第 5 章　子程序及宏指令设计

子程序及宏指令是汇编语言程序设计中的重要内容，可以简化程序结构，实现程序的模块化，提高汇编语言程序设计的质量和效率。本章主要介绍了子程序的定义、子程序的调用和返回、子程序的参数传递方法以及宏汇编中最具特色的部分：宏指令、重复汇编与条件汇编，并结合具体实例，讨论了子程序和宏指令的程序设计方法及技巧。

5.1　子程序设计方法

子程序是程序设计的基本概念。实际编程时，常把功能相对独立的程序段单独编写和调试，作为一个相对独立的模块供程序使用，这就是子程序，亦称过程，相当于高级语言中的过程和函数，调用子程序的程序称为主程序（或称为调用程序）。

5.1.1　子程序定义

子程序的定义是由一对过程定义伪指令 PROC 和 ENDP 来完成的，其一般格式如下：

```
子程序名 PROC[NEAR｜FAR]
        [保护现场]
        子程序体
        [恢复现场]
        RET
子程序名 ENDP
```

对子程序定义的具体规定如下：

（1）“子程序名”必须是一个合法的标识符，并且二者要前后一致；

（2）PROC 和 ENDP 必须是成对出现的关键字，它们分别表示子程序定义的开始和结束；

（3）子程序要有一条返回指令，返回指令是子程序的出口语句；

（4）子程序的类型有近（NEAR）、远（FAR）之分，其默认的类型是近类型，如果一个程序要被另一段程序调用，那么，其类型应定义为 FAR，否则，其类型可以是 NEAR。显然，NEAR 类型的子程序只能被与其同段的程序所调用。下面举例说明其使用方法。

1．调用程序和子程在同一个代码段的程序结构

【例 5.1】 代码段中含有主程序和一个子程序的情况，实际上可以含有多个子程序，子程序类型可以是 NEAR 或省略。

```
CODE    SEGMENT
MAIN    PROC FAR
```

```
            ⋮
        CALL SUB1
            ⋮
        RET
MAIN    ENDP
SUB1    PROC NEAR
            ⋮
        RET
SUB1    ENDP
CODE    ENDS
        END  MAIN
```

2．调用程序和子程序在不同段的程序结构

【例 5.2】 调用程序和子程序不在同一个代码段，其中的子程序为 FAR 类型，CALL 指令要显示说明是 FAR 类型。

```
CODE1   SEGMENT
MAIN    PROC FAR
            ⋮
        CALL  FAR PTR SUB1
            ⋮
        CALL  FAR PTR SUB2
            ⋮
        RET
MAIN    ENDP
CODE1   ENDS
CODE2   SEGMENT
SUB1    PROC FAR
            ⋮
        CALL  SUB2
            ⋮
        RET
SUB1    ENDP
SUB2    PROC FAR
            ⋮
        RET
SUB2    ENDP
CODE2   ENDS
        END MAIN
```

若一个子程序既被段间调用又被段内调用，则其类型必须是 FAR，如例 5.2 中的 SUB2。

5.1.2 寄存器内容的保存及恢复

由于 CPU 中的寄存器数量有限，所以主程序和子程序所使用的寄存器往往会发生冲

突。如果主程序在调用子程序之前的某个寄存器内容再从子程序返回后还有用，而子程序又恰好使用了同一个寄存器，这就破坏了该寄存器的原有内容，因而会造成程序运行错误，这是不允许的。因此为了保证子程序返回主程序时主程序能继续执行，就必须注意主程序现场的保护。现场是指主程序转到子程序前这一时刻主程序所使用的资源或状态，如标志寄存器、通用寄存器及存储器单元的内容。通常在转到子程序前将它们压入堆栈，以免子程序在执行时使用这些资源而发生冲突。而当子程序返回主程序时，主程序的现场必须恢复，即把它们从堆栈中弹出，保持原来的内容不被改变，主程序才能正确地继续执行。保护与恢复现场的工作通常安排在子程序中进行，在子程序的开始处安排一串保护现场的语句，在子程序返回前，再恢复有关内容。例如：

```
SUB1  PROC NEAR
      PUSH  AX
      PUSH  BX
      PUSH  CX
      PUSH  DX
        ⋮
      POP  DX
      POP  CX
      POP  BX
      POP  AX
SUB1  ENDP
```

在子程序设计时，应仔细考虑哪些寄存器是必须保存的，哪些寄存器是不必要或不应保存的。一般说来，子程序中用到的寄存器是应该保存的。但是，如果使用的寄存器在主程序和子程序之间传送参数的话，则这种寄存器就不一定需要保存，特别是用来向主程序传送结果的寄存器，就更不应该因保存和恢复寄存器而破坏了应该向主程序传送的信息。

5.1.3 子程序的调用及返回

子程序的调用及返回是通过 CALL 和 RET 指令实现的。应特别注意的是，为保证子程序的正确调用与返回，除定义时需正确选择属性外，还应该注意子程序运行期间的堆栈状态。当发生过程调用时，CALL 指令的功能之一是将返回地址压入堆栈，当子程序返回时，RET 则直接从当前栈顶取内容作为返回地址，而子程序中可能还有其他指令涉及堆栈操作。因此，要保证 RET 指令执行前堆栈栈顶的内容刚好是过程返回的地址，即相应 CALL 指令压栈的内容，否则将造成不可预测的错误。

关于子程序的调用有两种特殊的情况，即子程序嵌套调用和子程序递归调用。在子程序调用其他子程序时，称为子程序嵌套，只要堆栈空间允许，嵌套层次不限。若子程序中又调用该子程序本身则称为递归调用，递归的深度亦与堆栈大小有关。

5.1.4 子程序的参数传递

子程序一般是完成某种特定功能的程序段。当一个程序调用一个子程序时，通常都向子程序传递若干个数据让它来处理，当子程序处理完后，一般也向调用它的程序传递处理

结果，这种在调用程序和子程序之间的信息传递称为参数传递。调用程序向子程序传递的参数称为子程序的入口参数，子程序向调用它的程序传递的参数称为子程序的出口参数。

调用程序与子程序间通过参数传递建立联系，相互配合共同完成处理工作。传递参数的多少反映程序模块间的耦合程度。根据实际情况，子程序可以只有入口参数或只有出口参数，也可以同时存在入口参数和出口参数。参数的具体内容可以是数据本身（传数值）也可以是数据的存储地址（传地址）。方便灵活的参数传递是子程序设计的关键环节之一。

在汇编语言中，常用的3种参数传递方法包括利用寄存器传递参数、通过地址表传递参数地址和利用堆栈传递参数。下面分别进行讨论。

1．利用寄存器传递参数

由于CPU中的寄存器在任何程序中都是“可见”的，一个程序对某寄存器赋值后，在另一个程序中就能直接使用，所以用寄存器来传递参数最直接、简便，也是最常用的参数传递方式。但由于CPU中寄存器的个数和容量都是非常有限的，所以该方法适用于传递较少的参数信息。

主程序在调用子程序前，先将需要传递的参数保存在某些通用寄存器中，然后再调用子程序，这样，子程序就可直接从寄存器中获得入口参数。同样，出口参数可以通过寄存器返回给主程序。

【例5.3】 在字节型变量BCDBUF中有一个组合BCD码，试将其转换为二进制数后存入BINBUF单元中。

将组合BCD码分离出相应于十进制数的十位和个位，在进行十位数乘以10加上个位数的运算即可得到对应的二进制码。

```
.MODEL  SMALL
.STACK
.DATA
        BCDBUF  DB  65H
        BINBUF  DB  ?
.CODE
START:  MOV   AX,@DATA
        MOV   DS,AX
        MOV   AL,BCDBUF                 ;将要传递的参数放在寄存器AL中
        CALL  TRAN
        MOV   BINBUF,AL                 ;返回结果在AL中
        MOV   AX,4C00H
        INT   21H
TRAN    PROC  NEAR
        PUSHF
        PUSH  BX
        PUSH  CX
        MOV   AH,AL
        AND   AH,OFH                    ;分离出个位数
        MOV   BL,AH
        AND   AL,0F0H                   ;分离出十位数
```

```
        MOV   CL,04H
        ROR   AL,CL                          ;将十位数移至低 4 位
        MOV   BH,0AH
        MUL   BH                             ;十位数乘以 10
        ADD   AL,BL                          ;乘积与个位数相加
        POP   CX
        POP   BX
        POPF
        RET
TRAN    ENDP
        END START
```

2．通过地址表传递参数地址

有时直接传递参数本身不能方便地实现所要求功能，需要通过地址表传递参数地址的方法实现参数传递。具体方法是先建立一个地址表，该表由参数地址构成。然后把表的首地址通过寄存器或堆栈传递给子程序。

【例 5.4】 计算 ARY1、ARY2、ARY3 三个数组和，并把各自的和分别放入 SUM1、SUM2、SUM3 单元，其数组元素及结果均为字型数据。

显然数组求和应该用子程序完成，这样做使得代码真正共享。但在子程序中不能直接引用数组变量名，否则不能做到通用，在这种情况下可以通过传递参数地址的方法实现最终的参数传递。本例用数组首地址、元素个数的地址、结果地址构成一个地址表，通过寄存器把表的首地址传递给子程序，子程序通过地址表的参数地址访问到所需参数。

```
.MODEL  SMALL
.STACK
.DATA
ARY1    DW 1,2,3,4,5,6,7,8,9,10
COUNT1  DW ( $-ARY1) /2
SUM1    DW ?
ARY2    DW 10,20,30,40,50,60,70,80
COUNT2  DW ( $-ARY2) /2
SUM2    DW ?
ARY3    DW 100,200,300,400,500,600
COUNT3  DW ( $-ARY3) /2
SUM3    DW ?
TABLE
        DW 3    DUP (? )
.CODE
        MOV   AX,@DATA
        MOV   DS,AX
        MOV   TABLE,OFFSET ARY1              ;构造数组 1 的地址表
        MOV   TABLE+2,OFFSET COUNT1
        MOV   TABLE+4,OFFSET SUM1
        LEA   BX,TABLE                       ;通过寄存器传递地址表的首地址
```

```
        CALL  ARY-SUM                    ;求和数组 1 并保存结果
        MOV   TABLE,OFFSET ARY2          ;构造数组 2 的地址表
        MOV   TABLE+2,OFFSET COUNT2
        MOV   TABLE+4,OFFSET SUM2
        LEA   BX,TABLE                   ;通过寄存器传递地址表的首地址
        CALL  ARY-SUM                    ;求和数组 2 并保存结果
        MOV   TABLE,OFFSET ARY3          ;构造数组 3 的地址表
        MOV   TABLE+2,OFFSET COUNT3
        MOV   TABLE+4,OFFSET SUM3
        LEA   BX,TABLE                   ;通过寄存器传递地址表的首地址
        CALL  ARY-SUM                    ;求和数组 3 并保存结果
        MOV   AX, 4C00H
        INT   21H
ARY-SUM PROC  NEAR                       ;数组求和子程序
        PUSH  AX                         ;保存寄存器
        PUSH  CX
        PUSH  SI
        PUSH  DI
        MOV   SI,[BX]                    ;取数组起始地址
        MOV   DI,[BX+2]                  ;取元素个数地址
        MOV   CX,[DI]                    ;取元素个数
        MOV   DI,[BX+4]                  ;取结果地址
        XOR   AX,AX                      ;清 0 累加器
NEXT:   ADD   AX,[SI]                    ;累加和
        ADD   SI,2                       ;修改地址指针
        LOOP  NEXT
        MOV   [DI],AX                    ;存和
        POP   DI                         ;恢复寄存器
        POP   SI
        POP   CX
        POP   AX
        RET
ARY-SUM ENDP
        END START
```

可以看出，由于在子程序中没有直接访问模块中的变量名，而是通过地址访问变量的值，从而使得子程序更通用。

3．利用堆栈传递参数

通过堆栈传递参数或参数地址的步骤是：主程序把参数或参数地址压入堆栈；子程序使用堆栈中的参数或参数地址；子程序返回时要使用 RET N 指令调整 SP 指针，以便删除堆栈中已用过的参数，保持堆栈的正确状态，保证程序的正确返回。这种方式适用于参数较多，或子程序有多层嵌套、递归调用的情况。

【例 5.5】完成数组求和功能，其中求和由子程序实现，但要求通过堆栈传递参数地址。本例从功能实现上没有什么难度，但要特别注意堆栈的变化，以确保程序运行的正确性。

```
STACKSG SEGMENT STACK 'STK'
        DW   16 DUP (?)
TOS     LABEL  WORD
STACKSG ENDS
DATA    SEGMENT
ARY     DW  1,2,3,4,5,6,7,8,9,10
COUNT   DW($-ARY)/2
SUM     DW  ?
DATA    ENDS
CODE1   SEGMENT
MAIN    PROC FAR
        ASSUME CS：CODE1,DS：DATA
        PUSH  DS                          ;(1)
        XOR   AX,AX
        PUSH  AX                          ;(2)
        MOV   AX,DATA
        MOV   DS,AX
        MOV   AX,STACKSG
        MOV   SS,AX
        MOV   SP,OFFSET  TOS
        LEA   BX,ARY
        PUSH  BX                          ;(3)压入数组起始地址
        LEA   BX,COUNT
        PUSH  BX                          ;(4)压入元素个数地址
        LEA   BX,SUM
        PUSH  BX                          ;(5)压入和地址
        CALL  FAR PTR ARY-SUM             ;(6)调用求和子程序
        RET                               ;(18)
MAIN    ENDP
CODE1   ENDS
CODE2   SEGMENT
        ASSUME CS：CODE2
ARY-SUM PROC FAR                          ;数组求和子程序
        PUSH  BP                          ;(7)保存 BP 值
        MOV   BP,SP                       ;BP 作为堆栈数据的地址指针
        PUSH  AX                          ;(8),保存寄存器内容
        PUSH  CX                          ;(9)
        PUSH  SI                          ;(10)
        PUSH  DI                          ;(11)
        MOV   SI,[BP+10]                  ;得到数组起始地址
        MOV   DI,[BP+8]                   ;得到元素个数地址
        MOV   CX,[DI]                     ;得到元素个数
        MOV   DI,[BP+6]                   ;得到和地址
        XOR   AX,AX
NEXT:   ADD   AX,[SI]                     ;累加
```

```
        ADD   SI,2                  ;修改地址指针
        LOOP  NEXT
        MOV   [DI],AX               ;存和
        POP   DI                    ;(12),恢复寄存器内容
        POP   SI                    ;(13)
        POP   CX                    ;(14)
        POP   AX                    ;(15)
        POP   BP                    ;(16)
        RET 6                       ;(17),返回并调用 SP 指针
ARY-SUM ENDP
CODE2   ENDS
        END MAIN
```

可以看出，本例的子程序是通过把 BP 作为基址寄存器，并采用寄存器相对寻址方式访问堆栈中的参数的。注意：当 BP 作为基址寄存器时，其物理地址的计算约定与 SS 段寄存器配合。

下边跟踪一下本程序的堆栈变化。图 5.1 为程序中所有入栈操作对堆栈的影响。随着入栈数据的增加，SP 的值不断减小，堆栈可用空间也随之减少。

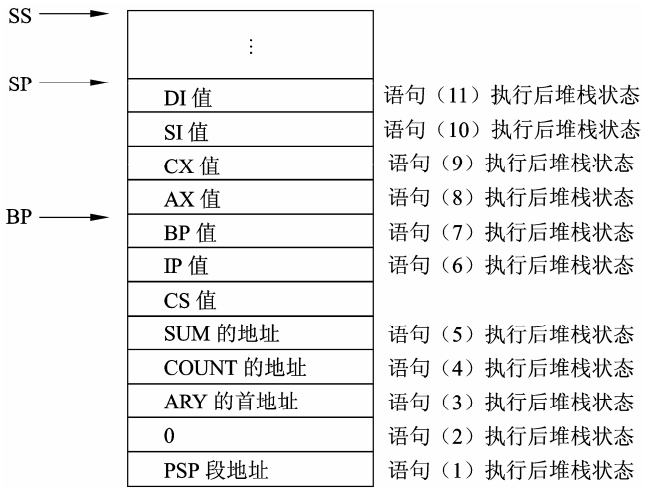

图 5.1　所有入栈操作对堆栈的影响

图 5.2 为已从子程序返回、而主程序的 RET 指令执行前的堆栈状态，其中 SP 指针前的数据表示执行语句（12）～（17）时已弹出的数据。随着弹出数据的增加，SP 的值不断增大，堆栈可用空间也随之增大。请特别注意子程序中语句（17）——RET 6 指令的使用，此指令从堆栈弹出返回地址后还要使 SP 值加 6，这样就跳过了通过堆栈传递的 3 个参数，或者说删除了它们，因此，当主程序的语句（18）——RET 指令被执行时，程序控制从栈顶弹出数字 0 给 IP，弹出 PSP 的段基址给 CS，于是执行 PSP∶0 处的 INT20H 指令，正确返回操作系统。

试想如果子程序返回时使用的是 RET 而不是 RET 6 指令，则 SP 的值就不会自动加 6，

虽然从子程序可以返回到主程序，但由于执行 RET 后 SP 指向的栈顶单元中存放的是 SUM 地址，当主程序执行到语句（18）的 RET 指令时，从栈顶弹出 SUM 的地址送 IP，弹出 COUNT 的地址送 CS，于是控制转向 CS：IP 所指向的地方去执行，显然它并不是所期望的返回地址 PSP：0，因此结果不可预料，经常可能发生的现象是子程序不能正常返回甚至于死机。从以上分析可以看出，通过堆栈传递参数时子程序的返回指令必须是 RET N 的形式，当堆栈操作是 16 位时 N 值应该是压入堆栈的参数个数的 2 倍。只有这样才能在执行 RET N 指令时删除堆栈中的无用参数，保持堆栈的正确状态，保证程序的正常运行。

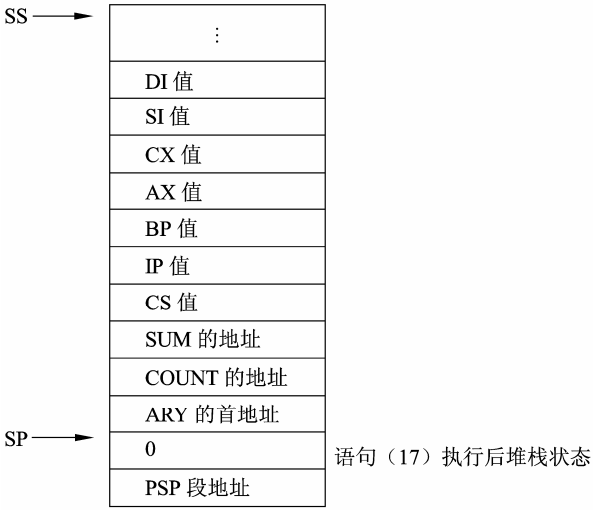

图 5.2　主程序的 RET 执行前堆栈状态

5.1.5　子程序嵌套

子程序不但可以被主程序调用，而且也可以被其他子程序调用。在一个子程序中调用另一个子程序被称为子程序的嵌套调用。只要堆栈空间允许，嵌套层次不限。子程序的嵌套调用示意图如图 5.3 所示。

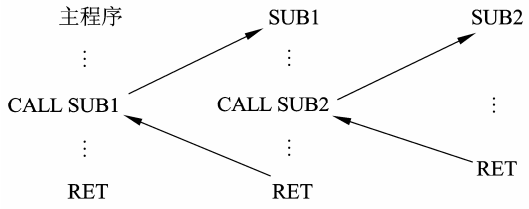

图 5.3　子程序的嵌套

嵌套子程序的设计并没有什么特殊要求，除子程序的调用和返回应正确使用 CALL 和 RET 指令外，要注意寄存器的保存和恢复，以避免各层次子程序之间因寄存器冲突而出错的情况发生。如果程序中使用了堆栈，则对堆栈的操作要格外小心，一方面要避免堆栈的溢出的发生，另一方面避免发生因堆栈使用中的问题而造成子程序不能正确返回的错误。

在子程序嵌套的情况下，如果一个子程序调用的子程序就是它自身，这就是递归调用。

这样的子程序称为递归子程序。递归子程序对应于数学上对函数的递归定义，它往往能设计出效率较高的程序，可完成相当复杂的计算，所以它是很有用的。因篇幅所限，本书不加进一步说明。

5.2 模块化程序设计

按照软件工程的思想，一个复杂的软件应该进行模块化设计，这样可以将一个复杂的问题分解成若干相对简单的问题，便于软件的开发和管理，同时也有利于软件质量的提高。模块化程序设计就是将一个大的软件按功能划分为许多功能相对独立的模块，模块间按统一规范连接，每个模块分别编写和调试，最后连接成一个完整的软件。在编写大型的汇编语言程序时，采用模块化程序设计方法显得十分必要，本节将简要介绍汇编语言支持的模块化程序设计方法。

5.2.1 模块划分

模块化程序设计的首要问题是合理地划分模块，这就要求将一个复杂问题进行分解，确定功能模块和接口关系。模块划分的一般原则如下。

（1）模块功能相对独立：每个模块的功能要明确，大小要适中，独立性要强，交换的接口信息要少，最好只有一个入口和一个出口。

（2）模块间的关系要明确：各模块最好再分层，形成树形层次结构。即上层模块可调用下层模块，下层模块可返回上层模块，反之则不然。这就使得各层间不会构成循环。

（3）程序中易变化的部分与不易变化的部分要分开，形成不同的模块，这样便于软件的升级。例如，系统软件中把与 CPU 有关的部分分出来形成一个专门模块。

MASM 汇编程序提供了两种模块划分的方法。一种是源程序级的模块划分，MASM 允许把一个大的源程序分别放在几个源程序文件中，但每个文件不能独立汇编和执行，汇编时必须通过包含伪指令（INCLUDE）将这些源程序文件结合起来，统一汇编后形成一个目标文件。这样划分的好处是便于源程序的管理与维护，同时，也利于这些文件的重复应用。

另一种模块划分方法是目标代码级的，每个模块可以单独编写和调试，形成若干个目标文件，最后由连接程序将这些目标文件连接起来形成一个完整的可执行文件。通常所说的模块化程序设计指的是后一种方式。

5.2.2 源程序文件包含的伪指令

MASM 汇编语言支持的源程序包含的伪指令（INCLUDE）与 C 语言中包含语句的作用类似，即将 INCLUDE 指令指定的源程序文件的内容插入到该伪指令所在位置。包含伪指令的格式为：

```
INCLUDE 源程序文件名
```

假如一个汇编程序由 F.ASM、F1.ASM、F2.ASM 3 个源程序文件组成，其中 F.ASM 为主程序文件，其余两个分别是不同类型的子程序文件，就可以方便地将不同的子程序文件

插入到主程序文件中，在 F.ASM 文件中，利用两条包含伪指令将 F1.ASM、F2.ASM 这两个文件包含到 F.ASM 文件中，此时只要对 F.ASM 进行汇编、连接后就能生成一个可执行程序 F.EXE。

在包含伪指令中，文件名可以含有路径，用来指明文件的存储位置，如果没有路径，MASM 则先在汇编命令行参数指定的目录下寻找，然后在当前目录下寻找，最后还会在环境参数 INCLUDE 指定的目录下寻找。

利用 INCLUDE 伪指令包含其他文件，其实质仍然是一个源程序，只是分成几个文件书写，被包含的文件不能独立汇编，因此，合并的源程序之间的各种标识符，如标号和名字等，应该统一规定，不能发生冲突。

5.2.3 模块间的连接

模块间的连接就是将多个相对独立的源程序文件分别单独汇编，形成若干个目标文件（.OBJ），然后用连接程序（LINK）将多个目标文件连接起来，生成一个可执行文件（.EXE）。连接程序的使用方法如下。

格式：LINK 目标文件 1+目标文件 2+ …

功能：实现对目标文件 1、目标文件 2 的连接，生成一个可执行文件。

说明：目标文件中，只能有一个是主程序模块，其余的应是子程序模块，程序的执行总是从主程序模块开始。

例如，一个汇编程序由 F.ASM、F1.ASM、F2.ASM 3 个源程序文件组成，其中 F.ASM 为主程序文件，其余两个分别是不同类型的子程序文件。首先对这 3 个文件进行汇编，分别得到 3 个目标文件 F.OBJ、F1.OBJ、F2.OBJ，然后利用 LINK 程序进行连接：

```
LLNK F.OBJ+F1.OBJ+F2.OBJ
```

最后便可得到一个可执行文件 F.EXE。

利用模块间连接方式开发源程序时，必须注意以下几个问题。

1．全局符号的使用

单个模块中使用的符号（变量、过程等）为局部符号，一个模块中定义的符号如不另加说明，均为局部符号，局部符号只能在定义它的模块中使用。

多个模块间可共同使用的符号为全局符号。在大型程序开发过程中，一个文件可能要利用另一个文件定义的变量或过程，为了实现这样的调用，必须将相应的变量或过程声明为全局符号。

PUBLIC 伪指令用于说明某个变量或过程可以被别的模块使用，其格式为：

```
PUBLIC  标识符[,标识符…]
```

EXTRN 伪指令用于说明某个变量或过程是在其他模块中定义的，其格式为：

```
EXTRN  标识符:类型,[标识符:类型,…]
```

其中，标识符是变量名、过程名等；类型是 BYTE/WORD/DWORD（变量）或 NEAR/ FAR（过程）。在一个源程序中，PUBLIC/EXTRN 语句可以有多条。各模块间的 PUBLIC/ EXTRN

伪指令要互相配对，并且指明的类型互相一致。

2．模块间的参数传递问题

模块间传递参数的基本方法与子程序间的参数传递方法相似。可以用寄存器或堆栈的方法传递数据或数据缓冲区指针，当然也可以用全局变量传递参数。

5.3 宏 汇 编

前面介绍的子程序设计方法有许多优点，如可以节省存储空间，优化程序结构，便于程序的调试和修改等。但是，子程序也存在一些不足，如使用子程序系统要额外付出存储空间和执行时间；调用子程序要进行参数传递及现场保护。若程序中重复部分只是一组较简单的语句序列，且要传送的参数较多的情况下，设计子程序就不合算。8086/8088 宏汇编语言提供了宏功能。宏是源程序中一段具有独立功能的程序段，将短小语句序列设计成宏指令，它只需要在源程序中定义一次，就可以在程序需要时多次调用，用一个类似语句代替语句序列，为程序设计提供了另外的途径。

使用宏功能可以减少由于重复书写而引起的错误，缩短源程序的长度，使源程序结构清晰、简洁，便于阅读，从而简化程序设计的工作。

5.3.1 宏定义、宏调用和宏展开

使用宏功能要按以下步骤进行：首先进行宏定义，然后在需要时进行宏调用，最后用实参代替形参进行宏展开。

1．宏定义

宏定义用伪指令 MACRO 和 ENDM 来定义。

格式：<宏指令名>　MACRO　[形参 1，形参 2，…]

　　<宏体>

　　ENDM

说明：

（1）MACRO 为宏的开始，ENDM 是宏的结束，它们必须成对出现。宏指令名是该宏定义的名称。宏指令代替的程序段由一系列指令语句和伪指令语句组成。

（2）参数表是任选的，可有可无。若存在多个参数，参数中间用逗号分隔。

（3）参数表的长度不允许超过 132 个字符。

（4）调用宏指令时，实参要与形参一一对应，当实参个数多于形参个数时，多余的实参被忽略；当实参个数少于形参时，系统自动填补 NUL。

（5）宏定义的指令名可用伪指令 PURGE 来取消，然后重新定义。

宏指令必须先定义后调用，宏指令具有比机器指令和伪指令更高的优先权，当宏指令与机器指令或伪指令同名时，宏汇编程序首先将它们一律处理成相应的宏展开，而不管与它同名的指令原来的功能。

2．宏调用

宏指令被定义后，在源程序中直接可以使用，将源程序中引用宏指令名代替某一特定程序段的过程称为宏调用。

格式：<宏指令名> [实参 1，实参 2，…]

实参可以是常数、寄存器、存储单元名以及用寻址方式能找到的地址或表达式等，宏汇编的这一特性是子程序所不及的。

3．宏展开

宏汇编程序在汇编期间，遇到宏调用，则嵌入宏体，将宏体中的指令插入到源程序宏指令所在的位置上，并用实参按位置对应关系一一替换宏体中的形参，这一过程称为宏展开。在汇编列表文件中，宏展开后留下的宏体语句在每行用符号“+”标志。

下面用一个例子说明宏定义、宏调用和宏展开的情况。

【例 5.6】 用宏指令定义两个字操作数相加，得到的第 3 个操作数作为结果。

宏定义：

```
ADDITION  MACRO OPR1,OPR2,RESULT
          PUSH  AX
          MOV   AX,OPR1
          ADD   AX,OPR2
          MOV   RESULT,AX
          POP   AX
          ENDM
```

宏调用：

```
          ADDITION  CX,VAR,XYZ[BX]
                ⋮
          ADDITION  240,BX,SAVE
```

宏展开：

```
+         PUSH  AX
+         MOV   AX,CX
+         ADD   AX,VAR
+         MOV   XYZ[BX],AX
+         POP   AX
              ⋮
+         PUSH  AX
+         MOV   AX,240
+         ADD   AX,BX
+         MOV   SAVE,AX
+         POP   AX
```

从上面的例子可以看出：由于宏指令可以带形参，调用时可以用实参取代，这就避免了子程序因参数传送带来的麻烦，使宏汇编的使用增加了灵活性。而且实参可以是常数、寄存器、存储单元名以及用寻址方式能找到的地址或表达式。但是，宏调用的工作方式和子程序调用的工作方式是完全不同的，图 5.4 说明了两者的区别。

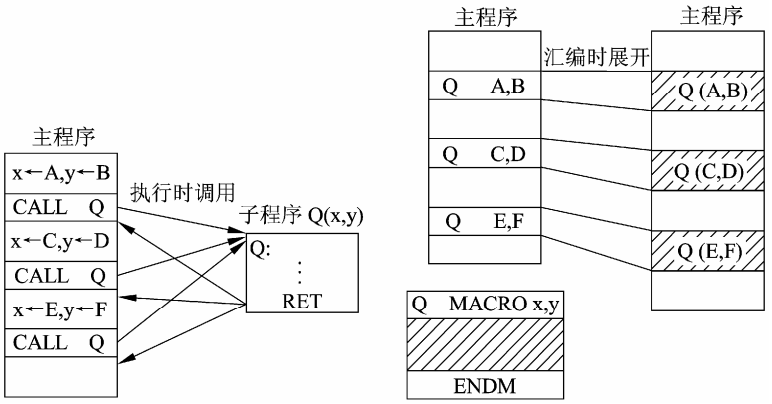

图 5.4　子程序调用和宏调用工作方式的区别

可以看出，子程序是在程序执行期间由主程序调用的，它只占有它自身大小的一个空间；而宏调用则是在汇编期间展开的，每调用一次就把宏定义展开一次，因而它占用的存储空间与调用次数有关。如果宏调用的次数较多，则其空间上的开销也是应该考虑的因素。一般来说，由于宏汇编可能占用较大的空间，所以代码较长的功能段往往使用子程序而不用宏汇编；而那些较短的且参数较多的功能段，则使用宏汇编就更为合理了。

5.3.2　宏定义和宏调用中的参数

下面以例子的形式说明宏定义和宏调用中的形参和实参的使用方法。

1．带间隔符的实参

在宏调用中，有时实参使用的是一串带间隔符（如空格、逗号等）的字符串，为使得间隔符成为实参的一部分，则要用尖括号将字符串括起来。

【例 5.7】 程序往往需要定义一个堆栈段，且定义的语句基本相同，不同的只是各个程序对堆栈段的大小和初值的要求，因此，可以用一个宏定义来实现对堆栈的定义。

```
STACKDEF  MACRO X
STACK     SEGMENT STACK
          DB   X
STACK     ENDS
          ENDM
```

若在当前程序中需要建立 200 个字节的堆栈区，初值为 0，则可以用宏调用：

```
STACKDEF  < 200 DUP(0) >
```

由于实参是一个带有空格符的字符串，因此要用尖括号括起来，成为一个整体。

宏展开如下：

```
+           STACK  SEGMENT STACK
+                  DB   200 DUP(0)
```

```
+           STACK   ENDS
```

2．宏代换参数

在有些场合，实参用符号表示，而用符号的值来替换形参，称为宏代换，这时符号前面要用特殊宏操作符 %，将 %后面的表达式的值来替换形参。

【例 5.8】 有宏定义如下：

```
NUM   MACRO X,Y,Z,W
      DB    X,Y,Z
      DW    W  DUP(0)
      ENDM
```

若宏调用为：

```
      J    EQU   100
      K    EQU   150
      NUM  30,%J+K,%K-80,%J*5
           ⋮
      NUM  30,J+K,K-80, J*5
```

则相应的宏展开为：

```
+     DB   30,250,70
+     DW   500 DUP(0)
       ⋮
+     DB   30,J+K,K-80
+     DW   J*5  DUP(0)
```

从上面两个宏调用语句展开的结果可以看出数字参数和一般参数的区别，如果有特殊宏代换符 %，则用 %后面的表达式的值来代换形参。符号要事先用 EQU 或“=”伪指令来定义，或者是汇编时能计算出值的表达式，而不能是变量名和寄存器名。

3．参数的连接

在宏定义中，形参可以是操作码的一部分、操作数的一部分或者是一个字符串，为了识别这种形参，需要在形参前面加上符号“&”，在用实参代换形参时，仍与前后符号连在一起，形成一个完整的符号或字符串。

【例 5.9】 形参是操作码的一部分。

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

若有如下的宏调用：

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

```
          ⋮
SHIFT  2,AX,HL
```

则宏展开为：

```
+       MOV   CL,4
+       SAR   DL,CL
          ⋮
+       MOV   CL,6
+       SAL   BX,CL
          ⋮
+       MOV   CL,2
+       SHL   AX,CL
```

从上面的例子可以看出，在宏体中，Z 与字符 S 相连，若 S 与 Z 之间没有特殊宏符号“&”，宏汇编程序就不将 Z 作为形参，而将 SZ 作为一个符号；若在 Z 的前面加“&”，则形参 Z 将被对应的实参所代换，并与字符 S 相连形成一个整体。

【例 5.10】 形参是操作数的一部分。

```
OPR  MACRO X
     JMP   T&X
     ENDM
```

若有如下的宏调用：

```
OPR  NEXT
```

则相应的宏展开为：

```
+         JMP   TNEXT
```

【例 5.11】 形参是字符串的情况。

```
STRING1  MACRO ABC,JKL,XYZ
         ABC&JKL  DB  'THIS  IS  A  &XYZ'
         ENDM
```

若有如下的宏调用：

```
STRING1  BUF,3,TRING
```

则相应的宏展开为：

```
+       BUF3  DB 'THIS  IS  A  STRING'
```

5.3.3 宏指令的嵌套

宏指令的嵌套有两种形式：一种是指在一个宏定义内套有另一个宏定义，另一种是宏定义中出现宏调用。

1．宏定义中再出现宏定义

这种嵌套结构的特点是内层宏定义是外层宏定义宏体的一部分，只有调用外层宏指令一次后，内层宏指令才被定义。也就是说，调用外层宏定义一次后，才能调用内层宏指令，否则就出错。

【例 5.12】 宏指令的嵌套。

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

若有如下的宏调用：

```
SETUP  4,AX,AL
    ⋮
SETUP  6,BX,AR
SHIFTAX
    ⋮
SHIFTBX
```

则相应的宏展开为：

```
+      SHIFTAX  MACRO
+      MOV   CL,4
+      SAL   AX,CL
+      ENDM
        ⋮
+      SHIFTBX  MACRO
+      MOV   CL,6
+      SAR   BX,CL
+      ENDM
+      MOV   CL,4
+      SAL   AX,CL
        ⋮
+      MOV   CL,6
+      SAR   BX,CL
```

2．宏定义中出现宏调用

这种宏嵌套结构主要是为了进一步简化宏定义而设计的，调用宏指令时，要求该宏指令涉及的宏调用必须已经定义。下面的例子可以实现求任意两个通用寄存器组成的有符号数的绝对值。

【例 5.13】 宏指令中出现宏调用。

```
INT21  MACRO FUNCTN
       MOV   AH,FUNCTN
       INT   21H
       ENDM
DISP   MACRO CHAR
       MOV   DL,CHAR
       INT21  O2H
       ENDM
```

宏调用：

```
DISP   '? '
```

则相应的宏展开为：

```
+      MOV  DL,'?'
+      MOV  AH,02H
+      INT  21H
```

5.3.4 宏汇编中的伪指令

1．MACRO 和 ENDM

这是使用宏操作时必不可少的指令，用于对宏进行定义。

2．PURGE

一个宏指令定义可以用伪指令 PURGE 来取消，然后就可以再重新定义。

格式：PURGE <宏指令名> [，<宏指令名>，…]。

功能：取消多个宏定义。

取消宏定义的含义是使该宏定义成为空，程序中如果出现一个已被取消宏定义的宏调用，则汇编程序将不会指示出错，但它将忽略该宏调用，当然也不会予以展开。

3．LOCAL

在某些宏定义中，常常需要定义一些变量或标号，当这些宏定义在同一个程序中多次调用并宏展开后，就会出现变量或标号重复定义的错误。

【例 5.14】 有如下宏定义：

```
SUM   MACRO X,Y
      MOV   CX,X
      MOV   BX,Y
      MOV   AX,0
NEXT: ADD   AX,BX
      ADD   BX,2
      LOOP  NEXT
      ENDM
```

若某些程序对此宏定义有两次调用：

```
SUM   100,1
```

```
      ⋮
SUM   50, 2
```

相应的宏展开为：

```
+         MOV   CX,100
+         MOV   BX,1
+         MOV   AX,0
+   NEXT: ADD   AX,BX
+         ADD   BX,2
+         LOOP  NEXT
            ⋮
+         MOV   CX,50
+         MOV   BX,2
+         MOV   AX,0
+   NEXT: ADD   AX,BX
+         ADD   BX,2
+         LOOP  NEXT
```

由此可见，标号 NEXT 出现了两次，这就引起了重复定义的错误。为了避免这种情况，可以将标号 NEXT 定义成形参，在每次调用时，均用不同的实参去代换。

格式：LOCAL <形参>[，<形参>]。

功能：在宏展开时，让宏汇编程序自动为其后的形参顺序生成特殊符号（范围为??0000～??FFFFH），并用这些特殊符号来取代宏体中的形参，从而避免了符号重复定义的错误。

LOCAL 语句只能作为宏体中的第一条语句，它后面的形参即为宏定义中所定义的变量和标号。如前面求若干个奇数（或偶数）的和的宏定义，只需在第 2 行加一条伪指令：

```
LOCAL  NEXT
```

则相应的宏展开为：

```
+           MOV   CX,100
+           MOV   BX,1
+           MOV   AX,0
+ ??0000:   ADD   AX,BX
+           ADD   BX,2
+           LOOP  ??0000
                 ⋮
+           MOV   CX,50
+           MOV   BX,2
+           MOV   AX,0
+ ??0001:   ADD   AX,BX
+           ADD   BX,2
+           LOOP  ??0001
```

5.3.5 重复汇编

有时汇编程序需要连续地重复完成相同的或者几乎完全相同的一组代码，这时可使用重复汇编，比把它们定义成宏指令更要简化程序设计。重复汇编可分为重复次数已知的重复汇编和重复次数未知的重复汇编。

1．给定次数的重复汇编伪指令

格式：REPT<表达式>

<重复块>

ENDM

功能：让宏汇编程序将重复块连续地汇编表达式所指定的次数。

【例 5.15】 把字符 A～Z 的 ASCII 码填入数组 ARRAY 中。

```
        CHAR ='A'
REPT    26
        DB    CHAR
        CHAR =CHAR +1
        ENDM
```

在汇编过程中，重复块被连续复制 26 次，其展开后的形式为：

```
+       DB  41H
+       DB  42H
        ⋮
+       DB  5AH
```

2．不定次数的重复汇编伪指令

1）IRP 伪操作

格式：IRP<形参>，<实参 1，实参 2,…，实参 *n* >

<重复块>

ENDM

功能：让宏汇编程序将重复块重复汇编由实参个数所给定的次数，并在每次重复时，依次用相应位置的实参代换形参。

注意：实参必须用尖括号括起来，并且各实参之间要用逗号分隔。

【例 5.16】 将 4 个数据寄存器 AX、BX、CX 和 DX 的内容入栈。

```
IRP REG,<AX,BX,CX,DX>
    PUSH  REG
    ENDM
```

宏汇编程序在汇编时，将对语句“PUSH　REG”连续汇编 4 次，并在每次重复时，依次以实参 AX、BX、CX、DX 来代换形参 REG，展开后的形式为：

```
+   PUSH  AX
+   PUSH  BX
```

```
+    PUSH  CX
+    PUSH  DX
```

2）IRPC 伪操作

格式：IRPC<形参>，<字符串>

<重复块>

ENDM

功能：将重复块重复汇编，重复的次数由字符串的字符个数决定，并在每次重复时，依次用相应位置的字符代换形参。

注意：字符串不带引号。

上例中，将 4 个数据寄存器的内容入栈，也可以用如下的宏定义来实现：

```
IRPC  REG,ABCD
      PUSH REG&X
      ENDM
```

5.3.6 条件汇编

条件汇编是根据某些条件是否成立（为真）来决定是否汇编某一段语句，即在编写源程序时利用条件汇编指令，采用类似于两个分支的方法编写出程序。在汇编时，宏汇编程序就可以根据条件汇编伪指令指定的条件进行测试，只有满足条件的那部分语句生成目标代码，而对不满足条件的部分则不予汇编，也就不会生成目标代码。大多数条件伪指令出现在宏定义中，只有把条件伪指令和宏指令结合使用才能显示出伪指令的优越性。条件汇编与重复汇编一样，仅在程序汇编期间，根据条件决定汇编或不汇编，而不是在程序的执行期间进行。

格式：IF<条件表达式或参数>

<语句体 1>

ELSE

<语句体 2>

ENDIF

功能：若条件成立，则汇编语句体 1 中的语句；否则，对语句体 2 进行汇编。当条件不满足，且语句中也没有 ELSE 及语句体 2，则汇编程序跳过这一组条件汇编语句，执行 ENDIF 后面的指令。

条件汇编伪指令共有 10 条，分成互补的 5 对，分别用来测试表达式、符号定义、参数、两个字符串和扫描次数的。

IF<表达式>	表达式≠0，则满足条件
IFE<表达式>	表达式=0，则满足条件
IFDEF<符号〉	符号已定义或被说明为 EXTRN
IFNDEF<符号>	符号未定义或未被说明为 EXTRN
IFB<变量>	变量为空格
IFNB<变量>	变量不为空格

IFIDN<变量1，变量2>	变量1与变量2的字符串相同
IFNIDN<变量1，变量2>	变量1与变量2的字符串不同
IF1	在汇编程序第一次汇编扫描期间满足条件
IF2	在汇编程序第二次汇编扫描期间满足条件

上述IF和IFE的表达式中可以使用关系操作符EQ、NE、LT、LE、GT和GE。

【例5.17】 条件汇编的简单应用。

```
    ARG1  EQU  35H
    ARG2=NOT ARG1
IF  ARG1 OR ARG2  EQ 0FFFFH
    MOV   AX,ARG1
    MOV   BX,ARG2
    ADD   AX,BX
IF  ARG1 AND ARG2 EQ 0FFFFH
    SUB   AX,CX
    IFE   ARG1
    ADD   AX,DX
ENDIF
    MOV   [SI],AX
ENDIF
    MOV   [DI],AX
END IF
```

上面的程序中，ARG1为35H，ARG2为0FFCAH，或的结果为0FFFFH，第一个条件汇编伪操作中条件成立，所以第一个IF与最后一个ENDIF之间的语句被汇编。在汇编这个条件块时，又遇到了IF条件，语句中的条件不成立，则第二个IF与ENDIF之间的条件块不被汇编。

习　　题

1．调用子程序指令的功能是什么？其操作过程包含哪几个步骤？

2．试编制一个多精度数求补的子程序，为提高程序的通用性，要求调用子程序时把多精度数的首地址放在 SI 中（低字节放低位、 高字节放高位），多精度数字节数放在CL中。

3．试编制两个长度不同的多精度整数求和子程序，为提高程序的通用性，要求调用子程序时把两个多精度数的首地址分别存放在SI、DI中（低字节放低位、高字节放高位），多精度数字节数分别存放在CL、CH中。

4．试编写一个子程序用以统计字数组中零元素的个数，参数采用堆栈传递，入口参数为：数组存储区首地址，数组长度 N。出口参数为零元素的个数，并写出CALL指令执行前后和RET指令执行前后的堆栈情况。

5．试编程计算两个数 X 和 Y 最大公倍数的子程序。

6．试编制一个计算两个正整数 X 和 Y 最大公约数的子程序。

7．设一维数组 LIST1、LIST2、LIST3 中分别存放了若干个单字节长的带符号数，试编制程序使三个表中的数据都按降序排列。表中元素的个数分别在 NUM1、NUM2、NUM3 三个单元中。

8．试编制一个通用多字节数求和的宏指令。

9．已知宏定义如下：

```
DIF     MACRO  X,Y
        MOV   AX,X
        SUB   AX,Y
        ENDM
ABSDIF  MACRO  X1,X2,X3
        LOCAL  CONT
        PUSH  AX
        DIF   X1,X2
        CMP   AX,0
        JGE   CONT
        NEG   AX
CONT:   MOV   X3,AX
        POP   AX
        ENDM
```

试展开以下调用，并判定调用是否有效。

（1）`ABSDIF  P1,P2,DISTANCE`

（2）`ABSDIF [BX],[SI],X[DI],CX`

（3）`ABSDIF [BX][SI],X[BX][SI],240H`

（4）`ABSDIF  AX,BX,CX`

（5）`ABSDIF  AX,AX,AX`

10．试编写一条含有重复汇编的宏指令，定义一个 0～9 数字的平方表。

11．用宏定义及重复伪操作把 TAB、TAB+1、TAB+2、…、TAB20 的内容存入堆栈。

12．用重复伪操作建立 100 个字的数组，要求数组中每个字的内容是其下一个字的地址，最后一个字的内容是第一个字的地址。