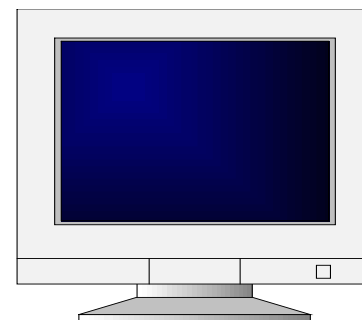
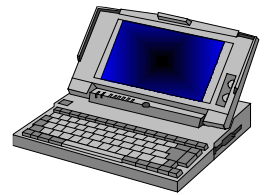


第4章

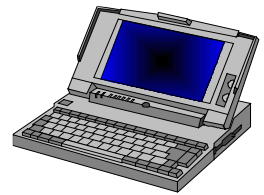
汇编语言程序设计





主要内容

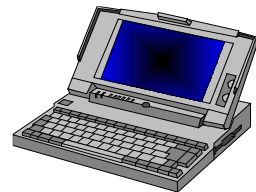
- 汇编语言源程序的结构
- 汇编语言语句格式
- 伪指令
- 功能调用
- 汇编语言程序设计



§ 4.1 汇编语言源程序

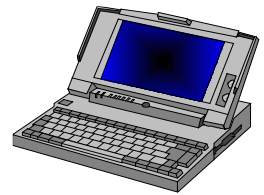
了解:

- 汇编语言源程序的结构
- 汇编语言语句类型及格式



汇编语言与汇编程序

前面介绍了指令、指令系统及程序的基本概念，由此了解到计算机所以能够自动地工作，是因为运行程序的结果。计算机能够按照程序中的安排去执行相应的指令，才使得计算机看起来工作得非常有序。通过第三章的内容还了解到计算机可直接识别的是机器指令，而用机器指令编写的程序称为**机器语言程序**。由于机器指令是用二进制编码来表示的，既不直观又难以记忆，所以使得机器指令编写的程序在使用上受到了限制。



为了解决机器语言使用上的不便，人们开始使用容易记忆和识别的符号指令编写程序。**汇编语言就是用与操作功能含义相应的缩写英文字符组成的符号指令作为编程用的语言。**因此说汇编语言实际上是一种符号语言，并且是一种面向机器的低级语言。**在使用汇编语言编写程序时需要对计算机硬件有一定的了解。**

□ 下面分别使用机器语言和汇编语言编写的一段小程序， 以此观察它们的不同。 □

机器语言程序

0000 B0 09

0002 04 08

0004 F4

汇编语言程序□

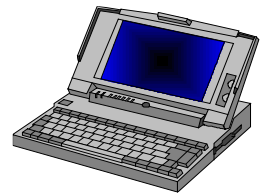
MOV AL, 9□

ADD AL, 8□

HLT□



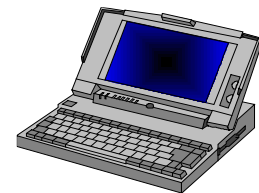
一、汇编语言源程序结构



1. 汇编语言源程序与汇编程序

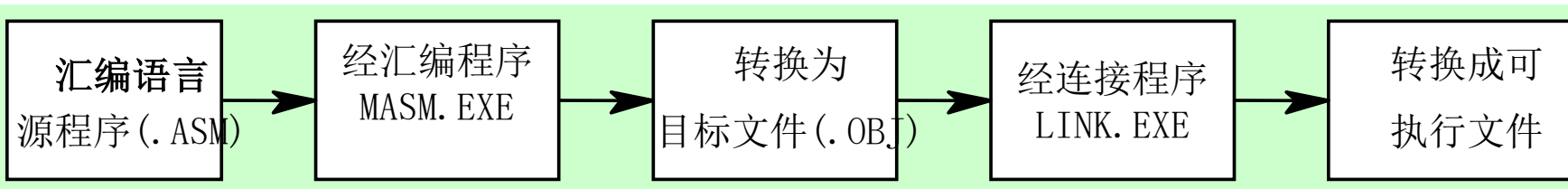
- 汇编语言源程序 → 用助记符编写
- 汇编程序 → 源程序的编译程序

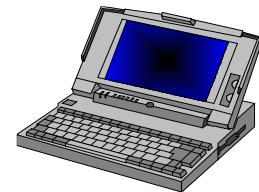




2. 汇编语言程序设计与执行过程

- 输入汇编语言源程序（记事本）→ 源文件 . **ASM**
- 汇编（MASM.EXE）→ 目标文件 **.OBJ**
- 链接（LINK.EXE）→ 可执行文件 **.EXE**
- 调试（DEBUG.EXE）→ 最终程序





汇编语言源程序结构

数据段名 SEGMENT

...

数据段名 ENDS

附加段名 SEGMENT

...

附加段名 ENDS

堆栈段名 SEGMENT

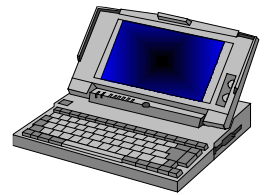
...

堆栈段名 ENDS

代码段名 SEGMENT

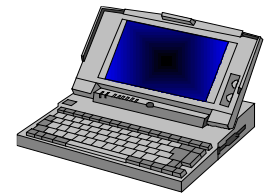
...

代码段名 ENDS
END



一个完整源程序结构例

```
DSEG  SEGMENT
        DATA1 DB  1, 2, 3  DUP ( ? )
        DATA2 DW 1234H
DSEG  ENDS
ESEG  SEGMENT
        DB 20  DUP ( ? )
ESEG  ENDS
SSEG  SEGMENT STACK 'STACK'
        DB 200  DUP ( ? )
SSEG  ENDS
```



一个完整源程序结构例

```
CSEG SEGMENT
```

```
    ASSUME CS: CSEG, DS: DSEG,  
           ES: ESEG, SS: SSEG
```

```
START: MOV AX, DSEG  
       MOV DS, AX  
       MOV AX, ESEG  
       MOV ES, AX  
       MOV AX, SSEG  
       MOV SS, AX
```

段寄存器初始化
——将段地址送
相应的段寄存器

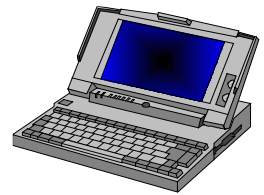
```
       !  
CSEG ENDS  
      END START
```

源程序
代码





二、汇编语言语句类型及格式



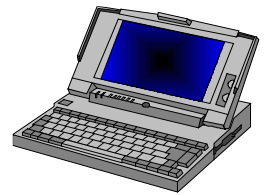
1. 汇编语言语句类型

指令性语句(指令)

CPU执行的语句，
能够生成目标代码

指示性语句(伪指令)

CPU不执行，而由汇
编程序执行的语句，
不生成目标代码



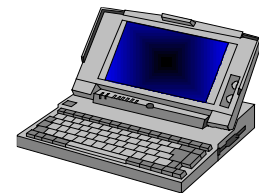
2. 汇编语言语句格式

指令性语句：

[标号：] [前缀] 指令助记符 [操作数], [操作数] [; 注释]

指令的符号地址
标号后要有冒号

操作码

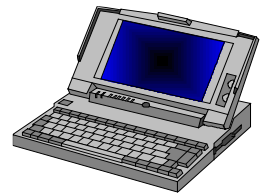


指示性语句格式

[名字] 伪指令助记符 操作数 [, 操作数, ...] [; 注释]

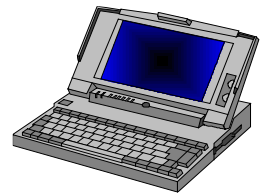
变量的符号地址
其后不加冒号

指示性语句中至少有一个操作数



3. 标号、名字

- 标号后有冒号，在指令性语句前；名字后不加冒号，在指示性语句前。
- 英文字母、数字及专用字符组成，最大长度不能超过31个，且不能由数字打头，**不能用保留字（如寄存器名，指令助记符，伪指令）。**



4. 操作数

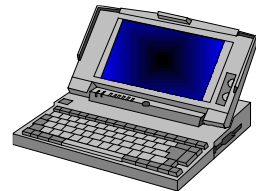
寄存器

存储器单元

常量

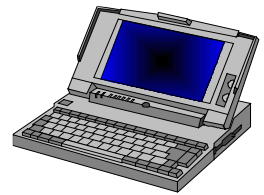
变量或标号

表达式



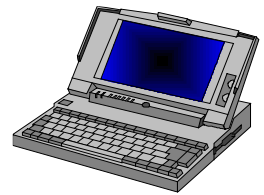
常量

- 数字常量
- 字符串常量 → 用单引号引起的字符或字符串
- 例： 'A'
 - MOV AL, 'A'
- 例： 'ABCD' → 汇编时被译成对应的ASCII码41H, 42H, 43H, 44H



变 量

- 代表内存中的数据区，程序中视为存储器操作数
- 变量的属性：
 - 段 值 —— 变量所在段的段地址
 - 偏移量 —— 变量单元地址与段首地址之间的位移量。
 - 类 型 —— 字节型、字型和双字型



表达式

算术运算(+、-、*、/、MOD)

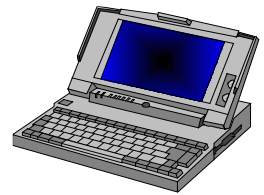
逻辑运算 (AND、OR、NOT、XOR)

关系运算(EQ、NE、LT、GT、LE、
GE)

(=、!=、<、>、<=、>=)

取值运算和属性运算

其它运算



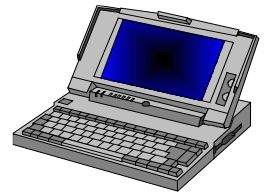
取值运算符

- 用于分析存储器操作数的属性

- 获取变量的属性值

- OFFSET** → 取得其后变量或标号的偏移地址
 - SEG** → 取得其后变量或标号的段地址

- TYPE** 取变量的类型
 - LENGTH** 取所定义存储区的长度
 - SIZE** 取所定义存储区的字节数



取值运算符例

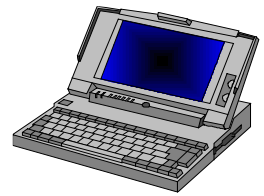
MOV AX, SEG DATA

MOV DS, AX

MOV BX, OFFSET DATA

等价于

LEA BX, DATA



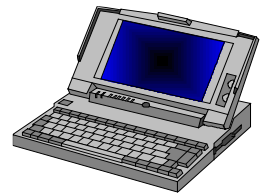
取值运算符例

- 若BUFFER存储区用如下伪指令定义：

BUFFER DW 200 DUP(0)

则：

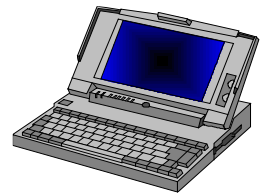
TYPE	BUFFER	等于2
LENGTH	BUFFER	等于200
SIZE	BUFFER	等于400



属性运算符

- 用于指定其后存储器操作数的类型
- 运算符: **PTR**
- 例:

MOV [BX], 12H ;汇编时会出错
应修改为 **MOV BYTE PTR [BX], 12H**



其它运算符

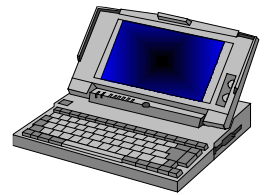
■ 方括号:

[] → 方括号中内容为操作数的偏移地址

■ 段重设符

段寄存器名: [] → 用于修改默认的段基地址

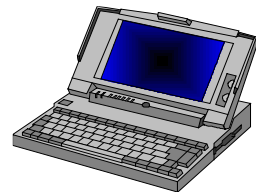
MOV AX, **ES**: [BX]



§ 4.2 伪指令

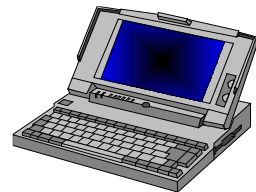
掌握：

- 伪指令的格式及实现的操作
- 伪指令的应用



伪指令

- 由汇编程序执行的“指令系统”
- 作用：
 - 定义变量；
 - 分配存储区
 - 定义逻辑段；
 - 指示程序开始和结束；
 - 定义过程等。



常用伪指令

数据定义伪指令

符号定义伪指令

段定义伪指令

结束伪指令

过程定义伪指令

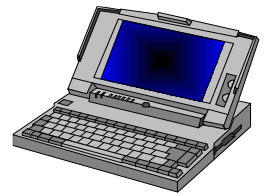
宏命令伪指令



- [变量名] 伪指令助记符 操作数, ... ; [注释]**

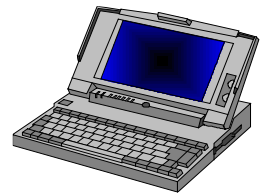
定义变量类型

定义变量值 及区域大小



1. 数据定义伪指令助记符

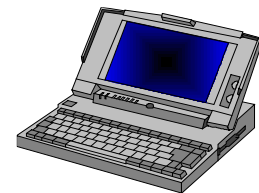
- DB(Define Byte) 定义变量为字节型
- DW(Define Word) 定义变量为字类型（双字节）
- DD(Define DoubleWord) 定义变量为双字型（4字节）
- DQ(Define QuadWord) 定义变量为4字型（8字节）
- DT(Define TenByte) 定义变量为10字节型



数据定义伪指令例

- DATA1 DB 11H, 22H, 33H, 44H
- DATA2 DW 11H, 22H, 3344H
- DATA3 DD 11H*2, 22H, 33445566H

以上变量在内存
中的存放形式



数据定义伪指令例_变量在内存中的分布

DATA1 DB 11H, 22H, 33H, 44H

DATA1

11

22

33

44

DATA2

11

00

22

00

44

33

DATA3 DD 11H*2, 22H,
33445566H

DATA3

22

0

0

0

22

0

0

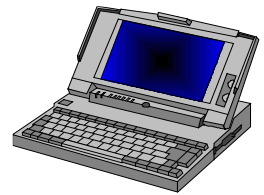
66

55

44

33

DATA2 DW 11H, 22H, 3344H



数据定义伪指令的几点说明

- 伪指令的性质决定所定义变量的类型；
- 定义字符串(超过2个字符)必须用DB伪指令
- 例：

■ DATA1 DB 'ABCD', 66H

DATA1

41H

'A'

42H

'B'

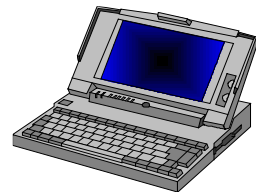
43H

'C'

44H

'D'

66H

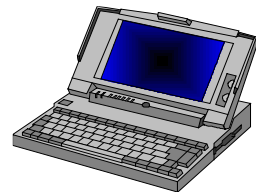


2. 重复操作符

- 作用：
 - 为一个数据区的各单元设置相同的初值
- 目的：
 - 常用于声明一个数据区
- 格式：

[变量名] 伪指令助记符 n DUP (初值, ...)
- 例：

```
N1 DW 20 DUP (0)
M1 DB 10 DUP (0)
```



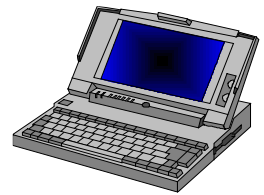
3. “?” 的作用

- 表示随机值，用于预留存储空间
- `MEM1 DB 34H, 'A', ?`

`DW 20 DUP (?)`

随机数
占1个字节单元

预留40个字节单元，每单元为随机值

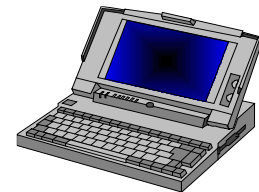


数据定义伪指令例

- M1 DB 'How are you?'
- M2 DW 3 DUP(11H), 3344H
- DB 4 DUP (?)
- M3 DB 3 DUP (22H, 11H, ?)

变量在内存中的分区





数据定义伪指令例

M1

'H'
'o'
'w'
' '
'a'
'r'
'e'
' '
'y'
'o'
'u'
'?'

M2

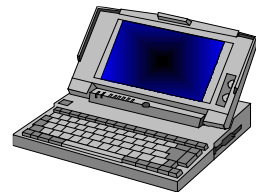
11H
00H
11H
00H
11H
00H
44H
33H
XX
XX
XX
XX

M3

22H
11H
XX
22H
11H
XX
22H
11H
XX

随机数





二、符号定义伪指令

- 格式:

符号名 EQU 表达式

- 操作:

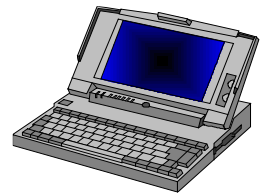
用符号名取代后边的表达式，不可重新定义

- 例:

CONSTANT EQU 100

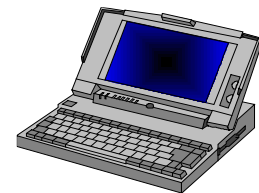
VAR EQU 30H+99H

EQU说明的表达式不占用内存空间



三、段定义伪指令

- 说明逻辑段的起始和结束；
- 说明不同程序模块中同类逻辑段之间的联系形态



段定义伪指令格式

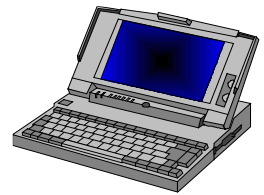
段名 SEGMENT [定位类型] [组合类型] ['类别']

⋮

段名 ENDS

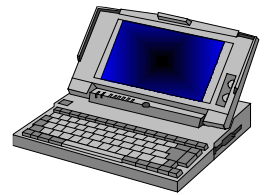
说明逻辑
段的起点

说明不同模块中同名
段的组和连接方式



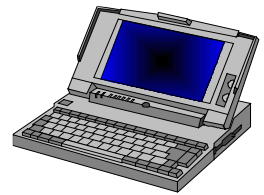
定位类型

- PARA: 段的起点从节边界开始(XXXX0H)
(16个字节为1节,默认)
- BYTE: 段的起点从存储器任何地址开始
- WORD: 段的起点从偶地址开始(地址为偶数)
- PAGE: 段的起点从页边界开始(XXX00H)
(256个字节为1页)



组合类型

- 与其它模块中的同名段在满足定位类型的前提下具有的组合方式：
 - NONE: 不组合（默认）
 - PUBLIC: 依次连接（顺序由LINK程序确定）
 - COMMON: 覆盖连接
 - STACK: 堆栈段的依次连接
 - AT 表达式: 段定义在表达式值为段基的节边界
 - MEMORY: 相应段在同名段的最高地址处。



段定义伪指令例

DATA SEGMENT

MEM1 DB 11H, 22H

MEM2 DB 'Hello! '

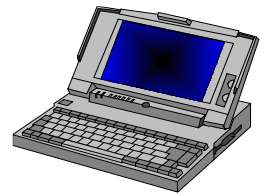
MEM3 DW 2 DUP (?)

DATA ENDS

变量在逻辑段中的位置就代表了它的偏移地址

表示变量所在逻辑段的段地址

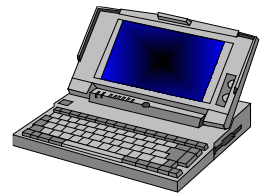
表示变量的类型



四、设定段寄存器伪指令

- 说明所定义逻辑段的性质
- 格式:

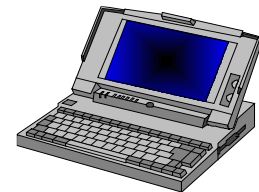
ASSUME 段寄存器名:段名[, 段寄存器名:段名, ...]



五、结束伪指令

- 表示源程序结束
- 格式:

END [标号]



六、过程定义伪指令

- 用于定义一个过程体

- 格式：

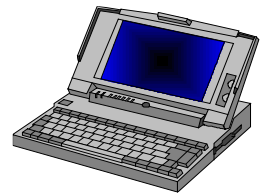
过程入口的
符号地址

过程名 PROC [NEAR / FAR]

⋮

RET

过程名 ENDP



过程定义及调用例

■ 定义延时10ms子程序

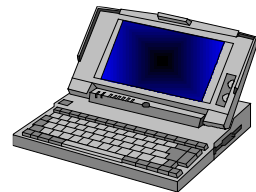
■ **DELAY PROC**

- `PUSH BX ;15T`
- `PUSH CX ;15T`
- `MOV BL, 2 ;4T`
- `NEXT: MOV CX, 4167 ;4T`
- `W10MS: LOOP W10MS ;17T/5T`
- `DEC BL ;3T`
- `JNZ NEXT ;16T/4T`
- `POP CX ;12T`
- `POP BX ;12T`
- `RET ;20T`

DELAY ENDP

■ 调用延时子程序:

■ **CALL DELAY**

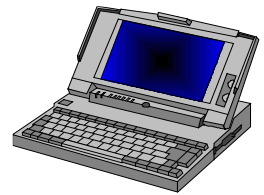


七、宏命令伪指令

- 宏 → 源程序中由汇编程序识别的具有独立功能的一段程序代码
- 格式：

宏命令名 MACRO <形式参数>

 ⋮
 └─────────── 宏体
 ⋮
 ENDM



八、其它伪指令

ORG → 段内程序代码或变量的起始偏移地址

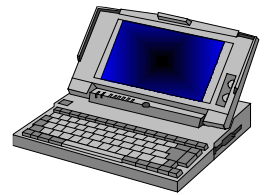
- 格式:

ORG 表达式

- 例:

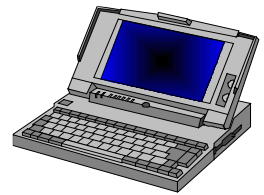
ORG 2000H

计算值为
非负常数



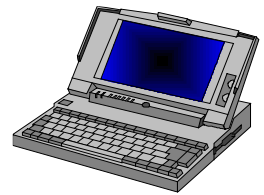
§ 4.3 BIOS和DOS功能调用

- BIOS(Basic Input and Output System)
 - 驻留在ROM中的基本输入/输出系统
 - BIOS功能调用使程序员不必了解硬件操作的细节而实现相应的操作。
- DOS(Disk Operation Sytem)
 - 磁盘操作系统
 - 相比BIOS，对硬件的依赖性小
- DOS功能与BIOS功能均通过中断方式调用。
格式： INT n (调用前准备好参数)



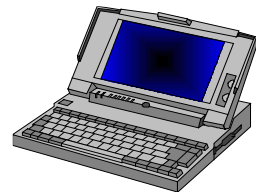
DOS中断与BIOS中断

- DOS中断包括：
 - 设备管理，目录管理，文件管理，其它
- 在某些情况下，同样的功能既可选择DOS中断，也可选择BIOS中断



DOS调用 and BIOS调用的基本步骤

- 将调用参数装入指定的寄存器；
- 将功能号装入AH；
- 按中断类型号调用DOS或BIOS中断；
- 检查返回参数是否正确。



一、DOS 功能调用

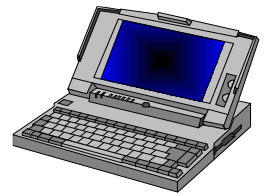
■ 说明：

- DOS中断是包含多个子功能的功能包；
- 各子功能用功能号区分；
- 用软中断指令调用，中断类型码固定为21H。

■ 调用格式：

- MOV AH, 功能号
 <置相应参数>

INT 21H



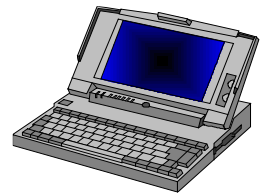
1. 单字符输入

- 调用方法:

MOV AH, 01

INT 21H

- 输入的字符在AL中

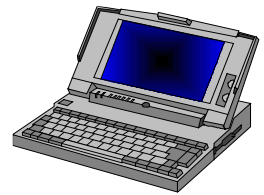


单字符输入例

```
GET_KEY:  MOV    AH, 1
          INT     21H
          CMP     AL, 'Y'
          JZ      YES
          CMP     AL, 'N'
          JZ      NO
          JMP     GET_KEY

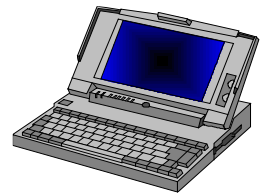
YES:      :
NO:       :
```

交互式应
答程序



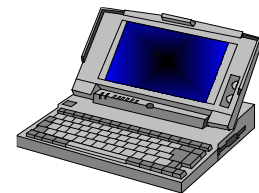
2. 字符串输入

- 注意问题：
 - 调用格式
 - 字符输入缓冲区的定义



调用格式

- **AH** ← 功能号0AH
- **DS: DX** ← 字符串在内存中的存放地址
- **INT 21H**



定义字符缓冲区

- 用户自定义缓冲区格式:

存放字符个数: ≤ 255

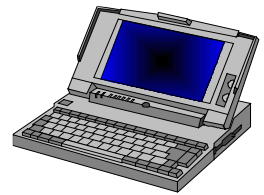


存放键入的字符

整个缓冲区

实际键入字符数

最大可键入字符数



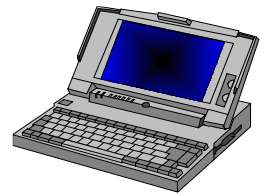
输入字符串程序段

- DAT1 DB 20, ? , 20 DUP (?)

⋮

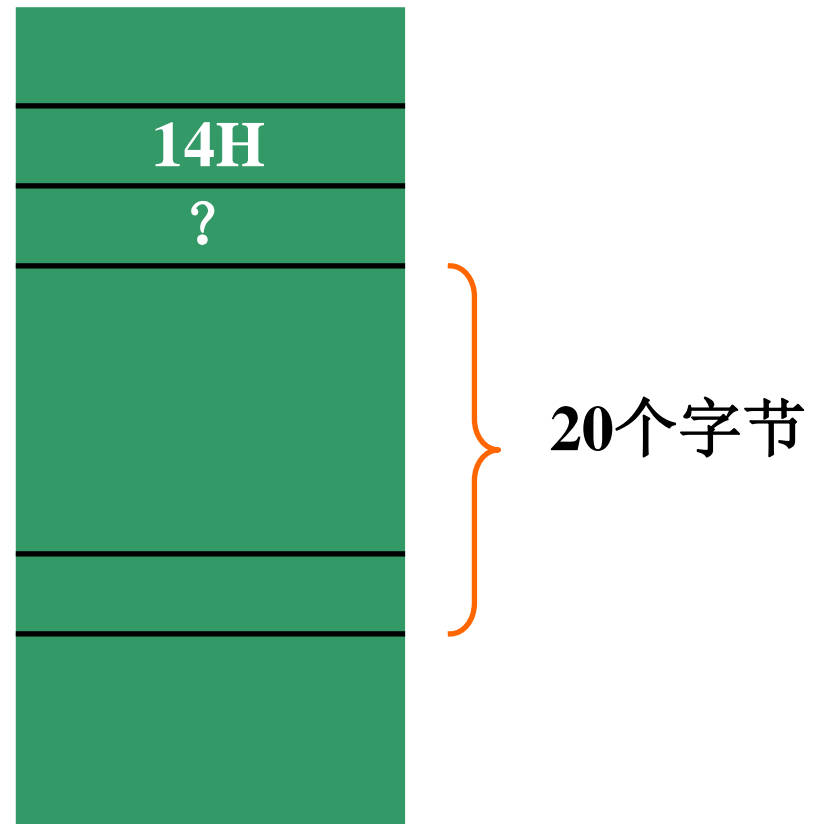
- LEA DX, DAT1
MOV AH, 0AH
INT 21H

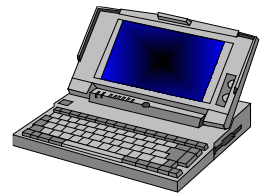
在数据段
中定义



输入缓冲区

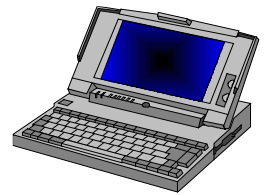
定义后的输入缓冲区初始状态:





3. 单字符显示输出

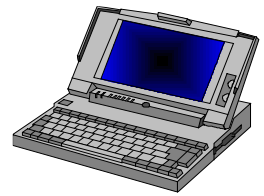
- AH ← 功能号2
- DL ← 待输出字符
- INT 21H



单字符显示输出例

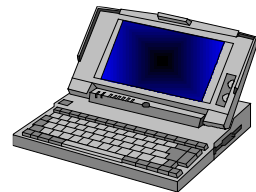
- **MOV AH, 2**
- **MOV DL, 41H**
- **INT 21H**

执行结果：
屏幕显示A



4. 字符串输出显示

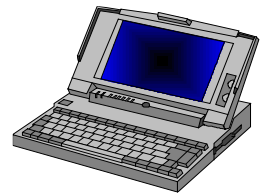
- AH ← 功能号09H
- DS: DX ← 待输出字符串的偏移地址
- INT 21H



字符串输出显示

■ 注意点：

- 被显示的字符串必须以 ‘\$’ 结束；
- 所显示的内容不应出现非可见的ASCII码；
- 若考虑输出格式需要，在定义字符串后，加上回车符和换行符。



字符串输出显示例

DATA SEGMENT

MESS1 DB 'Input String:', 0DH,0AH,'\$'

DATA ENDS

CODE SEGMENT

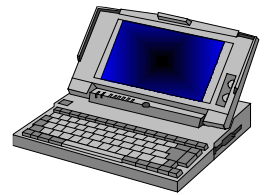
⋮

MOV AH, 09

MOV DX, OFFSET MESS1

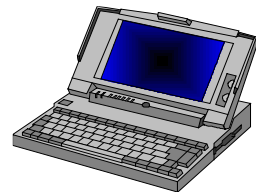
INT 21H

⋮



5. 返回操作系统 (DOS) 功能

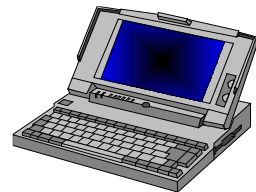
- 功能号：
 - 4CH
- 调用格式：
 - MOV AH, 4CH
 - INT 21H
- 功能：
 - 程序执行完该2条语句后能正常返回OS
 - 常位于程序结尾处。



二、BIOS功能调用

- 通过中断指令调用相应的BIOS中断服务程序
- BIOS中断服务程序实际上是一些对端口的输入输出操作，是微机系统中软件与硬件之间的一个可编程接口。
 - 光驱、硬盘管理；中断设置等

附录D



键盘状态检验

- 可利用类型码为 16H 的 BIOS 中断判断是否有任意键按下
- 调用格式：
 - AH ← 功能号 1
 - INT 16H
- 判断方法：
 - 若 ZF=0 → 有键按下
 - 若 ZF=1 → 无键按下

例： 在屏幕上显示信息，当有任意键按下时退出

DSEG SEGMENT

MESS DB 'Hello, World!', 0DH, 0AH, '\$'

DSEG ENDS

CSEG SEGMENT

ASSUME CS: CSEG, DS: DSEG

START: MOV AX, DSEG

MOV DS, AX

AGAIN: LEA DX, MESS

MOV AH, 9

INT 21H

MOV AH, 1

INT 16H

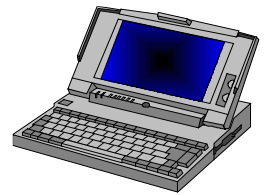
JZ AGAIN

MOV AH, 4CH

INT 21H

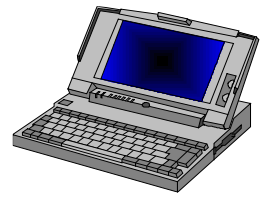
CSEG ENDS

END SATRT



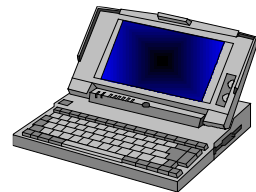
键盘状态检验

- 判断是否有任意键按下的方法可以用DOS软中断，功能号为0BH，出口参数为AL。
- 格式：
 - MOV AH, 0BH
 - INT 21H
- 若AL=FFH，则有键按下；
- 若AL=0，则无键按下



DOS和BIOS功能调用小结

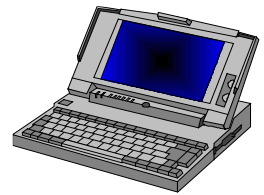
- 均通过中断指令调用。1个中断类型码对应1个功能程序包；
- 每个程序包中的子功能通过功能号区分，调用时功能号须送AH；
- 部分功能既可用DOS中断也可以用BIOS中断；
- 注意不同子功能的入口/出口参数要求；
- **DOS和BIOS中断均可能影响AX。**



§ 4.4 汇编语言程序设计

设计步骤:

- 根据实际问题抽象出数学模型
- 确定算法
- 画程序流程图
- 分配内存工作单元和寄存器
- 程序编码
- 调试



4.4.1 程序设计概述

基本源程序结构 (1) : 过程定义法

CODE SEGMENT

ASSUME CS: CODE, DS: DATA

START PROC FAR ; START为过程名

PUSH DS

MOV AX, 0

PUSH AX ;标准序

MOV AX, DATA

**MOV DS, AX ; 上述为固定写法, 如果用到ES, SS则
同样需赋值**

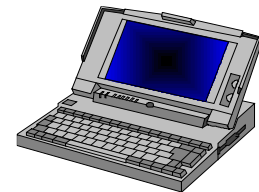
程序正文(指令集合)

RET ; 过程返回

START ENDP ; 结束过程定义

CODE ENDS ; 结束代码段

END START ; 结束汇编



基本源程序结构 (2) : 主程序定义法

CODE SEGMENT

ASSUME CS: CODE, DS: DATA

MAIN: MOV AX, DATA

MOV DS, AX ; 如果用到ES, SS则同样需赋值

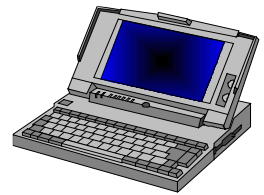
程序正文(指令集合)

MOV AH, 4CH

INT 21H ; 21H号中断, 返回DOS退出

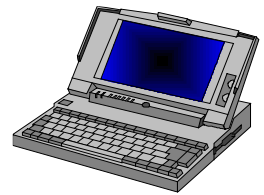
CODE ENDS ; 结束代码段

END MAIN ; 结束汇编



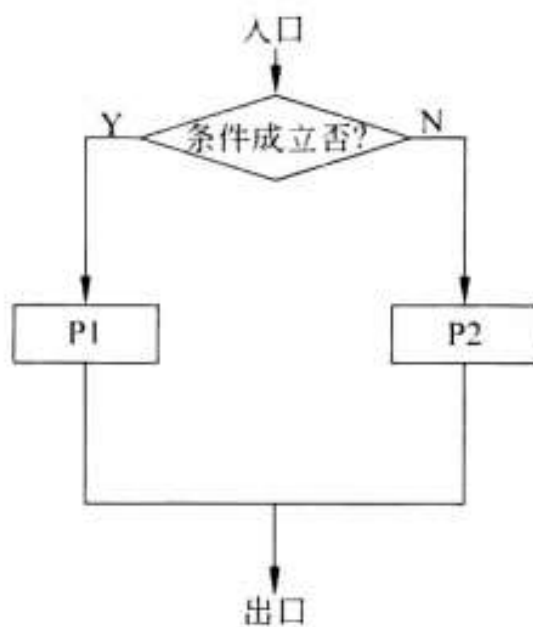
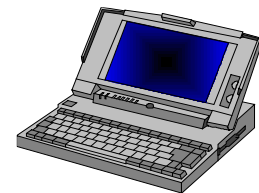
汇编语言程序控制结构的设计

- 顺序结构: 指令指针IP值线性增加
- 分支结构: IP值受标志位的影响而跳变
 - 一般用条件转移指令实现
- 循环结构: IP值受计数器CX中的值不为零而循环
 - 使用LOOP指令或条件转移指令实现
 - 注意点:
 - LOOP指令的循环次数须由CX给出
 - 条件转移指令的前一条指令的执行须影响标志位
- 子程序结构
 - 子程序的定义和调用方式

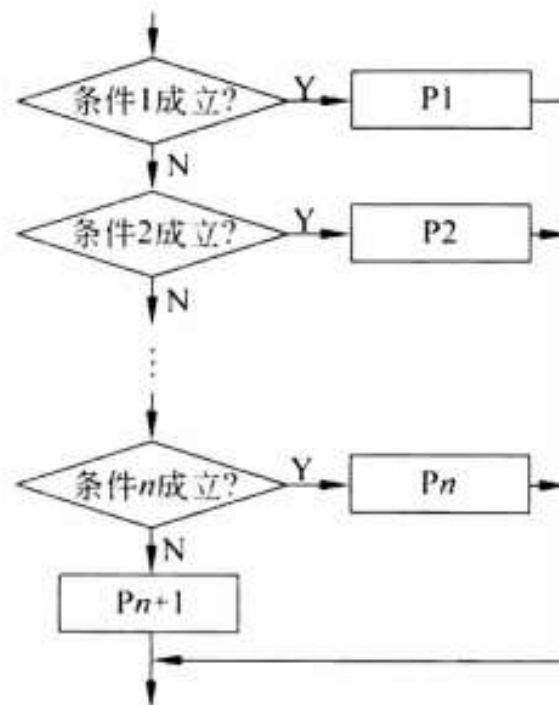


4.4.3 分支程序设计

在解决某些实际问题时，解决问题的方法随着某些条件的不同而不同，将这种在不同条件下处理过程的操作编写出的程序称为分支程序。程序中所产生的分支是由**条件转移指令**来完成的。汇编语言提供了多种条件转移指令，可以根据使用不同的转移指令所产生的结果状态选择要转移的程序段，对问题进行处理。采用分支结构设计的程序，结构清晰、易于阅读及调试。

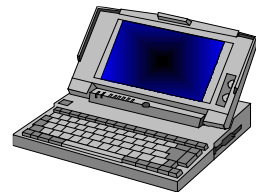


(a) 单分支(if-then)结构图



(b) 多分支结构图

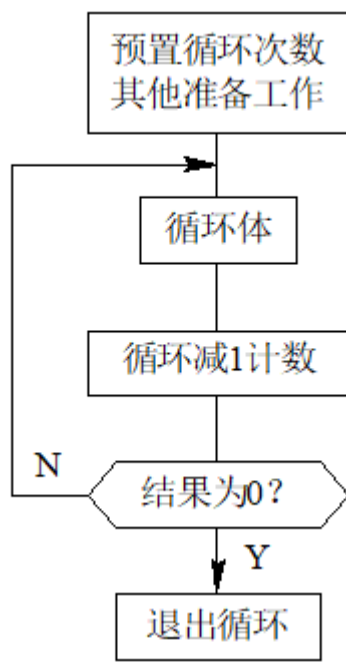
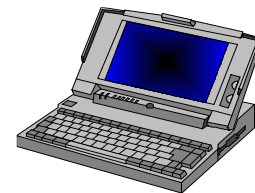
分支程序基本结构



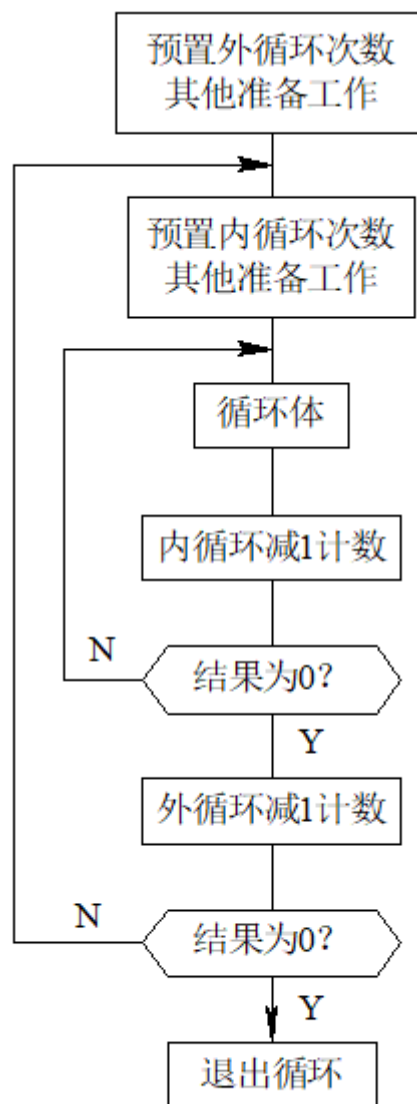
4.4.4 循环程序设计

循环结构程序设计针对的是处理一些重复进行的过程的操作。采用循环结构设计的程序，其长度缩短了，不仅节省了内存，也使得程序的可读性大大提高。使用循环结构形式设计程序时，通常将循环程序划分三个部分：

- ① 循环的初始化部分：主要为循环所需的变量赋初值。
- ② 循环体：程序所要完成的主要工作部分，这一部分的内容是由所要处理的问题来确定的，这一部分的执行结果可能影响到循环是否继续进行。
- ③ 循环控制部分：这一部分主要是以条件表达式的结果作为是否结束循环的条件。在事先知道了循环次数的情况下，可采用循环计数控制循环的结束；在事先不知道循环次数的情况下，多采用结果及给定特征作为条件来控制循环的结束。下图可以帮助我们很好地理解循环结构程序。



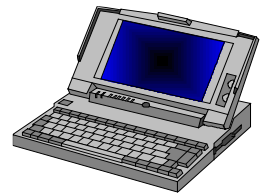
(a)



(b)

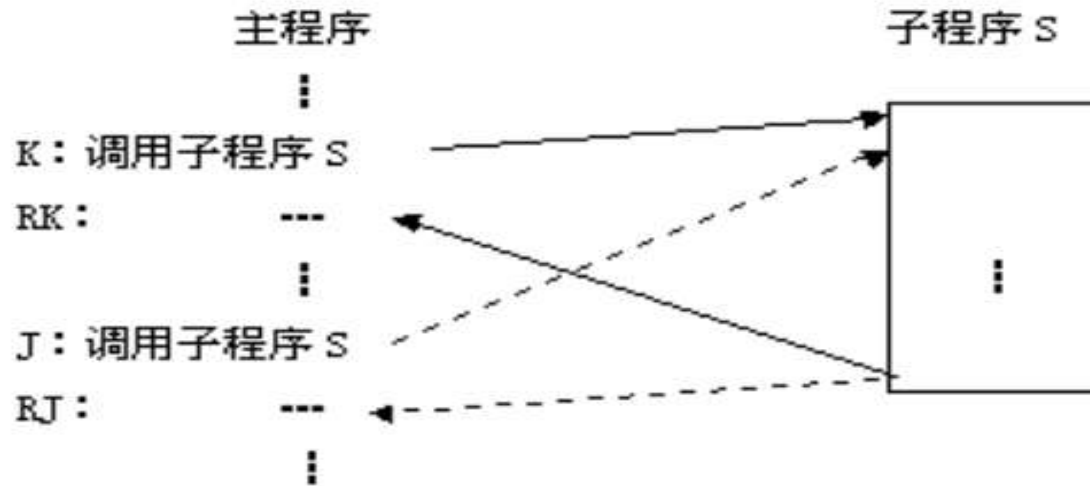
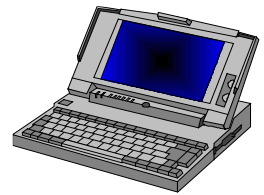
循环程序结构

(a) 单循环结构; (b) 双循环结构

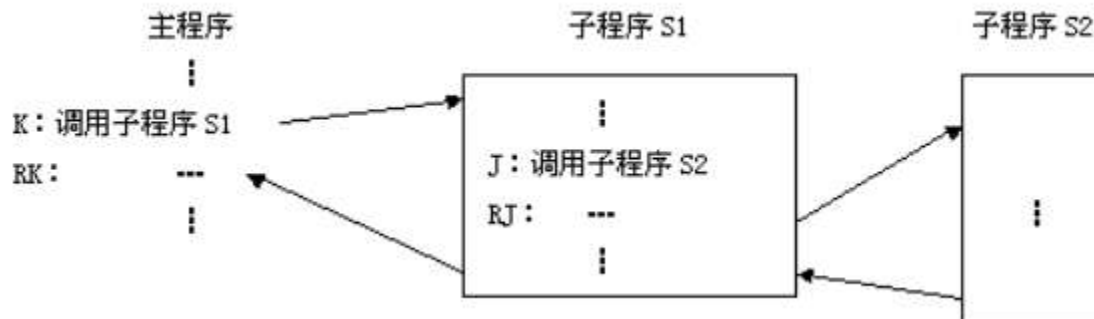


4.4.5 子程序设计

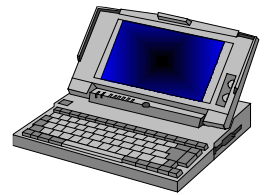
子程序设计是使程序模块化的一种重要手段。当设计一个比较复杂的程序时，将程序划分为若干个相对独立的模块，确定各模块的入口及出口参数，为各模块分配不同的名字，对每一个模块编制独立的程序段（即子程序），最后将这些子程序根据调用关系连成一个整体。这样既便于分工合作，又可避免重复劳动，节省存储空间，提高程序设计的效率和质量，使程序简洁、清晰、易读，便于修改和扩充。



子程序单重调用



子程序多重调用



一、定义子程序

子程序的定义格式是：

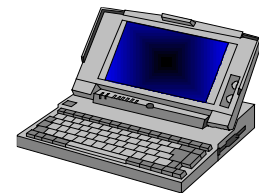
子程序名 PROC [NEAR/FAR]

⋮

RET

子程序名 ENDP

子程序也称为过程，PROC、ENDP是定义子程序时必须使用的保留字，PROC和ENDP相当于一对括号，将子程序的指令包括在内。如果主程序和子程序位于同一代码段，则称为段内调用，此时在PROC后可加NEAR说明此子程序是近过程。如果主程序和子程序不在同一代码段，则称为段间调用，此时在PROC后可加FAR说明此子程序是远过程。如果NEAR和FAR都不写，系统默认该子程序是近过程。



二、子程序的调用和返回

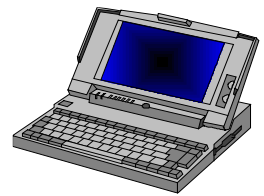
为了实现子程序的调用和返回，可使用子程序调用指令CALL和返回指令RET

1. 子程序调用指令CALL

子程序调用指令CALL的格式是：CALL OPRD。根据OPRD寻址方式的不同，又分为直接调用和间接调用。因此，子程序调用指令CALL共有4种组合，见下表。

子程序调用指令 CALL 的格式及功能表

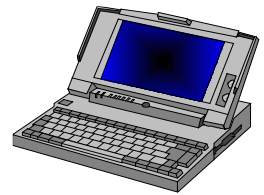
名称	格式	功能
段内直接调用	CALL 过程名	SP-2→SP, (IP)→(SP) 目的地址 EA→IP
段内间接调用	CALL WORD PTR OPD	SP-2→SP, (IP)→(SP) (OPD)→IP
段间直接调用	CALL FAR PTR 过程名	SP-2→SP, (CS)→(SP), 目的地址的段首址→CS SP-2→SP, (IP)→(SP), 目的地址 EA→IP
段间间接调用	CALL DWORD PTR OPD	SP-2→SP, (CS)→(SP), (OPD+2)→CS SP-2→SP, (IP)→(SP), (OPD)→IP



对于段间间接调用，需要双字单元存放子程序的入口地址信息，第一个字单元中放子程序入口的偏移地址，第二个字单元中放子程序所在段的段首址，见下表。

段间间接调用时子程序的入口地址信息表

OPD+0	子程序入口的偏移地址
+1	
+2	子程序所在段的段首址
+3	

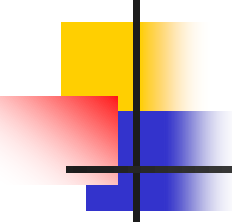
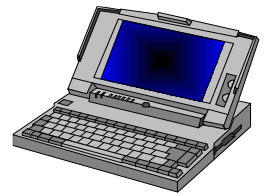


2. 返回指令RET

RET指令通常作为子程序的最后一条指令，用来控制CPU返回到主程序的断点处继续向下执行，RET指令的语句格式及功能见下表。

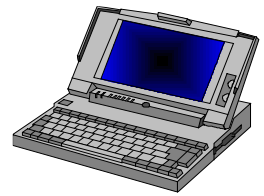
子程序返回指令 RET 的格式及功能表

名称	格式	功能
段内返回	RET	$(SP) \rightarrow IP, SP+2 \rightarrow SP$
段间返回	RET	$(SP) \rightarrow IP, SP+2 \rightarrow SP$ $(SP) \rightarrow CS, SP+2 \rightarrow SP$



无论是段内返回还是段间返回，当RET指令执行后，主程序的断点地址信息已送回到IP、CS中，堆栈恢复了转子前的状态。

RET指令的另一种格式是：RET N，其中N是偶数。该指令用来废除栈顶N个无用的参数。其操作是在正常RET操作之后再做 $SP+N \rightarrow SP$ 。这个功能允许废除一些在执行CALL指令之前入栈的参数。

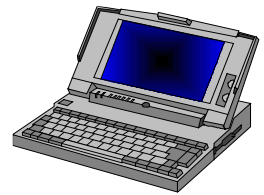


三、调用子程序前后怎样保存和恢复寄存器

如果在子程序中要用到某些寄存器（或存储单元），就会破坏这些寄存器（或存储单元）在调用之前原有的内容。当执行完子程序返回断点继续执行主程序时，只能以被破坏的现场为背景进行工作，这显然是不对的。因此，必须考虑**现场的保存和恢复**。一般情况下，在子程序的开始安排一些保存现场的指令，在子程序的返回指令之前再恢复现场。

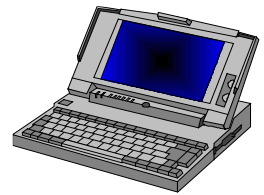
例如，若子程序SUBP中改变了寄存器AX、BX、CX的内容，则在子程序的开始处将这些寄存器的内容入栈保存，在子程序的返回指令之前用出栈指令依次恢复，具体实现方法如左图。

```
SUBP PROC  
    PUSH AX  
    PUSH BX  
    PUSH CX  
    !  
    POP CX  
    POP BX  
    POP AX  
SUBP ENDP
```



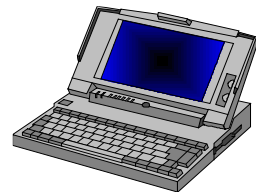
四、主程序和子程序间的参数传递

主程序在调用子程序之前，必须把需要子程序处理的原始数据传递给子程序，即为子程序准备入口参数。子程序对入口参数进行一系列处理之后得到处理结果，该结果必须送给调用它的主程序，即提供出口参数以便主程序使用。这种主程序为子程序准备入口参数、子程序为主程序提供处理结果的过程称为主程序和子程序间的**参数传递**。常用的参数传递方法有寄存器法、约定单元法和堆栈法三种，本节主要讲解前两种方法。



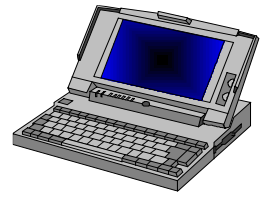
1. 寄存器法

寄存器法就是子程序的入口参数和出口参数都在**约定的**寄存器中。此法的优点是参数传递快,编程也较方便,且节省内存单元。但由于寄存器个数有限,而且在处理过程中要经常使用寄存器,如果要传递的参数很多,将导致无空闲寄存器供编写程序使用。所以寄存器法只适用于要传递参数较少的情况。



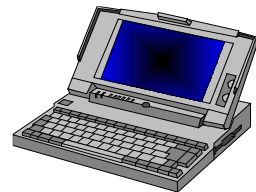
2. 约定单元法

约定单元法是把入口参数和出口参数都在约定的存储单元中。此法的优点是每个子程序要处理的数据或送出的结果都有独立的存储单元，编写程序时不易出错。缺点是要占用一定数量的存储单元。



3. 堆栈法

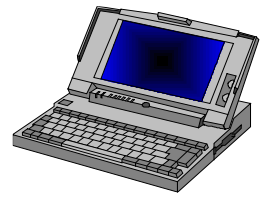
堆栈法是把入口参数和出口参数都放在堆栈的存储单元中。



本章注意点

- 完整的汇编语言源程序结构
 - 定义逻辑段，说明段的含义，初始化段寄存器
- 伪指令
 - 数据定义方式
- 字符及字符串的输入和显示输出
 - 字符输入缓冲区的定义，输出字符串的定义
- 源程序的编写
 - 几种结构（顺序、循环、分枝等）





作业

- P192
- 4.1, 4.2, 4.3, 4.7, 4.8, 4.15, 4.17