第5章 汇编语言的基本语法

本章首先介绍汇编语言程序的结构和基本语法，然后简要介绍 ROM BIOS 中断调用和 DOS 系统功能调用，最后介绍汇编语言程序上机调试的基本过程。

5.1 汇编语言的特点

用指令的助记符、符号常量、标号等符号形式书写程序的程序设计语言称为汇编语言(Assemble Language)。它是一种面向机器的程序设计语言，其基本内容是机器语言的符号化描述。

与机器语言相比，使用汇编语言来编写程序的突出优点就是可以使用符号。具体地说，就是可以用助记符来表示指令的操作码和操作数，可以用标号来代替地址，用符号表示常量和变量。助记符一般都是表示相应操作的英文字母的缩写，便于识别和记忆。不过，用汇编语言编写的程序不能由机器直接执行，而必须翻译成由机器代码组成的目标程序，这个翻译的过程称为汇编。当前绝大多数情况下，汇编过程是通过软件自动完成的。用来把汇编语言编写的程序自动翻译成目标程序的软件叫汇编程序(即汇编器 Assembler)。汇编过程的示意如图 5.1 所示。

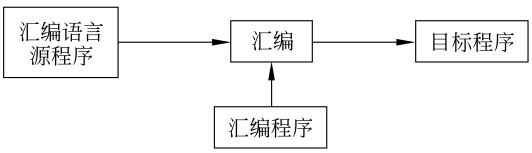

图 5.1 汇编过程示意

用汇编语言编写的程序叫汇编语言源程序。第 4 章中介绍的指令系统中的每条指令都是构成汇编语言源程序的基本语句。汇编语言的指令和机器语言的指令之间有一一对应的关系。

汇编语言是和机器硬件密切相关的，汇编代码基于特定平台，不同 CPU 的机器有不同的汇编语言。采用汇编语言进行程序设计时，可以充分利用机器的硬件功能和结构特点，从而可以有效地加快程序的执行速度，减少目标程序所占用的存储空间。

与高级语言相比，汇编语言提供了直接控制目标代码的手段，而且可以直接对输入/输出端口进行控制，实时性能好，执行速度快，节省存储空间。大量的研究与实践表明，为解决同一问题，用高级语言与用汇编语言所写的程序，经编译与汇编后，它们所占用的存储空间与执行速度存在很大差别。统计表明，汇编语言所占用的存储空间要节省 30%，执行速度要快 30%。所以，对这两方面要求都很高的实时控制程序往往用汇编语言编写。另外，要了解计算机是如何工作的，也要学习汇编语言。汇编语言的出现是计算机技术发展的一个重要里程碑。它迈出了走向今天我们所使用的高级语言的第一步。

汇编语言的缺点是编程效率较低，又由于它紧密依赖于机器结构，所以可移植性较差，即在一种机器系统上编写的汇编语言程序很难直接移植到不同的机器系统上去。

尽管如此，由于利用汇编语言进行程序设计具有很高的时空效率并能够充分利用机

器的硬件资源等方面的特点,使其在需要软、硬件结合的开发设计中尤其是计算机底层软件的开发中,仍有着其他高级语言所无法替代的作用。

早期的汇编语言程序设计主要是在 DOS 环境下进行的,随着计算机软、硬技术的发展,目前在 Windows、Linux、UNIX 环境下的汇编语言程序设计技术已引起人们普遍的关注和重视。汇编程序(汇编器)的功能也在不断增强,版本不断更新,早期有 8086 汇编程序 ASM-86,后来出现宏汇编程序 MASM-86,目前广泛使用的是 MASM 5.X、MASM 6.X 等。

5.2 汇编语言程序结构和基本语法

5.2.1 示例程序

为了更好地介绍汇编语言程序的格式和基本语法,下面给出一个完整的汇编语言源程序示例。该程序的具体功能是将程序中定义的 16 位二进制数转换为 4 位十六进制数,并输出到显示器上去。

```
DATA  SEGMENT                          ;数据段
    NUM  DW  0011101000000111B        ;即 3A07H
    NOTES  DB  'The result is:', '$'
DATA  ENDS
STACK  SEGMENT  STACK                  ;堆栈段
    STA  DB  50 DUP (?)
    TOP  EQU  LENGTH  STA
STACK  ENDS
CODE  SEGMENT                          ;代码段
    ASSUME  CS:CODE , DS: DATA , SS:STACK
BEGIN: MOV  AX , DATA
       MOV  DS , AX                    ;为 DS 赋初值
       MOV  AX , STACK
       MOV  SS , AX                    ;为 SS 赋初值
       MOV  AX , TOP
       MOV  SP , AX                    ;为 SP 赋初值
       MOV  DX , OFFSET NOTES          ;显示提示信息
       MOV  AH , 09H
       INT  21H
       MOV  BX , NUM                   ;将数据装入 BX
       MOV  CH , 4                     ;共 4 个十六进制数字
ROTATE:MOV  CL , 4                     ;CL 为移位位数
       ROL  BX , CL
       MOV  AL , BL
       AND  AL , 0FH                   ;AL 中为一个十六进制数
       ADD  AL , 30h                   ;转换为 ASCII 码值
       CMP  AL , '9'                   ;是 0～9 的数码
       JLE  DISPLAY
```

```
        ADD   AL , 07H                ;在 A～F 之间
DISPLAY:MOV   DL , AL                 ;显示这个十六进制数
        MOV   AH , 2
        INT   21H
        DEC   CH
        JNZ   ROTATE
        MOV   AX , 4C00H              ;退出程序并回到 DOS
        INT   21H
CODE    ENDS                          ;代码段结束
        END   BEGIN                   ;程序结束
```

从这个示例程序可以清楚地看到汇编语言源程序的两个组成特点：分段结构和语句行。下面简要说明这两个特点。

1. 分段结构

汇编语言源程序是按段来组织的。通过第 3 章的介绍我们已经知道，8086 汇编源程序最多可由 4 种段组成，即代码段、数据段、附加段和堆栈段，并分别由段寄存器 CS、DS、ES 和 SS 中的值(段基值)来指示段的起始地址，每段最大可占 64K 字节单元。

从示例程序可以看到，每段有一个名字，并以符号 SEGMENT 表示段的开始，以 ENDS 作为段的结束符号。两者的左边都必须有段的名字，而且名字必须相同。

示例程序中共有 3 个段，分别是数据段(段名为 DATA)、堆栈段(段名为 STACK)和代码段(段名为 CODE)。

2. 语句行

汇编语言源程序的段由若干语句行组成。语句是完成某种操作的指示和说明，是构成汇编语言程序的基本单位。上述示例程序共有 38 行，即共有 38 个语句行。汇编语言程序中的语句可分为 3 种类型：指令语句、伪指令语句和宏指令语句。

需要指出的是，对于指令语句，汇编程序将把它翻译成机器代码，并由 CPU 识别和执行；而对于伪指令语句(又称指示性语句)，汇编程序并不把它翻译成机器代码，它仅向汇编程序提供某种指示和引导信息，使之在汇编过程中完成相应的操作，如给特定符号赋予具体数值，将特定存储单元放入所需数据等；关于宏指令的特点，将在后面 5.2.5 节作具体介绍。

5.2.2 基本概念

下面结合示例程序，详细介绍汇编语言程序设计中的几个基本概念。

1. 标识符

标识符也叫名字，是程序员为了使程序便于书写和阅读所使用的一些字符串。例如示例程序中的数据段名 DATA，代码段名 CODE，程序入口名 BEGIN，标号名 DISPLAY

等。定义一个标识符有如下几点要求：

(1) 标识符可以由字母 A～Z,a～z,数字 0～9,专用字符?,·,@,$,_等符号构成；

(2) 标识符不能以数字开始,如果用到字符“·”则必须是第一个字符；

(3) 标识符长度不限,但是宏汇编程序仅识别前 31 个字符。

2. 保留字

保留字(也称关键字)是汇编语言中预先保留下来的具有特殊含义的符号,只能作为固定的用途,不能由程序员任意定义。例如示例程序中的 SEGMENT、MOV、INT、END 等。所有的寄存器名、指令操作助记符、伪指令操作助记符、运算符和属性描述符等都是保留字。

3. 数的表示

在没有 8087、80287、80387 等数学协处理器的系统中,所有的常数必须是整数。表示一个整数应遵循如下的规则：

(1) 默认情况下是十进制,但可以使用伪指令“RADIX n”来改变默认基数,其中 n 是要改变成的基数。

(2) 如果要用非默认基数的进位制来表示一个整数,则必须在数值后加上基数后缀。字母 B,D,H,O 或 Q 分别是二进制、十进制、十六进制、八进制的基数后缀。例如示例程序中的 0011101000000111B、21H 等整数。

(3) 如果一个十六进制数以字母开头,则必须在前面加数字 0。例如,十六进制数 F 应表示为 0FH。

(4) 可以用单引号括起一个或多个字符来组成一个字符串常数,如示例程序中的 'The result is:'。字符串常数以串中字符的 ASCII 码值存储在内存中,如'The'在内存中就是 54H、68H、65H。

在有数学协处理器的系统中,可以使用实数。实数的类型有多种,但其一般的表示形式如下：

$$\pm\text{整数部分.小数部分 E}\pm\text{指数部分}$$

例如,实数 5.213×10^{-6} 表示为 5.213E－6。

4. 表达式和运算符

表达式由运算符和操作数组成,可分为数值表达式和地址表达式两种类型。

操作数可以是常数、变量名或标号等,在内容上可能代表一个数据,也可能代表一个存储单元的地址。变量名和标号都是标识符。例如示例程序中的变量名 NUM、NOTES 和标号 BEGIN、ROTATE 等。

数值表达式能被计算产生一个数值的结果。而地址表达式的结果是一个存储器的

地址，如果这个地址的存储区中存放的是数据，则称它为变量；如果存放的是指令，则称它为标号。

汇编语言程序中的运算符的种类很多，可分为算术运算符、逻辑运算符、关系运算符、分析运算符、综合运算符、分离运算符、结构和记录中专用运算符和其他运算符等几类，如表 5.1 所示。

表 5.1　表达式中的运算符

类型	符号	功　能	实　例	运算结果
算术运算符	＋	加法	2＋7	9
	－	减法	9－7	2
	*	乘法	2 * 7	14
	/	除法	14/7	2
	MOD	取模	16/7	2
	SHL	按位左移	0010B SHL 2	1000B
	SHR	按位右移	1100B SHR 1	0110B
逻辑运算符	NOT	逻辑非	NOT　0110B	1001B
	AND	逻辑与	0101B　AND　1100B	0100B
	OR	逻辑或	0101B　OR　1100B	1101B
	XOR	逻辑异或	0101B　XOR　1100B	1001B
关系运算符	EQ	相等	2　EQ　11B	全 0
	NE	不等	2　NE　11B	全 1
	LT	小于	2　LT　10B	全 0
	LE	小于等于	2　LE　10B	全 1
	GT	大于	2　GT　10B	全 0
	GE	大于等于	2　GE　10B	全 1
分析运算符	SEG	返回段基值	SEG　DA1	
	OFFSET	返回偏移地址	OFFSET　DA1	
	LENGTH	返回变量单元数	LENGTH　DA1	
	TYPE	返回变量的类型	TYPE　DA1	
	SIZE	返回变量总字节数	SIZE　DA1	
综合运算符	PTR	指定类型属性	BYTE　PTR　[DI]	
	THIS	指定类型属性	ALPHA EQU　THIS　BYTE	
分离运算符	HIGH	分离高字节	HIGH　2277H	22H
	LOW	分离低字节	LOW　2277H	77H
专用运算符	.	连接结构与字段	FRM. YER	
	＜　＞	字段赋值	＜，2，7＞	
	MASK	取屏蔽	MASK　YER	
	WIDTH	返回记录/字段所占位数	WIDTH　YER	
其他运算符	SHORT	短转移说明	JMP　SHORT　LABEL2	
	(　)	改变运算优先级	(7－2) * 2	10
	[　]	下标或间接寻址	ARY [4]	
	段前缀	段超越前缀	CS：[BP]	

如果在一个表达式中出现多个上述的运算符,将根据它们的优先级别由高到低的顺序进行运算,优先级别相同的运算符则按从左到右的顺序进行运算。运算符的优先级别如表 5.2 所示。

表 5.2　运算符的优先级别

优先级别		运　算　符
高级 ↑	0	圆括号(),方括号[],尖括号<>,点运算符 · LENGTH,WIDTH,SIZE,MASK
	1	PTR,OFFSET,SEG,TYPE,THIS 段超越前缀
	2	HIGH,LOW
	3	*,/,MOD,SHL,SHR
	4	+,-
	5	EQ,NE,LT,LE,GT,GE
	6	NOT
	7	AND
低级	8	OR,XOR
	9	SHORT

下面对各种运算符做简单说明。

(1) 算术运算符

算术运算符的运算对象和运算结果都必须是整数。其中求模运算 MOD 就是求两个数相除后的余数。移位运算 SHL 和 SHR 可对数进行按位左移或右移,相当于对此数进行乘法或除数运算,因此归入算术运算符一类。注意,8086 指令系统中也有助记符为 SHL 和 SHR 的指令,但与表达式中的移位运算符是有区别的。表达式中的移位运算符是伪指令运算符,它是在汇编过程中由汇编器进行计算的;而机器指令中的移位助记符,它是在程序运行时由 CPU 执行的操作。例如:

```
MOV   AL , 00011010B SHL 2          ;相当于 MOV   AL , 01101000B
SHL   AL , 1                        ;移位指令,执行后 AL 中为 D0H
```

本例第一行中的"SHL" 是伪指令的移位运算符,它在汇编过程中由汇编器负责计算;第二行中的"SHL"是机器指令的移位助记符,它在程序运行时由 CPU 负责执行。

(2) 逻辑运算符

逻辑运算符对操作数按位进行逻辑运算。指令系统中也有助记符为 NOT、AND、OR、XOR 的指令,两者的区别同上述"移位运算符"与"移位指令助记符"的区别一样。例如:

```
MOV   AL , NOT 10100101B            ;相当于 MOV   AL , 01011010B
NOT   AL                            ;逻辑运算指令
```

(3) 关系运算符

关系运算符对两个操作数进行比较,若条件满足,则运算结果为全"1";若条件不满

足，则运算结果为全“0”。例如：

```
MOV  AX , 5 EQ 101B
MOV  BH , 10H GT 16
MOV  BL , 0FFH EQ 255
MOV  AL , 64H GE 100
```

等效于：

```
MOV  AX , 0FFFFH
MOV  BH , 00H
MOV  BL , 0FFH
MOV  AL , 0FFH
```

(4) 分析运算符

分析运算符可以“分析”出运算对象的某个参数，并把结果以数值的形式返回，所以又叫数值返回运算符。主要有 SEG、OFFSET、LENGTH、TYPE 和 SIZE 5 个分析运算符，下面分别予以介绍。

① SEG 运算符加在某个变量或标号之前，返回该变量或标号所在段的段基值。

② OFFSET 运算符加在某个变量或标号之前，返回该变量或标号的段内偏移地址。

③ LENGTH 运算符加在某个变量之前，返回的数值是一个变量所包含的单元(可以是字节、字、双字等)数，对于变量中使用 DUP 的情况，将返回以 DUP 形式表示的第一组变量被重复设置的次数；而对于其他情况则返回 1。

④ TYPE 运算符加在某个变量或标号之前，返回变量或标号的类型属性，返回值与类型属性的对应关系如表 5.3 所示。

表 5.3 TYPE 运算符的返回值

变量类型	返回值	标号类型	返回值
字节(BYTE)	1	近(NEAR)	−1(FFH)
字(WORD)	2	远(FAR)	−2(FEH)
双字(DWORD)	4		
四字(QWORD)	8		
十字节(TBYTE)	10		

⑤ SIZE 运算符加在某个变量之前，返回数值是变量所占的总字节数，且等于 LENGTH 和 TYPE 两个运算符返回值的乘积。

例如：

```
K1   DB   4 DUP(0)
K2   DW   10 DUP(?)
MOV  AH , LENGTH K1                ;LENGTH K1 = 4
MOV  AL , SIZE K1                  ;TYPE K1 = 1, SIZE K1 = LENGTH K1 × TYPE K1 = 4 × 1 = 4
MOV  BH , LENGTH K2                ;LENGTH K2 = 10
MOV  BL , SIZE K2                  ;TYPE K2 = 2, SIZE K2 = LENGTH K2 × TYPE K2 = 10 × 2 = 20
```

四条 MOV 指令分别等效于：

```
MOV  AH , 4
MOV  AL , 4
MOV  BH , 10
MOV  BL , 20
```

(5) 综合运算符

综合运算符可用于指定变量或标号的属性，因此也叫属性运算符。其主要有 PTR 和 THIS 两个综合运算符，下面分别予以介绍。

① PTR 运算符用来规定内存单元的类型属性，格式是：

```
类型  PTR  符号名
```

其含义是将 PTR 左边的类型属性赋给其右边的符号名。例如：

指令“MOV BYTE PTR [1000H] , 0 ”使 1000H 字节单元清 0；

指令“MOV WORD PTR [1000H] , 0 ”使 1000H 和 1001H 两个字节单元清 0。

② THIS 运算符可以用来改变存储区的类型属性。格式是：

```
符号名  EQU  THIS  类型
```

其含义是将 THIS 右边的类型属性赋给 EQU 左边的符号名，并且使该符号名的段基值和偏移量与下一个存储单元的地址相同。THIS 运算符并不为它所在语句中的符号名分配存储空间，其功能是为下一个存储单元另起一个名字并另定义一种类型，从而可以使同一地址单元具有不同类型的名字，便于引用。例如：

```
A  EQU  THIS  BYTE
B  DW  1234H
```

此时，A 的段基值和偏移量与 B 完全相同。相当于给变量 B 起了个别名叫 A，但 A 的类型是字节型，而 B 的类型为字型；以后当用名字 A 来访问存储器数据时，实际上访问的是 B 开始的数据区，但访问的类型是字节。换句话说，对于 B 开始的数据区既可用名字 A 以字节类型来访问，也可用名字 B 以字的类型来访问。如对于上面的例子，可有如下的访问结果：

```
MOV  AL , A                    ;指令执行后,AL = 34H
MOV  AX , B                    ;指令执行后,AX = 1234H
```

当 THIS 语句中的符号名代表一个标号时，则能够赋予该标号的类型为 NEAR 或 FAR，例如：

```
BEGIN  EQU  THIS  FAR
       ADD  CX , 100
```

从而使 ADD 指令有一个 FAR 属性的地址 BEGIN，于是允许其他段通过 JMP 指令(如“JMP FAR PTR BEGIN”)远跳转到这里来。

注意，PTR 运算符只在使用它的语句中有效，而 THIS 运算符则影响从使用处往后

的程序段。

(6) 分离运算符

HIGH 运算符用来从运算对象中分离出高字节,LOW 运算符用来从运算对象中分离出低字节。例如:

```
MOV  AL , HIGH  1234H                ;相当于 MOV AL,12H
MOV  AL , LOW  1234H                 ;相当于 MOV AL,34H
```

(7) 其他运算符

① 短转移说明运算符 SHORT 用来说明一个转移指令的目标地址与本指令的字节距离为－128～＋127。例如:

```
JMP  SHORT  LABEL2
```

② 圆括号运算符()用来改变运算符的优先级别,()中的运算符具有最高的优先级,与常见的算术运算的()的作用相同。

③ 方括号运算符[]常用来表示间接寻址。例如:

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

④ 段超越前缀运算符“:”表示后跟的操作数由指定的段寄存器提供段基值。例如:

```
MOV  BL , DS: [BP]                   ;把 DS: BP 单元中的值送 BL
```

5. 语句

和任何高级语言一样,语句是构成汇编语言程序的基本单位。汇编语言程序中的每个语句由四项组成,一般格式如下:

```
[名字项] 操作项 [操作数项][; 注释]
```

其中除“操作项”外,其他部分都是可选的。“名字项”是一个标识符,它可以是一条指令的标号或一个操作数的符号地址等;操作项是某种操作的助记符,例如加法指令的助记符 ADD 等;而“操作数项”由一个或多个操作数组成,它给所执行的操作提供原始数据或相关信息;注释由分号“;”开始,其后可为任意的文本。若一行的第一个字符为分号,则整行被视为注释。也可用 COMMENT 伪操作定义多行注释。注释会被汇编程序忽略,但对于读、写和调试源程序有很大帮助。提倡在源程序中给出充分的、恰如其分的注释。

程序中语句之间以及一条语句的各项之间都必须用分隔符分隔。其中分号“;”是注释开始的分隔符,冒号“:”是标号与汇编指令之间的分隔符,逗号“,”用来分隔两个操作数,“空格”(Space 键) 和“制表符”(Tab 键) 则可用于为了表示的清晰而在任意两部分之间插入若干个空格或制表符。

前面已指出,汇编语言程序中的语句分为指令语句、伪指令语句和宏指令语句三种,

下面分别详细介绍。

5.2.3 指令语句

指令语句是要求 CPU 执行某种操作的命令,可由汇编程序翻译成机器代码。指令语句的格式如图 5.2 所示。

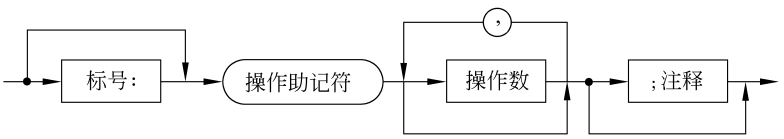

图 5.2 指令语句的格式

1. 标号

标号是一个标识符,是给指令所在地址取的名字。标号后必须跟冒号“:”。标号具有三种属性:段基值、偏移量及类型(NEAR 和 FAR)。

2. 操作助记符

操作助记符表示本指令的操作类型。它是指令语句中唯一不可缺少的部分。必要时可在指令助记符的前面加上一个或多个前缀,从而实现某些附加操作。

3. 操作数

操作数是参加指令运算的数据,可分为立即操作数、寄存器操作数、存储器操作数 3 种。有的指令不需要显式的操作数,如指令 XLAT;有的指令则需要不止一个的显式操作数,这时需用逗号“,”分隔两个操作数,如指令“ADD AX,BX”。

关于操作数,还有下面几个术语和概念应进一步说明,它们是常数、常量、变量、标号及偏移地址计数器 $ 。

(1) 常数

编程时已经确定其值,程序运行期间不会改变其值的数据对象称为常数。80x86 CPU 允许定义的常数类型有整数、字符串及实数。前面已经提到,在没有协处理器的环境中它不能处理实数,只能处理整数及字符串常数。整数可以有二进制、八进制、十进制、十六进制等不同表示形式。字符串常数可以用单引号括起一个或多个字符来组成。

(2) 常量

常量是用符号表示的常数。它是程序员给出的一个助记符作为一个确定值的标识,其值在程序执行过程中保持不变。常量可用伪指令语句 EQU 或“=”来定义。例如:

```
A  EQU  7 或 A =7 都可将常量 A 的值定义为常数 7
```

(3) 变量

编程时确定其初始值,程序运行期间可修改其值的数据对象称为变量。实际上,变量代表的就是存储单元。与存储单元有其地址和内容两重特性相对应,变量有变量名和值两个侧面,其中变量名与存储单元的地址相联系,变量的值则对应于存储单元的内容。

变量可由伪指令语句 DB、DW、DD 等来定义,通常定义在数据段和附加段。所谓定义变量,其实就是为数据分配存储单元,且对这个存储单元取一个名字,即变量名。变量名实际上就是存储单元的符号地址。存储单元的初值由程序员来预置。

变量有如下属性:

① 段基值:指变量所在段的段基值;

② 偏移地址:指变量所在的存储单元的段内偏移地址;

③ 类型:指变量所占存储单元的字节数。例如,用 DB 定义的变量类型属性为 BYTE(字节),用 DW 定义的变量类型属性为 WORD(字),用 DD 定义的变量类型属性为 DWORD(双字)等。

(4) 标号

需要时可给指令的地址取名字,标号就是指令地址的名字,也称指令的符号地址。标号定义在指令的前面(通常是左边),用冒号作为分隔符。标号只能定义在代码段中,它代表其后第一条指令的第一个字节的存储单元地址,用于说明指令在存储器中的存储位置,可作为转移类指令的直接操作数(转移地址)。例如,在下列指令序列中的 L 就是标号,它是 JNZ 指令的直接操作数(转移地址)。

```
   MOV  CX , 2
L: DEC  CX
   JNZ  L
```

标号有如下的属性:

① 段基值:即标号后面第一条指令所在代码段的段基值;

② 偏移地址:即标号后面第一条指令首字节的段内偏移地址;

③ 类型:也称距离属性,即标号与引用该标号的指令之间允许距离的远、近。近标号的类型属性为 NEAR(近),这样的标号只能被本段的指令引用;远标号的类型属性为 FAR(远),这样的标号可被任何段的指令引用。

(5) 偏移地址计数器 $

汇编程序在对源程序进行汇编的过程中,用偏移地址计数器 $ 来保存当前正在汇编的指令的偏移地址或伪指令语句中变量的偏移地址。用户可将 $ 用于自己编写的源程序中。

在每个段开始汇编时,汇编程序都将 $ 清为 0。以后,每处理一条指令或一个变量,$ 就增加一个值,此值为该指令或该变量所占的字节数。可见,$ 的内容就是当前指令或变量的偏移地址。

在伪指令中,$代表其所在地的偏移地址。例如,下列语句中的第一个$+4的偏移地址为A+4,第二个$+4的偏移地址为A+10。

```
A  DW  1,2,$+4,3,4,$+4
```

如果A的偏移地址是0074H,则汇编后,该语句中第一个$+4=(A+4)+4=(0074H+4)+4=007CH,第二个$+4=(A+10)+4=(0074H+0AH)+4=0082H。

于是,从A开始的字数据将依次为:

```
0001H,0002H,007CH,0003H,0004H,0082H
```

在机器指令中,$无论出现在指令的任何位置,都代表本条指令第一个字节的偏移地址。例如,"JZ $+6"的转向地址是该指令的首地址加上6,$+6还必须是另一条指令的首地址。

例如,在下述指令序列中:

```
    DEC  CX
    JZ   $+5
    MOV  AX,2
LAB: ...
```

因为$代表JZ指令的首字节地址,而JZ指令占2个字节,相继的MOV指令占3个字节,所以,在发生转移时,JZ指令会将程序转向LAB标号处的指令,且标号LAB可省。

5.2.4 伪指令语句

伪指令语句又称作指示性(directive)语句,它没有对应的机器指令,在汇编过程中不形成机器代码,这是伪指令语句与指令语句的本质区别。伪指令语句不要求CPU执行,而是让汇编程序在汇编过程中完成特定的功能,它在很大程度上决定了汇编语言的性质及其功能。伪指令语句的格式如图5.3所示。

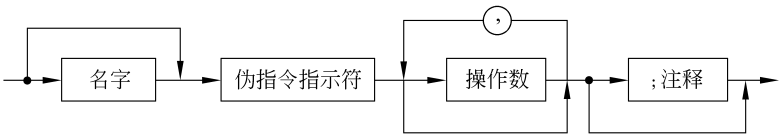

图5.3 伪指令语句的格式

从图5.3可以看出,伪指令语句与指令语句很相似,不同之处在于伪指令语句开始是一个可选的名字字段,它也是一个标识符,相当于指令语句的标号。但是名字后面不允许带冒号":",而指令语句的标号后面必须带冒号,这是两种语句形式上最明显的区别。伪指令语句很多,下面一一介绍。

1. 符号定义语句

汇编语言中所有的变量名、标号名、过程名、记录名、指令助记符、寄存器名等统称为“符号”,这些符号可由符号定义语句来定义,也可以定义为其他名字及新的类型属性。符号定义语句有三种,即 EQU 语句、=语句和 PURGE 语句。

(1) EQU 语句

EQU 语句给符号定义一个值,或定义为别的符号,甚至可定义为一条可执行的指令、表达式的值等。EQU 语句的格式为:

```
符号名  EQU  表达式
```

例如:

```
PORT1   EQU   78
PORT2   EQU   PORT1 + 2
COUNTER   EQU   CX
CBD EQU   DAA
```

这里,COUNTER 和 CBD 分别被定义为寄存器 CX 和指令助记符 DAA。

经 EQU 语句定义的符号不允许在同一个程序模块中重新定义。另外,EQU 语句只作为符号定义用,它不产生任何目标代码,也不占用存储单元。

(2) =语句

=语句与 EQU 语句功能类似,但此语句允许对已定义的符号重新定义,因而更灵活方便。其语句格式如下:

```
符号名 = 表达式
```

例如:

```
A = 6
A = 9
A = A + 2
```

(3) PURGE (取消)语句

PURGE 语句的格式为:

```
PURGE 符号名 1[,符号名 2[,…]]
```

PURGE 语句取消被 EQU 语句定义的符号名,然后即可用 EQU 语句再对该符号名重新定义。例如,可用 PURGE 语句实现如下操作:

```
A   EQU   7
PURGE   A                          ;取消 A 的定义
A   EQU   8                        ;重新定义
```

2. 数据定义语句

数据定义语句为一个数据项分配存储单元,用一个符号与该存储单元相联系,并可

以为该数据项提供一个任选的初始值。数据定义语句 DB、DW、DD、DQ、DT 可分别用来定义字节、字、双字、四字、十字节变量,并可用复制操作符 DUP 来复制数据项。例如:

```
FIRST  DB  27H
SECOND DD  12345678H
THIRD  DW  ? , 0A2H
FORTH  DB  2 DUP(2 DUP(1 , 2) , 3)
```

其中问号"?"表示相应存储单元没有初始值。上面定义的变量在存储器中的存放格式如图 5.4 所示。

数据项也可以写成字符串形式,但只能用 DB 和 DW 来定义,且 DW 语句定义的串只允许包含两个字符。例如:

```
ONE   DB   'AB'
TWO   DW   'AB','CD'
THREE  DB  'HELLO'
```

上述变量的存放格式如图 5.5 所示,注意 DB'AB'与 DW'AB'的存放格式不同。

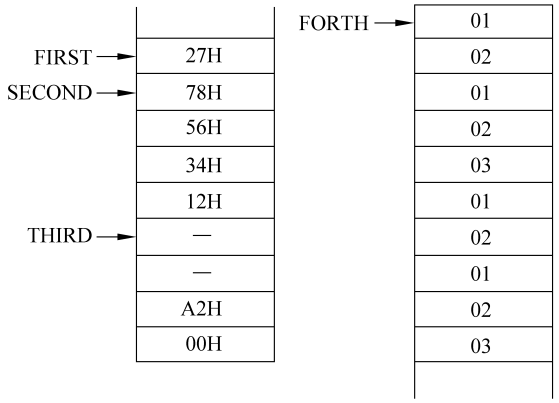

图 5.4　数据变量存储格式

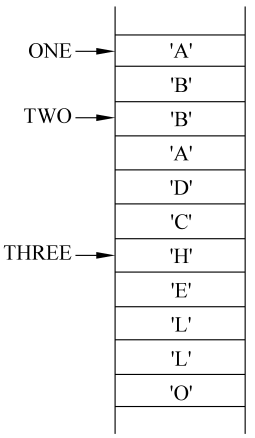

图 5.5　字符串变量存储格式

可以用 DW 语句把变量或标号的偏移地址存入存储器。也可以用 DD 语句把变量或标号的段基值和偏移地址都存入存储器,此时低位字存偏移地址,高位字存段基值。例如:

```
      VAR  DB  34H
LABL: MOV  AL , 04H
      ⋮
      PRV  DD  VAR
      PRL  DW  LABL
```

其存放格式如图 5.6 所示。

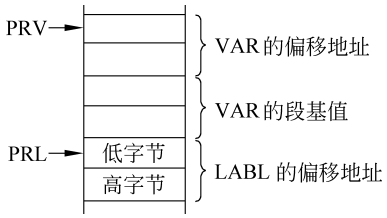

图 5.6　地址变量的存放格式

【例 5.1】 执行下列程序后，CX＝_____。

```
DATA  SEGMENT
  A  DW  1 , 2 , 3 , 4 , 5
  B  DW  5
DATA  ENDS
CODE  SEGMENT
  ASSUME  CS: CODE , DS: DATA
START:
  MOV  AX , DATA
  MOV  DS , AX
  LEA  BX , A
  ADD  BX , B
  MOV  CX , [BX]
  MOV  AH , 4CH
  INT  21H
CODE ENDS
  END  START
```

在例 5.1 所示程序中，当执行指令“LEA BX , A”时，将 A 相对数据段首址的偏移量 0 送入 BX 寄存器；执行指令“ADD BX , B”后，BX＝5；再执行指令“MOV CX ,[BX]”时，由于源操作数是寄存器间接寻址方式且该指令为字传送指令，因此应将相对数据段首址偏移量为 5 的字单元内容 0400 送入 CX 寄存器。所以上述程序执行完成后，CX＝0400。

3. 段定义语句

段定义语句指示汇编程序如何按段组织程序和使用存储器，主要有 SEGMENT、ENDS、ASSUME、ORG 等。下面分别予以介绍。

1）段开始语句 SEGMENT 和段结束语句 ENDS

一个逻辑段的定义格式如下：

```
段名 SEGMENT [定位类型] [组合类型] '类别'
     ⋮
段名 ENDS
```

整个逻辑段以 SEGMENT 语句开始，以 ENDS 语句结束。

其中段名是程序员指定的,SEGMENT 左边的段名与 ENDS 左边的段名必须相同。定位类型、组合类型和类别是赋给段名的属性,且都可以省略,若不省略则各项顺序不能错。

(1) 定位类型表示此段的起始地址边界要求,有 PAGE、PARA、WORD 和 BYTE 4 种方式,默认值为 PARA。它们的边界要求如下:

```
PAGE ××××  ××××  ××××  0000      0000
PARA ××××  ××××  ××××  ××××  0000
WORD ××××  ××××  ××××  ××××  ×××0
BYTE ××××  ××××  ××××  ××××  ××××
```

即分别要求地址的低 8 位为 0(页边界)、低 4 位为 0(节边界)、最低位为 0(字边界)及地址任意(字节边界)。

(2) 组合类型告诉连接程序本段与其他段的关系。有 NONE、PUBLIC、COMMON、STACK、MEMORY 和"AT 表达式"共 6 种,分别介绍如下:

① NONE 表示本段与其他段逻辑上不发生关系,每段都有自己的段基地址。这是默认的组合类型。

② PUBLIC 告诉连接程序首先把本段与用 PUBLIC 说明的同名同类别的其他段连接成一个段,所有这些段用一个相同的段基地址。

③ COMMON 表示本段与同名同类别的其他段共用同一段基地址,即同名同类段相重叠,段的长度是其中最长段的长度。

④ STACK 表示本段是堆栈段,连接方式同 PUBLIC。被连接程序中必须至少有一个堆栈段,有多个堆栈段时采用覆盖方式进行组合。连接后的段基地址在 SS 寄存器中。

⑤ MEMORY 表示该段在连接时被放在所有段的最后(最高地址)。若有几个 MEMORY 组合类型的段,汇编程序认为所遇到的第一个为 MEMORY,其余为 COMMON 型。

⑥ "AT 表达式"告诉连接程序把本段装在表达式的值所指定的段基地址处。例如:"AT 1234H"表示该段的段基地址为 12340H。

(3) 类别是用单引号括起来的字符串,可以是长度不超过 40 个字符的串。连接程序只使相同类别的段发生关联。典型的类别如'STACK'、'CODE'、'DATA'等。

2) 段分配语句 ASSUME

段分配语句 ASSUME 用来告诉汇编程序当前哪 4 个段分别被定义为代码段、数据段、堆栈段和附加段,以便对使用变量或标号的指令生成正确的目标代码。其格式是:

```
ASSUME 段寄存器: 段名[,段寄存器: 段名,…]
```

注意,使用 ASSUME 语句只是告诉汇编程序有关段寄存器将被设定为哪个段的段基值,而段基值的真正设定必须通过给段寄存器赋值的指令语句来完成。例如:

```
CODE  SEGMENT
```

```
ASSUME  DS: DATA , ES: DATA , CS: CODE , SS: STACK
   MOV  AX , DATA
   MOV  DS , AX
   MOV  ES , AX
   MOV  AX , STACK
   MOV  SS , AX
        ⋮
```

段寄存器 CS 的值是由系统设置的,因此程序中不必进行赋值。

3) 定位语句 ORG

定位语句 ORG 的格式为:

```
ORG  表达式
```

用来指出其后的程序块或数据块从表达式之值作为存放的起始地址(偏移地址)。若没有 ORG 语句则从本段的起始地址开始存放。

4. 过程定义语句

过程是程序的一部分,可被主程序调用。每次可调用一个过程,当过程中的指令执行完后,控制返回调用它的地方。

利用过程定义语句可以把程序分成若干独立的程序模块,便于理解、调试和修改。过程调用对模块化程序设计是很方便的。

8086 系统中过程调用和返回指令是 CALL 和 RET,可分为段内和段间操作两种情况。段间操作把过程返回地址的段基值和偏移地址都压栈(通过执行 CALL 指令实现)或退栈(通过执行 RET 指令实现),而段内操作则只把偏移地址压栈或退栈。过程定义语句的格式为:

```
过程名 PROC [NEAR/FAR]
        ⋮
过程名 ENDP
```

其中,过程名是一个标识符,是给被定义过程取的名字。过程名像标号一样,有 3 重属性:段基值、偏移地址和距离属性(NEAR 或 FAR)。

NEAR 或 FAR 指明过程的距离属性。NEAR 过程只允许段内调用,FAR 过程则允许段间调用。默认时为 NEAR 过程。

过程内部至少要设置一条返回指令 RET,以作为过程的出口。允许一个过程中有多条 RET 指令,而且可以出现在过程的任何位置上。

5. 其他伪指令语句

除了上面已介绍的伪指令语句外,汇编语言程序中还有一些其他的伪指令语句。

① 模块开始伪指令语句 NAME 指明程序模块的开始,并指出模块名,其格式为:

```
NAME  模块名
```

该语句在一个程序中不是必需的,可以不写。

② 模块结束伪指令语句 END 标志整个源程序的结束,汇编程序汇编到该语句时结束。其格式为:

```
END  [标号]
```

其中标号是程序中第一个指令性语句(或第一条指令)的符号地址。注意,当程序由多个模块组成时,只需在主程序模块的结束语句(END 语句)中写出该标号,其他子程序模块的结束语句中则可以省略。

③ 对准伪指令语句 EVEN 要求汇编程序将下一语句所指向的地址调整为偶地址,使用时直接用伪指令名 EVEN 就可以了。例如,下述 EVEN 伪指令将把字数组 ARY 调整到偶地址开始处。

```
EVEN
ARY  DW  100 DUP(?)
```

又如,下述伪指令序列:

```
ORG  1000H
A  DB  12H , 34H , 56H
EVEN
B  DB  78H
```

其中,ORG 1000H 将 A 的偏移地址指定为 1000H,从 A 开始存放 3 个字节变量,占用地址 1000H、1001H 和 1002H,B 的偏移地址本应是 1003H,但 EVEN 伪指令会将其调整为偶数地址 1004H。

说明:由于 80x86 系统在存储器结构上所采用的设计技术,使得对于 8086 这样的 16 位 CPU,如果从偶地址开始访问一个字,可以在一个总线周期内完成;但如果从奇地址开始访问一个字,则由于对两个字节必须分别访问,所以要用两个总线周期才能完成。同样,对于 80386 以上的 32 位 CPU,如果从双字边界(地址为 4 的倍数)开始访问一个双字数据,可以在一个总线周期内完成,否则需用多个总线周期。因此,在安排存储器数据时,为了提高程序的运行速度,最好将字型数据从字边界(偶地址)开始存放,双字数据从双字边界开始存放。对准伪指令 EVEN 就是专门为实现这样的功能而设置的。

④ 默认基数伪指令语句 RADIX,其作用在 5.2.2 节讲述数的表示时已有说明,其格式为:

```
RADIX  表达式
```

⑤ LABEL 伪指令语句可用来给已定义的变量或标号取一个别名,并重新定义它的属性,以便于引用。其格式为:

```
变量名/标号名 LABEL  类型
```

对于变量名,类型可为 BYTE、WORD、DWORD、QWORD、TBYTE 等。对于标号名,类型可为 NEAR 和 FAR。例如:

```
VARB   LABEL   BYTE          ;给下面的变量 VARW 取了一个新名字 VARB,并赋予另外的属性 BYTE
VARW   DW   4142H , 4344H
PTRF LABEL   FAR             ; 给下面的标号 PTRN 取了一个新名字 PTRF, 并赋予另外的属性 FAR
PTRN:   MOV   AX , [DI]
```

注意,LABEL 伪指令的功能与前述 THIS 伪指令类似,两者均不为所在语句的符号分配内存单元,区别是使用 LABEL 可以直接定义,而使用 THIS 伪指令则需要与 EQU 或“=”连用。

⑥ COMMENT 伪指令语句用于书写大块注释,其格式为:

```
COMMENT 定界符 注释 定界符
```

其中定界符是自定义的任何非空字符。例如:

```
COMMENT /
    注释文
        /
```

⑦ TITLE 伪指令语句为程序指定一个不超过 60 个字符的标题,以后的列表文件会在每页的第一行打印这个标题。SUBTTL 伪指令语句为程序指定一个小标题,打印在每一页的标题之后。格式如下:

```
TITLE     标题
SUBTTL    小标题
```

⑧ PAGE 伪指令语句指定列表文件每页的行数(10～255)和列数(60～132),默认值是每页 66 行 80 列。其格式如下:

```
PAGE   行数,列数
```

⑨ 模块连接伪指令语句主要解决多模块的连接问题。一个大的程序往往要分模块来完成编码、调试的工作,然后再整体连接和调试。它们的格式如下:

```
PUBLIC   符号名[,符号名,…]
EXTERN   符号名: 类型[,符号名: 类型,…]
INCLUDE   模块名
组名   GROUP   段名[,段名,…]
```

其中符号名可以是变量名、标号、过程名、常量名等。

以变量名为例,一个程序模块中用 PUBLIC 伪指令定义的变量可由其他模块引用,否则不能被其他模块引用;在一个模块中引用其他模块中定义的变量必须在本模块用 EXTERN 伪指令进行说明,而且所引用的变量必须是在其他模块中用 PUBLIC 伪指令定义的。换句话说,如果要在“使用模块”中访问其他模块中定义的变量,除要求该变量在其“定义模块”中定义为 PUBLIC 类型外,还需在“使用模块”中用 EXTERN 伪指令说明该变量,以通知汇编器该变量是在其他模块中定义的。

例如,一个应用程序包括 A、B、C 三个程序模块,而 VAR 是定义在模块 A 数据段中的一个变量,其定义格式如下:

```
PUBLIC  VAR
```

由于 VAR 被定义为 PUBLIC,所以在模块 B 或 C 中也可以访问这个变量,但必须在模块 B 或 C 中用 EXTERN 伪指令说明这个变量。其格式如下所示:

```
EXTERN  VAR: Type
```

注意,汇编器并不能检查变量类型 Type 和原定义是否相同,这需要编程者自己维护。

INCLUDE 伪指令告诉汇编程序把另外的模块插入本模块该伪指令处一起汇编,被插入的模块可以是不完整的。

GROUP 伪指令告诉汇编程序把其后指定的所有段组合在一个 64K 的段中,并赋予一个名字——组名。组名与段名不可相同。

5.2.5 宏指令

在汇编语言源程序中,有的程序段可能要多次使用,为了使在源程序中不重复书写这一程序段,可以用一条宏指令来代替,在汇编时由汇编程序进行宏扩展而产生所需要的代码。本节专门讨论宏指令的概念及相关伪指令语句。

1. 宏定义语句

宏指令的使用过程就是宏定义、宏调用和宏扩展的 3 个过程,下面分别予以说明。

(1) 宏定义

宏定义由伪指令 MACRO 和 ENDM 来定义,其语句格式为:

```
宏指令名  MACRO  [形式参数,形式参数,…]
            ⋮   } 宏体
          ENDM
```

其中宏指令名是一个标识符,是程序员给该宏指令取的名字。MACRO 是宏定义的开始符,ENDM 是宏定义的结束符,两者必须成对出现。注意,ENDM 左边不需加宏指令名。MACRO 和 ENDM 之间的指令序列称为宏定义体(简称宏体),即要用宏指令来代替的程序段。

宏指令具有接受参数的能力,宏体中使用的形式参数必须在 MACRO 语句中出现。形式参数可以没有,也可以有多个。当有两个以上参数时,需用逗号隔开。在宏指令被调用时,这些参数将被给出的一些名字或数值所替代,这里的名字或数值称为“实参数”。实际上,形式参数只是指出了在何处以及如何使用实参数的方法。形式参数的使用使宏指令在参数传递上更加灵活。

例如,移位宏指令 SHIFT 可定义如下:

```
SHIFT  MACRO  X
```

```
        MOV  CL , X
        SAL  AL , CL
        ENDM
```

其中 SHIFT 为宏指令名，X 为形式参数。

(2) 宏调用

经过宏定义后，在源程序中的任何位置都可以直接使用宏指令名来实现宏指令的引用，称为宏调用。它要求汇编程序把宏定义体(程序段)的目标代码复制到调用点。如果宏定义是带参数的，就用宏调用时的实参数替代形式参数，其位置一一对应。宏调用的格式为：

```
宏指令名  [实参数,实参数,…]
```

其中实参数将一一对应地替代宏定义体中的形式参数。同样，当有两个以上参数时，需用逗号隔开。注意，并不要求形式参数个数与实在参数一样多。若实在参数多于形式参数，多余的将被忽略；若实在参数比形式参数少，则多余的形式参数变为空(NULL)。

例如，调用前面定义的宏指令 SHIFT 时，可写为：

```
SHIFT  6 ;用参数 6 替代宏体中的参数 X,从而实现宏调用
```

(3) 宏扩展

在汇编宏指令时，宏汇编程序将宏体中的指令插入到源程序宏指令所在的位置上，并用实参数代替形式参数，同时在插入的每一条指令前加一个“+”，这个过程称为宏扩展。

例如，上例的宏扩展为：

```
+ MOV  CL , 6
+ SAL  AL , CL
```

又如，假设有下列的宏定义：

```
SHIFT  MACRO  X , Y
       MOV  CL , X
       SAL  Y , CL
       ENDM
```

那么宏调用“SHIFT 4，AL”将被扩展为：

```
+ MOV  CL , 4
+ SAL  AL , CL
```

另外，形式参数不仅可以出现在指令的操作数部分，而且可以出现在指令操作助记符的某一部分，但这时需在相应形式参数前加宏操作符 &，宏扩展时将把 & 前后两个符号合并成一个符号。例如，对于下面的宏定义：

```
SHIFT  MACRO  X , Y , Z
       MOV  CL , X
       S&Z  Y , CL
       ENDM
```

则下面两个宏调用

```
SHIFT  4 , AL , AL
SHIFT  6 , BX , AR
```

将被宏扩展为如下的指令序列：

```
+ MOV  CL , 4
+ SAL  AL , CL
+ MOV  CL , 6
+ SAR  BX, CL
```

实际上，这个例子中的宏指令 SHIFT 带上合适的参数可以对任一个寄存器进行任意的移位操作(算术/逻辑左移，算术右移，逻辑右移)，而且可以移位任意指定的位数。

2. 局部符号定义语句

当含有标号或变量名的宏指令在同一个程序中被多次调用时，根据宏扩展的功能，汇编程序会把它扩展成多个同样的程序段，也就会产生多个相同的符号名，这就违反了汇编程序对名字不能重复定义的规定，从而出现错误。局部符号定义语句用来定义仅在宏定义体内引用的符号，这样可以防止在宏扩展时引起符号重复定义的错误。其格式为：

```
LOCAL  符号[,符号…]
```

汇编时，对 LOCAL 伪指令说明的符号每宏扩展一次便建立一个唯一的符号，以保证汇编生成名字的唯一性。

需要注意的是，LOCAL 语句必须是 MACRO 伪指令后的第一个语句，且与 MACRO 之间不能有注释等其他内容。

3. 注销宏定义语句

该语句用来取消一个宏指令定义，然后就可以重新定义。其格式为：

```
PURGE  宏指令名[,宏指令名…]
```

由以上对宏指令的介绍可以看出，宏指令与子程序有某些相似之处，但两者也有区别，表现在以下几个方面：

① 在处理的时间上，宏指令是在汇编时进行宏扩展，而子程序是在执行时由 CPU 处理的。

② 在目标代码的长度上，由于采用宏指令方式时的宏扩展是将宏定义体原原本本地插入到宏指令调用处，所以它并不缩短目标代码的长度，而且宏调用的次数越多，目标代码长度越长，所占内存空间也就越大；而采用子程序方式时，若在一个源程序中多次调用同一个子程序，则在目标程序的主程序中只有调用指令的目标代码，子程序的目标代码在整个目标程序中只有一段，所以采用子程序方式可以缩短目标代码的长度。

③ 子程序每次执行都要进行返回地址的保护和恢复，因此延长了执行时间，而宏指令方式不会增加这样的时间开销。

④ 两者在传递参数的方式上也有所不同。宏指令是通过形式参数和实参数的方式来传递参数的，而子程序方式则是通过寄存器、堆栈或参数表的方式来进行参数的传递。可以根据使用需要在子程序方式和宏指令方式之间进行选择。

一般来说，当要代替的程序段不很长，执行速度是主要矛盾时，通常采用宏指令方式；当要代替的程序段较长，额外操作（返回地址的保存、恢复等）所增加的时间已不明显，而节省存储空间是主要矛盾时，通常宜采用子程序方式。

5.2.6 简化段定义

MASM 5.0 以上版本的宏汇编程序提供了简化段定义伪指令，Borland 公司的 TASM 也支持简化段。简化段伪指令根据默认值来提供段的相应属性，采用的段名和属性符合 Microsoft 高级语言的约定。简化段使编写汇编语言程序更为简单、不易出错，且更容易与高级语言相连接。表 5.4 给出了简化段伪指令的名称、格式及操作描述。其中的伪指令 MODEL 指定的各种存储模式及其使用环境如表 5.5 所示。关于简化段的其他详细内容可参阅相关文献。

表 5.4 简化段伪指令

名　称	格　式	操作描述
.MODEL	.MODEL mode	指定程序的内存模式为 mode，mode 可取 Tiny、Small、Medium、Compact、Large、Huge、Flat
.CODE	.CODE[name]	代码段
.DATA	.DATA	初始化的近数据段
.DATA?	.DATA?	未初始化的近数据段
.STACK	.STACK[size]	堆栈段，大小为 size 字节，默认值为 1K 字节
.FARDATA	.FARDATA[name]	初始化的远数据段
.FARDATA?	.FARDATA? [name]	未初始化的远数据段
.CONST	.CONST	常数数据段，在执行时无须修改的数据

表 5.5　存储模式

存储模式	使用条件和环境
Tiny	用来建立 MS-DOS 的.COM 文件,所有的代码、数据和堆栈都在同一个 64KB 段内,DOS 系统支持这种模式
Small	建立代码和数据分别用一个 64KB 段的.exe 文件,MS-DOS 和 Windows 支持这种模式
Medium	代码段可以有多个 64KB 段,数据段只有一个 64KB 段,MS-DOS 和 Windows 支持这种模式
Compact	代码段只有一个 64KB 段,数据段可以有多个 64KB 段,MS-DOS 和 Windows 支持这种模式
Large	代码段和数据段都可以有多个 64KB 段,但是单个数据项不能超过 64KB,MS-DOS 和 Windows 支持这种模式
Huge	同 Large,并且数据段里面的一个数据项也可以超过 64KB, MS-DOS 和 Windows 支持这种模式
Flat	代码和数据段可以使用同一个 4GB 段,Windows 32 位程序使用这种模式

下面给出一个使用简化段定义的程序例子。

```
.MODEL Small                                    ;定义存储模式为 Small
.DATA                                           ;数据段
    STRING DB 'Hello World !', 0AH , 0DH , '$ '
.STACK 100H                                     ;100H 字节的堆栈段
.CODE                                           ;代码段
    ASSUME CS: _TEXT , DS: _DATA , SS: STACK
START PROC FAR
    MOV AX , _DATA
    MOV DS , AX
    MOV DX , OFFSET STRING
    MOV AH , 9
    INT 21H
    MOV AH , 4CH
    INT 21H
START ENDP
    END START
```

5.3　ROM BIOS 中断调用和 DOS 系统功能调用

80x86 微机系统通过 ROM BIOS 和 DOS 提供了丰富的系统服务子程序,用户可以很容易地调用这些系统服务软件,给程序设计带来很大方便。

5.3.1　ROM BIOS 中断调用

80x86 微型计算机的系统板中装有 ROM,其中从地址 0FE00H 开始的 8KB 为 ROM　BIOS(Basic Input Output System)。驻留在 ROM 中的 BIOS 例行程序提供了

系统加电自检、引导装入以及对主要 I/O 接口控制等功能。其中对 I/O 接口的控制，主要是指对键盘、磁带、磁盘、显示器、打印机、异步串行通信接口等的控制。此外，BIOS 还提供了最基本的系统硬件与软件间的接口。

ROM BIOS 为程序员提供了很大的方便。程序员可以不必了解硬件的详细接口特性，而是通过直接调用 BIOS 中的例行程序来完成对主要 I/O 设备的控制管理。BIOS 由许多功能模块组成，每个功能模块的入口地址都在中断向量表中。通过软件中断指令“INT n”可以直接调用这些功能模块。CPU 响应中断后，把控制权交给指定的 BIOS 功能模块，由它提供相应服务。

BIOS 中断调用的入口参数和出口参数均采用寄存器传送。若一个 BIOS 子程序能完成多种功能，则用功能号来加以区分，并将相应的功能号预置于 AH 寄存器中。BIOS 中断调用的基本方法如下：

(1) 将所要调用功能的功能号送入 AH 寄存器；

(2) 根据所调用功能的规定设置入口参数；

(3) 执行“INT 中断号”指令，进入相应的服务子程序；

(4) 中断服务子程序执行完毕后，可按规定取得出口参数。

【例 5.2】 利用“INT 1AH”的 1 号功能将时间计数器的当前值设置为 0。

```
MOV  AH , 1                                  ;设置功能号
MOV  CX , 0        ;
MOV  DX , 0        ;  设置入口参数
INT  1AH                                     ;BIOS 中断调用
```

5.3.2 DOS 系统功能调用

BIOS 常驻在系统板的 ROM 中，独立于任何操作系统。DOS 则以 BIOS 为基础，为用户提供了一组可以直接使用的服务程序。这组服务程序共用 21H 号中断入口，也以功能号来区分不同的功能模块。这一组服务程序就称为 DOS 系统功能调用。

DOS 系统功能调用的方法与 BIOS 中断调用类似，只是中断号固定为 21H。

【例 5.3】 利用 6 号 DOS 系统功能调用在屏幕上输出字符“$”。

```
MOV  AH , 6                                  ;设置功能号为 6
MOV  DL , '$'                                ;设置入口参数
INT  21H                                     ;DOS 系统功能调用
```

虽然 BIOS 比 DOS 更接近硬件，但机器启动时 DOS 层功能模块是从系统硬盘装入内存的，它的功能比 BIOS 更齐全、完整，其主要功能包括文件管理、存储管理、作业管理及设备管理等。DOS 层子程序是通过 BIOS 来使用设备的，从而进一步隐蔽了设备的物理特性及其接口细节，所以在调用系统功能时总是先采用 DOS 层功能模块，如果这层内容达不到要求，再进一步考虑选用 BIOS 层的子程序。

关于 DOS 功能调用和 BIOS 中断调用的详细情况可以参见附录 A 和附录 B 或

DOS/BIOS 手册,这里不再一一说明。

5.4 汇编语言程序的上机调试

编好汇编语言源程序后,要想使它完成预定功能,还须经过建立源文件、汇编、连接等过程。如果出现错误,还要进行跟踪调试。

5.4.1 建立源文件

上机开始,首先要使用编辑程序完成源程序的建立和修改工作。编辑程序可分为行编辑程序和全屏幕编辑程序。现在一般都使用全屏幕编辑程序。DOS 5.0 以上版本提供了全屏幕编辑软件 EDIT。

启动 EDIT 的常用命令格式如下:

```
EDIT [文件名]
```

其中文件名是可选的,若为汇编语言源文件,其扩展名必须是.ASM。

例如,输入如下命令行:

```
C:\tools>EDIT EXAM.ASM (↙)
```

即可进入编辑源文件 EXAM.ASM 的过程,若该文件不存在时则开始建立它。用 File 选项的存盘功能可保存文件;最后可通过 File 的 Exit 选项退出 EDIT。

另外,如果 EDIT 是从 Windows 环境下的 MS-DOS 方式进入的,则在 DOS 提示符后面输入 EXIT 即可退出 DOS 返回 Windows。

5.4.2 汇编

经过编辑程序建立和修改的汇编语言源程序(扩展名为 .ASM)要在机器上运行,必须先由汇编程序(汇编器)把它转换为二进制形式的目标程序。汇编程序的主要功能是对用户源程序进行语法检查并显示出错信息,对宏指令进行宏扩展,把源程序翻译成机器语言的目标代码。经汇编程序汇编后的程序可建立三个文件:一是扩展名为.OBJ 的目标文件;二是扩展名为.LST 的列表文件,它把源程序和目标程序制表以供使用;三是扩展名为.CRF 的交叉索引文件,它给出源程序中的符号定义和引用的情况。在 DOS 提示符下,输入 MASM 并回车,屏幕的显示输出和相应的输入操作如下(以用汇编器 MASM V 5.00 汇编 5.2.1 示例程序 EXAM.ASM 为例):

```
C:\tools>MASM (↙)
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985,1987. All rights reserved.
Source filename[.ASM]: EXAM (↙)
```

```
Object  filename[EXAM.OBJ]:   (↙)
Source  Listing[NUL.LST]:     (↙)
Cross - Reference[NUL.CRF]:   (↙)
    50972 + 416020  Bytes  symbol  space  free
        0  Warning  Errors
        0  Severe   Errors
```

以上各处需要输入的地方除源程序文件名 EXAM 必须输入外，其余都可直接按 Enter 键。其中目标文件 EXAM.OBJ 是必须要产生的；NUL 表示在默认的情况下不产生相应的文件，若需要产生则应输入文件的名字部分，其扩展名将按默认情况自动产生。若汇编过程中出错，将列出相应的出错行号和出错提示信息以供修改。显示的最后部分给出“警告错误数”(Warning Errors)和“严重错误数”(Severe Errors)。

另外，也可以直接用命令行的形式一次顺序给出相应的 4 个文件名，具体格式如下：

```
C:\ tools > MASM 源文件名,目标文件名,列表文件名,交叉索引文件名;
```

命令行中的 4 个文件名均不必给出扩展名，汇编程序将自动按默认情况处理。若不想全部提供要产生文件的文件名，则可在不想提供文件名的位置用逗号隔开；若不想继续给出剩余部分的文件名，则可用分号结束。例如，对于源文件 EXAM.ASM，若只想产生目标文件和列表文件，则给出的命令行及相应的显示信息如下：

```
C:\tools > MASM EXAM , EXAM , EXAM; (↙) [或 C:\tools > MASM EXAM , , EXAM; (↙) ]
    Microsoft (R) Macro Assembler Version 5.00
    Copyright (C) Microsoft Corp 1981 - 1985, 1987. All rights reserved.
        50970 + 416022 Bytes symbol space free
            0 Warning Errors
            0 Severe Errors
```

产生的列表文件 EXAM.LST 的详细内容如下：

```
Microsoft (R) Macro Assembler Version 5.00                  12/4/7
Page     1 - 1
       1  0000                 DATA  SEGMENT
       2  0000  3A07               NUM  DW  0011101000000111B
       3  0002  54 68 65 20 72 65 73 NOTES  DB  'The result is:', '$'
       4        75 6C 74 20 69 73 3A
       5        24
       6  0011                 DATA  ENDS
       7  0000                 STACK  SEGMENT  STACK
       8  0000  0032[               STA  DB  50 DUP (?)
       9    ??
      10             ]
      11
      12  0032                 STACK  ENDS
      13  0000                 CODE  SEGMENT
      14                         ASSUME  CS:CODE , DS:DATA , SS:STACK
      15  0000                   BEGIN:
      16  0000  B8 ---- R            MOV  AX , DATA
```

```
      17  0003  8E D8              MOV  DS , AX
      18  0005  BA 0002 R          MOV  DX , OFFSET  NOTES
      19  0008  B4 09              MOV  AH , 9H
      20  000A  CD 21              INT  21H
      21  000C  8B 1E 0000 R       MOV  BX , NUM
      22  0010  B5 04              MOV  CH , 4
      23  0012                  ROTATE:
      24  0012  B1 04              MOV  CL , 4
      25  0014  D3 C3              ROL  BX , CL
      26  0016  8A C3              MOV  AL , BL
      27  0018  24 0F              AND  AL , 0FH
      28  001A  04 30              ADD  AL , 30H
      29  001C  3C 39              CMP  AL , '9'
      30  001E  7E 02              JLE  DISPLAY
      31  0020  04 07              ADD  AL , 07H
      32  0022                  DISPLAY:
      33  0022  8A D0              MOV  DL , AL
      34  0024  B4 02              MOV  AH , 2
      35  0026  CD 21              INT  21H
      36  0028  FE CD              DEC  CH
      37  002A  75 E6              JNZ  ROTATE
      38  002C  B8 4C00            MOV  AX , 4C00H
      39  002F  CD 21              INT  21H
      40  0031              CODE  ENDS
      41                        END  BEGIN
Microsoft (R) Macro Assembler Version 5.00                  12/4/7
Symbols-1
Segments and Groups:
    Name                    Length    Align    Combine Class
CODE . . . . . .            0031      PARA     NONE
DATA . . . . . .            0011      PARA     NONE
STACK  . . . . . .          0032      PARA     STACK
Symbols:
    Name                    Type      Value    Attr
BEGIN  . . . . . .          L NEAR    0000     CODE
DISPLAY  . . . . . .        L NEAR    0022     CODE
NOTES  . . . . . .          L BYTE    0002     DATA
NUM  . . . . . .            L WORD    0000     DATA
ROTATE . . . . . .          L NEAR    0012     CODE
STA  . . . . . .            L BYTE    0000     STACK     Length = 0032
@FILENAME  . . . . . .      TEXT EXAM
     36 Source  Lines
     36 Total   Lines
     11 Symbols
  49772 + 416996 Bytes symbol space free
      0 Warning Errors
      0 Severe  Errors
```

上述列表文件 EXAM.LST 共分两部分,在第一部分中,分 4 列对照列出了源程序语

句和目标程序代码。其中,左边第一列是行号,第二列是目标程序的段内偏移地址,第三列是目标程序代码,第四列是源程序语句。只有当要求生成.CRF 文件时才会在.LST 文件中给出行号,且.LST 文件中的行号可能和源程序文件中的行号不一致。在.LST 文件中,所有的数都是十六进制数,"R"的含义是"可再定位的"或"浮动的",含有"R"的行是汇编程序不能确定的行,"R"左边的数需要由连接程序确定或者修改。

在第二部分中,给出了每个段的名称、长度、定位类型、组合类型和"类别"类型,随后又给出了程序员定义的其他名字的类型、值及其段归属。其中,定位类型的 PARA 表示该段开始的偏移地址为 0000H;组合类型 NONE 表示各个段之间不进行组合;组合类型 STACK 表示本段按堆栈段的要求进行组合;L NEAR 表示是近标号;L BYTE 表示是字节变量;L WORD 表示是字变量;F PROC 表示是远过程,等等。

5.4.3 连接

虽然经过汇编程序处理而产生的目标文件已经是二进制文件了,但它还不能直接在机器上运行,还必须经连接程序连接后才能成为扩展名为.EXE 的可执行文件。这主要是因为汇编后产生的目标文件中还有需再定位的地址要在连接时才能确定下来,另外连接程序还有一个更重要的功能就是可以把多个程序模块连接起来形成一个装入模块,此时每个程序模块中可能有一些外部符号的值是汇编程序无法确定的,必须由连接程序来确定。因此连接程序需完成的主要功能是:

(1) 找到要连接的所有模块;

(2) 对要连接的目标模块的段分配存储单元,即确定段地址;

(3) 确定汇编阶段不能确定的偏移地址值(包括需再定位的地址及外部符号所对应的地址);

(4) 构成装入模块,将其装入内存。

使用连接程序的一般操作步骤是,在 DOS 提示符下,打入连接程序名 LINK,运行后会先显示版本信息,然后依次给出 4 个提示信息请求输入,如下所示:

```
C:\tools > LINK  (↙)
Microsoft (R) Overlay  Linker Version 3.60
Copyright (C) Microsoft Corp 1983 - 1987.  All  rights  reserved.
Object Modules [.OBJ]: EXAM (↙)
Run File [EXAM.EXE]:  (↙)
List File [NUL.MAP ]: EXAM (↙)
Libraries [.LIB]:  (↙)
```

给出的 4 个提示信息分别要求输入目标文件名、可执行文件名、内存映像文件名和库文件名。注意,目标文件(.OBJ 文件)和库文件(.LIB 文件)是连接程序的两个输入文件,而可执行文件(.EXE 文件)和内存映像文件(.MAP 文件)是它的两个输出文件。

第一个提示应该用前面汇编程序产生的目标文件名回答(不需输入扩展名.OBJ,这里是 EXAM),也可以用加号"+"来连接多个目标文件;第二个提示要求输入将要产生

的可执行文件名,通常可直接按 Enter 键,表示确认系统给出的默认文件名;第三个是产生内存映像文件的提示,默认情况为不产生,若需要则应输入文件名(这里是 EXAM);第四个是关于库文件的提示,通常直接按 Enter 键,表示不使用库文件。

回答上述提示后,连接程序开始连接,若连接过程中有错会显示错误信息。这时需修改源程序,再重新汇编、连接,直到无错。

说明:若用户程序中没有定义堆栈或虽然定义了堆栈但不符合要求时,会在连接时给出警告信息:"LINK:Warning L4021: no stack segment"。但该警告信息不影响可执行程序的生成及正常运行,运行时会自动使用系统提供的默认堆栈。

按上述对四个提示的回答,目标文件 EXAM.OBJ 连接后将在当前目录下产生 EXAM.EXE 和 EXAM.MAP 两个文件。其中的.MAP 文件是连接程序的列表文件,它给出每个段在内存中的分配情况。EXAM.MAP 文件的具体内容如下:

```
Start    Stop     Length  Name     Class
00000H   00010H   0011H   DATA
00020H   00051H   0032H   STACK
00060H   00090H   0031H   CODE
Program  entry    point   at  0006:0000
```

其中 Start 列是段起始地址,Stop 列是段结束地址,Length 列是段长度,Name 列是段名。最后一行给出了该程序执行时的入口地址。

5.4.4 运行

经连接生成.EXE 文件后,即可直接输入该文件名来运行程序(不需输入扩展名.EXE)。对于前面的例子,输入的命令行及程序的相应显示输出信息如下:

```
C:\tools> EXAM  (↙)
  The  result  is: 3A07
```

如果得不到正确结果或程序中未安排显示输出的操作,则可通过调试工具 DEBUG 来进行调试或跟踪检测。

5.4.5 调试

汇编语言源程序经汇编、连接成功后,并不一定就能正确地运行,程序中还可能存在各种逻辑错误,这时就需要用调试程序来找出这些错误并改正。

DEBUG 程序是 DOS 系统提供的一种基本的调试工具,其具体使用方法可参见附录 C。此外还有 Code View,Turbo Debugger 等调试工具,它们的功能比 DEBUG 要强,使用起来也更方便。

这些工具软件一般都有较完善的联机帮助功能,因此这里仅简要介绍 DEBUG 程序中的几个常用命令。

1) DEBUG 的启动

下面以 EXAM1.EXE(例 5.1 源程序生成的可执行代码)的调试情况为例,来具体说明 DEBUG 程序的启动过程,输入的命令行及相应的显示输出如下所示:

```
C:\tools> DEBUG  EXAM1.EXE  (↙)
-
```

此时,DEBUG 将 EXAM1.EXE 装入内存并给出提示符"-",等待输入各种操作命令。

另外,若在 Windows 图形界面下启动 DEBUG,只需双击 DEBUG 图标,当屏幕上出现 DEBUG 提示符"-"后,紧接其后输入 N 命令及被调试程序的文件名,然后输入 L 命令,装入后即可进行相关调试。

例如,若在 Windows 环境下装入 EXAM1.EXE 进行调试,在双击 DEBUG 图标并出现 DEBUG 提示符"-"后,再做如下两步操作即可完成 DEBUG 启动过程。

```
-N  EXAM1.EXE  (↙)
-L  (↙)
-
```

2) DEBUG 的主要命令

DEBUG 命令是在提示符"-"下由键盘输入的。每条命令以单个字母的命令符开头,其后是命令的操作参数(如果有)。命令符与操作参数之间用空格隔开,操作参数与操作参数之间也用空格隔开,所有的输入/输出数据都是十六进制形式,不用加字母"H"作后缀。下面分别介绍 DEBUG 命令中最常用的 R、U、T、G、D 和 Q 等命令,有关这些命令和其他命令的详细用法请见附录 C 或其他参考资料。

(1) R 命令

用 R 命令可以显示或修改 CPU 内部各寄存器的内容及标志位的状态,并给出下一条要执行指令的机器码及其汇编形式,同时给出该指令首字节的逻辑地址(段基值:偏移量)。例如,如果在装入 EXAM1.EXE 文件后,第一次输入的命令是"R"命令,则会显示:

```
-R
AX=0000  BX=0000  CX=0080  DX=0000  SP=0020  BP=0000  SI=0000  DI=0000
DS=10DE  ES=10DE  SS=10EF  CS=10F1  IP=0000  NV  UP  DI  PL  NZ  NA  PO  NC
10F1:0000  B8EE10  MOV  AX,10EE
```

其中的 NV、UP、DI、PL、NZ、NA、PO 和 NC 表示标志寄存器各位(不包括 TF 位)的当前值,如表 5.6 所示。显示出的各个段寄存器的内容在不同的计算机上也可能不同。

表 5.6 标志寄存器中各标志位值的符号表示

标 志 位	OF	DF	IF	SF	ZF	AF	PF	CF
为 0 时的符号	NV	UP	DI	PL	NZ	NA	PO	NC
为 1 时的符号	OV	DN	EI	NG	ZR	AC	PE	CY

(2) U 命令

用 U 命令可以反汇编(Unassemble)可执行代码。例如,若在执行上述 R 命令后,再输入 U 命令,就会有如下的显示输出:

```
-U
10F1 : 0000    B8EE10      MOV  AX , 10EE
10F1 : 0003    8ED8        MOV  DS , AX
10F1 : 0005    8D1E0000    LEA  BX , [0000]
10F1 : 0009    031E0A00    ADD  BX , [000A]
10F1 : 000D    8B0F        MOV  CX , [BX]
10F1 : 000F    B44C        MOV  AH , 4C
10F1 : 0011    CD21        INT  21
10F1 : 0013    0000        ADD  [BX + SI] , AL
10F1 : 0015    0000        ADD  [BX + SI] , AL
10F1 : 0017    0000        ADD  [BX + SI] , AL
```

在上面的反汇编输出中,最左边的部分给出了指令首址的段基值:偏移量,接着是对应指令的机器码,最后是反汇编出来的汇编形式指令。与汇编源程序不同的是,这里的数据一律用不带后缀的十六进制表示,地址直接用其值而不用符号形式表示。

另外,也可在 DOS 提示符下按下述步骤对给定的可执行代码(如 EXAM1.EXE) 进行反汇编操作:

```
C:\ tools > DEBUG  EXAM1.EXE  (↙)
-U  (↙)
10F1 : 0000    B8EE10      MOV  AX , 10EE
10F1 : 0003    8ED8        MOV  DS , AX
10F1 : 0005    8D1E0000    LEA  BX , [0000]
10F1 : 0009    031E0A00    ADD  BX , [000A]
10F1 : 000D    8B0F        MOV  CX , [BX]
10F1 : 000F    B44C        MOV  AH , 4C
10F1 : 0011    CD21        INT  21
10F1 : 0013    0000        ADD  [BX + SI] , AL
10F1 : 0015    0000        ADD  [BX + SI] , AL
10F1 : 0017    0000        ADD  [BX + SI] , AL
```

(3) T(Trace)命令

T 命令也称“追踪”命令,用它可以跟踪程序的执行过程。T 命令有两种格式,即单步追踪和多步追踪。

① 单步追踪,格式为:

```
T [ = 地址]
```

该命令从指定的地址处执行一条指令后停下来,显示各寄存器的内容和标志位的状态,并给出下一条指令的机器码及其汇编形式,同时给出该指令首字节的逻辑地址。若命令中没有指定地址,则执行当前 CS : IP 所指向的一条指令。

例如,在执行前面的 R 命令后,再输入 T 命令,则会执行 R 命令后显示的一条指令

(即 MOV AX, 10EE),并输出如下信息:

```
-T
AX=10EE   BX=0000   CX=0080   DX=0000   SP=0020   BP=0000   SI=0000   DI=0000
DS=10DE   ES=10DE   SS=10EF   CS=10F1   IP=0003   NV   UP   DI   PL   NZ   NA   PO   NC
10F1:0003   8ED8   MOV DS,AX
```

② 多步追踪,格式为:

```
T[=地址][值]
```

该命令与单步追踪基本相同,所不同的是它在执行了由[值]所规定的指令条数后停下来,并显示相关信息。

例如,在执行前面的 R 命令后,输入如下命令行即可从指定地址 10F1:0000 开始相继执行两条指令停下来,并显示相关信息。

```
-T=10F1:0000  2
```

(4) G(Go)命令

G 命令也称运行命令,格式为:

```
G[=地址 1][地址 2[地址 3…]]
```

其中,地址 1 规定了执行的起始地址的偏移量,段地址是 CS 的值。若不规定起始地址,则从当前 CS:IP 开始执行。后面的若干地址是断点地址。

例如,在启动 DEBUG 并装入 EXAM1.EXE 后,输入如下命令行即可从起始地址 10F1:0000 开始执行至断点处 10F1:000F 停下来,并显示相关信息。

```
-G=10F1:0000  000F  (↙)
AX=10EE    BX=0005    CX=0400   DX=0000    SP=0020   BP=0000   SI=0000   DI=0000
DS=10EE   ES=10DE   SS=10EF   CS=10F1   IP=000F   NV   UP   DI   PL   NZ   NA   PE   NC
10F1:000F   B44C   MOV   AH,4C
```

(5) D 命令

D 命令也称转储(Dump)命令,格式为:

```
D 段基值:偏移地址
```

它从给定的地址开始依次从低地址到高地址显示内存 80 个字节单元的内容。显示时,屏幕左边部分为地址(段基值:偏移量);中间部分为用十六进制表示的相应字节单元中的内容;右边部分给出可显示的 ASCII 码字符。如果该单元的内容不是可显示的 ASCII 码字符,则用圆点"."表示。例如,如果在相继执行"Debug EXAM1.EXE"和"-U"命令后,再输入"D 1109:0000"命令,则会出现如下所示的显示内容:

```
1109:0000   B8 03 11 8E D8 BA 02 00 - B4 09 CD 21 8B 1E 00 00   8...x:..5.M!....
1109:0010   B5 04 B1 04 D3 C3 8A C3 - 24 0F 04 30 3C 39 7E 02   5.1.SC.C$..0<9~.
1109:0020   04 07 8A D0 B4 02 CD 21 - FE CD 75 E6 B8 00 4C CD   ...P5.M!~Muf8.LM
1109:0030   21 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00    !  ..............
```

```
1109:0040  00 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 00 00     ..............
```

(6) Q 命令

执行 Q 命令将退出 DEBUG 环境,返回到 DOS 操作系统,如下所示:

```
-Q  (↙)
C:\tools>
```

习题 5

5.1 判别下列标识符是否合法:

```
Y3.5 , 3 DATA , BCD# , (one) , PL*1 , ALPHA-1 , PROC-A , AAA
```

5.2 下列语句在存储器中分别为变量分配多少字节?

```
(1) VAR1  DW  10
(2) VAR2  DW  5 DUP(2) , 0
(3) VAR3  DB  'HOW ARE YOU?', '$'
(4) VAR4  DB  2 DUP(0 , 4 DUP(?) , 0)
(5) VAR5  DD  -1, 1, 0
```

5.3 下列指令各完成什么功能?

```
(1) MOV  AX , 00FFH AND 1122H+2233H
(2) MOV  AL , 15 GE 1111B
(3) AND  AX , 0F00FH AND 1234H OR 00FFH
(4) OR   AL , 50 MOD 4+20
(5) ADD  WORD PTR [BX] , 1122H AND 00FFH
```

5.4 若定义 DAT DD 12345678H,则(DAT+1)字节单元的数据是_____。

A. 12H　　B. 34H　　C. 56H　　D. 78H

5.5 执行下面的程序段后,AX=_____。

```
TAB    DW  1 , 2 , 3 , 4 , 5 , 6
ENTRY  EQU  3
MOV    BX , OFFSET  TAB
ADD    BX , ENTRY
MOV    AX , [BX]
```

A. 0003H　　B. 0300H　　C. 0400H　　D. 0004H

5.6 根据下面的数据定义,指出数据项 $+10 的值(用十六进制表示)。

```
       ORG  10H
DAT1   DB  10  DUP (?)
DAT2   EQU  12H
DAT3   DW  56H , $+10
```

5.7 若程序中数据定义如下：

```
PARTNO  DW  ?
PNAME   DB  16 DUP (?)
COUNT   DD  ?
PLENTH  EQU $ - PARTNO
```

则 PLENTH＝_____。

5.8 下列程序段运行后，A 单元的内容为_____。

```
DSEG  SEGMENT
   A  DW  0
   B  DW  0
   C  DW  230H , 20H , 54H
DSEG  ENDS
SSEG  SEGMENT  STACK
   DB  256 DUP(?)
SSEG  ENDS
CSEG  SEGMENT
      ASSUME  CS: CSEG , DS: DSEG , SS: SSEG
START: MOV  AX, DSEG
       MOV  DS , AX
       MOV  BX, OFFSET  C
       MOV  AX , [BX]
       MOV  B , AX
       MOV  AX , 2[BX]
       ADD  AX , B
       MOV  A , AX
CSEG  ENDS
      END  START
```

5.9 执行下面的程序后，CX＝_____。

```
DATA  SEGMENT
  A  DW  1 , 2 , 3 , 4 , 5
  B  DW  5
DATA  ENDS
CODE  SEGMENT
   ASSUME  CS:CODE , DS:DATA
START:  MOV  AX , DATA
        MOV  DS , AX
        LEA  BX , A
        ADD  BX , B
        MOV  CX , [BX]
        MOV  AH , 4CH
        INT  21H
CODE  ENDS
      END  START
```

5.10　试编写一完整的汇编源程序,其功能为在 CHAR 为首址的 26 个字节单元中依次存放'A'～'Z'。

5.11　在数据段中从偏移地址 BUF 开始连续存放着 100 个字符,编写一完整汇编源程序,将该字符串中所有的字母 A 都替换成字母 B(要求在显示器上分别显示替换前及替换后两个不同的字符串)。