第四章　汇编语言程序设计基础

本章主要内容：

- 汇编语言基本语法
- 常用伪指令介绍
- 汇编语言源程序的基本框架
- 基于MS-Windows的输入输出编程

- 不同的汇编程序有不同的汇编语言编程规定。
- 目前支持Intel x86系列微机常用的汇编程序有ASM、MASM、TASM、OPTASM等 。
- 同一种汇编程序的不同版本之间也有一定的差异。
- 本章主要介绍基于MASM的汇编语言的一些基本语法规则以及源程序的基本结构。

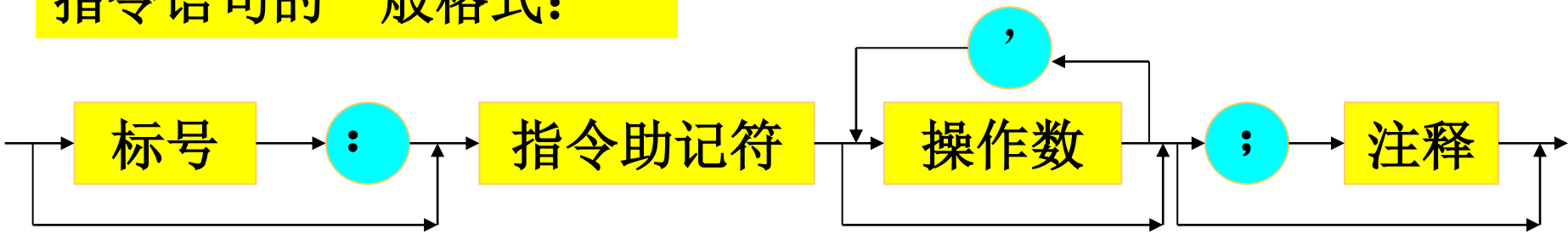
4.1 汇编语言语句种类及其格式

汇编语言语句 { 指令语句
伪指令语句

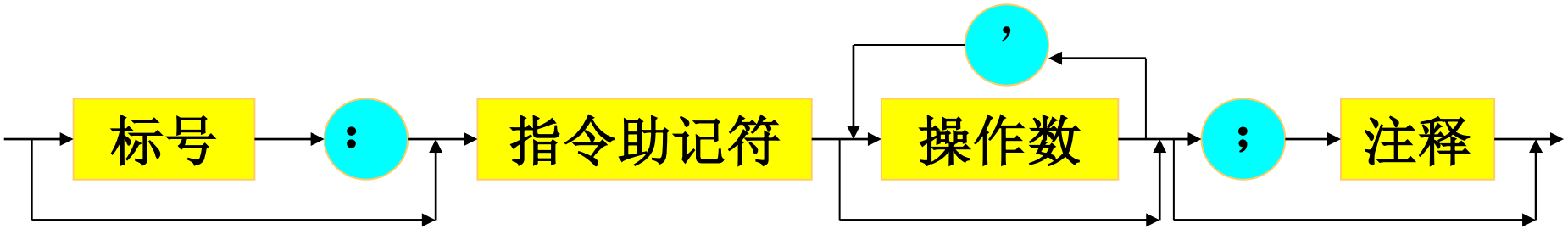
一、指令语句

每一条指令语句在汇编时都要产生一个可供CPU执行的机器目标代码，它又叫可执行语句。

指令语句的一般格式：



一条指令语句最多可以包含4个字段



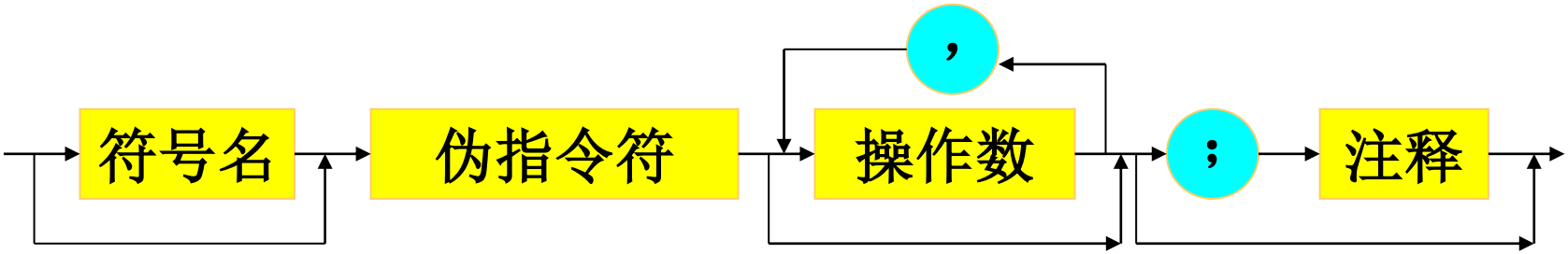
- 指令助记符和操作数两个字段就是前面介绍的指令
- 标号是可选字段，后面必须跟“：”。主要用于控制程序执行顺序。
- 注释字段为可选项，该字段以分号“;”开始。
  - ✓ 它不会产生机器目标代码，不影响程序的功能。
  - ✓ 注释可以加在指令的后面，也可以是整个语句行。

例：LABEL1：ADD AX，BX；功能为AX<=(AX)+(BX)
;后面的程序段将完成一次对存储器的访问
……

二、伪指令语句

- 伪指令语句又叫命令语句，是指示性语句。
- 伪指令本身不产生自己的机器目标代码。它指示汇编程序对其后面的指令语句和伪指令语句如何处理。

一条伪指令语句可以包含四个字段：



- 符号名、伪指令符和操作数这三个字段就构成了伪指令，这是本章后面介绍的主要内容之一。
- 注释字段与指令语句相同。

- 除了前面介绍的指令语句和伪指令语句中的注释方式之外，还可以定义块注释。
- 块注释用COMMENT伪指令定义

格式：

```
COMMENT  自定义符号
    被作为注释内容的多个文本行
自定义符号
```

- 汇编器汇编源程序时，将忽略COMMENT后面的所有文本行，直到出现自定义符号。
- 自定义符号可以任意选择，但它不能出现在注释文本行中。

例如：

```
COMMENT  @
    这是注释块的示例
    注释行中不能出现@，其他任何符号都可以出现。
@
```

三、标识符

指令语句中的标号和伪指令语句中的符号名统称为标识符。

标识符构成规则：

- 1.字符的个数为1~240个；
- 2.可以用字母、数字、下划线及符号@、$和?；
- 3.第一个字符不能是数字；
- 4.不能使用系统专用的保留字(关键字key words)。

保留字：

- CPU中各寄存器名（如AX、CS等）
- 指令助记符（如MOV、ADD等）
- 伪指令符（如SEGMENT、DB、BYTE等）
- 表达式中的运算符（如GE、EQ等）以及属性操作符（如PTR、OFFSET等）
- 表示高级语言的C、FORTRAN、BASIC等

汇编语言程序中的关键字不区分大小写。

常用的保留字

$	PARITY?	DWORD	STDCALL
?	PASCAL	FAR	SWORD
@B	QWORD	FAR16	SYSCALL
@F	REAL4	FORTRAN	TBYTE
ADDR	REAL8	FWORD	VARARG
BASIC	REAL10	NEAR	WORD
BYTE	SBYTE	NEAR16	ZERO?
C	SDORD	OVERFLOW?	
CARRY?	SIGN?		

4.2 汇编语言数据

- 数据：作为指令和伪指令语句中的操作数
- 常用的数据形式有：常数、变量和标号。
- 一个数据由数值和属性（比如是字节数据还是字数据）两部分构成。

一、常数

常数：经过汇编后其值已完全确定，并且在程序运行过程中，其值不会发生变化。

常数的整数表示形式可以是：二进制数、八进制数、十进制数和十六进制数等

常数还可以有以下两种表示形式

- 实数：

±整数部分• 小数部分E ±指数部分

尾数

例 2.134 E +10

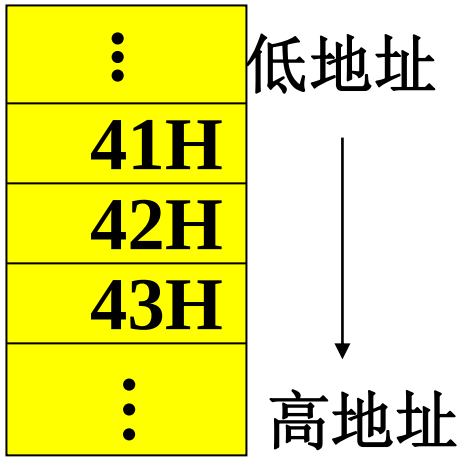
3.14

0.75

汇编器在汇编源程序时，可以把实数转换为4字节、8字节或10字节的二进制数形式存放。

- 字符串：用引号（单引号或双引号）括起来的一个或多个字符，其值为这些字符的ASCII码值。

例如，字符串`ABC`的值为41H 42H 43H。在内存中的存储如图所示。



常数在程序中的使用有以下三种：

（1）作指令语句的源操作数

```
MOV   AX，0B2F0H
ADD   AH，64H
```

（2）在指令语句的直接寻址、寄存器相对寻址或基址变址寻址方式中作位移量。

```
MOV   BX，[32H] ;此用法只能在实模式下使用
MOV   0ABH [EBX]，CX
ADC   DX，1234H [EBP][EDI*2]
```

（3）在数据定义伪指令中使用

```
DB   10H
DW   3210H
```

二、变量

变量：用来表示存放数据的存储单元，这些数据在程序运行期间可以被改变 。

程序中以变量名的形式来访问变量。变量名就是存放数据的存储单元地址。

- 定义变量：给变量在内存中分配一定的存储单元。也就是给这个存储单元赋与一个符号名--变量名.
- 定义变量的同时一般还要给分配的存储单元预置初置。

变量分为 { 全局变量 / 局部变量

1. 全局变量的定义

全局变量的作用域是整个程序，全局变量定义在 数据段内。

全局变量定义伪指令格式：

变量名　类型　表达式1,表达式2，……

MASM中可以使用的类型很多，具体如下表所示。

类型	缩写	长度（字节）	说明
byte	db	1	字节变量
word	dw	2	字变量
dword	dd	4	双字（doubleword）变量
fword	df	6	三字（farword）变量
qword	dq	8	四字（quadword）变量
tbyte	dt	10	十字节BCD码（tenbyte）变量
sbyte		1	有符号字节（signbyte）变量
sword		2	有符号字（signword）变量
sdword		4	有符号双字（signdword）变量
real4		4	单精度浮点数变量，初始化要用实数
real8		8	双精度浮点数变量，同上
real10		10	10字节浮点数变量，同上

其中表达式1、表达式2……是给存储单元赋的初值。

例如：
```
.DATA
DATA1   BYTE   12H
DATA2   DB     20H,30H
DATA3   WORD   5678H
```

当变量被定义后，就具有了以下三个属性：

（1）段属性(SEGMENT)

它表示变量存放在哪一个逻辑段中

例如上面例子中的变量DATA1、DATA2和DATA3三个变量都存放在数据段中。

（2）偏移量属性（OFFSET）

表示变量所在位置与段起始点之间的字节数

上例中，变量DATA1的偏移量为0，DATA2为1，DATA3为3。

在平坦模式，由于各个逻辑段的起始地址重合，各逻辑段的数据起始地址由系统指定，因此不能用这样的方法确定。

段属性和偏移量属性就构造了变量的逻辑地址

（3）类型属性

表示变量占用存储单元的字节数

例如

- **BYTE指令定义的变量为1字节**
- **WORD定义的变量为字（2字节）**
- **DWORD定义的为双字（4字节）**
- **DQ定义的为4字（8字节）**
- **DT定义的为5字（10字节）**

在变量定义语句中给变量赋初值的方式

（1）数值表达式

例如：

DAT1 DB 32，30H

; DAT1的内容为32（20H），DAT1+1单元内容为30H.

DAT2 SBYTE -20 ;DAT2的内容为-20

（2）？表达式：表示不特意预置初值

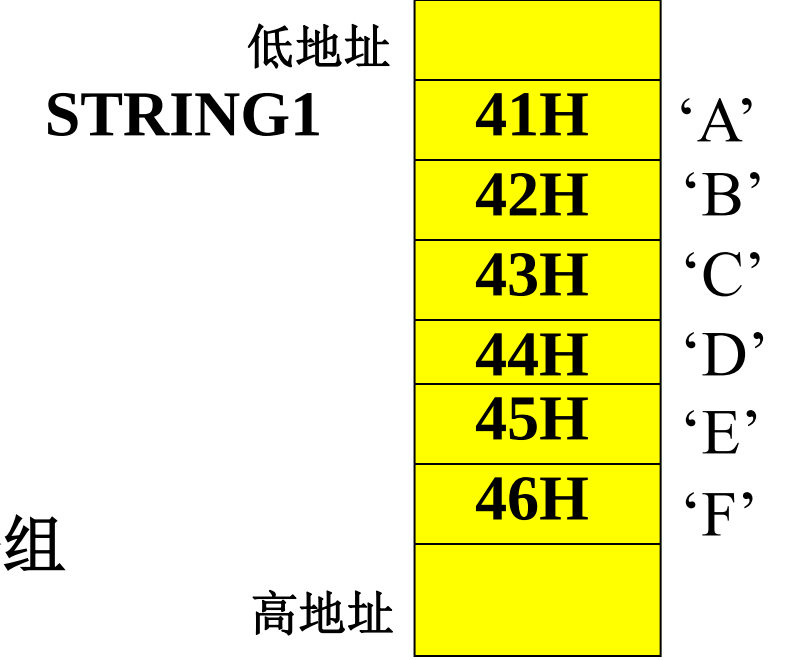
例如：**DA-BYTE DB ？，？，？**

表示让汇编程序分配三个字节存储单元。这些存储单元在使用之前没有赋值，则**它们的值为0—若初值为0都可用？**。

（3）字符串表达式

- 对于**DB**伪指令：
  - 字符串用引号括起来，长度不超过255个字符。
  - 字符串的每个字符分配一个字节单元，从左到右将各字符的**ASCII**码以地址递增的顺序依次存放。

例如：STRING1 DB ‘ABCDEF’

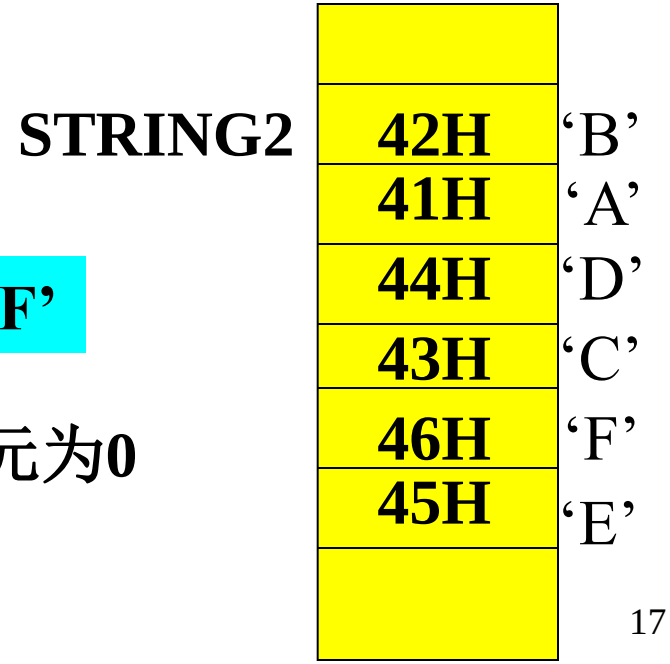


- 对于DW伪指令：给不超过两个字符组成的字符串分配两个字节存储单元。

注意：

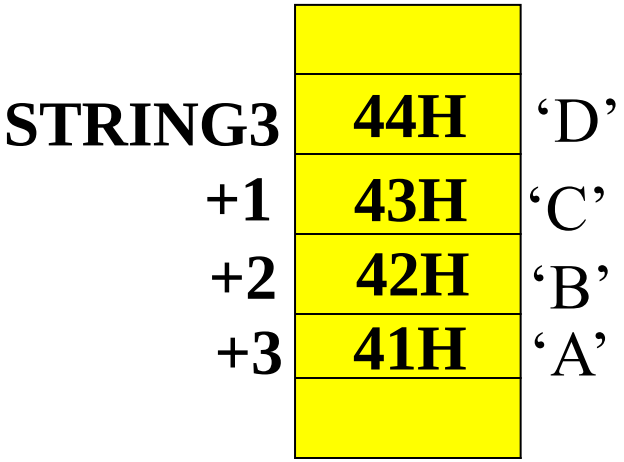
- 两个字符的存放顺序是前一个字符放在高地址，后一字符放低地址单元。
- 超过两个字符就会出错！

例如：STRING2 DW ‘AB’, ‘CD’, ’EF’



- 若字符串只有一个字符则高字节地址单元为0

- 对于DD伪指令
  - 给不超过四个字符组成的字符串分配4个字节单元。
  - 存放顺序与DW伪指令相同。
  - 如果字符数少于4，则较高地址的字节单元存0。



例如：STRING3 DD ‘ABCD’

- 对于类型更长的df、dq伪指令也只能使用不超过4个字符的字符串进行初始化，超过4个字符则出错。

注意：1.对于有符号定义伪指令sbyte、sword和sdword也可以使用字符串初始化，方法同上。

2.在实模式下只有db、dw和dd这三个伪指令，且dw和dd伪指令不能用超过两个字符的字符串赋初值，否则将出错。

（4）DUP表达式

DUP称为重复数据操作符。

使用DUP表达式的一般格式：

变量名 { DB / DW / DD / … } 表达式1 DUP (表达式2)

其中：表达式1是重复的次数，表达式2是重复的内容。

例如：DATA_A DB 10H DUP(?)

DATA_B DB 20H DUP('AB')

分配16个字节单元

分配20H*2=40H个 字节，其内容为重复字符串 'AB'。

DUP还可以嵌套使用：表达式2又可以是一个带DUP的表达式。

例如：DATA_C　DB　10H　DUP(4　DUP(2),7)

重复10H个数字序列“2，2，2，2，7”，共占用10H*5=50H个字节。

2.变量的使用

（1）在指令语句中引用

在指令语句中直接引用变量名就是对其存储单元的内容进行存取，也可以在变量外加中括号。

例如：
```
DA1   DB   0FEH
DA2   DW   52ACH

 DA3  DW   0
……
MOV  AL,DA1   ;将0FEH传送到AL中
MOV  BX,[DA2]   ;将52ACH传送到BX中
MOV  DA3,BX   ;将DA3单元赋值52ACH
```

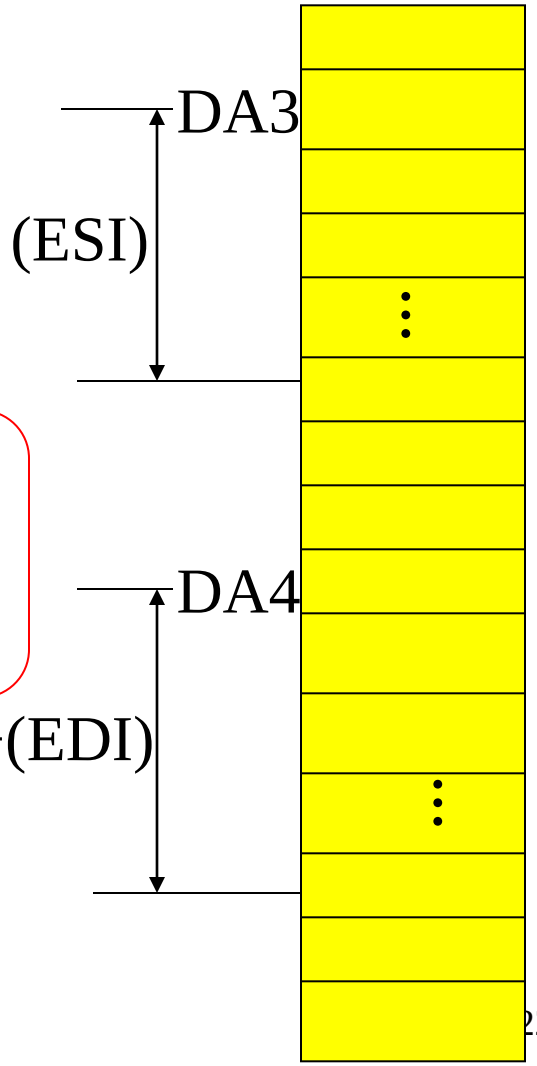
当变量出现在寄存器相对寻址或基址变址相对寻址的操作数中时，表示取用该变量的偏移量。

例如：

```
DA3  DB  10H  DUP(?)
DA4  DW  10H  DUP（1）
MOV  DA3[ESI]，AL
ADD  DX，DA4[EBX][EDI]
```

将从DA4开始再偏移(EBX)+(EDI)的字存储单元的内容与DX的内容相加，结果送回DX中。

将AL的内容送入从DA3 开始再偏移(ESI)的存储单元中



（2）在DD和DQ伪指令语句中引用

```
NUM     DB   75H
ADR1    DD   NUM
ADR2    DW   NUM
ADR3    DQ   NUM
```

表示取变 量的32位偏移地址

出错！

取变量的段基地址和偏移地址

- 注意：只能在DD和DQ伪指令语句中使用变量来初始化，其他变量定义伪指令中会出错。

2. 局部变量的定义

要让变量的作用域限制在某过程（子程序）中，则把它定义为局部变量。在MASM中使用local伪指令定义局部变量。

格式： local 变量名1[[重复数量]][:类型],变量名2[[重复数量]][:类型]……

- LOCAL语句要紧跟在过程定义PROC语句之后
- 默认的类型是dword，因此定义dword类型的局部变量，则类型可以省略。
- 当定义数组的时候，元素个数用[] 括号括起来。
- 局部变量不能和已定义的全局变量同名。
- 局部变量的作用域是当前的子程序，所以在不同的子程序中可以有同名的局部变量。
- 局部变量的初值是随机的，是其他子程序执行后在堆栈里留下的垃圾，所以使用时一定要对局部变量赋值。

```
例1 Local var1[1024]:byte ;定义了一个1024字节长的局部变量1var1
例2 Local var3,var4:byte ;定义了dword型局部变量var3,byte型局部变量var4
```

三、标号

- 标号加在一条指令的前面，它就是该指令在内存的存放地址的符号表示，也就是指令地址的别名。
- 标号主要用在程序中需要改变程序的执行顺序时，用来标记转移的目的地。

```
例如：        MOV   CX, 100
      LAB:  MOV  AX, BX
            ……
            LOOP  LAB
            JNE    NEXT   ;不为零转移
            ……
      NEXT: ……
```

- 上面方法所加标号属于局部标号，只能在所在的过程中被引用。
- 若标号名后加上双冒号，则该标号在整个程序中都可引用。

例如：Glob_LAB::MOV BX, AX

每个标号具有三属性

（1）段属性（SEGMENT）

它表示该标号所代表的地址在哪个逻辑段中。

（2）偏移量属性（OFFSET）

它表示该标号所代表的地址在段内与段起点间的字节数，即地址的偏移量。

（3）距离属性（也叫类型属性）

它表示该标号可以被段内还是段间的指令调用。

NEAR（近）： NEAR标号只能作段内转移，即只能是与该标号所指指令同在一个逻辑段的其它指令才能使用它。

FAR（远）： FAR标号可以被非本段的转移和调用指令使用。

标号的距离属性可以有两种方法来指定

a.隐含方式

当标号加在指令语句前面时，它隐含为NEAR属性。

例　SUB1：MOV AX，30H

SUB1的距离属性为NEAR，它只能被本段的转移指令和调用指令访问。

b.用LABEL伪指令给标号指定距离属性

格式：　标号名 LABEL 类型

类型为NEAR或FAR。该语句要与指令语句连用。

例如：SUB1_FAR　LABEL　FAR
　　SUB1：MOV　AX,30H
　　……

- SUB1_FAR与SUB1两个标号具有相同的逻辑地址。被转移指令或调用指令使用时是指同一个入口地址。
- SUB1只能被本段调用，SUB1-FAR可以被其它段的指令调用。

LABEL伪指令还可用来定义变量的属性，即改变一个变量的属性，如把字变量的高低字节作为字节变量来处理。

例如：DATA_BYTE　LABEL　BYTE
　　DATA_WORD　DW　20H　DUP（？）

- DATA_BYTE与DATA_WORD具有相同的段基址和偏移量。
- DATA_BYTE可以被用来存取一个字节数据，而DATA_WORD则不能。

注意：用LABEL定义的标号或变量的地址一定是相邻的下一指令语句或伪指令定义的变量。

4.3 符号定义语句

符号定义语句将常数或表达式等形式用某个指定的符号来表示。在80x86汇编语言中有两种符号定义语句。

一、等值语句

格式：　符号名　EQU　表达式

功能：用符号名来表示EQU右边的表达式。后面的程序中一旦出现该符号名，汇编程序将把它替换成该表达式。

表达式可以是任何形式，常见的有以下几种情况。

1.常数或数值表达式

例如　COUNT　EQU　5
　　NUM　　EQU　COUNT+5

2.地址表达式

```
ADR1  EQU  DS：[EBP+14]
```

ADR1被定义为在DS数据段中以BP作基址寻址的一个存储单元。

3.字符串

```
STR1 EQU “ABCD”
STR2  BYTE   STR1 ;STR1为符号常量,STR2为变量。
```

4.变量名、寄存器名或指令助记符

例如：CREG EQU CX；在后面的程序使用CREG就是使用CX

CBD EQU DAA；DAA为十进制调整指令。

注意：在同一源程序中，同一符号不能用EQU定义多次。

例：
```
CBD  EQU  DAA
CBD  EQU  ADD
```
} 错误用法

二、等号语句

格式：符号名=表达式

等号语句与等值语句具有相同的作用。但等号语句可以对一个符号进行多次重复定义。

例如：

CONT=5

NUM=14H

NUM=NUM+10H

下面是错误用法：

CBD=DAA

……

CBD=ADD

等号语句不能为助记符定义别名！

注意：等值语句与等号语句定义的符号都不会在内存分配存储单元。所定义的符号没有段、偏离量和类型等属性。

4.4 表达式与运算符

表达式是指令或伪指令语句操作数的常见形式。它由常数、变量、标号等通过操作运算符连接而成。

注意：任何表达式的值在程序被汇编的过程中进行计算确定，而不是到程序运行时才计算。

8086/8088宏汇编语言中的操作运算符非常丰富，可以分为以下五类。

一、算术运算符

+、—、＊、 / 、MOD、SHL、SHR、[ ]

1.运算符“+”和“-”也可作单目运算符，表示数的正负。

2.使用“+”、“-”、“＊”、和“/”运算符时，参加运算的数和运算结果都是整数。

3.“/”运算为取商的整数部分，而“MOD”运算取除法运算的余数。

例如：

```
NUM=15＊8          ；NUM=120
NUM=NUM/7          ；NUM=17
NUM=NUM MOD 3      ；NUM=2
NUM=NUM+5          ；NUM=7
NUM=-NUM-3         ；NUM=-10
NUM=-NUM-NUM       ；NUM=20
```

4. “SHR ”和 “ SHL ”为逻辑移位运算符

“SHR”为右移，左边移出来的空位用0补入。
“ SHL”为左移，右边移出来的空位用0补入。

注意：移位运算符与移位指令区别！

- 移位运算符的操作对象是某一具体的数（常数），在汇编时完成移位操作。
- 而移位指令是对一个寄存器或存储单元内容在程序运行时执行移位操作。

例如

```
NUM=11011011B
…….
MOV   AX , NUM SHL  1
MOV   BX ,  NUM SHR 2
ADD   DX ,  NUM SHR 6
```

能否改成指令：
SHL NUM,1

不能！

上面的指令序列等效下面三条指令。

移位运算将对象按照与AX等长的16bit数处理

```
MOV   AX ，110110110B
MOV   BX ，  00110110B
ADD   DX ，  3
```

5.下标运算符“[ ]”具有相加的作用

一般使用格式：　表达式1 [表达式2]

作用：将表达式1与表达式2的值相加后形成一个存储器操作数的地址。

下面两个语句是等效的。

```
MOV   AX，DA_WORD[20H]
MOV   AX，DA_WORD+20H
```

可以用寄存器来存放下标变量

例：下面几个语句是等价的

```
MOV  AX , ARRAY[EBX][ESI]; 基址变址寻址
MOV   AX,  ARRAY[EBX+ESI]
MOV   AX , [ARRAY+EBX][ESI]
MOV   AX , [ARRAY+ESI][EBX]
MOV   AX , [ARRAY+EBX+ESI]
MOV   AX,  [ARRAY][EBX][ESI]
```

下面是几个错误的语句：

```
MOV   AX，ARRAY+EBX+ESI
MOV   AX，ARRAY+EBX[ESI]
MOV   AX，ARRAY+DA_WORD
MOV   AX,  [EBX+ESI]DA_WORD
MOV   AX,  [EBX][ESI]DA_WORD
```

二、逻辑运算符

逻辑运算符有NOT、AND、OR和XOR等四个，它们执行的都是按位逻辑运算。

例如

```
MOV  AX，NOT 0F0H          =>MOV  AX ,0FF0FH
MOV   AL，NOT 0F0H         =>MOV  AL , 0FH
MOV   BL，55H AND 0F0H     =>MOV BL,50H
MOV   BH，55H OR 0F0H      =>MOV BH, 0F5H
MOV   CL，  55H XOR 0F0H   =>MOV CL, 0A5H
```

三、关系运算符

关系运算符包括：EQ（等于）、NE（不等于）、LT（小于）、 LE（小于等于）、GT（大于）、 GE（大于等于）

- 关系运算符用来比较两个表达式的大小。比较的两个表达式必须同为常数或同一逻辑段中的变量。
- 若是常量的比较，则按无符号数进行比较；若是变量的比较，则比较它们的偏移量的大小。
- 关系运算的结果只能是“真”（全1）或“假”（全0）

例1：

```
MOV   AX，0FH  EQ  1111B      =>MOV  AX ,0FFFFH
MOV   BX , 0FH  NE  1111B     =>MOV   BX, 0
```

例2

```
VAR    DW    NUM  LT  0ABH
```

该语句在汇编时，根据符号常量NUM的大小来决定VAR存储单元的值，当NUM<0ABH时，则变量VAR的内容为0FFFFH，否则VAR的内容为0。

四、数值返回运算符

该类运算符有5个，它们将变量或标号的某些特征值或存储单元地址的一部分提取出来。

1.SEG运算符

作用： 取变量或标号所在段的段基址。

该运算符为16位机或实模式下可用，在32位模式中不能使用！

例如：

```
DATA  SEGMENT
  K1   DW  1，2
  K2   DW  3，4
       ……
MOV   AX , SEG   K1
MOV   BX , SEG  K2
```

设DATA逻辑段的段基址为1FFEH，则两条传送指令将被汇编为：

```
MOV   AX ，1FFEH
MOV    BX ，1FFEH
```

2.OFFSET运算符

该运算符的作用是取变量或标号在段内的偏移地址。

注意：16位系统中偏移量为16位，32位系统为32位！

例：

```
;16位模式下的例程
DATA SEGMENT
VAR1  DB   20H DUP(0)
VAR2  DW  5A49H
ADDR  DW  VAR2  ;将VAR2的偏移量20H存入ADDR中
      …….
      MOV  BX，VAR2; (BX)=5A49H
      MOV   SI , OFFSET  VAR2 ;(SI)=20H
      MOV   DI, ADDR   ;DI的内容与SI相同
      MOV   BP，OFFSET  ADDR  ;(BP)=22H
```

获取偏移量还可以用什么方法？ 指令LEA

该程序能在32位flat模式运行吗？ 不能！

OFFSET运算符获取的偏移量为dword类型，只能存入32位寄存器。

这种用法：MOV EBP, OFFSET ADDR[ESI]是否正确？

错误！为什么？

怎样才能取到这个偏移量?

LEA

注意LEA和OFFSET在使用上的区别！

3.TYPE运算符

作用:取变量或标号的类型属性，并用数字形式表示。对变量来说就是取它的字节长度。

变量
- BYTE 1
- WORD 2
- DWORD 4
- QWORD 8
- ……

标号
- NEAR -1
- FAR -2

例如：

```
V1   DB   'ABCDE'
V2   DW   1234H, 5678H
V3   DD   V2
     ……
     MOV  AL，TYPE  V1
     MOV  CL，TYPE  V2
     MOV  CH，TYPE  V3
```

经汇编后的等效指令序列如下：

```
MOV  AL，1
MOV  CL，2
MOV  CH，4
```

4.LENGTH与LENGTHOF运算符

这两个运算符用于取变量定义时初始值个数

- **Length运算符**
  - 取DUP说明的重复次数,并且只取最外层DUP的重复次数。
  - 如果第一个初始值不是DUP说明，则LENGTH运算符返回值总是1。
- **Lengthof运算符—只能用于32位模式**
  - 取变量定义行中的初始值个数

例如

```
K1  DB   10H DUP(0), 20H
K2  DB    10H，20H，30H，40H
      DB   50H, 60H
K3  DW  20H DUP(0,1,2 DUP(0))
K4  DB   ‘ABCDEFGH’
```

```
MOV   AL,  LENGTH   K1; (AL)=10H
MOV   AH, LENGTHOF K1;(AH)=11H
MOV   BL,  LENGTH   K2 ; (BL)=1
MOV   BH, LENGTHOF K2;(BH)=4
MOV   CL,  LENGTH   K3 ;  (CL)=20H
MOV   CH, LENGTHOF K3; (CH)=4*20H=80H
MOV   DL,  LENGTH   K4 ;  (DL)=1
MOV   DH, LENGTHOF K4 ;(DH)=8
```

5.SIZE与SIZEOF运算符

- SIZE取值等于LENGTH和TYPE两个运算符返回值的乘积。
- SIZEOF取值等于LENGTHOF和TYPE返回值的乘积。

例如，对于上面例子，加上以下指令：

```
MOV   AL，SIZE  K1；  （AL）=10H
MOV   AH,  SIZEOF K1; (AH)=11H
MOV   BL，SIZE  K2；  （BL）=1
MOV   BH,  SIZEOF K2;  (BH)=4
MOV   CL，SIZE  K3；  （CL）=2*20H=40H
MOV   CH, SIZEOF K3;  (CH)=2*80H=A0H
MOV   DL，SIZE  K4；  （DL）=1
MOV   DH,  SIZEOF K4;  (DH)=8
```

五、属性修改运算符

这一类运算符用来对变量、标号或存储器操作数的类型属性进行修改或指定。

1.PTR运算符

格式：　类型　PTR　地址表达式

作用：将地址表达式所指定的标号、变量或用其它形式表示的存储器地址的类型属性修改为“类型”所指的值。

- 类型可以是BYTE、WORD、DWORD、NEAR和FAR等等。
- 这种修改是临时的，只在该运算符所在的语句内有效。

例如：
```
DA_BYTE   DB   20H  DUP(0)
DA_WORD  DW   30H  DUP(0)
    …….
MOV   AX , WORD  PTR  DA_BYTE[10]
ADD    BYTE   PTR   DA_WORD[20], BL
INC    BYTE  PTR  [EBX]
SUB    WORD  PTR  [ESI], 100
JMP    FAR   PTR  SUB1;指明SUB1不是本段中的地址
```

2.HIGH/LOW运算符

格式：

```
LOW  表达式
HIGH  表达式
```

- 表达式被汇编计算为一个常数，这两个运算符分别取常数的最低8位和其后的8位。

例如：

```
DATA   SEGMENT
CONST  EQU   0ABCDH

VAR1 WORD 1234H

DATA   ENDS
      …….
MOV   AH ,HIGH  CONST
MOV   AL, LOW  CONST
```

两条传送语句等效为：

```
MOV   AH ，  0ABH
MOV   AL ，  0CDH
```

注意：HIGH/LOW运算符不能用来分离一个变量、寄存器或存储器单元的内容。

下面语句中的用法是错误的。

```
MOV   AH，HIGH  VAR1
MOV   BH，  LOW  AX
```

六、运算符的优先级

优先级别	运算符
（最高）1	LENGTH,LENGTHOF,SIZE,SIZEOF, 圆括号
2	PTR，OFFSET，TYPE，THIS
3	HIGH，LOW
4	*，/，MOD，SHR，SHL
5	+，-
6	EQ，NE，LT，LE，GT，GE
7	NOT
8	AND
（最低）9	OR，XOR

汇编程序在计算表达式时的处理规则：

- 先执行优先级别高的运算，再算较低级别运算；
- 相同优先级别的操作，按照在表达式中的顺序，从左到右进行；
- 可以用圆括号改变运算的顺序。

例如：**K1= 10 OR 5 AND 1 ；结果为K1=11**

K2=（10 OR 5） AND 1 ；结果为K2=1

4.5 过程定义伪指令（PROC/ENDP)

- 在程序设计过程中，常常将具有一定功能的程序段设计成一个过程，过程也称为子程序。
- 汇编语言程序的主程序也要定义为过程，程序运行时操作系统调用这个过程。
- 在MASM宏汇编语言中，用过程(PROCEDURE)来构造子程序。

过程定义伪指令格式：

过程名 **PROC** [距离][语言类型][可视区域][USES 寄存器列表][,参数列表]...

LOCAL 局部变量列表

指令语句

ret

过程名 **ENDP**

PROC和**ENDP**伪指令定义了过程开始和结束的位置， **PROC**后面跟的参数是过程的属性和输入参数。

- 过程名是子程序的名称，用CALL指令或INVOKE伪指令来调用该过程。
- 距离可以是NEAR、FAR、NEAR16、NEAR32、FAR16或FAR32。Win32中只有一个平坦的段，段基地址都相同，可忽略该项。
- 语言类型表示参数的进栈顺序和堆栈恢复的方式，可以是StdCall，C，SysCall，BASIC、FORTRAN和PASCAL，如果忽略，则使用程序头部 .model定义的值。
- 可视区域可以是PRIVATE、PUBLIC和EXPORT。
  - PRIVATE表示过程只对本模块可见
  - PUBLIC表示对所有的模块可见
  - EXPORT表示是导出的函数，当编写DLL的时候要将某个函数导出的时候可以这样使用。
  - 默认设置是PUBLIC
- USES寄存器列表--用于保存执行环境，寄存器间用空格分隔。
  - 编译器在过程的功能语句开始前自动安排push指令将这些寄存器压栈，并且在ret指令前自动安排pop指令出栈。

例如：
```
Exam1 PROC USES EAX ECX
        ;过程的功能语句
        ret
    Exam1 ENDP
```
编译器生成代码：
```
Exam1 PROC USES EAX ECX
        PUSH EAX
        PUSH ECX
        ;过程的功能语句
        POP ECX
        POP EAX
        ret
    Exam1 ENDP
```

- 若过程使用寄存器返回参数，则USES后寄存器列表中不能包含返回参数的寄存器，否则会破坏返回值。
- 也可以在过程开头和结尾用pushad和popad指令一次保存和恢复所有寄存器，来代替此属性功能。

- 参数列表——用来定义过程使用的参数名和类型。
  - 格式：参数1:类型,参数2:类型,…
  - 若指定了参数，则过程要用invoke伪指令调用，否则应在CALL指令调用之前将参数按照规定的顺序压栈；
  - 若过程未指定参数则可用CALL指令，也可以用invoke调用。
  - 参数名不能与全局变量和过程中的局部变量重名。
  - 对于类型，由于**Win32**中的参数类型只有**32**位（**dword**）一种类型，所以可以省略。
  - 在参数定义的最后还可以跟**VARARG**，表示在已确定的参数后还可以跟多个数量不确定的参数。
- 在过程中一般最后执行的是ret指令，它控制程序返回调用该过程的上一级程序。
- 如果是用户主程序，由于最后要执行返回操作系统的API函数ExitProcess，故不需要ret指令。

4.6 win32汇编语言源程序的结构

- 早期的8086/8088为了实现数据和代码的安全，将内存按逻辑段进行管理和使用，不同的逻辑段用来存放不同目的的内容。
- 在IA-32中提供了保护模式，数据和代码的安全性得到了保障，在程序设计时仍然延续了分段思想，但是对逻辑段的管理和使用有了很大的变化。段的定义也大大简化。
- 我们将只介绍32位模式下的段结构定义，有关16位模式的介绍请参考相应的PPT资料。

先来看一个加法程序

```
; 三个数据相加程序
        .386
        .model flat,stdcall
        ExitProcess PROTO, dwExitCode:DWORD
        .data
        firstval     DWORD 20002000h
        secondval DWORD 11111111h
        thirdval    DWORD 22222222h
        sum          DWORD 0
        .code
        main PROC
            mov eax, firstval
            add  eax,secondval
            add  eax,thirdval
            mov sum,eax
            INVOKE ExitProcess,0
       main   ENDP
       END   main
```

该程序实现对内存中三个数据相加，然后再存入内存。

Win32汇编语言源程序一般框架

```
.386
.MODEL flat,stdcall
OPTION casemap:none
;   <一些include语句>
INCLUDE user32.inc
INCLUDE kernel32.inc
INCLUDELIB user32.lib
INCLUDELIB kernel32.lib
.STACK [堆栈段的大小]
.DATA
;   <一些初始化过的变量定义>
.DATA?
;   <一些没有初始化过的变量定义>
.CONST
;   <一些常量定义>
.CODE
main PROC
;       <其他语句>
mian ENDP
END main
```

一、模式与格式定义

程序的开始部分是模式和源程序格式的定义语句：

```
.386
.MODEL flat,stdcall
OPTION casemap:none
```

.386----该伪指令用来指定允许的指令集

可以是指定的指令集定义伪指令有：

.8086: 只允许8086/8088指令集。现在的VS环境下已不支持。

.386: 允许80386及之前的非特权指令集，不支持之后新增的指令

.386P: 允许80386及之前的全部指令（含特权指令），不支持之后新增的指令。

.586:允许Pentium及之前的非特权指令集，不支持之后新增的指令

.586P:允许Pentium及之前的全部指令集，不支持之后新增的指令

.mmx:扩充了MMX指令集

……

◆模式定义语句

.MODEL 内存模式[，语言模式][，其他模式]

内存模式	内 存 使 用 方 式
tiny	用来建立 .com文件，所有的代码、数据和堆栈都在同一个64 KB段内
small	建立代码和数据分别用一个64 KB段的 .exe文件
medium	代码段可以有多个64 KB段，数据段只有一个64 KB段
compact	代码段只有一个64 KB段，数据段可以有多个64 KB段
large	代码段和数据段都可以有多个64 KB段
huge	同large，并且数据段中的一个数组也可以超过64 KB。
flat	Win32使用的模式，代码和数据段使用同一个4 GB段。32位偏移地址。

说明：前六种是在实模式下的内存模式，偏移地址都为16位。保护模式下只有flat模式，偏移地址为32位，只能使用32位寄存器来指示地址。

例如：`.model flat,stdcall`

- 将编程模式设为平面存储模式，所有的段都使用同一个4GB内存，即各个段都从0开始，程序中将不出现段寄存器。
- 指定语言模式，即子程序的调用方式，Windows的API调用使用的是stdcall格式，所以在Win32汇编中必须在 .model中加上stdcall参数。

◆ OPTION语句

用**option**语句定义的选项有很多，在**Win32**汇编程序中，只需做如下定义即可。

option casemap:none

- 该语句设定了程序中的变量、符号常量和过程名对大小写敏感，即区分大小写。
- 由于**Win32 API**中的**API**名称是区分大小写的，所以必须指定这个选项，否则在调用**API**的时候会有问题。
- **Option**语句不设置对关键字（如指令助记符、伪指令符、寄存器名等）的大小写敏感，即关键字是不区分大小写的。

二、包含头文件和库文件

- 用来包含所使用到的系统定义的头文件和库文件；
- **Win32的系统功能模块放在Windows的库文件中，这些功能模块就以API的形式提供给用户调用。**

Win32 API的核心由3个库文件提供：

- **KERNEL32.LIB——系统服务功能。包括内存管理、任务管理和动态链接等。**
- **GDI32.LIB——图形设备接口。利用VGA与DRV之类的显示设备驱动程序完成显示文本和矩形等功能。**
- **USER32.LIB——用户接口服务。建立窗口和传送消息等。**

- 包含库文件使用伪指令**INCLUDELIB**

例如： **INCLUDELIB kernel32.lib**

- 当用户程序中要调用API函数时，需先用伪指令PROTO声明函数原型，否则编译器不能识别该函数。
- 格式：过程名 PROTO, 参数列表

  参数列表为：参数1:类型,参数2:类型……

  例如,前面的三个数相加程序中语句：

  ExitProcess PROTO, dwExitCode:DWORD

  API函数ExitProcess的功能是从用户程序退出，返回操作系统。申明它有一个参数dwExitCode，类型为DWORD.
- 为简化编程就把这些声明集中放在系统对应的头文件中。
- 每个库文件都有相应的头文件。
- 包含头文件使用伪指令INCLUDE

  例如 **INCLUDE kernel32.inc**

三、段的定义

- **.stack、.data、.data?、.const和 .code是分段伪指令**
- **Win32中只有代码和数据之分，.data、.data?和 .const都是数据段，.code是代码段。**
- **.data中一般放有初始化值的变量定义，所分配的存储单元被赋与初值，程序执行时可能会改变。**
- **.data?中放的是不需要初始化的变量定义，如定义缓冲区。这些变量也可以定义在.data中，但生成最终执行文件的大小有较大的差别。**
- **两种数据段分配的存储区（以4KB页为单位）属性都是可读可写。**
- **.const用于常量的定义，比如显示的字符串信息等。这种数据段分配的存储区属性为可读不可写。**

- 与DOS汇编语言不同，Win32汇编语言可以不考虑堆栈定义，操作系统会为程序分配一个向下扩展的、足够大的段作为堆栈段，所以 .stack段定义常常被忽略。
- . code段是代码段，所有的指令都必须写在代码段中，对于工作在特权级3的应用程序，该存储区属性为可读、不可写、可执行。
- Win32环境下的“段”实际上并不是DOS汇编语言那种意义的段，而是内存的“分段”。
- 上一个段的结束就是下一个段的开始，所有的“分段”合起来，包括系统使用的地址空间，就组成了整个可以寻址的4 GB空间。

四、调用Windows的API函数的伪指令invoke

把系统头文件和动态链接库包含到文件后就可以调用API函数

格式：

invoke 函数名[，参数1][，参数2]……

例如，调用API函数MessageBox 显示信息：

```
invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK
```

调用Windows的API函数ExitProcess，将控制权返回操作系统。

```
invoke ExitProcess,NULL
```

五、程序结束和程序入口

- 不像C语言程序有一个main函数，在汇编源程序中，程序员可以指定从代码段的任何一个地方开始执行。
- 程序的最后一句一定是伪指令end语句

  end　[开始地址]
- 该语句指定了开始执行的指令，同时还表示源程序结束，即所有的代码必须在end语句之前。
- 在第一条执行语句的前面要加上一个标号，以便在END语句中使用。一般主程序习惯使用标号main。

例如：end main　;main是在最开始语句前加的标号。

4.7 当前位置计数器$与定位伪指令ORG(Origin)

汇编程序在汇编源程序时，每遇到一个逻辑段，就要为其设置一个位置计数器，用来记录该逻辑段中定义的每一个数据或每一条指令在逻辑段中的相对位置。

- 在源程序中使用符号$来表示“位置计数器”的当前值。$也被称为当前位置计数器。
- $位于不同的位置具有不同的值。
- $在使用上完全类似变量的使用

定位伪指令**ORG**——用来改变位置计数器的值。

格式：**ORG 数值表达式**

作用：将数值表达式的值赋给当前位置计数器$ 。ORG语句为其后的变量或指令设置起始偏移量。

表达式的值必须为正值。表达式中也可以包含有当前位置计数器的现行值$。

若运行环境为flat模式，则该伪指令并不能把一个具体的偏移值赋值赋给位置计算器，系统会自动为每个段指定一个起始偏移地址，数值表达式的值为相对该起始偏移地址。

例：

```
.DATA
        ORG  30H
   DB1   DB  12H,34H
        ORG  $+20H
  STRING  DB  'ABCDEFGHI'
  COUNT   EQU   $-STRING
  DB2  DWORD  $
  DB3  DB   $
.CODE
       ORG  12H
  START  PROC
        ......
   START  ENDP
            END START
```

从数据段开始偏移30H字节存DB1

保留20H个字节单元，其后再存放'ABCD....

计算STRING的长度

取$的偏移量,类似变量的用法

此语句错误!

代码段开始留12H字节单元

4.8 基于MS-Windows的输入输出编程

- 若要实现用键盘、鼠标、显示器、磁盘等进行输入输出信息，在早期的DOS系统下一般是调用DOS提供的系统功能来实现。
- 在Win32系统下可以通过调用操作系统提供的API来实现。
- Win32提供的API非常丰富，它是Win32 Platform SDK中的一部分，覆盖了所有的输入输出功能，但使用上稍显繁琐。
- 美国的Irvine教授在Win32 SDK的基础上，整理和编写了一个API库—Irvine32.LIB。
- 下面介绍几个常用的简单的控制台I/O过程，完整的详细介绍可参考教材《汇编语言基于x86处理器》的第5章5.4节。

- 前面我们已经学习过在自己的程序中如何调用某个函数库的过程，有三个步骤：
  ① 在程序的开始用**PROTO**申明该过程的原型，也可以用**INCLUDE**将该函数库对应的头文件**(.INC)**包含进来；
  ② 用**INCLUDELIB**将库文件（或库导入文件）**(.LIB)**包含进来；
  ③ 在代码段中使用**CALL**指令或**INVOKE**伪指令调用过程。
- **Irvine32**中定义的过程不带参数，使用寄存器传参数，用**CALL**调用。

例1：编程实现从键盘输入4位16进制数，然后以二进制形式显示输出。

注意： Irvine32头文件中已经包含了**user32.inc**和**kernel32.inc**等系统头文件，不能重复包含，否则将发生符号定义冲突错误。

```
.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO,dwExitCode:DWORD
INCLUDE Irvine32.inc
INCLUDELIB Irvine32.lib
.code
main PROC
        call ReadHex     ;输入16进制数
        call WriteBin    ;输出二进制数
        INVOKE ExitProcess, 0
main ENDP
END  main
```

1. ReadChar

功能：从键盘读取一个字符， AL 寄存器为返回的字符ASCII码，字符不在控制台窗口中回显。

例如：

```
.data
char BYTE ?
.code
call ReadChar
mov char,al
```

如果用户按下的是扩展键，如功能键、方向键、Ins 键或 Del 键，则AL=0，而 AH 中是键盘扫描码。EAX 的高字没有使用。

下面的伪代码描述了调用 ReadChar 之后可能产生的结果：

```
if an extended key was pressed
   AL = 0
   AH = keyboard scan code
else
   AL = ASCII key value
endif
```

2. ReadDec

功能：从键盘读取一个 32 位无符号十进制整数，并用 EAX 返回该值，前导空格被忽略。

◆ 返回值为遇到第一个非数字字符之前的所有有效数字。

比如，用户输入123BAH，则 EAX 中的返回值为 123。

例如：

```
.data
intVal DWORD ?
.code
call ReadDec
mov intVal, eax
```

ReadDec 会影响进位标志位：

- **如果输入整数为空(NULL)，则 EAX=0 且 CF=1**
- **如果输入只有空格，则 EAX=0 且 CF=1**
- **如果输入整数大于（2^{32}-1），则 EAX=0 且 CF=1**
- **其他情况，EAX 为转换后的数，且 CF=0**

3. ReadFromFile

功能：读取存储缓冲区中的一个输入磁盘文件。

- 调用时用 EAX 传递打开文件的句柄，用 EDX 传递缓冲区的偏移量，用 ECX 传递读取的最大字节数。
- 返回时查看进位标志位的值即可知道输入是否正确：
  - CF=0，则 EAX 包含了从文件中读取的字节数；
  - CF=1，则 出错，EAX 包含了错误代码。可调用 WriteWindowsMsg过程显示该错误的文本信息。

例如，从文件读取 5000 个字节并复制到缓冲区变量中

```
.data
BUFFER_SIZE = 5000
buffer BYTE BUFFER_SIZE DUP(?)
bytesRead DWORD ?
.code
mov edx,OFFSET buffer  ;指向缓冲区
mov ecx,BUFFER_SIZE    ;读取的最大字节数
call ReadFromFile      ; 读文件
jc next1
mov bytesRead, eax ;实际读取的字节数
jmp next2
next1:call WriteWindowsMsg
next2: …
```

4. ReadHex

功能：从键盘读取一个 32 bit的十六进制整数，并用 EAX 返回相应的二进制数。

对无效字符不做错误检查。字母 A 到 F 的大小写都可以使用。最多能够输入 8 个数字（超出的字符将被忽略），前导空格将被忽略。

例如：

```
.data
hexVal DWORD ?
.code
call ReadHex
mov hexVal,eax
```

5. ReadInt

功能：从键盘读取一个 32 bit有符号整数，并用 EAX 返回该值。用户可以键入前置加号或减号，而其后跟的只能是数字。

- **ReadInt 设置溢出标志位，若输入数值超过 32 位有符号数的范围：$-2^{31} \sim +2^{31}-1$，则OF=1。**
- **返回值包括所有的有效数字，直到遇见第一个非数字字符。例如，如果用户输入 +123ABC，则返回值为 +123。**

6. ReadKey

功能：执行无等待键盘检查，它检查键盘输入缓冲区以查看用户是否有按键操作。

- 若没有发现键盘数据，则零标志位ZF= 1。
- 若有按键，则ZF=0，且向 AL 送入 0 或 ASCII 码。
  - 若 AL 为 0，表示用户可能按下了一个特殊键(功能键、方向键等)AH 为虚拟扫描码，DX 为虚拟键码，EBX 为键盘标志位。

7. ReadString

功能：从键盘读取一个字符串，直到用户键入回车键。

- 用 EDX 传递缓冲区的偏移量，用 ECX 传递用户能键入的最大字符数加 1（最后为空字符null），用 EAX 返回用户键入的字符数。
- ReadString 在缓冲区的字符串的末尾自动插入一个 null 终止符。

例如：

```
.data
buffer BYTE 21 DUP(0)        ;输入缓冲区
byteCount DWORD ?            ;定义计数器
.code
mov edx,OFFSET buffer        ;指向缓冲区
mov ecx,  SIZEOF buffer      ;定义最大字符数
call ReadString              ;输入字符串
mov byteCount, eax           ;取字符数
```

若用户输入“ABCDEFG”，buffer 中前 8 个字节的十六进制 ASCII 码：

```
41 42 43 44 45 46 47 00
```

8. WaitMsg

功能：显示“Press any key to continue…”消息，并等待用户按键。

- 当用户想在数据滚动和消失之前暂停屏幕显示时，这个过程就很有用。过程没有输入参数。

9. WriteBin

功能：以二进制格式向控制台窗口输出一个整数。

- 过程用 EAX 传递该整数。为了便于阅读，二进制位以四位一组的形式进行显示。

例如：
```
mov eax,12346AF9h
call WriteBin
```

显示如下：

0001 0010 0011 0100 0110 1010 1111 1001

10. WriteBinB

功能：以二进制格式向控制台窗口输出一个 32 位整数。

- 过程用 EAX 传递该整数，用 EBX 表示以字节为单位的显示位数（1、2，或 4）。
- 为了便于阅读，二进制位以四位一组的形式进行显示。

例如：
```
mov eax,00001234h
mov ebx, 1          ; 1个字节
call WriteBinB      ; 只显示最后一个字节  0011 0100
```

11. WriteChar

功能：向控制台窗口写一个字符。过程用 AL 传递字符（或其 ASCII 码）。

例如：
```
mov al, 'A'
call WriteChar          ;显示字符"A"
```

12. WriteDec

功能：以十进制格式向控制台窗口输出一个 32 位无符号整数，无前置 0显示。过程用 EAX 寄存器传递该整数。

```
例如： mov eax,295
       call WriteDec          ;显示："295"
```

13. WriteHex

功能：以 8 位十六进制格式向控制台窗口输出一个 32 位无符号整数，若不够8位自动插入前置 0。过程用 EAX 传递整数。

```
例如： mov eax,12345
       call WriteHex          ;显示："00003039"
```

14. WriteHexB

功能：以十六进制格式向控制台窗口输出一个 32 位无符号整数，不够位数插入前置 0。过程用 EAX 传递整数，用 EBX 表示显示格式的字节数（1、2，或 4）。

```
例如：mov eax, 7FFFh
      mov ebx, TYPE WORD       ;两个字节
      call WriteHexB           ;显示："7FFF"
```

15. WriteInt

功能：以十进制向控制台窗口输出一个 32 位有符号整数，有前置符号，但没有前置 0。过程用 EAX 传递整数。

例如：
```
mov eax, 216543
call WriteInt          ;显示："+216543"
```

16. WriteString

功能：向操作台窗口输出一个空字节结束的字符串。过程用 EDX 传递字符串的偏移量。

例如：
```
.data
prompt BYTE "Enter your name: ",0
.code
mov edx,OFFSET prompt
call WriteString
```

17. WriteToFile

功能：向一个输出文件写入缓冲区内容。过程用 EAX 传递有效的文件句柄，用 EDX 传递缓冲区偏移量，用 ECX 传递写入的字节数。

- 当过程返回时，如果 EAX 大于 0，则其包含的是写入的字节数；否则，发生错误。

例如：

```
BUFFER_SIZE = 5000
.data
fileHandle DWORD ?
buffer BYTE BUFFER_SIZE DUP(?)
.code
mov eax, fileHandle
mov edx, OFFSET buffer
mov ecx, BUFFER_SIZE
call WriteToFile
```

该过程的处理用伪代码表示：

```
if EAX = 0 then
    error occurred when writing to file
    call WriteWindowsMessage to see the
error
else
    EAX = number of bytes written to the file
endif
```

18. WriteWindowsMsg

功能：向控制台窗口输出应用程序在调用系统函数时最近产生的错误信息。

19. OpenInputFile

功能：打开一个已存在的文件进行输入。过程用 EDX 传递文件名的偏移量。当从过程返回时，如果文件成功打开，则 EAX 就包含有效的文件句柄。否则，EAX 等于 INVALID_HANDLE_VALUE（一个预定义的常数）。

例如：
```
.data
filename BYTE "myfile.txt",0
.code
mov edx,OFFSET filename
call OpenlnputFile
```

20. CloseFile

功能：关闭之前已经创建或打开的文件（参见 CreateOutputFile 和 OpenlnputFile）。该文件用一个 32 位整数的句柄来标识，句柄由 EAX 传递。如果文件成功关闭，EAX 中的返回值就是非零的。

例如：
```
mov eax,fileHandle
call CloseFile
```

作业：4.4，4.7，4.8, 4.10, 4.11，4.13， 4.14