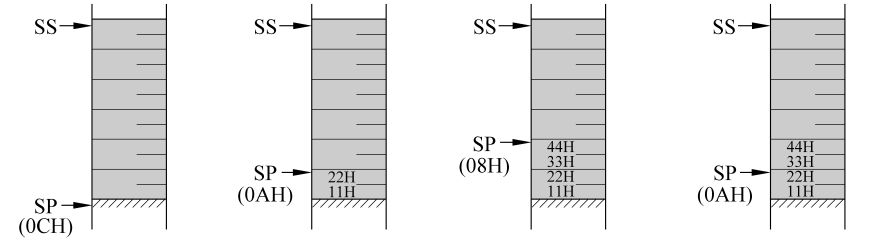
堆栈段的使用和数据段、代码段有以下不同。

① 从较大地址开始分配和使用（数据段、代码段从较小地址开始分配和使用）。

② SP 中地址指出的存储单元称为栈顶。进行堆栈操作时，数据总是在栈顶位置存入（称为压入），取出（称为弹出）。

③ 最先进入的数据最后被弹出（First In Last Out，FILO），最后进入的数据最先被弹出（Last In First Out，LIFO）。

以上面的定义为例，堆栈的初始状态，装入、弹出数据后的状态如图 3-5 所示。



(a) 堆栈的初始状态　(b) 压入一个数据之后　(c) 压入两个数据之后　(d) 弹出一个数据之后

图 3-5　堆栈段结构

上面定义的堆栈，占用 SS：0000H～SS：000BH 共 12 个字节单元。

堆栈尚未使用时，堆栈为空，SP 指向堆栈下面的单元，如图 3-5(a)所示，(SP)＝0012H。

对于 8086 CPU，进、出堆栈只能是 2B 数据。压入一个 2B 数据的操作如下：

```
SP←(SP)-2
SS:[SP]←数据
```

例如，数据 1122H 压入堆栈后，如图 3-5(b)所示，由 SP 指出的栈顶位置上移，(SP)＝000AH，SS:[SP]＝22H，SS:[SP＋1]＝11H。

第二个数据 3344H 进栈后堆栈的状态如图 3-5(c)所示。

从堆栈弹出一个数据的操作如下：

```
目的操作数←SS:[SP]
SP←(SP)+2
```

如图 3-5(d)所示，弹出一个数据后，栈顶的位置下移，(SP)＝000AH，堆栈段存储器的内容实际上没有发生变化，但是从逻辑上可以认为，堆栈中只有一项数据：1122H。

2) 8086 CPU 堆栈指令和标志寄存器指令

8086 CPU 堆栈指令和标志寄存器指令如表 3-2 所示。

表 3-2　8086 CPU 堆栈指令和标志寄存器指令

指令名称	操作码	指令格式	功　能	主要用途
压栈	PUSH	PUSH REG16/ MEM16/SEG	SP←(SP)－2， SS：[SP]←字数据	数据入栈保护

续表

指令名称	操作码	指令格式	功　能	主要用途
出栈	POP	POP REG16/ MEM16/SEG	目的操作数←SS:[SP], SP←(SP)+2	数据从堆栈恢复
标志寄存器入栈	PUSHF	PUSHF	SP←(SP)−2, SS:[SP]←FLAGS	标志寄存器入栈保护
标志寄存器出栈	POPF	POPF	FLAGS←SS:[SP], SP←(SP)+2	从堆栈恢复标志寄存器
装载 FLAGS	LAHF	LAHF	FLAGS 低 8 位←AH	装载 FLAGS 低 8 位
卸载 FLAGS	SAHF	SAHF	AH←FLAGS 低 8 位	保存 FLAGS 低 8 位

下面的程序段把 CS 寄存器内容存入 DS：

```
PUSH  CS
POP   DS
```

下面的程序段把 TF 标志位置位(置“1”)：

```
PUSHF
POP    AX                          ;AX←Flags
OR     AX, 0100H                   ;在 AX 内将 b8(TF 位)置"1"
PUSH   AX
POPF                               ;Flags←AX
```

6. 操作数表达式

指令中的操作数,包括立即数和存储器操作数都可以用一个表达式来代替,这个表达式在汇编成目标的时候进行计算,它的结果用来产生目标代码。例如,设变量 X 的偏移地址为 1020H,汇编指令 MOV　AL, X+5 时,把 X 的偏移地址 1020H 和 5 相加,得到结果 1025H,产生 MOV　AL, [1025H]对应的机器指令代码。

1) 符号定义伪指令

可以用 EQU 和“=”来定义一个符号,这个符号在后面的指令中用作立即操作数或者存储器操作数。符号定义伪指令的格式如下：

```
符号名  EQU    表达式
符号名  =      常数表达式
```

汇编时,对 EQU 定义的符号名用对应的表达式进行替换。例如,有以下定义：

```
NUM        EQU    3+2
ERR_MSG    EQU    "Data Override"
POINTER    EQU    BUFFER[DI]
WP         EQU    WORD PTR
```

下面是这些符号名使用的例子：

```
MESSAGE  DB    ERR_MSG             ;等价于 MESSAGE  DB    "Data Override "
         MOV   BX, POINTER         ;等价于 MOV  BX, BUFFER[DI]
```

```
MOV   CX, NUM * 4              ;等价于 MOV  CX, 3+2 * 4,11 送入 CX
MOV   WP POINTER, 0            ;等价于 MOV  WORD PTR BUFFER[DI], 0
```

使用“=”定义符号名时,只能使用常数表达式,而且对一个符号名可以多次定义。一个新的定义出现后,原来的定义自动终止。例如:

```
TIMES=0
⋮
TIMES=TIMES+1
⋮
```

用 EQU 定义的符号名不允许重复定义。

将多次出现、不便记忆的常数、表达式定义为符号名,有助于提高程序的可读性和可靠性。这个常数/表达式内容需要修改时,只需要修改一条符号定义伪指令,而不需要搜索整个程序多处修改它。两种符号定义有一个共同的规则:先定义,后使用。所以,符号定义伪指令一般出现在源程序的首部。

2) 地址表达式

指令中的存储器操作数最终都是以偏移地址为结果,产生对应的有效地址(Effective Address,EA)。用于计算有效地址的地址表达式有 3 个运算符:+、-、[]。

+和-运算符对构成有效地址的各个分量进行加、减操作。仍然设变量 X 的偏移地址为 1020H,X+5 产生 EA=1025H,指令 MOV BL, X-10H 产生 EA=1010H。

[]称为索引运算符,用来括起组成有效地址的一个分量,各分量相加,得到最后的有效地址。例如,指令 MOV AX, 2[BX][DI]等效于 MOV AX, [BX+DI+2],指令 MOV AX, BUFFER[BX][2]等效于 MOV AX, [BUFFER+BX+2]。

3) 立即数表达式

立即数表达式在汇编源程序时进行计算,它的结果用作指令中的立即数操作数。这种表达式中的运算对象必须是已知的,否则无法进行计算。用于产生立即数操作数的表达式有 4 类运算符:算术运算符、逻辑运算符、关系运算符和地址运算符。

(1) 算术运算符。算术运算符有+(相加)、-(相减)、*(相乘)、/(整除运算)和 MOD(取余数)。运算优先级从高到低依次为(*,/)→(MOD)→(+, -)。允许使用圆括号改变运算顺序。

例如,指令 MOV BX, 32+13/6 MOD 3 中,表达式计算顺序是 32+((13/6)MOD 3),得到结果 34,该指令汇编后产生与 MOV BX, 0022H 相同的机器指令代码。

(2) 逻辑运算符。逻辑运算符有 SHR(右移)、SHL(左移)、AND(逻辑与)、OR(逻辑加)、XOR(异或,半加)和 NOT(逻辑非、取反)。例如 30 SHR 1 产生结果 15。

(3) 关系运算符。关系运算符用于两个数的比较,结果为“真(-1)”或“假(0)”。运算符有 GT(大于)、GE(大于或等于)、LT(小于)、LE(小于等于)、EQ(等于)和 NE(不等于)。

例如,指令 MOV AX, 6000H GE 5000H 中的表达式结果为“真”,产生指令 MOV AX,0FFFFH 对应的机器代码。指令 MOV AX, -3 EQ 2 中的表达式结果为“假”,产生指令 MOV AX,0000H 对应的机器代码。

(4) 地址运算符。地址运算符对变量名、标号、地址表达式进行计算,得到作为立即数

的运算结果。

SEG 取地址表达式所在段的段基址。设变量 LIST 定义在 DATA 段中，下面 3 条指令都是把 DATA 段的段基址装入 AX：

```
MOV  AX, DATA                    ;符号名 DATA 代表该段的段基址，是一个立即数
MOV  AX, SEG DATA                ;取 DATA 的段基址，结果是立即数
MOV  AX, SEG LIST                ;取 LIST 的段基址，结果是立即数
```

OFFSET 取地址表达式的偏移地址，下面两条指令进行了不同的操作：

```
MOV  AX, LIST                    ;取出字变量 LIST 第一个元素(2B)送入 AX
MOV  AX, OFFSET LIST             ;取出变量 LIST 的偏移地址送入 AX
```

TYPE、LENGTH 和 SIZE 这 3 个运算符仅仅对变量名、标号进行操作，分别用于取变量、标号的类型，取变量定义时的元素个数，取变量占用的字节数。例如：

```
X  DB  "ABCDE"                   ;TYPE=1, LENGTH=1, SIZE=1
Y  DW  3 DUP(5), 4 DUP(-1)       ;TYPE=2, LENGTH=3, SIZE=6
Z  DD  34, 49, 18                ;TYPE=4, LENGTH=1, SIZE=4
```

不同的汇编语言版本对上面例子的处理结果可能不同，本书以 Borland TASM 5.x 为例。

再次强调一下：上面所有的表达式都必须是汇编期间可以求值的。MOV　AX，BX＋2 是一条错误的指令，汇编时将报告错误，原因在于 BX 的值是未知的，可变的，在汇编阶段无法进行相关的计算。需要把 BX 的值与常数 2 相加并存入 AX 的操作只能在程序执行阶段由以下两条指令完成：

```
MOV   AX, BX                     ;BX 寄存器的值存入 AX 寄存器
ADD   AX, 2                      ;AX 寄存器的值加上 2，结果存入 AX
```

3.1.4　简化段格式

除了前面所述的汇编语言源程序格式之外，还有一种称为简化段格式。整个源程序组成如下：

```
内存模式说明
[数据段定义]
[堆栈段定义]
[代码段定义]
END  入口标号
```

其中，带有方括号的内容不是必需的，根据需要选择使用，也可以改变出现的次序。

1. 内存模式说明

内存模式说明的格式如下：

```
.MODEL  内存模式
```

可选择的内存模式如下。

(1) Tiny：微型，整个程序只有一个段，供.COM 格式程序使用。

(2) Small：小型，程序由一个代码段、一个数据段(包括堆栈段)组成。

(3) Medium：中型，有多个代码段，但只有一个数据段。

(4) Large：大型，有多个代码段和多个数据段。

(5) Flat：平坦，供编写 32 位微处理器的汇编语言源程序，在 Windows 下运行。

2. 数据段定义

数据段定义格式如下。

```
.DATA                                    ;常用的格式
```

或者

```
.FARDATA [数据段段名]                    ;定义第二个数据段
```

或者

```
.DATA?                                   ;定义一个没有初始值的数据段
```

只有一个数据段时，.DATA 表示数据段的开始，随后可以定义各项数据。DATA 是保留字，不能随意更换。使用.DATA? 可以减少可执行文件大小。

3. 堆栈段定义

堆栈段定义格式如下：

```
.STACK [堆栈段大小]
```

省略堆栈段大小时，堆栈段自动设置为 1024B(1KB)。程序装入执行时，根据所定义的堆栈段自动装载 SS，SP 寄存器。

4. 代码段定义

代码段定义格式如下：

```
.CODE [代码段段名]
```

只有一个代码段时可以省略代码段段名。

使用上述格式时，一个段的开始同时意味着上一个段的结束，也就是说，只需要声明段的开始，不需要声明段的结束。

使用简化段定义格式重写例 3-1 如下：

```
        .MODEL  SMALL
        .DATA
        ARRAY   DB  4 DUP (0FFH)
        .CODE
START:  MOV     AX, @DATA                ;@DATA 代表.DATA 定义的数据段段基址
        MOV     DS, AX                   ;装载 DS
        …                                ;第 1 个元素清"0"
        ⋮                                ;……
        MOV     AX, 4C00H
        INT     21H
        END     START
```