第5章

汇编语言程序设计

汇编语言是计算机系统底层机器语言的符号表示形式，它用助记符代替二进制的指令代码，用标号或符号代表地址、常量或变量，克服了机器语言不容易记忆、不方便使用的缺点。汇编语言能够利用CPU的指令系统及相应的寻址方式，编写出占用内存少、运行速度快的程序，还能直接利用计算机硬件提供的寄存器、标志和中断，对寄存器、内存及I/O端口进行各种操作，是直接操作硬件的、效率最高的语言。

本章介绍汇编语言程序一般格式、基本语法、伪指令语句、宏指令、系统功能调用等，并通过程序实例介绍分支、循环、子程序等常用的汇编语言结构，最后介绍汇编语言程序的上机步骤和调试程序DEBUG的使用方法。

5.1 汇编语言程序基本格式

5.1.1 汇编语言源程序和汇编程序

通过2.1.1节大家知道，用汇编语言编写的源程序，计算机无法直接识别和执行，需要通过**汇编程序**翻译成**目标程序**。

汇编后形成的目标程序虽然是二进制代码，但还不能直接上机运行，必须经过**链接程序**连接，将库文件或其他目标文件连接到一起形成**可执行文件**后，才能送入计算机执行。汇编语言程序从建立到汇编、连接形成可执行程序的整个过程如图5.1所示。

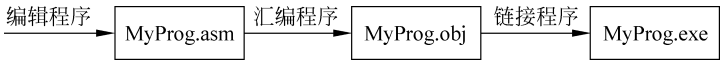


图5.1 汇编语言程序的建立及汇编过程

汇编程序是较早也较成熟的一种系统软件，它的主要功能是将汇编语言源程序转换为目标程序，同时还具有以下的一些功能：检查源程序中的语法错误，并给出出错信息；进行数制转换、计算表达式；分配内存空间；展开宏指令等。目前使用的汇编程序主要是宏汇编程序(Macro Assembler，MASM)。

5.1.2 汇编语言的特点

汇编语言远不如高级语言方便、实用，而且编写同样的程序，使用汇编语言比使用高级语言花费的时间更多，调试和维护更困难。既然如此，为什么还要使用汇编语言呢？主要有两个原因：性能和对计算机的完全控制。使用汇编语言编写的程序有如下特点。

(1) 执行速度快。

(2) 程序短小。

(3) 可以直接控制硬件。

(4) 可以方便编译。

(5) 辅助计算机工作者掌握计算机体系结构。

5.1.3　一般汇编语言程序的结构形式

与内存分段结构相对应,汇编语言源程序采用分段结构,一般一个完整的源程序由 3 种程序段组成,即代码段、数据段、堆栈段。每一个段都以 SEGMENT 开始,以 ENDS 结束,二者之间为语句体,整个源程序以 END 结束。汇编语言程序的一般结构形式为:

```
NAME1   SEGMENT        ; 段的起始
        语句 1         ;   ┐
        语句 2         ;   ├ n 条语句序列构成的语句体
        ⋮                  ┘
        语句 n         ;
NAME1   ENDS           ; 段的结束
NAME2   SEGMENT        ; 段的起始
        语句 1         ;   ┐
        语句 2         ;   ├ m 条语句序列构成的语句体
        ⋮                  ┘
        语句 m         ;
NAME2   ENDS           ; 段的结束
        END            ; 源程序结束
```

每一段的语句体由语句序列组成,8086 汇编语言语句分为如下 3 类。

(1) 指令语句:8086 指令系统的指令形式,与机器指令一一对应。

(2) 伪指令语句:又称管理语句。在汇编语言源程序的汇编过程中起主要作用,它是对汇编程序的命令语句,一般不生成的目标代码。

(3) 宏指令语句:是宏汇编程序能识别的、预先定义的指令代码序列。一旦定义以后,宏指令就像一条指令一样,可以在源程序中被引用,其效果等同于引入一段代码序列。

每条语句最多由 4 个域组成,一般格式如下。

```
指令语句:
[标号:]  操作符   操作数   [; 注释]
伪指令语句:
[名字]   伪指令符   参数   [; 注释]
```

其中,标号(或名字)和注释是可选的,操作数或参数的有无及个数根据具体的指令或伪指令而异。

【例 5.1】 一个简单的程序示例。

```
DATA   SEGMENT                          ; 数据段开始
       NUM1   DB     1AH,24H           ; 定义原始数据
       NUM2   DW     0                 ; 保存结果单元
DATA   ENDS                             ; 数据段结束
STACK1 SEGMENT    STACK                 ; 堆栈段开始
       SKTOP  DB      40 DUP(0)        ; 定义堆栈空间
STACK1 ENDS                             ; 堆栈段结束
CODE   SEGMENT                          ; 代码段开始
       ASSUME    CS: CODE, DS: DATA    ; 段指定
       ASSUME  SS: STACK1
START: MOV       AX,DATA               ; 初始化数据段基址
       MOV       DS,AX
       MOV       AL,NUM1               ; 取第一个数据
       ADD       AL,NUM1 + 1           ; 与第二个数据相加
       MOV       BYTE PTR NUM2,AL      ; 保存结果
       MOV       AH,4CH                ; 程序结束退出
       INT    21H
CODE   ENDS                             ; 代码段结束
       END       START                 ; 源程序结束
```

以上是一个完整的汇编程序,其中涉及的语法、伪指令和系统功能调用等内容,将在下面详细介绍。

5.2 汇编语言中的数据

5.2.1 常量

常量(Constant)是指在程序运行过程中不变的量,8086 汇编语言允许的常量如下。

1. 数值常量

汇编语言中的数值常量可以是二进制、八进制、十进制或十六进制数。

(1) 二进制数:由 0 和 1 组成的数字序列,以字母 B 结尾,如 00101100B。

(2) 八进制数:由数字 0~7 组成的数字序列,以字母 O 或 Q 结尾,如 1777O 或 1777Q。

(3) 十进制数:由 0~9 组成的数字序列,以字母 D 结尾,如 178D。一般情况下,基数默认为十进制数,因此可以省略后缀 D。

(4) 十六进制数:由 0~9 及 A~F 组成的数字和字母序列,以字母 H 结尾。这个数的第一个字符必须是 0~9,如果第一个字符是 A~F,则应在其前面加上数字 0,以避免和标识符混淆,如 0FFFFH。

2. 字符串常量

包含在单引号中的若干字符形成字符串常量,字符串在计算机中存储的是相应字符的 ASCII 码。例如,'A'的码值是 41H,'AB'的码值是 4142H 等。

3. 符号常量

常量用符号名来代替就是符号常量。

例如：

```
COUNT EQU 3
COUNT = 3
```

定义了一个符号常量 COUNT，与数值常量 3 等价。

5.2.2　变量

变量(Variable)是内存中某个存储区域的存储单元，其内存储的数据在程序运行期间随时可以修改。为了便于对变量的访问，它常常以变量名的形式出现在程序中。变量名是内存中一个数据区域的名称，即数据内存地址的符号表示，可以在数据段、附加数据段或堆栈段中定义。

1. 变量的定义

定义变量就是给变量分配存储单元，并且给这个单元赋予一个变量名。

定义变量是使用数据定义伪指令来实现的，其格式为：

```
[变量名] 伪指令 表达式 [,表达式 …]
```

(1) 这些伪指令可以把其后的数据存入指定的存储单元，并初始化数据，或者只分配存储空间而并不初始化数据。

其中，变量名字段是可有可无的，它是内存单元地址的符号表示，其作用与指令语句前的标号相同，但它的后面没有冒号。程序汇编时，将第一字节的偏移地址赋给变量名。

伪指令字段说明所定义的数据类型，常用的数据定义伪指令有以下几种。

① DB 伪指令：用来定义字节，其后的每个操作数都占有一字节(8 位)。

② DW 伪指令：用来定义字，其后的每个操作数占有一字(16 位，其低位字节在第一个字节地址中，高位字节在第二个字节地址中)。

③ DD 伪指令：用来定义双字，其后的每个操作数占有两字(32 位)。

④ DF 伪指令：用来定义 6 字节的字，其后的每个操作数占有 48 位，可存储由 16 位段地址及 32 位偏移地址组成的远地址指针。这一伪指令只能用于 386 及其后继机型中。

⑤ DQ 伪指令：用来定义 8 字节，其后的每个操作数占有 4 字(64 位)，可用来存放双精度浮点数。

⑥ DT 伪指令：用来定义 10 字节，其后的每个操作数占有 10 字节，形成压缩的 BCD 码。

(2) 表达式是给变量预置的初值，可以是下述情况之一。

① 数值表达式：数值允许用二进制、八进制、十进制、十六进制形式书写。

② ?：表示不预置确定的初值。

③ 字符串表达式：用引号括起来的不超过 255 个字符或其他 ASCII 码符号。DB 伪指令将按顺序为字符串中每一个字符或符号分配一字节单元，存放它们的 ASCII 编码，但除 DB 以外的数据定义伪指令只允许定义最多两个字符的字符串，且按逆序存放在低地址开始的单元。

④ 带 DUP 操作符的表达式：DUP 是定义重复数据操作符，它的使用格式为：

```
N   DUP   (EXP)
```

其中，N 为重复次数，EXP 为表达式。

例如：

```
DATA1   DB   25,25H,10011010B          ; 数值表达式
DATA2   DB   ?,?                       ; ?表达式
DATA3   DB   2 DUP(2 DUP(4),15)        ; DUP 表达式
DATA4   DB   'AB','CD'                 ; 字符串表达式
DATA5   DW   ?,?,-32768                ; 字类型
DATA6   DD   80000000H,36H             ; 双字类型
```

上述语句汇编后的内存分配情况如图 5.2 所示。

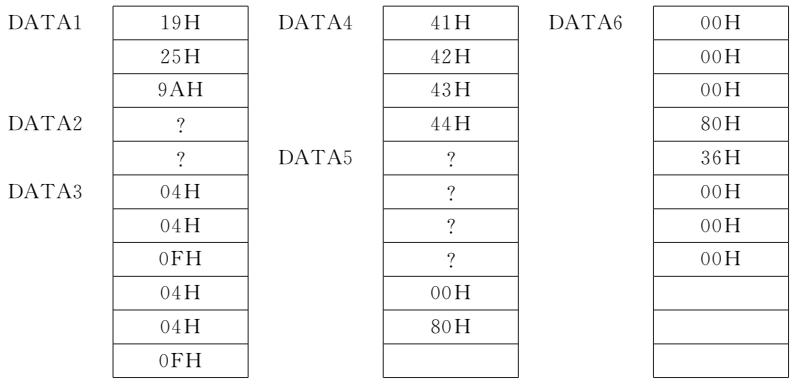


图 5.2　数据在内存中的分配情况

2. 变量的属性

经过定义的变量有 3 种属性：段属性、偏移属性、类型属性。

(1) 段属性(SEGMENT)：变量所在段的起始地址(16 位)，此值必须在一个段寄存器中。

(2) 偏移属性(OFFSET)：该变量与段的起始地址之间相距的字节数。对于 16 位段，是 16 位无符号数；对于 32 位段，则是 32 位无符号数。在当前段内给出变量的偏移值等于当前地址计数器的值，当前地址计数器的值可以用 $ 来表示。

(3) 类型属性(TYPE)：定义该变量的字节数，如 BYTE(DB，1 字节长)、WORD(DW，2 字节长)、DWORD(DD，4 字节长)、FWORD(DF，6 字节长)、QWORD(DQ，8 字节长)、TBYTE(DT，10 字节长)。

可以通过取值运算符 SEG、OFFSET 和 TPYE 取得变量的属性，详见 5.3 节。

3. 变量的使用

(1) 变量名作为存储单元的直接地址：变量名用于直接寻址时，变量的类型必须与指令的要求相符合。

例如：假设已定义字节变量 AB、字变量 AW，用变量名直接寻址的形式为：

```
MOV   AH,AB
MOV   AX, AW
```

(2) 用合成运算符 PTR 可以临时改变变量类型。

例如：假设已定义字节变量 AB、字变量 AW，在如下指令序列中：

```
MOV   CX, WORD PTR AB
MOV   CL, BYTE PTR AW
```

临时把 AB 变为字类型，AW 变为字节类型，但段和偏移属性不变。

(3) 变量名作为相对寻址中的偏移量。

例如，假设已定义字节变量 AB、字变量 AW，在如下指令序列中：

```
MOV  AX, AB[SI]
MOV  AX, AW[ BX][ SI]
```

AB、AW 分别表示它们的偏移量而不是它们所表示的数据，常用于数组或表格操作，AB[SI]就表示 AB 数组中第 SI 个元素。

(4) 变量名仅对应数据区第一个数据项。

例如：

```
WORD   DW 20 DUP( ?)
MOV    AX, WORD          ; 第一个元素送 AX
MOV    AX, WORD + 38     ; 第 20 个元素送 AX
```

5.2.3　标号

标号(Label)是某条指令所在内存单元地址的符号表示，经常在转移指令或子程序调用指令的地址码字段出现，用于表示转向的目标地址。

例如：

```
LOP1:   指令
          ⋮
     LOOP LOP1
          ⋮
     JNE NEXT
NEXT:    指令
          ⋮
```

标号在代码段中定义，后面跟着冒号，它也可以用 LABEL 或 EQU 伪操作来定义。此外，它还可以作为过程名来定义。

对于汇编程序来说,标号与变量是类似的,都是存储单元地址的符号表示。只是标号对应的存储单元中存放的是指令;而变量所对应的存储单元中存放的是数据。所以,标号也有3种属性:段属性、偏移属性和类型属性。

(1) 段属性:定义标号的程序段的起始地址,标号的段地址总是在CS寄存器中。

(2) 偏移属性:标号与所在段的段起始地址之间的字节数。对于16位段,是16位无符号数;对于32位段,则是32位无符号数。

(3) 类型属性:用来指出该标号是在本段内引用的,还是在其他段中引用的。如果是在本段内引用的,则称为NEAR。对于16位段,指针长度为2B;对于32位段,指针长度为4B。若在段外引用,则称为FAR。对于16位段,指针长度为4B(段地址2B,偏移地址2B);对于32位段,指针长度为6B(段地址2B,偏移地址4B)。

在同一个程序中,同样的标号或变量的定义只允许出现一次,否则汇编程序会提示出错。

5.3 运算符与表达式

8086汇编语言定义了多种类型的运算符,运算符与操作数组成表达式,表达式经汇编后形成新的操作数。表达式分为数值表达式和地址表达式。数值表达式的运算结果是一个数;地址表达式的运算结果是一个存储单元的地址。

1. 算术运算符

算术运算符有+(加)、-(减)、*(乘)、/(除)、MOD(取模)。

算术运算符可以用于数值表达式和地址表达式中,用于地址表达式中要注意地址表达式的物理意义。例如,同一段中的两个地址相减,其值为两个地址之间字节单元的个数;一个地址加上一个整数,其值为另一个单元的地址;一个地址减去一个整数,其值为另一个单元的地址,以上这些运算都是有意义的。而两个地址相加、相乘、相除则是没有意义的。

例如,下面的两条指令分别包含了数值表达式和地址表达式。

```
MOV AL,4 * 8 + 5                ; 数值表达式
MOV SI,OFFSET BUF + 12          ; 地址表达式
```

2. 逻辑运算符

逻辑运算符有AND(与)、OR(或)、XOR(异或)、NOT(非)。

逻辑运算符只能用于数值表达式中,不能用于地址表达式中。逻辑运算符和逻辑运算指令是有区别的。逻辑运算符的功能在汇编阶段完成,而逻辑运算指令的功能在程序执行阶段完成。

在汇编阶段,以下两条指令是等价的。

```
AND AL,78H AND 0FH
AND AL,08H
```

3. 关系运算符

关系运算符有 EQ(相等)、LT(小于)、LE(小于或等于)、GT(大于)、GE(大于或等于)、NE(不等于)。

关系运算符要有两个运算对象。两个运算对象要么都是数值,要么都是同一个段内的地址。关系运算的结果为数值,当关系成立时,结果为 0FFFFH;当关系不成立时,结果为 0000H。

例如,以下两条指令的结果是等价的。

```
MOV BX,32 EQ 45
MOV BX,0
```

以下两条指令的结果也是等价的。

```
MOV BX,56 GT 30
MOV BX,0FFFFH
```

4. 取值运算符

取值运算符(又称分析运算符)可以从变量和标号中分析出它们的段地址、偏移地址、变量的类型、元素的个数和占用内存的大小等。8086 提供的取值运算符有 SEG、OFFSET、TYPE、LENGTH、SIZE。

SEG:返回变量和标号的段地址。

OFFSET:返回变量和标号在段内的地址偏移量。

TYPE:返回变量和标号的类型,用一个数字表示。TYPE 返回值与变量和标号、类型的对应关系如表 5.1 所示。

LENGTH:如果一个变量已经用重复操作符 DUP 说明其变量的个数,则 LENGTH 运算符可返回该变量所包含的数据个数。

SIZE:返回变量所占用内存的字节数。它等于 LENGTH 与 TYPE 的乘积。

表 5.1　TYPE 返回值与变量和标号、类型的对应关系

类　型		返 回 值
变量	字节数据	1
	字数据	2
	双字数据	4
标号	NEAR 指令单元	−1
	FAR 指令单元	−2

例如:

```
SCORE DW 30 DUP(0)
```

定义了一个变量 SCORE,则 TYPE SCORE 为 2,LENGTH SCORE 为 30,而 SIZE SCORE 为 60。

5. 合成运算符

合成运算符也称为修改属性运算符,它能修改变量或标号原有的类型属性并赋予其新的类型。合成运算符包括 PTR 和 THIS 运算符。

1) PTR 运算符

格式:

```
类型 PTR 表达式
```

其中,类型可以是 BYTE、WORD、DWORD、NEAR、FAR,表达式是被修改的变量或标号。

例如,NUM 被语句

```
NUM DB  1,3,5,7
```

定义为字节类型,若要将 NUM 开始两字节的数据装入 AX,则指令:

```
MOV AX,NUM
```

是非法的,应修改为:

```
MOV AX,WORD PTR NUM
```

因为在这个指令中 WORD PTR NUM 将 NUM 一次性地修改为字型。若先用赋值语句:

```
DNUM EQU WORD PTR NUM
```

则上面的传送指令可写为:

```
MOV AX,DNUM
```

虽然上述的赋值语句重新定义了一个符号名 DNUM,但并未给 DNUM 分配新的内存,DNUM 仍指向 NUM 所指的单元,它们有相同的内存空间,即二者具有相同的段属性和偏移属性。它们的区别仅在于类型的不同,NUM 为字节型,而 DNUM 为字型。

2) THIS 运算符

THIS 的功能与 PTR 相同,只是格式不同。THIS 语句中建立一个新的符号名并指定它有 THIS 后的类型,而新符号名指向下一语句的原符号名的内存地址。

格式:

```
新符号名 EQU THIS 类型
原符号名 类型 参数,…
```

例如，前面用 PTR 修改 NUM 类型可用下面的 THIS 语句代替：

```
DNUM EQU THIS WORD
NUM DB 1,3,5,7
```

其中，DNUM 是字型并指向 NUM 所指的内存单元，DNUM 的存取以字为单位，而 NUM 仍是字节类型。

5.4　伪　指　令

除汇编指令外，汇编语言程序的语句还可以由伪指令和宏指令组成。

伪指令是构成汇编语言源程序的一种重要语句。它不像机器指令那样是在程序运行期间执行的，而是在汇编期间由汇编程序处理的操作。伪指令在汇编期间告诉汇编程序如何为数据项分配内存空间、如何设置逻辑段、段寄存器和各逻辑段的对应关系及源程序到哪里结束等信息，以便指导汇编程序分配内存、汇编源程序、指定段寄存器。在最后形成的目标代码及可执行程序中，伪指令已经不存在。也就是说，伪指令不产生相应的机器代码。

MASM 有 60 多种伪指令，本节介绍一些常用的伪指令。不同版本的汇编程序支持不同的伪指令。

1. 符号定义伪指令

符号定义伪指令用来给一个符号重新命名，或者定义新的类型属性等。

1）赋值伪指令 EQU

EQU 伪指令给表达式赋予一个标识符，此后，程序中凡是用到该表达式的地方，就都可以用这个标识符来代替了。这里的表达式可以是常数、符号、数值表达式、地址表达式，甚至可以是指令助记符。

格式：

```
符号名 EQU 表达式
```

EQU 的引入提高了程序的可读性，也使其更加易于修改。例如：

```
CONSTANT   EQU   256           ; 将数 256 赋给符号名 CONSTANT
DATA       EQU   HEIGHT + 12   ; HEIGHT 为标号，将地址表达式赋给符号名 DATA
ALPHA      EQU   7
BETA       EQU   ALPHA - 2     ; 把 7 - 2 = 5 赋给符号名 BETA
B          EQU   [BP + 8]      ; 变址引用赋给符号名 B
P8         EQU   DS: [BP + 8]  ; 加段前缀的变址引用赋给符号名 P8
```

在 EQU 语句中，如果出现变量或标号，则在该语句前应该先给出它们的定义。例如，若有以下伪指令语句：

```
BETA   EQU   ALPHA - 2
```

则在该语句之前必须有 ALPHA 的定义,否则汇编程序将指示出错。

另外,EQU 语句在使用 PURGE 语句解除之前,不允许重新定义。

2) 等号伪指令=

与 EQU 相类似,等号伪指令也可以用作赋值操作。它们之间的区别是:EQU 伪指令定义的标识符是不允许重复定义的,而等号伪指令则允许重复定义。

例如:

```
EMP = 6
```

或

```
EMP   EQU   6
```

它们都可以使数 6 赋以符号名 EMP,然而不允许二者同时使用。但是,下列语句:

```
  ⋮
EMP = 7
  ⋮
EMP = EMP + 1
  ⋮
```

在程序中是允许使用的,因为等号伪指令允许重复定义。在这种情况下,在第一条语句后的指令中,EMP 的值为 7,而在第二条语句后的指令中,EMP 的值为 8。

3) 类型定义伪指令 LABEL

LABEL 伪指令可以指定变量或标号所对应存储单元的类型。其中变量的类型值可以是 BYTE、WORD、DWORD,标号的类型值可以是 NEAR 和/或 AR。

格式:

```
变量 LABEL 类型
标号 LABEL 类型
```

变量或标号的段属性和偏移属性由下一条语句决定。例如:

```
DATW   LABEL WORD
DATB   DB     20 DUP(0)
```

这个 20 字节元素的数组被赋予两个不同类型的数组名,即 DATW 是 DATB 的别名,这两个变量具有同样段属性和偏移属性,只是类型不同,DATW 为字类型,DATB 为字节类型。换言之,同一数组定义了两种不同的类型,在接受不同数据类型访问时,可以指定相应的标号。例如:

```
MOV    AX,DATW
MOV    AL,DATB
```

如接收一个字类型数据访问时，使用 DATW，接收一个字节类型数据访问时，使用 DATB。否则会因为数据类型不匹配，编译器编译时将出现异常。

下面是 LABEL 伪指令定义标号的例子：

```
FLPT   LABEL FAR
NLPT:   MOV AX,BX
```

“MOV　AX,BX”指令有两个标号，即近类型的 NLPT 和远类型的 FLPT，既可以在段内引用这条指令，也可以用标号 FLPT 实现段间引用。

4）解除定义伪指令 PURGE

使用 EQU 伪指令定义过的符号，若以后不再使用了，可以使用 PURGE 语句来解除定义。

格式：

```
PURGE 符号 1,符号 2,…,符号 N
```

解除符号定义后，可用 EQU 语句重新定义。例如：

```
Y1     EQU    7          ; 定义 Y1 的值为 7
PURGE  Y1                ; 解除 Y1 的定义
Y1     EQU    36         ; 重新定义 Y1 的值为 36
```

2. 数据定义伪指令

数据定义伪指令用来定义变量、为变量分配存储单元并赋初值等，这在 5.2.2 节已经进行了介绍。

3. 段定义伪指令

80x86 的内存是分段的，程序必须按段来组织和利用存储器。一个程序允许使用代码段、数据段、堆栈段和附加段 4 个段，程序的不同部分应放在确定的段中。例如，程序中可执行的代码放在代码段中，程序使用的数据放在数据段中。

段定义伪指令就是为程序的分段而设置的。

1）段定义伪指令 SEGMENT 和 ENDS

格式：

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

其中，段名由用户自己定义；定位类型、组合类型、类别分别确定段名的属性。这三部分不是必需的，可视需要选取。

(1) 定位类型。定位类型用于指定段的起始地址在内存中所处的位置，它可以是 PARA、PAGE、WORD 和 BYTE 4 种类型。

PARA(节)：是默认类型，表示段起始边界地址的低 4 位为 0，即段的起始地址总是 16

的倍数。

PAGE(页):表示段起始边界地址的低8位为0,即段的起始地址总是256的倍数。

WORD(字):表示段从一个字边界地址开始,即段地址必须是偶数。当多个目标程序段要连接在同一个物理段时,各源程序的段首说明中选用WORD,以节省内存。

BYTE(字节):表示段可以从任何地址开始。

(2) 组合类型。组合类型用于告诉链接程序该段与其他段的链接关系。一个程序的源程序可以分为若干部分,每部分中都可能有代码段、数据段等。源程序经汇编后还需链接才能成为可执行的程序,链接时需要将分散在不同部分而又有共同特征的段进行组合,如将某些代码段组合在一起构成统一的代码段等,组合类型用于确定源程序中各段的链接关系。

组合类型有NONE、PUBLIC、COMMON、MEMORY、AT、STACK等多种类型。例如,NONE类型表示该段与其他段无任何关系,各自有自己的段基址,是默认的设置;PUBLIC表示该段与其他同名、同类别段链接成一个物理段时,所有这些段有一个共同的段基地址。

(3) 类别。程序在链接时只将同类别的段链接并放在一个连续的存储区构成段组,类别就是给这个段组命名的。类别可以是任何合法的名称,必须用单引号引起来,如'CODE'、'STACK'等。

2) 段寄存器指派伪指令ASSUME

段定义伪指令定义了不同的段,但它并没有说明所定义的段中,哪个是代码段、哪个是数据段、哪个是堆栈段等。ASSUME伪指令就是用来指定程序中定义的段分别是什么段、对应哪个段寄存器,以便在执行指令时,能够正确地计算物理地址。也就是说,它明确了源程序中的逻辑段和内存中的物理段之间的对应关系。

格式:

```
ASSUME 段寄存器: 段名[,段寄存器: 段名,…]
```

其中,段名必须是由SEGMENT定义过的,段寄存器则是CS、DS、SS和ES。由于不同的段可以彼此分离、重叠或完全重叠,因此,不同的段名既可以指派不同的段寄存器,也可以指派同一个段寄存器。应当注意,ASSUME伪指令只是定义段名与段寄存器的对应关系,并不能将段地址装入段寄存器中。因此,DS、ES和SS中的段地址还要在程序中通过MOV指令装入,代码段CS寄存器的初值由系统自动装入。程序代码为:

```
MOV AX,DATA
MOV DS,AX
```

4. 过程定义伪指令(PROC和ENDP)

过程也称为**子程序**,它是实现程序模块化设计的重要方法。过程作为一个独立存在的模块,能完成特定的任务。过程定义语句可以把程序分成模块,以便编写、阅读、调试和修改和组合。过程定义语句的格式为:

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

过程名是过程的标识符，也是过程的入口地址，它具有段属性和偏移属性。过程名是由用户自己定义的合法的名称。过程的属性有近调用(NEAR)和远调用(FAR)。若过程和调用过程的程序在同一段内，则属于近调用，该过程具有 NEAR 属性；若二者不在同一段内，则属于远调用，它具有 FAR 属性。

在一个过程中至少应有一条 RET 指令，以使程序能够正常返回。

5. 程序标题伪指令(TITLE)

TITLE 伪指令指定一个标题，以便能在列表文件每一页的第一行打印出这个标题，放置在程序的开始处。

格式：

```
TITLE 文本
```

其中，文本是用户给出的字符串，要求长度不超过 6 个字符。

6. 地址计数器与对准伪指令

1) 取地址伪指令 $

在汇编程序对源程序的汇编过程中，汇编程序使用一个地址计数器来保存当前正在被汇编的指令或数据的地址。$ 伪指令就是用来取这个当前汇编地址计数器中的值，它也被称为地址运算符、地址计数器。当编译完成后，代码中的“$”被一个实际的地址值取代了。

$ 用在指令中时，它表示当前指令的首地址。当开始汇编或在每一段开始时，$ 初始化为零，以后在汇编过程中，每处理一条指令，$ 就增加一个值，这个增量是该指令的字节数。

例如，指令

```
JNE $ + 6
```

的转向地址是 JNE 指令的首地址加上 6。

$ 用在伪指令时，它表示当前变量的位置，即地址计数器的当前值。例如：

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

若汇编时 ARRAY 分配的偏移地址为 0074，则汇编后的存储区如图 5.3 所示。

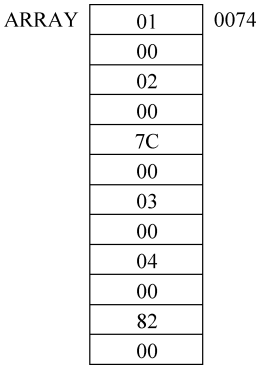


图 5.3　汇编结果

注意,ARRAY 数组中的两个($+4)得到的结果是不同的,这是由于 $ 的值是在不断变化的。

2) 移动地址指针伪指令 ORG

伪指令 ORG 可以设置当前汇编地址计数器中的值。

格式:

```
ORG 常量表达式
```

其中,常量表达式给出了地址指针相对于当前指针的偏移量。当 ORG 指定了新的地址指针之后,其后的程序和数据就从此指针指示的起始地址开始存放。

例如,在代码段开始有语句为:

```
ORG 100H
```

则从此语句起,其后的指令或数据从当前段的 100H 处开始存放。

ORG 也可以指定数据的地址,例如:

```
VECTORS SEGMENT
    ORG 10
VECT1   DW 47A5H
    ORG 20
VECT2   DW 0C596H
VECTORS ENDS
```

VECT1 的偏移地址值为 0AH,而 VECT2 的偏移地址值为 14H。

在 ORG 语句中若使用含有 $ 的表达式,例如:

```
ORG $ + 8
```

表示地址指针从当前地址跳过 8 字节,即建立了一个 8 字节的未初始化的数据缓冲区。若程序中需要访问该缓冲区,则可用 LABEL 伪指令来定义该缓冲区。

```
BUFFER LABEL BYTE
ORG $ + 8
```

其功能和

```
BUFFER DB 8   DUP(?)
```

是一样的。

3) EVEN 伪指令

EVEN 伪指令使下一个变量或指令开始于偶数字节地址。一个字的地址最好从偶地址开始。对于字数组,为保证其从偶地址开始,可以在其前用 EVEN 伪指令。例如:

```
DATA_SEGMENT
     ⋮
        EVEN
    WORD_ARRAY   DW   100DUP(?)
     ⋮
DATA_SEGENDS
```

4）ALIGN 伪指令

ALIGN 伪指令使下一个变量或指令开始于指定的位置。

格式：

```
ALIGN  BOUNDARY
```

其中，BOUNDARY 必须是 2 的幂。

例如，为保证双字数组边界从 4 的倍数开始，则可以使用如下语句。

```
.DATA
⋮
ALIGN  4
ARRAY DB 100 DUP(?)
⋮
```

显然，“ALIGN 4”可以保证下一个数据和指令是从偶地址开始，其功能和“EVEN”是等价的。

7. 基数控制伪指令（.RADIX）

汇编程序默认的数为十进制数，.RADIX 伪指令可以把默认的基数改变为 2～16 的任何基数。其格式为：

```
.RADIX  表达式
```

其中，表达式用来表示基数值（用十进制数表示）。

例如：

```
MOV BX,0FFH
MOV BX,178
```

与

```
.RADIX 16
MOV BX,OFF
MOV BX,178D
```

是等价的。

在用.RADIX 16 把基数定为十六进制后，十进制数后面都应跟字母 D。在这种情况下，如果某个十六进制数的末字符为 D，则其后应跟字母 H，以免与十进制数发生混淆。

5.5 汇编语言程序上机过程

汇编语言源文件是使用符号语言编写的文本文件,不能被机器识别,必须将其翻译成机器语言,这个过程称为汇编。能把用户编写的汇编语言源程序翻译成机器语言程序的程序系统,称为**汇编程序**。

汇编程序的主要功能是汇编和链接。汇编将源程序翻译并把它转换为用二进制代码表示的**目标文件**(OBJ 文件)。在汇编的过程中,首先对源程序进行语法检查,若存在错误,则给出错误提示,无错误的源程序即可转换为目标文件。

目标文件还要经过链接才能成为可以运行的可执行文件。链接能把多个目标文件、库文件链接成一个统一的模块,在此过程中还要为代码分配内存,形成**可执行文件**,可执行文件能由操作系统将其装入内存并运行。

汇编程序有多种,常用的是 Microsoft 公司的 ASM 和 MASM。其中,ASM 能够完成源文件的错误检查并给出错误提示、数制转换、表达式计算、翻译和内存的分配等汇编的基本功能,因此又称为**基本汇编**。而 MASM 除具有基本汇编功能外,还允许使用宏指令、结构和记录等高级汇编功能,因此 MASM 又称为**宏汇编**。本节主要介绍基本汇编程序的主要语句和基本汇编程序的使用方法。

编写能在计算机上运行的程序,应经历如下步骤。

(1) 利用文本文件编辑工具编辑源文件(.asm)。

(2) 用汇编程序将源文件(.asm)转换为目标文件(.obj)。

(3) 用链接程序将目标文件(.obj)转换为可执行文件(.exe)。

此后,就可以在 DOS 下执行以.exe 为扩展名的程序了。

1. 建立并编辑源程序

汇编语言的源程序是文本文件,任何文本文件编辑工具都可以用来编写源程序,如 MS DOS 自带的文本编辑程序 EDIT 等。应当注意,汇编语言源程序文件的扩展名为.asm。

2. 汇编形成目标文件

汇编语言源程序文件(.asm)经 MASM 汇编后可产生 3 个文件:目标文件、列表文件和交叉索引文件。目标文件的扩展名为.obj,在此文件中,操作码已被转换为机器码,但其中的地址还不是能将机器码装入内存的地址,而是一个相对的浮动地址;列表文件的扩展名为.lst,它是一个源程序和汇编后的目标程序列表,可供编程者参考;交叉索引文件的扩展名为.crf,其中列出了源程序中的符号和变量的定义、引用的情况。在汇编过程中,汇编程序会对是否建立这些文件,以及它们的文件名进行提问,以便用户选择。

在 DOS 状态下,使用 MASM 命令汇编源程序。例如,对 abc.asm 文件进行汇编的 DOS 命令如下:

```
C: > MASM abc ↙
```

此后屏幕的显示与操作如下:

```
Microsoft(R) Macro Assemble Version  5.00
Copyright(C) Microsoft Corp 1981—1985,1987. All right reserved.
Object filename [ABC.OBJ]:
Source Listing [NUL.LST]: abc
Cross - reference [NUL.CRF]: abc
51256 + 390090 Bytes Symbol Space free
0 Warning Errors
0 Severe Errors
C: >
```

在 MASM 命令执行后，首先显示版本号，然后出现第一个提示，询问要建立的目标程序的文件名，若 MASM 之后输入了文件名，则将输入的文件名加上扩展名.obj 作为默认的文件名显示在方括号中，如不修改目标文件名则直接按 Enter 键。若要修改，则输入新的文件名并按 Enter 键。之后出现第二、三个提示，询问是否建立列表文件和交叉索引文件，若建立，则输入文件名，否则直接按 Enter 键。在回答完第三个提示之后，汇编开始。在汇编过程中若发现源程序有语法错误，则显示出有错误语句的行号和错误的原因，以及错误的总数。此时，可根据错误提示，分析错误原因，并对源程序进行修改。修改后重新汇编，直到没有错误为止。汇编错误提示有两类：警告错误（Warning Errors）和严重错误（Severe Errors）。警告错误指示源程序存在的一般性错误，这类错误存在时，虽然可以继续汇编并生成目标文件，但以后的程序运行将可能出现错误。严重错误的存在将使汇编无法正确进行。当没有任何错误存在时，汇编才算结束。

3. 链接形成可执行文件

链接有两个作用：一是将多个目标文件、库文件等多个模块链接成统一的程序；二是将目标文件中的浮动地址转换为能将程序装入内存的地址。

链接使用 LINK.exe 程序文件，在 DOS 状态下，输入命令 LINK，即：

```
C: > LINK abc ↙
```

之后显示的版本号和依次给出的 3 个提示如下：

```
Microsoft(R) Overlay Linker Version 3、60
Copyright(C) MicrosoftCorp 1983 - 1987. Allrightsreserved.
RunFile [ABC.EXE]:
ListFile [NUL.MAP]: abc
Libraries [.LIB]:
```

在 LINK 命令之后可直接给出要连接的目标文件名，否则将提示用户输入它。若有多个文件要链接，则应输入所有链接的目标文件名并用“+”将其连接。第一个提示询问链接生成可执行文件的文件名，若采用方括号内给出的默认文件名，则直接按 Enter 键即可。第二个和第三个提示询问是否建立内存分配图文件和是否需要链接库文件，若需要则输入文件名，否则直接按 Enter 键。在上述操作完成之后开始链接，若链接过程有错误，则显示错误信息，修改程序后，再重新汇编、链接，直到没有错误为止。若链接程序给出“No STACK

segment"一般性的警告错误,并不影响程序的运行。

链接后建立的可执行文件(.exe)可以在DOS状态下运行。内存分配文件(.map)提供链接过程中内存地址分配的信息。

源程序经汇编、链接后,生成可执行文件(.exe),就可以在DOS状态下直接输入文件名运行了。例如:

```
C: > abc。
```

在DOS下运行程序文件时,DOS的外壳COMMAND.com将EXE文件装入内存,并将控制权交给调入程序后,程序就开始运行。COMMAND在装入EXE文件之前,先从可用内存的起点建立一个长度为100H字节的程序段前缀PSP,并自动设置DS和ES使其指向程序段前缀的段址。程序段前缀的结构如下:

```
00H～01H  INT  20H指令
02H～03H  内存的总容量(以16字节为单位)
04H～08H  FAR  JUMP  DOS子程序的调度程序
09H～0CH  程序结束地址
0DH～10H  Ctrl + Break退出地址
11H～14H  标准错误出口地址
5CH～6BH  FCB1
6CH～7BH  FCB2(若FCB1被打开,则FCB2被FCB1覆盖)
80H～FFH  隐含的磁盘传输区(DTA)
```

程序段前缀是被调入程序与DOS的接口,其中设有程序正常和非正常退出时应指向的地址及其他信息。程序段前缀开始的中断指令INT 20H能使当前程序返回操作系统。为此在程序的模块中设置了标准顺序,它首先将指向INT 20H的DS:0H压入堆栈,而后RET执行远调用将原来的DS:0H弹出并赋予CS:IP,于是程序指针指向指令INT 20H,程序就能正常地退回DOS了。

扩展名为.com的文件也是DOS下的可执行文件,它与EXE文件有不同的结构。

COM文件的源程序格式如下:

```
; 源程序的NAME或TITLE
; 源文件的注释部分
; 表达式赋值语句(EQU)部分
PROGRAM SEGMENT                  ; 定义代码段开始
     ORG 100H                    ; 地址指针指向偏移地址100H处
     ASSUME  CS: PROGRAM,DS: PROGRAM,SS: PROGRAM,ES: PROGRAM
     MAIN  PROC  NEAR            ; 主程序开始,近调用
                                 ; 程序的主体部分
           MOV    AX,4C00H       ; 返回DOS
           INT 21H
     MAIN  ENDP                  ; 主程序结束
                                 ; 数据定义部分
PROGRAM  ENDS                    ; 代码段结束
     END  MAIN                   ; 结束汇编
```

从上面的结构可以看出，COM 文件具有两个特点：①COM 文件的 4 个段是重叠在一起的同一个段；②COM 文件在装入内存时也产生一个程序段前缀。与 EXE 文件不同的是，程序段前缀是文件的一部分，被放在文件前面的 100H 字节中。

COM 文件被装入后，CS、DS、ES、SS 都设置为指向程序段前缀的段地址，IP 固定为 100H，整个程序只占一个物理段(64KB)，SP 指向这个物理段的末尾，并在栈顶存放了两字节 00H。对所有的过程都定义为近调用 NEAR 型。如果编写的程序符合 COM 文件的规定，经汇编、连接后生成的 EXE 文件就可以直接转换为 COM 文件。

5.6　汇编语言程序基本结构设计

5.6.1　程序基本结构

1966 年，Bohra 和 Jacopini 提出了以下 3 种基本结构，用这 3 种基本结构表示一个良好算法的基本单元。

1. 顺序结构

如图 5.4 所示，虚线框内是一个**顺序结构**。其中，A 和 B 两个框是顺序执行的，即在执行完 A 框所指定的操作后，必然接着执行 B 框所指定的操作。顺序结构是最简单的一种基本结构。

2. 选择结构

选择结构又称**分支结构**，如图 5.5 所示。虚线框内是一个选择结构，此结构中必包含一个判断框。根据给定的条件 P 是否成立而选择执行 A 框或 B 框。注意，无论 P 条件是否成立，都只能执行 A 框或 B 框之一，不可能既执行 A 框又执行 B 框。无论走哪一条路径，在执行完 A 框或 B 框之后，都经过 b 点，然后脱离本选择结构。A 框和 B 框中可以有一个是空的，即不执行任何操作。

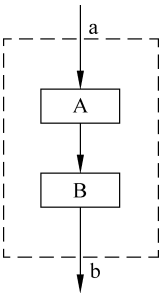


图 5.4　顺序结构图

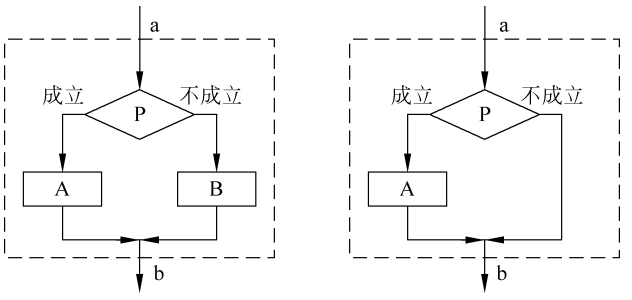


图 5.5　选择结构图

3. 循环结构

循环结构又称重复结构，即反复执行某一部分的操作。循环结构分为当型(WHILE 型)循环结构和直到型(UNTIL 型)循环结构两种。

5.6.2　顺序结构程序设计

顺序结构程序是最简单的程序，在顺序结构程序中，完全按照指令先后顺序逐条执行。

这在程序段中是大量存在的,但作为完整的程序则很少见,一般作为程序的部分使用。

【例 5.2】 编程序计算。

例如,SUM=3*(X+Y)+(Y+Z)/(Y-Z),其中,X、Y、Z 都是 16 位无符号数,要求结果存入 SUM 单元。假设运算过程中,中间结果都不超出 16 位二进制数的范围。程序片段为:

```
MOV  AX,X              ; 取 X
ADD  AX,Y              ; AX←X + Y,乘法操作数 1
MOV  CX,3              ; 乘法操作数 2
MUL  CX                ; (DX,AX)←3 * (X + Y)
MOV  CX,AX             ; CX←3 * (X + Y)保存
MOV  AX,Y              ; 取 Y
ADD  AX,Z              ; AX←Y + Z,被除数
XOR  DX,DX             ; DX←0
MOV  BX,Y              ; 取 Y
SUB  BX,Z              ; BX←Y - Z,除数
DIV  BX                ; AX←(Y + Z)/(Y - Z)的商
ADD  AX,CX             ; AX←3 * (X + Y) + (Y + Z)/(Y - Z),两项之和
MOV  SUM,AX            ; 存结果
```

【例 5.3】 将两字节数据相加,存放到一个结果单元中,并显示十六进制结果。

```
DATA    SEGMENT
      AD1 DB  4CH            ; 定义第 1 个加数
      AD2DB  25H             ; 定义第 2 个加数
      SUM  DB  ?             ; 定义结果单元
DATA    ENDS
CODE    SEGMENT
      ASSUME  CS:  CODE,DS: DATA
START:  MOV  AX,DATA
      MOV    DS,AX
      MOV    AL,AD1          ; AL←AD1
      ADD    AL,AD2          ; AL←AD1 + AD2
      MOV    SUM,AL          ; 将结果存放在 SUM 单元
      MOV    BL,AL           ; 显示十六进制结果
      MOV    CL,4            ; 取二进制高 4 位
      SHR    AL,CL
      AND    AL,0FH
      ADD    AL,30H          ; 高位十六进制 ASCII 码值
      MOV    DL,AL           ; 输出高位
      MOV    AH,2
      INT    21H
      MOV    AL,BL           ; 取二进制低 4 位
      AND    AL,0FH
      ADD    AL,30H          ; 低位十六进制 ASCII 码值
      MOV    DL,AL           ; 输出低位
      MOV    AH,2
```

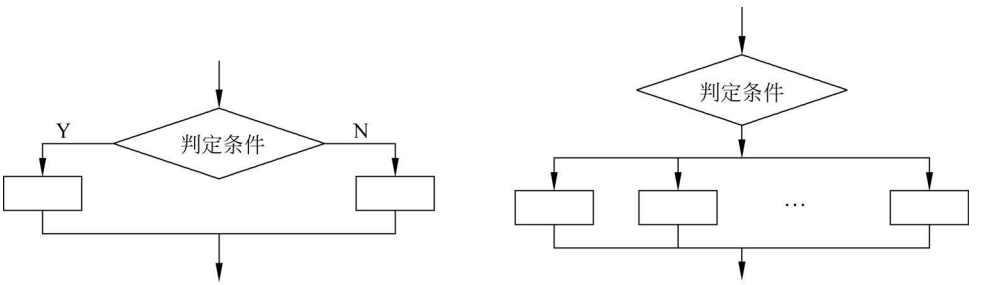
```
        INT     21H
        MOV     AH,4CH          ; 返回 DOS
        INT     21H
CODE    ENDS
        END     START
```

本程序采用了 DOS 中断调用的 4CH 号功能来退出程序段运行，返回 DOS 现场。这是一种常用的执行程序返回 DOS 现场的方法。

5.6.3　分支结构程序设计

1. 分支程序的结构形式

分支程序结构可以有两种形式，如图 5.6 所示。



(a) IF…THEN…ELSE结构　　(b) CASE结构

图 5.6　分支程序的结构形式

它们分别相当于高级语言中的 IF…THEN…ELSE 语句和 CASE 语句，适用于要求根据不同条件做不同处理的情况。IF…THEN…ELSE 语句可以引出两个分支；CASE 语句则可以引出多个分支。无论哪一种形式，它们的共同特点是：运行方向是向前的，在某一种特定条件下，只能执行多个分支中的一个分支。

2. 分支程序设计方法

程序的分支一般用条件转移指令来产生，利用转移指令不影响条件码的特性，连续地使用条件转移指令可以使程序产生多个不同的分支，如例 5.8 所示。

【例 5.4】 编制程序实现符号函数。

$$Y=\begin{cases}1, & X>0\\0, & X=0\\-1, & X<0\end{cases}\quad -128\leqslant X\leqslant +127$$

程序中要求对 X 的值加以判断，根据 X 的不同值，给 Y 单元赋予不同的值。程序流程图如图 5.7 所示。

程序部分代码如下：

```
        CMP   X,0
        JL    PNUM              ; X<0 转移到 PNUM
        JZ    ZERO              ; X=0 转移到 ZERO
```

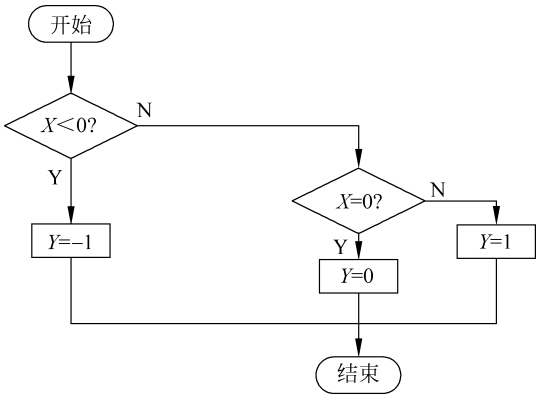


图 5.7　例 5.4 程序流程图

```
        MOV   Y,1              ; X>0 时,给 Y 单元赋值 1
        JMP   EXIT             ; 跳转到程序结束位置,结束程序
PNUM:   MOV   Y,-1             ; X<0 时,给 Y 单元赋值 -1
        JMP   EXIT             ; 跳转到程序结束位置,结束程序
ZERO:   MOV   Y,0              ; X=0 时,给 Y 单元赋值 0
EXIT:     ⋮                    ; 程序结束的代码
```

3. 跳跃表法

分支程序的两种结构形式都可以用上面所述的方法来实现。此外,在实现 CASE 结构时,还可以使用跳跃表法,使程序能够根据不同的条件转移到多个程序分支中,下面举例说明。

【例 5.5】 试根据 AL 寄存器中哪一位为 1(从低位到高位)把程序转移到 8 个不同的程序分支中。

下面列出了用变址寻址方式实现跳跃表法的程序。

```
DATA   SEGMENT
DATATAB     DW ROUTINE_1
        DW ROUTINE_2
        DW ROUTINE_3
        DW ROUTINE_4
        DW ROUTINE_5
        DW ROUTINE_6
        DW ROUTINE_7
        DW ROUTINE_8
DATA     ENDS
CODE     SEGMENT
MAIN   PROCFAR
        ASSUMECS: CODE,DS: DATA
START:PUSH   DS
        SUB     BX,BX
        PUSH    BX
```

```
        MOV       BX,DATA
        MOV       DS,BX
        CMP       AL,0
        JE        CONT
        MOV       SI,0
    LP: SHR       AL,1
        JNB       NOT_YET
        JMP       DATATAB[SI]
NOT_YET: ADD      SI,TYPE  BRANCH  TABLE
        JMP       LP
    CONT:
  ⋮

ROUTINE_1:   ⋮
ROUTINE_2:   ⋮
  ⋮
    RET
    MAIN  ENDP
    CODE  ENDS
    END     START
```

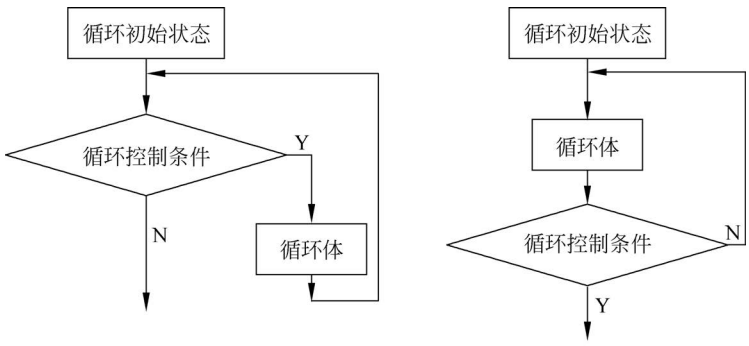
跳跃表法是一种很有用的分支程序设计方法。此外，还可以使用寄存器间接和基址变址寻址方式来达到同一目的。以上实现分支程序的方法并无实质的区别，只是其中关键的 JMP 指令所用的寻址方式不同而已。

5.6.4　循环结构程序设计

1. 循环程序结构

如果程序中有需要多次重复执行的程序段，则往往将它们设计为循环结构，这样不但使程序结构清晰，而且减少源程序的书写，节省占用的内存空间。

循环程序结构可以总结为两种结构形式：一种是 WHILE…DO 结构形式；另一种是 DO…UNTIL 结构形式，如图 5.8 所示。



(a) WHILE…DO结构　　(b) DO…UNTIL结构

图 5.8　循环程序的结构形式

1) WHILE…DO 结构

WHILE…DO 结构把对循环控制条件的判断放在循环的入口,先判断条件,满足条件就执行循环体,否则就退出循环。

2) DO…UNTIL 结构

DO…UNTIL 结构则先执行循环体,然后再判断控制条件,不满足条件则继续执行循环操作,一旦满足条件则退出循环。

这两种结构可以根据具体情况选择使用。如果循环次数等于 0,则应选择 WHILE…DO 结构,否则使用 DO…UNTIL 结构。

循环程序都可由如下三部分组成。

(1) 循环初始状态设置。它为循环做好必要的准备工作,以保证循环在正确的初始条件下开始工作。这部分完成的工作主要是设置循环次数、给地址指针赋初值、累加器清零、进位标志清零等。

(2) 循环工作部分。循环工作部分即需要重复执行的程序段,这部分是循环的主体。它是针对具体问题而设计的程序段,从初始状态开始,动态地执行相同的操作。

(3) 循环修改部分。它与循环工作部分协调配合,通过修改或恢复计数器、寄存器、操作数地址指针等,保证每一次循环时,参加执行的信息能发生有规律的变化。

循环工作部分和循环修改部分合称循环体。

2. 循环控制方法

每个循环程序必须选择一个循环控制条件来控制循环的运行和结束,而合理地选择该控制条件就成为循环程序设计的关键。有时,循环次数是已知的,此时可以用循环次数作为循环的控制条件,通过使用 LOOP 指令能够很容易地实现这种循环程序;某些情况下,虽然循环次数是已知的,但有可能通过其他特征或条件来提前结束循环,可以使用 LOOPZ 和 LOOPNZ 指令实现这种循环程序设计。然而,有时循环次数是未知的,那就需要根据具体情况找出控制循环结束的条件。循环控制条件的选择是很灵活的,有时可供选择的方案不止一种,此时就应分析比较,选择一种效率最高的方案来实现。

控制循环的执行并判断是否结束循环的方法主要有 3 种:计数控制、条件控制和逻辑尺控制。

1) 计数控制

计数控制是一种最常用的循环控制方法,适用于事先已知循环次数的情况。既可用循环指令 LOOP 实现,也可用条件转移指令实现。

【例 5.6】 在首地址为 BUFF 的内存缓冲区中,存放着 20H 个带符号的字数据。要求找出其中的最小值,并将最小值存入 MIN 单元。

对于这个问题,要找最小值,就要逐个比较这 20H 个数据,所以可用循环结构程序重复比较过程。比较的方法是:可以先假定第一个数据就是最小值(当前最小值),然后和其余数据比较,如果比当前最小值大,则不处理;否则将该数据置换为当前最小值,直至所有的数据都比较完。显然,这个循环的循环次数是 1FH。

程序片段如下:

```
        LEA     SI,BUFF              ; 设地址指针
        MOV     CX,20H               ; CX←循环次数
        MOV     AX,[SI]              ; AX←第一个数据
        INC     SI
        INC     SI                   ; SI 指向第二个数据
        DEC     CX                   ; 修改循环次数计数器
AGAIN:  CMP     AX,[SI]
        JLE     NEXT                 ; 小于或等于时转移
        MOV     AX,[SI]
NEXT:   INC     SI
        INC     SI                   ; 修改地址指针指向下一个数据
        LOOP    AGAIN
        MOV     MIN,AX
```

2）条件控制

条件控制适用于事先不知道循环次数的情况，但可以用给定的某种条件来判断是否结束循环。

【例 5.7】 编程统计 AX 寄存器中 1 的个数，并将结果存入 SUM 单元。

要统计二进制数中 1 的个数，最方便的方法是将这个数的各位依次移入 CF 标志，通过检测 CF 的值来判断该位是否为 1，以此统计所含 1 的个数。这是一个重复计数的过程，可以用循环程序实现。对循环的控制，可以用计数方法，共检测 16 次；还可以通过判断移位后二进制数是否变为 0 作为循环结束的条件。当二进制数的后几位全部为 0 时，用这种方法可以提前结束循环，提高程序的运行效率。程序流程图如图 5.9 所示。

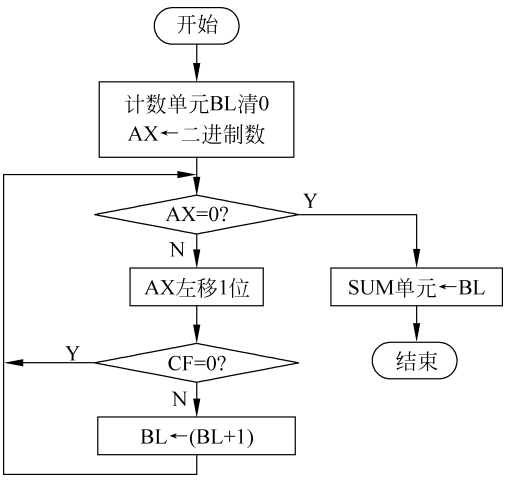


图 5.9　例 5.7 程序流程图

程序片段如下：

```
        MOV   BL,0              ; 计数单元 BL 清 0
AGAIN:  OR    AX,AX             ; 测试 AX 是否为 0
        JZ    EXIT              ; 若 AX = 0,则转移到结束点
```

```
        SHL   AX,1                ; 将 AX 最高位移至 CF
        JNC   NEXT                ; CF = 0,转去 AGAIN 继续
        INC   BL                  ; CF≠0,BL 加 1
NEXT:   JMP   AGAIN
EXIT:   MOV   SUM,BL
```

3) 逻辑尺控制

有时候,循环体内的处理任务在每次循环执行时并无规律,但确实需要连续运行。此时,可以给各处理操作标以不同的特征位,所有特征位组合在一起,就形成了一个逻辑尺。

【例 5.8】 在数据段中有两个数组 X 和 Y,每个数组含有 10 个双字节数据元素。现将两个数组的对应元素进行下列计算,形成一个新的数组 M。假定数组的对应元素计算后,结果不产生溢出。

$$M1=X1+Y1 \quad M2=X2+Y2 \quad M3=X3-Y3$$
$$M4=X4+Y4 \quad M5=X5-Y5 \quad M6=X6-Y6$$
$$M7=X7-Y7 \quad M8=X8+Y8 \quad M9=X9+Y9$$
$$M10=X10-Y10$$

很显然,这个问题可以用循环实现,而且循环次数确定为 10 次。但每次循环的操作是进行加还是减,无规律可循。为此,可以为每一次操作设置一个特征位,即 0 表示加,1 表示减,构成一个 16 位的逻辑尺,存放于 DX 寄存器中。本例逻辑尺为:

0010111001000000

从左到右依次为数组元素 1～10 的特征位。每次将逻辑尺左移 1 位,根据移入 CF 的特征位,判断本次循环体所进行的操作。程序流程图如图 5.10 所示。

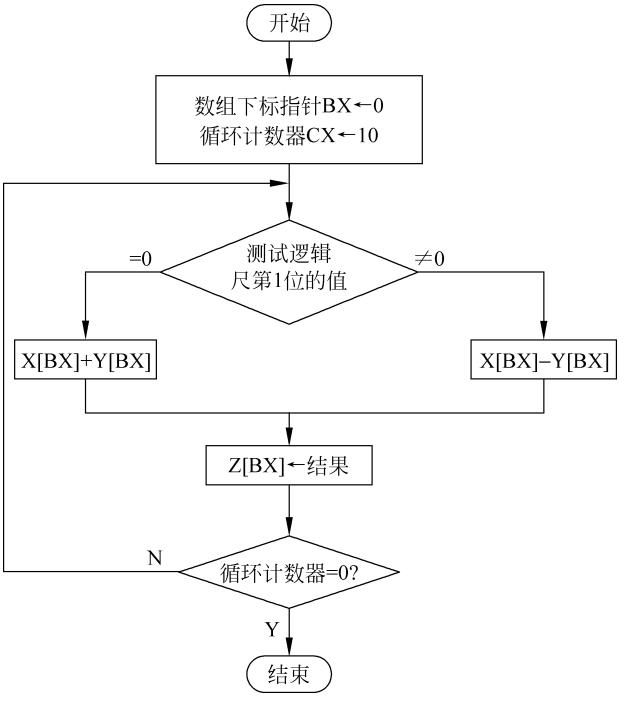


图 5.10 例 5.8 程序流程图

程序片段如下：

```
        MOV     BX,0            ; 设数组下标指针
        MOV     CX,10           ; 设循环计数器
AGAIN:  MOV     AX,X[BX]
        SHL     DX,1
        JC      SUBB            ; 若当前特征位为 1,则做减法; 否则做加法
        ADD     AX,Y[BX]
        JMP     NEXT
SUBB:   SUB     AX,Y[BX]
NEXT:   MOV     M[BX],AX        ; 送结果
        INC     BX
        INC     BX
        LOOP    AGAIN
```

3. 双重循环程序设计

【例 5.9】 编制程序实现延时 1ms。

延时程序就是让计算机执行一些空操作或无用操作，来占用 CPU 的时间，从而达到延时的目的。延时程序通常用循环程序实现。程序片段如下：

```
        MOV   CX,374
DELAY1: PUSHF               ; 10T
        POPF                ; 8T
        LOOP  DELAY1        ; 3.4T
```

上面程序段的循环体和循环控制部分由指令 PUSHF、POPF 和 LOOP 构成。这 3 条指令执行所花费的时钟周期个数和为 10＋8＋3.4＝21.4。如果 CPU 的主频为 8MHz，那么它的时钟周期为 0.125μs；如果要实现延时 1ms，则该循环体重复执行的次数为：

$$\text{循环次数}=1\text{ms}/(0.125\mu\text{s}\times 21.4)\approx 374$$

如果要延时 100ms，那么只需将这个程序再执行 100 次，从而构成一个双重循环。其程序片段如下：

```
SOFTDLY PROC  MOV     BL,100          ; 4T
DELAY2:       MOV     CX,374          ; 4T
DELAY1:       PUSHF                   ; 10T,标志寄存器进栈,内层循环,循环 374 次
              POPF                    ; 8T
              LOOP    DELAY1          ; 3.4T
              DEC     BL              ; 2T
SOFTDLY PROC  JNZ     DELAY2          ; 8T,外层循环,循环 100 次
```

显然，该程序的准确延时时间＝4T＋100(4T＋21.4T×374＋2T＋8T)＝100.22ms。

5.7　汇编语言子程序设计

如果在一个程序的多个位置，或者在多个程序的多个位置中用到了同一段程序，则可将这段程序抽取出来，单独存放在某一区域，每当需要执行这段程序时，就用调用指令转到这

段程序,执行完毕再返回原来的程序。抽取出来的这段程序称为**子程序**或**过程**,而调用它的程序称为**主程序**或**调用程序**。主程序向子程序的转移过程称为**子程序调用**或**过程调用**。

使用子程序是程序设计的一种重要方法。子程序的引入使程序功能的层次性更加分明,增强了程序的可读性,为较大软件设计的分工合作提供了方便。

5.7.1 子程序的定义

子程序的定义由过程定义伪指令 PROC/ENDP 实现,其格式为:

```
过程名  PROC[NEAR/FAR]
        语句1    ;
        语句2    ;   n条语句序列构成的过程体
        ⋮        ⋮
        语句n    ;
        RET
过程名  ENDP
```

所定义的子程序可以和主程序在同一个代码段内,也可以在不同的代码段内。

说明:

① 过程名(子程序名)用以标识不同的过程,是一个用户自定义的标识符号。

② PROC 与 ENDP 相当于一对括号,将子程序的处理过程(过程体)括在其中。过程体是一段相对独立的程序,是完成子程序功能的主体。过程的最后一条指令必须是 RET(返回指令)。

③ NEAR 或 FAR 是子程序属性的说明参数。NEAR 属性的子程序只允许段内调用,这时,子程序的定义必须和调用它的主程序在同一代码段内;FAR 属性的过程允许段间调用,即允许其他段的程序调用。过程的属性决定了调用指令 CALL 和返回指令 RET 的操作。

5.7.2 子程序的调用和返回

子程序的调用和返回由 CALL 指令和 RET 指令实现。从不同的角度,可对子程序的调用进行以下分类。

(1) 段内调用与段间调用。段内调用中,在子程序和调用返回过程中,转移地址和返回地址不涉及 CS 的变化,只通过 IP 内容的变化实现程序的转移和返回。

段间调用中,由 CS 和 IP 的变化共同决定程序的转移和返回。

显然,当主程序和子程序处于同一代码段时,可以把子程序定义为 NEAR 属性或 FAR 属性;而当主程序与子程序不在同一代码段时,子程序必须被定义为 FAR 属性。

(2) 直接调用与间接调用。当调用指令使用过程名调用某过程时,调用时通过把该过程的指令入口地址送入 CS 和 IP(段内调用仅修改 IP)来实现,这个调用过程称为直接调用。

当调用指令是通过某个寄存器或存储器单元指出被调用子程序的入口地址时,这个调用过程称为间接调用。间接调用可分为寄存器间接调用和存储器间接调用。

在实际使用时,直接调用因方便清楚而使用较多。无论采用哪种调用方式,为了能保证子程序执行完后顺利地返回主程序,CALL 指令在将控制转移到子程序之前,都将自动保护

返回地址。**返回地址**也称为**断点**，是 CALL 指令下一条指令的第一个字节地址（段内调用仅保存 IP，段间调用保存 CS 和 IP），然后才转入子程序执行。待执行完子程序后，RET 指令负责把保护的返回地址（即断点）恢复到 CS 和 IP 中（段内调用仅需恢复 IP），继续执行主程序。断点的保护和返回是通过堆栈指令 PUSH 和 POP 实现的。

子程序示例如下：

```
CODE1   SEGMENT
        ⋮
        CALL PROC1
AAA:
        ⋮
        PROC1   PROC
        ⋮
              RET
        PROC1   ENDP
        PROC2   PROC   FAR
        ⋮
              RET
        PROC2   ENDP
CODE1   ENDS
CODE2   SEGMENT
        ⋮
        CALL   PROC2
BBB:
        ⋮
CODE2   ENDS
```

在以上程序段中，CALL PROC1 是段内调用，CALL PROC2 是段间调用。AAA 和 BBB 是两个返回地址。子程序 PROC1 返回后，从 AAA 处开始执行；子程序 PROC2 返回后，从 BBB 处开始执行。

5.7.3　编写子程序时的注意事项

由于子程序可在程序的不同位置或在不同的程序中被多次调用，因此对于子程序的设计提出了很高的要求，如通用性强、独立性好、程序目标代码短、占用内存空间少、执行速度快、结构清晰，以及有详细的功能说明等。在设计子程序时，需要注意以下几点：

（1）信息保护。为了保证由子程序返回主程序后程序执行的正确性，通常要将子程序中用到的寄存器压入堆栈保护，子程序执行完成后再恢复出来。将寄存器压入堆栈保护的过程称为**保护现场**，将寄存器从堆栈中弹出恢复的过程称为**恢复现场**。保护和恢复现场的工作既可以在调用程序中进行，也可以在子程序中进行。

（2）子程序的说明。为了方便各类用户对子程序的调用，一个子程序应该有清晰的文本说明，以提供给用户足够的使用信息。通常子程序的文本说明包括以下一些内容。

① 子程序名。

② 子程序功能、技术指标。

③ 子程序的入口、出口参数。

④ 子程序使用到的寄存器和存储单元。

⑤ 是否又调用其他子程序。

⑥ 子程序的调用形式。

(3) 汇编语言子程序无参数。子程序中指令访问的存储单元寄存器等与主程序中访问的是同一物理设备,无需参数传递数据。

5.7.4 子程序举例

【例 5.10】 两个 16 位十进制数以压缩 BCD 码的形式存放在内存中,求它们的和。

可以通过 8 次字节数相加,每次相加后再进行十进制调整来实现。

```
DATA  SEGMENT
      DAT1   DB 34H,67H,98H,86H,02H,41H,59H,23H      ; 低位在前
      DAT2   DB 33H,76H,89H,90H,05H,07H,65H,12H      ; 低位在前
      SUM    DB 10 DUP(0)
DATA  ENDS
STACK SEGMENT  PARA  STACK
      DW  20H DUP(0)
STACK ENDS
CODE  SEGMENT
      ASSUME  CS: CODE,DS: DATA,SS: STACK
START: MOV   AX,DATA
      MOV    DS,AX
      MOV    CX,8                                    ; 设子程序入口参数
      CALL   ADDP                                    ; 调用加法子程序
      MOV    AH,4CH                                  ; 返回 DOS
      INT    21H
ADDP    PROC                                         ; 加法子程序,完成两位十进制数相加
      PUSH   AX                                      ; 保护现场
      PUSH   BX
      CLC                                            ; 清除进位标志
      MOV    BX,0
AGAIN: MOV   AL,DAT1[BX]                             ; 相加
      ADC    AL,DAT2[BX]
      DAA                                            ; 十进制调整
      MOV    SUM[BX],AL                              ; 存结果
      INC    BX                                      ; 修改下标
      LOOP   AGAIN                                   ; 循环执行 8 次
      ADC    SUM[BX],0
      POP    BX                                      ; 恢复现场
      POP    AX
      RET                                            ; 返回主程序
ADDP  ENDP
CODE  ENDS
      END    START
```

5.7.5　子程序的嵌套和递归调用

在一个子程序中又调用其他的子程序，这种情况称为子程序的**嵌套**。只要堆栈允许，嵌套的层次就可以不加限制。图 5.11 所示为一个两层的子程序嵌套调用示意图。

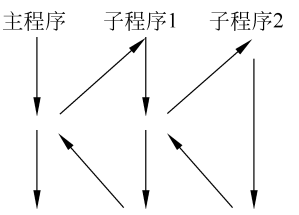


图 5.11　子程序嵌套调用示意图

在子程序嵌套调用时，每一个子程序执行完后都要返回上一级调用程序，所以对于堆栈的使用要格外小心，以防出现不能正确返回的错误。

所谓子程序的**递归**调用，就是指在子程序嵌套调用时，调用的子程序就是它本身。递归子程序和数学上递归函数的定义相对应，必须有一个结束的条件。在递归的过程中，每一次调用所用到的调用参数和运行结果都不相同，必须将本次调用的这些信息存放在堆栈中，这些信息称为一帧，下一次的调用必须保证这帧信息不被破坏。当递归满足结束条件时，开始逐级返回，每返回一级，就从堆栈中弹出一帧信息，计算一次中间结果。在递归结束后，堆栈恢复原状。

子程序的递归调用会用到大量的堆栈单元，因此要特别注意堆栈的溢出。在编制程序时，可以采取一些保护措施。

在实际应用中，程序结构往往不是单一的结构，而是多种结构的复合，应根据具体情况和要求做出合理的设计。

5.8　系统功能调用

为了减少程序设计的复杂度，微机系统提供了一些**系统功能子程序**，用户通过调用这些系统功能子程序，可以方便地实现对底层硬件接口的操作，从而提高汇编语言源程序的设计效率。

微机系统提供两组功能程序：一组固化在基本 I/O 系统(Basic Input and Output System，BIOS)内；另一组在 DOS(Disk Operating System)系统内。这些功能程序其实是由几十个内部子程序组成的，它们能完成对 I/O 设备、文件、作业、目录等的管理和操作。程序员不必了解所使用设备的物理特性、接口方式及内存分配等，不必编写烦琐的控制程序，在程序需要的地方，可直接调用，实现相应功能。

使用这些系统功能子程序编写的程序简单、清晰，可读性好，而且代码紧凑，调试方便。

5.8.1　系统功能调用方法

为了调用这些功能子程序，操作系统提供了一个调用接口，通过软中断指令来实现。

格式：

```
INT n
```

其中，n 是中断类型码。当 $n=5\sim1FH$ 时，调用 BIOS 中的服务程序，称为**系统中断调用**；

当 $n=20\sim3FH$ 时,调用 DOS 中的服务程序,称为**功能调用**。每一个不同的中断类型码,又包含若干子功能。为区分这些子功能,系统给每一个子功能分配一个功能号,要求在调用前将这个功能号送入 AH 寄存器。对于需要使用入口参数的功能调用,还要事先设置入口参数。它们的调用方法如下。

(1) 送分功能号给 AH。

(2) 设置入口参数。

(3) INT n。

每执行一条软中断指令,就调用一个相应的中断服务程序。

5.8.2 BIOS 调用

BIOS 是固化在 ROM 中的一组 I/O 服务程序,除系统测试程序、初始化引导程序及部分中断向量装入程序外,还为用户提供了常用设备的输入/输出程序,如键盘输入、打印机及显示器输出等。表 5.2 所示为部分常用的 BIOS 功能调用的简要说明。

表 5.2 部分常用的 BIOS 功能调用

软中断指令	功 能	软中断指令	功 能
INT 00H	除法出错	INT 0DH	硬盘中断
INT 01H	单步中断	INT 0EH	软盘中断
INT 02H	非屏蔽中断	INT 10H	显示器中断
INT 03H	断点中断	INT 12H	内存大小检查
INT 04H	溢出中断	INT 15H	盒式磁带机 I/O
INT 09H	键盘中断	INT 16H	键盘输入
INT 0BH	异步通信串行口 1 中断	INT 17H	打印机输出
INT 0CH	异步通信串行口 2 中断	INT 1AH	时钟

5.8.3 DOS 系统功能调用

DOS 系统功能调用和 BIOS 调用某些功能是类似的,相比较而言,DOS 系统功能调用还增加了许多必要的检测,因此比 BIOS 调用方便、操作简易、对硬件的依赖性少。

其中,

```
INT 21H
```

是一个具有调用多种功能服务程序的软中断指令,称为 DOS 系统功能调用。它内部又包含 80 多个子功能,大致可以分为设备管理、目录管理、文件管理和其他功能。用户可根据功能号区分调用。下面对几个常用的系统功能调用进行简单介绍。

1. 带显示的单字符键盘输入(01H 号功能)

此功能程序等待键盘输入,若有字符键按下,将输入字符的 ASCII 码送入 AL 寄存器,并在屏幕上显示。如果按下的键是 Ctrl+C 组合键,则停止程序运行;如果按下的键是 Tab 键,则屏幕上的光标自动移至下一个制表位。

入口参数:无

出口参数：AL＝输入字符的 ASCII 码

格式：

```
MOV AH,01H
INT 21H
```

2. 输出单字符(02H 号功能)

在屏幕上显示输出 DL 寄存器中的字符。如果 DL 中是 Backspace 键编码，则光标向左移动一个位置，并使该位置显示空格；如果是其他字符，则显示该字符。

入口参数：DL＝输出字符

出口参数：AL＝37H 成功

格式：

```
MOV DL,'A'        ; A 字符的 ASCII 码置入 DL 中
MOV AH,2
INT 21H
```

3. 不带显示的单字符键盘输入(07H 号)

07H 号与 01H 号功能类似，区别仅仅是输入的字符不在屏幕上显示。并且，07H 号功能调用对 Ctrl＋C 组合键和 Tab 键无反应。

入口参数：无

出口参数：AL＝输入字符

格式：

```
MOV  AH: 07H
INT  21H
```

或

```
MOV  AH,08H
INT  21H
```

4. 字符串输出(09H 号功能)

09H 号功能是在屏幕上显示输出字符串。它要求事先将要显示的字符串的段地址和段内偏移地址送入 DS 和 DX 寄存器，并且该字符串应以'$'结尾。

入口参数：DS＝字符串所在段的段基值

　　DX＝字符串的段内偏移量

出口参数：无

格式：

```
MOV  DX,字符串偏移量
MOV  AH,09H
INT  21H
```

例如:

```
STRING DB 'A EXAMPLE'0DH,0AH,'$'
 ⋮
MOV    DX,OFFSET STRING
MOV    AH,09H
INT    21H
```

5. 字符串输入(0AH 号功能)

从键盘输入一串字符到内存缓冲区,输入的字符串以 Enter 键结束。内存缓冲区的第一个字节内容由用户设置,设置为所能接收的最大字符个数(1～255);第二个字节预留,由系统填充实际输入的字符个数(Enter 键除外);从第三个字节开始,存放从键盘输入的字符。若输入的字符个数大于所能接收的最大字符个数,则系统发出响铃提示,多余的字符被略去;若输入的字符个数小于所能接收的最大字符个数,则空出的位置补零。

入口参数:DS:DX=缓冲区首址

[DS:DX]=缓冲区最大字符个数

出口参数:[DS:DX+1]=实际输入的字符个数

[DS:DX+2]单元开始存放实际输入的字符

格式:

```
MOV   DX,缓冲区偏移量
MOV   DS,缓冲区段基址
MOV   AH,0AH
INT   21H
```

例如:

```
BUFDB 30,?,30 DUP(?)
 ⋮
MOV   DX,OFFSET BUF
MOV   DS,SEG  BUF
MOV   AH,0AH
INT   21H
```

6. 返回操作系统(4CH 号功能)

4CH 号功能是将控制返回操作系统。

入口参数:AL=返回码

出口参数:无

格式:

```
MOV   AH,4CH
INT   21H
```

5.9　宏　指　令

为了简化程序的设计，可以将汇编语言源程序中多次重复使用的程序段用宏指令来代替，即**宏定义**。**宏指令**是指程序员事先定义的特定的“指令”，这种“指令”是一组重复出现的程序指令块的缩写和替代。宏指令定义以后，凡在宏指令出现的位置，宏汇编程序总是自动地把它们替换为对应的程序指令块，这个引用过程称为**宏调用**。汇编程序在汇编时遇到宏调用语句时，将把宏调用语句展开，即将宏定义的代码段插入宏调用语句的位置取而代之，这个过程称为**宏展开**。因此，宏的操作必定经过 3 个步骤：宏定义、宏调用和宏展开。

使用宏指令的优点是：简化源程序的编写，传递参数特别灵活，功能更强。

5.9.1　宏定义

宏指令是源程序中的一段具有独立功能的程序代码。它只要在源程序中定义一次，就可以多次调用，调用时只要使用一条宏指令语句就可以了。宏指令定义由开始伪指令 MACRO、宏指令体、宏指令定义结束伪指令 ENDM 组成。其格式为：

```
宏指令名   MACRO[形式参数 1,形式参数 2,…,形式参数 N]
          ⋮          ;宏指令体(简称宏体)
          ENDM
```

其中，宏指令名是宏定义为宏体程序指令块规定的名称，既可以是任一合法的名称，也可以是系统保留字（如指令助记符、伪指令运算符等），当宏指令名是某个系统保留字时，该系统保留字就被赋予新的含义，从而失去原有的意义。MACRO 语句到 ENDM 语句之间的所有汇编语句构成**宏指令体**，简称**宏体**，宏体中使用的形式参数必须在 MACRO 语句中列出。

形式参数是宏体内某些位置上可以变化的符号，可以默认，也可以有一个或多个。宏指令定义一般放在源程序的开头，以避免不应发生的错误。

宏指令必须先定义后调用。宏指令可以重新定义，也可以嵌套定义。嵌套定义是指在宏指令体内还可以再定义宏指令或调用另一宏指令。

【例 5.11】　定义一条从键盘输入一个字符的宏指令 INPUT。

```
INPUT  MACRO
       MOV  AH,01H
       INT  21H
       ENDM
```

采用宏指令语句 INPUT 编程，类似于高级语言语句。

【例 5.12】　定义一条换行宏指令 LF。

```
LF  MACRO
    MOV  DL,10
    MOV  AH,02H
    INT  21H
    ENDM
```

【例 5.13】 定义一条回车宏指令 CR。

```
CR  MACRO
    MOV  DL,13
    MOV  AH,02H
    INT  21H
    ENDM
```

5.9.2 宏调用

宏指令一旦定义后,就可以用宏指令名来调用了,宏调用的格式为:

```
宏指令名  [实际参数 1,实际参数 2,……,实际参数 N]
```

其中,实际参数的类型和顺序要与形式参数的类型和顺序保持一致,宏调用时将一一对应地替换宏指令体中的形式参数。宏指令调用时,实际参数的数目并不一定要和形式参数的数目一致,当实参个数多于形参的个数时,将忽略多余的实参;当实参个数少于形参个数时,多余的形参用空串代替。

【例 5.14】 定义一条 INOUT 宏指令,通过调用它,既可以输入一串字符,也可以显示一串提示字符。

宏定义:

```
INOUT  MACRO  X,Y
       MOV    AH,X          ; X 为功能号
       LEA    DX,Y          ; Y 为偏移量
       INT    21H
       ENDM
```

宏调用:

```
DATAS  SEGMENT
       INPUT   DB 'PLEASE INPUT ANY CHARACTERS: ','$'
       KEYBUF   DB 10,11 DUP(?),13,10,'$'
DATAS  ENDS
CODES  SEGMENT
START: ⋮
       INOUT   09H,INPUT           ; 宏调用,09H 号功能,显示字符串
       LF                          ; 调用例 5.2 宏指令,换行
       CR                          ; 调用例 5.3 宏指令,回车
       INOUT   0AH,KEYBUF          ; 宏调用,0AH 号功能,接收键盘输入字符串
       INOUT   09H,KEYBUF + 2      ; 宏调用,09H 号功能,显示输入的字符串
       ⋮
CODES  ENDS
    END  START
```

5.9.3　宏展开

宏汇编程序遇到宏定义时并不对它进行汇编，只有在程序中进行宏调用时，汇编程序才把对应的宏指令体调出进行汇编处理(语法检查和代码块的插入)，这个过程称为**宏展开**(或**宏扩展**)。宏指令调用后，在宏指令调用处将产生用实参替换形参的宏体指令语句。

在 MASM 汇编生成列表文件(.lst)的每行中间用符号“+”作为标志，表明本行语句为宏指令展开生成的语句。例如，上述 INOUT 宏指令调用后，宏展开后的语句为：

```
+   MOV   AH,9
+   LEA   DX,INPUT
+   INT   21H
+   MOV   DL,10
+   MOV   AH,2
+   INT   21H
+   MOV   DL,13
+   MOV   AH,2
+   INT   21H
+   MOV   AH,10
+   LEA   DX,KEYBUF
+   INT   21H
+   MOV   AH,9
+   LEA   DX,KEYBUF + 2
+   INT   21H
```

这里，实际参数以整体去替换形式参数的整体(即对应符号的整体代替)。如果只希望以数值代替形式参数，则可使用特殊宏计算符号“&”和“%”。

5.10　实用程序设计举例

1. 数值运算程序设计示例

数值运算程序就是利用加、减、乘、除及十进制调整指令对数值进行运算，是较为基础的一类程序设计。

【例 5.15】 假设有两个 3 位十进制数，以非压缩 BCD 码的形式表示，分别存放在 SUB1 和 SUB2 单元中。试编程序求两个数相减的绝对值，将结果存于 RESULT 单元，低位在高地址，高位在低地址，同时在屏幕上显示运算结果。

程序设计时，可将两个数按从高位到低位的顺序比较，判断出大小，设置被减数和减数指针，然后做二进制减法和十进制调整。程序如下：

```
DATA   SEGMENT                          ; 数据段
       SUB1    DB 2,4,7
       SUB2    DB 3,6,5
       RESUL     DB 3 DUP(0)
DATA   ENDS
```

```
STACK SEGMENT PARA STACK                ; 堆栈段
      DW 20H DUP(?)
STACK ENDS
CODE  SEGMENT                           ; 代码段
      ASSUME CS: CODE,DS: DATA
BEGIN:MOV  AX,DATA
      MOV   DS,AX
      LEA   SI,SUB1 + 2                 ; 设被减数指针
      LEA   DI,SUB2 + 2                 ; 设减数指针
      MOV   BX,0
      MOV   CX,3                        ; 循环次数计数器
CMPE: MOV   AL,SUB1[BX]                 ; 从高位到低位逐位比较
      SUB   AL,SUB2[BX]
      JE   EQUE                         ; 相等,继续比较
      JNC   GTEAT                       ; 如 SUB1 < SUB2,交换指针
      XCHG SI,DI
      JMP   GREAT
EQUE: INC   BX
      LOOP CMPE                         ; 循环执行
GREAT:LEA   BX,RESUL + 2                ; 设结果低位指针
      MOV   CX,3                        ; 设置循环次数
      CLC                               ; 清除进位标志
      MOV   AL,[SI]                     ; 两数相减
      SBB   AL,[DI]
      AAS                               ; 十进制调整
      MOV   [BX],AL
      DEC   SI                          ; 修改指针
      DEC   DI
      DEC   BX
      LOOP   SUBT
      LEA   BX,RESUL                    ; BX 指向结果高位
      MOV   CX,3                        ; 设置循环次数
DISP: MOV   DL,[BX]
      OR   DL,30H                       ; 转换为 ASCII 码
      MOV   AH,2                        ; 2 号功能调用,显示结果
      INT   21H
      LOOP DISP
      MOV   AH,4CH                      ; 返回 DOS
      INT   21H
CODE  ENDS
      END     BEGIN
```

本程序完成多位十进制数减法,若要完成多字节二进制减法,则不必进行十进制调整。

2. 表/串处理程序设计示例

对表/串的处理是程序设计中经常遇到的另一类问题。表/串是一组顺序存放的元素集合,其元素类型可以是数值型、字符型或具有某种意义的信息代码。对表/串的处理通常有传送、查找、删除、插入、排序、检索等,可以充分利用串操作类指令。

1）表的搜索、插入、删除、统计

表是一组连续存放的元素集合，对表的操作通常要求不破坏表原来各元素间的位置关系。要删除一个元素，首先要找到这个元素，然后将后续元素向前移动一个位置，覆盖掉这个元素。而要向表中插入一个元素，首先要将插入点以后的所有元素向后移动一个位置，留出空位，然后方可插入元素。在统计一个表的元素时，要事先给表的结束位置加一个结束标志，才可以进行统计。

【例 5.16】 STRA 缓冲区中存放有 100 个字符数据，按要求对它们进行下列处理。

① 删除 STRA 中所有的“E”字母。

② 统计 STRA 中删除后的字符个数送入 LENG 单元。

③ 从键盘读入一个字符，插入 STRA 串第一个字母 B 后。

④ 显示 STRA 中的字符串。

这个问题相对复杂一些，但层次比较清晰，可以将每个处理部分设计为一个或多个子程序，由主程序调用。另外，在插入和删除子程序中都要进行串的移动，所以另外设计一个移动串子程序 TRAS 供删除和插入子程序调用。各子程序共享数据段，可以对 STRA 缓冲区进行访问。各子程序的调用关系如图 5.12 所示。

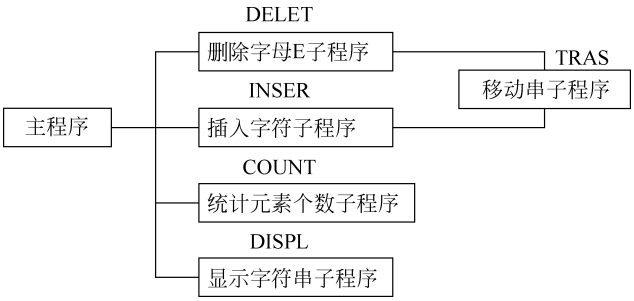


图 5.12　例 5.16 程序层次结构图

下面说明各子程序功能及入口、出口参数。

子程序 DELET，入口参数：DI 指向 STRA 首址，CX＝串长度。

子程序 COUNT，入口参数：SI 指向 STRA 首址；出口参数：LENG＝串长度。

子程序 INSER，入口参数：AL＝待插入字符，DI 指向 STRA 首址。

子程序 DISPL，入口参数：DX 指向 STRA 首址。

子程序 TRAS，入口参数：SI 指向源串首/末址，DI 指向目的串首/末址，CX＝移动次数，设置好方向标志 DF 的值。

另外，所有的子程序都利用数据段存储单元传递参数。

源程序如下：

```
DATA   SEGMENT
       BUF    DB 10 DUP('HB,5; 8VOML')
       LENG   DB ?
DATA ENDS
```

```
STACK   SEGMENT PARA STACK
        DW 50 DUP(0)
STACK ENDS
CODE SEGMENT
        ASSUME CS: CODE,DS: DATA,ES: DATA
START:  MOV AX,DATA
        MOV DS,AX
        MOV ES,AX
        LEA DI,STRA
        MOV CX,100
        CALL DELET
        LEA SI,STRA
        CALL COUNT
        MOV AH,1
        INT 21H                    ; 键盘送入单字符到 AL
        MOV CL,LENG
        MOV CH,0                   ; CX←串长度
        INC CX                     ; 长度加 1,包含 '$'符在内
        LEA DI,STRA
        CALL INSER
        LEA DX,STRA
        CALL DISPL
        MOV AH,4CH
        INT 21H
DELET PROC
        PUSH AX
        MOV AL,'E'
        CLD
LP:     REPNE SCASB                ; 扫描字符串
        JNZ DONE                   ; 若没有字母 E,则转移
        MOV SI,DI                  ; 设置移动的源串首址
        DEC DI                     ; 设置移动的目的串首址
        CALL TRAS                  ; 删除
        JMP LP                     ; 继续扫描
DONE:   MOV BYTE PTR [DI],'$'      ; 在删除后串尾送入结束标志
        POP AX
        RET
DELET   ENDP
COUNT   PROC
        MOV DL,0                   ; 累加单元清 0
COT:    CMP BYTE PTR [SI],'$'      ; 是串尾?
        JE EXIT                    ; 若是则退出
        INC SI
        INC DL                     ; 累计个数
        JMP COT
EXIT:   MOV LENG,DL                ; 送回个数到 LENG 单元
        RET
COUNT   ENDP
INSER   PROC
        PUSH AX
```

```
        MOV AL,'B'
        CLD
        REPNE SCASB             ; 扫描字符串,确定是否有字母 B
        JNZ STOP                ; 若没有就结束
        PUSH DI                 ; 保护 B 字母下一个字符位置
        ADD DI,CX               ; DI 指向移动串的末址
        MOV SI,DI
        DEC SI                  ; SI 指向移动串的末址
        STD
        CALL TRAS               ; 移动
        POP DI                  ; 恢复 B 字母下一个字符位置
        POP AX                  ; 恢复 AL 中的内容
        MOV [DI],AL             ; 插入
STOP:   RET
INSER   ENDP
DISPL   PROC
        PUSH DX                 ; 保护要显示的串首址
        MOV DL,0DH
        MOV AH,2
        INT 21H                 ; 显示回车
        MOV DL,0AH
        MOV AH,2
        INT 21H                 ; 显示换行
        POP DX                  ; 恢复要显示的串首址
        MOV AH,9
        INT 21H
DISPL   ENDP
TRAS    PROC
        PUSH SI
        PUSH DI
        PUSH CX
        REP MOVSB
        POP CX
        POP DI
        POP CX
        RET
TRAS    ENDP
CODE    ENDS
        END START
```

在删除子程序中,字母 E 可能存在多个,所以要多次扫描、多次删除。一次扫描后,若找到了字母 E,那么执行重复串扫描指令后,CX 的内容正好就是要向前移动的字符个数。DI 指向 E 下边的一个字符,可以作为移动时源串的首址,DI 减 1 的内容就可以作为移动时目的串的首址。然后调用移动子程序 TRAS 移动字符串,实现删除。TRAS 子程序将 CX、DI、SI 的值加以保护,所以移动后不改变 CX、DI、SI 的值,从 TRAS 子程序返回后可直接进行下一次扫描。删除了所有的字母 E 后,串长度发生了变化,为了能够统计出串的长度,在 DELET 子程序的最后,给串加了一个结束标志“$”。

插入子程序与删除子程序有些类似,都要先扫描,然后进行移动。有所不同的是串移动的方向,删除子程序删除时向前移动字符串,而插入子程序插入前要向后移动字符串,所以

要将 DF 设置为 1,SI、DI 修改为移动串的末址。

2) 排序与检索

对表的处理中,非常重要的一种就是排序。排序的算法有很多种,各有优缺点,在这里给大家介绍一种广泛使用的排序算法——冒泡排序。

【例 5.17】 假设有一个首地址为 ARRAY 的 N 字节数组,编程序将它们按从大到小的顺序排列。

算法思想:从第一个数开始,依次对相邻的两个数比较,若顺序符合要求,则不处理;若顺序不符合要求,则交换两个数的位置。重复这个过程,共比较($N-1$)次,比较完所有的元素,称为一轮比较。一轮比较后,已经将最小的数放在了最后,第二轮比较只需要比较前边的($N-1$)个元素,共比较($N-2$)次,比较完后,又将数组中的次小数排在了倒数第二个位置,然后,再进行下一轮的比较,以此类推,总共进行($N-1$)轮的比较,就可以完成排序。

本程序可以用双重循环实现,外循环控制比较的轮数,循环次数为($N-1$);内循环控制相邻元素的比较、交换,第 i 轮比较时的内循环次数为($N-i$)。

但在多数情况下,数组往往不需做完($N-1$)轮比较,就可能已经完成排序。为了提高程序效率,可以设置一个交换标志位。每次进入外循环时,将标志位设为 1,若内循环中有交换发生,就将标志位设为 0。内循环结束时检测标志位的值,若为 0,就再一次进入外循环;若为 1,则表明前一轮比较没有交换发生,已经完成排序,从而立即结束外循环。

程序流程如图 5.13 所示。

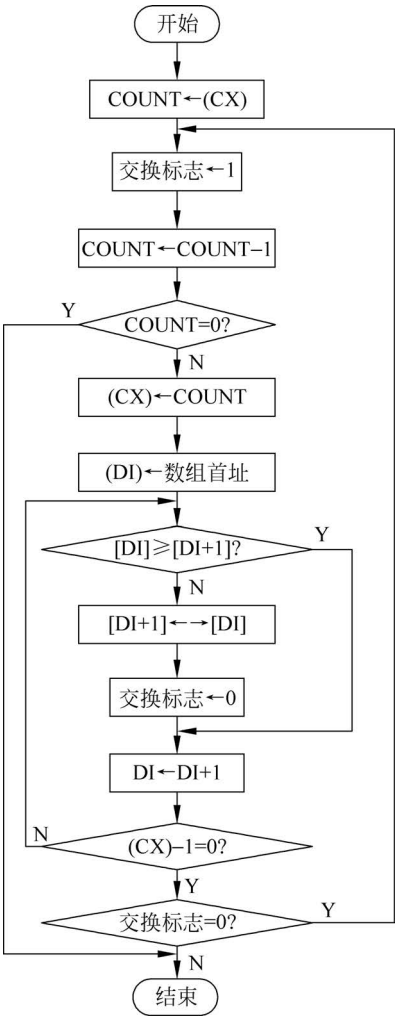


图 5.13 冒泡排序法程序流程

源程序如下:

```
DATA    SEGMENT
        ARRAY DB   'ERDFHABKLMNDOEF'
        N EQU  $ - ARRAY                    ; 取表长度
        ENDA    DB   '$'                    ; 设表结束标志
        COUNT   DB ?                        ; 记录比较轮数
DATA    ENDS
STACK   SEGMENT PARA STACK
        DW 20H DUP(0)
STACK   ENDS
CODE    SEGMENT
        ASSUME CS: CODE,DS: DATA
```

```
START:  MOV AX,DATA
        MOV DS,AX
        MOV AL,N
        MOV COUNT,AL                ; COUNT←表长度 N
REPEAT: MOV BX,1                    ; 设交换标志 BX = 1
        DEC COUNT                   ; 设比较轮数 COUNT - 1
        JZ EXIT                     ; 若为 0 则退出
        MOV CH,0
        MOV CL,COUNT                ; 设内循环次数
        LEA DI,ARRAY                ; DI←表首址
AGAIN:  MOV AL,[DI]
        CMP AL,[DI + 1]             ; [DI]与[DI + 1]比较
        JAE NEXT                    ; 若大于或等于则转移
        XCHG [DI + 1],AL
        MOV [DI],AL                 ; 交换[DI],[DI + 1]
        MOV BX,0                    ; 设交换标志 BX = 0
NEXT:   INC DI                      ; 修改 DI
        LOOP AGAIN                  ; 若 CX - 1≠0 则继续比较
        CMP BX,0                    ; 交换标志(BX) = 0?
        JE REPEAT                   ; 若为 0 则进行下一轮比较
EXIT:   LEA DX,ARRAY
        MOV AH,09H
        INT 21H                     ; 显示有序的表
        MOV AH,4CH
        INT 21H
CODE    ENDS
        END START
```

在这个程序中，外循环有两种退出方法：一种是测试交换标志 BX 的值，若为 1，则结束外循环；另一种是对 COUNT 单元中存放的比较轮数进行测试，每比较一轮，COUNT 减 1，若减为 0，则结束外循环。

3. 代码转换程序设计示例

代码转换是在程序设计中经常遇到的一类问题。例如，从键盘输入的数据都是 ASCII 码的形式，计算机进行处理时，必须将它们转换为二进制数值；要显示输出的数据必须先转换为 ASCII 码；用户输入的十进制数据，要转换为二进制处理；处理后的数据，要转换为十进制输出，等等。

代码转换通常有两种方法：对于没有规律的转换代码，可通过查表来实现；对于有规律的转换代码，可根据它们的转换规则进行处理，实现转换。

1）二进制码和 ASCII 码之间的相互转换

实际上，二进制数据通常都以十六进制的形式表示，所以这里主要讨论十六进制码和 ASCII 码之间的相互转换。

十六进制数据的 16 个符号 0～9、A～F(a～f)，对应的 ASCII 码为 30H～39H、41H～46H(61H～66H)。对于 0～9 的数，要转换为 ASCII 码，只要加(逻辑“或”)30H 就可以了；而对于 A～F(a～f)的数，则要加(逻辑“或”)37H(57H)。

【例 5.18】 将以 BUF 为首址的存储单元中的字数据显示输出，每两个字之间用空格

分隔。

要显示输出,必须首先将数据转换为 ASCII 码。每个字的转换通过循环左移指令,依次将字的高位十六进制数移入低位来进行转换。以此类推,循环执行 4 次,即可实现一个字的转换。多个字的转换可以用双重循环实现,外循环用于控制待转换的字数,内循环控制每个字需要转换的次数。字符显示可用 2 号功能调用。

程序如下:

```
DATA   SEGMENT
       BUF DW 347AH,7CBAH,0D698H
       COUNT EQU $ - BUF
DATA   ENDS
CODE   SEGMENT
       ASSUME CS: CODE,DS: DATA
START: MOV AX,DATA
       MOV DS,AX
       LEA SI,BUF                ; SI←BUF 首址
       MOV DH,COUNT/2            ; DH←字的个数
LP1:   MOV CX,4                  ; CX←每个字需要转换的次数
       MOV BX,[SI]               ; BX←取一个字
NUM:   PUSH CX
       MOV CL,4
       ROL BX,CL                 ; BX 循环左移 4 次,高 4 位移入低 4 位
       MOV DL,BL
       AND DL,0FH                ; DL←分离字的低 4 位
       ADD DL,30H                ; DL + 30H
       CMP DL,3AH                ; 判断是否为 A～F
       JB NEXT                   ; 若小于,则为 0～9,转移
       ADD DL,07H                ; 若为 A～F,再加 7
NEXT:  MOV AH,2
       INT 21H                   ; 显示
       POP CX
       LOOP NUM                  ; 若 4 位没有转换完,则继续
       MOV DL,''
       MOV AH,2
       INT 21H                   ; 显示空格
       ADD SI,2                  ; 修改 SI,指向下一个字
       DEC DH
   JNZ LP1                       ; 若 DH - 1≠0,则继续
   MOV AH,4CH
   INT 21H
CODE ENDS
   END START
```

使用 PUSH CX 和 POP CX 指令的原因是:内循环用 CX 作计数器,而循环移位指令要求将移位位数置入 CL 寄存器,这样会改变循环计数器的值,使之不能正常退出。所以应在 CL 移位位数前,将 CX 压栈保护,而在修改内循环计数前,应恢复 CX。

2）十进制数转换为二进制数

【例 5.19】 将键盘输入的十进制数转换为二进制数，十进制数串以回车符结束，要求转换后的二进制数存入 DX 寄存器(假设不超过 65535)。

首先要将键盘输入的 ASCII 码形式的十进制数符转换为 BCD 码。若输入的十进制数是 $D_4D_3D_2D_1D_0$，则可用下述公式将它转换为二进制值。

$$(D_4D_3D_2D_1D_0)=((((0\times10+D_4)\times10+D_3)\times10+D_2)\times10+D_1)\times10+D_0$$

由展开式可见，整个转换过程是：从待转换数码的高位开始，重复执行中间结果乘 10 再加待转换数码。

程序如下：

```
DATA   SEGMENT
       STRING DB 'INPUT A DECIMAL NUMBER BETWEEN 0 - 65535',0DH,0AH,'$ '
DATA   ENDS
CODE   SEGMENT
       ASSUME CS: CODE,DS: DATA
BEGIN: MOV AX,DATA
       MOV DS,AX
       LEA DX,STRING
       MOV AH,9
       INT 21H                      ; 显示提示信息
       XOR DX,DX                    ; DX 清 0
NEXT:   MOV AH,1
       INT 21H                      ; 等待从键盘输入字符
       CMP AL,0DH
       JE DONE                      ; 若为回车符,则结束
       SUB AL,30H
       MOV AH,0                     ; AX←转为 BCD 码
       SAL DX,1
       MOV BX,DX
       SAL DX,1
       SAL DX,1
       ADD DX,BX                    ; DX 乘 10
       ADD DX,AX                    ; DX 加输入数字
       JMP NEXT
DONE:     MOV AH,4CH
       INT 21H
CODE      ENDS
       END BEGIN
```

程序中 DX 乘 10 部分由移位和加法指令实现。

3）二进制数转换为十进制数

【例 5.20】 将一个 16 位二进制数转换为十进制数，形成的十进制数以非组合 BCD 码的形式表示。

与十进制转换为二进制相似，若 16 位二进制数是 $B_{15}B_{14}\cdots B_1B_0$，那么可以用下述公式实现转换。

$$(B_{15}B_{14}\cdots B_1B_0)=(\cdots((0\times2+B_{15})\times2+B_{14})\times2+\cdots+B_1)\times2+B_0$$

从待转换数码的高位开始,重复执行中间结果乘2再加待转换数码,并在每一步的乘或加运算操作后,用十进制调整指令调整,那么,最后的运算结果一定是十进制数,并且结果不超过5位。

下面分别用子程序实现中间结果乘2运算和相加运算,运算的中间和最后结果都存放在DECIM单元,DECIM初始化为全0。

程序如下:

```
DATA   SEGMENT
       BIN DB 5634H
       DECIM DB 5 DUP(0)
DATA   ENDS
CODE   SEGMENT
       ASSUME CS: CODE,DS: DATA
START: MOV AX,DATA
       MOV DS,AX
       MOV DX,BIN
       MOV CX,16                    ; 设乘、加次数
AGAIN: CALL MU
       SAL DX,1                     ; 将DX的最高位移入CF
       CALL AD
       LOOP AGAIN
       MOV AH,4CH
       INT 21H
; 采用自身相加的方法实现中间结果乘2的运算操作,与调用程序共享DECIM存储单元
MU     PROC
       PUSH DI
       PUAH AX                      ; 保护现场
       LEA DI,DECIM + 4             ; DI指向已转换十进制数的最低位
       MOV AH,5                     ; 自身相加5次
       CLC
LP1:   MOV AL,[DI]
       ADC AL,AL                    ; 带进位加
       AAA                          ; 十进制调整
       MOV [DI],AL                  ; 存结果
       DEC DI                       ; 指向高一位
       DEC AH
       JNZ LP1                      ; 若5次不够,则重复
       POP AX
       POP DI                       ; 恢复现场
       RET
MU     ENDP
; 实现将MU子程序执行的结果与待转换的数码相加,与调用程序共享DECIM单元
; 入口参数: 待加的数码事先存入标志寄存器的CF中
AD     PROC
       PUSH AX
       PUSH DI                      ; 保护现场
```

```
        LEA DI,DECIM + 4                    ; DI 指向已转换十进制数的最低位
        MOV AH,5                            ; 加 5 次进位
LP2:    MOV AL,[DI]
        ADC AL,0                            ; 加进位
        AAA                                 ; 十进制调整
        MOV[DI],AL                          ; 存结果
        DEC DI                              ; 指向高一位
        DEC AH                              ; 若不够 5 次,则重复
        JNZ LP2
        POP DI
        POP AX                              ; 恢复现场
        RET
        AD ENDP
        CODE ENDS
        END START
```

对于二进制转换为十进制的问题,还可以用除法的方法实现,即将 16 位二进制数除以 10,所得余数为十进制数的个位；再将所得商除以 10,得到的余数为十进制数的十位；以此类推,最后一次除法中所得的余数便是万位,总共需做 5 次除法运算。

下面给出除法实现的二进制数转换为十进制数的代码,其数据段的定义与例 5.19 相同。

```
CODE    SEGMENT
        ASSUME CS: CODE,DS: DATA
START:  MOV AX,DATA
        MOV DS,AX
        MOV BX,10
        MOV CX,5
        LEA DI,DECIM + 4
        MOV AX,BIN
AGAIN:  XOR DX,DX
        DIV BX
        DEC DI
        LOOP AGAIN
        MOV AH,4CH
        INT 21H
CODE    ENDS
```

二进制数转换为十进制数还可以通过循环减法来实现：二进制数减去 10000,若够减,则万位累加 1,重复减,直至不够减为止,然后恢复余数,将余下的数减去 1000,以此类推,最后减 10 运算的余数,即为个位数。这种实现方法虽然思想简单,但程序实现效率不高。读者可自己编程实现。

5.11　调试程序 DEBUG 的使用

MS DOS 附带的 DEBUG.com 的调试程序是一个功能较强的调试工具。它不仅可以直接装入、启动运行汇编语言程序,还可以跟踪程序的运行过程,直接修改目标程序,检查和

修改内存单元和寄存器,实现在运行中对程序的调试。

1. 调试程序 DEBUG 的调用

在 DOS 提示符下(假设在 C 盘有 DEBUG.com),输入如下命令:

```
C>DEBUG↙
```

屏幕出现 DEBUG 调试程序的提示符"-",此时可输入 DEBUG 命令。调试程序启动后,CPU 各寄存器和标志位设置为以下状态。

(1) 段寄存器(CS、DS、ES、SS)置于自由存储空间的底部,也就是 DEBUG 程序装入以后的第一个段。

(2) 指令指针(IP)置为 0100H。

(3) 堆栈指针(SP)置于段的结尾处或装入程序的临时底部,取决于哪一个更低。

(4) 通用寄存器(AX、BX、CX、DX、BP、SI、DI)置为 0。

(5) 所有标志位都处于复位状态。

若在调用 DEBUG 时包含一个要调试的程序文件名,则 DEBUG 把段寄存器、堆栈指针置为程序中规定的值。对 EXE 文件,IP 置为 0000H;对 COM 文件,则 IP 置为 0100H。BX 和 CX 中包含文件长度,CX 为长度的低字节,BX 为长度的高字节。

2. DEBUG 的主要命令

DEBUG 的命令都是一个英文字母,它反映该命令的功能,命令字符后面有一个或多个参数,参数之间用空格或逗号分隔。所有数据都是十六进制,不必写 H。可以用 Ctrl+Break 组合键来停止一个命令的执行,返回 DEBUG 提示符。Ctrl+NumLock 组合键可以暂停屏幕上卷,而按其他键继续。

1) 内存显示命令 D(Dump)

格式:

```
-D[地址]
-D[地址范围]
```

功能:显示指定内存单元内容。

(1) 若命令中有指定地址,则从指定地址开始显示 8 行,每行 16 字节。地址中若无段地址,则默认段地址为 DS。

(2) 若命令中没有指定地址,则从上一个 D 命令所显示的最后一个单元的下一个单元开始显示。若以前没有用过 D 命令,则从 0100H 开始显示。

(3) 地址范围包含起始地址和结束地址,中间用空格分开。若起始地址中未包含段地址,则默认的段地址为 DS,结束地址只允许为偏移量。

例如:

```
-D  100          (显示从 DS:100H 到 DS:017FH 的内容,共 128 字节内容)
-D  100  200     (显示从 DS:100H 到 DS:200H 的内容)
```

DEBUG 把输入的数字均看成十六进制数，所以若输入十进制数，则其后应加以说明，如 100D。

2）内存修改命令 E(Enter)

格式：

```
E <地址><字节表>
E <地址>
```

功能：用<字节表>内容去修改指定地址内存单元的内容，<字节表>是以空格或逗号分隔的十六进制字节或字符串。

命令输入后，屏幕显示指定内存单元的地址和原有内容，输入新的两位十六进制数，以替代原来的内容，按空格键完成修改并显示下一个高地址单元的内容。若不修改所显示的单元，则按空格键跳过，按回车键结束修改。输入一个减号(“－”)，则修改从高地址向低地址进行。若不修改，则直接按空格键跳过，按回车键结束此命令。

例如：

```
-E  100  12 34 "ABC"
```

用 12、34、'A'、'B'、'C'依次修改从 DS：100H 起的 5 个单元。

3）比较命令 C(Compare)

格式：

```
-C<源地址范围><目标地址>
```

功能：比较指定区域中的内容是否相同。

<源地址范围>中包含起始地址和结束地址。若起始地址中未包含段地址，则默认段地址为 DS，结束地址只包含地址偏移量，目标地址只含起始地址。从源起始地址开始逐个与目标地址开始的单元进行比较，直到源结束地址为止。对不同的单元，则显示出它们的地址和内容。

例如：

```
-C  100  108  200
```

对从 DS：100H 和 DS：200H 起两个区域中 9 个单元的内容进行比较。

4）内存填充命令 F(Fill)

格式：

```
F<范围><字节表>
```

功能：将<字节表>内容逐个写入指定内存单元中。

若填充内容的长度大于范围，则超出部分被截断。若填充内容不足，则重复使用所列的

填充内容。

例如:

```
-F 1400:100 200  'XYZ',3B,46
```

表示用'X'、'Y'、'Z'、3B、46这5字节的内容反复填充从1400:100H到1400:200H的内存单元。

5) 搜索命令S(Search)

格式:

```
-S<范围><字节表>
```

功能:在指定的内存范围搜索<字节表>指定的字符串,找到后显示元素所在的地址。

例如:

```
-S  CS:100 120 48
```

表示在CS:100H到CS:120H的内存范围内查找内容为48H的单元。

6) 十六进制运算命令H(Hex)

格式:

```
-H<数值1><数值2>
```

功能:计算并显示两个十六进制数的和与差。

例如:

```
-H 124C 49AB
```

7) 寄存器命令R

格式一:

```
-R
```

功能:显示CPU内部所有寄存器的内容和标志位的状态,并反汇编CS:IP所指的指令(下一条将要执行的指令)。

输入命令-R后,系统显示如下:

```
AX=0000 BX=0000 CX=0000 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=1D64 ES=1D64 SS=1D64 CS=1D64 IP=0100 NV UP DI PL NZ NA PO NC
1D64:0100  B83412  MOV  AX,1234
```

其中,前两行显示CPU所有寄存器的内容和标志位的状态,第三行显示当前CS:IP所指的

将要执行的下一条指令的机器代码和指令助记符。

格式二：

```
-R <寄存器名>
```

功能：显示指定寄存器的值，并等待用户输入新的值，按回车键结束 R 命令。

例如，输入命令

```
-R  AX
```

则系统显示如下：

```
AX 0000
```

若不需要修改其内容，则直接按回车键；若需要修改其内容，则可输入 1～4 位十六进制数值，再按回车键。

<寄存器名>只能是 8086 的寄存器，如 AX、BX、CX、DX、SP、BP、SI、DI、DS、ES、SS、CS、IP、F(标志寄存器)。

格式三：

```
-RF
```

显示和修改标志位状态。

8 个标志位的显示顺序和置位、复位的代号如表 5.3 所示。

表 5.3　标志位的置位和复位对照表

标　志　位	置位	复位	标　志　位	置位	复位
OF 溢出标志(有/无)	OV	NV	ZF 零标志(是/否)	ZR	NZ
DF 方向标志(减量/增量)	DN	UP	AF 辅助进位标志(是/否)	AC	NA
IF 中断标志(允许/屏蔽)	EI	DI	PF 奇偶标志(偶/奇)	PO	PE
SF 符号标志(负/正)	NG	PL	CF 进位标志(有/无)	NC	CY

例如，输入命令-RF，则系统显示如下：

```
OV DN EI NG ZR AC PE CY -
```

最后是 DEBUG 的提示符。若不需要修改，则可直接按回车键；若需要修改，则可以输入一个或多个标志位相应的复位或置位代号。输入标志的次序可以改变，各标志之间也可以没有空格。按回车键确认并完成修改。

8) 反汇编命令 U(Unassemble)

把程序的目标机器码转换为汇编前指令助记符的过程，称为**反汇编**。反汇编命令能显

示装入内存某一区域中程序的机器码及相应的指令助记符。

格式:

```
-U
-U[地址]
-U[地址范围]
```

功能:把二进制的机器代码反汇编为符号指令,并按行显示指令的内存地址、机器代码、汇编指令。U命令执行后,IP指向已反汇编过的下一条指令。

(1) 若在命令中没有指定地址,则从IP所指的那条指令起开始进行32字节的反汇编。

(2) 若命令中指定地址,则从指定地址开始对其后连续32字节的内容进行反汇编。

(3) 若指令中指定地址范围,则对指定范围的内存单元进行反汇编。范围可以由起始地址、结束地址或长度来确定,长度前用L标记。

(4) 若地址中未指定段地址,则默认的是当前CS所指的代码段。

例如:

```
-U  100
-U  100 010F
```

这两条命令的结果是相同的,表示从CS:100H起反汇编,至CS:010F结束。

9) 汇编命令A(Assemble)

格式:

```
-A[地址]
```

功能:在指定地址处开始编写汇编程序。该命令允许在DEBUG环境下输入汇编语言程序,并把它们汇编成机器代码,相继存放在从指定地址开始的存储区中。

地址参数指定在CS段中开始写入指令代码的起始地址,若在命令中没有指定地址,则默认的地址是由IP所确定的。

例如:

```
-A  100
```

表示从CS:100单元开始存放汇编指令代码。

10) 运行命令G(Go)

格式:

```
-G[=起始地址][断点地址1[断点地址2[断点地址3…]]]
```

功能:按照命令确定的起始地址和断点运行程序。起始地址规定执行的第一条指令的地址,当指令执行到断点地址时,就停止执行并显示CPU中所有寄存器内容和标志位的状

态，以及下一条将要执行的指令，并返回 DEBUG，以便进一步检查或进行必要的修改。DEBUG 最多允许设置 10 个断点。

默认的段址为 CS，命令中的地址参数值作为地址偏移量。地址参数必须是有效指令的第一字节，否则会出现不可预料的结果。若无地址参数，则以当前的 CS：IP 为起始地址。

11）跟踪命令 T（Trace）

格式：

```
-T
-T[=起始地址][指令条数]
```

功能：逐条运行指令，每执行一条指令就停下来并显示 CPU 所有寄存器的内容和标志位的状态，以及下一条指令的地址和内容。利用此命令可跟踪程序执行的过程，并在跟踪的过程中对程序进行调试或修改。

（1）若命令中没有指定地址，默认为 CS：IP。

（2）若只给出偏移地址，则以 CS 的当前值为段地址。

（3）若命令中给出起始地址和指令条数，则从指定或默认的地址开始，连续执行多条指令，并显示每条指令执行后的寄存器和标志位的状态，执行的指令数由“指令条数”决定。

（4）若未给出<指令条数>，则默认为 1，每次执行一条指令。

（5）遇到 CALL 或 INT n，则会跟踪进入相应过程和中断服务程序的内部，对于带重复前缀（REP）的指令，每执行一次算一步。

12）继续命令 P（Proceed）

格式：

```
P[=<起始地址>] [<指令条数>]
```

功能：类似于 T 命令，二者的区别是 T 命令可进入子程序或中断并跟踪其运行过程，P 命令则不是跟踪子程序，而是接着执行下一条指令。例如，P 命令把 CALL、INT n 或 REP 当作一步，不会进入相应过程或中断程序内部。

13）文件命名命令 N（Name）

格式：

```
-N<文件路径名><文件名>
```

功能：给当前的程序文件命名，以便为 L 和 W 命令对指定文件进行读、写做准备。<文件路径名>包括盘符和文件路径。

14）装入命令 L（Load）

格式：

```
-L
-L<内存目标地址><磁盘源地址>
```

功能：将指定的磁盘文件装入内存指定区域。磁盘文件由〈磁盘源地址〉指定，内存区域由〈内存目标地址〉指定。

(1) 磁盘源地址依次包括驱动器号、扇区号和扇区数，其中驱动器号用数字表示，A 盘为 0，B 盘为 1，以此类推。若命令中未指定磁盘源地址，则默认是由 N 命令指定的磁盘文件。此时，在 L 命令之前应先执行一条 N 命令。读入的文件长度反映在 BX 和 CX 中。

(2) 内存目标地址的默认段地址为 CS。若命令中未指定内存的目标地址，则文件装入 CS：0100 开始的内存区域中；若命令中指定了目标地址，则装入指定地址开始的内存区域中。但对扩展名为.com 或.exe 的文件，则始终装入 CS：0100 的内存区中，即使指定了地址，此地址也被忽略。

例如：

```
-L 146D: 100 1 20 1E
```

表示从 B 驱动器相对扇区号为 20H 的扇区开始将 1EH 个扇区的内容读入内存 146D：100H 开始的区域中。

15) 存盘命令 W (Write)

格式：

```
-W<内存源地址><磁盘目标地址>
```

功能：将内存中的指定区域内容写到磁盘指定扇区或磁盘文件中。内存区域由〈内存源地址〉开始，要写入磁盘的字节数由 BX：CX 决定。因此，写入前，应正确设置 BX、CX 寄存器的值。

(1) 若未给出〈内存源地址〉，则默认从 CS：0100H 开始。

(2) 〈磁盘目标地址〉是磁盘指定的扇区，依次包括驱动器号、扇区号和扇区数，其中驱动器号用数字表示，A 盘为 0，B 盘为 1，以此类推。若命令未指定磁盘目标地址，则默认是由 N 命令指定的磁盘文件，文件的长度由 BX 和 CX 确定。

例如：

```
-W 100 0 10 32
```

将内存 CS：100 开始存放的数据写入 A 驱动器起始扇区号为 10H 连续的 32H 个扇区中，字节数由 BX：CX 确定。

16) 退出命令 Q (Quit)

格式：

```
-Q
```

功能：退出 DEBUG，返回到 DOS 状态。本命令无存盘功能，如需存盘，应先使用 W 命令。

5.12　例 题 解 析

1. 某数据段内有如下数据定义。

```
X1   db  20,20H, 'AB', 3-2, ?, 11000011B
X2   dw  0AAH, -1, 'AB'
Z    dd  5 dup(3, 2 dup(?), 0)
W    dw  Z-X2
```

假设 X1 的偏移地址为 100H，(1)写出变量 X1、X2 各数据在内存中的具体位置和相关内存单元的值；(2)写出变量 Z、W 的偏移地址；(3)写出变量 W 的值。

【解析】

(1) X1、X2 各数据在内存中存放的位置和内存单元的值如下。

0100H	14H
	20H
	41H
	42H
	01H
	00H
	C3H
0107H	AAH
	00H
	FFH
	FFH
	42H
	41H
010D	

(2) 根据 X1 和 X2 在内存中的分布，可以发现 Z 的偏移地址是 010DH；由于 Z 中的每个数据都是双字，占 4 字节，一共占用 5×(4+2×4+4)=80=50H 字节，则 W 的偏移地址是 015DH。

(3) Z−X2 的值是 06H。

2. 根据下列要求编写一个汇编语言程序。

(1) 代码段的段名为 COD_SG。

(2) 数据段的段名为 DAT_SG。

(3) 堆栈段的段名为 STK_SG。

(4) 变量 HIGH_DAT 所包含的数据为 95。

(5) 将变量 HIGH_DAT 装入寄存器 AH、BH 和 DL。

(6) 程序运行的入口地址为 START。

【解析】

```
DAT_SG    SEGMENT
          HIGH_DAT   DB  95
DAT_SG    ENDS

STK_SG    SEGMENT
DW        64 DUP(?)
STK_SG    ENDS

COD_SG    SEGMENT
ASSUME    CS: COD_SG ,  DS: DAT_SG ,  SS: STK_SG
START:    MOV   AX , DAT_SG
          MOV  DS , AX
          MOV  AH , HIGH_DAT
          MOV  BH , AH
          MOV  DL,AH
          MOV  AH,4CH
          INT    21H
COD_SG    ENDS
          END    START
```

3. 指出下列程序中的错误。

```
STAKSG  SEGMENT
        DB   100  DUP(?)
STA_SG  ENDS

DTSEG   SEGMENT
        DATA1    DB  ?
DTSEG   END
CDSEG   SEGMENT
MAIN    PROC    FAR
START:  MOV     DS , DATSEG
        MOV     AL , 34H
        ADD     AL , 4FH
        MOV     DATA , AL
START   ENDP
CDSEG   ENDS
     END
```

【解析】

首先段名和子程序名的标号必须前后一致,但是堆栈段和主程序标号前后不对应;其次需使用 ASSUME 将程序中的段和段寄存器对应起来;然后主程序要有结束本程序并返回操作系统的操作;最后 END [label]中的标号指示程序开始执行的起始地址,只有在多个程序模块相连接时主程序需要使用标号,其他子程序模块只使用 END 而不必使用标号。

改正后程序为:

```
STAKSG   SEGMENT
   DB    100 DUP(?)
STAKSG    ENDS
DTSEG     SEGMENT
DATA1     DB ?
DTSEG     ENDS
CDSEG     SEGMENT
MAIN      PROC   FAR
   ASSUME CS: CDSEG ,   DS: DTSEG , SS: STAKSG
START: MOV  AX, DTSEG
       MOV  DS,AX
       MOV  AL,34H
       ADD  AL,4FH
       MOV  DATA1,AL
       MOV  AH,4CH
       INT   21H
MAIN   ENDP
CDSEG  ENDS
       END    START
```

4. 对于下面两个数据段，偏移地址 10H 和 11H 的两字节中的数据是一样的吗？为什么？

```
; 数据段 1
DTSEG   SEGMENT
   ORG     10H
   DATA1   DB  72H
   DB  04H
DTSEG   ENDS
```

```
; 数据段 2
DTSEG   SEGMENT
    ORG     10H
    DATA1   DW    7204H
DTSEG   ENDS
```

【解析】

不一样。数据段 1 从 10H 开始两字节依序存放了 72H、04H；数据段 2 则存放着 04H、72H。究其原因是数据段 1 中存放的是两字节类型的数，数据段 2 存放的是一个字类型的数，存储字时低 8 位存在低字节，高 8 位存在高字节。

5. 假设 X 和 $X+2$ 单元的内容为双精度数 p，Y 和 $Y+2$ 单元的内容为双精度数 q（X 和 Y 为低位字），试说明下列程序段做什么工作？

```
    MOV   DX, X + 2
    MOV   AX, X
    ADD   AX, X
    ADC   DX, X + 2
    CMP   DX, Y + 2
    JL    L2
    JG    L1
    CMP   AX, Y
    JBE   L2
L1: MOV   AX, 1
    JMP   SHORT EXIT
L2: MOV   AX, 2
   EXIT:  INT 20H
```

【解析】

此程序段判断 $p\times2>q$,则使(AX)=1 后退出;若判断 $p\times2\leqslant q$,则使(AX)=2 后退出。

6. 有三个 3 位的 ASCII 数串 ASC1、ASC2 和 ASC3,分别定义为 ASC1 DB '578'、ASC2 DB '694'和 ASC3 DB '0000',请编写程序段实现 ASC3←ASC1+ASC2。

【解析】

```
    CLC
    MOV   CX,3
    MOV   BX,2
BACK:
    MOV   AL, ASC1[BX]
    ADC   AL, ASC2[BX]
    AAA
    OR    ASC3[BX+1], AL
    DEC   BX
    LOOP  BACK
    RCL   CX, 1
    OR    ASC3[BX],CL
```

7. 试编写程序,要求从键盘输入 3 个十六进制数,并根据对 3 个数的比较显示如下信息。

(1) 如果 3 个数都不相等则显示 0。

(2) 如果 3 个数中有两个数相等则显示 2。

(3) 如果 3 个数都相等则显示 3。

【解析】

```
DATA    SEGMENT
        ARRAY     DW   3 DUP(?)
DATA    ENDS
CODE    SEGMENT
MAIN    PROC  FAR
        ASSUME CS: CODE,DS: DATA
START:
        PUSH   DS
        SUB    AX,AX
        PUSH   AX
        MOV    AX,DATA
        MOV    DS,AX
        MOV    CX,3
        LEA    SI,ARRAY
BEGIN:
        PUSH   CX
        MOV    CL,4
        MOV    DI,4
```

```
        MOV     DL,''
        MOV     AH,02
        INT     21H
        MOV     DX,0
INPUT:
        MOV     AH,01
        INT     21H
        AND     AL,0FH
        SHL     DX,CL
        OR      DL,AL
        DEC     DI
        JNE     INPUT
        MOV     [SI],DX
        ADD     SI,2
        POP     CX
        LOOP    BEGIN
COMP:
        LEA     SI,ARRAY
        MOV     DL,0
        MOV     AX,[SI]
        MOV     BX,[SI + 2]
        CMP     AX,BX
        JNE     NEXT1
        ADD     DL,2
NEXT1:
        CMP     [SI + 4],AX
        JNE     NEXT2
        ADD     DX,2
        NEXT2:
        CMP     [SI + 4],BX
        JNE     NUM
        ADD     DL,2
NUM:
        CMP     DX,3
        JL      DISP
        MOV     DL,3
DISP:
        MOV     AH,2
        ADD     DL,30H
        INT     21H
        RET
MAIN  ENDP
CODE  ENDS
          END   START
```

8. 试编写程序，它轮流测试两个设备的状态寄存器，只要一个状态寄存器的第 0 位为 1，则与其相应的设备就输入一个字符；如果其中任一状态寄存器的第 3 位为 1，则整个输入过程结束。两个状态寄存器的端口地址分别为 0024H 和 0036H，与其相应的数据输入寄存

器的端口则为0026H和0038H,输入字符分别存入首地址为BUFF1和BUFF2的存储区中。

【解析】

```
        MOV    DI, 0
        MOV    SI, 0
BEGIN:  IN     AL, 24H
        TEST   AL, 08H              ; 查询第一个设备的输入是否结束
        JNZ    EXIT
        TEST   AL, 01H              ; 查询第一个设备的输入是否准备好
        JZ     BEGIN1
        IN     AL, 26H              ; 输入数据并存入缓冲区 BUFF1
        MOV    BUFF1[DI], AL
        INC    DI
BEGIN1: IN     AL, 36H
        TEST   AL, 08H              ; 查询第二个设备的输入是否结束
        JNZ    EXIT
        TEST   AL, 01H              ; 查询第二个设备的输入是否准备好
        JZ     BEGIN
        IN     AL, 38H              ; 输入数据并存入缓冲区 BUFF2
        MOV    BUFF2[SI], AL
        INC    SI
        JMP    BEGIN
EXIT:
        ⋮
```

9. 给定(SP)=0100H,(SS)=0300H,(FLAGS)=0240H,存储单元的内容为(00020H)=0040H,(00022H)=0100H,在段地址为0900H及偏移地址为00A0H的单元中有一条中断指令INT 8,试问执行INT 8指令后,SP、SS、IP、FLAGS的内容各是什么?栈顶的3个字是什么?

【解析】

SP是堆栈寄存器,堆栈是向下生长的(减法),SS是源地址段寄存器,IP是当前运行代码指针地址寄存器,FLAGS是16位的运行标志寄存器,其中第0、2、4、6、7、8、9、10、11分别为CF、PF、AF、ZF、SF、TF、IF、DF、OF,这里的第9位IF就是代表Interrupt Flag发生中断的标志位。当中断发生时,系统将标志寄存器FLAGS和下一条指令的地址CS:IP的值分别压入堆栈,然后将中断服务程序的入口地址装入CS和IP寄存器。这样CPU就会转去执行中断服务程序。中断返回时,系统从栈顶分别弹出CS、IP、FLAGS的值,CPU继续从断点开始执行。

INT 8是调用8号中断,在DOS中8号是时钟中断,发生中断时,中断的位置是固定的。根据中断向量表首地址=中断型号×4(中断向量表存放中断服务程序入口地址),那么8号中断的入口地址就存放在20H~23H中(高字为CS,低字为IP),即0040H和0100H。

于是,当执行INT8时,首先把原FLAGS 0240H入栈保存,SP需要减两字节,然后把当前地址0900H:00A0H的下一地址(断点、返回位置)0900H:00A1H入栈保存,高位先进,那么SP需要减去4字节;最后SP为0100H-6H=00FAH,再将8号中断的入口地址

0040H：0100H 装载到 CS：IP 中以便进行跳转执行。于是 SS 不变，CS 和 IP 分别变为 0100H 和 0040H，同时 FLAGS 里面的中断位发生变化，从而变成 0040H。

根据以上的分析可以得出：

(SP)＝00FAH

(SS)＝0300H

(IP)＝0040H

(FLAGS)＝0040H

堆栈内容：

00A1H
0900H
0240H

5.13　本章实验项目

【实验 1】 宏汇编上机练习。

第一步：用 edit 编辑程序打开编辑窗口，并将例 5.1 输入计算机中，保存为 lianxi.asm 源文件。

第二步：用 masm 汇编程序汇编 lianxi.asm 程序，产生目标文件 lianxi.obj。若有语法错误，则返回第一步修改源程序；否则进入第三步。

第三步：用 link 链接程序连接 lianxi.obj 文件，产生可执行程序文件 lianxi.exe。

第四步：直接输入文件名 lianxi 执行文件。

【实验 2】 调试工具 debug 常用命令练习。

用 DEBUG 命令将实验 1 生成的可执行程序 lianxi.exe 装入内存。

(1) 使用反汇编命令 U 对二进制文件进行反汇编，观察和源程序的差别。

(2) 使用跟踪命令 T 观察每条指令的执行，然后显示各寄存器的内容。

(3) 在适当的时候使用显示内存命令 R，观察数据段中数据的变化。

【实验 3】 编程实现：计算 Z＝X－Y，其中 X、Y、Z 为 BCD 码，设 X、Y 为 40、12，则 Z 为 28。用 DEBUG 调试工具观察数据段结果是否为 28(即 00101000)。

提醒：可利用十六进制将 BCD 码赋值给变量。

【实验 4】 设计一数据块间数据搬移程序。

要求：(1) 把内存中一数据区(称为源数据块)传送到内存另一数据区(称为目的数据块)。

(2) 用 DEBUG 调试工具观察目的数据块数据是否和源数据块数据一致。

提醒：源数据块与目的数据块可以在同一数据段内，也可以不在同一个数据段内。

【实验 5】 编程实现：判断 10 个无符号字节数的最大值和最小值，并在显示器上显示出来。

要求：(1) 用子程序实现判断最大值和最小值。

(2) 用十六进制显示最大值和最小值。

扫一扫
自测题

习 题 5

1. 下列语句在存储器中分别为变量分配多少字节空间？画出存储空间的分配图。

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

2. 假定 VAR1 和 VAR2 为字变量，LAB 为标号，试指出下列指令的错误之处。

(1) ADD VAR1,VAR2　　(2) SUBAL,VAR1

(3) JMP LAB[SI]　　(4) JNZVAR1

3. 对于下面的符号定义，指出下列指令的错误。

```
A1  DB  ?
A2  DB  10
K1  EQU  1024
```

(1) MOV K1,AX　　(2) MOV A1,AX

(3) CMP A1,A2　　(4) K1EQU2048

4. 数据定义语句如下：

```
FIRST   DB  90H,5FH,6EH,69H
SECOND  DB  5 DUP(?)
THIRD   DB  5 DUP(?)
FORTH   DB  5 DUP(?)
```

自 FIRST 单元开始存放的是一个 4 字节的十六进制数(低位字节在前)，要求如下。

(1) 编写一段程序将这个数左移两位、右移两位后存放到自 SECOND 开始的单元(注意保留移出部分)。

(2) 编写一段程序将这个数求补以后存放到自 FORTH 开始的单元。

5. 试编写程序将内存从 40000H 到 4BFFFH 的每个单元中均写入 55H，再逐个单元读出比较，看写入的与读出的是否一致。若全对，将 AL 置 7EH；只要有错，则将 AL 置 81H。

6. 在当前数据段 4000H 开始的 128 个单元中存放一组数据，试编程序将它们顺序搬移到 A000H 开始的 128 个顺序单元中，并将两个数据块逐个单元进行比较；若有错，将 BL 置 00H；若全对，则将 BL 置 FFH，试编写程序。

7. 设变量单元 A、B、C 存放 3 个数，若 3 个数都不为零，则求 3 个数的和，存放在 D 中；若有一个为零，则将其余两个也清零，试编写程序。

8. 有一个 100 字节的数据表，表内元素已按从大到小的顺序排列好，现给定一元素，试编程序在表内查找，若表内已有此元素，则结束；否则，按顺序将此元素插入表中适当的位置，并修改表长。

9. 内存中以 FIRST 和 SECOND 开始的单元中分别存放着两个 16 位组合的十进制(BCD 码)数,低位在前。编写程序求这两个数的组合的十进制和,并存到以 THIRD 开始的单元。

10. 编写一段程序,接收从键盘输入的 10 个数,输入回车符表示结束,然后将这些数加密后存于 BUFF 缓冲区中。加密表如下。

输入数字:0,1,2,3,4,5,6,7,8,9;密码数字:7,5,9,1,3,6,8,0,2,4。

11. 试编写程序,统计由 40000H 开始的 16KB 个单元中所存放的字符“A”的个数,并将结果存放在 DX 中。

12. 在当前数据段(DS),偏移地址为 DATAB 开始的顺序 80 个单元中,存放着某班 80 个同学某门考试成绩。按如下要求编写程序。

(1) 统计≥90 分、80～89 分、70～79 分、60～69 分、<60 分的人数各为多少,并将结果放在同一数据段、偏移地址为 BTRX 开始的顺序单元中。

(2) 求该班这门课的平均成绩为多少,并放在该数据段的 AVER 单元中。

13. 编写一个子程序,对 AL 中的数据进行偶校验,并将经过校验的结果放回 AL 中。

14. 利用上题的子程序,对以 80000H 开始的 256 个单元的数据加上偶校验,试编写程序。