第 5 章　汇编语言程序设计

汇编语言是一种面向机器的低级语言，利用汇编语言设计的程序效率高、实时性强，还能直接控制硬件，能充分发挥硬件的潜力。但现在主流的开发工具是高级语言，所以本章不仅介绍 DOS 环境下的汇编语言程序设计，而且介绍了 Windows 下的汇编开发方法。汇编语言源程序的组成部分有：模块、段、子程序和宏等。第 1 部分介绍汇编语言的语法规定，着重叙述汇编语言程序的格式、各种伪指令的格式及用法、常用标识符的定义及应用、宏汇编的应用；第 2 部分介绍汇编语言程序设计的方法、步骤，着重叙述汇编语言程序的三大结构，并通过编程举例介绍常见问题的解决方法；第 3 部分介绍子程序结构及设计方法，着重叙述子程序调用/返回的原理、子程序入/出口参数传递方法、子程序的嵌套/递归调用，并结合实例说明子程序设计方法；最后两部分介绍 Windows 汇编语言程序设计以及与高级语言的混合编程方法。

- 汇编语言源程序书写格式及常用伪指令语句
- 结构化程序设计方法：顺序结构程序设计、分支结构程序设计、循环结构程序设计
- 宏汇编
- 子程序设计方法
- Windows 汇编语言程序设计
- 汇编语言与高级语言的混合编程

5.1　汇编语言的特点

汇编语言（Assembly Language）是一种以处理器指令系统为基础的低级程序设计语言，它采用助记符来表示操作码，用标识符来表示操作数地址码。

用汇编语言编写的程序具有以下特性：

（1）与机器相关。汇编语言是一种面向机器的程序设计语言，是机器指令的符号表示。不同类型 CPU 的机器指令系统不同，对应的汇编语言也就不同，所以汇编语言程序与机器有着密切的关系。因此，汇编语言程序的通用性和可移植性较低。

（2）执行效率高。汇编语言程序能直接管理和控制硬件设备，直接与存储器、接口电路等打交道，还能申请中断。程序员在编写程序时，可以对机器内部的各种资源进行合理的安排，

编写出最优化的程序，因而用汇编语言编写的程序执行效率高、占用存储空间小、运行速度快。

（3）编程复杂。利用汇编语言编写程序完成某项工作时，必须了解 CPU 完成该项工作的每一个细节，用一系列汇编指令一步一步来实现；另外，在编写汇编语言程序时，还要考虑机器资源的限制、汇编指令的细节和限制等，增加了编写程序的复杂性，需要程序员对计算机硬件和操作系统有相当深入的了解。

5.2　汇编语言程序结构和基本语法

在 MS DOS 环境下，用 8086/8088 汇编语言开发设计出的既节省空间、又快速高效的程序代码，可以直接控制计算机的硬件。到了 Windows 环境下，虽然仍可使用 8086/8088 汇编语言，但这个环境已不是原有的 MS DOS 环境，不能像以前那样，随意对系统编程去控制计算机了。下面首先介绍 DOS 环境下的汇编程序例子。

5.2.1　示例程序

通过下面一个完整的汇编语言源程序来讨论汇编语言程序的格式，该程序的功能是实现 C=A+B，其中 A、B、C 均为字节数据。

例 5-1

```
DATA    SEGMENT                 ;定义段 DATA
    A   DB   12H                ;定义变量 A，其值为 12H
    B   DB   34H                ;定义变量 B，其值为 34H
    C   DB   ?                  ;定义变量 C，但没有赋值
DATA    ENDS                    ;DATA 段定义结束
CODE    SEGMENT                 ;定义段 CODE
ASSUME  CS:CODE, DS:DATA        ;规定 DATA、CODE 分别为数据段和代码段
START:  MOV   AX,DATA           ;用标号 START 指明程序执行的起始点
        MOV   DS,AX             ;给数据段寄存器 DS 赋值
        MOV   AL,A              ;将变量 A 的值送入寄存器 AL
        ADD   AL,B              ;将 AL 的值与变量 B 的值相加，并将其和存入 AL
        MOV   C,AL              ;将 AL 的值送给变量 C
        MOV   AH,4CH
        INT   21H               ;调用 DOS 中断，退出程序并返回 DOS 状态
CODE  ENDS                      ;CODE 段定义结束
END   START                     ;整个源程序结束
```

从该例中可以看出汇编语言源程序具有以下特点：

（1）汇编语言源程序由若干个段组成（完整的汇编语言源程序由数据段、代码段、附加段、堆栈段组成，其中代码段是不可缺少的），在代码段中用 ASSUME 伪指令将段地址与段寄存器的对应关系告诉汇编程序，每个段以 SEGMENT 语句开始，以 ENDS 语句结束，整个源程序以 END 结束。

（2）段由若干语句组成，一条语句一般写在一行上，书写时语句的各部分应尽量对齐。

（3）汇编语言程序中至少要有一个启动标号，作为程序开始执行时目标代码的入口地址。启动标号常用 START、BEGIN 等命名。

（4）为增加程序的可读性，可在汇编语言语句“;”后加上注释。

（5）为保证在执行过程中数据段地址的正确性，在源程序中需要对 DS 寄存器进行初始化。

（6）为了在程序结束时返回 DOS，一般通过调用 DOS 中断的 4CH 子功能来实现。

5.2.2 基本概念

1. 汇编语言中的语句

汇编语言源程序由语句序列构成，其语句序列可分为指令语句、伪指令语句、宏指令语句 3 种类型。

（1）指令语句。指令语句是可执行语句（即第 3 章中介绍的处理器指令系统），在汇编后要产生对应的目标代码，CPU 根据这些代码执行相应的操作。

格式：[标号:] <指令助记符> [操作数] [;注释]

例如：

```
START:    MOV    AX,DATA    ;用标号START指明程序执行的起始点
```

（2）伪指令语句。伪指令是不可执行语句，在汇编中不产生目标代码，用于指示汇编程序如何汇编源程序，利用它定义和说明常量与变量的属性及存储器单元的分配等。

格式：[名字] <伪指令助记符> [操作数] [;注释]

例如：

```
A    DB    12H                ;定义变量A，其值为12H
```

（3）宏指令语句。宏指令是以一个宏名定义的一段指令序列，在汇编中凡是出现宏指令语句的地方，都会有相应的指令语句序列的目标代码插入。

格式：[标号:] <宏名> [实参表] [;注释]

汇编中的大部分指令语句与 8086 指令相对应，这里不再赘述。本节将着重介绍伪指令语句和宏指令语句。

2. 汇编语句使用说明

（1）标号和名字称为标识符，汇编语言中标识符的组成规则如下：

- 标识符由字母、数字及规定的特殊符号（如 _、$、?、@）组成。
- 标识符必须以字母打头。
- 标识符字符长度不得超过 31。
- 默认情况下，汇编程序不区别标识符中字母的大小写。
- 用户定义标识符必须是唯一的，且不能与汇编语言专用的保留字重名。

（2）标号用来指向一条指令或宏指令，表示后面的指令第一个字节存放的内存地址，标号常作为转移指令的操作数，确定程序转移的目标地址；名字用来指向一条伪指令，用作变量名时，表示变量存放在内存中首字节的地址。

（3）名字和标号都具有以下 3 种属性：

- 段属性：表示标号或变量所在段基址，标号的段基址在 CS 段寄存器中，变量的段基址在 DS 或 ES 中。
- 偏移属性：表示标号或变量所在的段内偏移地址，它代表从段的起始地址到定义标号或变量的位置之间的字节数，段基址和偏移地址组成标号或变量的逻辑地址。
- 类型属性：当标号作为转移类指令的操作数时，可在段内或段间转移，其属性有 NEAR（段内转移）和 FAR（段间转移）两种，若没有对标号进行类型说明，就默认为 NEAR 属性；对于变量，类型属性说明变量在内存中占多少个字节，其属性有 BYTE（字节）、WORD（字）、DOUBLE WORD（双字）3 种。

（4）指令的操作数可以是立即数、寄存器和存储单元；伪指令的操作数可以是常数、变量名、表达式等；若有多个操作数时，操作数之间用逗号间隔。

（5）分号“;”后的部分为注释内容，用以增加源程序的可读性，汇编程序在翻译源程序时将跳过该部分，对它们不做任何处理。

3. 汇编语言中的常量与变量

（1）常量：汇编中允许的常量有整数常量和字符串常量两种。

1）整数常量：整数常量可以采用 4 种表示方法：

- 二进制常量：由数字 0、1 组成的序列，且以字母 B 结尾，如 10101010B。
- 十进制常量：由数字 0～9 组成的序列，结尾可以加上字母 D，如 9876D 或 6575。
- 八进制常量：由数字 0～7 组成的序列，且以字母 Q（或字母 O）结尾，如 255Q、377O。
- 十六进制常量：由数字 0～9、字母 A～F（或 A～F）组成的序列，且以字母 H 结尾，如 3456H、0AB19H（为了避免与标识符相混淆，十六进制数在语句中必须以数字打头，凡是以字母 A～F 开始的十六进制数，必须在前面加上数字 0）。

2）字符串常量：字符串常量是由单引号或双引号括起来的单个字符或多个字符构成的，汇编程序把引号中的字符翻译成它的 ASCII 码值，如'A'（等于 41H）、'BC'（等于 4243H）、"HELLO"等。

（2）变量：汇编语言中的变量用来表示存放在内存中的操作数，它的值是可以改变的，变量的值就是操作数在内存中首字节的地址，变量要事先定义才能使用（详见 5.2.3 节）。

4. 汇编语言中的运算符与表达式

（1）运算符：汇编语言中的运算符分为六大类：算术运算符、移位运算符、逻辑运算符、关系运算符、分析运算符、合成运算符，如表 5-1 所示。

表 5-1　汇编语言中的运算符

运算符			实例	表达式的值/功能说明
类型	符号	名称		
算术运算符	+	加	1+2	3
	-	减	4-3	1
	*	乘	5*6	30
	/	除	64/8	8
	MOD	取余	9MOD7	2
移位运算符	SHL	逻辑左移	0011B SHL 2	1100B
	SHR	逻辑右移	1100B SHR 2	0011B
逻辑运算符	NOT	非	NOT 0001B	1110B
	AND	与	1000B AND 0001B	0000B
	OR	或	1000B OR 0001B	1001B
	XOR	异或	1000B XOR 0001B	1001B
关系运算符	EQ	相等	10H EQ 10	假（用全 0 表示）
	NE	不等	10H NE 10	真（用全 1 表示）
	LT	小于	10H LT 10	假（用全 0 表示）

续表

运算符			实例	表达式的值/功能说明
类型	符号	名称		
关系运算符	LE	小于等于	10 LE 0AH	真（用全 1 表示）
	GT	大于	10H GT 10	真（用全 1 表示）
	GE	大于等于	10H GE 100	假（用全 0 表示）
分析运算符	SEG	求段基址	SEG X	X 所在段的段基址
	OFFSET	求偏移地址	OFFSET X	X 在段内的偏移地址
	LENGTH	求变量包含的单元数	LENGTH X	X 包含的单元数（详见例 5-4）
	TYPE	求变量的字节数	TYPE X	X 的字节数（详见例 5-4）
	SIZE	求变量的总字节数	SIZE X	X 的总字节数（详见例 5-4）
合成运算符	PTR	修改类型	WORD PTR X	访问 X 对应的字数据（详见例 5-4）
	THIS	指定类型	X EQU THIS BYTE	指定变量 X 为字节属性（详见例 5-6）
	:	段超越	ES:[1000H]	指定访问附加段中偏移地址为 1000H 的单元
	HIGH	求高字节	HIGH 1234H	12H
	LOW	求低字节	LOW 1234H	34H
	SHORT	短转移说明	JMP SHORT NEXT	说明转移地址在下一条指令地址的 -128～127 个字节范围

（2）表达式：表达式是常数、寄存器、标号、变量与一些运算符和操作码相组合的序列。表达式的运算不由 CPU 完成，而是在程序汇编过程中进行计算确定，并将表达式的结果作为操作数参与指令所规定的操作。

当各种运算符同时出现在同一表达式中时，按照运算符的优先级进行计算，对于优先级相同的运算符，按照从左到右的顺序进行计算（运算符优先级顺序如表 5-2 所示）。

表 5-2　汇编语言中运算符的优先级

优先级		运算符
高 ↑ 低	1	LENGTH、SIZE、WIDTH、MASK、( )、[ ]、< >
	2	PTR、OFFSET、SEG、TYPE、THIS
	3	HIGH、LOW
	4	+、–（单目运算，表示取正、取负）
	5	*、/、MOD、SHL、SHR
	6	+、–（双目运算，表示加、减）
	7	EQ、NE、LT、LE、GT、GE
	8	NOT
	9	AND
	10	OR、XOR
	11	SHORT

例 5-2

```
MOV   AX, 1*2+3                    ;等价于 MOV AX,5
MOV   AX, X+4                      ;等价于 MOV AX,X[4]，注意这里是 X 地址加 4
                                   ;不是 X 的值加 4
MOV   AH, 0001B   SHL 3            ;等价于 MOV AH,00001000B
MOV   BH, 1000B   SHL(1+2)         ;等价于 MOV BH,00000001B
MOV   DH, NOT 10000000B            ;等价于 MOV DH,01111111B
MOV   DL, 00011010B AND 00101011B  ;等价于 MOV DL,00001010B
MOV   AX, 10 EQ 1010B              ;等价于 MOV AX, 0FFFFH
MOV   BX, 10H GT 10                ;等价于 MOV BH, 0FFFFH
ADD   CX, 99H LE 99                ;等价于 ADD CX, 0000H
MOV   DX, SEG X                    ;假设 X 为数据段内定义的变量，则该语句等价于
                                   ;MOV DX, DS
MOV   AX, OFFSET X                 ;等价于 LEA AX,X
MOV   DL, 00011010B AND 00101011B  ;等价于 MOV DL,00001010B
MOV   AX, 10 EQ 1010B              ;等价于 MOV AX, 0FFFFH
MOV   BX, 10H GT 10                ;等价于 MOV BH, 0FFFFH
ADD   CX, 99H LE 99                ;等价于 ADD CX, 0000H
MOV   DX, SEG X                    ;假设 X 为数据段内定义的变量，则该语句等价于
                                   ;MOV DX, DS
MOV   AX, OFFSET X                 ;等价于 LEA AX,X
MOV   AX, [1000H]                  ;将数据段中偏移地址为 1000H 的字数据送给 AX
MOV   AH,HIGH 0ABCDH               ;等价于 MOV AH, 0ABH
MOV   AL,LOW 0ABCDH                ;等价于 MOV AL, 0CDH
MOV   AX, ES:[1000H]               ;将附加段中偏移地址为 1000H 的字数据送给 AX
                                   ;段超越用来指定地址是在附加段中
```

5.2.3　伪指令

汇编语言中有丰富的伪指令。依其功能可将其分为数据定义伪指令、符号定义伪指令、段定义伪指令、段分配伪指令、过程定义伪指令、模块定义伪指令、结构定义伪指令和记录定义伪指令等。

1. 数据定义伪指令

数据定义伪指令用来为变量申请固定长度的存储空间，并可同时将相应的存储单元初始化。

格式：[变量名]　伪指令助记符　初值表

（1）变量名为用户自定义标识符，表示初值表中首个元素的逻辑地址，可以通过变量名来访问它所指示的存储单元，有时也可以省略变量名。

（2）变量定义伪指令有 5 种形式：

- DB 定义字节变量，即其后的每个操作数均占 1 个字节。
- DW 定义字变量，即其后的每个操作数均占 2 个字节。
- DD 定义双字变量，即其后的每个操作数均占 4 个字节。
- DQ 定义 4 字变量，即其后的每个操作数均占 8 个字节。

- DT 定义 10 字节变量，即其后的每个操作数均占 10 个字节。

注意：*存放多字节数据时，数据高字节存放在高地址单元，低字节存放在低地址单元。*

（3）初值表给出变量的初始化值，有多个值时用逗号分隔，初始化值可以是数值常数，也可以是表达式、？，还可以由$、重复操作符 DUP 组成。其中：

- ?：表示未赋初值。
- $：表示将要分配的内存单元的偏移地址。
- DUP：表示重复初值，其格式为：

```
重复次数  DUP(重复参数)          ;重复参数可以是多个，之间用逗号间隔
```

例如：

```
2  DUP(1, 2)                    ;等价于 1，2，1，2
```

2. *起始位置定位伪指令* ORG

在数据段内一般从偏移地址为 0 的存储单元开始，依次按顺序分配内存单元；使用 ORG 可以指定从某一个偏移地址开始分配内存单元。

ORG 指令格式：

```
ORG 表达式              ;从表达式的值指定的偏移地址开始分配的内存单元
```

例 5-3

```
DATA  SEGMENT
    X   DB   64*2-100, 'D'          ;两初始化值的十六进制表示分别为 1CH、44H
        DB   'CHN'                  ;省略了变量名，初始化值的十六进制表示为 43H、48H、4EH
        DW   ?                      ;只分配两个字节空间，未赋初值
    ORG  0100H                      ;指定从偏移地址为 0100H 单元开始分配内存单元
    Y   DW   2   DUP(100)           ;定义了两个字数据 0064H
    M   DB   2   DUP(1,2   DUP(1,2))  ;定义了十个字节数据 01H、01H、02H、01H、02H、01H、
                                    ;01H、02H、01H、02H
    Z   DW   $-10                   ;此时要分配的单元偏移地址为 010EH，故初始化值
                                    ;为 010EH-10=0104H
DATA  ENDS
```

本例中数据段中的数据在内存中的存放如图5-1所示。

例 5-4　分析运算符的使用（数据定义同例 5-3）。

```
;说明：LENGTH 用来求其后的变量包含的单元数，即变量用 DUP 初始化时，返回 DUP 重复次数
;对于不是用 DUP 初始化的变量，则返回 1
MOV   AX,  LENGTH  Y            ;等价于 MOV   AX, 2
MOV   AX,  LENGTH  M            ;等价于 MOV   AX, 2
MOV   AX,  LENGTH  X            ;等价于 MOV   AX, 1
;说明：TYPE 用来求其后的变量或标号的属性值（变量或标号的属性值如表 5-3 所示）
MOV   AX,  TYPE  Y              ;等价于 MOV AX,2
MOV   AX,  TYPE  X              ;等价于 MOV AX,1
;说明：SIZE 用来求其后的变量包含多少个字节，即 SIZE=LENGTH×TYPE
MOV   AX,  SIZE  Y              ;等价于 MOV AX,4
MOV   AX,  SIZE  X              ;等价于 MOV AX,1
;说明：PTR 用来求其后的变量或标号的类型，格式为：类型 PTR 表达式（其中，类型为 BYTE、
;WORD、DWORD、NEAR 或 FAR）
MOV   AX,  X                    ;（×）本语句源操作数 X 为字节类型，目的操作数 AX 为
                                ;字类型，两者类型不匹配
MOV   AX,  WORD PTR X           ;（√）利用 PTR 运算符修改 X 类型为字类型，即源操作数
                                ;为从 X 开始的字数据，等价于 MOV AX, 441CH
```

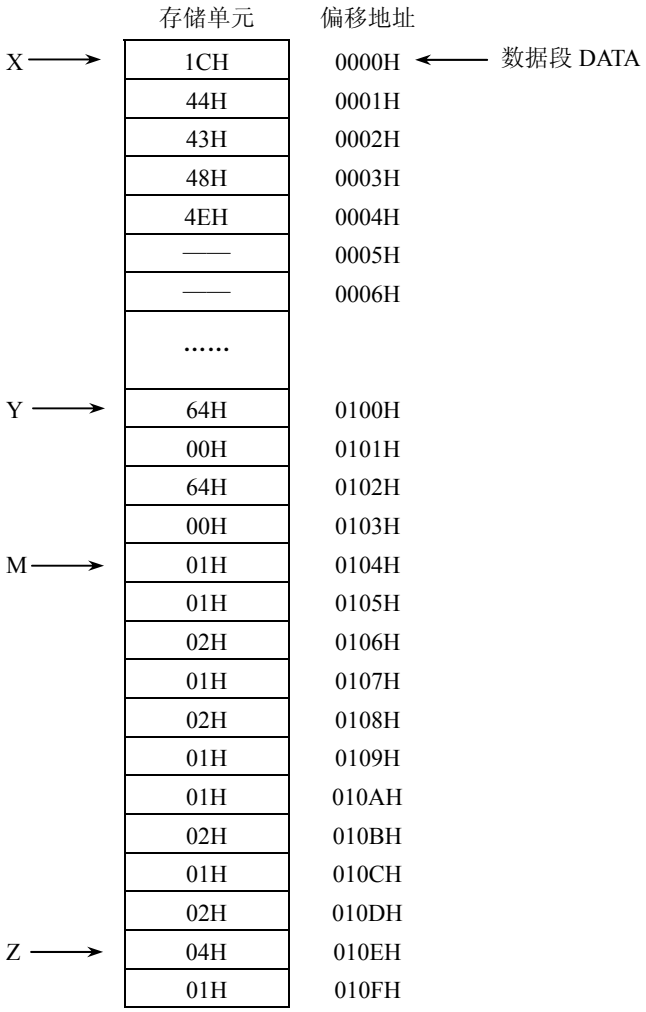


图 5-1　例 5-3 中内存数据的存放

表 5-3　变量或标号的属性值

	属性名	属性值
变量	DB	1
	DW	2
	DD	4
	DQ	8
	DT	10
标号	NEAR	-1
	FAR	-2

3. 符号定义伪指令

符号定义伪指令用来定义符号常量，系统不会给符号常量分配内存空间。其指令有 EQU、=。

指令格式：符号 EQU 表达式

符号 = 表达式 ;左边符号的值为右边表达式的值

两者的区别是，用“=”定义的符号常量可以被重新定义，而用 EQU 定义的符号常量不能被重新定义。

例 5-5

```
VAR1    EQU   10H
MOV     AL,VAR1                    ;等价于 MOV   AL,10H
VAR2    EQU   Z
MOV     AX,VAR2                    ;等价于 MOV   AX,Z
VAR3    EQU   VAR1*3+10
MOV     AL,VAR3                    ;等价于 MOV   AL,3AH
VAR4    EQU   [BX+SI+100]
MOV     AL,VAR4                    ;等价于 MOV   AL,[BX+SI+100]
VAR5    EQU   ADD
VAR5    AX,BX                      ;等价于 ADD   AX,BX
VAR6    EQU   01H
VAR6    EQU   02H                  ;（×）前面已经定义了符号常量 VAR6，不能再重复定义 VAR6
MAX   = 100
MAX   = MAX + 100                  ;（√）前面符号常量 MAX 的值为 100，现在其值被修改为 200
;说明：可以用 PURGE 指令解除对符号常量的定义，之后就可以对该符号重新定义了
;其格式为：PURGE 符号 1,符号 2,…,符号 N
          MIN     EQU   01H
          PURGE   MIN
          MIN     EQU   02H        ;（√）前面已经定义了符号常量 MIN，PURGE 解除了对 MIN
                                   ;的定义，所以可以重新定义 MIN
```

4. LABEL 伪指令

LABEL 伪指令为其后定义的变量或标号定义一个不同类型的别名。其格式为：

变量或标号 LABEL 类型

其中，类型为 BYTE、WORD、DWORD、NEAR 或 FAR。

例 5-6

```
VAR     LABEL   WORD
X       DB   'AB'                  ;变量 VAR、X 指向内存中的同一单元，但两者类型分别为
                                   ;字类型、字节类型
MOV     AX,VAR                     ;等价于 MOV   AX,4241H
MOV     AL,X                       ;等价于 MOV   AL,41H
```

例 5-6 还可以改成：

```
VAR     EQU   THIS   WORD
X       DB   'AB'                  ;变量 VAR、X 指向内存中的同一单元，但两者类型分别
                                   ;为字类型、字节类型
MOV     AX, VAR                    ;等价于 MOV   AX,4241H
MOV     AL,X                       ;等价于 MOV   AL,41H
;说明：THIS 为其后定义的变量或标号定义一个不同类型的别名。其格式为：变量名 EQU THIS 类型
;其中，类型为 BYTE、WORD、DWORD、NEAR 或 FAR
```

5. 段定义伪指令

汇编语言源程序由若干个段组成，段定义伪指令（SEGMENT/ENDS）用来定义一个段，

要求给出段名，由 SEGMENT 指定段的开始，ENDS 指定段的结束。其格式为：

```
段名　SEGMENT [定位类型] [组合类型] ['类别']
…　　;语句序列
段名 ENDS
```

说明：

（1）SEGMENT 和 ENDS 必须成对出现。

（2）段名由用户自己命名，必须符合标识符命名规则，前后段名必须保持一致。每个段的段名即为该段的段基址。

（3）定位类型用来说明对段起始地址的要求，可以省略。定位类型有以下 4 种：

- BYTE：段的起始地址可在任意字节边界上，即段起始地址是任意的。
- WORD：要求段的起始地址在任意字边界上，即段起始地址最低位为 0，亦即段起始地址必须为偶地址。
- PARA：要求段的起始地址在节（16 字节）的边界上，即段起始地址低 4 位全部为 0，如 XXXX0H。缺省定位类型时，默认为 PARA 类型。
- PAGE：要求段的起始地址在页（256 字节）边界上，即段起始地址低 8 位全部为 0，如 XXX00H。

（4）组合类型用来说明同类别名的段的连接方式，可以省略。定位类型有以下 6 种：

- NONE：不与其他段连接。缺省组合类型时，默认为 NONE 类型。
- PUBLIC：将不同程序模块中同名同类型的段按顺序连接成一个共同的段装入内存。
- STACK：指定该段为堆栈段，并将不同程序模块中的堆栈段按顺序连接成一个堆栈段，即所有程序模块共用一个堆栈段。
- COMMON：将不同程序模块中同名同类型的段都从同一个地址开始装入，即以覆盖方式连接，各个逻辑段将发生重叠，段长度为最大段的长度。
- AT 表达式：按照表达式的值指定的段基址将段装入内存。
- MEMORY：多个逻辑段连接时，连接程序将把本段连接在其他所有段之上。若多个段均为 MEMORY 类型时，则将第一个 MEMORY 段置于所有段之上，其他 MEMORY 段当成 COMMON 类型来处理。

（5）类别名必须用‘’引起来，用来说明该段类别名，在连接时将同类别名的段按照组合类型进行组合。类别名由用户自定义，长度不超过 40 个字符。

例 5-7

```
CODE　SEGMENT 'CODE'
…
CODE　ENDS　　　;定义一个段，段名为 CODE，类别名为 CODE
STACKSEG　SEGMENT STACK
…
STACKSEG　ENDS　　;定义一个堆栈段，段名为 STACKSEG，组合类型为 STACK
DATA1　SEGMENT WORD PUBLIC 'CONST'
…
DATA1　ENDS　　;定义一个段，段名为 DATA1，定位类型为 WORD，组合类型为 PUBLIC
　　;类别名为 CONST
CODESEG　SEGMENT PARA PUBLIC 'CODE'
…
CODESEG　ENDS　　;定义一个段，段名为 CODESEG，定位类型为 PARA，组合类型为 PUBLIC
　　;类别名为 CODE
```

6. 段分配伪指令

段分配伪指令用来说明当前哪些逻辑段为代码段、哪些为数据段、哪些为堆栈段、哪些为附加段。其格式为：

ASSUME 段寄存器: 段名[, 段寄存器: 段名, …]

说明：

（1）ASSUME 伪指令只能设置在代码段内，放在段定义语句之后。

（2）ASSUME 伪指令只是建立了逻辑段与段寄存器之间的关系，并没有为段寄存器赋值。对于代码段和堆栈段，由连接程序来设置 CS、IP、SS、SP 的值；而数据段和附加段则需要由用户在程序中对 DS、ES 赋值。

（3）每个段的段名即为该段的段基址，它是一个16位的立即数，因此不能直接将它送给段寄存器，通常先将段名送给一个通用寄存器，然后将该通用寄存器的值再送给段寄存器，来对DS、ES赋值。

例 5-8

```
DATA1    SEGMENT                        ;定义一个段，段名为 DATA1
         X   DB   100
DATA1    ENDS
EXTRA    SEGMENT                        ;定义一个段，段名为 EXTRA
         STR   DW   10   DUP(?)
EXTRA   ENDS
STACKSEG   SEGMENT   STACK              ;定义一个堆栈段，段名为 STACKSEG
         BUF   DW   50   DUP(?)
STACKSEG   ENDS
CODE   SEGMENT                          ;定义一个段，段名为 CODE
ASSUME CS: CODE,DS: DATA1,ES: EXTRA,SS: STACKSEG
;指定 CODE 为代码段，DATA1 为数据段，EXTRA 为附加段，STACKSEG 为堆栈段
START:   MOV AX, DATA1
         MOV DS, AX                     ;将数据段段基址送入 DS
         MOV AX, EXTRA
         MOV ES, AX                     ;将附加段段基址送入 ES
         …
CODE   ENDS
END START
```

7. 过程定义伪指令

对于程序中经常用到的具有独立功能的语句组，可将它定义成一个子过程，通过 CALL 来调用执行，可以简化主程序，实现模块化程序设计，提高编程效率。

（1）过程定义的格式：

```
过程名    PROC    [属性]
     …  ;语句序列
     RET
过程名    ENDP
```

说明：

① 过程名由用户自己命名，但必须符合标识符命名规则，前后过程名必须保持一致。过程名代表过程的入口地址。

② PROC 指定过程的开始，ENDP 指定过程的结束，PROC 和 ENDP 必须成对出现。

③ 属性：过程属性有 NEAR（段内近调用）、FAR（段间远调用）两种，若缺省则为 NEAR。

NEAR 属性的过程只能被本代码段内的其他程序调用；FAR 属性的过程既可以被本代码段内的程序调用，又可以被其他代码段内的程序调用。

④ 过程必须以 RET 结尾，以便返回调用它的程序。

⑤ 子过程应安排在代码段的主程序之外，最好放在主程序执行终止后的位置（返回 DOS 后、汇编结束 END 伪指令前），也可以放在主程序开始执行之前的位置。

（2）过程调用格式：CALL　过程名

例 5-9

```
CODE   SEGMENT                          ;定义代码段 CODE
ASSUME CS:CODE
BEGIN:
          …
          CALL   SUB                    ;调用过程 SUB
          …
SUB   PROC   NEAR                       ;定义过程 SUB，其属性为 NEAR
          …
          RET                           ;返回主程序
SUB    ENDP                             ;过程 SUB 定义结束
CODE   ENDS                             ;代码段 CODE 定义结束
END   BEGIN
```

主程序与子过程位于同一个代码段时称为段内近调用，主程序执行到 CALL 指令时，只需将下一条指令的偏移地址 IP 压入堆栈，然后转到以 SUB 为偏移地址（只需修改 IP 的值）的过程去执行，过程执行到 RET 指令时，从堆栈弹出一个字送入 IP，这样就返回到主程序中，去执行主程序 CALL 后的指令。

例 5-10

```
CODE1   SEGMENT
          …
          CALL   SUB
          …
SUB    PROC FAR
          …
          RET
SUB    ENDP
CODE1 ENDS
          …
CODE2   SEGMENT
          …
          CALL   SUB
          …
CODE2   ENDS
          …
```

主程序与子过程不在同一个代码段时称为段间远调用，主程序执行到 CALL 指令时，将下一条指令的段基址 CS 和偏移地址 IP 都压入堆栈（CS 先入栈），然后转到以 SUB 为入口地址（既要修改 CS 的值，又要修改 IP 的值）的过程去执行，过程执行到 RET 指令时，从堆栈弹出两个字分别送入 IP、CS，这样就返回到主程序中，去执行主程序 CALL 后的指令。

例 5-10 中，在代码段 CODE2 中，调用代码段 CODE1 里定义的子过程 SUB 就属于这种情况；但是，在代码段 CODE1 中，调用子过程 SUB 时，虽然属于段内近调用，但 SUB 属性

为 FAR，仍然要当作段间远调用处理，即调用时，CS、IP 都要入栈，返回时，也要分别弹出 IP、CS 的值，才能正确地返回主程序。

8. 全局标识符伪指令

开发较复杂的大型应用程序时，通常把程序分解成多个功能独立的模块，分别编写子程序来实现各个模块的功能，对各个子程序单独进行汇编产生相应的目标模块（OBJ 文件），最后再用连接程序把它们连接成一个完整的可执行程序，称之为模块化程序设计方法。

采用模块化程序设计，各模块之间会存在着数据的交流，即在一个模块中需要引用在另一个模块中定义的变量、标号或过程。

模块中的标识符有两种：①仅供本模块使用的标识符，称为局部标识符；②既可供本模块使用，又可供另外的模块使用的标识符，称为全局标识符。

（1）全局标识符定义伪指令。要想让其他模块能调用本模块中的标识符，就需要在本模块中将该标识符定义为全局标识符，其格式为：

PUBLIC　标识符 1,标识符 2,…

（2）全局标识符声明伪指令。要想在本模块中调用其他模块里的全局标识符，需要用 EXTRN 进行声明，其格式为：

EXTRN　标识符 1:类型,标识符 2:类型,…

其中，类型可为 BYTE、WORD、DWORD、NEAR 或 FAR。

例 5-11

```
;模块 1，文件 1.ASM
EXTRN    SUB2:FAR                          ;声明全局远过程 SUB2
PUBLIC   DATA1,RESULT                      ;定义全局变量 DATA1、RESULT
DSEG     SEGMENT
         DATA1    DB   3 DUP(1)
         RESULT   DB   ?
DSEG     ENDS
CODE     SEGMENT
         ASSUME   CS:CODE,DS:DSEG
START:   MOV   AX, DSEG
         MOV   DS, AX                      ;初始化 DS
         CALL  SUB2                        ;调用远过程 SUB2
         ADD   RESULT, '0'                 ;将结果转换成字符显示
         MOV   DL,RESULT
         MOV   AH,2
         INT   21H
         MOV   AH,4CH                      ;程序结束，返回 DOS
         INT   21H
CODE     ENDS
END START
;模块 2，文件 2.ASM
EXTRN   DATA1:BYTE, RESULT:BYTE            ;声明全局变量字节变量 DATA1、RESULT
PUBLIC   SUB2                              ;定义全局远过程 SUB2
DSEG2    SEGMENT
         DATA2   DB 3 DUP(2)
DSEG2    ENDS
CODE2    SEGMENT
         ASSUME   CS:CODE2,ES:DSEG2
SUB2   PROC   FAR                          ;定义过程 SUB2
```

```
        MOV   AX, DSEG2
        MOV   ES, AX                    ;初始化 ES
        MOV   SI, 0
        MOV   CX, 3                     ;循环次数
        MOV   BL, 0                     ;存放累加和，初始化值为 0
LOOP1:  MOV   AH, DATA1[SI]             ;循环累加，结果存入 RESULT
        MOV   AL, DATA2[SI]
        ADD   AH, AL
        ADD   BL, AH
        INC   SI
        LOOP  LOOP1
        MOV   RESULT,BL
        RET
SUB2    ENDP
CODE2   ENDS
END
```

程序功能为：ESULT=(DATA1[0]+DATA2[0])+(DATA1[1]+DATA2[1])+(DATA1[2]+DATA2[2])。

编译，连接过程如下：

```
E:\MASM5>MASM 1.ASM                     ;编译模块 1
MICROSOFT (R) MACRO ASSEMBLER VERSION 5.00
COPYRIGHT (C) MICROSOFT CORP 1981-1985, 1987.    ALL RIGHTS RESERVED.
OBJECT FILENAME [1.OBJ]:
SOURCE LISTING    [NUL.LST]:
CROSS-REFERENCE [NUL.CRF]:
50908 + 415700 BYTES SYMBOL SPACE FREE
0 WARNING ERRORS
0 SEVERE    ERRORS
E:\MASM5>MASM 2.ASM                     ;编译模块 2
MICROSOFT (R) MACRO ASSEMBLER VERSION 5.00
COPYRIGHT (C) MICROSOFT CORP 1981-1985, 1987.    ALL RIGHTS RESERVED.
OBJECT FILENAME [2.OBJ]:
SOURCE LISTING    [NUL.LST]:
CROSS-REFERENCE [NUL.CRF]:
50794 + 415814 BYTES SYMBOL SPACE FREE
0 WARNING ERRORS
0 SEVERE    ERRORS
E:\MASM5>LINK 1 2                       ;链接模块 1、模块 2，模块 1 为主模块
MICROSOFT (R) OVERLAY LINKER    VERSION 3.60
COPYRIGHT (C) MICROSOFT CORP 1983-1987.    ALL RIGHTS RESERVED.
RUN FILE [1.EXE]:
LIST FILE [NUL.MAP]:
LIBRARIES [.LIB]:
LINK : WARNING L4021: NO STACK SEGMENT
E:\MASM5>1                              ;运行
9                                       ;输出结果
```

9. 程序结束伪指令

格式：END [标号]

功能：表示程序的结束，汇编程序遇到 END 时结束汇编，其后的标号为程序执行的起始

地址。

5.2.4 结构与记录

1. 结构

当程序中的数据是由多个数据成员组成时，如学生信息表中的学生数据，包含学号、姓名、性别、年龄等多个成员数据，若用前面的方式来定义多个学生数据就比较麻烦。8086 宏汇编提供了结构（STRUCTURES）来实现对这种数据的处理，结构就是相互关联的一组数据的某种组合形式。使用结构数据前，需要先定义结构类型，再用定义好的结构类型去定义结构变量，并完成结构变量的初始化。

（1）结构类型的定义。

格式：结构类型名　STRUC
　　成员数据变量序列
　　结构类型名　ENDS

对于上述学生信息数据，可定义一个结构类型 STUDENT。

例 5-12

```
STUDENT   STRUC
    NO        DB   ?
    NAMEX     DB   'JACK'
    SEX       DB   'M'
    AGE       DB   ?
STUDENT ENDS
```

其结构类型名为 STUDENT，它包含有 4 个成员变量（又叫结构字段名）：NO、NAMEX、SEX、AGE。定义结构类型时，结构字段变量可以指定其初始值，也可以用“？”代替。

注意：这里只是定义了一个结构类型，系统并不为它分配存储单元。

（2）结构类型变量的定义与初始化。

格式：结构变量名　结构类型名　<字段值表>

说明：

①字段值表用来给结构变量中各结构字段赋初值，其类型、顺序应与结构类型定义中的字段保持一致，各个字段初始化值之间用逗号间隔。

②给结构变量中各结构字段赋初值时，有一定的限制：在结构类型定义中只具有一项数据的结构字段，可以通过字段值表来修改代替初始定义时的值；用 DUP 定义的字段或一个字段后有多个数据的字段，则不能修改其定义时的值，即不能通过字段值表来修改这些字段的值。

例 5-13

```
DATA   STRUC
    X     DB   10H              ;简单元素，可以修改
    Y     DB   1,2              ;多重元素，不能修改
    Z     DW   ?                ;简单元素，可以修改
    M     DB   ' ZXC'           ;可用同长度的字符串修改
    N     DW   20 DUP (?)       ;多重元素，不能修改
DATA ENDS
```

③若不需要修改某些字段的值（即仍采用其定义时的值），则在字段值表中的对应位置仅写一个逗号即可。

④若所有字段的值均采用其定义时的值，不需要修改，则仅写一对尖括号即可。

例 5-14

```
DATA   SEGMENT
   STUDENT    STRUC
      NO   DB   ?
      NAMEX   DB   'JACK'
      SEX   DB   'M'
      AGE   DB   ?
   STUDENT   ENDS                              ;定义结构类型
   X1   STUDENT <1,,,21>
   X2   STUDENT <2,'ANDY',,22>
   X3   STUDENT <3, 'ROSE', 'F',20>
   X4   STUDENT <,,,>                          ;定义结构类型变量 X1、X2、X3、X4
DATA   ENDS
```

（3）结构类型变量的引用。

在程序中可以直接引用结构类型变量，也可以引用结构类型变量中的某一字段，其格式为：结构变量名.字段名。

例如，将前面定义的 STUDENT 结构类型变量 X1 中的 AGE 字段值送到 AL 中。

```
MOV   AL,X1.AGE
```

也可以写成：

```
MOV   BX,OFFSET   X1
MOV   AL,[BX].AGE
```

例 5-15　将四个学生的学号、姓名、性别、年龄用结构的形式存入内存，并编程求所有男生年龄之和。

```
DATA   SEGMENT
  STUDENT   STRUC                   ;定义结构类型
          NO       DB   ?
          NAMEX DB   'JACK'
          SEX      DB   'M'
          AGE      DB   ?
  STUDENT   ENDS
  X1 STUDENT <1, , ,21>             ;定义结构类型变量 X1、X2、X3、X4
  X2 STUDENT <2,'ANDY', 'F',22>
  X3 STUDENT <3,'ROSE', ,20>
  X4 STUDENT <4,'JOHN', ,23>
DATA ENDS
CODE SEGMENT
  ASSUME CS:CODE,DS:DATA
START:    MOV   AX, DATA
          MOV   DS, AX
          MOV   AX, 0               ;AX 清零
          MOV   CX, 4               ;设置循环次数
          MOV   BX, OFFSET X        ;BX 指向  X1 的第一字节
LP1:      CMP   [BX].SEX, 'F'       ;取出结构变量的 SEX 字段，判断是否为女生
          JZ      LP2               ;若为女生，则转到 LP2
          MOV   DH, [BX].AGE        ;若不为女生，则取出结构变量的 AGE 字段到 DH
          ADD   AL, DH              ;将 DH 的值累加到 AL 中
LP2:      ADD   BX, 7               ;让 BX 指向下一个结构变量的第一字节
          LOOP  LP1                 ;循环
```

```
        MOV   CL, 10
        DIV   CL                ;所有男生年龄之和已存放在 AX 中，AX/10 得到的商（即
                                ;其十位上的数）在 AL 中，余数（即其个位上的数）
                                ;在 AH 中
        MOV   CL, AH            ;将 AH 中的余数保存到 CL 中
        ADD   AL, '0'           ;将 AL 中的商转换成其对应的数字字符，并显示
        MOV   DL, AL
        MOV   AH, 2
        INT   21H
        ADD   CL, '0'           ;将保存到 CL 中的余数转换成其对应的数字字符，并显示
        MOV   DL, CL
        MOV   AH, 2
        INT   21H
        MOV   AH, 4CH           ;程序结束，返回 DOS
        INT   21H
CODE ENDS
END START
```

2. 记录

一般来说，访问存储器的最小单位是字节，但在实际应用中，某些数据只需要用一个二进制位来表示，如何按位访问这些数据呢？8086 宏汇编提供了记录（RECORD）来实现对这类数据的处理。使用记录前，也需要先定义记录类型，再用定义好的记录类型去定义记录变量，并完成记录变量的初始化。

（1）记录类型的定义。

格式：记录类型名 RECORD <字段名>:宽度[=表达式][,<字段名>:宽度[=表达式]…]

说明：

①记录类型可由多个字段（至少要有一个字段）组成，每个字段之间要用逗号分开。

②字段的属性包括字段名、宽度和初值。宽度表示该字段所占的二进制位数，它必须是一个常数，且其取值范围为 1～16，并且各字段的宽度之和应在 1～16 之间。用“表达式”来给相应字段赋初值，且表达式的值不能超过该字段的表示范围能容纳下的正整数，若缺省初值，则默认该字段的初值为 0。

③如果记录的总宽度小于等于 8 位，系统只为该记录分配一个字节空间；如果记录的总宽度大于 8 位且小于等于 16 位，则系统为该记录分配两个字节空间。

例如，定义一个表示学生某门功课成绩的记录。

```
SCORE   RECORD   NO:3,SEX:1,COURSE:2,GRADE:2
```

记录类型名为 SCORE，它包含有 4 个字段：NO（占 3 位）、SEX（占 1 位）、COURSE（占 2 位）、GRADE（占 2 位）。

（2）记录类型变量的定义与初始化。

格式：记录变量名　记录类型名<字段值表>

（3）记录变量中字段值的存放。

记录中各字段靠右对齐到字节或字的最低有效位置，即记录的最后一个字段排在所分配空间的最低位，然后对记录中的字段依次“从右向左”分配二进制位，左边没有分完的二进制位补 0。

例 5-16

```
DATA   SEGMENT
```

```
    SCORE   RECORD   NO:3,SEX:1,COURSE:2,GRADE:2          ;定义记录类型
    Y1   SCORE<111B,1B,11B,11B>                          ;定义记录类型变量 Y1
DATA ENDS
```

本例中数据段中的数据在内存中的存放如图 5-2 所示。

	NO			SEX	COURSE		GRADE	
Y1	1	1	1	1	1	1	1	1

图 5-2　例 5-16 中内存数据的存放

（4）记录运算符。

1）WIDTH：求记录或记录字段所占的位数。

格式：WIDTH 记录类型名或记录字段名

例如，对前面已定义的记录类型SCORE：

```
WIDTH   SCORE                    ;表达式的值为 8
WIDTH   NO                       ;表达式的值为 3
```

2）MASK：返回一个记录值，将指定字段各位置为 1，其他字段各位全部置为 0。

格式：MASK 记录字段名

例如，对前面已定义的记录类型SCORE：

```
MASK   NO                        ;表达式的值为 11100000B
MASK   SEX                       ;表达式的值为 00010000B
MASK   COURSE                    ;表达式的值为 00001100B
MASK   GRADE                     ;表达式的值为 00000011B
```

3）记录字段名。记录字段名可以作为一个操作数在程序中单独出现，它表示该字段最低位距该记录的最低位有多少位。

例如，对前面已定义的记录类型 SCORE：

```
MOV   AL,NO                      ;等价于 MOV   AL,5
MOV   AH,COURSE                  ;等价于 MOV   AH,2
MOV   BH,GRADE                   ;等价于 MOV   BH,0
```

例 5-17　将四个学生的学号、性别、成绩用记录的形式存入内存，并编程求所有女生成绩之和（以十六进制形式显示）。

```
DATA   SEGMENT
          SCORE   RECORD   NO:3,SEX:1,GRADE:4      ;定义记录类型，SEX 字段为 1 时表示男生
          Y1   SCORE<00B,0B,1000B>                 ;定义四个记录类型变量 Y1、Y2、Y3、Y4
          Y2   SCORE<010B,0B,1011B>
          Y3   SCORE<101B,0B,1101B>
          Y4   SCORE<111B,0B,0011B>
          DISP   DB 30H,31H,32H,33H,34H,35H,36H,37H,38H,39H,41H,42H,43H,44H,45H,46H
DATA   ENDS
CODE   SEGMENT
ASSUME CS:CODE,DS:DATA
START:    MOV   AX, DATA
          MOV   DS,AX
          MOV   CX,4                    ;设置循环次数
          MOV   DH,0                    ;求和初值 DH 置零
          MOV   BX,OFFSET Y1            ;BX 指向第一个记录变量
L1:       MOV   AL,[BX]                 ;取出记录变量的值到 AL 中
          TEST   AL,MASK SEX            ;测试记录变量 SEX 字段的值
```

```
        JNZ   L2                    ;若为男生，则不累加，转去处理下一条记录
        AND   AL,MASK GRADE         ;若为女生，则取出记录变量 GRADE 字段的值
        ADD   DH,AL                 ;将女生成绩累加到 DH 中
L2:     INC   BX                    ;BX 指向下一个记录
        LOOP  L1                    ;循环
        LEA   SI,DISP
        MOV   BH,0                  ;将和的高四位以十六进制形式显示
        MOV   BL,DH
        AND   BL,0F0H
        MOV   CL,4
        SHR   BL,CL
        MOV   DL,[BX][SI]
        MOV   AH,02H
        INT   21H
        MOV   BH,0                  ;将和的低四位以十六进制形式显示
        MOV   BL,DH
        AND   BL,0FH
        MOV   DL,[BX][SI]
        MOV   AH,02H
        INT   21H
        MOV   DL,'H'                ;显示字符 H
        MOV   AH,02H
        INT   21H
        MOV   AH,4CH                ;程序结束，返回 DOS
        INT   21H
CODE    ENDS
END START
```

5.2.5 宏指令

前面讲到，对于程序中需要重复多次用到的具有独立功能的语句组，可将它定义成一个子过程，通过 CALL 来调用执行。实际上，也可以把它们定义成一个宏指令，在程序中反复调用，以达到简化主程序、提高编程效率的目的。

1. 宏与子程序的区别

（1）宏与子程序的相同点：用一条指令来代替一段程序，子程序和宏指令定义好之后都可以被多次调用，可以起到简化源程序的作用。

（2）宏与子程序的不同点：

①从代码开销的角度来讲，子程序优于宏指令。编译宏指令时，需要将每一个宏调用指令展开，有多少次调用，就要在目标程序中插入多少次宏体程序段，因而调用次数越多，占用内存空间就越大；编译子程序时只占用一个程序段（即使是调用多次），因而汇编后产生的目标程序占用内存空间少。

②从时间开销的角度来讲，宏指令优于子程序。每次调用子程序时都要保护/恢复现场和断点，额外增加了时间开销；而宏指令在执行时不存在保护/恢复现场和断点的问题，执行的时间短，速度快。

一般来说，当要重复执行的程序不长，重复次数又多时，速度是主要问题，通常用宏指令；而要重复执行的程序较长，重复次数又不是太多时，额外操作所附加的时间就不明显了，

节省内存空间应视为主要问题，通常采用子程序结构。

2. 宏定义

格式：宏指令名　MACRO　[形式参数1][,形式参数2……]
　　宏体　;语句序列
　　ENDM

说明：

（1）宏指令名由用户自己命名，但必须符合标识符命名规则。

（2）MACRO 指定宏定义的开始，ENDM 指定宏定义的结束，它们必须成对出现。

（3）宏体为实现宏指令功能的语句序列。

（4）形式参数列表用来给出宏定义中所用到的参数，形式参数可有一个或多个，也可以没有，有多个形式参数时，参数之间以逗号隔开。

（5）宏定义不必在任何逻辑段中，通常写在源程序的开头。

（6）宏定义中的注释语句以“;;”开头。

例 5-18

```
ADDCAB   MACRO                ;定义宏指令 ADDCAB（没有参数），功能：CX=AX+BX
         ADD   AX,BX
         MOV   CX,AX
ENDM
PUTCHAR   MACRO   CHAR
                              ;定义宏指令 PUTCHAR，参数为 CHAR，功能：输出参数
                              ;CHAR 对应的字符
         PUSH AX
         PUSH DX              ;保护寄存器 AX 和 DX 的值
         MOV DL,CHAR
         MOV AH,2
         INT 21H
         POP DX
         POP AX               ;恢复寄存器 AX 和 DX 的值
ENDM
```

3. 宏调用

格式：宏指令名　[实际参数1][,实际参数2…]

在程序中使用已经定义过的宏指令，称为宏调用。如果宏指令有形式参数，在宏调用时，必须在宏指令名后面写上实际参数，并与形式参数一一对应，有多个实际参数时，参数之间以逗号隔开。

具有宏调用的源程序被汇编时，汇编程序将用宏定义时设计的宏体去代替宏指令名，并且用实际参数一一代替形式参数，称为宏展开。汇编程序在所展开的指令前加上“1”号以示区别。

例 5-19　用宏指令定义两个字操作数相除，第一个操作数为被除数，第二个操作数为除数，并将商存入第三个操作数，余数存入第四个操作数。

```
M_DIVIDE   MACRO   OPR1,OPR2,OPR3,OPR4
                                   ;定义宏 M_DIVIDE，其功能为：OPR3=OPR1/OPR2
                                   ;余数存放在 OPR4 中
        PUSH   DX
        PUSH   AX
        MOV    AX, OPR1
```

```
            CWD
            DIV     OPR2
            MOV     OPR3, AX
            MOV     OPR4,DX
            POP     AX
            POP     DX
ENDM
DATA    SEGMENT
            A1      DW      2424H
            A2      DW      1212H
            A3      DW      ?
            A4      DW      ?
DATA    ENDS
CODE    SEGMENT
            ASSUME CS:CODE,DS:DATA
START:      MOV     AX,DATA
            MOV     DS,AX
            M_DIVIDE    A1,A2,A3,A4             ;调用宏 M_DIVIDE
            MOV     AH,4CH
            INT     21H
CODE    ENDS
END     START
```

经宏展开后：

```
START:      MOV     AX,DATA
            MOV     DS,AX
1           PUSH    DX
1           PUSH    AX
1           MOV     AX, A1
1           CWD
1           DIV     A2
1           MOV     A3, AX
1           MOV     A4,DX
1           POP     AX
1           POP     DX
            MOV     AH,4CH
            INT     21H
```

补充说明：

（1）宏定义中的参数还可以是操作码。

宏定义：

```
OP    MACRO    OPR1,OPR2,OPR3
      MOV    AX, OPR1
      OPR2    AX，OPR3
ENDM
```

宏调用：

```
OP    X,ADD, Y                          ;假设 X、Y 为已经在数据段定义好的两个字变量
```

宏展开：

```
1     MOV    AX,X
1     ADD    AX,Y
```

（2）在宏定义中还可以使用分隔符&，展开时把&前后的两个符号连接起来，形成操作

码、操作数或字符串。

宏定义：

```
SHIFT   MARCO   OPR1,OPR2,OPR3
        ;定义宏指令 SHIFT，用来将 OPR1 逻辑移位 OPR2 次，OPR3 指定是左移还是右移
        MOV   CL,OPR2
        SH&OPR3   OPR1,CL
ENDM
```

宏调用：

```
SHIFT   AL,4,L
```

宏展开：

```
1   MOV   CL,4
1   SHL    AL,CL
```

（3）在宏定义中可以调用之前已经定义过的宏。

例 5-20　定义一个宏指令求两个数相除的商。

```
PUSHDA      MACRO                                   ;定义宏指令 PUSHDA
        PUSH   DX
        PUSH   AX
ENDM
POPDA       MACRO                                   ;定义宏指令 POPDA
        POP    AX
        POP    DX
ENDM
M_DIVIDE   MACRO   OPR1,OPR2,OPR3,OPR4             ;定义宏指令 DIVIDE
        PUSHDA                                      ;调用宏指令 PUSHDA
        MOV   AX, OPR1
        CWD
        DIV    OPR2
        MOV   OPR3,AX
        MOV   OPR4,DX
        POPDA                                       ;调用宏指令 POPDA
ENDM
```

宏调用：

```
M_DIVIDE   A1,A2,A3,A4              ;假设 A1、A2、A3、A4 为已经在数据段定义好的字变量
```

宏展开：

```
1    PUSH     DX
1    PUSH     AX
1    MOV      AX, A1
1    CWD
1    DIV      A2
1    MOV      A3, AX
1    MOV      A4,DX
1    POP      AX
1    POP      DX
```

4. 局部标号伪指令 LOCAL

对于使用了标号的宏，若多次调用，展开时将多次出现相同的标号，这在汇编语言程序中是不允许的，汇编时将报错。8086 宏汇编提供了局部标号伪指令 LOCAL 来解决这一问题。

格式：LOCAL　　标号 1[,标号 2…]

说明：

（1）标号 1、标号 2……为宏定义中的标号。

（2）LOCAL 伪指令只能用在宏定义体内，还必须是 MACRO 伪操作后的第一个语句，且在 MACRO 与 LOCAL 之间不能出现注释和分号标志。

（3）对 LOCAL 后的标号，汇编程序将用“??0000”……“??FFFF”来依次取代宏展开时的标号。这样，在宏展开后，程序中标号都是唯一的。

例 5-21

宏定义：

```
ABS   MACRO OPRW                    ;定义宏指令 ABS 求字数据 OPRW 的绝对值
      LOCAL   L
      CMP     OPRW,0
      JGE     L
      NEG     OPRW
L:    MOV     AX,OPRW
ENDM
```

宏调用：

```
ABS   X
ABS   Y                             ;假设 X、Y 为已经在数据段定义好的两个字变量
```

宏展开：

```
1             CMP     X,0
1             JGE     ??0000
1             NEG     X
1   ??0000:   MOV     AX,X
1             CMP     Y,0
1             JGE     ??0001
1             NEG     Y
1   ??0001:   MOV     AX,Y
```

5. 文件包含伪指令 INCLUDE

当多个程序要调用同一个宏时，可以把这些宏组合起来，建立一个独立的文件，称为宏库，其扩展名是 MAC 或 INC。当需要调用宏库中的宏时，只需要在该程序的开始用 INCLUDE 伪指令把该宏库文件包含进来即可。

格式：INCLUDE　宏库文件名

汇编时，将用 INCLUDE 伪指令指定的文件的内容插入到该伪指令所在的位置，与源程序一起进行汇编，所以要注意宏库文件中的标识符不能与源程序中的标识符重名。

例如：

```
INCLUDE   OUTPUT.MAC
INCLUDE   D:\MASM5\INPUT.MAC
```

6. 重复汇编伪指令

（1）REPT。

格式：REPEAT 重复次数

　　重复体

　　ENDM

说明：使汇编程序按照指定次数对重复体进行重复汇编。

例 5-22

```
CHAR = 0
```

```
REPEAT 10
      DB CHAR
      CHAR = CHAR +1
ENDM
```

展开：

```
1     DB   CHAR                    ;等价于 DB   0
1     CHAR = CHAR +1
1     DB   CHAR                    ;等价于 DB   1
1     CHAR = CHAR +1
      …
1     DB   CHAR                    ;等价于 DB   9
1     CHAR = CHAR +1
```

（2）IRP。

格式：IRP　形式参数,<实际参数表>

　　重复体

　　ENDM

说明：重复汇编时，每作一次汇编就依次将实参表中的一个实参取代重复体中的形参。

例 5-23

```
IRR   REG,<AX,BX,CX,DX>
      POP   REG
ENDM
```

展开：

```
1     POP   AX
1     POP   BX
1     POP   CX
1     POP   DX
```

（3）IRPC。

格式：IRPC　形参,字符串

　　重复体

　　ENDM

说明：重复汇编时，每作一次汇编就依次用字符串中的一个字符取代重复体中的形参。

例如：

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

展开：

```
1     PUSH   AX
1     PUSH   BX
1     PUSH   CX
1     PUSH   DX
```

7. 条件汇编伪指令

利用条件汇编伪指令可以有选择地汇编某段源程序。

格式：

```
IFXX   表达式              ; 定义条件
      语句组 1             ; 满足条件时编译语句组 1
[ ELSE                     ; ELSE 部分也可以省略
      语句组 2 ]           ; 不满足条件时编译语句组 2
```

ENDIF

常用条件汇编伪指令如表 5-4 所示。

表 5-4 常用条件汇编伪指令

条件汇编伪指令	成立的条件
IF 表达式	表达式的值不为 0
IFE 表达式	表达式的值为 0
IFDEF 符号	符号已定义
IFIDN <串 1>,<串 2>	<串 1>=<串 2>（区分大小写）
IFIDNI <串 1>,<串 2>	<串 1>=<串 2>（不区分大小写）

例 5-24 利用条件汇编定义宏指令，完成多种 DOS 系统功能调用。

```
DOSYS   MACRO   N,BUF
        IFE     N
        EXITM
ENDIF
IFDEF   BUF
        LEA     DX,BUF
        MOV     AH,N
        INT     21H
ELSE
        MOV     AH,N
        INT     21H
        ENDIF
ENDM
DATA    SEGMENT
        MSG     DB      'INPUT   STRING:$'
        BUF     DB      81,0,80 DUP(0)
DATA    ENDS
STACK   SEGMENT  STACK
        DB      200   DUP(0)
STACK   ENDS
CODE    SEGMENT
        ASSUME  DS:DATA,CS:CODE,SS:STACK
BEGIN:  MOV     AX,DATA
        MOV     DS,AX
        DOSYS   9,MSG
        DOSYS   10,BUF
        DOSYS   4CH
CODE    ENDS
END     BEGIN
```

以上三条宏指令展开后的语句为：

```
1    LEA    DX,MSG
1    MOV    AH,9
1    INT    21H
1    LEA    DX,BUF
1    MOV    AH,10
1    INT    21H
1    MOV    AH,4CH
```

```
1     INT     21H
```

5.2.6　简化段定义

实际上，汇编语言源程序可以用两种格式书写：一种是前面介绍的完整段定义格式，另一种是简化段定义格式。

例 5-25

```
.MODEL SMALL                                ;定义程序的存储模式
.STACK 20H
.DATA                                       ;定义数据段
    STRING DB 'HELLO, ASSEMBLY',0DH,0AH, '$'
.CODE                                       ;定义代码段
.STARTUP                                    ;程序起始点，建立 DS、SS
    MOV DX,OFFSET STRING                    ;指定字符串
    MOV AH,9
    INT 21H                                 ;利用功能调用显示信息
.EXIT 0                                     ;程序结束点，返回 DOS
.END                                        ;汇编结束
```

1. 存储模式伪指令 MODEL

用来表示存储模式，说明在存储器中代码段、数据段等是如何存放的，段的大小有什么限制，数据、代码寻址是近属性还是远属性。存储模式伪指令语句必须位于所有段定义语句之前。

格式：.MODEL 存储模型

常用的存储模型有：

（1）TINY：编译时，所有段地址寄存器都被设置为同一值，即代码段、数据段、堆栈段都在同一个段内。段的大小不超过 64KB，指令转移、程序调用、数据访问等都是近属性（NEAR）。TINY 模式是 MASM 6.0 才引入的，用于创建 COM 类型程序（其他模式产生 EXE 程序），COM 程序必须从段内偏移地址为 0100H 的存储单元开始。一般用于小程序。

（2）SMALL：最多只有两个段：一个代码段，一个数据段，并且两者是独立的，即两个段基址不同。两个段的大小都不超过 64KB，指令转移、程序调用、数据访问等都是近属性（NEAR），此模式下程序的最大长度为 128KB。如果还有堆栈段和附加段，则数据段、堆栈段、附加段共用同一个段基址，段长度仍不超过 64KB。是一般应用程序常用的模型。

（3）MEDIUM：可以有多个代码段，但只有一个数据段，并且段的大小都不超过 64KB。所以数据访问是近属性（NEAR），而指令转移、程序调用可以是近属性（NEAR），也可以是远属性（FAR），缺省时为远属性（FAR）。适合于数据量小但代码量大的程序。

（4）COMPACK：可以有多个数据段，但只有一个代码段，并且段的大小都不超过 64KB。所以指令转移、程序调用是近属性（NEAR），而数据访问可以是近属性（NEAR），也可以是远属性（FAR），缺省时为远属性（FAR）。适合于数据量大但代码量小的程序。

（5）LARGE：允许有多个代码段和多个数据段。数据段可以超过 64KB，但静态数据（不能改变的数据）仍限制在 64KB 之内。指令转移、程序调用、数据访问可以是近属性（NEAR），也可以是远属性（FAR），缺省时均为远属性（FAR）。适用于较大型程序。

（6）HUGE：与 LARGE 模型相同，只是静态数据不再被限制在 64KB 之内。

（7）FLAT：允许用 32 位偏移量，只能在 80386 及其以后的计算机系统中运行，只能用于 32 位程序，只能在 OS/2 下或其他保护模式的操作系统下使用，在 DOS 下不允许使用这种

模型。MASM 6.0 可以支持这种模型，但 MASM 5.0 版本不支持。

2. 段定义伪指令

（1）定义堆栈段。

.STACK [SIZE]

参数 SIZE 指定堆栈段所占存储区的字节数，缺省时为 1KB。段名为 STACK。

（2）定义数据段。

简化段定义中把数据段分得很细。首先把数据段分为常量数据段和变量数据段，变量数据段又可分为远数据段和近数据段，然后根据变量是否初始化，进一步将其分为初始化数据段和未初始化数据段。所以简化段定义中的数据段有以下几种：

1）CONSTANTS（常数段）。

2）INITIALIZED DATA（初始化数据段）。

3）UN INITIALIZED DATA（未初始化数据段）。

4）FAR INITIALIZED DATA（远初始化数据段）。

5）FAR UN INITIALIZED DATA（远未初始化数据段）。

所以相对应数据段定义伪指令有：

1）.CONST

```
...                    ;定义常量
```

用来定义只读的常量数据段，段名为 CONST。

2）.DATA

```
...                    ;定义数据
```

用来定义初始化数据段，段中的变量具有初值，段名为_DATA。

3）.DATA?

```
...                    ;定义数据
```

用来定义未初始化数据段，段中的变量没有初值，段名为_BSS。

4）.FARDATA [NAME]

```
...                    ;定义数据
```

用来定义远初始化数据段，段中的变量有初值，默认段名为 FAR_DATA。

5）.FARDATA? [NAME]

```
...                    ;定义数据
```

用来定义远未初始化数据段，段中的变量没有初值，默认段名为 FAR_BSS。

（3）定义代码段。

.CODE [段名]

用于定义代码段，参数 NAME 用来指定代码段的段名。缺省时则采用默认段名。在 TINY、SMALL、COMPACT 和 FLAT 模式下，默认的代码段名是_TEXT。在 MEDIUM、LARGE 和 HUGE 模式下，默认的代码段名是模块名_TEXT。

说明：

①采用简化段定义伪指令前，需要有.MODEL 语句。

②段定义伪指令指明一个逻辑段的开始，同时自动结束前面的一个段，所以不必用 ENDS 作为段的结束符。

③在简化段定义中，可以使用汇编程序提供的预定义符号，预定义符号如表 5-5 所示。

表 5-5　预定义符号表

符号	说明	符号	说明
@CODE	返回代码段段值	@FARDATA	返回 FAR DATA 定义的段值
@CODESIZE	返回代码段长度值	@FARDATA?	返回 FAR DATA?定义的段值
@CURSEG	返回当前段段值	@MODEL	返回选择的内存模式
@DATA	返回数据段段值	@STACK	返回堆栈段段值
@DATASIZE	返回数据段长度值	@WORDSIZE	返回当前段类型属性

例如：

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

3. .STARTUP

指定程序开始执行的起始点，（在 DOS 下）用于自动初始化寄存器 DS、SS 和 SP。

它等价于：

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

4. .EXIT　[返回参数]

用于终止程序执行，返回操作系统。它的参数是一个返回的数码，用 0 表示没有错误。

它等价于：

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

5. .END　[标号]

指示汇编程序到此结束汇编，标号用于指定程序开始执行点，连接程序将据此设置 CS:IP 值。若采用.STARTUP 指明了程序开始执行点，则可以省略标号。

注意：MASM 5.0/5.1 不支持.STARTUP、.EXIT 0 和.END。

5.3　汇编语言程序设计基本方法

5.3.1　程序设计的基本步骤

（1）分析问题。

对题目给出的已知条件和要完成的任务进行详细的了解和分析，将实际问题转化为计算机可以处理的问题。

（2）确定算法。

算法，即利用计算机解决问题的方法和步骤。计算机一般只能进行最基本的算术运算和逻辑运算，要完成较为复杂的运算和控制操作，就必须选择合适的算法。

（3）设计流程。

将算法以流程图的方式画出来。

画流程图是指用各种图形、符号、指向线等来说明程序设计的过程。国际通用的图形和符号说明如下：

1）椭圆框：起止框，在程序的开始和结束时使用，如图 5-3（a）所示。

2）矩形框：处理框，表示要进行的各种操作，如图 5-3（b）所示。

3）菱形框：判断框，表示条件判断，以决定程序的流向，如图 5-3（c）所示。

4）指向线：流程线，表示程序执行的流向，如图 5-3（d）所示。

（a）起止框　（b）处理框　（c）判断框　（d）流程线

图 5-3　国际通用的流程图图形

（4）分配空间。

合理分配存储空间，即分段和数据定义，合理地使用寄存器。

（5）编写程序。

根据前面确定的算法流程图，采用汇编程序设计语言编写程序。

（6）调试运行。

程序编写好以后，检查语法错误，上机汇编、连接、调试运行，检验程序是否正确，能否实现预期功能。

5.3.2　顺序、分支与循环程序设计

利用计算机解决实际问题时，其操作控制执行步骤有时是按顺序执行的，有时需要根据实际情况选择某一个分支的操作执行，有时需要对某一些操作步骤反复执行，与之相对应，就有 3 种程序结构：顺序结构、分支结构、循环结构。

1. *顺序结构*

顺序结构程序完全按指令书写的前后顺序，从头至尾逐条执行，是最常用、最基本的程序结构。常用于处理查表程序、计算表达式程序。

例 5-26　编写程序计算表达式：$f=\dfrac{a*b+c}{d-e}$。

```
DATA    SEGMENT
          A      DB      5
          B      DB      10
          C      DB      15
          D      DB      30
          E      DB      20
          F      DB      ?
DATA    ENDS
CODE    SEGMENT
ASSUME     CS: CODE, DS: DATA
START:    MOV    AX,DATA                    ;初始化 DS
          MOV    DS,AX
          MOV    AX,0
          MOV    AL,A
          MUL    B                          ;AX=A*B
          MOV    BL,C
          MOV    BH,0
          ADD    AX,BX                      ;AX=A*B+C
          MOV    CL, D
          SUB     CL, E                     ;CL=D-E
```

```
        DIV     CL                      ;AL=AX/CL=(A*B+C)/(D-E)
        MOV   F,AL                      ;F=AL=(A*B+C)/(D-E)
        MOV   AH,4CH                    ;程序结束，返回 DOS
        INT    21H
CODE   ENDS
END START
```

2. 分支结构

根据指定的条件选择程序执行的方向，这种程序结构称为分支程序结构。常根据 CMP、TEST 等指令执行后形成的状态标志，通过转移指令 JXX 判断标志位的变化，来实现条件判断控制程序转向某个分支执行；或通过 JMP 实现无条件转移。

根据分支转向的不同结构，可将分支结构分为 3 种：单分支结构、双分支结构和多分支结构。

（1）单分支程序。满足条件时转向分支执行，否则顺序执行。流程图如图 5-4（a）所示。

例 5-27　求数的绝对值。

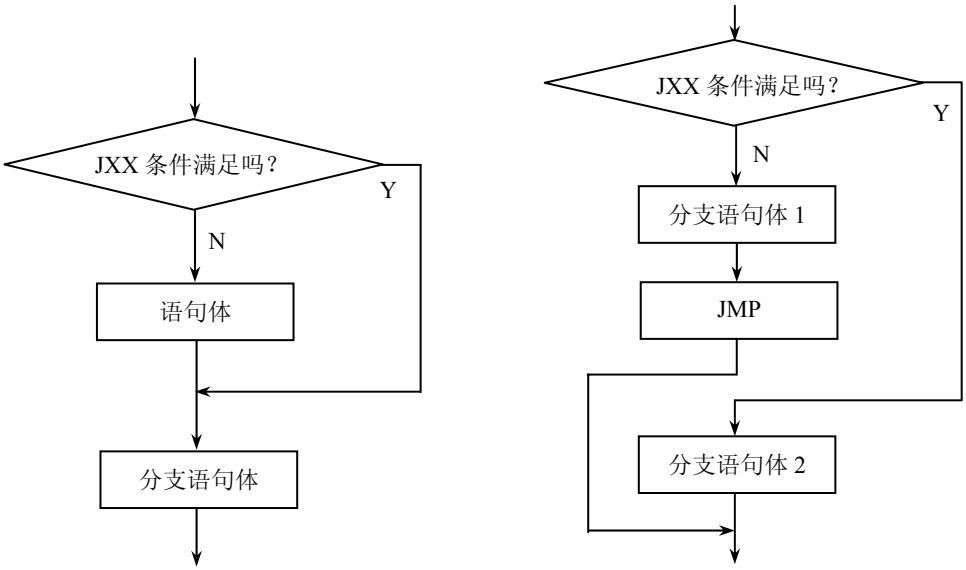
```
DATA    SEGMENT
X   DB   -25
X_ABS   DB   ?
DATA ENDS
CODE SEGMENT
ASSUME   DS:DATA,CS:CODE
START:  MOV   AX,DATA
        MOV   DS,AX                     ;初始化
        MOV   AL,X                      ;X 取到 AL 中

        CMP   AL,0                      ;测试 AL 正负
        JGE   NEXT                      ;X≥0，转 NEXT
        NEG   AL                        ;否则 X<0，AL 求补
NEXT:   MOV   X_ABS,AL                  ;送结果
        MOV   AH,4CH
        INT   21H                       ;返回 DOS
CODE ENDS
END   START
```

（2）双分支程序。条件成立转向分支语句体 2 执行，否则顺序执行分支语句体 1，并且执行完分支语句体 1 后要跳过分支语句体 2，用 JMP 无条件跳转到分支语句体 2 后执行。流程图如图 5-4（b）所示。

例 5-28　奇偶数判断，是奇数时输出 N，是偶数时输出 Y。

```
DATA    SEGMENT
        X    DB    -30
DATA ENDS
CODE SEGMENT
ASSUME   DS:DATA,CS:CODE
START:  MOV   AX,DATA
        MOV   DS,AX                     ;初始化
        MOV   AL,X
        SHR   AL,1                      ;将 X 最低位右移至 CF 中
        JC    NEXT                      ;CF 为 1，则为奇数，转 NEXT
        MOV   DL,'Y'                    ;否则 CF 为 0，则为偶数，输出字符'Y'
        JMP   DISP
NEXT:   MOV   DL,'N'                    ;为奇数时输出字符'N'
DISP:   MOV   AH,02H
        INT   21H                       ;显示 DL 中的字符
```

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



（a）单分支结构　　（b）双分支结构

图 5-4　单、双分支结构流程图

（3）多分支程序。需要对多个条件进行判断，每个条件都对应一个分支，满足某个条件时就进入相对应的分支执行。流程图如图 5-5 所示。

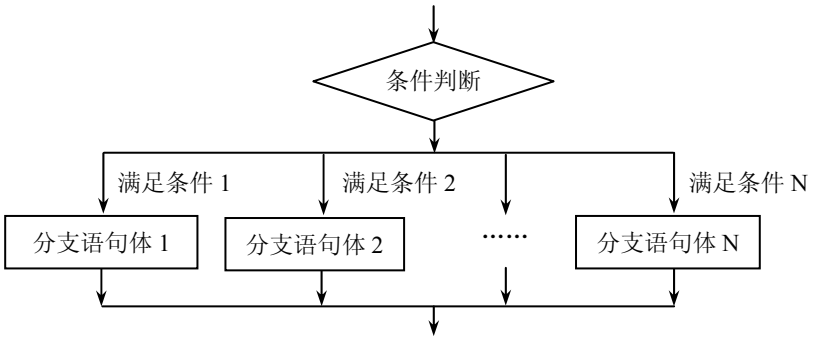


图 5-5　多分支结构流程图

对于多分支程序结构，可以采用以下两种解决方法：

1）逻辑分解方法。将多分支结构以逻辑等效的方法分解为一串双分支结构。

例 5-29　求符号函数 Y 的值：$Y=\begin{cases}1 & (X>0)\\ 0 & (X=0)\\ -1 & (X<0)\end{cases}$

```
DATA   SEGMENT
   X    DB    -10
   Y    DB    ?
DATA ENDS
CODE SEGMENT
   ASSUME    DS:DATA,CS:CODE
```

```
START:  MOV   AX,DATA
        MOV   DS,AX                ;初始化
        CMP   X,0                  ;比较 X 与 0 的大小
        JG    L1                   ;X>0，转 L1
        JE    L0                   ;X=0，转 L0
        MOV   Y,-1                 ;否则 X<0，Y=-1
        JMP   EXIT
L1:     MOV   Y,1                  ;Y=1
        JMP   EXIT
L0:     MOV   Y,0                  ;Y=0
EXIT:   MOV   AH,4CH
        INT   21H                  ;返回 DOS
CODE    ENDS
END     START
```

2）地址表方法。在数据段定义一个地址表，依次存放各分支语句体的入口地址，用寄存器间接寻址或寄存器相对寻址方式产生转移目标地址，实现转移。

分支入口地址=地址表首地址+偏移地址

例 5-30　假设一个程序有 5 个分支，根据用户输入的数字（0～4）转入相应的分支去执行，试编写程序。

```
DATA    SEGMENT
        FUN   DW   FUN0,FUN1,FUN2,FUN3,FUN4
;将标号 FUN0、FUN1、FUN2、FUN3、FUN4 偏移地址存入变量 FUN 中
DATA   ENDS
CODE   SEGMENT
        ASSUME   DS:DATA,CS:CODE
START:  MOV   AX,DATA
        MOV   DS,AX
        MOV   AH,01H           ;选择输入数字字符 0、1、2、3、4 中的一个
        INT   21H
        SUB   AL,'0'           ;将输入的数字字符转化成对应的数，存入 AX 中
        MOV   AH,0
        SHL   AX,1             ;AX=AX*2，各标号偏移地址占两个字节
        LEA   BX,FUN
        ADD   BX,AX            ;BX=地址表 FUN 首地址+2*N（输入的数字）
        JMP   [BX]             ;根据输入的数字，形成分支语句体入口地址，转移到相应的
                               ;分支语句体去执行，采用的是寄存器间接寻址方式
FUN0:
        …                      ;分支语句体 0
        JMP   EXIT             ;结束
FUN1:
        …                      ;分支语句体 1
        JMP   EXIT             ;结束

FUN2:
        …                      ;分支语句体 2
        JMP   EXIT             ;结束

FUN3:
        …                      ;分支语句体 3
        JMP   EXIT             ;结束
FUN4:
        …                      ;分支语句体 4
EXIT:   MOV   AH,4CH           ;返回 DOS
```

```
        INT    21H
CODE    ENDS
END     START
```

（4）说明：

①对于既能用双分支结构，又能用单分支结构实现的程序，宜采用单分支结构，以减少转移次数，程序结构简单。

②对于多分支结构程序，宜采用地址表法，以减少转移次数，程序结构简单。

③对分支结构程序进行测试时，应对每一个分支都进行检测，才能保证整个程序的正确性。

3. 循环结构

根据某一条件是否成立判断是否需要重复执行某个语句组，这种程序结构称为循环结构。

（1）说明：

①一个循环结构一般由循环条件控制、循环体两部分组成。

循环条件控制：对循环条件进行判断，决定是否继续循环。

循环体：重复执行的语句组。

注意：循环体中应对循环条件的值进行修改，否则将会成为死循环（循环无限次）。

②根据循环条件控制所在的位置，可将循环结构分为两种：

- “先判断、后循环”：先判断循环条件，再决定是否执行循环体。
- “先循环、后判断”：先执行循环体（至少一次），再判断循环条件。

③用的循环指令有：

- 循环指令：LOOP、LOOPE/LOOPZ、LOOPNE/LOOPNZ。
- 转移指令：JCXZ、JXX。

其中 LOOP、JCXZ 常用于循环次数固定的循环结构，称这种循环结构为计数循环；LOOPE/LOOPZ、LOOPNE/LOOPNZ、JXX 常用于循环次数不定的循环结构，称这种循环结构为条件控制循环。

（2）应用举例。

1）计数循环：循环次数已知，用计数器 CX 计数来控制循环次数，要求在循环之前设置 CX 的值，即将循环次数送入 CX 中，然后每循环一次计数器值减 1，直至其值减为 0 则不再循环。

例 5-31 设计一个程序，求 1+2+…+99+100 的和，结果保存在变量 RESULT 中。

```
DATA    SEGMENT
   RESULT   DW   0
DATA    ENDS
CODE    SEGMENT
   ASSUME CS:CODE,DS:DATA
START:   MOV   AX,DATA
         MOV   DS,AX
         MOV   AX,1
         MOV   CX,100
L0:      ADD   RESULT,AX
         INC   AX
         LOOP  L0
         MOV   AH,4CH
         INT   21H
CODE   ENDS
```

```
END     START
```

2）条件控制循环：循环次数未知，需要设置一个循环条件，每次对条件进行判断来确定是否继续转去执行循环体。

例5-32　计算数组ARRAY中元素的平均值、最大值、最小值，数组以-1为结束标志。

分析：数组元素个数不确定，因而循环次数也不确定，需要通过判断数组元素是否为-1来确定是否结束循环。这里需要取出数组的每个元素累加求和，并统计数组元素个数，以计算出平均值；求最大值（或最小值）时，可以先默认数组第一个元素即为最大值（或最小值），送入 MAX（或 MIN）中，然后依次取出数组剩下的元素与 MAX（或 MIN）比较，若该元素的值大于 MAX（或小于 MIN），则将它送入 MAX（或 MIN）中，这样循环结束后，MAX（或 MIN）里面存放的就是数组的最大值（或最小值）。

```
DATA    SEGMENT
        ARRAY   DB   10,58,63,94,85,32,-1
        SUM     DW   0
        COUNT   DB   0
        AVERAGE DB   ?
        MAX     DB   ?
        MIN     DB   ?
DATA    ENDS
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA
START:  MOV     AX,DATA
        MOV     DS,AX
        LEA     SI,ARRAY
        MOV     BL,[SI]
        MOV     MAX,BL          ;取出第一个元素送入 MAX、MIN 中
        MOV     MIN,BL
L0:     MOV     AL,[SI]
        CMP     AL,-1
        JZ      L3              ;判断数组元素是否为-1，若是则结束循环
        CBW
        ADD     SUM,AX          ;若数组元素不为-1，则加入 SUM 中
        INC     COUNT           ;统计元素个数，个数值加 1
        CMP     MAX,AL
        JGE     L1
        MOV     MAX,AL          ;若数组元素值大于 MAX，则将它送入 MAX 中
L1:     CMP     MIN,AL
        JLE     L2
        MOV     MIN,AL          ;若数组元素值小于 MIN，则将它送入 MIN 中
L2:     INC     SI              ;SI 指向数组的下一个元素
        JMP     L0              ;循环处理数组的下一个元素
L3:     MOV     AX,SUM
        DIV     COUNT
        MOV     AVERAGE,AL      ;将数组元素的和 SUM 除以数组元素个数 COUNT，得到商
                                ;即为平均值，送入 AVERAGE
        MOV     AH,4CH          ;返回 DOS
        INT     21H
CODE    ENDS
END     START
```

3）循环嵌套。

实际应用中，经常出现在一个循环中又包含另一个循环，称为循环嵌套，也称多重循环。

例 5-33 数组 ARRAY 的长度为 N（即数组 ARRAY 中共有 N 个数，假设均为无符号字节数），请将数组中的数按升序（从小到大）排序。

分析：排序算法有很多种，这里采用冒泡排序算法，其算法思想为：

①第 1 趟：从数组的最左边开始，依次将相邻两个数作比较，若前者大于后者，则交换两者的值，经 N-1 次两两相邻比较后，最大的数已交换到最后一个位置。

②第 2 趟：对前 N-1 个数，按上法两两相邻比较，经 N-2 次比较后得到次大的数，安置在第 N-1 个元素的位置。

③重复上述过程，经过 N-1 趟冒泡排序后，数据呈升序排列。

```
DATA    SEGMENT
    ARRAY   DB   10,58,23,94,85,32,70,5,42,62
    N   EQU   $-ARRAY              ;N 为数组长度
DATA    ENDS
CODE    SEGMENT
    ASSUME CS:CODE,DS:DATA
START:  MOV   AX,DATA
        MOV   DS,AX
        MOV   CX,N-1               ;设置外层循环次数为 N-1，即冒泡排序趟数
L0:     PUSH  CX                   ;保存外层循环次数计数器
        LEA   SI,ARRAY
L1:     MOV   AL,[SI]
        MOV   AH,[SI+1]
        CMP   AL,AH                ;相邻两个数作比较
        JLE   L2                   ;若前者小于等于后者，则不交换，转去比较下一对相邻
                                   ;的两个数
        MOV   [SI+1],AL            ;否则（即前者大于后者），交换两者的值
        MOV   [SI],AH
L2:     INC   SI
        LOOP  L1                   ;内存循环两两比较，循环次数刚好等于外层循环次数计数器
                                   ;的值，所以不需要另外设置内层循环计数器 CX 的值
        POP   CX                   ;恢复外层循环次数计数器
        LOOP  L0                   ;继续外层循环
        MOV   AH,4CH               ;返回 DOS
        INT   21H
CODE    ENDS
        END   START
```

5.3.3 子程序设计

通常将一个大的程序按照功能划分为几个子程序（子程序就是一个功能上相对独立的程序段，可以被多次重复调用。在一个完整的程序中，可以有多个子程序，子程序能被别的程序所调用，也可以调用其他子程序，也称过程），通过调用各个子程序来实现程序的功能。

在定义子程序时，一般需要包含以下几个部分：

- 保护现场
- 子程序体
- 恢复现场
- 子程序返回

调用子程序时，子程序与主程序之间往往存在着数据的交流，称主程序传递给子程序的

数据为入口参数，称子程序返回给主程序的结果数据为出口参数。

常采用的参数传递方法有：通过寄存器传递、通过共享变量传递、通过堆栈传递。

（1）通过寄存器传递参数。

把入口参数、出口参数存放于约定的寄存器中，这是最常用的参数传递方式。通过寄存器传递参数时，需要视具体情况来选择是否需要对入口参数、出口参数进行保护和恢复。由于通用寄存器个数有限，通过寄存器传递参数的方法只适合参数个数较少的场合。

例 5-34　编写一个子程序，在数据块中查找某个指定数据，若找到则把该数据在数据块中的序号返回，若找不到则返回-1。

```
DATA    SEGMENT
    ARRAY   DB   10,58,23,94,85,32,70,5,42,62
    N   EQU   $-ARRAY                    ;N 为数组长度
DATA    ENDS
CODE SEGMENT
    ASSUME CS:CODE,DS:DATA
START:    MOV   AX,DATA
          MOV   DS,AX
          LEA   SI,ARRAY
          MOV   CX,N
          MOV   DL,94
          CALL  LOOKUP                 ;调用子程序
          MOV   AH,4CH                 ;返回 DOS
          INT   21H
;子程序名：LOOKUP
;功能：片内 RAM 中的数据检索
;入口参数：SI 存放数据块首地址，CX 存放数据块长度，DL 存放要查找的数据
;出口参数：若找到，则将数据的序号存入 DI，否则存-1 到 DI
LOOKUP    PROC
          PUSH  CX
          PUSH  SI
          MOV   DI ,-1
L0:       CMP   DL,[SI]
          JZ    L1
          INC   SI
          LOOP  L0
          JMP   L2
L1:       MOV   DI,SI
          SUB   DI,OFFSET ARRAY
L2:       POP   SI
          POP   CX
          RET
LOOKUP    ENDP
CODE      ENDS
END       START
```

（2）通过共享变量传递参数。把入口参数、出口参数存放于约定的内存共享变量中。若子程序和调用程序在同一程序模块中，则子程序可直接访问模块中的变量，进行参数传递；若子程序和调用程序在两个不同的程序模块中，需要利用 PUBLIC、EXTREN 对共享变量进行声明才能访问共享变量。

若调用程序还要引用共享变量原来的值，则需要对共享变量进行保护和恢复。

通过共享变量传递参数的方法适合于传递参数较多的情况，以及在多个程序段间传递参数的情况。但是采用这种参数传递方式的子程序的通用性比较差。

例 5-35 编写一个子程序，从键盘输入若干字符，以“$”结束，并将输入的字符存入数组 STRING 中。要求：若输入的字符为大写，则需要将其改为小写后存入数组。

```
DATA    SEGMENT
      STRING    DB    100    DUP(?)
DATA    ENDS
CODE    SEGMENT
      ASSUME    CS:CODE,DS:DATA
START:    MOV    AX,DATA
          MOV    DS,AX
          MOV    SI,OFFSET STRING
          CALL    STOSTR                    ;调用子程序
          MOV    AH,4CH                     ;返回 DOS
          INT     21H
;子程序名：STOSTR
;功能：输入字符串，并将其中的大写字母改为小写字母，然后存入数组 STRING
;入口参数：SI 存放数组 STRING 的首址
;出口参数：数组 STRING
STOSTR    PROC
      PUSH    AX
AGAIN:    MOV     AH,1
          INT     21H
          CMP     AL,'$'
          JZ      OVER                      ;结束
          CMP     AL,'A'
          JL      NEXT                      ;不是大写
          CMP     AL,'Z'
          JG      NEXT                      ;不是大写
          ADD     AL,32                     ;是大写
NEXT:     MOV     [SI],AL                   ;存入
          INC     SI
          JMP     AGAIN
OVER:     POP     AX
          RET
STOSTR    ENDP
CODE      ENDS
END       START
```

（3）通过堆栈传递参数。把入口参数、出口参数存放于堆栈当中。在调用子程序前，主程序将入口参数压入堆栈，子程序从堆栈中取出入口参数；在子程序返回前，子程序将出口参数压入堆栈，主程序从堆栈中取到出口参数。

采用堆栈传递参数方法是编译程序处理参数传递，以及汇编语言与高级语言混合编程时的常规方法。通过堆栈传递参数的方法适合于传递参数较多的情况，采用堆栈传递参数时要：保证子程序中堆栈操作的正确性，对堆栈的压入和弹出操作要成对使用，保持堆栈的平衡，避免因堆栈操作而造成子程序不能正确返回的错误。

例 5-36 编写程序，求数据块 BUF 中存放的若干个无符号字节数据的平均值。

```
DATA    SEGMENT
```

```
        BUF   DW   10,58,23,94,85,32,70,5,32,62
        N   EQU   ($-BUF)/2                    ;N 为数据块长度
DATA    ENDS
CODE    SEGMENT
        ASSUME CS:CODE,DS:DATA
START:  MOV    AX,DATA
        MOV    DS,AX
        MOV    AX,OFFSET BUF
        PUSH   AX
        CALL   AVERAGE                         ;调用子程序
        MOV    AH,4CH                          ;返回 DOS
        INT    21H
;子程序名：AVERAGE
;功能：求数据块中若干数据的平均值
;入口参数：数据块的首地址，压入堆栈
;出口参数：平均值，压入堆栈
AVERAGE PROC
        POP    BX
        POP    SI
        MOV    AX,0
        MOV    CX,N
L0:     ADD    AX,[SI]
        ADD    SI,2
        LOOP   L0
        MOV    CL,N
        DIV    CL
        CBW
        PUSH   AX
        PUSH   BX
        RET
AVERAGE ENDP
CODE    ENDS
END START
```

5.3.4　子程序的嵌套与递归

1. 子程序的嵌套

在一个子程序中调用其他的子程序，称为子程序的嵌套，如图 5-6 所示。嵌套的层数不限，只要堆栈空间足够即可。

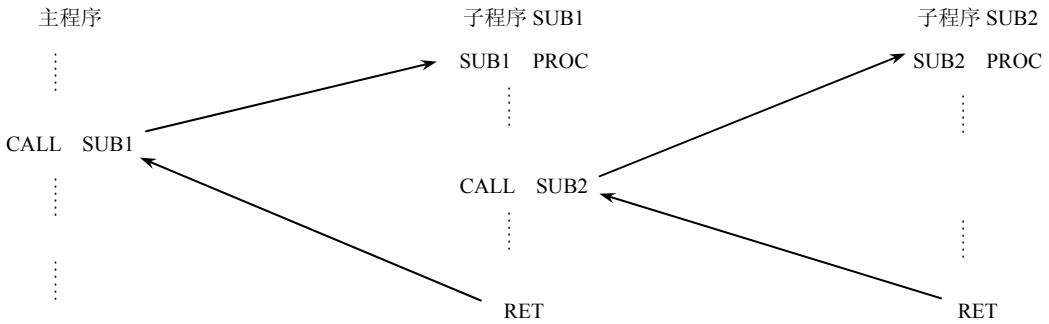


图 5-6　子程序嵌套

例 5-37

```
ALDISP   PROC
         PUSH    AX
         PUSH    CX                  ;实现 AL 内容的显示
         PUSH    AX                  ;暂存 AX
         MOV     CL,4
         SHR     AL,CL               ;转换 AL 的高 4 位
         CALL    HTOASC              ;子程序调用（嵌套）
         POP     AX                  ;转换 AL 的低 4 位
         CALL    HTOASC              ;子程序调用（嵌套）
         POP     CX
         POP     AX
         RET
ALDISP   ENDP
                                     ;将 AL 低 4 位表达的一位十六进制数转换为 ASCII 码
HTOASC   PROC
         PUSH    AX
         PUSH    BX
         PUSH    DX
         MOV     BX,OFFSET ASCII     ;BX 指向 ASCII 码表
         AND     AL,0FH              ;取得一位十六进制数
         XLAT    ASCII               ;换码：AL←CS:[BX+AL]，注意数据在代码段 CS
         MOV     DL,AL               ;显示
         MOV     AH,2
         INT     21H
         POP     DX
         POP     BX
         POP     AX
         RET                         ;子程序返回
;子程序的数据区
ASCII    DB 30H,31H,32H,33H,34H,35H,36H,37H
         DB 38H,39H,41H,42H,43H,44H,45H,46H
HTOASC   ENDP
```

注意：子程序可以与主程序共用一个数据段，也可以使用不同的数据段（注意修改 DS），还可以在子程序最后设置数据区（利用 CS 寻址）。

2. 子程序的递归

在一个子程序中又直接或间接调用子程序本身，称为子程序的递归，如图 5-7 所示。

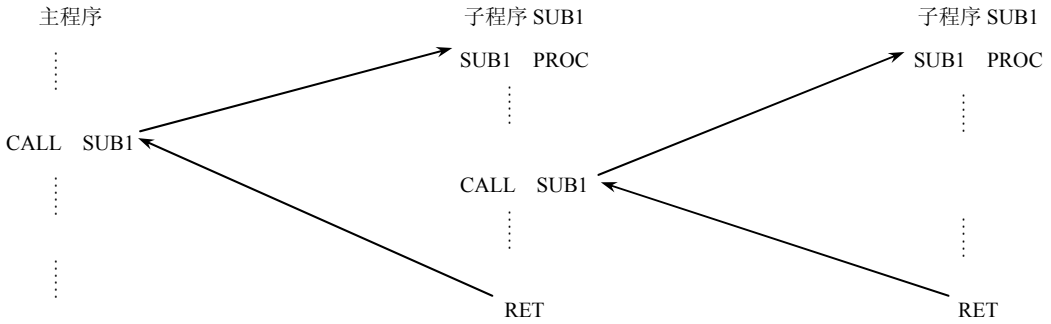


图 5-7 子程序递归

例5-38　求自然数N（N≥1）的阶乘。

```
DATA   SEGMENT
    N   DB   3
    F_MUL   DW   ?
DATA   ENDS
STACK   SEGMENT    PARA   STACK   'STACK'
    DB   100   DUP(?)
STACK   ENDS
CODE   SEGMENT
    ASSUME   CS:CODE, DS:DATA, SS:STACK
BEGIN:     MOV     AX, DATA
           MOV     DS, AX
           MOV     AH, 0
           MOV     AL, N
           CALL    FACTOR
           MOV     F_MUL,AX
           MOV     AH,4CH
           INT     21H
FACTOR     PROC
           PUSH    AX
           SUB     AX, 1
           JNE     RECUR
           POP     AX
           JMP     OVER
RECUR:     CALL    FACTOR
           POP     CX
           MUL     CL
OVER:      RET
FACTOR     ENDP
CODE   ENDS
END    BEGIN
```

5.4　Windows 汇编语言程序设计

前面介绍了基于 DOS 的 8086 汇编语言程序设计，了解了 8086 汇编语言程序设计的思路和方法。本节对 Windows 下的汇编程序设计进行简要介绍。

在 Windows 环境下，不需要掌握更多的硬件知识就能够应用 Win32 汇编语言编制应用程序。在 DOS 系统中用汇编语言编制程序是借助中断来调用操作系统内核提供的功能，而 Win32 汇编是借助应用编程接口 API（Application Program Interface）去调用操作系统内核。Windows 环境下的很多高级语言对功能调用与实现的细节进行了不同程度的封装，如多线程处理和消息循环等都被隐藏封装起来。虽然能够使用它们进行可视化编程，却无法全面了解 Win32 操作系统程序的具体运行方式。由于封装使操作出现了某种缺陷和不足。如 VB 不支持指针，而程序员编程需要有指针的 API，结果操作起来十分不方便，使多线程一类特征在 VB 中无法实现。但用 Win32 汇编语言能洞察到操作系统的真实工作情况，充分发挥 Windows 系统的各种功能，如对 Windows 环境的文件进行加密保护，编制出在 Windows 操作系统管理下各种文件的防病毒程序。

5.4.1 Windows 汇编语言程序的例子

在 Windows 操作系统下用消息框显示一个字符串“Good morning!”的汇编程序如下：

```
.386
.MODEL          FLAT,STDCALL
OPTION          CASEMAP:NONE
INCLUDE         WINDOWS.INC
INCLUDE         USER32.INC
INCLUDELIB      USER32.LIB
INCLUDE         KERNEL32.INC
INCLUDELIB      KERNEL32.LIB
.DATA
GDCAPTION       DB    "A Good cation!",0
GOOD            DB    "Good morning!",0
.CODE
START:
INVOKE          MessageBox,NULL,OFFSET GOOD,OFFSET GDCAPTION,MB_YESNO
INVOKE          ExitProcess,NULL
END             start
```

1. 386 伪指令

在上述程序中，首先进行模式定义。“.386”表明 Win32 汇编程序工作于 80386 及以上的处理器。也可以使用. 486、. 586、. 686 等伪指令。但是，如果程序没有使用特定的基于 386 以上 CPU 的指令，从软件的兼容性考虑，建议使用.386。

2. MODEL 伪指令

.MODEL 是用来指定内存模式的伪指令，在 Win32 下，只有一种内存模式，即 FLAT。Windows 操作系统为每一个应用程序建立一个 4GB 的线性空间。代码段、数据段和堆栈段都使用同一个段，内存寻址从 0 到 4GB，没有 64KB 的段大小限制。汇编程序自动为各段寄存器做如下段约定：

```
ASSUME CS:FLAT,DS;FLAT,SS:FLAT,ES:FLAT
```

即 CS、DS、ES、SS 指向同一段 FLAT，FS 和 GS 在 Win32 汇编中不用。STDCALL 用于指出调用子程序或编程接口 API 时参数传递的次序和堆栈平衡的方法。

3. OPTION 伪指令

语句 OPTION CASEMAP:NONE，用以说明程序中的变量和子程序名是不分大小写的。与 DOS 汇编程序不同，Win32 汇编程序不考虑堆栈，系统会为程序分配一个向下扩展的段作为堆栈段，因此，堆栈段定义会被忽略。

4. INCLUDE 伪指令

MASM32 工具包包含有各种 DLL 的 API 函数声明列表，每个 DLL 都有相对应的 DLL.INC 文件，程序中如果用到某个 DLL 文件中包含的函数，必须用 INCLUDE 语句将其包含进来，如：

```
INCLUDE    USER32.INC
INCLUDE    KERNEL32.INC
```

5. INCLUDELIB 伪指令

不同类的 API 函数存放在不同的动态链接库 DLL 中，为了让连接程序快速地搜索到 API 函数在哪个 DLL 库，Win32 还定义了一种库文件，称为导入库文件。一个 DLL 库对应一个导

入库。INCLUDELIB 语句用于指定链接时所用的导入库，以便通知连接程序在哪个 DLL 库中去找连接所需的 API 函数。例如，USER32.DLL 对应的导入库是 USER32.LIB，导入 USER32DLL 中的 API 函数可通过语句 INCLUDELIB USER32.DLL 来实现。

LIBC.LIB 库中包含了大部分的运行库函数连接信息，程序的入口函数_main 也是在 LIBC.LIB 中声明的。因此在程序中包含“INCLUDELIB LIBC.LIB”伪指令是非常有必要的，否则连接时将会出现无法解析外部符号的错误。

6. “分段”伪指令

.DATA、.DATA?、.CONST 和.CODE 是 4 个“分段”伪指令。

Win32 FLAT 内存模式隐藏了分段机制，因此只有两种性质的“分段”：DATA 和 CODE。

在程序的.CODE 分段中，包括函数定义，是程序“代码”部分。对于 Win32 控制台程序，入口函数名必须是_main，而不能像 DOS 汇编程序那样自定义入口函数名称。在 Win32 环境中，可以像 DOS 汇编那样对程序入口进行定义。本 Win32 例程入口为标号 START，程序结束处用 END 语句加标号“START”来实现，其用法与 DOS 汇编中完全相同。

Win32 汇编程序的数据段有 3 类：.DATA、.DATA?和.CONST，在生成的可执行文件中不同的数据会被放在相应的节区中。

第一类是可读写的已定义变量，这些数据必须定义在.DATA 段中；第二类是可读写的未定义变量，这些数据一般定义在.DATA?段中，也可以定义在.DATA 段中；第三类是常量，这些数据在程序装入时已经有效，在执行过程中也不需要修改，可以放在.CONST 段中。.CONST 段作为常量段，它是可读不可写的。当然，也可以像本例一样把常量放到.DATA 段中。

7. END 伪指令

指示汇编结束位置，其后的任何文本（包括指令和伪指令）都会被汇编程序忽略。

再次强调，由于 Win32 控制台程序的入口函数名必须是_main，所以 end 伪指令不能像在 DOS 汇编程序中那样指定除_main 外的任何其他标号。

8. Windows API 调用

Win32 环境中的编程接口 API 代替了 DOS 调用系统功能的中断方式。但和 DOS 不同，Win32 把系统功能模块放在 Windows 的动态链接库中，Win32 汇编程序中的功能实现是通过编程接口 API 调用存放在 DLL 中的函数，从而完成 DOS 环境中借助中断方式来调用系统的功能。

在本例中以 INVOKE 伪操作指令实现对 MessageBox 的调用：

```
INVOKE MessageBox,NULL,OFFSET GOOD,OFFSET GDCAPTION,MB_YESNO
```

需要指出的是，INVOKE 并不是 80386 处理器的指令，而是宏汇编 MASM 编译器的伪指令，它完成了汇编调用 MessageBox 函数的功能。此外，本例还用到另外一个 API 函数：ExitProcess，它位于 KERNEL32.DLL 中。

通过以上实例，可以了解在 Windows 环境下应用汇编语言编程的方法，以及 Win32 汇编程序的基本结构。

5.4.2　Windows 程序设计的特点

汇编语言和微处理器及操作系统是紧密相关的。随着 Windows 操作系统占领市场，汇编语言程序设计也相应地从 DOS 下的实地址模式过渡到 Windows 下的 32 位保护模式。Windows 汇编程序设计和 DOS 汇编程序设计有许多相似之处，如它们有相同的指令系统、类似的寻址

方式等。但也有不少根本性的区别，如内存管理、寻址模式、中断和异常处理等。与 DOS 汇编程序设计相比，Windows 汇编程序设计主要有以下不同特点：

（1）工作模式不同。DOS 应用程序工作在实模式方式下。Windows 应用程序工作在保护模式下，系统的一些重要资源对 Windows 应用程序来说是受保护的，Windows 应用程序不能直接访问这些资源，而必须通过某种方式进行间接访问。

（2）内存使用方式不同。DOS 汇编采用分段机制，通过段地址加偏移地址得到相应内存单元的物理地址。在 Windows 中，则使用了“平坦”内存模型，每个 Windows 应用程序都可以使用 32 位地址来访问 4GB 空间的内存单元，不过这个地址一般是虚地址，需要经过分段和分页机构的转换才能得到相应的物理地址。

（3）提供丰富的 API 函数。与 DOS 提供的中断调用类似，Windows 系统提供了丰富的 API 函数供选用。Windows API 支持上千种函数的调用，涉及网络、消息、文件处理、打印、文本、字体、菜单、位图、图标、光栅运算、绘图、设备场景、硬件与系统、进程和线程、控件与消息等各个方面，从而使程序员可以把更多的时间放在程序的逻辑结构和用户界面上。

（4）基于事件的消息驱动机制。Windows 应用程序的重要特点是采用基于事件的消息驱动机制。应用程序对象的每一次操作都对应一个事件的发生，该事件完成应用程序的相关操作，如鼠标的移动、窗口的缩小和关闭等。消息实际上就是 Windows 系统预先定义的常量标识，它具有唯一性。消息发生时，该消息被送往消息队列，应用程序或操作系统依据消息的种类调用相应的事件处理过程。

Windows 的消息种类繁多，大致有以下 4 种：

- 控件消息，主要是控件子窗口向父窗口发送的消息，如 WM_COMMAND。
- 菜单、快捷键等的 WM_COMMAND 消息。
- 标准消息，除以上消息外，多为“WM_”的格式，如 WM_CHAR、WM_NOTIFY。另外，各控件也具有自身的特定消息，如“LB_”开头的列表框消息、“TV_”开头的树形视图消息等。各控件消息不可混用，其使用情况可参考 MSDN（Microsoft Development Network）相关文档。
- 自定义消息，该类消息的消息号不小于 400H。

在 Windows 应用程序中，消息产生源相对固定，程序设计的重点是完善消息对应的事件处理过程。

（5）应用程序独立于硬件设备。Windows 提供了图形设备接口（GDI）技术，使应用程序能够真正独立于硬件设备，与设备无关，从而方便程序的移植。在 Windows 中，相关设备的驱动程序安装完毕并置于当前设备状态后，会形成一个相关设备的环境，即设备上下文 DC（Device Context）。当应用程序需要与相关设备通信时，只要获得 DC 并与之通信即可，其余的全部交由操作系统去处理。

（6）中断。保护模式下的微处理器设置了 4 个特权级，从高到低分别为特权级 0、1、2、3 级。Windows 操作系统只使用了其中的两个级别，操作系统内核及各种设备驱动程序运行在最高级（0 级），应用程序运行在最低级（3 级）。

在 Windows 操作系统中，使用 API 来代替中断服务子程序提供的系统功能，所以在 Win32 汇编中，INT n 指令失去了存在的意义。Windows 的 API 函数能够被应用程序直接调用，并且它比 DOS 下的中断调用具有更丰富的功能。

（7）程序结构不同。DOS 程序普遍采用结构化程序设计方法，程序整体通常采用顺序执

行的方式，中间穿插有分支结构和循环结构。Windows 应用程序一般不采用顺序执行的方式，而是采用 GUI（图形用户界面）和基于消息的机制，应用程序的主要任务是捕获用户在图形用户界面触发的消息并对其做出响应。

（8）资源丰富，使用方便。Windows 提供了丰富的资源，如菜单、对话框、字符串表、图标、位图、光标、字体，甚至快捷键等。这些资源在 Windows 中都有固定的定义格式，操作方法非常简便。

5.4.3　Windows 汇编程序设计基础

为了充分理解 Windows 下汇编运行的机理和工作过程，需要理解以下知识和概念：

（1）微处理器工作模式选择。

80386 及后续的 x86 系列是典型的 32 位 CPU，它们提供了实模式、保护模式和虚拟 8086 模式等多种操作模式。在保护模式下，32 位处理器可以充分利用 CPU 架构特性并使用所有的指令，拥有最高的性能和兼容性。同时提供了内存分段和分页保护，在硬件级别实现了逻辑地址到线性地址、线性地址再到物理地址的转换，最大程度地保护了操作系统和应用程序。下面以 80386 为例介绍 3 种不同的工作模式。

1）实模式。80386 处理器在复位或加电时以实模式方式启动。此时处理器中的各寄存器以实模式的初始化值工作。在实模式下，80386 处理器只使用了 32 位地址线中的低 20 位，不能对内存进行分页管理，所以指令寻址的地址就是内存中实际的物理地址。实模式下，所有的段均可以读、写和执行。

实模式下的 80386 不支持优先级，所有的指令都工作在特权级（优先级 0），因此可以执行所有特权指令，包括读写控制寄存器 CR0 等。实际上，80386 就是通过在实模式下初始化控制寄存器、GDTR、LDTR、IDTR 与 TR 等管理寄存器以及页表，然后再通过加载 CR0，使 CR0 中的“保护模式使能位”置 1 而进入保护模式的。实模式不支持硬件上的多任务切换。

实模式下的中断处理方式和 8086 处理器相同，也用中断向量表来定位中断服务程序地址。中断向量表的结构也和 8086 处理器一样，每 4 个字节组成一个中断向量，其中包括两个字节的段地址和两个字节的偏移地址。

从编程的角度看，在 80386 处理器的实模式方式下，程序员可以访问 80386 新增的一些寄存器，同时可以使用 80386 处理器中的 32 位寄存器（在 8086 中只能使用 16 位寄存器），从而使程序运行过程更加简洁，同时也加快了执行速度。

2）保护模式。32 位 Windows 操作系统和应用程序都运行在保护模式下，操作系统把每一个 Win32 应用程序放到单独的虚拟地址空间中去运行，每一个应用程序都拥有相互独立的 4GB 逻辑地址空间。在应用程序运行时，操作系统完成虚拟地址到物理地址的转换。而 DOS 运行于实模式下，操作系统和各应用程序运行于同一个地址空间中，并通过物理地址访问代码与数据，很容易导致一个应用程序破坏另一个应用程序甚至是操作系统的数据或代码。

Win32 只有一种内存模式，即 FLAT 模式，又称为“平坦”的内存模式，操作系统和应用程序运行在一个平坦、连续、单独的 4GB 的逻辑地址空间中。FLAT 模式隐藏了内存的分段机制，操作系统负责对段寄存器和描述符表进行初始化，不必像 DOS 汇编编程时那样再对段寄存器进行设置。FLAT 内存模式如图 5-8 所示。Win32 为实现 FLAT 内存模式创建了两个段

描述符，一个引用到代码段而另一个引用到数据段。但是这些段都映射到整个线性地址空间，段描述符有相同的基地址 0，段界限最大为 4GB。

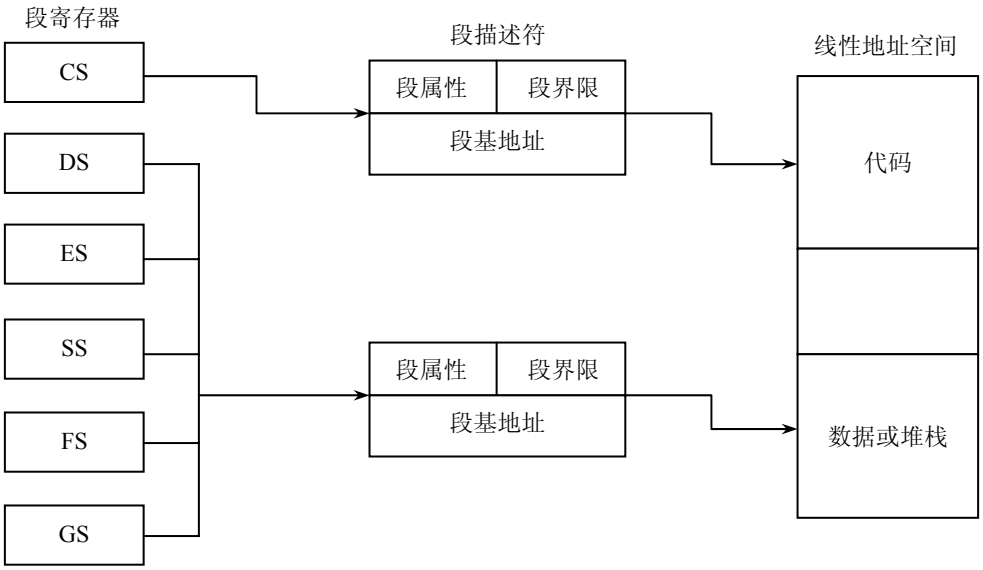


图 5-8　FLAT 内存模式

当 80386 工作在保护模式时，它的所有功能都是可用的。这时 80386 所有的 32 根地址线都可供寻址，寻址空间高达 4GB。保护模式支持内存分段和分页机制，提供了对虚拟内存的良好支持。80386 支持多任务，通过硬件可以在一条指令中实现任务切换。任务环境的保护工作也是由处理器自动完成的。在保护模式下，不同的程序可以运行在不同的优先级上。通过良好的检查机制，一方面使 80386 可在任务间实现数据的安全共享，同时，也可实现各个任务之间的安全隔离。实模式切换到保护模式是通过修改控制寄存器 CR0 的控制位 PE（位 0，又称保护模式使能位）来实现的。在此之前还要建立保护模式所必需的一些数据表，如全局描述符表 GDT、局部描述符表 LDT 和中断描述符表 IDT 等。

80386 支持分页机制，从而有效解决了内存碎片问题。80386 通过将物理内存地址通过“页目录”和“页表”映射成连续的线性地址，页表中除了映射信息外，还记录了页的访问属性等信息，以支持虚拟内存的实现，每个应用程序都有自己的 4GB 的寻址空间。可以用 CR0 寄存器中的位 31（PG 位）开启或关闭分页机制。

3）虚拟 8086 模式。虚拟 8086 模式是为了在保护模式下执行 8086 程序而设置的。虽然 80386 处理器提供了实模式来兼容 8086 程序，但实模式下的 8086 程序实际上只是运行得快了一点，对 CPU 的资源还是独占的。在保护模式的多任务环境下运行这些程序时，它们中的很多指令和保护模式环境格格不入，如段寻址方式、对中断的处理和 I/O 操作的特权问题等。

虚拟 8086 模式是以任务形式在保护模式上执行的，在 80386 上可以同时支持由多个真正的 80386 任务和虚拟 8086 模式构成的任务。在虚拟 8086 模式下，80386 支持任务切换和内存分页。在 Windows 操作系统中，有一部分程序专门用来管理虚拟 8086 模式的任务，称为虚拟 8086 管理程序。

虚拟 8086 模式实际上是以保护模式为基础，实模式和保护模式的混合。为了和 8086 程

序的寻址方式兼容，虚拟 8086 模式采用和 8086 一样的寻址方式，即用段寄存器乘以 16 作为段基址再配合偏移地址形成线性地址，寻址空间为 1MB。但显然多个虚拟 8086 任务不能同时使用同一物理位置的 1MB 地址空间，否则必然引发冲突。操作系统利用分页机制将不同虚拟 8086 任务的地址空间映射到不同的物理地址上去，这样每个虚拟 8086 任务看起来都认为自己在使用 0～1MB 的地址空间。

8086 代码中有相当一部分指令在保护模式下属于特权指令，如屏蔽中断的 CLI 和中断返回指令 IRET 等，它们是无法在保护模式下运行的。但这些指令在 8086 程序中是合法的。如果不让这些指令执行，8086 代码就无法工作。为了解决这个问题，虚拟 8086 管理程序采用模拟的方式来完成这些指令。这些特权指令执行的时候引发保护异常。虚拟 8086 管理程序在异常处理程序中检查产生异常的指令，如果是中断指令，则从虚拟 8086 任务的中断向量表中取出中断处理程序的入口地址，并将控制转移过去；如果是危及操作系统的指令，如 CLI 等，则简单地忽略这些指令，在异常处理程序返回的时候直接返回到下一条指令。通过这些措施，8086 程序既可以正常地运行下去，同时在执行这些特权指令的时候又觉察不到已经被虚拟 8086 管理程序执行了特殊处理。

由上可见，80386 处理器的 3 种工作模式各有特点且相互联系。实模式是 80386 处理器工作的基础，这时 80386 作为一个快速的 8086 处理器工作。在实模式下可以通过指令切换到保护模式，也可以从保护模式退回到实模式。虚拟 8086 模式则以保护模式为基础，在保护模式和虚拟 8086 模式之间可以互相切换，但不能从实模式直接进入虚拟 8086 模式或从虚拟 8086 模式直接退到实模式。

2. 动态链接库

动态链接库（Dynamic-Link Libraries，DLL）不能直接被执行，它们一般也不会接收消息。而只是一些包含着函数的独立文件，其中的函数可以被 Windows 程序或者其他 DLL 调用以完成某项任务。

“动态链接”是与“静态链接”相对而言的。“动态链接”是指 Windows 程序在运行时才把自己需要的存在于某个库中的函数链接进来。“静态链接”是指 Windows 程序在编译阶段就把各种对象模块（.OBJ）、运行时库（.LIB）和资源文件（.RES）链接到一起以创建一个可执行文件（.EXE）。

动态链接库标准的扩展名是.dll。具有标准扩展名的动态链接库模块才可以被 Windows 自动加载。而其他扩展名的动态链接库模块，必须使用 LoadLibrary 或 LoadLibraryEx 函数来显式加载。

某些动态链接库（如字体文件）称为 resource-only。它们只包括数据，而不包括代码。这些动态链接库为许多不同的程序提供资源。

动态链接库模块也可以作为一个单独的产品来发布。这样第三方程序开发人员就可以使用该动态链接库的模块来开发自己的应用程序，不但提高了程序的复用率，也节省了大量的时间和精力。

除了动态链接库之外，还有目标库（Object Libraries）和导入库（Import Libraries）。下面介绍这三种库的异同点。

目标库是扩展名为.lib 的文件，包括了用户程序要用到的各种函数。它在用户程序进行链接时，“静态链接”到可执行程序文件当中。例如，在 Visual C++中最常使用到的 C 运行时目标库文件就是 LIBC.LIB。

导入库是一种特殊形式的目标库文件形式。导入库文件的扩展名也是.LIB，在用户程序被链接时，被“静态链接”到可执行文件当中。但不同的是，导入库文件中并不包含有程序代码。它包含了相关的链接信息，帮助应用程序在可执行文件中建立起正确的对应于动态链接库的重定向表。比如 KERNEL32.LIB、USER32.LIB 和 GDI32.LIB 是我们常用到的导入库，通过它们，就可以调用 Windows 提供的函数了。如果程序中使用了 Rectangle 这个函数，GDI32.LIB 就可以告诉链接器，这个函数在 GDI32.DLL 动态链接库文件中。这样，当用户程序运行时，它就“动态链接”到 GDI32.DLL 模块中以使用这个函数。

目标库和导入库都是在程序开发过程中才用到的，而动态链接库是在程序运行时使用的。在程序运行时，相应的动态链接库文件必须已经存在于程序运行机器的硬盘上。

3. 指令集选择

在汇编程序设计中，当使用处理器特定的指令时，必须由相应的伪指令指定相应的处理器。汇编程序中说明处理器类型的伪指令及相应功能如下：

.8086：只支持对 8086 指令的汇编。

.186：只支持对 80186 指令的汇编。

.286：支持对 80286 指令的汇编（不包括特权指令）。

.286P：支持对 80286 所有指令的汇编。

.386：支持对 80386 指令的汇编（不包括特权指令）。

.386P：支持对 80386 所有指令的汇编。

.486：支持对 80486 指令的汇编（不包括特权指令）。

.486P：支持对 80486 所有指令的汇编。

.586：支持对 Pentium 指令的汇编（不包括特权指令）。

.586P：支持对 Pentium 所有指令的汇编。

.686：支持对 Pentium Pro 指令的汇编（不包括特权指令）。

.686P：支持对 Pentium Pro 所有指令的汇编。

只有用伪指令说明了处理器类型，汇编程序才知道如何更好去编译、连接程序，更好地去检错报错。

4. 函数的原型定义

对于程序中所有要用到的 API 函数，在程序的开始部分都必须预先声明，也就是进行函数的原型定义，包括函数的名称、参数的类型等。函数原型定义的作用是告诉编译器和连接器该函数的属性（即如上所说的函数名、参数个数及相应的类型），以便在编译和连接时编译器和连接器进行相关的类型检查。

函数原型定义的语法如下：

FunctionName PROTO [ParameterName]:DataType,[ParameterName]:DataType,…

通过在函数名后面加伪指令 PROTO，表明该语句是函数原型定义语句。函数参数列表紧跟在 PROTO 伪指令之后，参数名称和参数类型之间用冒号分隔。

比如经常在 Windows 应用程序中使用的消息框，其函数原型定义如下：

MessageBox PROTO hWnd:DWORD,lpText:DWORD,lpCaption:DWORD,uType:DWORD

其中，hWnd 是父窗口的句柄，lpText 是指向消息框要显示的文本的指针，lpCaption 是指向消息框标题文本串的指针，uType 是显示在对话框窗口上的小图标的类型。

Windows API 函数的原型声明已经写好，保存在 INC 格式的文件中，使用时，只要用伪

指令 INCLUDE 将其包含进来即可。比如以下包含语句：INCLUDE \MASM32\INCLUDE\KERNEL32.INC，编译器会打开文件夹\MASM32\INCLUDE 中的文件 KERNEL32.INC，并使用其中的函数声明执行编译过程。

5.4.4　Win32 汇编语言知识介绍

1. Windows 窗口知识介绍

在 Windows 系统中，窗口是指屏幕上的一块矩形区域。它可以从键盘或者鼠标接收用户的输入，并在其内部向用户显示输出信息。应用程序窗口通常包含标题栏、菜单栏、边框、滚动条和状态栏等。

窗口以“消息”的形式接收用户的输入，同时也可以通过消息与其他窗口通信。比如当用户通过最大化按钮改变窗口的大小时，操作系统会捕获用户发送的消息，然后将其转发给相应的应用程序，从而使程序能够响应这个系统功能，并调整窗口中的内容，以响应窗口大小的变化。

窗口是在“窗口类”的基础上创建的。Windows 定义了默认的窗口过程，如果所有的消息都让 Windows 自己处理，将能得到一个标准的窗口，当然，也可以选择处理自己感兴趣的消息，这样，相当于产生了不同的子类，也就形成了不同的应用程序。子窗口也是基于同一个窗口类，并且使用同一个窗口过程。

Windows 程序开始执行后，Windows 为该程序创建一个“消息队列”。这个消息队列用来存放该程序可能创建的各种不同窗口的消息。程序中有一段代码，叫做“消息循环”，其作用是从队列中取出消息，并且将这些消息发送给相应的窗口过程。

创建一个窗口的过程如下：

（1）取得程序的实例句柄（hInstance）。

（2）注册窗口类，实际上就是为所创建的窗口指定处理消息的过程，定义光标、窗口风格、颜色等参数。

（3）创建窗口。

（4）显示窗口。

（5）进入消息循环，也就是不停地检测有无消息，并把它发送给窗口进程去处理。

下面，来看一个创建窗口的简单程序。

2. 创建窗口程序的源代码

```
.386
.model flat, stdcall
option casemap :none
;-------------------------------------------------------------------------
; Include  文件定义
;-------------------------------------------------------------------------
include         ..\include\windows.inc
include         ..\include\user32.inc
include         ..\include\kernel32.inc
include         ..\include\comctl32.inc
include         ..\include\comdlg32.inc
include         ..\include\gdi32.inc
includelib      ..\lib\user32.lib
```

```
includelib      ..\lib\kernel32.lib
includelib      ..\lib\comctl32.lib
includelib      ..\lib\comdlg32.lib
includelib      ..\lib\gdi32.lib
;-------------------------------------------------------------------------
; 数据段
;-------------------------------------------------------------------------
.data
IDI_MAIN        equ     1000            ;icon
IDM_MAIN        equ     4000            ;menu
IDM_EXIT        equ     4001
hInstance       dd      ?
hWinMain        dd      ?
hMenu           dd      ?
hIcon           dd      ?
szBuffer        db      256 dup (?)
szClassName     db      "An Example of Window",0
szCaptionMain   db      '一个窗口的例子',0
;-------------------------------------------------------------------------
; 代码段
;-------------------------------------------------------------------------
 .code
 start:
                call    _WinMain
                invoke  ExitProcess,NULL

_WinMain        proc
                local   @stWcMain:WNDCLASSEX
                local   @stMsg:MSG
                invoke  InitCommonControls
                invoke  GetModuleHandle,NULL
                mov     hInstance,eax
                invoke  LoadIcon,hInstance,IDI_MAIN
                mov     hIcon,eax
                invoke  LoadMenu,hInstance,IDM_MAIN
                mov     hMenu,eax
;-------------------------------------------------------------------------
; 注册窗口类
;-------------------------------------------------------------------------
                invoke  LoadCursor,0,IDC_ARROW
                mov     @stWcMain.hCursor,eax
                mov     @stWcMain.cbSize,sizeof WNDCLASSEX
                mov     @stWcMain.hIconSm,0
                mov     @stWcMain.style,CS_HREDRAW or CS_VREDRAW
                mov     @stWcMain.lpfnWndProc,offset WndMainProc
                mov     @stWcMain.cbClsExtra,0
                mov     @stWcMain.cbWndExtra,0
                mov     eax,hInstance
                mov     @stWcMain.hInstance,eax
                mov     @stWcMain.hIcon,0
```

```
                    mov         @stWcMain.hbrBackground,COLOR_WINDOW + 1
                    mov         @stWcMain.lpszClassName,offset szClassName
                    mov         @stWcMain.lpszMenuName,0
                    invoke      RegisterClassEx,addr @stWcMain
;-------------------------------------------------------------------------
; 建立输出窗口
;-------------------------------------------------------------------------
                    invoke    CreateWindowEx,WS_EX_CLIENTEDGE,\
                              offset szClassName,offset szCaptionMain,\
                              WS_OVERLAPPEDWINDOW OR \
                              WS_VSCROLL OR WS_HSCROLL,\
                              0,0,550,300,\
                              NULL,hMenu,hInstance,NULL

                    invoke    ShowWindow,hWinMain,SW_SHOWNORMAL
                    invoke    UpdateWindow,hWinMain
;-------------------------------------------------------------------------
; 消息循环
;-------------------------------------------------------------------------
                    .while    TRUE
                              invoke    GetMessage,addr @stMsg,NULL,0,0
                              .break    .if eax == 0
                              invoke    TranslateMessage,addr @stMsg
                              invoke    DispatchMessage,addr @stMsg
                    .endw
                    ret
_WinMain            endp
;-------------------------------------------------------------------------
; _WinMain 结束
;-------------------------------------------------------------------------
WndMainProc         proc    uses ebx edi esi, \
                    hWnd:DWORD,uMsg:DWORD,wParam:DWORD,lParam:DWORD
                    mov     eax,uMsg
                    .if         eax == WM_CREATE
                          mov     eax,hWnd
                          mov     hWinMain,eax
                          call    _Init
                    .elseif   eax == WM_COMMAND
                        .if   lParam == 0
                              mov   eax,wParam
                                    .if   ax == IDM_EXIT
                                          call   _Quit
                                    .endif
                        .endif
                    .elseif   eax == WM_CLOSE
                              call   _Quit
                    .else
                              invoke    DefWindowProc,hWnd,uMsg,\
                              wParam,lParam
                              ret
.endif
xor    eax,eax
```

```
ret
WndMainProc    endp
;-------------------------------------------------------------------------
; WndMainProc 结束
;-------------------------------------------------------------------------
  _Init    proc
invoke    SendMessage,hWinMain,WM_SETICON,ICON_SMALL,hIcon
ret
_Init     endp
;-------------------------------------------------------------------------
; _Init 结束
;-------------------------------------------------------------------------
_Quit     proc

          invoke    DestroyWindow,hWinMain
          invoke    PostQuitMessage,NULL
          ret
_Quit    endp
;-------------------------------------------------------------------------
; _Quit 结束
;-------------------------------------------------------------------------
end    start
```

3. 代码分析

程序首先调用_WinMain，在_WinMain 中定义了两个局部变量：@stMsg 和@stWinMain，数据类型分别是 MSG 和 WNDCLASSEX 结构，WNDCLASSEX 结构定义了一个窗口的所有参数，包括所使用的菜单、光标、颜色、窗口过程等，接下来的一系列 mov 指令实际上就是在填写这个数据结构。mov @stWcMain.lpfnWndProc,offset WndMainProc 定义了处理消息的窗口过程，mov @stWcMain.lpszClassName,offset szClassName 定义了要创建的类的名称，然后使用 RegisterClassEx API 函数注册这个窗口类。此时窗口并没有创建，只是定义好了一个子类，接下来的工作是用刚才定义的类去创建一个窗口。这是通过 CreateWindowEx 函数实现的。在窗口创建好之后，应用程序的主要任务就是执行消息循环，捕获用户在窗口中发送的各种消息并进行处理，当用户单击关闭窗口按钮后，执行窗口的销毁操作，窗口生命周期结束。

5.5 汇编语言与高级语言的混合编程

与高级语言相比，汇编语言编写的程序目标代码占用存储空间小、运行效率高，它的运行速度是高级语言无法比拟的。但是如果全部采用汇编语言编程，则工作量又太大。高级语言采用结构化程序设计技术，代码简洁、开发效率高，但在有些要求由硬件直接进行操作的场合，或者要求执行速度很快的场合，仍然需要用汇编语言实现。为了更好地发挥高级语言和汇编语言各自的优点，将两者有机地结合起来，采用混合编程方法能更好地达到设计要求，完成设计功能。

5.5.1 汇编语言与 C/C++的混合编程

C/C++语言是一种被广泛使用的程序设计语言，它具有一些汇编语言的特点，如可以使用寄存器变量、可以进行位操作等。所以，C/C++语言与汇编语言程序之间能很平滑地衔接。另

外，目前主要的 C/C++语言程序开发环境，如 Turbo C/C++、Borland C/C++和 Visual Studio 2005（或 Visual Studio 2008）等，也提供了很好的混合编程手段。本节主要介绍汇编语言和 C 语言的混合编程方法。C/C++和汇编语言的混合编程有两种方式：一种是在 C/C++程序中嵌入汇编指令；另一种是将独立的汇编模块与 C/C++模块相连接。它们的编程规则有一定的区别。

1. 混合编程的基本规则

混合编程主要应解决两种语言的接口及参数传递等问题。

（1）参数传递规则。编译器的调用规范将影响函数命名、参数传递顺序、堆栈清理责任、参数传递机制等，常用的函数调用规范有_cdecl、_stdcall 和_pascal。C 语言默认的调用规范是_stdcall，即函数调用时通过堆栈传递参数，传递顺序是从右向左，由调用者恢复堆栈指针。如对于函数 Add(x,y)，函数被调用时，先将参数 y 压栈，再将参数 x 压栈，关于函数调用的详细讨论请参见“2．函数调用约定：_stdcall 和_cdecl”。

（2）返回值规则。C 语言函数调用返回后，根据返回值的长度不同，由不同的寄存器提供返回值。若返回值为 32 位，则返回值存放在 EAX 中；若返回值为 64 位，则将返回值存放在 EDX:EAX 中，其中 EDX 存放高 32 位，EAX 存放低 32 位。

（3）寄存器使用规则。对于寄存器 CS、EIP、SS、ESP、EDS、EBP、ESI 和 EDI，在函数中使用前应该先保存，使用完后再恢复它们的值。寄存器 EAX、EBX、ECX、EDX 和 ES 的值不需要保存，可以在程序中任意使用。

2. 函数调用约定：_stdcall 和_cdecl

函数调用约定规定了参数传递顺序、参数传递规则（传值还是传引用）、堆栈参数维护、函数名修饰。进行高级语言和汇编语言混合编程时，必须满足函数调用约定，否则无法连接。

Visual C++有两种常用的函数调用约定：_stdcall 和_cdecl，如表 5-6 所示。

表 5-6　函数调用约定

	_stdcall	_cdecl
参数传递顺序	从右向左	从左向右
参数传递规则	传值，除非传指针或引用	传值，除非传指针或引用
堆栈参数维护	被调用函数清除堆栈参数	调用函数清除堆栈参数
函数名修饰	编译器自定	编译器自定

从表 5-5 可以看出，_stdcall 和_cdecl 的一个重要区别在于：谁对堆栈参数的维护负责，是调用函数还是被调用函数负责清理堆栈。

_stdcall 约定被调用函数清除堆栈参数，这样做的好处是可以减少源代码的大小，因为每次函数调用完成后，调用函数不再需要清理堆栈的指令；但是有些函数必须使用_cdecl 调用约定，这种情况是参数个数未知的函数，这时只能由调用函数清除堆栈参数，比如库函数 printf。代码片段如表 5-7 和表 5-8 所示。

3. 使用内嵌汇编器进行混合编程

有时为了提高 C/C++语言源程序中某段程序的处理效率，可以在 C/C++程序中嵌入一段汇编程序。虽然这样做可以提高程序处理效率，但必须明确，这是以丧失程序的可移植性为代价的。因此，对 C 语言和汇编语言进行混合编程时，一定要仔细权衡采用该方法的利与弊。

表 5-7 _stdcall 调用代码

调用函数	被调用函数
…	Push ebp
Push param3	Mov ebp,esp
Push param2	…
Push param1	Mov esp,ebp
Call subproc	Pop ebp
…	Ret12
	（清除堆栈）

表 5-8 _cdec 调用代码

调用函数	被调用函数
…	Push ebp
Push param3	Mov ebp,esp
Push param2	…
Push param1	Mov esp,ebp
Call subproc	Pop ebp
Add esp,12	Ret
（清除堆栈）	
…	

在 C/C++语言中使用内嵌汇编指令不需要额外的编译器和连接器，而且可以访问在 C/C++中定义的变量，非常方便。在 C/C++中使用内嵌汇编是通过使用关键字_asm 来实现的。这个关键字有以下两种使用方法：

（1）使用_asm 语句块，即所有内嵌汇编指令用大括号括起来。例如，用下面的_asm 块可以输出字符“B”。

```
_asm{
    MOV   AH,2                ;显示单个字符的 DOS 调用
    MOV   DL,42H              ;输出字符“B”的 ASCII 码
    INT   21H
}
```

（2）在每条汇编指令之前加_asm关键字，这样，上面的例子可改为：

```
_asm    MOV    AH, 2
_asm    MOV    DL,42H
_asm    INT    21H
```

在第二种方法中，一行内可以包含多条汇编语句，语句之间只需空格即可，不需要任何间隔标志，而以_asm 作为下一条语句的开始。

虽然在_asm 块中允许使用汇编语言的大部分功能，但也有一些限制。内联汇编器不支持 MASM 的条件汇编、宏指令、结构和记录等高级汇编特性，也不能用 DB、DW、DD 等伪指令进行数据定义，而是由 C/C++进行数据分配。

内嵌汇编语句在 DOS 和 Win32 环境下的应用有一些区别，主要是 Win32 的应用程序中不能直接使用 INT 21H 的 DOS 功能调用，而是通过控制台命令或 C/C++提供的库函数来实现相应的功能。

下面通过两个例子对如何在 C/C++语言中嵌入汇编语言进行简单介绍。

例 5-39 若要在 C 语言源程序中嵌入汇编语言语句将整型变量 A 与 B 的值相乘，结果保存于整型变量 data1 和 data2 中，data2 存放乘积的高位部分，data1 存放乘积的低位部分。

解：

```
_asm {
    PUSH    EAX              //求整型变量 A 与 B 的乘积，结果存放于 data1 和 data2
    PUSH    EDX              //data2 存放乘积的高位部分，data1 存放乘积的低位部分
    XOR     EDX,EDX
    MOV     EAX, A
    IMUL    EAX, B
```

```
        MOV     data1,EAX
        MOV     data2,EDX
        POP     EDX
        POP     EAX
}
```

例5-40　从键盘输入0～1000之间的数，并分别以二进制、十进制和十六进制显示在屏幕上。

程序如下：

```
#include<conio.h>
#include<stdio.h>
//下面的函数是以指定的进制显示数据
void disp (int base,int data)
{
    int temp;
    _asm   {
            MOV     EAX,data
            MOV     EBX,base
            PUSH    EBX                 //保存基数，作为标志
    LOP1:   MOV     EDX,0               //进制转换
            DIV     EBX
            PUSH    EDX                 //存储余数
            CMP     EAX,0
            JNZ     LOP1
    LOP2:   POP     EDX                 //显示结果
            CMP     EDX,EBX             //判断是否是基数（栈底为基数）
            JE      EXIT0               //是基数，转移
            ADD     EDX,30H             //是数字位，转换为 ASCII 码
            CMP     EDX,39H             //大于数字'0'
            JBE     NEXT                //小于等于数字'9'，转移
            ADD     EDX,7               //大于'9'的字符转换为 ASCII 码
    NEXT:   MOV     temp,EDX
}
     printf("%c",temp);                 //显示当前字符
     _asm   JMP   LOP2
EXIT0:
     return;
}                                       //继续下一位
void main()
{
     char*info="Enter a number between 0 and 1000:";
     //提示信息
     char *bs="base";
     int a=0;                           //从键盘接收的数据
     int dbase[]={2,10,16};             //三种进制的基数
     int i=0;                           //循环变量
     printf("%s",info);                 //显示提示信息
     scanf("%d",&a);                    //接收键盘输入
     while(i<3)                         //用不同进制显示从键盘接收的数
     {
          printf("%s",bs);
          disp(10,dbase[i]);            //显示基数
```

```
            printf(":");
            disp(dbase[i],a);
            printf("\n");
            i++;
        }
}
```

在上述程序中，main 函数实现输入输出。函数 disp 将参数 data 指定的数以 base 指定的值为基数进行显示。其中，循环 LOP1 进行进制转换，并将转换后得到的每一位数压栈保存。循环 LOP2 则从堆栈中逐位取出，转换为 ASCII 码并显示在屏幕上。

在函数 disp 中，多次出现内嵌汇编语句直接访问用 C 语言定义的数据的情形，这正是混合编程的灵活之处。但在应用时需要注意数据长度的一致性。

本小节通过具体的例子分析了汇编函数和高级语言函数间通过内嵌汇编代码方式进行混合调用的情况，下面通过由 C/C++和汇编语言编译器分别生成目标文件，然后进行连接，生成可执行程序，进一步讨论 C/C++和汇编语言进行混合编程的问题。

4. 将汇编源程序和 C/C++源程序分别编译然后对目标文件连接

如前所述，内嵌汇编器有一定的限制，不能充分利用 MASM 的高级宏汇编功能。在某些情况下，可开发独立的汇编语言模块，然后与 C/C++模块连接，生成一个独立的可执行程序，这种方法更加灵活。

下面的实例介绍了在 C/C++中调用外部汇编代码的过程。

例 5-41 主函数用 C 实现，从键盘接收两个整数，并作为参数传递给汇编过程，汇编过程实现两个数相加的功能，程序如下：

```
#include<stdio.h>
extern "C" int plus(int,int);                    //声明为外部函数
void main()
{
    int x,y;
    printf("Enter two numbers:");
    scanf("%d","%d",&x,&y);                      //接收键盘输入
    printf("x+y=%d",plus(x,y));
}
```

上述程序的 extern 语句说明 plus 是一个外部过程（在本例中是用汇编语言编写的），本程序中将会使用该函数。

注意：过程名（函数名）是大小写敏感的，应该保证它在汇编程序和 C 程序模块中一致。在 extern 命令后面的“C”用于告诉编译器该外部函数遵循 C 语言的命名约定，编译时不加任何修饰。

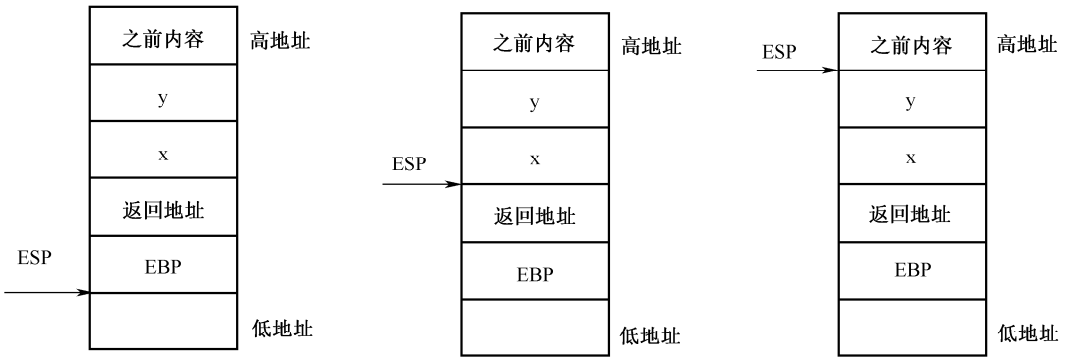
用汇编程序实现加法功能，程序如下：

```
. 386                                            ;386 指令集
. MODEL   FLAT,C                                 ;内存使用模式为 FLAT
. CODE
PUBLIC    PLUS
PLUS      PROC                                   ;过程定义
          PUSH    EBP
          MOV     EBP,ESP
          MOV     EAX,[EBP+8]                    ;取参数
          MOV     EBX,[EBP+12]                   ;取参数
```

```
        DD      EAX,EBX
        POP     EBP
        RET
PLUS    ENDP
END
```

在上述汇编程序中，语句.386 指示汇编器选择实模式下的 386 指令系统，若没有该语句，则汇编程序采用默认的 8086 指令系统。.MODEL 语句告诉汇编程序使用何种内存模式。MODEL 语句中的“C”指示汇编程序，该程序中的过程将被 C 语言程序调用。第 4 行的 PUBLIC 语句将 PLUS 过程声明为全局过程。

在 32 位应用程序中，C/C++与汇编过程传递参数时，除数组参数外，所有参数都采用传值的方式，并且都扩展为 32 位。数组参数传递的是 32 位的数组首地址偏移量。在 PLUS 过程中，通过寄存器 EBP 来访问参数。在使用 EBP 前，先将 EBP 压栈，以保证使用前后一致。在 EBP 压栈后，堆栈中的数据如图 5-9(a)所示。这时，参数 x 和参数 y 的偏移量分别是[EBP+8]和[EBP+12]。两数相加的结果通过寄存器 EAX 返回到主调函数。执行 RET 指令后，ESP 即指向参数 x，如图 5-9（b）所示，这时的参数 x 和 y 已经不再有用，应该及时清除这两个参数。这里没有使用带参数的 RET 指令，是因为 C 语言的调用规范中规定由主函数自动调整堆栈指针，因此函数返回到 main 之后，堆栈指针恢复到调用函数之前，如图 5-9（c）所示。



（a）EBP 压栈后　　（b）执行 RET 后　　（c）返回到 main 函数后

图 5-9　堆栈变化示意图

假设 C 语言程序的名字是 exam.c，汇编源程序的名字是 plus.asm。完成了 C 程序和汇编源程序的编写后，将 exam.c 编译为 exam.obj，将 plus.asm 汇编为 plus.obj。在 C 中建立项目文件，将 C 程序和汇编后的目标代码都加入到工程项目文件中，然后进行连接生成可执行文件。

这里的汇编程序必须是 MASM 6.11 以上版本，汇编程序名是 ML.EXE，使用格式为：

```
ml /c /coff plus.asm
```

这里的开关/c 指示汇编程序只进行汇编，不进行连接；开关/coff 指示汇编程序生成 COFF（Common Object File Format）格式的文件。这两个参数是必需的，且参数只能出现在文件名的左边，否则，参数将被忽略。

5.5.2　MASM 32 汇编与连接命令

MASM 32 是自由软件开发者推出的基于 32 位操作系统的宏汇编程序开发环境，它包括宏编译器（ML.EXE）、连接器（LINK.EXE）和资源编译器（RC.EXE）。编译器 ML.EXE 的作

用是把源程序编译成二进制文件，与宏汇编程序 MASM 的作用是一样的。LINK.EXE 的作用是把各个模块连接起来，并完成再定位的工作，对于 COFF 格式的 LIB 文件中的代码和数据也要连接进来（如 MASM32.LIB）。RE.EXE 的作用是把资源编译成二进制 RES 文件。连接时将调用 cvtres.exe 将 RES 文件转换为 COFF 格式，组成一个应用程序。

1. 编译和连接

MASM 32 提供了集成开发环境，即集编辑、编译、连接于一体，集成编辑器的执行文件为 QEditor.exe。但建议不要使用集成环境进行源程序编辑。

Win32 汇编源程序的编辑与在 DOS 下完全相同，任何文本编辑器都可以以纯文本方式保存，扩展名为.asm，如 main.asm。可以直接在 DOS 命令提示符下编译和连接。

（1）编译格式：

ml[/options]filelist[/link linkoptions]

该编译格式中的选项很多，常用的选项及其含义如表 5-9 所示。

表 5-9 ml 的常用选项

选项	含义
/c	只编译不连接
/coff	生成 COFF 格式的目标文件
/Cp	源程序区分大小写
/Fo filename	指定输出的 obj 文件名
/Fe filename	指定连接后输出的 exe 文件名
/I pathname	指定 include 文件的路径
/link 选项	指定连接时使用的选项
/Zi （调试程序常用）	增加符号调试信息

假设编辑好的源程序文件为 main.asm，只进行编译不连接，生成 COFF 格式的目标文件，区分源程序中的大小写，输出调试符号信息，则编译命令为：

ml /c /coff /Cp main.asm

如果源程序中没有任何语法错误，则生成 main.obj 目标文件。

（2）连接格式：

link[/options][filelist]

link 命令可以和 ml 命令一起使用，也可以单独使用。link 的常用选项及含义如表 5-10 所示。

表 5-10 link 的常用选项

选项	含义
/debug	在执行文件中加入调试信息
/libpath：路径	指定库文件的目录
/out：文件名	指定输出文件名，默认的扩展名是.exe，如果要生成其他文件，如屏幕保护 *.scr 等，则在这里指定
/subsystem：系统名	指定程序运行的操作系统，可以是 Windows 或其他
/stack：尺寸	设定堆栈尺寸

续表

选项	含义
/driver：类型	连接 Windows 的 WDM 驱动程序时使用
/dll	连接动态链接库文件时使用
/def：文件名	编写链接库文件时使用的 def 文件名，用来指定要导出的函数列表

例如：

```
link　/subsystem:windows　/libpath:lib　main.obj
```

（3）如果汇编程序中有资源脚本文件（*.rc），需要用 rc.exe 编译成二进制文件后方可连接。例如：

```
rc scheme.rc
```

连接二进制资源文件格式如下：

```
Link /subsystem：windows　/libpath：lib　main.obj　scheme. res
```

注意：运行此命令需要 cvtres.exe 在同一目录下。

如果编译、连接参数较多，可以使用批命令处理。如不带资源文件的批处理文件为 corn_link.bat，语句如下：

```
ml /c /coff /Cp %1.asm
link / subsystem：windows　/libpath：lib%l
```

在 DOS 提示符下，执行批命令，如 corn_link　main，即可对 main.asm 进行编译和连接。

此外，MASM 32 提供 nmake.exe，默认操作文件为 makefile，无扩展名。makefile 文件内容如下：

```
NAME=源程序名
$(NAME).exe：$(NAME). obj：$(NAME).res
Link /SUBSYSTEM：Windows/LIBPATH：c：\masm \iib $(NAME). obj $(NAME). res
$(NAME). res：$(NAME).rc
rc $(NAME).rc
$(NAME).obj：$(NAME). asm
Ml /c /coff /Cp $(NAME).asm
```

执行 nmake.exe 文件，将进行编译、资源编译、连接，生成与源程序同名的可执行文件。另外，nmake 只对修改过的文件进行重新操作。要注意的是，无论用批处理还是 nmake，都必须保证编译、连接等文件路径的有效性，否则将无法编译、连接。

2. 调试

Win32 汇编语言程序开发，需要额外选择调试工具。Numega 公司的 SoftICE 是目前比较流行的一种调试工具，用它来调试 Win32 汇编程序十分方便。SoftICE 既可用于调试各种运行于特权级 3 的应用程序，也可用于调试运行于特权级 0 的设备驱动程序，甚至可以调试 Windows 操作系统本身。由于 SoftICE 运行于硬件与 Windows 之间，因此 SoftICE 的用户界面与 Windows 系统相互独立。当按下其激活热键 Ctrl+D 后，Windows 桌面隐藏，SoftICE 界面弹出，这时可对用户程序进行调试。

注意：如果要对 Win32 源程序进行调试，可执行文件（.exe）中必须含有调试信息。因此必须使用带/Zi 选项的 ml.exe 对源程序进行汇编，并用带/DEBUG 选项的 link.exe 对目标程序进行连接。

一、选择题

1．在汇编语言程序设计中，伪指令 OFFSET 表达的含义是回送变量或标号的（　　）。

A．段地址值　　B．偏移地址值

C．物理地址值　　D．操作数

2．DEC BYTE PTR[BX]中的操作数的数据类型是（　　）。

A．字　　B．双字　　C．字节　　D．四字

3．将数据 5618H 存放在存储单元中的伪指令是（　　）。

A．DATA1 DW 1856H　　B．DATA1 DB 18H,56H

C．DATA1 EQU 5618H　　D．DATA1 DB 18H,00H,56H,00H

4．要在程序中定义缓冲区 BUF，保留 9 个字节存储空间的语句是（　　）。

A．BUF DW 9　　B．BUF DB 9

C．BUF DB 9 DUP(?)　　D．BUF DW 9 DUP(?)

5．有定义 NUM　DB　'12345'，汇编后，NUM 占有（　　）字节存储单元。

A．1　　B．5　　C．6　　D．7

6．BUF　DW　10H　DUP(3 DUP(2,10H),3,5)汇编后，为变量 BUF 分配的存储单元字节数是（　　）。

A．80H　　B．100H　　C．124　　D．192

7．若定义 DAT DW 'A'，则 DAT 和 DAT+1 两个相邻的内存单元中存放的数据是（　　）。

A．00H41H

B．41H00H

C．XXH41H

D．41HXXH（选项 C、D 中的 XX 表示任意数据）

8．已定义数据段：

```
DATA    SEGMENT
ORG     0213H
DA1     DB   15H,34H,55H
ADR     DW   DA1
DATA    ENDS
```

能使AX中数据为偶数的语句是（　　）。

A．MOV　AX,WORD PTR DA1　　B．MOV　AL,DA1+2

C．MOV　AL,BYTE PTR ADR+1　　D．MOV　AX,WORD PTR DA1+2

9．下列指令序列执行后完成的运算，正确的算术表达式应是（　　）。

```
MOV   AL,BYTE PTR  X
SHL   AL,1
DEC   AL
MOV   BYTE PTR Y,AL
```

A．Y=X*2+1　　B．X=Y*2+1

C．X=Y*2-1　　D．Y=X*2-1

10．对于下列程序段：

```
AGAIN:   MOV AL,[SI]
         MOV ES:[DI],AL
         INC SI
         INC DI
         LOOP AGAIN
```

也可用（　　）指令完成同样的功能。

A．REP MOVSB　　B．REP LODSB

C．REP STOSB　　D．REPE SCASB

二、简答题

1．写出完成下列要求的变量定义语句：

（1）在变量 var1 中保存 3 个字变量：1234H、-10、80/6。

（2）在变量 var2 中保存字符串'BYTE'。

（3）在缓冲区 buf1 中留出 10 个字节的存储空间。

（4）在变量 var3 中保存缓冲区 buf1 的长度。

（5）在缓冲区 buf2 中，保存 2 个字节的 22H，再保存 3 个字节的 33，并将这一过程重复 5 次。

（6）在变量 pointer 中保存变量 var1 和缓冲区 buf1 的偏移地址。

2．已知一数据段中的数据如下，画出该数据段数据存储分配图：

```
DATA     SEGMENT
         BUF   DB   'AB',0DH,0AH
         M     DB   2   DUP(1),2   DUP(2,'B')
         VAR4  DW   'AB', 'CD'
         ORG   100H
         TABLE DB   10,3*4,10H
         ADDRESS DD TABLE
DATA     ENDS
```

3．试分析下列程序段的功能：

```
CMP   AL, 'A'
JC    OTHER
CMP   AL, 'Z'+1
JNC   OTHER
JMP   LETTER
…
OTHER:
      …
LETTER:
      …
```

4．分析下列子程序 FUNC1，并回答相应的问题。

```
FUNC1  PROC    NEAR
       XOR     CX,CX
       MOV     DX,01
       MOV     CL,X
       JCXZ    A20
```

```
        INC     DX
        INC     DX
        DEC     CX
        JCXZ    A20
A10:    MOV     AX,02
        SHL     AX,CL
        ADD     DX,AX
        LOOP    A10
A20:    MOV     Y,DX
        RET
FUNC1   ENDP
```

若该子程序的入口参数为 X（0≤X≤10），其输出参数为 Y，则：

（1）该子程序的功能是 Y=f(X)=_____。

（2）若 X=0，则 Y=_____；若 X=3，则 Y=_____；若 X=5，则 Y=_____。

5．x86 系列 CPU 有哪几种工作模式，Win32 汇编程序工作于其中哪种模式？该模式有哪些特点？

6．Win32 汇编程序设计具有哪些不同于 DOS 汇编的特点？

7．一个完整的 Win32 汇编程序包括哪些部分，这些部分的作用是什么？

8．动态链接库是否可以直接执行？它和静态链接库有什么区别？

9．请查阅相关资料，进一步深入了解 Win32 编程中的消息处理机制，并简要叙述之。

10．C/C++和汇编混合编程时应考虑哪些原则？在进行混合编程时有哪些不利因素应注意避免？

11．函数调用规范主要规定哪些方面的内容？常用的函数调用规范有哪些？请查阅相关资料，简要叙述它们之间的主要区别。

三、程序填空题

1．以 BUF 为首址的字节单元中，存放了 COUNT 个无符号数，以下程序段是找出其中最大的数并送入 MAX 中。

```
        BUF   DB   5,6,7,58H,62,45H,127
        COUNT EQU  $-BUF                    ;COUNT 等于变量 BUF 的字节数
        MAX   DB ?
        …
        MOV   BX,OFFSET BUF
        MOV   CX,COUNT-1
        MOV   AL,[BX]
LOP1:   INC   BX
        _______
        JAE   NEXT
        MOV   AL,[BX]
NEXT:   DEC   CX
        _______
        MOV   MAX,AL
```

2．DA1 数据区中有 50 个字节数据，下面程序段将每个数据的低四位变反，高四位不变，并依次存入 DA2 数据区中。请将程序补充完整。

```
        MOV   CX,50
        LEA   SI,DA1
        LEA   DI,DA2
K1:     MOV   AL,[SI]
        ________
        MOV   [DI],AL
        INC   SI
        INC   DI
        DEC   CX
        ________
```

3．在数据段 ADDR1 地址处有 200 个字节数据，要传送到数据段地址 ADDR2 处。

```
MOV   AX, SEG ADDR1
MOV   DS, AX
MOV   ES, ________
MOV   SI, ________
MOV   DI, OFFSET ADDR2
MOV   ________ , 200
CLD
REP   ________
```

4．下列子程序统计字符串的长度，入口参数 DI 为字符串首地址，字符串以 0DH 结束；返回参数 CX 中为统计得到的字符串长度。

```
STRLEN   PROC
         PUSH   AX
         PUSH   DI
         MOV    CX,0
REPEAT:  MOV    AL,[DI]
         ________
         JE   EXIT
         ________
         INC   DI
         ________
EXIT:    POP DI
         ________
         RET
STRLEN   ENDP
```

四、编程题

1．试编程求解表达式 S=(23000-(X*Y+Z))/Z，其中 X=600，Y=25，Z=-2000。

2．试编写一个汇编语言程序，要求对键盘输入的小写字母用大写字母显示出来。

3．编写一个程序段，其功能为：将两个四字节无符号数 X、Y 相加，结果存入 RESULT 中。

4．根据用户输入的数字（0～6），显示星期（0：SUNDAY、1：MONDAY、2：TUESDAY、3：WEDNESDAY、4：THURSDAY、5：FRIDAY、6：SATURDAY），试用地址表法编写程序。

5．统计 AX 寄存器中为 1 位数的，并将统计结果放在 CL 寄存器中。

6．编写程序，比较两个字符串 MSG1 和 MSG2 所含字符是否完全相同，若相同则显示“EQUAL”，若不同则显示“DIFFERENT”。

7．定义一条宏指令，实现将指定数据段的段地址传送到段寄存器 ES 或 DS 的功能。

8．从键盘上读入一个正整数 N（0≤N≤65535），存入 AX，再转换成十六进制数并在屏幕上显示出来。要求分别定义两个子过程 D_B_I 和 B_H_O，前者用来实现以十进制形式输入数据，后者用来实现以十六进制形式显示数据，然后在主程序中调用这两个子过程。

9．存储器数据段从 BUF 开始存放一个字数组，数组中第一个字中存放该数组的长度 N，编制一个程序使此数组中的数据按照从小到大的次序排列。

10. 在单独的模块中编写一远程过程：SEARCH，完成在一个字节数组中查找给定的字节，如找到则将其在数组中的下标（即数组中的偏移量）返回给变量 VAR1；如没有找到则给变量 VAR1 返回–1。变量 VAR1 和数组均为外部标识符（可在主程序模块中定义）。请同时写出调用此过程的主程序模块。